The Reference Manual for **SPOOLES**, Release 2.2: An Object Oriented Software Library for Solving Sparse Linear Systems of Equations

Cleve Ashcraft[1] Daniel Pierce[2] David K. Wah[3] Jason Wu[4]

January 26, 1999

[1]Boeing Shared Services Group, P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124, cleve.ashcraft@boeing.com. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

[2]Boeing Shared Services Group, P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124, dpierce@redwood.rt.cs.boeing.com. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

[3]Boeing Shared Services Group, P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124, david.wah@pss.boeing.com. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

[4]Boeing Shared Services Group, P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124, jwu@redwood.rt.cs.boeing.com. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

Abstract

Solving sparse linear systems of equations is a common and important application of a multitude of scientific and engineering applications. The **SPOOLES** software package[1] provides this functionality with a collection of software objects. The first step to solving a sparse linear system is to find a good low-fill ordering of the rows and columns. The library contains several ways to perform this operation: minimum degree, generalized nested dissection, and multisection. The second step is to factor the matrix as a product of triangular and diagonal matrices. The library supports pivoting for numerical stability (when required), approximation techniques to reduce the storage for and work to compute the matrix factors, and the computations are based on BLAS3 numerical kernels to take advantage of high performance computing architectures. The third step is to solve the linear system using the computed factors.

The library is written in ANSI C using object oriented design. Good design and efficient code sometimes conflict; generally we have preferred to cater to design. For large sparse matrices the serial code outperforms its FORTRAN predecessors, the reverse holds for moderate sized matrices or those that do not have good block structure. The present release of the library contains a serial factorization and solve, a multithreaded version using the Solaris and Posix thread packages, and an MPI version. There is considerable code overlap between the serial, threaded and MPI versions.

This release of the package is totally within the public domain; there are absolutely no licensing restrictions as with other software packages. The development of this software was funded by DARPA[2] and the DoD[3] with the express purpose that others (academic, government, industrial and commercial) could easily incorporate the data structures and algorithms into application codes. All we ask is an acknowledgement in derivative codes and any publications from research that uses this software. And, we hope that any improvements will be communicated to others.

[1] **SPOOLES** is an acronym for **SP**arse **O**bject-**O**riented **L**inear **E**quations **S**olver.

[2] DARPA Contract DABT63-95-C-0122.

[3] DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

Contents

I Introduction 15

1 Introduction 17

1.1 Software Design 18

1.2 Changes from Release 1.0 21

1.3 Changes from Release 2.0 21

II Utility Objects and Methods 23

2 `A2`: Real or complex 2-D array 25

2.1 Data Structure 25

2.2 Prototypes and descriptions of `A2` methods 25

2.2.1 Basic methods 26

2.2.2 Instance methods 26

2.2.3 Initialize methods 28

2.2.4 Methods used in the QR factorization 28

2.2.5 Norm methods 29

2.2.6 Sort methods 30

2.2.7 Utility methods 30

2.2.8 IO methods 33

2.3 Driver programs for the `A2 object` 34

3 `Coords`: Coordinates Object 36

3.1 Data Structure 36

3.2 Prototypes and descriptions of `Coords` methods 36

3.2.1 Basic methods 36

3.2.2 Initializer methods 37

3.2.3 Utility methods 38

3.2.4 IO methods 38

3.3 Driver programs for the `Coords` object 39

4 DV: Double Vector Object **41**
4.1 Data Structure 41
4.2 Prototypes and descriptions of DV methods 42
4.2.1 Basic methods 42
4.2.2 Instance methods 42
4.2.3 Initializer methods 43
4.2.4 Utility methods 44
4.2.5 IO methods 45
4.3 Driver programs for the DV object 47

5 Drand: Simple Random Number Generator **48**
5.1 Data Structure 48
5.2 Prototypes and descriptions of Drand methods 48
5.2.1 Basic methods 49
5.2.2 Initializer methods 49
5.2.3 Utility methods 50
5.3 Driver programs for the Drand object 50

6 I2Ohash: Two Key Hash Table **52**
6.1 Data Structure 52
6.2 Prototypes and descriptions of I2Ohash methods 53
6.2.1 Basic methods 53
6.2.2 Initializer methods 53
6.2.3 Utility methods 54
6.2.4 IO methods 54
6.3 Driver programs for the I2Ohash object 54

7 IIheap: (Key, Value) Heap **56**
7.1 Data Structure 56
7.2 Prototypes and descriptions of IIheap methods 56
7.2.1 Basic methods 56
7.2.2 Initializer methods 57
7.2.3 Utility methods 57

8 IV: Integer Vector Object **58**
8.1 Data Structure 58
8.2 Prototypes and descriptions of IV methods 59
8.2.1 Basic methods 59
8.2.2 Instance methods 59
8.2.3 Initializer methods 60
8.2.4 Utility methods 61
8.2.5 IO methods 63
8.3 Driver programs for the IV object 65

9 IVL: Integer Vector List Object **66**
9.1 Data Structure . . . 66
9.2 Prototypes and descriptions of IVL methods . . . 67
9.2.1 Basic methods . . . 67
9.2.2 Instance methods . . . 68
9.2.3 Initialization and resizing methods . . . 68
9.2.4 List manipulation methods . . . 69
9.2.5 Utility methods . . . 70
9.2.6 Miscellaneous methods . . . 71
9.2.7 IO methods . . . 72
9.3 Driver programs for the IVL object . . . 73

10 Ideq: Integer Dequeue **74**
10.1 Data Structure . . . 74
10.2 Prototypes and descriptions of Ideq methods . . . 74
10.2.1 Basic methods . . . 74
10.2.2 Initializer methods . . . 75
10.2.3 Utility methods . . . 75
10.2.4 IO methods . . . 76

11 Lock: Mutual Exclusion Lock object **77**
11.1 Data Structure . . . 77
11.2 Prototypes and descriptions of Lock methods . . . 77
11.2.1 Basic methods . . . 77
11.2.2 Initializer method . . . 78
11.2.3 Utility methods . . . 78

12 Perm: Permutation Object **79**
12.1 Data Structure . . . 79
12.2 Prototypes and descriptions of Perm methods . . . 79
12.2.1 Basic methods . . . 79
12.2.2 Initializer methods . . . 80
12.2.3 Utility methods . . . 80
12.2.4 IO methods . . . 81

13 Utilities directory **83**
13.1 Data Structures . . . 83
13.2 Prototypes and descriptions of Utilities methods . . . 83
13.2.1 CV : char vector methods . . . 84
13.2.2 DV : double vector methods . . . 84
13.2.3 ZV : double complex vector methods . . . 90
13.2.4 IV : int vector methods . . . 96
13.2.5 FV : float vector methods . . . 98

13.2.6 `PCV` : `char *` vector methods . . . 101
13.2.7 `PDV` : `double *` vector methods . . . 101
13.2.8 `PFV` : `float *` vector methods . . . 102
13.2.9 Sorting routines . . . 102
13.2.10 Sort and compress routines . . . 104
13.2.11 `IP` : `(int, pointer)` singly linked-list methods . . . 105
13.2.12 `I2OP` : `(int, int, void*, pointer)` singly linked-list methods . . . 106
13.3 Driver programs . . . 107

14 `ZV`: Double Complex Vector Object **108**
14.1 Data Structure . . . 108
14.2 Prototypes and descriptions of `ZV` methods . . . 109
14.2.1 Basic methods . . . 109
14.2.2 Instance methods . . . 109
14.2.3 Initializer methods . . . 110
14.2.4 Utility methods . . . 111
14.2.5 IO methods . . . 112
14.3 Driver programs for the `ZV object` . . . 113

III Ordering Objects and Methods **115**

15 `BKL`: Block Kernighan-Lin Object **117**
15.1 Data Structure . . . 117
15.2 Prototypes and descriptions of `BKL` methods . . . 118
15.3 Basic methods . . . 118
15.3.1 Initializer methods . . . 118
15.3.2 Utility methods . . . 119
15.3.3 Partition evaluation methods . . . 120
15.3.4 Partition improvement methods . . . 120

16 `BPG`: Bipartite Graph Object **122**
16.1 Data Structure . . . 124
16.2 Prototypes and descriptions of `BPG` methods . . . 124
16.2.1 Basic methods . . . 124
16.2.2 Initializer methods . . . 125
16.2.3 Generate induced graphs . . . 125
16.2.4 Utility methods . . . 125
16.2.5 Dulmage-Mendelsohn decomposition method . . . 126
16.2.6 IO methods . . . 126
16.3 Driver programs for the `BPG` object . . . 127

17 DSTree:
A Domain/Separator Tree Object **129**
17.1 Data Structure . . . 129
17.2 Prototypes and descriptions of DSTree methods . . . 129
17.2.1 Basic methods . . . 130
17.2.2 Instance methods . . . 130
17.2.3 Initializer methods . . . 130
17.2.4 Stage methods . . . 131
17.2.5 Utility methods . . . 132
17.2.6 IO methods . . . 132
17.3 Driver programs for the DSTree object . . . 133

18 EGraph: Element Graph Object **135**
18.1 Data Structure . . . 135
18.2 Prototypes and descriptions of EGraph methods . . . 135
18.2.1 Basic methods . . . 136
18.2.2 Initializer methods . . . 136
18.2.3 Utility methods . . . 136
18.2.4 IO methods . . . 137
18.3 Driver programs for the EGraph object . . . 138

19 ETree: Elimination and Front Trees **140**
19.1 Data Structure . . . 140
19.2 Prototypes and descriptions of ETree methods . . . 141
19.2.1 Basic methods . . . 141
19.2.2 Instance methods . . . 141
19.2.3 Initializer methods . . . 143
19.2.4 Utility methods . . . 144
19.2.5 Metrics methods . . . 146
19.2.6 Compression methods . . . 146
19.2.7 Justification methods . . . 147
19.2.8 Permutation methods . . . 148
19.2.9 Multisector methods . . . 148
19.2.10 Transformation methods . . . 150
19.2.11 Parallel factorization map methods . . . 151
19.2.12 Storage profile methods . . . 152
19.2.13 IO methods . . . 153
19.3 Driver programs for the ETree object . . . 154

20 GPart: Graph Partitioning Object **165**
20.1 Data Structures . . . 166
20.2 Prototypes and descriptions of GPart methods . . . 167

20.2.1 Basic methods 167
20.2.2 Initializer methods 168
20.2.3 Utility methods 168
20.2.4 Domain decomposition methods 169
20.2.5 Methods to generate a 2-set partition 170
20.2.6 Methods to improve a 2-set partition 170
20.2.7 Recursive Bisection method 172
20.2.8 `DDsepInfo` methods 172
20.3 Driver programs for the `GPart` object 173

21 `Graph`: A Graph object **177**
21.1 Data Structure 178
21.2 Prototypes and descriptions of `Graph` methods 178
21.2.1 Basic methods 178
21.2.2 Initializer methods 179
21.2.3 Compress and Expand methods 180
21.2.4 Wirebasket domain decomposition ordering 180
21.2.5 Utility methods 181
21.2.6 IO methods 182
21.3 Driver programs for the `Graph` object 183

22 `MSMD`:
`M`ulti-Stage `M`inimum `D`egree Object **187**
22.1 Data Structure 188
22.1.1 `MSMDinfo` : define your algorithm 189
22.1.2 `MSMD` : driver object 190
22.1.3 `MSMDstageInfo` : statistics object for a stage of the elimination 190
22.1.4 `MSMDvtx` : vertex object 191
22.2 Prototypes and descriptions of `MSMDinfo` methods 191
22.2.1 Basic methods 191
22.2.2 Utility methods 192
22.3 Prototypes and descriptions of `MSMD` methods 192
22.3.1 Basic methods — public 192
22.3.2 Initialization methods — public 193
22.3.3 Ordering methods — public 193
22.3.4 Extraction methods — public 193
22.3.5 Internal methods — private 194
22.4 Prototypes and descriptions of `MSMDvtx` methods 195
22.5 Driver programs for the `MSMD` object 195

23 `Network`: Simple Max-flow solver **198**
23.1 Data Structure . . . 199
23.2 Prototypes and descriptions of `Network` methods . . . 200
23.2.1 Basic methods . . . 200
23.2.2 Initializer methods . . . 201
23.2.3 Utility methods . . . 201
23.2.4 IO methods . . . 202

24 `SolveMap`: Forward and Backsolve Map **203**
24.1 Data Structure . . . 203
24.2 Prototypes and descriptions of `SolveMap` methods . . . 204
24.2.1 Basic methods . . . 204
24.2.2 Instance methods . . . 204
24.2.3 Initialization method . . . 205
24.2.4 Map creation methods . . . 206
24.2.5 Solve setup methods . . . 206
24.2.6 Utility methods . . . 206
24.2.7 IO methods . . . 207

25 `Tree`: A Tree Object **209**
25.1 Data Structure . . . 209
25.2 Prototypes and descriptions of `Tree` methods . . . 209
25.2.1 Basic methods . . . 210
25.2.2 Instance methods . . . 210
25.2.3 Initializer methods . . . 211
25.2.4 Utility methods . . . 211
25.2.5 Metrics methods . . . 213
25.2.6 Compression methods . . . 214
25.2.7 Justification methods . . . 214
25.2.8 Permutation methods . . . 214
25.2.9 Drawing method . . . 215
25.2.10 IO methods . . . 216
25.3 Driver programs for the `Tree` object . . . 217

IV Numeric Objects and Methods **219**

26 `Chv`: Block chevron **221**
26.1 Data Structure . . . 223
26.2 Prototypes and descriptions of `Chv` methods . . . 223
26.2.1 Basic methods . . . 224
26.2.2 Instance methods . . . 224
26.2.3 Initialization methods . . . 226

26.2.4 Search methods 227
26.2.5 Pivot methods 228
26.2.6 Update methods 228
26.2.7 Assembly methods 229
26.2.8 Factorization methods 229
26.2.9 Copy methods 230
26.2.10 Swap methods 232
26.2.11 Utility methods 233
26.2.12 IO methods 234
26.3 Driver programs for the `Chv object` 235

27 `ChvList`: `Chv` list object **240**
27.1 Data Structure 241
27.2 Prototypes and descriptions of `ChvList` methods 241
27.2.1 Basic methods 241
27.2.2 Initialization methods 242
27.2.3 Utility methods 242
27.2.4 IO methods 242

28 `ChvManager`: `Chv` manager object **243**
28.1 Data Structure 244
28.2 Prototypes and descriptions of `ChvManager` methods 244
28.2.1 Basic methods 244
28.2.2 Initialization methods 245
28.2.3 Utility methods 245
28.2.4 IO methods 245

29 `DenseMtx`: Dense matrix object **246**
29.1 Data Structure 246
29.2 Prototypes and descriptions of `DenseMtx` methods 247
29.2.1 Basic methods 247
29.2.2 Instance methods 247
29.2.3 Initialization methods 249
29.2.4 Utility methods 250
29.2.5 IO methods 252

30 `FrontMtx`: Front matrix **254**
30.1 Data Structures 257
30.2 Prototypes and descriptions of `FrontMtx` methods 259
30.2.1 Basic methods 259
30.2.2 Instance methods 259
30.2.3 Initialization methods 261
30.2.4 Utility Factorization methods 262

30.2.5 Serial Factorization method . . . 264
30.2.6 QR factorization utility methods . . . 265
30.2.7 Serial QR Factorization method . . . 266
30.2.8 Postprocessing methods . . . 266
30.2.9 Utility Solve methods . . . 267
30.2.10 Serial Solve method . . . 268
30.2.11 Serial QR Solve method . . . 269
30.2.12 Utility methods . . . 269
30.2.13 IO methods . . . 270
30.3 Driver programs for the `DFrontMtx` object . . . 271

31 `ILUMtx`: Incomplete LU Matrix Object 273
31.1 Data Structure . . . 273
31.2 Prototypes and descriptions of `ILUMtx` methods . . . 274
31.2.1 Basic methods . . . 274
31.2.2 Initialization Methods . . . 275
31.2.3 Factorization Methods . . . 275
31.2.4 Solve Methods . . . 275
31.2.5 Utility methods . . . 276
31.2.6 IO methods . . . 276
31.3 Driver programs for the `ILUMtx` object . . . 277

32 `InpMtx`: Input Matrix Object 278
32.1 Data Structure . . . 279
32.2 Prototypes and descriptions of `InpMtx` methods . . . 281
32.2.1 Basic methods . . . 281
32.2.2 Instance Methods . . . 281
32.2.3 Methods to initialize and change state . . . 284
32.2.4 Input methods . . . 285
32.2.5 Permutation, map and support methods . . . 286
32.2.6 Matrix-matrix multiply methods . . . 287
32.2.7 Graph construction methods . . . 289
32.2.8 Submatrix extraction method . . . 290
32.2.9 Utility methods . . . 290
32.2.10 IO methods . . . 292
32.3 Driver programs for the `InpMtx` object . . . 293

33 `Iter`: Iterative Methods 301
33.1 Data Structure . . . 301
33.2 Prototypes and descriptions of `Iter` methods . . . 301
33.2.1 Utility methods . . . 301
33.2.2 Iterative methods . . . 303
33.3 Driver programs . . . 306

34 PatchAndGoInfo: Pivot Modification Object 311
34.1 Data Structure . . . 312
34.2 Prototypes and descriptions of PatchAndGoInfo methods . . . 312
34.2.1 Basic methods . . . 312
34.2.2 Initializer methods . . . 313

35 Pencil: Matrix pencil 314
35.1 Data Structure . . . 314
35.2 Prototypes and descriptions of Pencil methods . . . 314
35.2.1 Basic methods . . . 314
35.2.2 Initialization methods . . . 315
35.2.3 Utility methods . . . 315
35.2.4 IO methods . . . 316

36 SemiImplMtx: Semi-Implicit Factorization 317
36.1 Data Structure . . . 318
36.2 Prototypes and descriptions of SemiImplMtx methods . . . 318
36.2.1 Basic methods . . . 318
36.2.2 Initialization Methods . . . 319
36.2.3 Solve Methods . . . 319
36.2.4 Utility methods . . . 320
36.2.5 IO methods . . . 320
36.3 Driver programs for the SemiImplMtx object . . . 320

37 SubMtx: Submatrix object 322
37.1 Data Structure . . . 323
37.2 Prototypes and descriptions of SubMtx methods . . . 324
37.2.1 Basic methods . . . 325
37.2.2 Instance methods . . . 325
37.2.3 Initialization methods . . . 328
37.2.4 Vector scaling methods . . . 329
37.2.5 Solve methods . . . 329
37.2.6 Utility methods . . . 330
37.2.7 IO methods . . . 331
37.3 Driver programs for the SubMtx object . . . 333

38 SubMtxList: SubMtx list object 337
38.1 Data Structure . . . 338
38.2 Prototypes and descriptions of SubMtxList methods . . . 338
38.2.1 Basic methods . . . 338
38.2.2 Initialization methods . . . 339
38.2.3 Utility methods . . . 339
38.2.4 IO methods . . . 339

39 SubMtxManager: SubMtx object manager **340**
39.1 Data Structure . . . 341
39.2 Prototypes and descriptions of SubMtxManager methods . . . 341
39.2.1 Basic methods . . . 341
39.2.2 Initialization methods . . . 342
39.2.3 Utility methods . . . 342
39.2.4 IO methods . . . 342

40 SymbFac: Symbolic Factorization **343**
40.1 Data Structure . . . 343
40.2 Prototypes and descriptions of SymbFac methods . . . 343
40.2.1 Symbolic factorization methods . . . 343
40.3 Driver programs . . . 344

V Miscellaneous Methods **346**

41 Misc directory **348**
41.1 Prototypes and descriptions of methods in the Misc directory . . . 348
41.1.1 Theoretical nested dissection methods . . . 348
41.1.2 Multiple minimum degree, Nested dissection and multisection wrapper methods . . . 350
41.1.3 Graph drawing method . . . 351
41.1.4 Linear system construction . . . 351
41.2 Driver programs found in the Misc directory . . . 352

VI Multithreaded Methods **361**

42 MT directory **363**
42.1 Data Structure . . . 364
42.2 Prototypes and descriptions of MT methods . . . 364
42.2.1 Matrix-matrix multiply methods . . . 364
42.2.2 Multithreaded Factorization methods . . . 365
42.2.3 Multithreaded QR Factorization method . . . 366
42.2.4 Multithreaded Solve method . . . 366
42.2.5 Multithreaded QR Solve method . . . 367
42.3 Driver programs for the multithreaded functions . . . 367

VII MPI Methods **373**

43 MPI directory **375**
43.1 Data Structure . . . 375
43.1.1 MatMulInfo : Matrix-matrix multiply information object . . . 375

43.2 Prototypes and descriptions of MPI methods 376
43.2.1 Split and redistribution methods 376
43.2.2 Gather and scatter methods 379
43.2.3 Symbolic Factorization methods 379
43.2.4 Numeric Factorization methods 380
43.2.5 Post-processing methods 381
43.2.6 Numeric Solve methods 382
43.2.7 Matrix-matrix multiply methods 383
43.2.8 Broadcast methods 384
43.2.9 Utility methods 385
43.3 Driver programs 386

List of Figures

19.1 GRD7x7: Working storage for the forward sparse factorization of the nested dissection ordering. On the left is the storage required to factor $\widehat{J}$ and its update matrix. On the right is the storage required to factor J and all of its ancestors. Both plots have the same scale. . . . 158

19.2 GRD7x7x7: Four tree plots for a $7 \times 7 \times 7$ grid matrix ordered using nested dissection. The top left tree measure number of original matrix entries in a front. The top right tree measure number of factor matrix entries in a front. The bottom left tree measure number of factor operations in a front for a forward looking factorization, e.g., forward sparse. The bottom right tree measure number of factor operations in a front for a backward looking factorization, e.g., general sparse. 162

25.1 R2D100: domain/separator tree. On the left `heightflag = 'H'` and `coordflag = 'C'`, on the right `heightflag = 'D'` and `coordflag = 'C'`. 218

25.2 R2D100: domain/separator tree. On the left `heightflag = 'H'` and `coordflag = 'P'`, on the right `heightflag = 'D'` and `coordflag = 'P'`. 218

41.1 R2D100 355

41.2 R2D100: FISHNET DOMAIN DECOMPOSITION 356

Part I

Introduction

Chapter 1

Introduction

The **SPOOLES** package is used to solve two types of real or complex linear systems:

- $AX = Y$ or $(A + \sigma B)X = Y$ where A and B are square. A and B can be real or complex, symmetric, Hermitian or nonsymmetric. The factorization can proceed with or without pivoting for numerical stability. The factor matrices can be stored with or without dropping small entries.

- Minimize $\|AX_{*,j} - Y_{*,j}\|_2$ for each column of the solution matrix X and right hand side matrix Y. This is done by computing a QR factorization of A and then solving $R^T RX = A^T Y$ or $R^H RX = A^H Y$.

In both cases, the linear systems can be permuted to reduce the fill in the factor matrices.

The **SPOOLES** software is written in an object oriented fashion in the C language. Parts of the software run in serial mode, multithreading using Solaris or POSIX threads, and with MPI.

The software objects are naturally partitioned into three families of objects.

Utility objects

`A2`	dense two dimensional array
`Coords`	object to hold coordinates in any number of dimensions
`DV`	double precision vector
`Drand`	random number generator
`I2Ohash`	hash table for the factor submatrices
`IIheap`	simple heap object
`IV`	int vector
`IVL`	int list object
`Ideq`	simple dequeue object
`Lock`	abstract mutual exclusion lock
`Perm`	permutation vector object
`Utilities`	various vector and linked list utility methods
`ZV`	double precision complex vector

Ordering objects

BKL	Block Kernihan-Lin algorithm object
BPG	bipartite graph object
DSTree	domain/separator tree object
EGraph	element graph object
ETree	front tree object
GPart	graph partitioning algorithm object
Graph	graph object
MSMD	multi-stage minimum degree algorithm object
Network	network object for solving max flow problems
SolveMap	map of submatrices to processes for solves
Tree	tree object

Numeric objects

Chv	block chevron object for fronts
ChvList	object to hold lists of Chv objects
ChvManager	object to manager instances of Chv objects
DenseMtx	dense matrix object
FrontMtx	front matrix object
ILUMtx	simple preconditioner matrix object
InpMtx	sparse matrix object
Iter	Krylov methods for iterative solves
PatchAndGoInfo	modified factors in the presence of zero or small pivots
Pencil	object to contain $A + \sigma B$
SemiImplMtx	semi-implicit factorization matrix object
SubMtx	object for dense or sparse submatrices
SubMtxList	object to hold lists of SubMtx objects
SubMtxManager	object to manager instances of SubMtx objects
SymbFac	algorithm object to compute a symbolic factorization

The `MT` directory contains all the multithreaded methods and drivers programs. The `MPI` directory contains all the `MPI` methods and drivers. The `misc` directory contains miscellaneous methods and drivers.

Each of the following objects that hold numeric entries — `A2`, `Chv`, `DenseMtx`, `FrontMtx`, `ILUMtx`, `InpMtx`, `Pencil`, `SemiImplMtx` and `SubMtx` — can hold real or complex entries. An object knows its `type`, `1` for real (define'd constant `SPOOLES_REAL`) or `2` for complex (define'd constant `SPOOLES_COMPLEX`). Since C does not yet have a standard structure for complex numbers, we have followed the FORTRAN convention of storing the real and imaginary parts of a complex number in consecutive memory locations. Internally, we unroll the complex arithmetic into real arithmetic. The user need not be burdened by this process if (s)he uses the input/output methods for the different object. For example, `DenseMtx_setRealEntry()` sets an entry of a real dense matrix, while `DenseMtx_setComplexEntry()` sets an entry of a complex dense matrix.

All the heavily used computational tasks have been expanded where possible into BLAS2 or BLAS3 kernels, for both the real and complex cases. There are a multitude of driver programs that test the functionality of the objects. A common output of a driver program is a file that can be input into Matlab to check the errors of the computations. This convention inspires confidence in the correctness of the kernel computations.

1.1 Software Design

The **SPOOLES** library is written in the C language and uses object oriented design. There are some routines that manipulate native C data types such as vectors, but the vast bulk of the code is centered around objects, data objects and algorithm objects. By necessity, the implementation of an object is through the C `struct` data type. We use the following naming convention — a method (i.e., function) associated with an object of type `Object` has the form

(return value type) `Object_`*methodName*`(Object * obj, ...);`

The method's name begins with the name of the object it is associated with and the first parameter in the calling sequence is a pointer to the instance of the object. Virtually the only exception to this rule is the *constructor* method.

`Object * Object_new(void) ;`

Two objects, the `Chv` and `DenseMtx` objects, have methods that return the number of bytes needed to hold their data, e.g.,

`int Chv_nbytesNeeded(int nD, int nL, int nU, int type, int symflag) ;`

Scan the directory structure of the source code and you will notice a number of subdirectories — each deals with an object. For example, the `Graph` directory holds code and documentation for an object that represents a graph: its `doc` subdirectory holds LaTeXfiles with documentation; its `src` subdirectory holds C files that contain methods associated with the object ; and its `driver` subdirectory holds driver programs to test or validate some behavior of the object.

The directory structure is fairly flat — no object directory contains another — because the C language does not support inheritance. This can be inelegant at times. For example, a bipartite graph (a `BPG` object) *is–a* graph (a `Graph` object), but instead of `BPG` inheriting from `Graph` data fields and methods from `Graph`, we must use the *has–a* relation. A `BPG` object contains a pointer to a `Graph` object that represents the adjacency structure. The situation is even more cumbersome for the objects that deal with trees of one form or another: an elimination tree `ETree` and a domain/separator tree `DSTree` each contain a pointer to a generic tree object `Tree` in their structure.

Predecessors to this library were written in C++ and Objective-C.[1] The port to the present C library was painless, almost mechanical. We expect the port back to C++ and/or Objective-C to be simple.

Objects are one of two types: *data objects* whose primary function is to store data and *algorithm objects* whose function is to manipulate some data object(s) and return new information. Naturally this distinction can be fuzzy — algorithm objects have their own data that may be persistent and data objects can execute some simple functionality — but it holds in general. To be more explicit, data objects have the following properties:

- There is a delicate balance between encapsulation and openness. The C language does not support any private or protected data fields, so the C `struct` that holds the data for an object is completely open. As an example, the `Graph` object has a function to return the size of and pointers to a vector that contains an adjacency list, namely

  ```
  void Graph_adjAndSize(Graph *g, int v, int *psize, int **padj)
  ```

  where the pointers `psize` and `padj` are filled with the size of the adjacency structure and a pointer to its vector. One can get this same information by chasing pointers as follows.

  ```
  vsize = g->adjIVL->sizes[v] ;
  vadj  = g->adjIVL->p_ind[v] ;
  ```

  One can do the latter but we encourage the former. As an experiment we replaced every instance of `Graph_adjAndSize()` with the appropriate pointer chasing (and a similar operation for the `IVL` object) and achieved around a ten per cent reduction in the ordering time. For a production code, this savings might drive the change in code, but for our research code we kept the function call.

[1] The knowledgeable reader is encouraged to peruse the source to discover the prejudices both pro and con towards these two languages.

- Persistent storage needs to be supported. Each data object has eight different methods to deal with file I/O. Two methods deal with reading from and writing to a file whose suffix is associated with the object name, e.g., `*graph{f,b}` for a formatted or binary file to hold a `Graph` object. Four methods deal with reading and writing objects from and to a file that is already opened and positioned, necessary for composite objects (e.g., a `Graph` object contains an `IVL` object). Two methods deal with writing the objects to a formatted file to be examined by the user. We strongly encourage any new data object added to the library to supply this functionality.
- Some data objects need to have compact storage requirements. Two examples are our `Chv` and `SubMtx` objects. Both objects need to be communicated between processes in the MPI implementation, the former during the factorization, the latter during the solve. Each has a workspace buffer that contains all the information needed to *regenerate* the object upon reception by another process.
- By and large, data objects have simple methods. A `Graph` object does **not** have methods to find a good bisector; this is a sufficiently sophisticated function that it should be implemented by an algorithm object. The major exception to this rule is that our `FrontMtx` object *contains* the factorization data but also *performs* the factorization, forward and backsolves. In the future we intend to separate these two functionalities. For example, one can implement an alternative forward and backsolve by using methods to *access* the factor data stored in the `FrontMtx` object. As a second example, massive changes to the storage format, e.g., in an out-of-core implementation, can be encapsulated in the access methods for the data, and any changes to the factorization or solve functions could be minimal.

Algorithm objects have these properties.

- Algorithm objects use data objects. Some data objects are created within an algorithm objects method; these are owned by the algorithm object and free'd by that object. Data objects that are passed to algorithm objects can be queried or *temporarily* changed.
- They do not destroy or free data objects that are passed to them. Any side effects on the data objects should be innocent, e.g., when a `Graph` object is passed to the graph partitioning object (`GPart`) or the multistage minimum degree object (`MSMD`), on return the adjacency lists may not be in the input order, but they contain the values they had on input.
- Algorithm objects should support diagnostic, logfile and debug output. This convention is not entirely thought out or supported at present. The rationale is that an algorithm object should be able to respond to its environment to a greater degree than a data object.

Data and algorithm objects share two common properties.

- Each object has four basic methods: to allocate storage for an object, set the default fields of an object, clear the data fields of an object, and free the storage occupied and owned by an object.
- Ownership of data is very rigidly defined. In most cases, an object owns all data that is allocated inside one of its methods, and when this does not hold it is very plainly documented. For example, the bipartite graph object `BPG` has a data field that points to a `Graph` object. One of its initialization methods has a `Graph` pointer in its calling sequence. The `BPG` object then owns the `Graph` object and when it is free'd or has its data cleared, the `Graph` object is free'd by a call to its appropriate method.

By and large these conventions hold throughout the library. There are fuzzy areas and objects still "under construction". Here are two examples.

- We have an `IIheap` object that maintains integer $\langle$ key, value $\rangle$ pairs in a priority heap. Normally we think of a heap as a data structure, but another perspective is that of a continuously running algorithm that supports insert, delete and identification of a minimum pair.

- Our BPG bipartite graph object is a data object, but it has a method to find the Dulmage-Mendelsohn decomposition, a fairly involved algorithm used to refine a separator of a graph. At present, we are not willing to create a new algorithm object just to find the Dulmage-Mendelsohn decomposition, so we leave this method to the domain of the data object. The desired functionality, identifying minimal weight separators for a region of a graph, can be modeled using max flow techniques from network optimization. We also provide a BPG method that finds this Dulmage-Mendelsohn decomposition by solving a max flow problem on a bipartite network. Both these methods have been superceded by the Network object that contains a method to find a max flow and one or more min-cuts of a network (not necessarily bipartite).

The **SPOOLES** software library is continuously evolving in an almost organic fashion. Growth and change are to be expected, and welcomed, but some discipline is required to keep the complexity, both code and human understanding, under control. The guidelines we have just presented have two purposes: to let the user and researcher get a better understanding of the design of the library, and to point out some conventions to be used in extending the library.

1.2 Changes from Release 1.0

There are two major changes from the first release of the **SPOOLES** package: we now support complex linear systems, and the storage format of the sparse factor matrices has changed from a one-dimensional data decomposition to a two-dimensional decomposition. The factors are now submatrix based, and thus allow a parallel solve to be much faster than in Release 1.0.

In the first release, all numeric objects had a 'D' as the leading letter in their name, e.g., DA2, DChv, etc. A natural way to implement complex data types would be to write "parallel" objects, e.g., ZA2, ZChv, etc, as is done in LINPACK and LAPACK for subroutine names. However, a DA2 and ZA2 object share so much common code that it is a better decision to combine the real and complex functionality into one object. This is even more pronounced for the FrontMtx object where there is virtually no code that is dependent on whether or not the matrix is real or complex.

Virtually no new work has been done on the ordering objects and methods. Their algorithms were state of the art two years ago, but a recent comparison with the **EXTREME** [13] and **METIS** [14] packages on a large collection of finite element problems shows that the **SPOOLES** orderings are still competitive.

The serial, multithreaded and MPI code has been modified to force greater sharing of code between the environments. "What" is done is identical in the three cases. The multithreaded and MPI codes share the same "choreography", in other words, who does what and how. The main differences between multithreaded and MPI are that the data structures are global versus local, and that explicit message passing is done in the latter. This common structure of the codes has a nonzero impact on the speed and efficiency of the individual codes, but the gains from a common code base are well worth the cost.

The MPI methods have been extensively reworked since the first release. A number of bugs and logic errors have been detected and fixed. The code appears to be more robust than the first release.

1.3 Changes from Release 2.0

Release 2.2 is partly a maintenance release. Some bugs were found and fixed in the MPI factors and solves. Some minor new methods were added to the DenseMtx, FrontMtx, InpMtx and Utilities directories. The multithreaded methods and drivers have been removed from the FrontMtx directory and placed in a new MT directory, much like the MPI methods have their own directory.

Some new functionality has been added.

- There are now multithreaded and distributed matrix-matrix multiply methods. See the `MT` and `MPI` directories.
- The `FrontMtx` object now supports more robust reporting of errors encountered during the factorization. There is one additional parameter in the factorization calling sequences, an error return that signals that the factorization has failed.
- In response to customer requirements, we have added some "patch-and-go" functionality to the sparse LU and U^TDU factorizations without pivoting. There are applications in optimization and structural analysis where pivoting is not necessary for stabilty, but where the location of small or zero pivots on the diagonal is meaningful. Normally the factorization would be ustable or stop, but special action is taken, the factors are "patched" and the factorization continues.

  There is a new `PatchAndGoInfo` object that encapsulates the "patch-and-go" strategy and gathers optional statistics about the action that was taken during the factorization. This object is attached to the `FrontMtx` object which passes it unchanged to the `Chv` object that performs the factorization of each front. If the user does not need this functionality, no changes are necessary to their code, i.e., no calling sequences are affected.
- New MPI broadcast methods for the `Graph`, `IVL` and `ETree` objects have been added to the library.
- The `Iter` directory contains the following Krylov accelerators for the iterative solution of linear systems: Block GMRES, BiCGStab, conjugate gradient and transpose-free QMR. Each is available in both left- and right-preconditioned forms. The preconditioner that these methods use is a `FrontMtx` object that contains a drop tolerance approximate factorization. The `ILUMtx` object contains a simple vector-based drop tolerance factorization object. (The `FrontMtx` approximate factorization is submatrix-based in both its data structures and computational kernels, and supports pivoting for numerical stability, which the `ILUMtx` object does not.) We have not written Krylov methods that use the `ILUMtx` object, but it would be simple to replace the `FrontMtx` preconditioner with the `ILUMtx` preconditioner.
- The `SemiImplMtx` object contains a *semi-implicit* factorization, a technique that can require less storage and solve operations than the present explicit factorization. It is based on the equation

$$\left[\begin{array}{cc} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{array}\right] = \left[\begin{array}{cc} L_{0,0} & 0 \\ L_{1,0} & L_{1,1} \end{array}\right] \left[\begin{array}{cc} U_{0,0} & U_{0,1} \\ 0 & U_{1,1} \end{array}\right], = \left[\begin{array}{cc} L_{0,0} & 0 \\ A_{1,0}U_{0,0}^{-1} & L_{1,1} \end{array}\right] \left[\begin{array}{cc} U_{0,0} & L_{0,0}^{-1}U_{0,1} \\ 0 & U_{1,1} \end{array}\right].$$

  A solve of $AX = B$ with the explicit factorization does the following steps

  - solve $L_{0,0}Y_0 = B_0$
  - solve $L_{1,1}U_{1,1}X_1 = B_1 - L_{1,0}Y_0$
  - solve $U_{0,0}X_0 = Y_0 - U_{0,1}X_1$

  while an implicit factorization has the following form.

  - solve $L_{0,0}U_{0,0}Z_0 = B_0$
  - solve $L_{1,1}U_{1,1}X_1 = B_1 - A_{1,0}Z_0$
  - solve $L_{0,0}U_{0,0}X_0 = B_0 - A_{0,1}X_1$

  The difference is that the semi-implicit factorization stores and computes with $A_{1,0}$ and $A_{0,1}$ instead of $L_{1,0}$ and $U_{0,1}$, (this can be a modest savings in storage and operation count), and performs two solves with $L_{0,0}$ and $U_{0,0}$ instead of one. This technique works with either a direct or approximate factorization of A. The semi-implicit factorization is constructed via a post-processing of any factorization computed by the `FrontMtx` object.

Part II

Utility Objects and Methods

Chapter 2

A2: Real or complex 2-D array

The A2 object is one way to store and operate on and with a dense matrix. The matrix can contain either double precision real or complex entries. It is used as a lower level object for the DenseMtx object, and during the QR factorization to hold a staircase matrix.

2.1 Data Structure

The A2 structure has six fields.

- `int type` : type of entries, SPOOLES_REAL or SPOOLES_COMPLEX
- `int n1` : size in first dimension, number of rows
- `int n2` : size in second dimension, number of columns
- `int inc1` : increment or stride in first dimension
- `int inc2` : increment or stride in second dimension
- `int nowned` : the number of entries that are "owned" by this object. When `nowned > 0`, `entries` points to storage for `nowned` entries, (`nowned double`'s for the real case, `2*nowned double`'s for the complex case), that have been allocated by this object and can be free'd by the object. When `nowned == 0` but `n1 > 0` and `n2 > 0`, this object points to entries that have been allocated elsewhere, and these entries will not be free'd by this object.
- `double *entries` : pointer to the base address of the *double* vector

One can query the properties of the front matrix object using these simple macros.

- `A2_IS_REAL(mtx)` is `1` if `mtx` has real entries and `0` otherwise.
- `A2_IS_COMPLEX(mtx)` is `1` if `mtx` has complex entries and `0` otherwise.

The `A2_copyEntriesToVector()` method uses the following constants: `A2_STRICT_LOWER`, `A2_LOWER`, `A2_DIAGONAL`, `A2_UPPER`, `A2_STRICT_UPPER`, `A2_ALL_ENTRIES`, `A2_BY_ROWS` and `A2_BY_COLUMNS`.

2.2 Prototypes and descriptions of A2 methods

This section contains brief descriptions including prototypes of all methods that belong to the A2 object.

2.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `A2 * A2_new ( void ) ;`

   This method simply allocates storage for the `A2` structure and then sets the default fields by a call to `A2_setDefaultFields()`.

2. `void A2_setDefaultFields ( A2 *mtx ) ;`

   The structure's fields are set to default values: `type` = `SPOOLES_REAL`, `n1` = `inc1` = `n2` = `inc2` = `nowned` = 0 and `entries` = `NULL` .

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `void A2_clearData ( A2 *mtx ) ;`

   This method clears the object and free's any owned data. If `nowned > 0` and `entries` is not `NULL`, then `DVfree(entries)` is called to free the storage. It calls `A2_setDefaultFields()`.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. `void A2_free ( A2 *mtx ) ;`

   This method releases any storage by a call to `A2_clearData()` and then free the space for `mtx`.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

2.2.2 Instance methods

1. `int A2_nrow ( A2 *mtx ) ;`

   This method returns the number of rows in the matrix.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

2. `int A2_ncol ( A2 *mtx ) ;`

   This method returns the number of columns in the matrix.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `int A2_inc1 ( A2 *mtx ) ;`

   This method returns the primary increment, the stride in memory (with respect to real or complex entries) between adjacent entries in the same column.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. `int A2_inc2 ( A2 *mtx ) ;`

   This method returns the secondary increment, the stride in memory (with respect to real or complex entries) between adjacent entries in the same row.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

5. `double * A2_entries ( A2 *mtx ) ;`

   This method returns a pointer to the base address of the entries.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

6. `double * A2_row ( A2 *mtx, int irow ) ;`

   This method returns a pointer to the leading element of row `irow`.

   Error checking: If `mtx` or `entries` is `NULL`, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

7. `double * A2_column ( A2 *mtx, int jcol ) ;`

   This method returns a pointer to the leading element of column `jcol`.

   Error checking: If `mtx` or `entries` is `NULL`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

8. `void A2_realEntry ( A2 *mtx, int irow, int jcol, double *pValue ) ;`

   This method fills `*pValue` with the entry in location `(irow, jcol)`.

   Error checking: If `mtx` or `pValue` is `NULL`, or if the matrix is not real, or `irow` is not in `[0,n1-1]`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

9. ```
   void A2_complexEntry ( A2 *mtx, int irow, int jcol,
                          double *pReal, double *pImag ) ;
   ```

   This method fills `(*pReal,*pImag)` with the entry in location `(irow, jcol)`.

   Error checking: If `mtx`, `pReal` or `pImag` is `NULL`, or if the matrix is not complex, or `irow` is not in `[0,n1-1]`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

10. `void A2_setRealEntry ( A2 *mtx, int irow, int jcol, double value ) ;`

    This method sets entry `(irow,jcol)` to `value`.

    Error checking: If `mtx` is `NULL`, or if the matrix is not real, or `irow` is not in `[0,n1-1]` or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

11. ```
    void A2_setComplexEntry ( A2 *mtx, int irow, int jcol,
                              double real, double imag ) ;
    ```

    This method sets entry `(irow,jcol)` to `(real,imag)`.

    Error checking: If `mtx` is `NULL`, or if the matrix is not complex, or `irow` is not in `[0,n1-1]` or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

12. `void A2_pointerToRealEntry ( A2 *mtx, int irow, int jcol, double **ppValue ) ;`

    This method sets `*ppValue` to the pointer of the `(irow,jcol)` entry.

    Error checking: If `mtx` or `ppValue` is `NULL`, or if the matrix is not real, or if `irow` is not in `[0,n1-1]`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

13. ```
    void A2_pointerToComplexEntry ( A2 *mtx, int irow, int jcol,
                                    double **ppReal, double **ppImag ) ;
    ```

    This method sets `*ppReal` to the pointer to the real part of the `(irow,jcol)` entry, and sets `*ppImag` to the pointer to the imaginary part of the `(irow,jcol)` entry.

    Error checking: If `mtx`, `ppReal` or `ppImag` is `NULL`, or if the matrix is not complex, or if `irow` is not in `[0,n1-1]`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.
    ```

2.2.3 Initialize methods

1. `void A2_init ( A2 *mtx, int type, int n1, int n2, int inc1, int inc2, double *entries ) ;`

   This is the basic initializer method. We require that `mtx` not be `NULL`, `type` be either `SPOOLES_REAL` or `SPOOLES_COMPLEX`, `n1` and `n2` both be positive, and both `inc1` and `inc2` both be positive and that one of them be equal to one. Also, we only initialize a full matrix, i.e., one of `inc1 = 1` and `inc2 = nrow` or `inc1 = ncol` and `inc2 = 1` must hold.

   The object is first cleared with a call to `A2_clearData()`. If `entries` is `NULL` then `n1*n2` new entries are found, `mtx->entries` is set to this address and `nowned` is set to `n1*n2`. If `entries` is not `NULL`, then `mtx->entries` is set to `entries` and `nowned` is set to zero.

   Error checking: If `mtx` is `NULL`, or if `n1`, `n2`, `inc1` or `inc2` are less than or equal to zero, or if the matrix is not full matrix (i.e., `inc1` must be `1` and `inc2` must be `n1`, *or* `inc1` must be `n2` and `inc2` must be `1`), an error message is printed and zero is returned.

2. `void A2_subA2 ( A2 *mtxA, A2 *mtxB, int firstrow, int lastrow, int firstcol, int lastcol ) ;`

   This initializer method makes the object `mtxA` point into a submatrix of object `mtxB`, as

   `A(0:lastrow-firstrow,0:lastcol-firstcol) = B(firstrow:lastrow, firstcol:lastcol)`

   Note, `firstrow`, `lastrow`, `firstcol` and `lastcol` must satisfy `0 <= firstrow <= lastrow < mtxB->n1` and `0 <= firstcol <= lastcol < mtxB->n2`. Object `mtxA` does not own its entries, but points into the entries of `mtxB`.

   Error checking: If `mtxA` or `mtxB` are `NULL`, or if `firstrow` or `lastrow` are out of range, or if `firstcol` or `lastcol` are out of range, an error message is printed and zero is returned.

2.2.4 Methods used in the QR factorization

1. `void A2_makeStaircase ( A2 *A ) ;`

   This method permutes the rows of `A` by the location of the leading nonzero of each row. Upon return, the matrix is in *staircase* form.

   Error checking: If `A` is `NULL`, an error message is printed and the program exits.

2. `double A2_QRreduce ( A2 *A, DV *workDV, int msglvl, FILE *msgFile ) ;`

   This method computes $A = QR$ factorization. On return, the matrix Q is not available, and R is found in the upper triangle or upper trapezoid of `A`. The Householder vectors are stored in the lower triangle of `mtxA`, with $v_j(j) = 1.0$. The return value is the number of floating point operations that were executed.

   Error checking: If `A` or `workDV` is `NULL`, or if `msglvl > 0` and `msgFile` if `NULL`, an error message is printed and the program exits.

3. `void A2_computeQ ( A2 *Q, A2 *A, DV *workDV, int msglvl, FILE *msgFile ) ;`

   This method computes Q from the $A = QR$ factorization computed in `A2_QRreduce()`. Note: `A` and `Q` must be column major.

   Error checking: If `Q`, `A` or `workDV` is `NULL`, or if `msglvl > 0` and `msgFile` if `NULL`, an error message is printed and the program exits.

4. `void A2_applyQT ( A2 *Y, A2 *A, A2 *X, DV *workDV, int msglvl, FILE *msgFile ) ;`

   This method computes $Y = Q^T X$ (if real) or $Y = Q^H X$ (if complex), where Q is stored in Householder vectors inside A. We assume that *A2_reduce*() has been previously called with A as an argument. Since Y is computed column-by-column, X and Y can be the same `A2` object. The `workDV` object is resized as necessary. Note: `Y`, `A` and `X` must be column major.

   Error checking: If `Y`, `A`, `X` or `workDV` is `NULL`, or if `msglvl > 0` and `msgFile` if `NULL`, or if `Y`, `A` or `X` is not column major, or if the types of `Y`, `A` and `X` are not the same, an error message is printed and the program exits.

2.2.5 Norm methods

These methods return a norm of a row or a column, or the easily computable norms of the matrix.

1. `double A2_maxabs ( A2 *mtx ) ;`

   This method returns magnitude of the entry with largest magnitude.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

2. `double A2_frobNorm ( A2 *mtx ) ;`

   This method returns the Frobenius norm of the matrix.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `double A2_oneNorm ( A2 *mtx ) ;`

   This method returns the one norm of the matrix.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. `double A2_infinityNorm ( A2 *mtx ) ;`

   This method returns the infinity norm of the matrix.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

5. `double A2_oneNormOfColumn ( A2 *mtx, int jcol ) ;`

   This method returns the one-norm of column `jcol` of the matrix.

   Error checking: If `mtx` is `NULL`, or `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

6. `double A2_twoNormOfColumn ( A2 *mtx, int jcol ) ;`

   This method returns the two-norm of column `jcol` of the matrix.

   Error checking: If `mtx` is `NULL`, or `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

7. `double A2_infinityNormOfColumn ( A2 *mtx, int jcol ) ;`

   This method returns the infinity-norm of column `jcol` of the matrix.

   Error checking: If `mtx` is `NULL`, or `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

8. `double A2_oneNormOfRow ( A2 *mtx, int irow ) ;`

   This method returns the one-norm of row `irow` of the matrix.

   Error checking: If `mtx` is `NULL`, or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

9. `double A2_twoNormOfRow ( A2 *mtx, int irow ) ;`

   This method returns the two-norm of row `irow` of the matrix.

   Error checking: If `mtx` is `NULL`, or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

10. `double A2_infinityNormOfRow ( A2 *mtx, int irow ) ;`

    This method returns the infinity-norm of row `irow` of the matrix.

    Error checking: If `mtx` is `NULL`, or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

2.2.6 Sort methods

1. `void A2_permuteRows ( A2 *mtx, int nrow, int index[] ) ;`

   The `index[]` vector contains the *row ids* of the leading `nrow` rows. This method permutes the leading `nrow` rows of the matrix so that the `index[]` vector is in ascending order. This method calls `A2_permuteRows()` but does not overwrite the `index[]` vector.

   Error checking: If `mtx` or `index[]` is `NULL`, or if `nrow < 0` or `nrow > n1`, an error message is printed and the program exits.

2. `void A2_permuteColumns ( A2 *mtx, int nrow, int index[] ) ;`

   The `index[]` vector contains the *column ids* of the leading `ncol` rows. This method permutes the leading `ncol` columns of the matrix so that the `index[]` vector is in ascending order. This method calls `A2_permuteColumns()` but does not overwrite the `index[]` vector.

   Error checking: If `mtx` or `index[]` is `NULL`, or if `ncol < 0` or `ncol > n2`, an error message is printed and the program exits.

3. `int A2_sortRowsUp ( A2 *mtx, int nrow, int rowids[] ) ;`

   This method sorts the leading `nrow` rows of the matrix into ascending order with respect to the `rowids[]` vector. The return value is the number of row swaps made.

   Error checking: If `mtx` or `rowids` is `NULL`, or if `nrow < 0` or `nrow > n1`, an error message is printed and the program exits.

4. `int A2_sortColumnsUp ( A2 *mtx, int ncol, int colids[] ) ;`

   This method sorts the leading `ncol` columnss of the matrix into ascending order with respect to the `colids[]` vector. The return value is the number of column swaps made.

   Error checking: If `mtx` or `colids` is `NULL`, or if `ncol < 0` or `ncol > n2`, an error message is printed and the program exits.

2.2.7 Utility methods

1. `int A2_sizeOf ( A2 *mtx ) ;`

   This method returns the number of bytes owned by this object.

   Error checking: If `mtx` is `NULL` an error message is printed and the program exits.

2. `void A2_shiftBase ( A2 *mtx, int rowoff, int coloff ) ;`

   This method is used to shift the base of the entries and adjust dimensions of the `A2` object.

   ```
   mtx(0:n1-rowoff-1,0:n2-coloff-1) := mtx(rowoff:n1-1,coloff:n2-1)
   ```

Error checking: If `mtx` is `NULL` an error message is printed and the program exits.

3. `int A2_rowMajor ( A2 *mtx ) ;`

   This method returns 1 if the storage is row major, otherwise it returns zero.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. `int A2_columnMajor ( A2 *mtx ) ;`

   This method returns 1 if the storage is column major, otherwise it returns zero.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

5. `void A2_transpose ( A2 *mtx ) ;`

   This method replaces `mtx` with its transpose. Note, this takes $O(1)$ operations since we just swap dimensions and increments.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

6. `void A2_extractRow ( A2 *mtx, double row[], int irow ) ;`

   This method fills the `row[]` vector with row `irow` of the matrix.

   Error checking: If `mtx`, `entries` or `row` are `NULL`, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

7. `void A2_extractRowDV ( A2 *mtx, DV *rowDV, int irow ) ;`

   This method fills the `rowDV` object with row `irow` of the matrix.

   Error checking: If `mtx` or `rowDV` are `NULL`, or if the matrix is not real, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

8. `void A2_extractRowZV ( A2 *mtx, ZV *rowZV, int irow ) ;`

   This method fills the `rowZV` object with row `irow` of the matrix.

   Error checking: If `mtx` or `rowZV` are `NULL`, or if the matrix is not complex, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

9. `void A2_extractColumn ( A2 *mtx, double col[], int jcol ) ;`

   This method fills the `col[]` vector with column `jcol` of the matrix.

   Error checking: If `mtx`, `entries` or `col` are `NULL`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

10. `void A2_extractColumnDV ( A2 *mtx, DV *colDV, int jcol ) ;`

    This method fills the `colDV` object with column `jcol` of the matrix.

    Error checking: If `mtx` or `colDV` are `NULL`, or if the matrix is not complex, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

11. `void A2_extractColumnZV ( A2 *mtx, ZV *colZV, int jcol ) ;`

    This method fills the `colZV` object with column `jcol` of the matrix.

    Error checking: If `mtx` or `colZV` are `NULL`, or if the matrix is not complex, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

12. `void A2_setRow ( A2 *mtx, double row[], int irow ) ;`

    This method fills row `irow` of the matrix with the entries in the `row[]` vector.

    Error checking: If `mtx`, `entries` or `row[]` are `NULL`, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

13. `void A2_setRowDV ( A2 *mtx, DV rowDV, int irow ) ;`

    This method fills row `irow` of the matrix with the entries in the `rowDV` object.

    Error checking: If `mtx` or `rowDV` are `NULL`, or if the matrix is not real, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

14. `void A2_setRowZV ( A2 *mtx, ZV rowZV, int irow ) ;`

    This method fills row `irow` of the matrix with the entries in the `rowZV` object.

    Error checking: If `mtx` or `rowZV` are `NULL`, or if the matrix is not complex, or if `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

15. `void A2_setColumn ( A2 *mtx, double col[], int jcol ) ;`

    This method fills column `jcol` of the matrix with the entries in the `col[]` vector.

    Error checking: If `mtx` or `colZV` are `NULL`, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

16. `void A2_setColumnDV ( A2 *mtx, DV colDV, int jcol ) ;`

    This method fills column `jcol` of the matrix with the entries in the `colDV` object.

    Error checking: If `mtx` or `colDV` are `NULL`, or if the matrix is not complex, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

17. `void A2_setColumnZV ( A2 *mtx, ZV colZV, int jcol ) ;`

    This method fills column `jcol` of the matrix with the entries in the `colZV` object.

    Error checking: If `mtx` or `colZV` are `NULL`, or if the matrix is not complex, or if `jcol` is not in `[0,n2-1]`, an error message is printed and the program exits.

18. `void A2_fillRandomUniform ( A2 *mtx, double lower, double upper, int seed ) ;`

    This method fills the matrix with random numbers taken from a uniform distribution on `[lower,upper]` using the `Drand` object.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

19. `void A2_fillRandomNormal ( A2 *mtx, double mean, double variance, int seed ) ;`

    This method fills the matrix with random numbers taken from a normal distribution with mean `mean` and variance `variance` using the `Drand` object.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

20. `void A2_fillWithIdentity ( A2 *mtx ) ;`

    This method fills the matrix with the identity matrix.

    Error checking: If `mtx` is `NULL` or if `n1 != n2`, an error message is printed and the program exits.

21. `void A2_zero ( A2 *mtx ) ;`

    This method fills the matrix with zeros.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

22. `void A2_copy ( A2 *mtxA, A2 *mtxB ) ;`

    This method copies entries from matrix `mtxB` into matrix `mtxA`. Note, `mtxA` and `mtxB` need not be of the same size, the leading `min(mtxA->n1,mtxB->n1)` rows and `min(mtxA->n2,mtxB->n2)` columns are copied.

    Error checking: If `mtxA` or `mtxB` is `NULL`, or if the matrices are not of the same type, an error message is printed and the program exits.

23. `void A2_sub ( A2 *mtxA, A2 *mtxB ) ;`

    This method subtracts entries in matrix `mtxB` from entries in matrix `mtxA`. Note, `mtxA` and `mtxB` need not be of the same size, the leading `min(mtxA->n1,mtxB->n1)` rows and `min(mtxA->n2,mtxB->n2)` columns are subtracted.

    Error checking: If `mtxA` or `mtxB` is `NULL`, or if the matrices are not of the same type, an error message is printed and the program exits.

24. `void A2_swapRows ( A2 *mtx, int irow1, int irow2 ) ;`

    This method swaps rows `irow1` and `irow2` of the matrix.

    Error checking: If `mtxA` or `mtxB` is `NULL`, or if `irow1` or `irow2` are out of range, an error message is printed and the program exits.

25. `void A2_swapColumns ( A2 *mtx, int irow1, int irow2 ) ;`

    This method swaps columns `jcol1` and `jcol2` of the matrix.

    Error checking: If `mtxA` or `mtxB` is `NULL`, or if `jcol1` or `jcol1` are out of range, an error message is printed and the program exits.

26. ```
    int A2_copyEntriesToVector ( A2 *mtx, int length, double dvec[],
                                 int copyflag, int storeflag ) ;
    ```

    This method copies selected entries from `mtx` into the vector `dvec[]` with length `length`. The return value is the number of entries copied. This method is used during the QR factorization to extract factor entries and update matrix entries from a front. All entries may be copied, or only the diagonal, lower or upper entries, and the entries may be copied to `dvec[]` by rows or by columns.

    Error checking: If `mtx` or `dvec` is `NULL`, or if `length` is not as large as the number of entries to be copied, or if `copyflag` is not one of `A2_STRICT_LOWER`, `A2_LOWER`, `A2_DIAGONAL`, `A2_UPPER`, `A2_STRICT_UPPER` or `A2_ALL_ENTRIES`, or if `storeflag` is not one of `A2_BY_ROWS` or `A2_BY_COLUMNS`, an error message is printed and the program exits.

2.2.8 IO methods

There are the usual eight IO routines plus a method to write the object to a Matlab file.

1. `int A2_readFromFile ( A2 *mtx, char *fn ) ;`

   This method reads a `A2` object from a file. It tries to open the file and if it is successful, it then calls `A2_readFromFormattedFile()` or `A2_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `mtx` or `fn` are `NULL`, or if `fn` is not of the form `*.a2f` (for a formatted file) or `*.a2b` (for a binary file), an error message is printed and the method returns zero.

2. `int A2_readFromFormattedFile ( A2 *mtx, FILE *fp ) ;`

   This method reads a `A2` object from a formatted file whose pointer is `fp`. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

3. `int A2_readFromBinaryFile ( A2 *mtx, FILE *fp ) ;`

   This method reads a `A2` object from a binary file whose pointer is `fp`. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.
    ```

4. `int A2_writeToFile ( A2 *mtx, char *fn ) ;`

   This method writes a `A2` object to a file. It tries to open the file and if it is successful, it then calls `A2_writeFromFormattedFile()` or `A2_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `mtx` or `fn` are `NULL`, or if `fn` is not of the form `*.a2f` (for a formatted file) or `*.a2b` (for a binary file), an error message is printed and the method returns zero.

5. `int A2_writeToFormattedFile ( A2 *mtx, FILE *fp ) ;`

   This method writes a `A2` object to a formatted file whose pointer is `fp`. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

6. `int A2_writeToBinaryFile ( A2 *mtx, FILE *fp ) ;`

   This method writes a `A2` object to a binary file whose pointer is `fp`. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

7. `void A2_writeForHumanEye ( A2 *mtx, FILE *fp ) ;`

   This method writes a `A2` object to a file in an easily readable format. The method `A2_writeStats()` is called to write out the header and statistics.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

8. `void A2_writeStats ( A2 *mtx, FILE *fp ) ;`

   This method writes a header and some statistics to a file.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

9. `void A2_writeForMatlab ( A2 *mtx, char *mtxname, FILE *fp ) ;`

   This method writes the entries of the matrix to a file in Matlab format. The name of the matrix is `mtxname`.

   Error checking: If `mtx`, `mtxname` or `fp` are `NULL`, an error message is printed and zero is returned.

2.3 Driver programs for the `A2 object`

1. `test_norms msglvl msgFile type nrow ncol inc1 inc2 seed`

   This driver program tests the `A2` norm methods. Use the script file `do_norms` for testing. When the output file is loaded into matlab, the last two lines contain matrices whose entries should all be around machine epsilon.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `nrow` parameter is the number of rows.
   - The `ncol` parameter is the number of rows.
   - The `inc1` parameter is the row increment.
   - The `inc2` parameter is the column increment.

- The `seed` parameter is a random number seed.

2. `test_QR msglvl msgFile type nrow ncol inc1 inc2 seed`

   This driver program tests the `A2_QRreduce()` and `A2_QRreduce2()` methods which reduce A to QR via rank-1 and rank-2 updates. Use the script file `do_QR` for testing. When `msglvl > 1`, the matrix A and matrices $R1$ and $R2$ (computed from `A2_QRreduce()` and `A2_QRreduce2()`, respectively) are printed to the message file. When the output file is loaded into matlab, the errors $A^T A - R_1^T R_1$ and $A^T A - R_2^T R_2$ (if A is real) or the errors $A^H A - R_1^H R_1$ and $A^H A - R_2^H R_2$ (if A is complex) are computed.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `nrow` parameter is the number of rows.
   - The `ncol` parameter is the number of rows.
   - The `inc1` parameter is the row increment.
   - The `inc2` parameter is the column increment.
   - The `seed` parameter is a random number seed.

Chapter 3

Coords: Coordinates Object

The Coords object is used to hold (x, y), (x, y, z) or larger dimensional coordinates. We use it to visualize two- and three-dimensional graphs.

3.1 Data Structure

The Coords object has four fields.

- `int type` : coordinate type. When `type = 1`, coordinates are stored by tuples, $(x_0, y_0, \dots)$ first, $(x_1, y_1, \dots)$ next, etc. When `type = 2`, coordinates are stored by x-coordinates first, y-coordinates next, etc.
- `int ndim` : number of dimensions for the coordinates, e.g., for (x, y) coordinates `ndim = 2`, for (x, y, z) coordinates `ndim = 3`.
- `int ncoor` : number of coordinates (i.e., number of grid points).
- `float *coors` : pointer to a `float` vector that holds the coordinates

A correctly initialized and nontrivial Coords object will have `type` be `1` or `2`, positive `ndim` and `ncoor` values, and a non-`NULL` `coors` field.

3.2 Prototypes and descriptions of Coords methods

This section contains brief descriptions including prototypes of all methods that belong to the Coords object.

3.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Coords * Coords_new ( void ) ;`

   This method simply allocates storage for the Coords structure and then sets the default fields by a call to `Coords_setDefaultFields()`.

2. `void Coords_setDefaultFields ( Coords *coords ) ;`

   This method sets the structure's fields are set to default values: `type = COORDS_BY_TUPLE`, `ndim = ncoor = 0` and `coors = NULL`.

   Error checking: If `coords` is `NULL`, an error message is printed and the program exits.

3. `void Coords_clearData ( Coords *coords ) ;`

   This method clears data and releases any storage allocated by the object. If `coords->coors` is not `NULL`, then `FVfree(coords->coors)` is called to free the `float` vector. It then sets the structure's default fields with a call to `Coords_setDefaultFields()`.

   Error checking: If `coords` is `NULL`, an error message is printed and the program exits.

4. `void Coords_free ( Coords *coords ) ;`

   This method releases any storage by a call to `Coords_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `coords` is `NULL`, an error message is printed and the program exits.

3.2.2 Initializer methods

1. `void Coords_init ( Coords *coords, int type, int ndim, int ncoor ) ;`

   This method initializes a `Coords` object given the type, number of dimensions and number of grid points. It clears any previous data with a call to `Coords_clearData()`. The `float` vector is initialized by a call to `FVinit()`.

   Error checking: If `coords` is `NULL` or `type` is not `COORDS_BY_TUPLE` or `COORDS_BY_COORD`, or if either `ndim` or `ncoor` are nonpositive, an error message is printed and the program exits.

2. ```
   void Coords_init9P ( Coords *coords, float bbox[], int type,
                        int n1, int n2, int ncomp ) ;
   ```

   This method initializes a `Coords` object for a 9-point operator on a `n1` $\times$ `n2` grid with `ncomp` degrees of freedom at a grid point. The grid's location is given by the bounding box vector, `bbox[0]` $=$ x-coordinate of the southwest point, `bbox[1]` $=$ y-coordinate of the southwest point, `bbox[2]` $=$ x-coordinate of the northeast point, and `bbox[3]` $=$ y-coordinate of the northeast point.

   Error checking: If `coords bbox` is `NULL`, or if `type` is not `COORDS_BY_TUPLE` or `COORDS_BY_COORD`, or if any of `n1`, `n2` or `ncomp` are nonpositive, an error message is printed and the program exits.

3. ```
   void Coords_init27P ( Coords *coords, float bbox[], int type,
                         int n1, int n2, int n3, int ncomp ) ;
   ```

   This method initializes a `Coords` object for a 27-point operator on a `n1` $\times$ `n2` $\times$ `n3` grid with `ncomp` degrees of freedom at a grid point. The grid's location is given by the bounding box vector, `bbox[0]` $=$ x-coordinate of the southwest point, `bbox[1]` $=$ y-coordinate of the southwest point, `bbox[2]` $=$ z-coordinate of the southwest point, `bbox[3]` $=$ x-coordinate of the northeast point, `bbox[4]` $=$ y-coordinate of the northeast point, and `bbox[5]` $=$ z-coordinate of the northeast point.

   Error checking: If `coords bbox` is `NULL`, or if `type` is not `COORDS_BY_TUPLE` or `COORDS_BY_COORD`, or if any of `n1`, `n2`, `n3` or `ncomp` are nonpositive, an error message is printed and the program exits.

3.2.3 Utility methods

There are three utility methods.

1. `int Coords_sizeOf ( Coords *coords ) ;`

   This method returns the number of bytes that the object occupies.

   Error checking: If `coords` is `NULL`, an error message is printed and the program exits.

2. `float Coords_min ( Coords *coords, int dim )`

   This method returns the minimum coordinate value for the `dim`'th entry in the coordinates. For example, `Coords_min(coords, 1)` is the minimum x-value and `Coords_min(coords, 2)` is the minimum y-value.

   Error checking: If `coords` is `NULL`, or if `idim` does not lie in the range `[1,ndim]`, an error message is printed and the program exits.

3. `float Coords_max ( Coords *coords, int dim )`

   This method returns the maximum coordinate value for the `dim`'th entry in the coordinates. For example, `Coords_max(coords, 1)` is the maximum x-value and `Coords_max(coords, 2)` is the maximum y-value.

   Error checking: If `coords` is `NULL`, or if `idim` does not lie in the range `[1,ndim]`, an error message is printed and the program exits.

4. `float Coords_value ( Coords *coords, int idim, int icoor ) ;`

   This method returns the `float` value of the `idim`-th coordinate of the `icoor`-th grid point. For example, `Coords_value(coords, 1, 27)` returns x_{27}, `Coords_value(coords, 2, 16)` returns y_{16}, and `Coords_value(coords, 3, 118)` returns z_{118}.

   Error checking: If `coords` is `NULL`, or if `idim` does not lie in the range `[1,ndim]`, or if `icoor` does not lie in the range `[0,ncoor)`, an error message is printed and the program exits.

5. `void Coords_setValue ( Coords *coords, int idim, int icoor, float val ) ;`

   This method sets the `float` value of the `idim`-th coordinate of the `icoor`-th grid point. For example, `Coords_setValue(coords, 1, 27, 1.2)` sets $x_{27} = 1.2$, `Coords_setValue(coords, 2, 16, 3.3)` sets $y_{16} = 3.3$, and `Coords_setValue(coords, 3, 118, 0)` sets $z_{118} = 0$.

   Error checking: If `coords` is `NULL`, or if `idim` does not lie in the range `[1,ndim]`, or if `icoor` does not lie in the range `[0,ncoor)`, an error message is printed and the program exits.

3.2.4 IO methods

There are the usual eight IO routines. The file structure of a `Coords` object is simple: `type`, `ndim`, `ncoor` followed by the `coors[]` vector.

1. `int Coords_readFromFile ( Coords *coords, char *filename ) ;`

   This method read a `Coords` object from a file. It tries to open the file and if it is successful, it then calls `Coords_readFromFormattedFile()` or `Coords_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `coords` or `filename` is `NULL`, or if `filename` is not of the form `*.coordsf` (for a formatted file) or `*.coordsb` (for a binary file), an error message is printed and the method returns zero.

2. `int Coords_readFromFormattedFile ( Coords *coords, FILE *fp ) ;`

   This method reads in a `Coords` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `coords` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int Coords_readFromBinaryFile ( Coords *coords, FILE *fp ) ;`

   This method reads in a `Coords` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `coords` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int Coords_writeToFile ( Coords *coords, char *fn ) ;`

   This method write a `Coords` object to a file. The method tries to open the file and if it is successful, it then calls `Coords_writeFromFormattedFile()` or `Coords_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `coords` or `fn` is `NULL`, or if `fn` is not of the form `*.coordsf` (for a formatted file) or `*.coordsb` (for a binary file), an error message is printed and the method returns zero.

5. `int Coords_writeToFormattedFile ( Coords *coords, FILE *fp ) ;`

   This method writes a `Coords` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `coords` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int Coords_writeToBinaryFile ( Coords *coords, FILE *fp ) ;`

   This method writes a `Coords` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `coords` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int Coords_writeForHumanEye ( Coords *coords, FILE *fp ) ;`

   This method write the `Coords` object to a file in an easy to read fashion. The method `Coords_writeStats()` is called to write out the header and statistics. The `coors[]` vector is then printed out. The value `1` is returned.

   Error checking: If `coords` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int Coords_writeStats ( Coords *coords, FILE *fp ) ;`

   The header and statistics are written. The value `1` is returned.

   Error checking: If `coords` or `fp` are `NULL` an error message is printed and zero is returned.

3.3 Driver programs for the `Coords` object

This section contains brief descriptions of the driver programs.

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads and write `Coords` files, useful for converting formatted files to binary files and vice versa. One can also read in a `Coords` file and print out just the header information (see the `Coords_writeStats()` method).

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Coords` object is written to the message file.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inFile` parameter is the input file for the `Coords` object. It must be of the form `*.coordsf` or `*.coordsb`. The `Coords` object is read from the file via the `Coords_readFromFile()` method.
- The `outFile` parameter is the output file for the `Coords` object. If `outFile` is `none` then the `Coords` object is not written to a file. Otherwise, the `Coords_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.coordsf`), or a binary file (if `outFile` is of the form `*.coordsb`).

2. `mk9PCoords msglvl msgFile n1 n2 outCoordsFile`

   This driver program creates a `Coords` object for 9-point finite difference operator on a `n1` $\times$ `n2` grid and optionally writes it to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - The `outCoordsFile` parameter is the output file for the `Coords` object. If `outCoordsFile` is `none` then the `Coords` object is not written to a file. Otherwise, the `Coords_writeToFile()` method is called to write the object to a formatted file (if `outCoordsFile` is of the form `*.coordsf`), or a binary file (if `outCoordsFile` is of the form `*.coordsb`).

Chapter 4

DV: Double Vector Object

The DV object is a wrapper around a double vector, thus the acronym **D**ouble **V**ector. The driving force for its creation is more convenience than performance. There are three cases that led to its development.

- Often a method will create a vector (allocate storage for and fill the entries) whose size is not known before the method call. Instead of having a pointer to int and a pointer to double* in the calling sequence, we can return a pointer to an DV object that contains the newly created vector and its size.
- In many cases we need a persistent double vector object, and file IO is simplified if we have an object to deal with. The filename is of the form *.dvf for a formatted file or *.dvb for a binary file.
- Prototyping can go much faster with this object as opposed to working with an double array. Consider the case when one wants to accumulate a list of doubles, but one doesn't know how large the list will be. The method DV_setSize() can be used to set the size of the vector to zero. Another method DV_push() appends an element to the vector, growing the storage if necessary.
- Sometimes an object needs to change its size, i.e., vectors need to grow or shrink. It is easier and more robust to tell an DV object to resize itself (see the DV_setSize() and DV_setMaxsize() methods) than it is to duplicate code to work on an double vector.

One must choose where to use this object. There is a substantial performance penalty for doing the simplest operations, and so when we need to manipulate an double vector inside a loop, we extract out the size and pointer to the base array from the DV object. On the other hand, the convenience makes it a widely used object.

4.1 Data Structure

The DV structure has three fields.

- int size : present size of the vector.
- int maxsize : maximum size of the vector.
- int owned : owner flag for the data. When owned = 1, storage for owned double's has been allocated by this object and can be free'd by the object. When owned == 0 but size > 0 , this object points to entries that have been allocated elsewhere, and these entries will not be free'd by this object.
- double *vec : pointer to the base address of the *double* vector

The size, maxsize, nowned and vec fields need never be accessed directly — see the DV_size(), DV_maxsize(), DV_owned(), DV_entries(), DV_sizeAndEntries() methods.

4.2 Prototypes and descriptions of DV methods

This section contains brief descriptions including prototypes of all methods that belong to the DV object.

4.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `DV * DV_new ( void ) ;`

   This method simply allocates storage for the DV structure and then sets the default fields by a call to `DV_setDefaultFields()`.

2. `void DV_setDefaultFields ( DV *dv ) ;`

   This method sets the default fields of the object, `size = maxsize = owned = 0` and `vec = NULL`.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

3. `void DV_clearData ( DV *dv ) ;`

   This method releases any data owned by the object. If `vec` is not `NULL` and `owned = 1`, then the storage for `vec` is free'd by a call to `DVfree()`. The structure's default fields are then set with a call to `DV_setDefaultFields()`.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

4. `void DV_free ( DV *dv ) ;`

   This method releases any storage by a call to `DV_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

4.2.2 Instance methods

These method allow access to information in the data fields without explicitly following pointers. There is overhead involved with these method due to the function call and error checking inside the methods.

1. `int DV_owned ( DV *dv ) ;`

   This method returns the value of `owned`. If `owned > 0`, then the object owns the data pointed to by `vec` and will free this data with a call to `DVfree()` when its data is cleared by a call to `DV_free()` or `DV_clearData()`.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

2. `int DV_size ( DV *dv ) ;`

   This method returns the value of `size`, the present size of the vector.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

3. `int DV_maxsize ( DV *dv ) ;`

   This method returns the value of `size`, the maximum size of the vector.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

4. `double DV_entry ( DV *dv, int loc ) ;`

   This method returns the value of the `loc`'th entry in the vector. If `loc < 0` or `loc >= size`, i.e., if the location is out of range, we return `0.0`. This design `feature` is handy when a list terminates with a `0.0` value.

   Error checking: If `dv` or `vec` is `NULL`, an error message is printed and the program exits.

5. `double * DV_entries ( DV *dv ) ;`

   This method returns `vec`, a pointer to the base address of the vector.

   Error checking: If `dv` is `NULL`, an error message is printed and the program exits.

6. `void DV_sizeAndEntries ( DV *dv, int *psize, double **pentries ) ;`

   This method fills `*psize` with the size of the vector and `**pentries` with the base address of the vector.

   Error checking: If `dv`, `psize` or `pentries` is `NULL`, an error message is printed and the program exits.

7. `void DV_setEntry ( DV *dv, int loc, double value ) ;`

   This method sets the `loc`'th entry of the vector to `value`.

   Error checking: If `dv` is `NULL` or `loc < 0`, an error message is printed and the program exits.

4.2.3 Initializer methods

There are three initializer methods.

1. `void DV_init ( DV *dv, int size, double *entries ) ;`

   This method initializes the object given a size for the vector and a possible pointer to the vectors' storage. Any previous data is cleared with a call to `DV_clearData()`. If `entries != NULL` then the `vec` field is set to `entries`, the `size` and `maxsize` fields are set to `size`, and `owned` is set to zero because the object does not own the entries. If `entries` is `NULL` and `size > 0` then a vector is allocated by the object, and the object owns this storage.

   Error checking: If `dv` is `NULL` or `size < 0`, an error message is printed and the program exits.

2. `void DV_init1 ( DV *dv, int size ) ;`

   This method initializes the object given a size size for the vector via a call to `DV_init()`.

   Error checking: Error checking is done with the call to $\widehat{\rm D}$V_init().

3. `void DV_init2 ( DV *dv, int size, int maxsize, int owned, double *vec ) ;`

   This is the total initialization method. The data is cleared with a call to `DV_clearData()`. If `vec` is `NULL`, the object is initialized via a call to `DV_init1()`. Otherwise, the objects remaining fields are set to the input parameters. and if `owned` is not `1`, the data is not owned, so the object cannot grow.

   Error checking: If `dv` is `NULL`, or if `size < 0`, or if `maxsize < size`, or if `owned` is not equal to `0` or `1`, of if `owned = 1` and `vec = NULL`, an error message is printed and the program exits.

4. `void DV_setMaxsize ( DV *dv, int newmaxsize ) ;`

   This method sets the maximum size of the vector. If `maxsize`, the present maximum size of the vector, is not equal to `newmaxsize`, then new storage is allocated. Only `size` entries of the old data are copied into the new storage, so if `size > newmaxsize` then data will be lost. The `size` field is set to the minimum of `size` and `newmaxsize`.

   Error checking: If `dv` is `NULL` or `newmaxsize < 0`, or if `0 < maxsize` and `owned == 0`, an error message is printed and the program exits.

5. `void DV_setSize ( DV *dv, int newsize ) ;`

   This method sets the size of the vector. If `newsize > maxsize`, the length of the vector is increased with a call to `DV_setMaxsize()`. The `size` field is set to `newsize`.

   Error checking: If `dv` is `NULL`, or `newsize < 0`, or if `0 < maxsize < newsize` and `owned = 0`, an error message is printed and the program exits.

4.2.4 Utility methods

1. `void DV_shiftBase ( DV *dv, int offset ) ;`

   This method shifts the base entries of the vector and decrements the present size and maximum size of the vector by `offset`. This is a dangerous method to use because the state of the vector is lost, namely `vec`, the base of the entries, is corrupted. If the object owns its entries and `DV_free()`, `DV_setSize()` or `DV_setMaxsize()` is called before the base has been shifted back to its original position, a segmentation violation will likely result. This is a very useful method, but use with caution.

   Error checking: If `dv` is `NULL`, an error message is printed and the program exits.

2. `void DV_push ( DV *dv, double val ) ;`

   This method pushes an entry onto the vector. If the vector is full, i.e., if `size == maxsize - 1`, then the size of the vector is doubled if possible. If the storage cannot grow, i.e., if the object does not own its storage, an error message is printed and the program exits.

   Error checking: If `dv` is `NULL`, an error message is printed and the program exits.

3. `double DV_min ( DV *dv ) ;`
   `double DV_max ( DV *dv ) ;`
   `double DV_sun ( DV *dv ) ;`

   These methods simply return the minimum entry, the maximum entry and the sum of the entries in the vector.

   Error checking: If `dv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

4. `void DV_sortUp ( DV *dv ) ;`
   `void DV_sortDown ( DV *dv ) ;`

   This method sorts the entries in the vector into ascending or descending order via calls to `DVqsortUp()` and `DVqsortDown()`.

   Error checking: If `dv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

5. `void DV_ramp ( DV *dv, double base, int double ) ;`

   This method fills the object with a ramp vector, i.e., entry `i` is `base + i*incr`.

   Error checking: If `dv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

6. `void DV_shuffle ( DV *dv, int seed ) ;`

   This method shuffles the entries in the vector using `seed` as a seed to a random number generator.

   Error checking: If `dv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

7. `int DV_sizeOf ( DV *dv ) ;`

   This method returns the number of bytes taken by the object.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

8. `double * DV_first ( DV *dv ) ;`
   `double * DV_next ( DV *dv, int *pd ) ;`

   These two methods are used as iterators, e.g.,

   ```
   for ( pd = DV_first(dv) ; pd != NULL ; pd = DV_next(dv, pd) ) {
      do something with entry *pd
   }
   ```

   Each method checks to see if `dv` or `pd` is `NULL`, if so an error message is printed and the program exits. In method `DV_next()`, if `pd` is not in the valid range, an error message is printed and the program exits.

   Error checking: If `dv` is `NULL` an error message is printed and the program exits.

9. `void DV_fill ( DV *dv, double value ) ;`

   This method fills the vector with a scalar value.

   Error checking: If `dv` is `NULL`, an error message is printed and the program exits.

10. `void DV_zero ( DV *dv ) ;`

    This method fills the vector with zeros.

    Error checking: If `dv` is `NULL`, an error message is printed and the program exits.

11. `void DV_copy ( DV *dv1, DV *dv2 ) ;`

    This method fills the `dv1` object with entries in the `iv2` object. Note, this is a *mapped* copy, `dv1` and `dv2` need not have the same size. The number of entries that are copied is the smaller of the two sizes.

    Error checking: If `dv1` or `dv2` is `NULL`, an error message is printed and the program exits.

12. ```
    void DV_log10profile ( DV *dv, int npts, DV *xDV, DV *yDV, double tausmall,
                           double taubig, int *pnzero, int *pnsmall, int *pnbig ) ;
    ```

    This method scans the entries in the `DV` object and fills `xDV` and `yDV` with data that allows a simple $\log_{10}$ distribution plot. Only entries whose magnitudes lie in the range `[tausmall, taubig]` contribute to the distribution. The number of entries whose magnitudes are zero, smaller than `tausmall`, or larger than `taubig` are placed into `pnzero`, `*pnsmall` and `*pnbig`, respectively. On return, the size of the `xDV` and `yDV` objects is `npts`.

    Error checking: If `dv`, `xDV`, `yDV`, `pnsmall` or `pnbig` are `NULL`, or if `npts` ≤ 0, or if `taubig` < 0.0 or if `tausmall` $>$ `taubig`, an error message is printed and the program exits.

4.2.5 IO methods

There are the usual eight IO routines. The file structure of a `DV` object is simple: the first entry is `size`, followed by the `size` entries found in `vec[]`.

1. `int DV_readFromFile ( DV *dv, char *fn ) ;`

   This method reads a `DV` object from a file. It tries to open the file and if it is successful, it then calls `DV_readFromFormattedFile()` or `DV_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.
    ```

Error checking: If dv or fn are NULL, or if fn is not of the form *.dvf (for a formatted file) or *.dvb (for a binary file), an error message is printed and the method returns zero.

2. `int DV_readFromFormattedFile ( DV *dv, FILE *fp ) ;`

   This method reads in a DV object from a formatted file. If there are no errors in reading the data, the value 1 is returned. If an IO error is encountered from fscanf, zero is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

3. `int DV_readFromBinaryFile ( DV *dv, FILE *fp ) ;`

   This method reads in a DV object from a binary file. If there are no errors in reading the data, the value 1 is returned. If an IO error is encountered from fread, zero is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

4. `int DV_writeToFile ( DV *dv, char *fn ) ;`

   This method writes a DV object from a file. It tries to open the file and if it is successful, it then calls DV_writeFromFormattedFile() or DV_writeFromBinaryFile(), closes the file and returns the value returned from the called routine.

   Error checking: If dv or fn are NULL, or if fn is not of the form *.dvf (for a formatted file) or *.dvb (for a binary file), an error message is printed and the method returns zero.

5. `int DV_writeToFormattedFile ( DV *dv, FILE *fp ) ;`

   This method writes a DV object to a formatted file. If there are no errors in writing the data, the value 1 is returned. If an IO error is encountered from fprintf, zero is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

6. `int DV_writeToBinaryFile ( DV *dv, FILE *fp ) ;`

   This method writes a DV object to a binary file. If there are no errors in writing the data, the value 1 is returned. If an IO error is encountered from fwrite, zero is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

7. `int DV_writeForHumanEye ( DV *dv, FILE *fp ) ;`

   This method writes a DV object to a file in a human readable format. is called to write out the header and statistics. The entries of the vector then follow in eighty column format using the DVfprintf() method. The value 1 is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

8. `int DV_writeStats ( DV *dv, FILE *fp ) ;`

   This method writes the header and statistics to a file. The value 1 is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

9. `int DV_writeForMatlab ( DV *dv, char *name, FILE *fp ) ;`

   This method writes the entries of the vector to a file suitable to be read by Matlab. The character string name is the name of the vector, e.g, if name = "A", then we have lines of the form

   ```
   A(1) = 1.000000000000e0 ;
   A(2) = 2.000000000000e0 ;
   ...
   ```

   for each entry in the vector. Note, the output indexing is 1-based, not 0-based. The value 1 is returned.

   Error checking: If dv or fp are NULL, an error message is printed and zero is returned.

4.3 Driver programs for the `DV object`

1. `testIO msglvl msgFile inFile outFile`

   This driver program tests the `DV` IO methods, and is useful for translating between the formatted `*.dvf` and binary `*.dvb` files.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the name of the file from which to read in the object. `inFile` must be of the form `*.dvf` for a formatted file or `*.dvb` for a binary file.
   - The `outFile` parameter is the name of the file to which to write out the object. If `outfile` is of the form `*.dvf`, the object is written to a formatted file. If `outfile` is of the form `*.dvb`, the object is written to a binary file. When `outFile` is *not* `"none"`, the object is written to the file in a human readable format. When `outFile` is `"none"`, the object is not written out.

Chapter 5

`Drand`: Simple Random Number Generator

Finding the same random number generator on a variety of UNIX systems is not guaranteed to be a success. Therefore, we wrote a simple random number generator object taken from [2]. The `Drand` object provides both normally distributed and uniformly distributed random numbers.

5.1 Data Structure

The `Drand` object has nine fields.

- `double seed1` : first seed
- `double seed2` : second seed
- `double base1` : first base
- `double base2` : second base
- `double lower` : lower bound for a uniform distribution
- `double upper` : upper bound for a uniform distribution
- `double mean` : mean for a normal distribution
- `double sigma` : variation for a normal distribution
- `int mode`: mode of the object, uniform is `1`, normal is `2`

5.2 Prototypes and descriptions of `Drand` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Drand` object.

5.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Drand * Drand_new ( void ) ;`

   This method simply allocates storage for the `Drand` structure and then sets the default fields by a call to `Drand_setDefaultFields()`.

2. `void Drand_setDefaultFields ( Drand *drand ) ;`

   This method sets the structure's fields to default values.

   ```
   drand->seed1 =  123456789.0 ; drand->seed2 =  987654321.0 ;
   drand->base1 = 2147483563.0 ; drand->base2 = 2147483399.0 ;
   drand->lower =          0.0 ; drand->upper =          1.0 ;
   drand->mean  =          0.0 ; drand->sigma =          1.0 ;
   drand->mode  =            1 ;
   ```

   The default mode is a uniform distribution on `[0,1]`.

   Error checking: If `drand` is `NULL`, an error message is printed and the program exits.

3. `void Drand_clearData ( Drand *drand ) ;`

   This method clears any data owned by the object. It then sets the default fields with a call to `Drand_setDefaultFields()`.

   Error checking: If `drand` is `NULL`, an error message is printed and the program exits.

4. `void Drand_free ( Drand *drand ) ;`

   This method frees the object. It releases any storage by a call to `Drand_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `drand` is `NULL`, an error message is printed and the program exits.

5.2.2 Initializer methods

1. `void Drand_init ( Drand *drand ) ;`

   This initializer simply sets the default fields with a call to `Drand_setDefaultFields()`.

   Error checking: If `drand` is `NULL`, an error message is printed and the program exits.

2. `void Drand_setSeed ( Drand *drand, int seed1 ) ;`

   This method sets the random number seeds using a single input seed.

   Error checking: If `drand` is `NULL`, or if `seed1` ≤ 0, or if `seed1` ≥ 2147483563, an error message is printed and the program exits.

3. `void Drand_setSeeds ( Drand *drand, int seed1, int seed2 ) ;`

   This method sets the random number seeds using two input seeds.

   Error checking: If `drand` is `NULL`, an error message is printed and the program exits.

   Error checking: If `drand` is `NULL`, or if `seed1` ≤ 0, or if `seed1` ≥ 2147483563, or if `seed2` ≤ 0, or if `seed2` ≥ 2147483399, an error message is printed and the program exits.

4. `void Drand_setNormal ( Drand *drand, double mean, double sigma ) ;`

   This method sets the mode to be a normal distribution with mean `mean` and variation `sigma`.

   Error checking: If `drand` is `NULL`, or if `sigma` ≤ 0, an error message is printed and the program exits.

5. `void Drand_setUniform ( Drand *drand, double lower, double upper ) ;`

   This method sets the mode to be a uniform distribution over the interval `[lower,upper]` ;

   Error checking: If `drand` is `NULL`, or if `lower` $\geq$ `upper`, an error message is printed and the program exits.

5.2.3 Utility methods

1. `double Drand_value ( Drand *drand ) ;`

   This method returns a double precision random number.

   Error checking: If `drand` is `NULL`, an error message is printed and the program exits.

2. `void Drand_fillZvector ( Drand *drand, int n, double vec[] ) ;`

   This method fills a double precision complex vector `vec[]` with `n` complex random numbers.

   Error checking: If `drand` or `vec` are `NULL` or if `n < 0` , an error message is printed and the program exits.

3. `void Drand_fillDvector ( Drand *drand, int n, double vec[] ) ;`

   This method fills double precision vector `vec[]` with `n` random numbers.

   Error checking: If `drand` or `vec` are `NULL` or if `n < 0` , an error message is printed and the program exits.

4. `void Drand_fillIvector ( Drand *drand, int n, int vec[] ) ;`

   This method fills `vec[]` with `n` `int` random numbers.

   Error checking: If `drand` or `vec` are `NULL` or if `n < 0` , an error message is printed and the program exits.

5.3 Driver programs for the `Drand` object

This section contains brief descriptions of the driver programs.

1. `testDrand msglvl msgFile distribution param1 param2 seed1 seed2 n`

   This driver program test the `Drand` random number generator.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `distribution` parameter specifies the mode of the object. If `1`, the distribution is uniform. If `2`, the distribution is normal.
   - When `distribution = 1`, `param1` is the lower bound for the interval. When `distribution = 2`, `param1` is the mean for the normal distribution.
   - When `distribution = 1`, `param2` is the upper bound for the interval. When `distribution = 2`, `param2` is the variance for the normal distribution.

- `seed1` is the first random number seed.
- `seed2` is the second random number seed.
- `n` is the length of the vector of random numbers to be generated.

Chapter 6

I2Ohash: Two Key Hash Table

The I2Ohash is a object that manages a hash table where there are two integer keys and the data to be stored is void * pointer. This object was created to support a block sparse matrix, where each block has two keys, a row and column id, and the value is a pointer to a SubMtx object.

This is a very simple implementation. Each <key1,key2> is mapped to a list. Each list contains <key1,key2,value> triples whose keys are mapped to the list, and the triples are in lexicographic order of their <key1,key2> fields. The size of the hash table (the number of lists) is fixed upon initialization. The number of allowable <key1,key2,value> triples can either be fixed (upon initialization) or can grow by a user supplied amount.

The methods that are supported are

- insert a <key1,key2,value> triple
- locate a <key1,key2,*> triple and return the value
- remove a <key1,key2,*> triple and return the value

6.1 Data Structure

The I2Ohash object has a number of lists that contain <key1,key2,value> triples. Each triple is stored in an I2OP object, a simple structure found in the Utilities directory that holds two integer *key* fields, a void * *data* field, and a single *pointer* field to allow us to use it in singly linked lists.

The I2Ohash object has six fields.

- int nlist : number of lists in the hash table
- int grow : when no I2OP objects are available to insert a new <key1,key2,value> triple, the object can allocate grow more I2OP objects and put them on the free list.
- nitem : number of items in the hash table.
- I2OP *baseI2OP : pointer to an I2OP object that keeps track of all the I2OP objects that have been allocated by the hash table.
- I2OP *freeI2OP : pointer to the first I2OP object on the free list.
- I2OP **heads : pointer to a vector of pointers to I2OP objects, used to hold a pointer to the first I2OP object in each list.

A correctly initialized and nontrivial `I2Ohash` object will have `nlist > 0`. If `grow` is zero and a new `<key1,key2,value>` triple is given to the hash table to be inserted, a fatal error occurs.

6.2 Prototypes and descriptions of I2Ohash methods

This section contains brief descriptions including prototypes of all methods that belong to the `I2Ohash` object.

6.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `I2Ohash * I2Ohash_new ( void ) ;`

   This method simply allocates storage for the `I2Ohash` structure and then sets the default fields by a call to `I2Ohash_setDefaultFields()`.

2. `void I2Ohash_setDefaultFields ( I2Ohash *hashtable ) ;`

   This method sets the structure's fields to default values: `nlist`, `grow` and `nitem` are zero, `baseI2OP`, `freeI2OP` and `heads` are `NULL`.

   Error checking: If `hashtable` is `NULL`, an error message is printed and the program exits.

3. `void I2Ohash_clearData ( I2Ohash *hashtable ) ;`

   This method clears any data owned by the object. It releases any `I2OP` objects that have been allocated by the hash table, and then free's the `heads[]` vector. It then sets the structure's default fields with a call to `I2Ohash_setDefaultFields()`.

   Error checking: If `hashtable` is `NULL`, an error message is printed and the program exits.

4. `void I2Ohash_free ( I2Ohash *hashtable ) ;`

   This method releases any storage by a call to `I2Ohash_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `hashtable` is `NULL`, an error message is printed and the program exits.

6.2.2 Initializer methods

There is one initializer method.

1. `void I2Ohash_init ( I2Ohash *hashtable, int nlist, int nobj, int grow ) ;`

   This method is the basic initializer method. It clears any previous data with a call to `I2Ohash_clearData()`. It allocates storage for `nlist` lists and if `nobj` is positive, it loads the free list with `nobj` `I2OP` objects.

   Error checking: If `hashtable` is `NULL`, or if $\texttt{nlist} \leq 0$, or if `nobj` and `grow` are both zero, an error message is printed and the program exits.

6.2.3 Utility methods

1. `void I2Ohash_insert ( I2Ohash *hashtable, int key1, int key2, void * value ) ;`

   This method inserts the triple `(key1,key2,value)` into the hash table.

   Error checking: If `hashtable` is `NULL`, an error message is printed and the program exits.

2. `int I2Ohash_locate ( I2Ohash *hashtable, int key1, int key2, void **pvalue ) ;`

   If there is a `<key1,key2,value>` triple in the hash table, `*pvalue` is set to the value, and `1` is returned. If there is no `<key1,key2,value>` triple in the hash table, `0` is returned.

   Error checking: If `hashtable` or `pvalue` is `NULL`, an error message is printed and the program exits.

3. `int I2Ohash_remove ( I2Ohash *hashtable, int key1, int key2, void **pvalue ) ;`

   If there is a `<key1,key2,value>` triple in the hash table, `*pvalue` is set to the value, the triple is removed and its `I2OP` structure is placed on the free list, and `1` is returned. If there is no `<key1,key2,value>` triple in the hash table, `0` is returned.

   Error checking: If `hashtable` or `pvalue` is `NULL`, an error message is printed and the program exits.

4. `double I2Ohash_measure ( I2Ohash *hashtable ) ;`

   This method returns a floating point number that is some measure of how even a distribution of the `<key1,key2,value>` triples are made to the lists. We measure the imbalance using $\sqrt{\sum_i \texttt{count}_i^2}$, where i ranges over the lists and $\texttt{count}_i$ is the number of triples in list i. If the triples were perfectly evenly distributed, then each list would have `nitem/nlist` triples, and this value is $\texttt{nitem}/\sqrt{\texttt{nlist}}$. We return the ratio of $\sqrt{\sum_i \texttt{count}_i^2}$ over $\texttt{nitem}/\sqrt{\texttt{nlist}}$. A value of 1.0 means that the triples are perfectly distributed. A value of $\sqrt{\texttt{nlist}}$ means that the triples are distributed in the worst possible way (all are found in one list). In general, if the triples are evenly distributed among `nlist/k` lists, the value is $\sqrt{\texttt{k}}$.

   Error checking: If `hashtable` is `NULL`, an error message is printed and the program exits.

6.2.4 IO methods

1. `void I2Ohash_writeForHumanEye ( I2Ohash *hashtable, FILE *fp ) ;`

   This method prints the hash table in a human-readable format.

   Error checking: If `hashtable` or `fp` is `NULL`, an error message is printed and the program exits.

6.3 Driver programs for the `I2Ohash object`

1. `test_hash msglvl msgFile size grow maxkey nent seed`

   This driver program tests the `I2Ohash` insert method. It inserts a number of triples into a hash table and prints out the "measure" of how well distributed the entries are in the hash table.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `size` parameter is the number of lists.
   - The `grow` parameter is how much the pool of `I2OP` objects can grow. Setting `grow` to zero is helpful when the number of items that can be placed into the hash table is known a priori. If one tries to insert an items when the free pool is empty and `grow` is zero, an error message is printed and the program exits.

- The `maxkey` parameter an upper bound on key value.
- The `nent` parameter is the number of `<key1, key2, pointer>` triples to insert.
- The `seed` parameter is a random number seed.

Chapter 7

IIheap: (Key, Value) Heap

The IIheap is a object that manages a heap of data. Both the *key* and the *value* are of type int. The heap has fixed size, each item must be in [0,maxsize-1], where maxsize is set on initialization. The IIheap object requires three vectors of size maxsize. Three methods are supported: *find_min*, *insert* and *delete* which take $O(1)$, $O(\log_2 n)$ and $O(\log_2 n)$ time, respectively, where n is the present size of the heap.

7.1 Data Structure

The IIheap object has five fields.

- int size : present size of the heap, 0 <= size < maxsize
- int maxsize : maximum size of the heap, set on initialization
- int *heapLoc : pointer to an int vector of size maxsize, heapLoc[i] contains the location of item i, heapLoc[i] = -1 if item i is not in the heap
- int *keys : pointer to an int vector of size maxsize, keys[loc] contains the key at location loc
- int *values : pointer to an int vector of size maxsize, values[loc] contains the value at location loc

A correctly initialized and nontrivial IIheap object will have maxsize > 0 and 0 <= size < maxsize.

7.2 Prototypes and descriptions of IIheap methods

This section contains brief descriptions including prototypes of all methods that belong to the IIheap object.

7.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. IIheap * IIheap_new (void) ;

   This method simply allocates storage for the IIheap structure and then sets the default fields by a call to IIheap_setDefaultFields().

2. `void IIheap_setDefaultFields ( IIheap *heap ) ;`

   This method sets the structure's fields to default values: `size` and `maxsize` are zero, `heapLoc`, `keys` and `values` are `NULL`.

   Error checking: If `heap` is `NULL`, an error message is printed and the program exits.

3. `void IIheap_clearData ( IIheap *heap ) ;`

   This method clears any data owned by the object. It releases any storage held by the `heap->heapLoc`, `heap->keys` and `heap->values` vectors, then sets the structure's default fields with a call to `IIheap_setDefaultFields(`

   Error checking: If `heap` is `NULL`, an error message is printed and the program exits.

4. `void IIheap_free ( IIheap *heap ) ;`

   This method releases any storage by a call to `IIheap_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `heap` is `NULL`, an error message is printed and the program exits.

7.2.2 Initializer methods

There is one initializer method.

1. `void IIheap_init ( IIheap *heap, int maxsize ) ;`

   This method is the basic initializer method. It clears any previous data with a call to `IIheap_clearData()`, and allocates storage for the `heapLoc`, `keys` and `values` vectors using `IVinit()`. The entries in the three vectors are set to `-1`.

   Error checking: If `heap` is `NULL`, or if `maxsize` ≤ 0, an error message is printed and the program exits.

7.2.3 Utility methods

1. `int IIheap_sizeOf ( IIheap *heap ) ;`

   This method returns the number of bytes taken by this object.

   Error checking: If `heap` is `NULL`, an error message is printed and the program exits.

2. `void IIheap_root ( IIheap *heap, int *pkey, int *pvalue ) ;`

   This method fills `*pid` and `*pkey` with the key and value, respectively, of the root element, an element with minimum value. If `size == 0` then `-1` is returned.

   Error checking: If `heap`, `pkey` or `pvalue` is `NULL`, an error message is printed and the program exits.

3. `void IIheap_insert ( IIheap *heap, int key, int value ) ;`

   This method inserts the pair `(key,value)` into the heap.

   Error checking: If `heap` is `NULL`, of if `key` is out of range, or if `key` is already in the heap, or if the heap is full, an error message is printed and the program exits.

4. `void IIheap_remove ( IIheap *heap, int key ) ;`

   This method removes `(key,*)` from the heap.

   Error checking: If `heap` is `NULL`, of if `key` is out of range, or if `key` is not in the heap, an error message is printed and the program exits.

5. `void IIheap_print ( IIheap *heap, FILE *fp ) ;`

   This method prints the heap in a human-readable format.

   Error checking: If `heap` or `fp` is `NULL`, an error message is printed and the program exits.

Chapter 8

IV: Integer Vector Object

The IV object is a wrapper around an int vector, thus the acronym **I**nteger **V**ector. The driving force for its creation is more convenience than performance. There are several reasons that led to its development.

- Often a method will create a vector (allocate storage for and fill the entries) whose size is not known before the method call. Instead of having a pointer to int and a pointer to int* in the calling sequence, we can return a pointer to an IV object that contains the newly created vector and its size.
- In many cases we need a persistent int vector object, and file IO is simplified if we have an object to deal with. The filename is of the form *.ivf for a formatted file or *.ivb for a binary file. Another case is where an object contains one or more int vectors. When they are held as IV objects, (e.g., the ETree object contains a Tree object and three IV objects), the method to read and write the object is much cleaner.
- Prototyping can go much faster with this object as opposed to working with an int array. Consider the case when one wants to accumulate a list of integers, but one doesn't know how large the list will be. The method IV_setSize() can be used to set the size of the vector to zero. Another method IV_push() appends an element to the vector, growing the storage if necessary.
- Having the size of a vector and a pointer to the base location wrapped together makes it easier to check for valid input inside a method.
- Sometimes an object needs to change its size, i.e., vectors need to grow or shrink. It is easier and more robust to tell an IV object to resize itself (see the IV_setSize() and IV_setMaxsize() methods) than it is to duplicate code to work on an int vector.

One must choose where to use this object. There is a substantial performance penalty for doing the simplest operations, and so when we need to manipulate an int vector inside a loop, we extract out the size and pointer to the base array from the IV object. On the other hand, the convenience makes it a widely used object. Originally its use was restricted to reading and writing *.iv{f,b} files, but now IV objects appear much more frequently in new development.

8.1 Data Structure

The IV structure has four fields.

- int size : present size of the vector.

- `int maxsize` : maximum size of the vector.
- `int owned` : owner flag for the data. When `owned = 1`, storage for `maxsize int`'s has been allocated by this object and can be free'd by the object. When `nowned = 0` but `maxsize > 0`, this object points to entries that have been allocated elsewhere, and these entries will not be free'd by this object.
- `int *vec` : pointer to the base address of the *int* vector

The `size`, `maxsize`, `owned` and `vec` fields need never be accessed directly — see the `IV_size()`, `IV_maxsize()`, `IV_owned()`, `IV_entries()`, `IV_sizeAndEntries()` methods.

8.2 Prototypes and descriptions of IV methods

This section contains brief descriptions including prototypes of all methods that belong to the `IV` object.

8.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `IV * IV_new ( void ) ;`

   This method simply allocates storage for the `IV` structure and then sets the default fields by a call to `IV_setDefaultFields()`.

2. `void IV_setDefaultFields ( IV *iv ) ;`

   This method sets the structure's fields to default values: `size` = `maxsize` = `owned` = 0 and `vec` = `NULL` .

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

3. `void IV_clearData ( IV *iv ) ;`

   This method clears any data owned by the object. If `vec` is not `NULL` and `owned = 1`, then the storage for `vec` is free'd by a call to `IVfree()`. The structure's default fields are then set with a call to `IV_setDefaultFields()`.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

4. `void IV_free ( IV *iv ) ;`

   This method releases any storage by a call to `IV_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

8.2.2 Instance methods

These method allow access to information in the data fields without explicitly following pointers. There is overhead involved with these method due to the function call and error checking inside the methods.

1. `int IV_owned ( IV *iv ) ;`

   This method returns the value of `owned`. If `owned = 1`, then the object owns the data pointed to by `vec` and will free this data with a call to `IVfree()` when its data is cleared by a call to `IV_free()` or `IV_clearData()`.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

2. `int IV_size ( IV *iv ) ;`

   This method returns the value of `size`, the present size of the vector.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

3. `int IV_maxsize ( IV *iv ) ;`

   This method returns the value of `maxsize`, the maximum size of the vector.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

4. `int IV_entry ( IV *iv, int loc ) ;`

   This method returns the value of the `loc`'th entry in the vector. If `loc < 0` or `loc >= size`, i.e., if the location is out of range, we return `-1`. This design `feature` is handy when a list terminates with a `-1` value.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

5. `int * IV_entries ( IV *iv ) ;`

   This method returns `vec`, a pointer to the base address of the vector.

   Error checking: If `iv` is `NULL` an error message is printed and the program exits.

6. `void IV_sizeAndEntries ( IV *iv, int *psize, int **pentries ) ;`

   This method fills `*psize` with the size of the vector and `*pentries` with the base address of the vector.

   Error checking: If `iv`, `psize` or `pentries` is `NULL` an error message is printed and the program exits.

7. `void IV_setEntry ( IV *iv, int loc, int value ) ;`

   This method sets the `loc`'th entry of the vector to `value`.

   Error checking: If `iv`, `loc < 0` or `loc >= size`, or if `vec` is `NULL` an error message is printed and the program exits.

8.2.3 Initializer methods

1. `void IV_init ( IV *iv, int size, int *entries ) ;`

   This method initializes the object given a size for the vector and a possible pointer to the vectors storage. Any previous data with a call to `IV_clearData()`. If `entries != NULL` then the `vec` field is set to `entries`, the `size` and `maxsize` fields are set to `size` , and `owned` is set to zero because the object does not own the entries. If `entries` is `NULL` and if `size > 0` then a vector is allocated by the object, and the object owns this storage.

   Error checking: If `iv` is `NULL` or `size < 0`, an error message is printed and the program exits.

2. `void IV_init1 ( IV *iv, int size ) ;`

   This method initializes the object given a size for the vector. Any previous data is cleared with a call to `IV_clearData()`. Then `size` and `maxsize` are set to `size`. If `size > 0`, then the vector is created via a call to `IVinit()` and `owned` is set to `1`.

   Error checking: If `iv` is `NULL` or `size < 0`, an error message is printed and the program exits.

3. `void IV_init2 ( IV *iv, int size, int maxsize, int owned, int *vec ) ;`

   This is the total initialization method. Any previous data is cleared with a call to `IV_clearData()`. If `vec` is `NULL`, the object is initialized via a call to `IV_init1()`. Otherwise, the object's remaining fields are set to the input parameters.

   Error checking: If `iv` is `NULL` or `maxsize < 0` or `size < 0`, or if `owned` is not equal to `0` or `1`, or if `owned = 0` and `vec == NULL`, an error message is printed and the program exits.

4. `void IV_setMaxsize ( IV *iv, int newmaxsize ) ;`

   This method sets the maximum size of the vector. If `maxsize`, the present maximum size, is not equal to `newmaxsize`, then new storage is allocated. Only `size` entries of the old data are copied into the new storage, so if `size > newmaxsize` then data will be lost. The `size` field is set to the minimum of `size` and `newmaxsize`.

   Error checking: If `iv` is `NULL` or `newmaxsize < 0`, or if `0 < maxsize` and `owned == 0`, an error message is printed and the program exits.

5. `void IV_setSize ( IV *iv, int newsize ) ;`

   This method sets the size of the vector. If `newsize > maxsize`, the length of the vector is increased with a call to `IV_setMaxsize()`. The `size` field is set to `newsize`.

   Error checking: If `iv` is `NULL` or `newsize < 0`, or if `0 < maxsize < newsize` and `owned == 0`, an error message is printed and the program exits.

8.2.4 Utility methods

1. `void IV_shiftBase ( IV *iv, int offset ) ;`

   This method shifts the base entries of the vector and decrements the present size and maximum size of the vector by `offset`. This is a dangerous method to use because the state of the vector is lost, namely `vec`, the base of the entries, is corrupted. If the object owns its entries and `IV_free()`, `IV_setSize()` or `IV_setMaxsize()` is called before the base has been shifted back to its original position, a segmentation violation will likely result. This is a very useful method, but use with caution.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

2. `void IV_push ( IV *iv, int val ) ;`

   This method pushes an entry onto the vector. If the vector is full, i.e., if `size = maxsize - 1`, then the size of the vector is doubled if possible. If the storage cannot grow, i.e., if the object does not own its storage, an error message is printed and the program exits.

   Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

3. `int IV_min ( IV *iv ) ;`
   `int IV_max ( IV *iv ) ;`

   These methods simply return the minimum and maximum entries in the vector.

   Error checking: If `iv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

4. `void IV_sortUp ( IV *iv ) ;`
   `void IV_sortDown ( IV *iv ) ;`

   This method sorts the entries in the vector into ascending or descending order via calls to `IVqsortUp()` and `IVqsortDown()`.

   Error checking: If `iv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

5. `void IV_ramp ( IV *iv, int base, int incr ) ;`

   This method fill the object with a ramp vector, i.e., entry `i` is `base + i*incr`.

   Error checking: If `iv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

6. `void IV_shuffle ( IV *iv, int seed ) ;`

   This method shuffles the entries in the vector using `seed` as a seed to a random number generator.

   Error checking: If `iv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

7. `int IV_sizeOf ( IV *iv ) ;`

   This method returns the number of bytes taken by the object.

   Error checking: If `iv` is `NULL` an error message is printed and the program exits.

8. `void IV_filterKeep ( IV *iv, int tags[], int keepTag ) ;`

   This method examines the entries in the vector. Let `k` be entry `i` in the vector. If `tags[k] != keepTag`, the entry is moved to the end of the vector, otherwise it is moved to the beginning of the vector. The size of the vector is reset to be the number of tagged entries that are now in the leading locations.

   Error checking: If `iv` of `tags` is `NULL` an error message is printed and the program exits.

9. `void IV_filterPurge ( IV *iv, int tags[], int purgeTag ) ;`

   This method examines the entries in the vector. Let `k` be entry `i` in the vector. If `tags[k] == purgeTag`, the entry is moved to the end of the vector, otherwise it is moved to the beginning of the vector. The size of the vector is reset to be the number of untagged entries that are now in the leading locations.

   Error checking: If `iv` of `tags` is `NULL` an error message is printed and the program exits.

10. `int * IV_first ( IV *iv ) ;`
    `int * IV_next ( IV *iv, int *pi ) ;`

    These two methods are used as iterators, e.g.,

    ```
    for ( pi = IV_first(iv) ; pi != NULL ; pi = IV_next(iv, pi) ) {
       do something with entry *pi
    }
    ```

    Error checking: Each method checks to see if `iv` or `pi` is `NULL`. If so an error message is printed and the program exits. In method `IV_first()`, if `pi` is not in the valid range, an error message is printed and the program exits.

11. `void IV_fill ( IV *iv, int value ) ;`

    This method fills the vector with a scalar value.

    Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

12. `void IV_copy ( IV *iv1, IV *iv2 ) ;`

    This method fills the `iv1` object with entries in the `iv2` object. Note, this is a *mapped* copy, `iv1` and `iv2` need not have the same size. The number of entries that are copied is the smaller of the two sizes.

    Error checking: If `iv1` or `iv2` is `NULL`, an error message is printed and the program exits.

13. `int IV_increment ( IV *iv, int loc ) ;`

    This method increments the `loc`'th location of the `iv` object by one and returns the new value.

    Error checking: If `iv` is `NULL` or if `loc` is out of range, an error message is printed and the program exits.

14. `int IV_decrement ( IV *iv, int loc ) ;`

    This method decrements the `loc`'th location of the `iv` object by one and returns the new value.

    Error checking: If `iv` is `NULL` or if `loc` is out of range, an error message is printed and the program exits.

15. `int IV_findValue ( IV *iv, int value ) ;`

    This method looks for `value` in its entries. If `value` is present, the first location is returned, otherwise `-1` is returned. The cost is linear in the number of entries.

    Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

16. `int IV_findValueAscending ( IV *iv, int value ) ;`

    This method looks for `value` in its entries. If `value` is present, a location is returned, otherwise `-1` is returned. This method assumes that the entries are sorted in ascending order. The cost is logarthmic in the number of entries.

    Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

17. `int IV_findValueDescending ( IV *iv, int value ) ;`

    This method looks for `value` in its entries. If `value` is present, a location is returned, otherwise `-1` is returned. This method assumes that the entries are sorted in descending order. The cost is logarthmic in the number of entries.

    Error checking: If `iv` is `NULL`, an error message is printed and the program exits.

18. `IV * IV_inverseMap ( IV *listIV ) ;`

    This method creates and returns an inverse map for a list of nonnegative integers. This function is used when `listIV` contains a list of global ids and we need a map from the global ids to their location in the list. The size of the returned `IV` object is equal to one plus the largest value found in `listIV`. If `value` is not found in `listIV`, then the corresponding entry in the returned `IV` object is -1.

    Error checking: If `listIV` is `NULL`, or if its size if zero or less or if its entries are `NULL`, or if one of its entries is negative, or if any entry in listIV is repeated, an error message is printed and the program exits.

19. `IV * IV_targetEntries ( IV *listIV ) ;`

    This method creates and returns a list of locations where `target` is found in `listIV` .

    Error checking: If `listIV` is `NULL`, or if its size if zero or less or if its entries are `NULL`, an error message is printed and the program exits.

8.2.5 IO methods

There are the usual eight IO routines. The file structure of an `IV` object is simple: the first entry is `size`, followed by the `size` entries found in `vec[]`.

1. `int IV_readFromFile ( IV *iv, char *fn ) ;`

   This method reads an `IV` object from a formatted file. It tries to open the file and if it is successful, it then calls `IV_readFromFormattedFile()` or `IV_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `iv` or `fn` are `NULL`, or if `fn` is not of the form `*.ivf` (for a formatted file) or `*.ivb` (for a binary file), an error message is printed and the method returns zero.

2. `int IV_readFromFormattedFile ( IV *iv, FILE *fp ) ;`

   This method reads in an `IV` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `iv` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int IV_readFromBinaryFile ( IV *iv, FILE *fp ) ;`

   This method reads in an `IV` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `iv` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int IV_writeToFile ( IV *iv, char *fn ) ;`

   This method writes an `IV` object to a formatted file. It tries to open the file and if it is successful, it then calls `IV_writeFromFormattedFile()` or `IV_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `iv` or `fn` are `NULL`, or if `fn` is not of the form `*.ivf` (for a formatted file) or `*.ivb` (for a binary file), an error message is printed and the method returns zero.

5. `int IV_writeToFormattedFile ( IV *iv, FILE *fp ) ;`

   This method writes an `IV` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `iv` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int IV_writeToBinaryFile ( IV *iv, FILE *fp ) ;`

   This method writes an `IV` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `iv` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int IV_writeForHumanEye ( IV *iv, FILE *fp ) ;`

   This method writes an `IV` object to a file in a human readable format. The method `IV_writeStats()` is called to write out the header and statistics. The entries of the vector then follow in eighty column format using the `IVfp80()` method. The value `1` is returned.

   Error checking: If `iv` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int IV_writeStats ( IV *iv, FILE *fp ) ;`

   This method writes the header and statistics to a file. The value `1` is returned.

   Error checking: If `iv` or `fp` are `NULL` an error message is printed and zero is returned.

9. `int IV_fp80 ( IV *iv, FILE *fp, int column, int *pierr ) ;`

   This method is just a wrapper around the `IVfp80()` method for an `int` method. The entries in the vector are found on lines with eighty columns and are separated by a whitespace. The value `1` is returned.

   Error checking: If `iv` or `fp` or `pierr` are `NULL`, an error message is printed and zero is returned.

10. `int IV_writeForMatlab ( IV *iv, char *name, FILE *fp ) ;`

    This method writes the entries of the vector to a file suitable to be read by Matlab. The character string `name` is the name of the vector, e.g, if `name = "A"`, then we have lines of the form

```
A(1) = 32 ;
A(2) = -433 ;
...
```

for each entry in the vector. Note, the output indexing is 1-based, not 0-based. The value `1` is returned.

Error checking: If `iv` or `fp` are `NULL`, an error message is printed and zero is returned.

8.3 Driver programs for the `IV object`

1. `testIO msglvl msgFile inFile outFile`

   This driver program tests the `IV` IO methods, and is useful for translating between the formatted `*.ivf` and binary `*.ivb` files.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the name of the file from which to read in the object. `inFile` must be of the form `*.ivf` for a formatted file or `*.ivb` for a binary file.
   - The `outFile` parameter is the name of the file to which to write out the object. If `outfile` is of the form `*.ivf`, the object is written to a formatted file. If `outfile` is of the form `*.ivb`, the object is written to a binary file. When `outFile` is *not* `"none"`, the object is written to the file in a human readable format. When `outFile` is `"none"`, the object is not written out.

Chapter 9

IVL: Integer Vector List Object

The IVL object is used to handle a list of int vectors, thus the acronym **I**nteger **V**ector **L**ist. The most common use is to represent a graph or the adjacency structure of a matrix. We have tried to make this object easy to use, and much hinges on the ability to create new lists or change the size of a list. In the interests of efficiency, this object is not a general purpose storage object, i.e., free'd data is not reused.

9.1 Data Structure

The IVL structure has seven fields.

- `int type` : storage type, one of IVL_CHUNKED, IVL_SOLO, IVL_UNKNOWN, and IVL_NOTYPE (which means the object has not yet been initialized.) Here is a description of the three types of storage management.
  - IVL_CHUNKED
    A chunk of data is allocated by the object and each list occupies contiguous entries in a chunk. More than one chunk may exist at one time, but only one contains free entries to be used for a new list. If there is not enough space in the chunk to contain the entries in a new list, another chunk is allocated to hold the list. When the IVL object is free'd, all the chunks of data are free'd. The number of entries in a chunk can be set by the user by changing the `incr` field, whose default value is 1024. This type of storage is used most often.
  - IVL_SOLO
    Each list is allocated separately using the `IVinit()` function. When the IVL object is free'd, each list is free'd separately using the `IVfree()` function.
  - IVL_UNKNOWN
    This storage mode is available for the cases where storage for a list is aliased to another location. Absolutely no free'ing of data is done when the IVL object is free'd.

  The storage management is handled by `IVL_setList()` and `IVL_setPointerToList()`.

- `int maxnlist` : maximum number of lists.

  `int nlist` : number of lists.

  We may not know how many lists we will need for the object — `maxnlist` is the dimension of the `sizes[]` and `p_vec[]` arrays and `nlist` is the present number of active lists. When we initialize the object using one of the `IVL_init{1,2,3}()` methods, we set `nlist` equal to `maxnlist`. We resize the object using `IVL_setMaxnlist()`.

- `int tsize` : total number of list entries.
- `int *sizes` : pointer to an `int` vector of size `maxnlist`.

  `int **p_vec` : pointer to an `int*` vector of size `maxnlist`.

  The size of list `ilist` is found in `sizes[ilist]` and `p_vec[ilist]` points to the start of the list. The `sizes` and `p_vec` fields need never be accessed directly — see the `IVL_listAndSize()`, `IVL_setList()` and `IVL_setPointerToList()` methods.
- `int incr` : increment for a new chunk of data, used for type `IVL_CHUNKED`
- `Ichunk *chunk` : pointer to the active `Ichunk` structure, a helper object to manage the allocated storage. It has the following fields.
  - `int size` : number of entries in the chunk, also the dimension of the array `base[]`.
  - `int inuse` : number of entries in use from this chunk, the number of available entries in `size - inuse`.
  - `int *base` : base address of the `int` vector of size `size` from which we find storage for the individual lists.
  - `Ichunk *next` : pointer to the next `Ichunk` object in the list of active `Ichunk` objects.

9.2 Prototypes and descriptions of IVL methods

This section contains brief descriptions including prototypes of all methods that belong to the `IVL` object.

9.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `IVL * IVL_new ( void ) ;`

   This method simply allocates storage for the `IVL` structure and then sets the default fields by a call to `IVL_setDefaultFields()`.
2. `void IVL_setDefaultFields ( IVL *ivl ) ;`

   This method sets the default fields of the object — `type = IVL_NOTYPE`, `maxnlist`, `nlist` and `tsize` are zero, `incr` is `1024`, and `sizes`, `p_vec` and `chunk` are `NULL`.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.
3. `void IVL_clearData ( IVL *ivl ) ;`

   This method clears any data allocated by this object and then sets the default fields with a call to `IVL_setDefaultFields()`. Any storage held by the `Ichunk` structures is free'd, and if `sizes` or `p_vec` are not `NULL`, they are free'd.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.
4. `void IVL_free ( IVL *ivl ) ;`

   This method releases any storage by a call to `IVL_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

9.2.2 Instance methods

1. `int IVL_type ( IVL *ivl ) ;`

   This method returns `type`, the storage type.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

2. `int IVL_maxnlist ( IVL *ivl ) ;`

   This method returns `maxnlist`, the maximum number of lists.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

3. `int IVL_nlist ( IVL *ivl ) ;`

   This method returns `nlist`, the present number of lists.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

4. `int IVL_tsize ( IVL *ivl ) ;`

   This method returns `tsize`, the present number of list entries.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

5. `int IVL_incr ( IVL *ivl ) ;`

   This method returns `incr`, the storage increment.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

6. `int IVL_setincr ( IVL *ivl, int incr ) ;`

   This method sets the storage increment to `incr`.

   Error checking: If `ivl` is `NULL` or `incr` is negative, an error message is printed and the program exits.

9.2.3 Initialization and resizing methods

1. `void IVL_init1 ( IVL *ivl, int type, int maxnlist ) ;`

   This method is used when only the number of lists is known. Any previous data is cleared with a call to `IVL_clearData()`. The `type` field is set. If `maxnlist > 0`, storage is allocated for the `sizes[]` and `p_vec[]` arrays and `nlist` is set to `maxnlist`.

   Error checking: If `ivl` is `NULL` or `type` is invalid or `maxnlist` is negative, an error message is printed and the program exits.

2. `void IVL_init2 ( IVL *ivl, int type, int nlist, int tsize ) ;`

   This method is used when the number of lists and their total size is known — `type` must be `IVL_CHUNKED`. The `IVL_init1()` initializer method is called. If `tsize > 0` an `Ichunk` object with `tsize` entries is allocated.

   Error checking: If `ivl` is `NULL` or `type` is not `IVL_CHUNKED` or if `nlist` or `tsize` are negative, an error message is printed and the program exits.

3. `void IVL_init3 ( IVL *ivl, int type, int nlist, int sizes[] ) ;`

   This method is used when the number of lists and the size of each list is known — `type` must be `IVL_CHUNKED` or `IVL_SOLO`. If `type` is `IVL_CHUNKED`, then `IVL_init2()` is called to initialize the object, else `type` is `IVL_SOLO` and `IVL_init1()` is called. The size and pointer for each list is then set using the value from the `sizes[]` array.

   Error checking: If `ivl` is `NULL`, or if `type` is not `IVL_CHUNKED` or `IVL_SOLO`, or if `nlist` is nonpositive, or if `sizes[]` is `NULL`, an error message is printed and the program exits.

4. `int IVL_initFromSubIVL ( IVL *subIVL, IVL *ivl, IV *keeplistIV, IV *keepentriesIV ) ;`

   This method initializes the `subIVL` object from the `ivl` object. The lists found in `keeplistIV` are placed into the `subIVL` object; if `keeplistIV` is `NULL`, all lists are included. The list entries found in `keepentriesIV` are placed into lists in the the `subIVL` object; if `keepentriesIV` is `NULL`, all entries are included.

   Return values: 1 is a normal return, -1 means `subIVL` is `NULL`, -2 means `ivl` is `NULL`, -3 means `keeplistIV` is invalid.

5. `void IVL_setMaxnlist ( IVL *ivl, int newmaxnlist ) ;`

   This method is used to resize the object by changing the maximum number of lists. If `newmaxnlist == maxnlist`, nothing is done. Otherwise, new storage for `sizes[]` and `p_vec[]` is allocated, the information for the first `nlist` lists is copied over, and the old storage free'd. Note, `maxnlist` is set to `newmaxnlist` and `nlist` is set to the minimum of `nlist` and `newmaxnlist`.

   Error checking: If `ivl` is `NULL` or if `newmaxnlist` is negative, an error message is printed and the program exits.

6. `void IVL_setNlist ( IVL *ivl, int newnlist ) ;`

   This method is used to change the number of lists. If `newnlist > maxnlist`, storage for the lists is increased via a call to the `IVL_setMaxnlist()` method. Then `nlist` is set to `newnlist`.

   Error checking: If `ivl` is `NULL`, or if `newnlist` is negative, an error message is printed and the program exits.

9.2.4 List manipulation methods

1. `void IVL_listAndSize ( IVL *ivl, int ilist, int *psize, int **pivec) ;`

   This method fills `*psize` with `sizes[ilist]` and `*pivec` with `p_vec[ilist]`.

   Error checking: If `ivl` is `NULL`, or if `ilist < 0` or `ilist >= nlist` or if `psize` or `pivec` is `NULL`, an error message is printed and the program exits.

2. `int * IVL_firstInList ( IVL *ivl, int ilist ) ;`
   `int * IVL_nextInList ( IVL *ivl, int ilist, int *pi ) ;`

   These two methods are used as iterators, e.g.,

   ```
   for ( pi = IVL_firstInList(ivl, ilist) ;
         pi != NULL ;
         pi = IVL_nextInList(ivl, ilist, pi) ) {
      do something with entry *pi
   }
   ```

   Error checking: Each method checks to see if `ivl` is `NULL` or `ilist < 0` or `ilist >= nlist`, if so an error message is printed and the program exits. In method `IVL_firstInList()`, if `sizes[ilist] > 0` and `p_vec[ilist] = NULL`, an error message is printed and the program exits. In method `IVL_nextInList()`, if `pi` is not in the valid range for list `ilist`, an error message is printed and the program exits.

3. `void IVL_setList ( IVL *ivl, int ilist, int isize, int ivec[] ) ;`

   This method sets the size and (possibly) pointer to a list of entries. The behavior of the method depends on the type of the `ivl` object. Here is the flow chart:

```
if ilist >= maxnlist then
      the number of lists is increased via a call to IVL_setMaxnlist()
endif
if ilist >= nlist then
      nlist is increased
endif
if isize = 0 then
      release the storage for that list, reclaim storage if possible
else if type is IVL_UNKNOWN then
      set the pointer
else
      if the present size of list ilist is smaller than isize then
            get new storage for a new larger list
      endif
      set the size
      if ivec is not NULL then
            copy the entries
      endif
endif
```

Error checking: If `ivl` is `NULL` or `ilist < 0` then an error message is printed and the program exits.

4. `void IVL_setPointerToList ( IVL *ivl, int ilist, int size, int ivec[] ) ;`

   This method is similar to `IVL_setList()` but is used only with `type = IVL_CHUNKED`. It simply sets a size and pointer. The maximum number of lists and the number of lists are resized as necessary.

   Error checking: If `ivl` is `NULL` or `type != IVL_CHUNKED`. or `ilist < 0` then an error message is printed and the program exits.

9.2.5 Utility methods

1. `int IVL_sizeOf ( IVL *ivl ) ;`

   This method returns the number of bytes taken by this object.

   Error checking: If `ivl` is `NULL`, an error message is printed and the program exits.

2. `int IVL_min ( IVL *ivl ) ;`
   `int IVL_max ( IVL *ivl ) ;`
   `int IVL_maxListSize ( IVL *ivl ) ;`
   `int IVL_sum ( IVL *ivl ) ;`

   These methods return some simple information about the object.

   Error checking: If `ivl` is `NULL` then an error message is printed and the program exits.

3. `int IVL_sortUp ( IVL *ivl ) ;`

   This method sorts each list into ascending order.

   Error checking: If `ivl` is `NULL` or `nlist < 0` then an error message is printed and the program exits.

4. `int * IVL_equivMap1 ( IVL *ivl ) ;`
   `IV  * IVL_equivMap2 ( IVL *ivl ) ;`

   Two lists are equivalent if their contents are identical. These methods are used to find the natural compressed graph of a matrix [3]. The returned `int` vector or `IV` object has size `ivl->nlist` and contains a map from the lists in `ivl` to the lists in the new `IVL` object. If `nlist` is zero, `NULL` is returned.

Error checking: As usual, if `ivl` is `NULL` or `nlist < 0` then an error message is printed and the program exits.

5. `void IVL_overwrite ( IVL *ivl, IV *oldToNewIV ) ;`

   This method overwrite the entries in each list using an old-to-new vector. If an entry in a list is out of range, i.e., it is not in `[0,size-1]` where `size` is the size of `oldToNewIV`, the entry is not changed.

   Error checking: If `ivl` or `oldToNewIV` is `NULL`, an error message is printed and the program exits.

6. `IVL * IVL_mapEntries ( IVL *ivl, IV *mapIV ) ;`

   This method creates and returns a new `IVL` object. List `ilist` in the new `IVL` object contains the image of the entries in list `ilist` of the old `IVL` object, i.e., the old entries are mapped using the `mapIV` map vector and duplicates are purged.

   Error checking: If `ivl` or `mapIV` is `NULL`, an error message is printed and the program exits.

7. `void IVL_absorbIVL ( IVL *ivl1, IVL *ivl2, IV *mapIV ) ;`

   In this method, object `ivl1` absorbs the lists and entries of object `ivl2`. List `ilist` of object `ivl1` is mapped into list `map[ilist]` of object `ivl2`, where `map[]` is the vector from the `mapIV` object. All `Ichunk` objects once owned by `ivl2` are now owned by `ivl1`.

   Error checking: If `ivl1`, `ivl2` or `mapIV` is `NULL`, or if the size pf `mapIV` is not equal to the number of lists in `ivl2`, or if the vector in `mapIV` is `NULL`, then an error message is printed and the program exits.

8. `IVL * IVL_expand ( IVL *ivl, IV *eqmapIV ) ;`

   This method was created in support of a symbolic factorization. An `IVL` object is constructed using a compressed graph. it must be expanded to reflect the compressed graph. The number of lists does not change (there is one list per front) but the size of each list may change. so we create and return a new `IVL` object that contains entries for the uncompressed graph.

   Error checking: If `ivl` or `eqmapIV` is `NULL`, an error message is printed and the program exits.

9.2.6 Miscellaneous methods

1. `IVL * IVL_make9P ( int n1, int n2, int ncomp ) ;`

   This method returns an `IVL` object that contains the full adjacency structure for a 9-point operator on a `n1` $\times$ `n2` grid with `ncomp` components at each grid point.

   Error checking: If `n1`, `n2` or `ncomp` is less than or equal to zero, an error message is printed and the program exits.

2. `IVL * IVL_make13P ( int n1, int n2 ) ;`

   This method returns an `IVL` object that contains the full adjacency structure for a 13-point two dimensional operator on a `n1` $\times$ `n2` grid.

   Error checking: If `n1` or `n2` is less than or equal to zero, an error message is printed and the program exits.

3. `IVL * IVL_make5P ( int n1, int n2 ) ;`

   This method returns an `IVL` object that contains the full adjacency structure for a 5-point two dimensional operator on a `n1` $\times$ `n2` grid.

   Error checking: If `n1` or `n2` is less than or equal to zero, an error message is printed and the program exits.

4. `IVL * IVL_make27P ( int n1, int n2, int ncomp ) ;`

   This method returns an `IVL` object that contains the full adjacency structure for a 27-point operator on a `n1` × `n2` × `n3` grid with `ncomp` components at each grid point.

   Error checking: If `n1`, `n2`, `n3` or `ncomp` is less than or equal to zero, an error message is printed and the program exits.

9.2.7 IO methods

There are the usual eight IO routines. The file structure of a `IVL` object is simple: `type`, `nlist` and `tsize`, followed by `sizes[nlist]`, followed by the lists pointed to by `p_vec[]`.

1. `int IVL_readFromFile ( IVL *ivl, char *fn ) ;`

   This method reads an `IVL` object from a file. If the the file can be opened successfully, the method calls `IVL_readFromFormattedFile()` or `IVL_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `ivl` or `fn` are `NULL`, or if `fn` is not of the form `*.ivlf` (for a formatted file) or `*.ivlb` (for a binary file), an error message is printed and the method returns zero.

2. `int IVL_readFromFormattedFile ( IVL *ivl, FILE *fp ) ;`

   This method reads an `IVL` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `ivl` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int IVL_readFromBinaryFile ( IVL *ivl, FILE *fp ) ;`

   This method reads an `IVL` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `ivl` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int IVL_writeToFile ( IVL *ivl, char *fn ) ;`

   This method writes an `IVL` object to a file. If the the file can be opened successfully, the method calls `IVL_writeFromFormattedFile()` or `IVL_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `ivl` or `fn` are `NULL`, or if `fn` is not of the form `*.ivlf` (for a formatted file) or `*.ivlb` (for a binary file), an error message is printed and the method returns zero.

5. `int IVL_writeToFormattedFile ( IVL *ivl, FILE *fp ) ;`

   This method writes an `IVL` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `ivl` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int IVL_writeToBinaryFile ( IVL *ivl, FILE *fp ) ;`

   This method writes an `IVL` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `ivl` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int IVL_writeForHumanEye ( IVL *ivl, FILE *fp ) ;`

   This method writes an `IVL` object to a file in an easily readable format. The method `IVL_writeStats()` is called to write out the header and statistics. The value `1` is returned.

   Error checking: If `ivl` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int IVL_writeStats ( IVL *ivl, FILE *fp ) ;`

   This method writes some statistics about an `IVL` object to a file. The value `1` is returned.

   Error checking: If `ivl` or `fp` are `NULL`, an error message is printed and zero is returned.

9.3 Driver programs for the IVL object

This section contains brief descriptions of six driver programs.

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads in a `IVL` object from `inFile` and writes out the object to `outFile`

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `IVL` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `IVL` object. It must be of the form `*.ivlf` or `*.ivlb`. The `IVL` object is read from the file via the `IVL_readFromFile()` method.
   - The `outFile` parameter is the output file for the `IVL` object. It must be of the form `*.ivlf` or `*.ivlb`. The `IVL` object is written to the file via the `IVL_writeToFile()` method.

2. `testExpand msglvl msgFile inIVLfile inEqmapFile outIVLfile`

   This program is used to test the expand function. One application is the symbolic factorization. We need the adjacency structure of the factor matrix. We could compute it directly from the original graph, or we could compute the adjacency structure of the compressed graph and then expand it into the full adjacency structure. The second method is usually faster.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `IVL` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inIVLfile` parameter is the input file for the first, unexpanded `IVL` object. It must be of the form `*.ivlf` or `*.ivlb`. The `IVL` object is read from the file via the `IVL_readFromFile()` method.
   - The `inEqmapFile` parameter is the input file for the map from uncompressed vertices to compressed vertices. It must be of the form `*.ivf` or `*.ivb`. The `IV` object is read from the file via the `IV_readFromFile()` method.
   - The `outIVLfile` parameter is the output file for the second, expanded `IVL` object. It must be of the form `*.ivlf` or `*.ivlb`. The `IVL` object is read from the file via the `IVL_readFromFile()` method.

Chapter 10

`Ideq`: Integer Dequeue

The `Ideq` is a object that manages a *dequeue*, a list data structure that supports inserts and deletes at both the head and the tail of the list. We wrote this application in support of a max flow code where visiting an out-edge put a vertex on the head of the list and visiting an in-edge put a vertex on the tail of the list. The goal was to be close to a depth first traversal of the network. This object is also used in multithreaded and MPI factorizations and forward and back solves, where each process must perform a bottom-up or top-down traversal of a tree. The `Ideq` object is used to specify which nodes of the tree to visit (possibly repeatedly) in which order.

The dequeue has fixed size though it can grow using the `Ideq_resize()` method.

10.1 Data Structure

The `Ideq` object has four fields.

- `int maxsize` : maximum size of the dequeue.
- `int head` : head of the list.
- `int tail` : tail of the list.
- `IV iv` : an `IV` object to hold the list vector.

A correctly initialized and nontrivial `Ideq` object will have `maxsize > 0`. When the dequeue is empty, `head = tail = -1`.

10.2 Prototypes and descriptions of `Ideq` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Ideq` object.

10.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Ideq * Ideq_new ( void ) ;`

   This method simply allocates storage for the `Ideq` structure and then sets the default fields by a call to `Ideq_setDefaultFields()`.

2. `void Ideq_setDefaultFields ( Ideq *deq ) ;`

   This method sets the structure's fields to default values: `head` and `tail` are set to `-1`, `maxsize` is set to zero, and the fields for `iv` are set via a call to `IV_setDefaultFields()`.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

3. `void Ideq_clearData ( Ideq *deq ) ;`

   This method clears any data owned by the object. It releases any storage held by the `deq->iv` object with a call to `IV_clearData()`. It then calls `Ideq_setDefaultFields()`.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

4. `void Ideq_free ( Ideq *deq ) ;`

   This method releases any storage by a call to `Ideq_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

10.2.2 Initializer methods

There is one initializer method.

1. `int Ideq_resize ( Ideq *deq, int newsize ) ;`

   This method resizes the dequeue list, either increasing the size or decreasing the size (if possible). Since after `Ideq_new()` the size of the list is zero, this method also serves as an initializer.

   If the present size of the list (the number of entries between `head` and `tail` inclusive) is larger than `newsize`, the method returns `-1`. Otherwise, a new `int` vector is allocated and filled with the entries in the list. The old `int` vector is free'd, the new vector is spliced into the `IV` object, and the `head`, `tail` and `maxsize` fields are set. The method then returns `1`.

   Error checking: If `deq` is `NULL`, or if `newsize` < 0, an error message is printed and the program exits.

10.2.3 Utility methods

1. `void Ideq_clear ( Ideq *deq ) ;`

   This method clears the dequeue. The `head` and `tail` fields are set to `-1`.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

2. `int Ideq_head ( Ideq *deq ) ;`

   This method returns the value at the head of the list without removing that value. If `head == -1` then `-1` is returned. Note, the list may be nonempty and the first value may be `-1`, so `-1` may signal an empty list or a terminating element.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

3. `int Ideq_removeFromHead ( Ideq *deq ) ;`

   This method returns the value at the head of the list and removes that value. If `head == -1` then `-1` is returned. Note, the list may be nonempty and the first value may be `-1`, so `-1` may signal an empty list or a terminating element.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

4. `int Ideq_insertAtHead ( Ideq *deq, int val ) ;`

   This method inserts a value `val` into the list at the head of the list. If there is no room in the list, `-1` is returned and the dequeue must be resized using the `Ideq_resize()` method. Otherwise, the item is placed into the list and `1` is returned.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

5. `int Ideq_tail ( Ideq *deq ) ;`

   This method returns the value at the tail of the list without removing that value. If `tail == -1` then `-1` is returned. Note, the list may be nonempty and the first value may be `-1`, so `-1` may signal an empty list or a terminating element.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

6. `int Ideq_removeFromTail ( Ideq *deq ) ;`

   This method returns the value at the tail of the list and removes that value. If `tail == -1` then `-1` is returned. Note, the list may be nonempty and the first value may be `-1`, so `-1` may signal an empty list or a terminating element.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

7. `int Ideq_insertAtTail ( Ideq *deq, int val ) ;`

   This method inserts a value `val` into the list at the tail of the list. If there is no room in the list, `-1` is returned and the dequeue must be resized using the `Ideq_resize()` method. Otherwise, the item is placed into the list and `1` is returned.

   Error checking: If `deq` is `NULL`, an error message is printed and the program exits.

10.2.4 IO methods

1. `void Ideq_writeForHumanEye ( Ideq *deq ) ;`

   This method write the state of the object, (the size, head and tail) and the list of entries to a file.

   Error checking: If `deq` or `fp` is `NULL`, an error message is printed and the program exits.

Chapter 11

Lock: Mutual Exclusion Lock object

The Lock object is an object that is used to insulate the rest of the library from the particular thread package that is active. The FrontMtx, ChvList, ChvManager, SubMtxList and SubMtxManager objects all may contain a mutual exclusion lock to govern access to their critical sections of code in a multithreaded environment. Instead of putting the raw code that is specific to a particular thread library into each of these objects, each has a Lock object. It is this Lock object that contains the code and data structures for the different thread libraries.

At present we have the Solaris and POSIX thread libraries supported by the Lock object. The header file Lock.h contains #if/#endif statements that switch over the supported libraries. The THREAD_TYPE parameter is used to make the switch. Porting the library to another thread package requires making changes to the Lock object. The parallel factor and solve methods that belong to the FrontMtx object also need to have additional code inserted into them to govern thread creation, joining, etc, but the switch is made by the THREAD_TYPE definition found in the header file Lock.h. It is possible to use the code without any thread package — simply set THREAD_TYPE to TT_NONE in the Lock.h file.

11.1 Data Structure

The Lock structure has three fields.

- int nlocks : number of locks made.
- int nunlocks : number of unlocks made.
- the mutual exclusion lock

  For Solaris threads we have mutex_t *mutex.

  For POSIX threads we have pthread_mutex_t *mutex.

  For no threads we have void *mutex.

11.2 Prototypes and descriptions of Lock methods

11.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Lock * Lock_new ( void ) ;`

   This method simply allocates storage for the `Lock` structure and then sets the default fields by a call to `Lock_setDefaultFields()`.

2. `void Lock_setDefaultFields ( Lock *lock ) ;`

   This method sets the structure's fields to default values: `nlocks` and `nunlocks` are zero, and `mutex` is `NULL`.

   Error checking: If `lock` is `NULL`, an error message is printed and the program exits.

3. `void Lock_clearData ( Lock *lock ) ;`

   This method clears the data for the object. If `lock->mutex` is not `NULL`, then `mutex_destroy(lock->mutex)` is called (for the Solaris thread package) or `pthread_mutex_destroy(lock->mutex)` is called (for the POSIX thread package), The method concludes with a call to `Lock_setDefaultFields()`.

   Error checking: If `lock` is `NULL`, an error message is printed and the program exits.

4. `void Lock_free ( Lock *lock ) ;`

   This method releases any storage by a call to `Lock_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `lock` is `NULL`, an error message is printed and the program exits.

11.2.2 Initializer method

1. `void Lock_init ( Lock *lock, int lockflag ) ;`

   This is the basic initializer method. Any previous data is cleared with a call to `Lock_clearData()`. If `lockflag = 0`, then no lock is initialized. For the Solaris thread package, `lockflag = 1` means the lock will be initialized to synchronize only threads in this process, while `lockflag = 2` means the lock will be initialized to synchronize threads across processes. For the POSIX thread package, `lockflag != 0` means the lock will be initialized to synchronize only threads in this process.

   Error checking: If `lock` is `NULL`, an error message is printed and the program exits.

11.2.3 Utility methods

1. `void Lock_lock ( Lock *lock ) ;`

   This method locks the lock.

   Error checking: If `lock` is `NULL`, an error message is printed and the program exits.

2. `void Lock_unlock ( Lock *lock ) ;`

   This method unlocks the lock.

   Error checking: If `lock` is `NULL`, an error message is printed and the program exits.

Chapter 12

Perm: Permutation Object

The Perm object is used to store a pair of permutation vectors. The main function of this object is to read and write permutations from and to files.

12.1 Data Structure

The Perm object can contain two permutation vectors, an *old-to-new* and a *new-to-old* permutation. One or both may be present in the structure.

The Perm structure has four fields.

- int isPresent : flag to tell which vectors are present
  - 0 ⟶ neither is present
  - 1 ⟶ newToOld is present, oldToNew is not
  - 2 ⟶ oldToNew is present, newToOld is not
  - 3 ⟶ both newToOld and oldToNew are present
- int size : dimension of the vectors
- int *newToOld : pointer to the new-to-old vector
- int *oldToNew : pointer to the old-to-new vector

12.2 Prototypes and descriptions of Perm methods

This section contains brief descriptions including prototypes of all methods that belong to the Perm object.

12.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. Perm * Perm_new (void) ;

   This method simply allocates storage for the Perm structure and then sets the default fields by a call to Perm_setDefaultFields().

2. `void Perm_setDefaultFields ( Perm *perm ) ;`

   This method sets the structure's fields to default values.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

3. `void Perm_clearData ( Perm *perm ) ;`

   This method clears data owned by the object. If `perm->newToOld` is not `NULL`, its storage is free'd with a call to `IVfree()`. If `perm->oldToNew` is not `NULL`, its storage is free'd with a call to `IVfree()`. The structure's default fields are then set with a call to `Perm_setDefaultFields()`.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

4. `void Perm_free ( Perm *perm ) ;`

   This method releases any storage by a call to `Perm_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

12.2.2 Initializer methods

There is one initializer method.

1. `void Perm_initWithTypeAndSize ( Perm *perm, int isPresent, int size ) ;`

   This method is the primary initializer. It clears any previous data with a call to `Perm_clearData()`. Then the `isPresent` and `size` fields are set. If `isPresent == 1` then `newToOld` is set with a call to `IVinit()`. If `isPresent == 2` then `oldToNew` is set with a call to `IVinit()`. If `isPresent == 3` then `newToOld` and `newToOld` are set with calls to `IVinit()`.

   Error checking: If `perm` is `NULL`, or if `isPresent` is invalid, or if `size <= 0`, an error message is printed and the program exits.

12.2.3 Utility methods

1. `int Perm_sizeOf ( Perm *perm ) ;`

   This method returns the number of bytes taken by this object.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

2. `int Perm_checkPerm ( Perm *perm ) ;`

   This method checks the validity of the `Perm` object. If `oldToNew` is present, it is checked to see that it is a true permutation vector, i.e., a one-one and onto map from `[0,size)` to `[0,size)`, and similarly for `newToOld` if it is present. If the permutation vector(s) are valid, `1` is returned, otherwise `0` is returned.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

3. `void Perm_fillOldToNew ( Perm *perm ) ;`

   If `oldToNew` is already present, the function returns. Otherwise, `oldToNew` is initialized with a call to `IVinit()` and has its values set from `newToOld`.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

4. `void Perm_fillNewToOld ( Perm *perm ) ;`

   If `NewToOld` is already present, the function returns. Otherwise, `NewToOld` is initialized with a call to `IVinit()` and has its values set from `oldToNew`.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

5. `void Perm_releaseOldToNew ( Perm *perm ) ;`

   If `oldToNew` is not present, the function returns. Otherwise, the storage for `oldToNew` is released with a call to `IVfree()` the `isPresent` field is changed appropriately.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

6. `void Perm_releaseNewToOld ( Perm *perm ) ;`

   If `NewToOld` is not present, the function returns. Otherwise, the storage for `NewToOld` is released with a call to `IVfree()` the `isPresent` field is changed appropriately.

   Error checking: If `perm` is `NULL`, an error message is printed and the program exits.

7. `Perm * Perm_compress ( Perm *perm, IV *eqmapIV ) ;`

   This method takes as input a `Perm` object and an equivalence map `IV` object (they must be the same size). The equivalence map is from vertices to indistinguishable vertices in a compressed graph. A second `Perm` object is produced that contains the equivalent permutation on the compressed graph.

   Error checking: If `perm` or `eqmapIV` are `NULL`, an error message is printed and zero is returned.

12.2.4 IO methods

There are the usual eight IO routines. The file structure of a `Perm` object is simple:

`isPresent size`
`oldToNew[size]` (if present)
`newToOld[size]` (if present)

1. `int Perm_readFromFile ( Perm *perm, char *fn ) ;`

   This method reads a `Perm` object from a file. It tries to open the file and if it is successful, it then calls `Perm_readFromFormattedFile()` or `Perm_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `perm` or `fn` are `NULL`, or if `fn` is not of the form `*.permf` (for a formatted file) or `*.permb` (for a binary file), an error message is printed and the method returns zero.

2. `int Perm_readFromFormattedFile ( Perm *perm, FILE *fp ) ;`

   This method reads in a `Perm` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned. Note, if the permutation vectors are one-based (as for Fortran), they are converted to zero-based vectors.

   Error checking: If `perm` or `fp` are `NULL`, an error message is printed and zero is returned.

3. `int Perm_readFromBinaryFile ( Perm *perm, FILE *fp ) ;`

   This method reads in a `Perm` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned. Note, if the permutation vectors are one-based (as for Fortran), they are converted to zero-based vectors.

   Error checking: If `perm` or `fp` are `NULL`, an error message is printed and zero is returned.

4. `int Perm_writeToFile ( Perm *perm, char *fn ) ;`

   This method writes a `Perm` object to a file. It tries to open the file and if it is successful, it then calls `Perm_writeFromFormattedFile()` or `Perm_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `perm` or `fn` are `NULL`, or if `fn` is not of the form `*.permf` (for a formatted file) or `*.permb` (for a binary file), an error message is printed and the method returns zero.

5. `int Perm_writeToFormattedFile ( Perm *perm, FILE *fp ) ;`

   This method writes out a `Perm` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `perm` or `fp` are `NULL`, an error message is printed and zero is returned.

6. `int Perm_writeToBinaryFile ( Perm *perm, FILE *fp ) ;`

   This method writes out a `Perm` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `perm` or `fp` are `NULL`, an error message is printed and zero is returned.

7. `int Perm_writeForHumanEye ( Perm *perm, FILE *fp ) ;`

   This method writes out a `Perm` object to a file in a human readable format. The method `Perm_writeStats()` is called to write out the header and statistics. The value `1` is returned.

   Error checking: If `perm` or `fp` are `NULL`, an error message is printed and zero is returned.

8. `int Perm_writeStats ( Perm *perm, FILE *fp ) ;`

   This method writes out a header and statistics to a file. The value `1` is returned.

   Error checking: If `perm` or `fp` are `NULL`, an error message is printed and zero is returned.

Chapter 13

`Utilities` directory

The `Utilities` directory contains a multitude of routines that manipulate vectors: `char`, `int`, `float`, `double` and `double` vectors that are used to contain double precision complex entries. There are a variety of routines to sort vectors with and without companion vectors. Our sort routines are based on [7], a quicksort algorithm that uses the “nither” function (a median of three medians) to select the partition element, and tripartite partitioning.

Aside from vector routines, the `Utilities` directory contains some methods used to manipulate elements in singly linked lists. The `IP` structure (an `int` data element and a pointer) is used by several objects to manage singly linked lists. The `I2OP` structure (two `int` and one `void *` data elements) is used in a two-keyed hash table.

13.1 Data Structures

There are two data structures used in singly linked lists.

- `IP`: a singly linked list element with an `int` data field.

```
typedef struct _IP IP ;
struct _IP {
   int   val   ;
   IP    *next ;
} ;
```

- `I2OP`: a singly linked list element with two `int` and one `void *` data fields.

```
typedef struct _I2OP I2OP ;
struct _I2OP {
   int    value0  ;
   int    value1  ;
   void   *value2 ;
   I2OP   *next   ;
} ;
```

13.2 Prototypes and descriptions of `Utilities` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Utilities` directory.

13.2.1 CV : char vector methods

1. `char * CVinit ( int n, char c ) ;`

   This is the allocator and initializer method for `char` vectors. Storage for an array with size `n` is found and each entry is filled with character `c`. A pointer to the array is returned.

2. `char * CVinit2 ( int n ) ;`

   This is an allocator method for `char` vectors. Storage for an array with size `n` is found. A pointer to the array is returned. Note, on return, there will likely be garbage in the array.

3. `void CVfree ( char cvec[] ) ;`

   This method releases the storage taken by `cvec[]`.

4. `void CVcopy ( int n, char y[], char x[] ) ;`

   This method copies `n` entries from `x[]` to `y[]`, i.e., `y[i] = x[i]` for `0 <= i < n`.

5. `void CVfill ( int n, char y[], char c ) ;`

   This method fills `n` entries in `y[]` with the character `c`, i.e., `y[i] = c` for `0 <= i < n`.

6. `void CVfprintf ( FILE *fp, int n, char y[] ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The format is new line followed by lines of eighty columns of single characters.

7. `int CVfp80 ( FILE *fp, int n, char y[], int column, int *pierr ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The method splices vectors together or naturally breaks the large vectors into lines. The `column` value is the present location, one can add (80 − `column`) more characters before having to form a new line. The number of the present character in the line is returned. If `*pierr < 0`, an IO error has occured.

8. `int CVfscanf ( FILE *fp, int n, char y[] ) ;`

   This method scans in characters from file `fp` and places them in the array `y[]`. It tries to read in `n` characters, and returns the number that were actually read.

13.2.2 DV : double vector methods

1. `double * DVinit ( int n, double val ) ;`

   This is the allocator and initializer method for `double` vectors. Storage for an array with size `n` is found and each entry is filled with `val`. A pointer to the array is returned.

2. `double * DVinit2 ( int n ) ;`

   This is an allocator method for `double` vectors. Storage for an array with size `n` is found. A pointer to the array is returned. Note, on return, there will likely be garbage in the array.

3. `void DVfree ( int vec[] ) ;`

   This method releases the storage taken by `vec[]`.

4. `void DVfprintf ( FILE *fp, int n, double y[] ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The format is new line followed by lines of six `double`'s in `"%12.4e"` format.

5. `int DVfscanf ( FILE *fp, int n, double y[] ) ;`

   This method scans in `double`'s from file `fp` and places them in the array `y[]`. It tries to read in `n` `double`'s, and returns the number that were actually read.

6. `void DVadd ( int n, double y[], double x[] ) ;`

   This method adds `n` entries from `x[]` to `y[]`, i.e., `y[i] += x[i]` for `0 <= i < n`.

7. `void DVaxpy ( int n, double y[], double alpha, double x[] ) ;`

   This method adds a scaled multiple of `n` entries from `x[]` into `y[]`, i.e., `y[i] += alpha * x[i]` for `0 <= i < n`.

8. ```
   void DVaxpy2 ( int n, double z[], double a, double x[],
                  double b, double y[] ) ;
   ```

   This method adds a scaled multiple of two vectors `x[]` and `y[]` to another vector `z[]`, i.e., i.e., `z[i] += a * x[i] + b * y[i]` for `0 <= i < n`.

9. ```
   void DVaxpy33 ( int n, double y0[], double y1[], double y2[],
                   double alpha, double x0[], double x1[], double x2[] ) ;
   ```

   This method computes this computation.

   ```
   y0[] = y0[] + alpha[0] * x0[] + alpha[1] * x1[] + alpha[2] * x2[]
   y1[] = y1[] + alpha[3] * x0[] + alpha[4] * x1[] + alpha[5] * x2[]
   y2[] = y2[] + alpha[6] * x0[] + alpha[7] * x1[] + alpha[8] * x2[]
   ```

10. ```
    void DVaxpy32 ( int n, double y0[], double y1[], double y2[],
                    double alpha, double x0[], double x1[] ) ;
    ```

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[] + alpha[1] * x1[]
    y1[] = y1[] + alpha[2] * x0[] + alpha[3] * x1[]
    y2[] = y2[] + alpha[4] * x0[] + alpha[5] * x1[]
    ```

11. ```
    void DVaxpy31 ( int n, double y0[], double y1[], double y2[],
                    double alpha, double x0[], double x1[] ) ;
    ```

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[]
    y1[] = y1[] + alpha[1] * x0[]
    y2[] = y2[] + alpha[2] * x0[]
    ```

12. ```
    void DVaxpy23 ( int n, double y0[], double y1[],
                    double alpha, double x0[], double x1[], double x2[] ) ;
    ```

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[] + alpha[1] * x1[] + alpha[2] * x2[]
    y1[] = y1[] + alpha[3] * x0[] + alpha[4] * x1[] + alpha[5] * x2[]
    ```

13. ```
    void DVaxpy22 ( int n, double y0[], double y1[],
                    double alpha, double x0[], double x1[] ) ;
    ```

    This method computes this computation.

```
y0[] = y0[] + alpha[0] * x0[] + alpha[1] * x1[]
y1[] = y1[] + alpha[2] * x0[] + alpha[3] * x1[]
```

14. `void DVaxpy21 ( int n, double y0[], double y1[], double alpha, double x0[] ) ;`

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[]
    y1[] = y1[] + alpha[1] * x0[]
    ```

15. ```
    void DVaxpy13 ( int n, double y0[],
                    double alpha, double x0[], double x1[], double x2[] ) ;
    ```

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[] + alpha[1] * x1[] + alpha[2] * x2[]
    ```

16. `void DVaxpy12 ( int n, double y0[], double alpha, double x0[], double x1[] ) ;`

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[] + alpha[1] * x1[]
    ```

17. `void DVaxpy11 ( int n, double y0[], double alpha, double x0[] ) ;`

    This method computes this computation.

    ```
    y0[] = y0[] + alpha[0] * x0[]
    ```

18. `void DVaxpyi ( int n, double y[], int index[], double alpha, double x[] ) ;`

    This method scatteradds a scaled multiple of `n` entries from `x[]` into `y[]`, i.e., `y[index[i]] += alpha * x[i]` for `0 <= i < n`.

19. ```
    void DVcompress ( int n1, double x1[], double y1[],
                      int n2, double x2[], double y2[] ) ;
    ```

    Given a pair of arrays `x1[n1]` and `y1[n1]`, fill `x2[n2]` and `y2[n2]` with a subset of the `(x1[j],y1[j]` entries whose distribution is an approximation.

20. `void DVcopy ( int n, double y[], double x[] ) ;`

    This method copies `n` entries from `x[]` to `y[]`, i.e., `y[i] = x[i]` for `0 <= i < n`.

21. `int DVdot ( int n, double y[], double x[] ) ;`

    This method returns the dot product of the vector `x[]` and `y[]`, i.e., return $\sum_{\mathtt{i=0}}^{\mathtt{n-1}}(\mathtt{x[i]} * \mathtt{y[i]})$.

22. ```
    int DVdot33 ( int n, double row0[], double row1[], double row2[],
                  double col0[], double col1[], double col2[], double sums[] ) ;
    ```

    This method computes nine dot products.

$$
\begin{array}{lll}
\mathtt{sums}[0] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] & \mathtt{sums}[1] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] & \mathtt{sums}[2] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}] \\
\mathtt{sums}[3] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] & \mathtt{sums}[4] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] & \mathtt{sums}[5] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row1}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}] \\
\mathtt{sums}[6] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row2}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] & \mathtt{sums}[7] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row2}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] & \mathtt{sums}[8] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row2}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}]
\end{array}
$$
    ```

23. `int DVdot32 ( int n, double row0[], double row1[], double row2[],`
    `double col0[], double col1[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[2] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$

24. `int DVdot31 ( int n, double row0[], double row1[], double row2[],`
    `double col0[], double sums[] ) ;`

    This method computes three dot products.

$$\mathtt{sums}[0] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[2] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$

25. `int DVdot23 ( int n, double row0[], double row1[],`
    `double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] \quad \mathtt{sums}[2] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}]$$
$$\mathtt{sums}[3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[4] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] \quad \mathtt{sums}[5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}]$$

26. `int DVdot22 ( int n, double row0[], double row1[],`
    `double col0[], double col1[], double sums[] ) ;`

    This method computes four dot products.

$$\mathtt{sums}[0] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[2] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$

27. `int DVdot21 ( int n, double row0[], double row1[],`
    `double col0[], double sums[] ) ;`

    This method computes two dot products.

$$\mathtt{sums}[0] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$

28. `int DVdot13 ( int n, double row0[],`
    `double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes six dot products.

    $$\texttt{sums}[0] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{row0}[\texttt{i}] * \texttt{col0}[\texttt{i}] \quad \texttt{sums}[1] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{row0}[\texttt{i}] * \texttt{col1}[\texttt{i}] \quad \texttt{sums}[2] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{row0}[\texttt{i}] * \texttt{col2}[\texttt{i}]$$

29. `int DVdot12 ( int n, double row0[], double row1[],`
    `double col0[], double col1[], double sums[] ) ;`

    This method computes two dot products.

    $$\texttt{sums}[0] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{row0}[\texttt{i}] * \texttt{col0}[\texttt{i}] \quad \texttt{sums}[1] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{row0}[\texttt{i}] * \texttt{col1}[\texttt{i}]$$

30. `int DVdot11 ( int n, double row0[], double col0[], double sums[] ) ;`

    This method computes one dot product.

    $$\texttt{sums}[0] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{row0}[\texttt{i}] * \texttt{col0}[\texttt{i}]$$

31. `int DVdoti ( int n, double y[], int index[], double x[] ) ;`

    This method returns the indexed dot product $\sum_{\texttt{i}=0}^{\texttt{n}-1} \texttt{y}[\texttt{index}[\texttt{i}]] * \texttt{x}[\texttt{i}]$.

32. `void DVfill ( int n, double y[], double val ) ;`

    This method fills `n` entries in `y[]` with `val`, i.e., `y[i] = val` for `0 <= i < n`.

33. `void DVgather ( int n, double y[], double x[], int index[] ) ;`

    `y[i] = x[index[i]]` for `0 <= i < n`.

34. `void DVgatherAddZero ( int n, double y[], double x[], int index[] ) ;`

    `y[i] += x[index[i]]` and `x[index[i]] = 0` for `0 <= i < n`.

35. `void DVgatherZero ( int n, double y[], double x[], int index[] ) ;`

    `y[i] = x[index[i]]` and `x[index[i]] = 0`

36. `void DVinvPerm ( int n, double y[], int index[] ) ;`

    This method permutes the vector y as follows. i.e., `y[index[i]] := y[i]`. See `DVperm()` for a similar function.

37. `double DVmax ( int n, double y[], int *ploc ) ;`

    This method returns the maximum entry in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

38. `double DVmaxabs ( int n, double y[], int *ploc ) ;`

    This method returns the maximum magnitude of entries in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

39. `double DVmin ( int n, double y[], int *ploc ) ;`

    This method returns the minimum entry in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

40. `double DVminabs ( int n, double y[], int *ploc ) ;`

    This method returns the minimum magnitude of entries in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

41. `void DVperm ( int n, double y[], int index[] ) ;`

    This method permutes the vector y as follows. i.e., `y[i] := y[index[i]]`. See `DVinvPerm()` for a similar function.

42. `void DVramp ( int n, double y[], double start, double inc ) ;`

    This method fills `n` entries in `y[]` with values `start`, `start + inc`, `start + 2*inc`, `start + 3*inc`, etc.

43. `void DVscale ( int n, double y[], double alpha ) ;`

    This method scales a vector `y[]` by `alpha`, i.e., `y[i] *= alpha`. for `0 <= i < n`.

44. `void DVscale2 ( int n, double x[], double y[],`
    `double a, double b, double c, double d ) ;`

    This method scales two vectors `y[]` by a 2×2 matrix, i.e.,

$$\left[\begin{array}{ccc} \mathtt{x[0]} & \ldots & \mathtt{x[n-1]} \\ \mathtt{y[0]} & \ldots & \mathtt{y[n-1]} \end{array}\right] := \left[\begin{array}{cc} \mathtt{a} & \mathtt{b} \\ \mathtt{c} & \mathtt{d} \end{array}\right] \left[\begin{array}{ccc} \mathtt{x[0]} & \ldots & \mathtt{x[n-1]} \\ \mathtt{y[0]} & \ldots & \mathtt{y[n-1]} \end{array}\right].$$

45. `void DVscatter ( int n, double y[], int index[], double x[] ) ;`

    This method scatters `n` entries of `x[]` into `y[]` as follows, `y[index[i]] = x[i]` for `0 <= i < n`.

46. `void DVscatterAdd ( int n, double y[], int index[], double x[] ) ;`

    This method scatters/adds `n` entries of `x[]` into `y[]` as follows, `y[index[i]] += x[i]` for `0 <= i < n`.

47. `void DVscatterAddZero ( int n, double y[], int index[], double x[] ) ;`

    This method scatters/adds `n` entries of `x[]` into `y[]` as follows, `y[index[i]] += x[i]` for `0 <= i < n`, and then zeros the entries in `x[*]`.

48. `void DVscatterZero ( int n, double y[], int index[], double x[] ) ;`

    This method scatters `n` entries of `x[]` into `y[]` as follows, `y[index[i]] = x[i]` for `0 <= i < n` and then zeros the entries in `x[*]`.

49. `void DVsub ( int n, double y[], double x[] ) ;`

    This method subtracts `n` entries from `x[]` to `y[]`, i.e., `y[i] -= x[i]` for `0 <= i < n`.

50. `double DVsum ( int n, double y[] ) ;`

    This method returns the sum of the first `n` entries in the vector `x[]`, i.e., return $\sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{x[i]}$.

51. `double DVsumabs ( int n, double y[] ) ;`

    This method returns the sum of the absolute values of the first `n` entries in the vector `x[]`, i.e., return $\sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{abs}(\mathtt{x[i]})$.

52. `void DVswap ( int n, double y[], double x[] ) ;`

    This method swaps the `x[]` and `y[]` vectors as follows. i.e., `y[i] := x[i]` and `x[i] := y[i]` for `0 <= i < n`.

53. `void DVzero ( int n, double y[] ) ;`

    This method zeroes `n` entries in `y[]`, i.e., `y[i] = 0` for `0 <= i < n`.

54. `void DVshuffle ( int n, double y[], int seed ) ;`

    This method shuffles the first `n` entries in `y[]`. The value `seed` is the seed to a random number generator, and one can get repeatable behavior by repeating `seed`.

13.2.3 `ZV : double complex` vector methods

A double precision complex vector of length `n` is simply a double precision vector of length `2n`. There is a separate `ZVinit()` allocator and initializer method, since it requires a real and imaginary part to fill the vector. However, there is no `ZVinit2()` method (which allocates without initializing the entries) nor a `ZVfree()` method to free the entries; the `DVinit2()` and `DVfree()` methods can be used. Similarly, there is no `ZVfscanf()` method, instead the `DVfscanf()` method can be used.

1. `double * ZVinit ( int n, double real, double imag ) ;`

   This is the allocator and initializer method for `double complex` vectors. Storage for an array with size `n` is found and each entry is filled with `(real,imag)`. A pointer to the array is returned.

2. `void ZVfprintf ( FILE *fp, int n, double y[] ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The format is new line followed by `"< %12.4e, %12.4e >"` format.

3. `double Zabs ( double real, double imag ) ;`

   This method returns the magnitude of `(real,imag)`.

4. `int Zrecip ( double areal, double aimag, double *pbreal, double *pbimag ) ;`

   This method fills `*pbreal` and `*pbimag` with the real and imaginary parts of the reciprocal of `(areal,aimag)`. The return value is `0` if `areal` and `aimag` are zero, otherwise the return value is `1`.

5. ```
   int Zrecip2 ( double areal, double aimag, double breal, double bimag,
                 double creal, double cimag, double dreal, double dimag,
                 double *pereal, double *peimag, double *pfreal, double *pfimag,
                 double *pgreal, double *pgimag, double *phreal, double *phimag ) ;
   ```

   This method computes $\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1}$. If `pereal` is not `NULL`, then `*pereal` is loaded with the real part of e. If `peimag` is not `NULL`, then `*peimag` is loaded with the imaginary part of e. Similarly for f, g and h. The return value is `0` if 2×2 matrix is singular, otherwise the return value is `1`.

6. `void ZVaxpy ( int n, double y[], double areal, double aimag, double x[] ) ;`

   This method adds a scaled multiple of `n` entries from `x[]` into `y[]`, i.e., `y[i] += (areal,aimag) * x[i]` for `0 <= i < n`.

7. ```
   void ZVaxpy2 ( int n, double z[], double areal, double aimag,
                  double x[], double breal, double bimag, double y[] ) ;
   ```

   This method adds a scaled multiple of two vectors `x[]` and `y[]` to another vector `z[]`, i.e., i.e., `z[i] += (areal,aimag) * x[i] + (breal,bimag) * y[i]` for `0 <= i < n`.

8. ```
   void ZVaxpy33 ( int n, double y0[], double y1[], double y2[],
                   double alpha[], double x0[], double x1[], double x2[] ) ;
   ```

   This method computes the following.
   ```

```
y0[] = y0[] + alpha[0:1] * x0[] + alpha[2:3] * x1[] + alpha[4:5] * x2[]
y1[] = y1[] + alpha[6:7] * x0[] + alpha[8:9] * x1[] + alpha[10:11] * x2[]
y2[] = y2[] + alpha[12:13] * x0[] + alpha[14:15] * x1[] + alpha[16:17] * x2[]
```

9. `void ZVaxpy32 ( int n, double y0[], double y1[], double y2[], double alpha[], double x0[], double x1[] ) ;`

   This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[] + alpha[2:3] * x1[]
y1[] = y1[] + alpha[4:5] * x0[] + alpha[6:7] * x1[]
y2[] = y2[] + alpha[8:9] * x0[] + alpha[10:11] * x1[]
```

10. `void ZVaxpy31 ( int n, double y0[], double y1[], double y2[], double alpha[], double x0[] ) ;`

    This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[]
y1[] = y1[] + alpha[2:3] * x0[]
y2[] = y2[] + alpha[4:5] * x0[]
```

11. `void ZVaxpy23 ( int n, double y0[], double y1[], double alpha[], double x0[], double x1[], double x2[] ) ;`

    This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[] + alpha[2:3] * x1[] + alpha[4:5] * x2[]
y1[] = y1[] + alpha[6:7] * x0[] + alpha[8:9] * x1[] + alpha[10:11] * x2[]
```

12. `void ZVaxpy22 ( int n, double y0[], double y1[], double alpha[], double x0[], double x1[] ) ;`

    This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[] + alpha[2:3] * x1[]
y1[] = y1[] + alpha[4:5] * x0[] + alpha[6:7] * x1[]
```

13. `void ZVaxpy21 ( int n, double y0[], double y1[], double alpha[], double x0[] ) ;`

    This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[]
y1[] = y1[] + alpha[2:3] * x0[]
```

14. `void ZVaxpy13 ( int n, double y0[], double alpha[], double x0[], double x1[], double x2[] ) ;`

    This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[] + alpha[2:3] * x1[] + alpha[4:5] * x2[]
```

15. `void ZVaxpy12 ( int n, double y0[], double alpha[], double x0[], double x1[] ) ;`

    This method computes the following.

```
y0[] = y0[] + alpha[0:1] * x0[] + alpha[2:3] * x1[]
```

16. `void ZVaxpy11 ( int n, double y0[], double alpha[], double x0[] ) ;`

    This method computes the following.

    ```
    y0[] = y0[] + alpha[0:1] * x0[]
    ```

17. `void ZVcopy ( int n, double y[], double x[] ) ;`

    This method copies `n` entries from `x[]` to `y[]`, i.e., `y[i] = x[i]` for `0 <= i < n`.

18. `void ZVdotU ( int n, double y[], double x[], double *prdot, double *pidot ) ;`

    This method fills `*prdot` and `*pidot` with the real and imaginary part of $\mathtt{y}^T\mathtt{x}$, the dot product of the vector `x[]` and `y[]`.

19. `void ZVdotC ( int n, double y[], double x[], double *prdot, double *pidot ) ;`

    This method fills `*prdot` and `*pidot` with the real and imaginary part of $\mathtt{y}^H\mathtt{x}$, the dot product of the vector `x[]` and `y[]`.

20. ```
    void ZVdotiU ( int n, double y[], int index[], double x[],
                   double *prdot, double *pidot ) ;
    ```

    This method fills `*prdot` and `*pidot` with the real and imaginary parts of the indexed dot product $\sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{y}[\mathtt{index}[\mathtt{i}]] * \mathtt{x}[\mathtt{i}]$.

21. ```
    void ZVdotiC ( int n, double y[], int index[], double x[],
                   double *prdot, double *pidot ) ;
    ```

    This method fills `*prdot` and `*pidot` with the real and imaginary parts of the indexed dot product $\sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{y}[\mathtt{index}[\mathtt{i}]]} * \mathtt{x}[\mathtt{i}]$.

22. ```
    int ZVdotU33 ( int n, double row0[], double row1[], double row2[],
                  double col0[], double col1[], double col2[], double sums[] ) ;
    ```

    This method computes nine dot products.

$$
\begin{array}{ll}
\mathtt{sums}[0;1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] & \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] \\
\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}] & \mathtt{sums}[6:7] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \\
\mathtt{sums}[8:9] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] & \mathtt{sums}[10:11] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}] \\
\mathtt{sums}[12:13] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] & \mathtt{sums}[14:15] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] \\
\mathtt{sums}[16:17] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}] &
\end{array}
$$

23. ```
    int ZVdotU32 ( int n, double row0[], double row1[], double row2[],
                  double col0[], double col1[], double sums[] ) ;
    ```

    This method computes six dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[6:7] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[8:9] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[10:11] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$

24. `int ZVdotU31 ( int n, double row0[], double row1[], double row2[],`
    `double col0[], double sums[] ) ;`

    This method computes three dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row2}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$

25. `int ZVdotU23 ( int n, double row0[], double row1[],`
    `double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}] \quad \mathtt{sums}[6:7] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[8:9] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}] \quad \mathtt{sums}[10:11] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}]$$

26. `int ZVdotU22 ( int n, double row0[], double row1[],`
    `double col0[], double col1[], double sums[] ) ;`

    This method computes four dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[6:7] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$

27. `int ZVdotU21 ( int n, double row0[], double row1[],`
    `double col0[], double sums[] ) ;`

    This method computes two dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{row1}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$

28. `int ZVdotU13 ( int n, double row0[],`
    `double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col2}[\mathtt{i}]$$

29. `int ZVdotU12 ( int n, double row0[], double row1[],`
    `double col0[], double col1[], double sums[] ) ;`

    This method computes two dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col1}[\mathtt{i}]$$

30. `int ZVdotU11 ( int n, double row0[], double col0[], double sums[] ) ;`

    This method computes one dot product.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{row0}[\mathtt{i}] * \mathtt{col0}[\mathtt{i}]$$

31. `int ZVdotC33 ( int n, double row0[], double row1[], double row2[],`
    `double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes nine dot products.

$$\mathtt{sums}[0;1] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \qquad \mathtt{sums}[2:3] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col2}[\mathtt{i}] \qquad \mathtt{sums}[6:7] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[8:9] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}] \qquad \mathtt{sums}[10:11] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col2}[\mathtt{i}]$$
$$\mathtt{sums}[12:13] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row2}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[14:15] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row2}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[16:17] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row2}[\mathtt{i}]} * \mathtt{col2}[\mathtt{i}]$$

32. `int ZVdotC32 ( int n, double row0[], double row1[], double row2[],`
    `double col0[], double col1[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[6:7] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[8:9] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row2}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[10:11] = \sum_{\mathtt{i=0}}^{\mathtt{n-1}} \overline{\mathtt{row2}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$

33. `int ZVdotC31 ( int n, double row0[], double row1[], double row2[], double col0[], double sums[] ) ;`

    This method computes three dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row2}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$

34. `int ZVdotC23 ( int n, double row0[], double row1[], double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col2}[\mathtt{i}] \quad \mathtt{sums}[6:7] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[8:9] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}] \quad \mathtt{sums}[10:11] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col2}[\mathtt{i}]$$

35. `int ZVdotC22 ( int n, double row0[], double row1[], double col0[], double col1[], double sums[] ) ;`

    This method computes four dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[6:7] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$

36. `int ZVdotC21 ( int n, double row0[], double row1[], double col0[], double sums[] ) ;`

    This method computes two dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$
$$\mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row1}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}]$$

37. `int ZVdotC13 ( int n, double row0[], double col0[], double col1[], double col2[], double sums[] ) ;`

    This method computes six dot products.

$$\mathtt{sums}[0:1] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col0}[\mathtt{i}] \quad \mathtt{sums}[2:3] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col1}[\mathtt{i}]$$
$$\mathtt{sums}[4:5] = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \overline{\mathtt{row0}[\mathtt{i}]} * \mathtt{col2}[\mathtt{i}]$$

38. `int ZVdotC12 ( int n, double row0[], double row1[], double col0[], double col1[], double sums[] ) ;`

    This method computes two dot products.

    $$\texttt{sums}[0:1] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \overline{\texttt{row0[i]}} * \texttt{col0[i]} \quad \texttt{sums}[2:3] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \overline{\texttt{row0[i]}} * \texttt{col1[i]}$$

39. `int ZVdotC11 ( int n, double row0[], double col0[], double sums[] ) ;`

    This method computes one dot product.

    $$\texttt{sums}[0:1] = \sum_{\texttt{i}=0}^{\texttt{n}-1} \overline{\texttt{row0[i]}} * \texttt{col0[i]}$$

40. `void ZVgather ( int n, double y[], double x[], int index[] ) ;`

    `y[i] = x[index[i]]` for `0 <= i < n`.

41. `double ZVmaxabs ( int n, double y[] ) ;`

    This method returns the maximum magnitude of entries in `y[0:n-1]`.

42. `double ZVminabs ( int n, double y[] ) ;`

    This method returns the minimum magnitude of entries in `y[0:n-1]`.

43. `void ZVscale ( int n, double y[], double areal, double aimag ) ;`

    This method scales a vector `y[]` by `(areal,aimag)`, i.e., `y[i] *= (areal,aimag)`. for `0 <= i < n`.

44. `void ZVscale2 ( int n, double x[], double y[], double areal, double aimag, double breal, double bimag, double creal, double cimag, double dreal, double dimag ) ;`

    This method scales two vectors `y[]` by a 2×2 matrix, i.e.,

    $$\left[\begin{array}{ccc} \texttt{x}[0] & \ldots & \texttt{x}[\texttt{n}-1] \\ \texttt{y}[0] & \ldots & \texttt{y}[\texttt{n}-1] \end{array}\right] := \left[\begin{array}{cc} (\texttt{areal},\texttt{aimag}) & (\texttt{breal},\texttt{bimag}) \\ (\texttt{creal},\texttt{cimag}) & (\texttt{dreal},\texttt{dimag}) \end{array}\right] \left[\begin{array}{ccc} \texttt{x}[0] & \ldots & \texttt{x}[\texttt{n}-1] \\ \texttt{y}[0] & \ldots & \texttt{y}[\texttt{n}-1] \end{array}\right].$$

45. `void ZVscatter ( int n, double y[], int index[], double x[] ) ;`

    This method scatters `n` entries of `x[]` into `y[]` as follows, `y[index[i]] = x[i]` for `0 <= i < n`.

46. `void ZVsub ( int n, double y[], double x[] ) ;`

    This method subtracts `n` entries from `x[]` to `y[]`, i.e., `y[i] -= x[i]` for `0 <= i < n`.

47. `void ZVzero ( int n, double y[] ) ;`

    This method zeroes `n` entries in `y[]`, i.e., `y[i] = 0` for `0 <= i < n`.

13.2.4 `IV` : `int` vector methods

1. `int * IVinit ( int n, int val ) ;`

    This is the allocator and initializer method for `int` vectors. Storage for an array with size `n` is found and each entry is filled with `val`. A pointer to the array is returned.

2. `int * IVinit2 ( int n ) ;`

    This is an allocator method for `int` vectors. Storage for an array with size `n` is found. A pointer to the array is returned. Note, on return, there will likely be garbage in the array.

3. `void IVfree ( int vec[] ) ;`

   This method releases the storage taken by `vec[]`.

4. `void IVfprintf ( FILE *fp, int n, int y[] ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The format is new line followed by lines of five `int`'s in `" %4d"` format.

5. `int IVfp80 ( FILE *fp, int n, int y[], int column, int *pierr ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The method splices vectors together or naturally breaks the large vectors into lines. The `column` value is the present location. If the printed value of an array entry will not fit within the eighty columns of the present line, a newline character is written and the value starts a new line. The number of the present column in the line is returned. If `*pierr < 0`, an IO error has occured.

6. `int IVfscanf ( FILE *fp, int n, int y[] ) ;`

   This method scans in `int`'s from file `fp` and places them in the array `y[]`. It tries to read in `n` `int`'s, and returns the number that were actually read.

7. ```
   void IVcompress ( int n1, int x1[], int y1[],
                     int n2, int x2[], int y2[] ) ;
   ```

   Given a pair of arrays `x1[n1]` and `y1[n1]`, fill `x2[n2]` and `y2[n2]` with a subset of the `(x1[j],y1[j]` entries whose distribution is an approximation.

8. `void IVcopy ( int n, int y[], int x[] ) ;`

   This method copies `n` entries from `x[]` to `y[]`, i.e., `y[i] = x[i]` for `0 <= i < n`.

9. `void IVfill ( int n, int y[], int val ) ;`

   This method fills `n` entries in `y[]` with `val`, i.e., `y[i] = val` for `0 <= i < n`.

10. `void IVgather ( int n, int y[], int x[], int index[] ) ;`

    `y[i] = x[index[i]]` for `0 <= i < n`.

11. `int * IVinverse ( int n, int y[] ) ;`

    This method allocates and returns a vector of size `n` that it is the inverse of `y[]`, a permutation vector. The new vector `x[]` has the property that `x[y[i]] = y[x[i]] = i`; If `y[]` is not truly a permutation vector, an error message will be printed and the program exits.

12. `void IVinvPerm ( int n, int y[], int index[] ) ;`

    This method permutes the vector y as follows. i.e., `y[index[i]] := y[i]`. See `IVperm()` for a similar function.

13. `int IVlocateViaBinarySearch ( int n, int y[], int target ) ;`

    The `n` entries of `y[]` must be in nondecreasing order. If `target` is found in `y[]`, this method returns a location where `target` is found. If `target` is not in `y[]`, `-1` is returned.

14. `int IVmax ( int n, int y[], int *ploc ) ;`

    This method returns the maximum entry in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

15. `int IVmaxabs ( int n, int y[], int *ploc ) ;`

    This method returns the maximum magnitude of entries in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.
   ```

16. `int IVmin ( int n, int y[], int *ploc ) ;`

    This method returns the minimum entry in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

17. `int IVminabs ( int n, int y[], int *ploc ) ;`

    This method returns the minimum magnitude of entries in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

18. `void IVperm ( int n, int y[], int index[] ) ;`

    This method permutes the vector y as follows. i.e., `y[i] := y[index[i]]`. See `IVinvPerm()` for a similar function.

19. `void IVramp ( int n, int y[], int start, int inc ) ;`

    This method fills `n` entries in `y[]` with values `start`, `start + inc`, `start + 2*inc`, `start + 3*inc`, etc.

20. `void IVscatter ( int n, int y[], int index[], int x[] ) ;`

    This method scatters `n` entries of `x[]` into `y[]` as follows, `y[index[i]] = x[i]` for `0 <= i < n`.

21. `int IVsum ( int n, int y[] ) ;`

    This method returns the sum of the first `n` entries in the vector `x[]`, i.e., return $\sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{x[i]}$.

22. `int IVsumabs ( int n, int y[] ) ;`

    This method returns the sum of the absolute values of the first `n` entries in the vector `x[]`, i.e., return $\sum_{\mathtt{i=0}}^{\mathtt{n-1}} \mathtt{abs}(\mathtt{x[i]})$.

23. `void IVswap ( int n, int y[], int x[] ) ;`

    This method swaps the `x[]` and `y[]` vectors as follows. i.e., `y[i] := x[i]` and `x[i] := y[i]` for `0 <= i < n`.

24. `void IVzero ( int n, int y[] ) ;`

    This method zeroes `n` entries in `y[]`, i.e., `y[i] = 0` for `0 <= i < n`.

25. `void IVshuffle ( int n, int y[], int seed ) ;`

    This method shuffles the first `n` entries in `y[]`. The value `seed` is the seed to a random number generator, and one can get repeatable behavior by repeating `seed`.

13.2.5 FV : float vector methods

1. `float * FVinit ( int n, float val ) ;`

   This is the allocator and initializer method for `float` vectors. Storage for an array with size `n` is found and each entry is filled with `val`. A pointer to the array is returned.

2. `float * FVinit2 ( int n ) ;`

   This is an allocator method for `float` vectors. Storage for an array with size `n` is found. A pointer to the array is returned. Note, on return, there will likely be garbage in the array.

3. `void FVfree ( int vec[] ) ;`

   This method releases the storage taken by `vec[]`.

4. `void FVfprintf ( FILE *fp, int n, float y[] ) ;`

   This method prints `n` entries in `y[]` to file `fp`. The format is new line followed by lines of six `float`'s in `" %12.4e"` format.

5. `int FVfscanf ( FILE *fp, int n, float y[] ) ;`

   This method scans in `float`'s from file `fp` and places them in the array `y[]`. It tries to read in `n` `float`'s, and returns the number that were actually read.

6. `void FVadd ( int n, float y[], float x[] ) ;`

   This method adds `n` entries from `x[]` to `y[]`, i.e., `y[i] += x[i]` for `0 <= i < n`.

7. `void FVaxpy ( int n, float y[], float alpha, float x[] ) ;`

   This method adds a scaled multiple of `n` entries from `x[]` into `y[]`, i.e., `y[i] += alpha * x[i]` for `0 <= i < n`.

8. `void FVaxpyi ( int n, float y[], int index[], float alpha, float x[] ) ;`

   This method scatteradds a scaled multiple of `n` entries from `x[]` into `y[]`, i.e., `y[index[i]] += alpha * x[i]` for `0 <= i < n`.

9. ```
   void FVcompress ( int n1, double x1[], double y1[],
                     int n2, double x2[], double y2[] ) ;
   ```

   Given a pair of arrays `x1[n1]` and `y1[n1]`, fill `x2[n2]` and `y2[n2]` with a subset of the `(x1[j],y1[j]` entries whose distribution is an approximation.

10. `void FVcopy ( int n, float y[], float x[] ) ;`

    This method copies `n` entries from `x[]` to `y[]`, i.e., `y[i] = x[i]` for `0 <= i < n`.

11. `float FVdot ( int n, float y[], float x[] ) ;`

    This method returns the dot product of the vector `x[]` and `y[]`, i.e., return $\sum_{\mathtt{i=0}}^{\mathtt{n-1}}(\mathtt{x[i]} * \mathtt{y[i]})$.

12. `void FVfill ( int n, float y[], float val ) ;`

    This method fills `n` entries in `y[]` with `val`, i.e., `y[i] = val` for `0 <= i < n`.

13. `void FVgather ( int n, float y[], float x[], int index[] ) ;`

    `y[i] = x[index[i]]` for `0 <= i < n`.

14. `void FVgatherAddZero ( int n, float y[], float x[], int index[] ) ;`

    `y[i] += x[index[i]]` and `x[index[i]] = 0` for `0 <= i < n`.

15. `void FVgatherZero ( int n, float y[], float x[], int index[] ) ;`

    `y[i] = x[index[i]]` and `x[index[i]] = 0`

16. `void FVinvPerm ( int n, float y[], int index[] ) ;`

    This method permutes the vector y as follows. i.e., `y[index[i]] := y[i]`. See `FVperm()` for a similar function.

17. `float FVmax ( int n, float y[], int *ploc ) ;`

    This method returns the maximum entry in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.
   ```

18. `float FVmaxabs ( int n, float y[], int *ploc ) ;`

    This method returns the maximum magnitude of entries in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

19. `float FVmin ( int n, float y[], int *ploc ) ;`

    This method returns the minimum entry in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

20. `float FVminabs ( int n, float y[], int *ploc ) ;`

    This method returns the minimum magnitude of entries in `y[0:n-1]` and puts the first location where it was found into the address `ploc`.

21. `void FVperm ( int n, float y[], int index[] ) ;`

    This method permutes the vector y as follows. i.e., `y[i] := y[index[i]]`. See `FVinvPerm()` for a similar function.

22. `void FVramp ( int n, float y[], float start, float inc ) ;`

    This method fills `n` entries in `y[]` with values `start`, `start + inc`, `start + 2*inc`, `start + 3*inc`, etc.

23. `void FVscale ( int n, float y[], float alpha ) ;`

    This method scales a vector `y[]` by `alpha`, i.e., `y[i] *= alpha`. for `0 <= i < n`.

24. `void FVscatter ( int n, float y[], int index[], float x[] ) ;`

    This method scatters `n` entries of `x[]` into `y[]` as follows, `y[index[i]] = x[i]` for `0 <= i < n`.

25. `void FVscatterAddZero ( int n, float y[], int index[], float x[] ) ;`

    This method scatters/adds `n` entries of `x[]` into `y[]` as follows, `y[index[i]] += x[i]` and `x[i]` for `0 <= i < n`.

26. `void FVscatterZero ( int n, float y[], int index[], float x[] ) ;`

    This method scatters `n` entries of `x[]` into `y[]` as follows, `y[index[i]] = x[i]` and `x[i]` for `0 <= i < n`.

27. `void FVsub ( int n, float y[], float x[] ) ;`

    This method subtracts `n` entries from `x[]` to `y[]`, i.e., `y[i] -= x[i]` for `0 <= i < n`.

28. `float FVsum ( int n, float y[] ) ;`

    This method returns the sum of the first `n` entries in the vector `x[]`, i.e., return $\sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{x}[\mathtt{i}]$.

29. `float FVsumabs ( int n, float y[] ) ;`

    This method returns the sum of the absolute values of the first `n` entries in the vector `x[]`, i.e., return $\sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{abs}(\mathtt{x}[\mathtt{i}])$.

30. `void FVswap ( int n, float y[], float x[] ) ;`

    This method swaps the `x[]` and `y[]` vectors as follows. i.e., `y[i] := x[i]` and `x[i] := y[i]` for `0 <= i < n`.

31. `void FVzero ( int n, float y[] ) ;`

    This method zeroes `n` entries in `y[]`, i.e., `y[i] = 0` for `0 <= i < n`.

32. `void FVshuffle ( int n, float y[], int seed ) ;`

    This method shuffles the first `n` entries in `y[]`. The value `seed` is the seed to a random number generator, and one can get repeatable behavior by repeating `seed`.

13.2.6 PCV : char * vector methods

1. `char ** PCVinit ( int n ) ;`

   This is the allocator and initializer method for `char*` vectors. Storage for an array with size `n` is found and each entry is filled with `NULL`. A pointer to the array is returned.

2. `void PCVfree ( char **p_vec ) ;`

   This method releases the storage taken by `p_vec[]`.

3. `void PCVcopy ( int n, char *p_y[], char *p_x[] ) ;`

   This method copies `n` entries from `p_x[]` to `p_y[]`, i.e., `p_y[i] = p_x[i]` for `0 <= i < n`.

4. `void PCVsetup ( int n, int sizes[], char vec[], char *p_vec[] ) ;`

   This method sets the entries of `p_vec[]` as pointers into `vec[]` given by the `sizes[]` vector, i.e., `p_vec[0] = vec`, and `p_vec[i] = p_vec[i-1] + sizes[i-1]` for `0 < i < n`.

13.2.7 PDV : double * vector methods

1. `double ** PDVinit ( int n ) ;`

   This is the allocator and initializer method for `double*` vectors. Storage for an array with size `n` is found and each entry is filled with `NULL`. A pointer to the array is returned.

2. `void PDVfree ( double **p_vec ) ;`

   This method releases the storage taken by `p_vec[]`.

3. `void PDVcopy ( int n, double *p_y[], double *p_x[] ) ;`

   This method copies `n` entries from `p_x[]` to `p_y[]`, i.e., `p_y[i] = p_x[i]` for `0 <= i < n`.

4. `void PDVsetup ( int n, int sizes[], double vec[], double *p_vec[] ) ;`

   This method sets the entries of `p_vec[]` as pointers into `vec[]` given by the `sizes[]` vector, i.e., `p_vec[0] = vec`, and `p_vec[i] = p_vec[i-1] + sizes[i-1]` for `0 < i < n`.

PIV : int * vector methods

1. `int ** PIVinit ( int n ) ;`

   This is the allocator and initializer method for `int*` vectors. Storage for an array with size `n` is found and each entry is filled with `NULL`. A pointer to the array is returned.

2. `void PIVfree ( int **p_vec ) ;`

   This method releases the storage taken by `p_vec[]`.

3. `void PIVcopy ( int n, int *p_y[], int *p_x[] ) ;`

   This method copies `n` entries from `p_x[]` to `p_y[]`, i.e., `p_y[i] = p_x[i]` for `0 <= i < n`.

4. `void PIVsetup ( int n, int sizes[], int vec[], int *p_vec[] ) ;`

   This method sets the entries of `p_vec[]` as pointers into `vec[]` given by the `sizes[]` vector, i.e., `p_vec[0] = vec`, and `p_vec[i] = p_vec[i-1] + sizes[i-1]` for `0 < i < n`.

13.2.8 PFV : float * vector methods

1. `float ** PFVinit ( int n ) ;`

   This is the allocator and initializer method for `float*` vectors. Storage for an array with size `n` is found and each entry is filled with `NULL`. A pointer to the array is returned.

2. `void PFVfree ( float **p_vec ) ;`

   This method releases the storage taken by `p_vec[]`.

3. `void PFVcopy ( int n, float *p_y[], float *p_x[] ) ;`

   This method copies `n` entries from `p_x[]` to `p_y[]`, i.e., `p_y[i] = p_x[i]` for `0 <= i < n`.

4. `void PFVsetup ( int n, int sizes[], float vec[], float *p_vec[] ) ;`

   This method sets the entries of `p_vec[]` as pointers into `vec[]` given by the `sizes[]` vector, i.e., `p_vec[0] = vec`, and `p_vec[i] = p_vec[i-1] + sizes[i-1]` for `0 < i < n`.

13.2.9 Sorting routines

Validation routines

1. `int IVisascending ( int n, int ivec[] ) ;`
   `int IVisdescending ( int n, int ivec[] ) ;`

   These methods returns `1` if the array `ivec[]` is in ascending or descending order and returns `0` otherwise.

2. `int DVisascending ( int n, double dvec[] ) ;`
   `int DVisdescending ( int n, double dvec[] ) ;`

   These methods returns `1` if the array `dvec[]` is in ascending or descending order and returns `0` otherwise.

Insert sort routines

1. `void IVisortUp ( int n, int ivec[] ) ;`
   `void IVisortDown ( int n, int ivec[] ) ;`

   These methods sort an `int` array into ascending or descending order using an insertion sort.

2. `void IV2isortUp ( int n, int ivec1[], int ivec2[] ) ;`
   `void IV2isortDown ( int n, int ivec1[], int ivec2[] ) ;`

   These methods sort the array `ivec1[]` into ascending or descending order using an insertion sort and permutes the `int` companion array `ivec2[]` in the same fashion.

3. `void IVDVisortUp ( int n, int ivec[], double dvec[] ) ;`
   `void IVDVisortDown ( int n, int ivec[], double dvec[] ) ;`

   This sorts the array `ivec[]` into ascending or descending order using an insertion sort and permutes the companion array `dvec[]` in the same fashion.

4. `void IV2DVisortUp ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`
   `void IV2DVisortDown ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`

   These methods sort the array `ivec1[]` into ascending or descending order using an insertion sort and permutes the `int` and `double` companion array `ivec2[]` and `dvec[]` in the same fashion.

5. `void IVZVisortUp ( int n, int ivec[], double dvec[] ) ;`
   `void IVZVisortDown ( int n, int ivec[], double dvec[] ) ;`

   This sorts the array `ivec[]` into ascending or descending order using an insertion sort and permutes the double precision complex companion array `dvec[]` in the same fashion.

6. `void IV2ZVisortUp ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`
   `void IV2ZVisortDown ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`

   These methods sort the array `ivec1[]` into ascending or descending order using an insertion sort and permutes the companion arrays `ivec2[]` and `dvec[]` in the same fashion. The `dvec[]` array is double precision complex.

7. `void DVisortUp ( int n, double dvec[] ) ;`
   `void DVisortDown ( int n, double dvec[] ) ;`

   These methods sort a `double` array into ascending or descending order using an insertion sort.

8. `void DV2isortUp ( int n, double dvec1[], double dvec2[] ) ;`
   `void DV2isortDown ( int n, double dvec1[], double dvec2[] ) ;`

   These methods sort the array `dvec1[]` into ascending or descending order using an insertion sort and permutes the companion array `dvec2[]` in the same fashion.

9. `void DVIVisortUp ( int n, double dvec[], int ivec[] ) ;`
   `void DVIVisortDown ( int n, double dvec[], int ivec[] ) ;`

   These methods sort the array `dvec[]` into ascending or descending order using an insertion sort and permutes the companion array `ivec[]` in the same fashion.

Quicksort routines

1. `void IVqsortUp ( int n, int ivec[] ) ;`
   `void IVqsortDown ( int n, int ivec[] ) ;`

   These methods sort an `int` array into ascending or descending order using a quick sort.

2. `void IV2qsortUp ( int n, int ivec1[], int ivec2[] ) ;`
   `void IV2qsortDown ( int n, int ivec1[], int ivec2[] ) ;`

   These methods sort the array `ivec1[]` into ascending or descending order using a quick sort and permutes the companion array `ivec2[]` in the same fashion.

3. `void IVDVqsortUp ( int n, int ivec[], double dvec[] ) ;`
   `void IVDVqsortDown ( int n, int ivec[], double dvec[] ) ;`

   These methods sort the array `ivec[]` into ascending or descending order using a quick sort and permutes the companion array `dvec[]` in the same fashion.

4. `void IV2DVqsortUp ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`
   `void IV2DVqsortDown ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`

   These methods sort the array `ivec1[]` into ascending or descending order using a quick sort and permutes the companion arrays `ivec2[]` and `dvec[]` in the same fashion.

5. `void IVZVqsortUp ( int n, int ivec[], double dvec[] ) ;`
   `void IVZVqsortDown ( int n, int ivec[], double dvec[] ) ;`

   These methods sort the array `ivec[]` into ascending or descending order using a quick sort and permutes the double precision complex companion array `dvec[]` in the same fashion.

6. `void IV2ZVqsortUp ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`
   `void IV2ZVqsortDown ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`

   These methods sort the array `ivec1[]` into ascending or descending order using a quick sort and permutes the companion arrays `ivec2[]` and `dvec[]` in the same fashion. The `dvec[]` array is double precision complex.

7. `void DVqsortUp ( int n, double dvec[] ) ;`
   `void DVqsortDown ( int n, double dvec[] ) ;`

   Thes methods sort a `double` array into ascending or descending order using a quick sort.

8. `void DV2qsortUp ( int n, double dvec1[], double dvec2[] ) ;`
   `void DV2qsortDown ( int n, double dvec1[], double dvec2[] ) ;`

   These methods sort the array `dvec1[]` into ascending or descending order using a quick sort and permutes the companion array `dvec2[]` in the same fashion.

9. `void DVIVqsortUp ( int n, double dvec[], int ivec[] ) ;`
   `void DVIVqsortDown ( int n, double dvec[], int ivec[] ) ;`

   These methods sort the array `dvec[]` into ascending or descending order using a quick sort and permutes the companion array `ivec[]` in the same fashion.

13.2.10 Sort and compress routines

1. `int IVsortUpAndCompress ( int n, int ivec[] ) ;`

   This method sorts `ivec[]` into ascending order, and removes (compresses) any duplicate entries. The return value is the number of unique entries stored in the leading locations of the vector `ivec[]`.

   Error checking: If `n < 0` or `ivec` is `NULL`, an error message is printed and the program exits.

2. `int IVDVsortUpAndCompress ( int n, int ivec[], double dvec[] ) ;`

   This method sorts `ivec[]` into ascending order with `dvec[]` as a companion vector. It then compresses the pairs, adding the `dvec[]` entries together when their `ivec[]` values are identical. The return value is the number of unique entries stored in the leading locations of the vectors `ivec[]` and `dvec[]`.

   Error checking: If `n < 0`, or if `ivec` or `dvec` is `NULL`, an error message is printed and the program exits.

3. `int IVZVsortUpAndCompress ( int n, int ivec[], double dvec[] ) ;`

   This method sorts `ivec[]` into ascending order with the double precision complex `dvec[]` companion vector. It then compresses the pairs, adding the complex `dvec[]` entries together when their `ivec[]` values are identical. The return value is the number of unique entries stored in the leading locations of the vectors `ivec[]` and `dvec[]`.

   Error checking: If `n < 0`, or if `ivec` or `dvec` is `NULL`, an error message is printed and the program exits.

4. `int IV2sortUpAndCompress ( int n, int ivec1[], int ivec2[] ) ;`

   This method sorts `ivec1[]` into ascending order with `ivec2[]` as a companion vector. It then compresses the pairs, dropping all but one of identical pairs. The return value is the number of unique entries stored in the leading locations of the vectors `ivec1[]` and `ivec2[]`.

   Error checking: If `n < 0`, or if `ivec1` or `ivec2` is `NULL`, an error message is printed and the program exits.

5. `int IV2DVsortUpAndCompress ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`

   This method sorts `ivec1[]` into ascending order with `ivec2[]` and `dvec[]` as companion vectors. It then compresses the pairs, summing the `dvec[]` entries for identical `(ivec1[], ivec2[])` pairs. The return value is the number of unique entries stored in the leading locations of the vectors `ivec1[]`, `ivec2[]` and `dvec[]`.

   Error checking: If `n < 0`, or if `ivec1`, `ivec2` or `dvec` is `NULL`, an error message is printed and the program exits.

6. `int IV2ZVsortUpAndCompress ( int n, int ivec1[], int ivec2[], double dvec[] ) ;`

   This method sorts `ivec1[]` into ascending order with `ivec2[]` and the double precision `dvec[]` as companion vectors. It then compresses the pairs, summing the complex `dvec[]` entries for identical `(ivec1[], ivec2[])` pairs. The return value is the number of unique entries stored in the leading locations of the vectors `ivec1[]`, `ivec2[]` and `dvec[]`.

   Error checking: If `n < 0`, or if `ivec1`, `ivec2` or `dvec` is `NULL`, an error message is printed and the program exits.

13.2.11 `IP : (int, pointer)` singly linked-list methods

```
typedef struct _IP IP ;
struct _IP {
   int   val   ;
   IP    *next ;
} ;
```

1. `IP * IP_init ( int n, int flag ) ;`

   This is the allocator and initializer method for a vector of `(int,pointer)` structures. Storage for an array with size `n` is found. A pointer to an array `ips[]` is returned with `ips[i].val = 0` for `0 <= i < n`. The `flag` parameter determines how the `next` field is filled.

   - If `flag = 0`, the elements are not linked, i.e., `ips[i].next = NULL` for `0 <= i < n`.
   - If `flag = 1`, the elements are linked in a forward manner, i.e., `ips[i].next = &ips[i+1]` for `0 <= i < n-1` and `ips[n-1].next = NULL`.
   - If `flag = 2`, the elements are linked in a backward manner, i.e., `ips[i].next = &ips[i-1]` for `0 < i < n` and `ips[0].next = NULL`.

2. `void IP_free ( IP *ip ) ;`

   This method releases the storage based at `*ip`.

3. `void IP_fprintf ( FILE *fp, IP *ip ) ;`

   This method prints the singly linked list that starts with `ip`.

4. `int IP_fp80 ( FILE *fp, int n, int y[], int column, int *pierr ) ;`

   This method prints the singly linked list that starts with `ip`. See `IVfp80()` for a description of how the entries are placed on a line.

5. `IP * IP_mergeUp ( IP *ip1, IP *ip2 ) ;`

   This method merges two singly linked lists into one. If the two lists are in ascending order, the new list is also in ascending order. The head of the new list is returned.

6. `IP * IP_mergeSortUp ( IP *ip ) ;`

   This method sorts a list into ascending order using a merge sort.

7. `IP * IP_radixSortUp ( IP *ip ) ;`

   This method sorts a list into ascending order using a radix sort.

8. `IP * IP_radixSortDown ( IP *ip ) ;`

   This method sorts a list into descending order using a radix sort.

13.2.12 `I2OP : (int, int, void*, pointer)` singly linked-list methods

```
typedef struct _I2OP I2OP ;
struct _I2OP {
   int    value0 ;
   int    value1 ;
   void   value2 ;
   I2OP   *next  ;
} ;
```

1. `I2OP * I2OP_init ( int n, int flag ) ;`

   This is the allocator and initializer method for a vector of `I2OP` structures. Storage for an array with size `n` is found. A pointer to an array `ips[]` is returned with `ips[i].val = 0` for `0 <= i < n`. The `flag` parameter determines how the `next` field is filled.

   - If `flag = I2OP_NULL`, the elements are not linked, i.e., `ips[i].next = NULL` for `0 <= i < n`.
   - If `flag = I2OP_FORWARD`, the elements are linked in a forward manner, i.e., `ips[i].next = &ips[i+1]` for `0 <= i < n-1` and `ips[n-1].next = NULL`.
   - If `flag = I2OP_BACKWARD`, the elements are linked in a backward manner, i.e., `ips[i].next = &ips[i-1]` for `0 < i < n` and `ips[0].next = NULL`.

2. `I2OP * I2OP_initStorage ( int n, int flag, I2OP *base ) ;`

   This is an initializer method for a vector of `I2OP` structures. We set `base[i].value0 = base[i].value1 = -1`. The `flag` parameter determines how the `next` field is filled.

   - If `flag = I2OP_NULL`, the elements are not linked, i.e., `ips[i].next = NULL` for `0 <= i < n`.
   - If `flag = I2OP_FORWARD`, the elements are linked in a forward manner, i.e., `ips[i].next = &ips[i+1]` for `0 <= i < n-1` and `ips[n-1].next = NULL`.
   - If `flag = I2OP_BACKWARD`, the elements are linked in a backward manner, i.e., `ips[i].next = &ips[i-1]` for `0 < i < n` and `ips[0].next = NULL`.

3. `void I2OP_free ( I2OP *i2op ) ;`

   This method releases the storage based at `*i2op`.

4. `void I2OP_fprintf ( FILE *fp, I2OP *i2op ) ;`

   This method prints the singly linked list that starts with `i2op`.

13.3 Driver programs

1. `test_sort msglvl msgFile target sortType n range mod seed`

   This driver program tests the sort methods. Use the script file `do_test_sort` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `target` parameter denotes the type of vector(s) to be sorted.
     - `IV` — `int` vector sort
     - `IV2` — `(int, int)` vector sort
     - `IVDV` — `(int, double)` vector sort
     - `IV2DV` — `(int, int, double)` vector sort
     - `IVZV` — `(int, complex)` vector sort
     - `IV2ZV` — `(int, int, complex)` vector sort
     - `DV` — `double` vector sort
     - `DV2` — `(double, double)` vector sort
     - `DVIV` — `(double, int)` vector sort
   - The `sortType` parameter denotes the type of sort.
     - `IU` — ascending insert sort
     - `ID` — descending insert sort
     - `QU` — ascending quick sort
     - `QD` — descending quick sort
   - The `n` parameter is the length of the vector(s).
   - Integer entries are of the form `k` mod `mod`, where `k` in `[0,range]`.
   - The `seed` parameter is a random number seed.

2. `test_sortUpAndCompress msglvl msgFile target n range mod seed`

   This driver program tests the "sort in ascending order and compress" methods. Use the script file `do_test_sortUpAndCompress` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `target` parameter denotes the type of vector(s) to be sorted.
     - `IV` — `int` vector sort
     - `IV2` — `(int, int)` vector sort
     - `IVDV` — `(int, double)` vector sort
     - `IV2DV` — `(int, int, double)` vector sort
     - `IVZV` — `(int, complex)` vector sort
     - `IV2ZV` — `(int, int, complex)` vector sort
   - The `n` parameter is the length of the vector(s).
   - Integer entries are of the form `k` mod `mod`, where `k` in `[0,range]`.
   - The `seed` parameter is a random number seed.

Chapter 14

ZV: Double Complex Vector Object

The ZV object is a wrapper around a double precision complex vector. In Fortran's LINPACK and LAPACK libraries, a leading Z denotes double precision complex, and we have followed this convention. The driving force for its creation of this object is more convenience than performance. There are three cases that led to its development.

- Often a method will create a vector (allocate storage for and fill the entries) whose size is not known before the method call. Instead of having a pointer to int and a pointer to double* in the calling sequence, we can return a pointer to an ZV object that contains the newly created vector and its size.
- In many cases we need a persistent double vector object, and file IO is simplified if we have an object to deal with. The filename is of the form *.zvf for a formatted file or *.zvb for a binary file.
- Prototyping can go much faster with this object as opposed to working with an double array. Consider the case when one wants to accumulate a list of doubles, but one doesn't know how large the list will be. The method ZV_setSize() can be used to set the size of the vector to zero. Another method ZV_push() appends an element to the vector, growing the storage if necessary.
- Sometimes an object needs to change its size, i.e., vectors need to grow or shrink. It is easier and more robust to tell an ZV object to resize itself (see the ZV_setSize() and ZV_setMaxsize() methods) than it is to duplicate code to work on an double vector.

One must choose where to use this object. There is a substantial performance penalty for doing the simplest operations, and so when we need to manipulate an double vector inside a loop, we extract out the size and pointer to the base array from the ZV object. On the other hand, the convenience makes it a widely used object.

14.1 Data Structure

The ZV structure has three fields.

- int size : present size of the vector.
- int maxsize : maximum size of the vector.
- int owned : owner flag for the data. When owned = 1, storage for owned double's has been allocated by this object and can be free'd by the object. When owned == 0 but size > 0 , this object points to entries that have been allocated elsewhere, and these entries will not be free'd by this object.

- `double *vec` : pointer to the base address of the *double* vector

The `size`, `maxsize`, `nowned` and `vec` fields need never be accessed directly — see the `ZV_size()`, `ZV_maxsize()`, `ZV_owned()`, `ZV_entries()`, `ZV_sizeAndEntries()` methods.

14.2 Prototypes and descriptions of ZV methods

This section contains brief descriptions including prototypes of all methods that belong to the `ZV` object.

14.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `ZV * ZV_new ( void ) ;`

   This method simply allocates storage for the `ZV` structure and then sets the default fields by a call to `ZV_setDefaultFields()`.

2. `void ZV_setDefaultFields ( ZV *zv ) ;`

   This method sets the default fields of the object, `size = maxsize = owned = 0` and `vec = NULL`.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

3. `void ZV_clearData ( ZV *zv ) ;`

   This method releases any data owned by the object. If `vec` is not `NULL` and `owned = 1`, then the storage for `vec` is free'd by a call to `ZVfree()`. The structure's default fields are then set with a call to `ZV_setDefaultFields()`.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

4. `void ZV_free ( ZV *zv ) ;`

   This method releases any storage by a call to `ZV_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

14.2.2 Instance methods

These method allow access to information in the data fields without explicitly following pointers. There is overhead involved with these method due to the function call and error checking inside the methods.

1. `int ZV_owned ( ZV *zv ) ;`

   This method returns the value of `owned`. If `owned > 0`, then the object owns the data pointed to by `vec` and will free this data with a call to `ZVfree()` when its data is cleared by a call to `ZV_free()` or `ZV_clearData()`.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

2. `int ZV_size ( ZV *zv ) ;`

   This method returns the value of `size`, the present size of the vector.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

3. `int ZV_maxsize ( ZV *zv ) ;`

   This method returns the value of `size`, the maximum size of the vector.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

4. `void ZV_entry ( ZV *zv, int loc, double *pReal, double *pImag ) ;`

   This method fills `*pReal` with the real part and `*pImag` with the imaginary part of the `loc`'th entry in the vector. If `loc < 0` or `loc >= size`, i.e., if the location is out of range, we return `0.0`. This design `feature` is handy when a list terminates with a `0.0` value.

   Error checking: If `zv`, `pReal` or `pImag` is `NULL`, an error message is printed and the program exits.

5. `void ZV_pointersToEntry ( ZV *zv, int loc, double **ppReal, double **ppImag ) ;`

   This method fills `**ppReal` with a pointer to the real part and `**ppImag` with a pointer to the imaginary part of the `loc`'th entry in the vector. If `loc < 0` or `loc >= size`, i.e., if the location is out of range, we return `0.0`. This design `feature` is handy when a list terminates with a `0.0` value.

   Error checking: If `zv`, `pReal` or `pImag` is `NULL`, an error message is printed and the program exits.

6. `double * ZV_entries ( ZV *zv ) ;`

   This method returns `vec`, a pointer to the base address of the vector.

   Error checking: If `zv` is `NULL`, an error message is printed and the program exits.

7. `void ZV_sizeAndEntries ( ZV *zv, int *psize, double **pentries ) ;`

   This method fills `*psize` with the size of the vector and `**pentries` with the base address of the vector.

   Error checking: If `zv`, `psize` or `pentries` is `NULL`, an error message is printed and the program exits.

8. `void ZV_setEntry ( ZV *zv, int loc, double real, double imag ) ;`

   This method sets the `loc`'th entry of the vector to `(real,imag)`.

   Error checking: If `zv` is `NULL` or `loc < 0`, an error message is printed and the program exits.

14.2.3 Initializer methods

There are three initializer methods.

1. `void ZV_init ( ZV *zv, int size, double *entries ) ;`

   This method initializes the object given a size for the vector and a possible pointer to the vectors' storage. Any previous data is cleared with a call to `ZV_clearData()`. If `entries != NULL` then the `vec` field is set to `entries`, the `size` and `maxsize` fields are set to `size`, and `owned` is set to zero because the object does not own the entries. If `entries` is `NULL` and `size > 0` then a vector is allocated by the object, and the object owns this storage.

   Error checking: If `zv` is `NULL` or `size < 0`, an error message is printed and the program exits.

2. `void ZV_init1 ( ZV *zv, int size ) ;`

   This method initializes the object given a size size for the vector via a call to `ZV_init()`.

   Error checking: Error checking is done with the call to $\widehat{\text{Z}}$V_init().

3. `void ZV_init2 ( ZV *zv, int size, int maxsize, int owned, double *vec ) ;`

   This is the total initialization method. The data is cleared with a call to `ZV_clearData()`. If `vec` is `NULL`, the object is initialized via a call to `ZV_init1()`. Otherwise, the objects remaining fields are set to the input parameters. and if `owned` is not `1`, the data is not owned, so the object cannot grow.

   Error checking: If `zv` is `NULL`, or if `size < 0`, or if `maxsize < size`, or if `owned` is not equal to `0` or `1`, of if `owned = 1` and `vec = NULL`, an error message is printed and the program exits.

4. `void ZV_setMaxsize ( ZV *zv, int newmaxsize ) ;`

   This method sets the maximum size of the vector. If `maxsize`, the present maximum size of the vector, is not equal to `newmaxsize`, then new storage is allocated. Only `size` entries of the old data are copied into the new storage, so if `size > newmaxsize` then data will be lost. The `size` field is set to the minimum of `size` and `newmaxsize`.

   Error checking: If `zv` is `NULL` or `newmaxsize < 0`, or if `0 < maxsize` and `owned == 0`, an error message is printed and the program exits.

5. `void ZV_setSize ( ZV *zv, int newsize ) ;`

   This method sets the size of the vector. If `newsize > maxsize`, the length of the vector is increased with a call to `ZV_setMaxsize()`. The `size` field is set to `newsize`.

   Error checking: If `zv` is `NULL`, or `newsize < 0`, or if `0 < maxsize < newsize` and `owned = 0`, an error message is printed and the program exits.

14.2.4 Utility methods

1. `void ZV_shiftBase ( ZV *zv, int offset ) ;`

   This method shifts the base entries of the vector and decrements the present size and maximum size of the vector by `offset`. This is a dangerous method to use because the state of the vector is lost, namely `vec`, the base of the entries, is corrupted. If the object owns its entries and `ZV_free()`, `ZV_setSize()` or `ZV_setMaxsize()` is called before the base has been shifted back to its original position, a segmentation violation will likely result. This is a very useful method, but use with caution.

   Error checking: If `zv` is `NULL`, an error message is printed and the program exits.

2. `void ZV_push ( ZV *zv, double val ) ;`

   This method pushes an entry onto the vector. If the vector is full, i.e., if `size == maxsize - 1`, then the size of the vector is doubled if possible. If the storage cannot grow, i.e., if the object does not own its storage, an error message is printed and the program exits.

   Error checking: If `zv` is `NULL`, an error message is printed and the program exits.

3. `double ZV_minabs ( ZV *zv ) ;`
   `double ZV_maxabs ( ZV *zv ) ;`

   This method simply returns the minimum and maximum magnitudes of entries in the vector.

   Error checking: If `zv` is `NULL`, `size <= 0` or if `vec == NULL`, an error message is printed and the program exits.

4. `int ZV_sizeOf ( ZV *zv ) ;`

   This method returns the number of bytes taken by the object.

   Error checking: If `zv` is `NULL` an error message is printed and the program exits.

5. `void ZV_fill ( ZV *zv, double real, double imag ) ;`

   This method fills the vector with a scalar value.

   Error checking: If `zv` is `NULL`, an error message is printed and the program exits.

6. `void ZV_zero ( ZV *zv ) ;`

   This method fills the vector with zeros.

   Error checking: If `zv` is `NULL`, an error message is printed and the program exits.

7. `void ZV_copy ( ZV *zv1, ZV *zv2 ) ;`

   This method fills the `zv1` object with entries in the `iv2` object. Note, this is a *mapped* copy, `zv1` and `zv2` need not have the same size. The number of entries that are copied is the smaller of the two sizes.

   Error checking: If `zv1` or `zv2` is `NULL`, an error message is printed and the program exits.

8. ```
   void ZV_log10profile ( ZV *zv, int npts, DV *xDV, DV *yDV, double tausmall,
                          double taubig, int *pnzero, int *pnsmall, int *pnbig ) ;
   ```

   This method scans the entries in the `ZV` object and fills `xDV` and `yDV` with data that allows a simple $\log_{10}$ distribution plot. Only entries whose magnitudes lie in the range `[tausmall, taubig]` contribute to the distribution. The number of entries whose magnitudes are zero, smaller than `tausmall`, or larger than `taubig` are placed into `pnzero`, `*pnsmall` and `*pnbig`, respectively. On return, the size of the `xDV` and `yDV` objects is `npts`.

   Error checking: If `zv`, `xDV`, `yDV`, `pnsmall` or `pnbig` are `NULL`, or if `npts` ≤ 0, or if `taubig` < 0.0 or if `tausmall` $>$ `taubig`, an error message is printed and the program exits.

14.2.5 IO methods

There are the usual eight IO routines. The file structure of a `ZV` object is simple: the first entry is `size`, followed by the `size` entries found in `vec[]`.

1. `int ZV_readFromFile ( ZV *zv, char *fn ) ;`

   This method reads a `ZV` object from a file. It tries to open the file and if it is successful, it then calls `ZV_readFromFormattedFile()` or `ZV_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `zv` or `fn` are `NULL`, or if `fn` is not of the form `*.zvf` (for a formatted file) or `*.zvb` (for a binary file), an error message is printed and the method returns zero.

2. `int ZV_readFromFormattedFile ( ZV *zv, FILE *fp ) ;`

   This method reads in a `ZV` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.

3. `int ZV_readFromBinaryFile ( ZV *zv, FILE *fp ) ;`

   This method reads in a `ZV` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.
   ```

4. `int ZV_writeToFile ( ZV *zv, char *fn ) ;`

   This method writes a `ZV` object from a file. It tries to open the file and if it is successful, it then calls `ZV_writeFromFormattedFile()` or `ZV_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `zv` or `fn` are `NULL`, or if `fn` is not of the form `*.zvf` (for a formatted file) or `*.zvb` (for a binary file), an error message is printed and the method returns zero.

5. `int ZV_writeToFormattedFile ( ZV *zv, FILE *fp ) ;`

   This method writes a `ZV` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.

6. `int ZV_writeToBinaryFile ( ZV *zv, FILE *fp ) ;`

   This method writes a `ZV` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.

7. `int ZV_writeForHumanEye ( ZV *zv, FILE *fp ) ;`

   This method writes a `ZV` object to a file in a human readable format. is called to write out the header and statistics. The entries of the vector then follow in eighty column format using the `ZVfprintf()` method. The value `1` is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.

8. `int ZV_writeStats ( ZV *zv, FILE *fp ) ;`

   This method writes the header and statistics to a file. The value `1` is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.

9. `int ZV_writeForMatlab ( ZV *zv, char *name, FILE *fp ) ;`

   This method writes the entries of the vector to a file suitable to be read by Matlab. The character string `name` is the name of the vector, e.g, if `name = "A"`, then we have lines of the form

   ```
   A(1) = 1.000000000000e0 + 2.000000000000e0*i;
   A(2) = 3.463738459493e0 + 2.728482384840e0*i;
   ```

   for each entry in the vector. Note, the output indexing is 1-based, not 0-based. The value `1` is returned.

   Error checking: If `zv` or `fp` are `NULL`, an error message is printed and zero is returned.

14.3 Driver programs for the `ZV object`

1. `testIO msglvl msgFile inFile outFile`

   This driver program tests the `ZV` IO methods, and is useful for translating between the formatted `*.zvf` and binary `*.zvb` files.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the name of the file from which to read in the object. `inFile` must be of the form `*.zvf` for a formatted file or `*.zvb` for a binary file.

- The `outFile` parameter is the name of the file to which to write out the object. If `outfile` is of the form `*.zvf`, the object is written to a formatted file. If `outfile` is of the form `*.zvb`, the object is written to a binary file. When `outFile` is *not* `"none"`, the object is written to the file in a human readable format. When `outFile` is `"none"`, the object is not written out.

Part III

Ordering Objects and Methods

Chapter 15

BKL: Block Kernighan-Lin Object

Our BKL object is used to find an initial separator of a graph. Its input is a BPG bipartite graph object that represents the domain-segment graph of a domain decomposition of the graph. After a call to the BKL_fidmat() method, the object contains a two-color partition of the graph that is accessible via the colors[] and cweights[] vectors of the object.

15.1 Data Structure

The BKL object has the following fields.

- BPG *bpg : pointer to a BPG bipartite graph object, not owned by the BKL object.
- int ndom : number of domains, domain ids are in [0,ndom)
- int nseg : number of segments, segment ids are in [ndom,ndom + nseg)
- int nreg : number of regions, equal to ndom + nseg
- int totweight : total weight of the domains and segments
- int npass : number of Fiduccia-Mattheyes passes
- int npatch : number of patches evaluated, not used during the Fiduccia-Mattheyes algorithm
- int nflips : number of domains that were flipped
- int nimprove : number of improvements in the partition
- int ngaineval : number of gain evaluations, roughly equivalent to the number of degree evaluations in the minimum degree algorithm
- int *colors : pointer to an int vector of size nreg, colors[idom] is 1 or 2 for domain idom, colors[iseg] is 0, 1 or 2 for segment iseg.
- int *cweights : pointer to an int vector of size 3, cweights[0] contains the weight of the separator, cweights[1] and cweights[2] contains the weights of the two components
- int *regwghts : pointer to an int vector of size nreg, used to store the weights of the domains and segments

- `float alpha` : number used to store the partition evaluation parameter, the cost of the partition is

```
balance = max(cweights[1], cweights[2])/min(cweights[1], cweights[2]) ;
cost    = cweights[0]*(1. + alpha*balance) ;
```

15.2 Prototypes and descriptions of BKL methods

This section contains brief descriptions including prototypes of all methods that belong to the BKL object.

15.3 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `BKL * BKL_new ( void ) ;`

   This method simply allocates storage for the BKL structure and then sets the default fields by a call to `BKL_setDefaultFields()`.

2. `void BKL_setDefaultFields ( BKL *bkl ) ;`

   This method sets the fields of the structure to their default values: `bpg`, `colors` and `regwghts` are set to `NULL`, the `int` parameters are set to zero, and the `cweights` vector is filled with zeros.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

3. `void BKL_clearData ( BKL *bkl ) ;`

   This method clears any data allocated by the object, namely the `colors` and `regwghts` vectors. It then fills the structure's fields with default values with a call to `BKL_setDefaultFields()`.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

4. `void BKL_free ( BKL *bkl ) ;`

   This method releases any storage by a call to `BKL_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

15.3.1 Initializer methods

1. `void BKL_init ( BKL *bkl, BPG *bpg, float alpha ) ;`

   This method initializes the BKL object given a bipartite graph object and cost function parameter as input. Any previous data is cleared with a call to `BKL_clearData()`. The `ndom`, `nseg` and `nreg` scalars are set, the `regwghts[]` vector allocated and filled, and the `colors[]` vector allocated and filled with zeros.

   Error checking: If `bkl` or `bpg` is `NULL`, an error message is printed and the program exits.

15.3.2 Utility methods

1. `void BKL_setRandomColors ( BKL *bkl, int seed ) ;`

   If `seed > 0` a random number generator is set using `seed`. The domains are then colored `1` or `2` randomly and `BKL_setColorWeights()` is called to set the segment weights.

   Error checking: If `bkl` or `bkl->bpg` is `NULL`, an error message is printed and the program exits.

2. `void BKL_setColorWeights ( BKL *bkl ) ;`

   This method sets the color weights for the region. It assumes that all domains are colored `1` or `2`. The segments are then colored. If a segment is adjacent only to domains of one color, its color is that color, otherwise its color is `0`.

   Error checking: If `bkl` or `bkl->bpg` is `NULL`, an error message is printed and the program exits. The colors of the domains are checked to ensure they are `1` or `2`.

3. `int BKL_segColor ( BKL *bkl, int iseg ) ;`

   This method returns the color of segment `iseg`.

   Error checking: If `bkl` is `NULL`, or if `iseg` is not in `[bkl->ndom, bkl->nreg)`, an error message is printed and the program exits.

4. `void BKL_flipDomain ( BKL *bkl, int idom ) ;`

   This method flips the color of domain `idom`, adjusts the colors of neighboring segments and the `cweights[]` vector.

   Error checking: If `bkl` is `NULL`, or if `idom` is not in `[0,bkl->ndom)`, an error message is printed and the program exits.

5. `int BKL_greyCodeDomain ( BKL *bkl, int count ) ;`

   This method returns the next domain id in a grey code sequence, used to exhaustively search of a subspace of partitions defined by set of candidate domains to flip. The value `count` ranges from `1` to $2^{\texttt{ndom}}$.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

6. `float BKL_setInitPart ( BKL *bkl, int flag, int seed, int domcolors[] ) ;`

   This method sets the initial partition by coloring the domains and segments. The `flag` parameter has the following values.

   - `flag = 1` ⟶ random coloring of the domains
   - `flag = 2` ⟶ one black domain, (`seed` % `ndom`), rest are white
   - `flag = 3` ⟶ one black pseudoperipheral domain, found using domain (`seed` % `ndom`) as root, rest are white
   - `flag = 4` ⟶ roughly half-half split, breadth first search of domains, (`seed` % `ndom`) as root
   - `flag = 5` ⟶ roughly half-half split, breadth first search of domains, (`seed` % `ndom`) as root to find a pseudoperipheral domain as root
   - `flag = 6` ⟶ use `domcolors[]` to seed the `colors[]` array

   The `seed` input parameter is for a random number generator. The `domcolors[]` input array is used only for `flag = 6`.

   Error checking: If `bkl` is `NULL`, or if `flag = 6` and `domcolors` is `NULL`, or if `flag` is not in `[1,6]`, an error message is printed and the program exits.

7. `int BKL_domAdjToSep ( BKL *bkl, int dom ) ;`

   This method returns `1` if domain `dom` is adjacent to the separator and `0` otherwise.

   Error checking: If `bkl` is `NULL`, or if `dom` is not in `[0,ndom)`, an error message is printed and the program exits.

15.3.3 Partition evaluation methods

There are three functions that evaluate the cost of a partition.

1. `void BKL_evalgain ( BKL *bkl, int dom, int *pdeltaS, int *pdeltaB, int *pdeltaW ) ;`

   This method evaluates the change in the components ΔS, ΔB and ΔW that would occur if domain `dom` were to be flipped. These *gain* values are put into the storage pointed to by `pdeltaS`, `pdeltaB` and `pdeltaW`. The method checks that `bkl`, `pdeltaS`, `pdeltaB` and `pdeltaW` are not `NULL` and that `idom` is in `[0,bkl->ndom)`.

   Error checking: If `bkl`, `pdeltaS`, `pdeltaB` or `pdeltaW` is `NULL`, or if `dom` is not in `[0,ndom)`, an error message is printed and the program exits.

2. `float BKL_evalfcn ( BKL *bkl ) ;`

   The $|S|$, $|B|$ and $|W|$ values are taken from the `cweights[]` vector. If $\min(|B|,|W|) > 0$, this function returns
   $$|S|\left(1+\alpha*\frac{\max(|B|,|W|)}{\min(|B|,|W|)}\right),$$
   otherwise it returns $(|S|+|B|+|W|)^2$.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

3. `float BKL_eval ( BKL *bkl, int Sweight, int Bweight, int Wweight ) ;`

   The $|S|$, $|B|$ and $|W|$ values are taken from the `Sweight`, `Bweight` and `Wweight` parameters. If $\min(|B|,|W|) > 0$, this function returns
   $$|S|\left(1+\alpha*\frac{\max(|B|,|W|)}{\min(|B|,|W|)}\right),$$
   otherwise it returns $(|S|+|B|+|W|)^2$. The method checks that `bkl` is not `NULL`.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

15.3.4 Partition improvement methods

There are two functions that take a given partition and some input parameters and return a (hopefully) improved partition.

1. `float BKL_exhSearch ( BKL *bkl, int mdom, int domids[], int tcolors[] ) ;`

   This method performs an exhaustive search of a subspace of partitions and returns the best partition. The starting partition is given by the `BKL` object's `colors[]` vector. The subspace of domains to flip is defined by the `domids[mdom]` vector. The `tcolors[]` vector is a work vector. There are $2^{\texttt{mdom}}$ distinct partitions in the subspace to be explored. We flip the domains using a grey code sequence so a total of $2^{\texttt{mdom}}$ domain flips are performed. The `bkl->colors[]` vector is filled with the colors of the best partition and its cost is returned.

   Error checking: If `bkl`, `domids` or `tcolors` is `NULL`, or if `mdom < 1`, an error message is printed and the program exits.

2. `float BKL_fidmat ( BKL *bkl ) ;`

   If the number of domains is eight or less, an exhaustive search is made. Otherwise, this method finds a good partition using a variant of the Fiduccia-Mattheyes algorithm. At any step, only the domains that are adjacent to the separator are eligible to be flipped. For each eligible domain, we maintain ΔS, ΔB and ΔW, the change in the three component weights if this domain were to be flipped. These values must be updated whenever a neighboring domain has been flipped, and so is *local* information. The cost of the partition that would result if a domain were to be flipped is a function of the local information ΔS, ΔB and ΔW, as well as the present weights of the components (global information). At each step we evaluate the cost of the resulting partition for each domain that is eligible to be flipped. This is relatively expensive when compared to using a heap to contain ΔS for each domain, but we have found the resulting partitions to be better. The eligible domains are kept on a doubly linked list to allow easy insertions and deletions.

   Error checking: If `bkl` is `NULL`, an error message is printed and the program exits.

Chapter 16

BPG: Bipartite Graph Object

The `BPG` object is used to represent a bipartite graph. A bipartite graph naturally *is-a* graph, but since we are working in C, without inheritance, we have chosen to use the *has-a* relationship, i.e., our `BPG` bipartite graph object *has-a* `Graph` object inside itself.

A bipartite graph is a triple $H = (X, Y, E)$ where X and Y are two disjoint sets of vertices and the edge set E is a subset of $X \times Y$. In other words, nodes in X are adjacent to node in Y, but no edge connects two vertices in X or two vertices in Y.

We do not support bipartite graphs that are *subgraphs* of other bipartite graphs (in the sense that there are `Graph` objects that are subgraphs of other `Graph` objects) because we haven't found any reason to do so.

This bipartite graph object is very rudimentary. We have used it in two instances.

- Given a domain decomposition of a graph, we want to find a bisector of the graph that is a subset of the interface vertices. To do this we construct a bipartite graph such that the X nodes are the domains and the Y nodes are the segments (a partition of the interface vertices). We then apply a variant of the Kernighan-Lin algorithm to find an edge separator that is a subset of the segments. (Details are found in [5].)
- Given a 2-set partition of a graph $[S, B, W]$ where S is the separator and B and W are the two components, we want to find an improved partition $[\widehat{S}, \widehat{B}, \widehat{W}]$. One way to do this is to construct a bipartite graph where $X = S$ and $Y = Adj(S) \cap B$ or $Y = Adj(S) \cap W$ and the edge set E is constructed naturally from the appropriate edges in the graph. We then find the Dulmage-Mendelsohn decomposition of this bipartite graph to look for a better 2-set partition. (Details are found in [6].)

Our bipartite graph object illustrates software in evolution. In both cases, our desired output is a separator and the problem can be formulated as a bipartite graph. Does the *data* (the bipartite graph) *own* the *process* (the Kernighan-Lin algorithm or the Dulmage-Mendelsohn decomposition)? Or does the process operate on the data? There is no cut and dried answer. In fact, we did it both ways.

To find a separator from a domain decomposition, we took the approach that the process works on the data. (See the `BKL` block Kernighan-Lin object.) The process was sufficiently involved that soon the `BKL` code for the process outweighed (outline'd?) the `BPG` code for the data. Now if someone wants to modify (and hopefully improve) the Kernighan-Lin process, they won't alter the behavior of the bipartite graph object.

Finding the Dulmage-Mendelsohn decomposition of a bipartite graph is a little less clear cut. When the vertices in the bipartite graph have unit weight, the process is straightforward.

- Find a maximum matching.

- Drop an alternating level structure from exposed nodes in X.
- Drop an alternating level structure from exposed nodes in Y.
- Based on the two previous steps, partition X into three pieces and Y into three pieces and form a new separator from the pieces.

(If these terms are not familiar, see [6]; our present purpose is a discussion of software design, not algorithms.) A matching is a very common operation on a bipartite graph, so it is not unreasonable to expand the data object to include some mechanism for matching, e.g., a `mate[]` vector. Finding a maximum matching is a bit more tricky for there are a number of algorithms to do so, some fast, some slow, some simple, some complex. Which to choose?

If we only worked with unit weight bipartite graphs, then we probably would have added methods to find a maximum matching, and dropping alternating level structures, and then to find the Dulmage-Mendelsohn decomposition. If someone wanted to use a faster algorithm to find a maximum matching it would be a simple case of replacing a method. However, one of the strengths of this software package is that we do not work on unit weight graphs unless we have to, we work on the natural compressed graph.

The Dulmage-Mendelsohn decomposition was not defined for non-unit weight graphs. We were in new territory, at least to us. We could always expand the weighted bipartite compressed graph into a larger unit weight graph, find the Dulmage-Mendelsohn decomposition and map it back to the weighted graph. (It turns out that the DM partition is conformal with the compressed graph, i.e., a weighted vertex is completely contained inside one of the six sets.) This would have been a very ugly feature of an otherwise clean code.

Our first remedy was to design a method that found the DM decomposition of the unit weight graph while *using* the compressed graph plus a work vector whose size was the sum of the vertex weights. See the method `BPG_DMdecomposition()`. The code is appreciably faster than expanding the weighted graph to a unit weight graph, finding the decomposition and then mapping back. It is not really a method, but a module, for the fourteen hundred lines of code contain many static functions. Though the code is adequately documented, this isn't an algorithm that we felt like publicizing, so we export the method but not the internals.

After some time, thought and reflection, we came to realize that we can find the decomposition by solving a max flow problem. In some sense this is obvious, for bipartite graph matching is nothing more than a special case of max flow. Just how to formulate the max flow problem is what eluded us for an embarassing amount of time. Once we were able to formulate the problem as max flow, we wrote a new method to find the decomposition for a weighted graph. The line count for `BPG_DMviaMaxFlow()` is about one half that of `BPG_DMdecomposition()` and it is easier to understand. Both methods use a simple Ford-Fulkerson augmenting flow approach.

At this time we thought about writing an object to solve max flow problems and shifting most of the responsibility of finding the decomposition to a specialized object that solves a max flow problem on a bipartite network. Had we more time, we would have done so. The advantages are clear. In fact, that is the approach we have taken, but in a different context. To explain, we must return to our original problem.

The goal is to improve a 2-set partition $[S, B, W]$. Let B be the larger of B and W. We look at the subgraph induced by $S \cup (Adj(S) \cap B)$. The goal is to find a set $Z \subseteq S$ that will be absorbed by the smaller component W that results in a smaller separator. As a result, some nodes in $Adj(S) \cap B$ move from B into the separator set. The DM decomposition lets us identify a set Z that results in the *largest* decrease in the separator size. But, if we consider $S \cup (Adj(S) \cap B)$ to be a *wide* separator, the resulting separator $\widehat{S}$ need not be a separator with minimal weight that is found within the wide separator. The trick is that some nodes in $Adj(S) \cap B$ might be absorbed into W.

One can find a separator with minimal weight from the wide separator $S \cup (Adj(S) \cap B)$, in fact from *any* wide separator that contains S, by solving a max flow problem. The drawback is that the network induced by $S \cup (Adj(S) \cap B)$ need not be bipartite. In other words, a bipartite induced graph necessarily implies two layers to the wide separator, but the converse does not hold. We were then free to examine wide separators that had more than two layers from which to find a minimal weight separator. It turns out that three layers is better than two, in practice.

We did write a separate object to solve our max flow problem; see the `Network` object. To smooth a separator, i.e., to improve a 2-set partition, we no longer have need of the bipartite graph object. We leave the two Dulmage-Mendelsohn methods in the `BPG` object for historical and sentimental reasons.

16.1 Data Structure

A bipartite graph is a triple (X, Y, E) where X and Y are disjoint sets of vertices and $E \subseteq X \times Y$ is a set of edges connecting vertices in X and Y. The `BPG` structure has three fields.

- `int nX` : number of vertices in X
- `int nY` : number of vertices in Y
- `Graph *graph` : pointer to a graph object $G = (X \cup Y), E)$.

16.2 Prototypes and descriptions of `BPG` methods

This section contains brief descriptions including prototypes of all methods that belong to the `BPG` object.

16.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `BPG * BPG_new ( void ) ;`

   This method simply allocates storage for the `BPG` structure and then sets the default fields by a call to `BPG_setDefaultFields()`.

2. `void BPG_setDefaultFields ( BPG *bpg ) ;`

   This method sets the fields of the structure to their default values: `nX = nY = 0` and `graph = NULL`.

   Error checking: If `bpg` is `NULL`, an error message is printed and the program exits.

3. `void BPG_clearData ( BPG *bpg ) ;`

   This method releases the storage for `graph` via a call to `Graph_clearData()`, and then the structure's fields are then set to their default values with a call to `BPG_setDefaultFields()`.

   Error checking: If `bpg` is `NULL`, an error message is printed and the program exits.

4. `void BPG_free ( BPG *bpg ) ;`

   This method releases any storage by a call to `BPG_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `bpg` is `NULL`, an error message is printed and the program exits.

16.2.2 Initializer methods

There are two initializer methods.

1. `void BPG_init ( BPG *bpg, int nX, int nY, Graph *graph ) ;`

   This method initializes the `BPG` object when all three of its fields are given in the calling sequence. The `Graph` object has `nX + nY` vertices. Note, the `BPG` object now "owns" the `Graph` object and so will free the `Graph` object when it is free'd. The `Graph` object may contains edges between nodes in `X` and `Y`, but these edges are swapped to the end of each adjacency list and the size of each list is then set.

   Error checking: If `bpg` or `graph` are `NULL`, or if `nX` ≤ 0, or if `nY` ≤ 0, an error message is printed and the program exits.

2. `void BPG_initFromColoring ( BPG *bpg, Graph *graph, int colors[], int cX,`
   `int cY, int cmap[], int indX[], int indY[] ) ;`

   This method extracts a bipartite graph from a `Graph` object where the `X` vertices are those with `cmap[]` value equal to `cX` and the `Y` vertices are those with `cmap[]` value equal to `cY`. The vectors `indX[]` and `indY[]` hold the global vertex ids of the `X` and `Y` vertices respectively.

   Error checking: If `bpg`, `graph`, `colors` or `cmap` are `NULL`, or if `cX` ≤ 0, or if `cY` ≤ 0, or if `cX` $=$ `cY`, an error message is printed and the program exits.

16.2.3 Generate induced graphs

Sometimes we need to know which `X` or `Y` vertices share an edge, e.g., in the `BKL` object we need the domain-domain adjacency graph (the domains are the `X` vertices) to efficiently implement the Fiduccia-Mattheyses algorithm. We have two methods to generate the two induced graphs.

1. `Graph * BPG_makeGraphXbyX ( BPG *bpg ) ;`

   This method constructs and returns a `Graph` object whose vertices are `X` and an edge `(x1,x2)` is in the graph when there is a `Y` vertex `y` such that `(x1,y)` and `(x2,y)` are in the bipartite graph.

   Error checking: If `bpg` is `NULL`, an error message is printed and the program exits.

2. `Graph * BPG_makeGraphYbyY ( BPG *bpg ) ;`

   This method constructs and returns a `Graph` object whose vertices are `Y` and an edge `(y1,y2)` is in the graph when there is a `X` vertex `x` such that `(x,y1)` and `(x,y2)` are in the bipartite graph.

   Error checking: If `bpg` is `NULL`, an error message is printed and the program exits.

16.2.4 Utility methods

1. `int BPG_pseudoperipheralnode ( BPG *bpg, int seed ) ;`

   This method finds and returns a pseudoperipheral node for the bipartite graph.

   Error checking: If `bpg` is `NULL`, an error message is printed and the program exits.

2. `int BPG_levelStructure ( BPG *bpg, int root, int list[], int dist[],`
   `int mark[], int tag ) ;`

   This method drops a level structure from vertex `root`, fills the `dist[]` vector with the distances from `root`, and returns the number of levels created. The `mark[]` vector is used to mark nodes with the `tag` value as they are placed in the level structure. The `list[]` vector is used to accumulate the nodes as they are placed in the level structure.

   Error checking: If `bpg`, `list`, `dist` or `mark` is `NULL`, or if `root` is not in `[0, nX+nY)`, an error message is printed and the program exits.

16.2.5 Dulmage-Mendelsohn decomposition method

There is one method to find the Dulmage-Mendelsohn decomposition that uses matching when the graph is unit weight and a generalized matching technique otherwise. There is a second method to find the decomposition using a Ford-Fulkerson algorithm to find a max flow and a min-cut on a bipartite network. This has largely been superceded by the `Network` object.

1. `void BPG_DMdecomposition ( BPG *bpg, int dmflags[], int stats[], int msglvl, FILE *msgFile )`

   This method constructs and returns the Dulmage-Mendelsohn decomposition for a unit weight graph and its generalization for a non-unit weight graph. On return, the `dmflags[]` vector is filled with the following values:

$$\texttt{dmflags[x]} = \begin{cases} 0 & \text{if } \texttt{x} \in X_R \\ 1 & \text{if } \texttt{x} \in X_I \\ 2 & \text{if } \texttt{x} \in X_E \end{cases} \qquad \texttt{dmflags[y]} = \begin{cases} 0 & \text{if } \texttt{y} \in Y_R \\ 1 & \text{if } \texttt{y} \in Y_I \\ 2 & \text{if } \texttt{y} \in Y_E \end{cases}$$

   The set $X_I \cup Y_E$ contains all nodes that are reachable via alternating paths starting from exposed nodes in X. The set $Y_I \cup X_E$ contains all nodes that are reachable via alternating paths starting from exposed nodes in Y. The remaining two sets are $X_R = X \setminus (X_I \cup X_E)$ and $Y_R = Y \setminus (Y_I \cup Y_E)$. On return, the `stats[]` vector is filled with the following values:

`stats[0]`	—	weight of X_I	`stats[3]`	—	weight of Y_I
`stats[1]`	—	weight of X_E	`stats[4]`	—	weight of Y_E
`stats[2]`	—	weight of X_R	`stats[5]`	—	weight of Y_R

   Error checking: If `bpg`, `dmflags` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

2. `void BPG_DMviaMaxFlow ( BPG *bpg, int dmflags[], int stats[], int msglvl, FILE *msgFile ) ;`

   This method has the same functionality, calling sequence and returned values as the preceding `BPG_DMdecomposition()` method.

   Error checking: If `bpg`, `dmflags` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

16.2.6 IO methods

There are the usual eight IO routines. The file structure of a `BPG` object is simple: the two scalar fields `nX` and `nY` come first and the `Graph` object follows.

1. `int BPG_readFromFile ( BPG *bpg, char *fn ) ;`

   This method reads a `BPG` object from a file. The method tries to open the file and if it is successful, it then calls `BPG_readFromFormattedFile()` or `BPG_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `bpg` or `fn` is `NULL`, or if `fn` is not of the form `*.bpgf` (for a formatted file) or `*.bpgb` (for a binary file), an error message is printed and the method returns zero.

2. `int BPG_readFromFormattedFile ( BPG *bpg, FILE *fp ) ;`

   This method reads a `BPG` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `bpg` or `fp` is `NULL` an error message is printed and zero is returned.

3. `int BPG_readFromBinaryFile ( BPG *bpg, FILE *fp ) ;`

   This method reads a `BPG` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `bpg` or `fp` is `NULL` an error message is printed and zero is returned.

4. `int BPG_writeToFile ( BPG *bpg, char *fn ) ;`

   This method writes a `BPG` object to a file. The method tries to open the file and if it is successful, it then calls `BPG_writeFromFormattedFile()` or `BPG_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `bpg` or `fn` is `NULL`, or if `fn` is not of the form `*.bpgf` (for a formatted file) or `*.bpgb` (for a binary file), an error message is printed and the method returns zero.

5. `int BPG_writeToFormattedFile ( BPG *bpg, FILE *fp ) ;`

   This method writes a `BPG` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `bpg` or `fp` is `NULL`, an error message is printed and zero is returned.

6. `int BPG_writeToBinaryFile ( BPG *bpg, FILE *fp ) ;`

   This method writes a `BPG` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `bpg` or `fp` is `NULL`, an error message is printed and zero is returned.

7. `int BPG_writeForHumanEye ( BPG *bpg, FILE *fp ) ;`

   This method writes a `BPG` object to a file in a human readable format. The method `BPG_writeStats()` is called to write out the header and statistics. Then the `bpg->graph` object is written via a call to `Graph_writeForHumanEye()`. The value `1` is returned.

   Error checking: If `bpg` or `fp` is `NULL`, an error message is printed and zero is returned.

8. `int BPG_writeStats ( BPG *bpg, FILE *fp ) ;`

   This method writes a header with statistics to a file. A header is written and the value `1` is returned.

   Error checking: If `bpg` or `fp` is `NULL`, an error message is printed and zero is returned.

16.3 Driver programs for the BPG object

This section contains brief descriptions of the driver programs.

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads and write `BPG` files, useful for converting formatted files to binary files and vice versa. One can also read in a `BPG` file and print out just the header information (see the `BPG_writeStats()` method).

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `BPG` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `BPG` object. It must be of the form `*.bpgf` or `*.bpgb`. The `BPG` object is read from the file via the `BPG_readFromFile()` method.

- The `outFile` parameter is the output file for the `BPG` object. If `outFile` is `none` then the `BPG` object is not written to a file. Otherwise, the `BPG_writeToFile()` method is called to write the graph to a formatted file (if `outFile` is of the form `*.bpgf`), or a binary file (if `outFile` is of the form `*.bpgb`).

2. `extractBPG msglvl msgFile inGraphFile inCompidsIVfile icomp outMapFile outBPGfile`

   This driver program reads in a `Graph` object and an `IV` object that contains the component ids. (A separator vertex has component id zero; other vertices have positive component ids to identify the subgraph that contains them.) It then extracts out the bipartite graph formed by the separator and nodes in the target component that are adjacent to the separator.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `inCompidsIVfile` parameter is the input file for the `IV` object that contains the component ids. It must be of the form `*.ivf` or `*.ivb`. The `IV` object is read from the file via the `IV_readFromFile()` method.
   - The `icomp` parameter defines the target component to form the Y nodes of the bipartite graph. (The separator nodes, component zero, form the X nodes.)
   - The `outMapFile` parameter is the output file for the `IV` object that holds the map from vertices in the bipartite graph to vertices in the original graph. If `outMapFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the `IV` object to a formatted file (if `outMapFile` is of the form `*.ivf`), or a binary file (if `outMapFile` is of the form `*.ivb`).
   - The `outBPGFile` parameter is the output file for the compressed `BPG` object. If `outBPGFile` is `none` then the `BPG` object is not written to a file. Otherwise, the `BPG_writeToFile()` method is called to write the graph to a formatted file (if `outBPGFile` is of the form `*.graphf`), or a binary file (if `outBPGFile` is of the form `*.graphb`).

3. `testDM msglvl msgFile inBPGfile`

   This driver program reads in a `BPG` object from a file. It then finds the Dulmage-Mendelsohn decomposition using two methods.

   - `BPG_DMdecomposition()` which uses matching techniques on the weighted graph.
   - `BPG_DMviaMaxFlow()` which forms bipartite network and solves the max flow problem using a simple Ford-Fulkerson algorithm.

   This provides a good check on the two programs (they must have the same output) and writes their execution times.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - The `inBPGFile` parameter is the input file for the `BPG` object. It must be of the form `*.graphf` or `*.graphb`. The `BPG` object is read from the file via the `BPG_readFromFile()` method.

Chapter 17

DSTree: A Domain/Separator Tree Object

The DSTree object represents a recursive partition of a graph, as is constructed during a nested dissection procedure. The graph is split by a vertex *separator* into subgraphs, and this process continues recursively up to some point. A subgraph which is not split is a domain. The DSTree object is normally created by the GPart graph partitioning object and then used to determine the stages vector used as input to the MSMD multistage minimum degree object.

The DSTree object contains a Tree object that stores the natural tree links between separators and domains. The top level separator has no parent. Once a separator S splits a graph, each subgraph is either split again (in this case S is the parent of the separator that splits the subgraph) or S is the parent of the subgraph (which is a domain). The DSTree object also contains an IV object that stores a map from the vertices to the domains and separators.

The DSTree object is a natural output from a nested dissection or other graph partitioning algorithm that uses vertex separators. Presently it has only one active function — it creates a map from the vertices to the *stages* needed as input for the multi-stage minimum degree algorithm (see the MSMD object). Multisection orders the vertices in two stages: all vertices in the domains first, then the vertices in the separators. Nested dissection orders the vertices in as many stages as there are levels in the DSTree object.

17.1 Data Structure

The DSTree object has a very simple data structure. It contains a Tree object to represent the tree fields of the domains and separators, and an IV object to hold the map from the vertices to the domains and separators.

- Tree *tree : pointer to the Tree object
- IV *mapIV : pointer to the IV object that holds the map from vertices to domains and separators.

17.2 Prototypes and descriptions of DSTree methods

This section contains brief descriptions including prototypes of all methods that belong to the DSTree object.

17.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `DSTree * DSTree_new ( void ) ;`

   This method allocates storage for an instance of the `DSTree` object. The default fields are set by a call to `DSTree_setDefaultFields()`.

2. `void DSTree_setDefaultFields ( DSTree *dstree ) ;`

   This method sets the data fields to default values: `tree` and `mapIV` are set to `NULL` ;

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

3. `void DSTree_clearData ( DSTree *dstree ) ;`

   This method clears the data fields, free'ing storage that has been allocated by the object and free'ing objects that it owns. This method checks to see whether `dstree` is `NULL`. If `tree` is not `NULL`, then `Tree_free(tree)` is called. If `mapIV` is not `NULL`, then `IV_free(mapIV)` is called. Then the structure's default fields are set via a call to `DSTree_setDefaultFields()`.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

4. `void DSTree_free ( DSTree *dstree ) ;`

   This method checks to see whether `dstree` is `NULL`. If so, an error message is printed and the program exits. Otherwise, it releases any storage by a call to `DSTree_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

17.2.2 Instance methods

1. `Tree * DSTree_tree ( DSTree *dstree ) ;`

   This method returns a pointer to its `Tree` object.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

2. `IV * DSTree_mapIV ( DSTree *dstree ) ;`

   This method returns a pointer to its `IV` object that maps vertices to domains and separators.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

17.2.3 Initializer methods

There are three initializers and two helper functions to set the dimensions of the dstree, allocate the three vectors, and fill the information.

1. `void DSTree_init1 ( DSTree *dstree, int ndomsep, int nvtx ) ;`

   This method initializes an object given the number of vertices, (the dimension of `mapIV`) and domains and separators (the number of nodes in `tree`). It then clears any previous data with a call to `DSTree_clearData()`. The `tree` field is created and initialized via a call to `Tree_init1()`. The `mapIV` field is created and initialized via a call to `IV_init1()`.

   Error checking: If `dstree` is `NULL`, or `ndomsep` or `nvtx` are negative, an error message is printed and the program exits.

2. `void DSTree_init2 ( DSTree *dstree, Tree *tree, IV *mapIV) ;`

   Any previous data is cleared with a call to `DSTree_clearData()`. Then the `tree` and `mapIV` fields are set to the pointers in the calling sequence.

   Error checking: If `dstree`, `tree` or `mapIV` are `NULL`, an error message is printed and the program exits.

17.2.4 Stage methods

The only active function of a `DSTree` object is to construct the stages vector needed as input to the multi-stage minimum degree `MSMD` object. Each domain and separator has a particular level associated with it. A domain is a leaf of the domain/separator tree, and has level zero. Each separator has a level that is one greater than the maximum level of its children.

1. `IV * DSTree_NDstages ( DSTree *dstree ) ;`

   This method returns the stages for natural nested dissection. The levels of the domains and separators are obtained via a call to `Tree_setHeightImetric()`. A `stagesIV` `IV` object is created of size `nvtx = mapIV->size`, filled and then returned. The stage of a vertex is the level of the domain or separator which contains the vertex.

   Error checking: If `dstree` is `NULL`, or if the object has not been initialized, an error message is printed and the program exits.

2. `IV * DSTree_ND2stages ( DSTree *dstree ) ;`

   This method returns the stages for a nested dissection variant, separators on two adjacent levels are put into the same stage. The levels of the domains and separators are obtained via a call to `Tree_setHeightImetric()`. A `stagesIV` `IV` object is created of size `nvtx = mapIV->size`, filled and then returned. If a vertex is found in a domain, its stage is zero. If a vertex is found in a separator at level k, its stage is $\lceil k/2 \rceil$.

   Error checking: If `dstree` is `NULL`, or if the object has not been initialized, an error message is printed and the program exits.

3. `IV * DSTree_MS2stages ( DSTree *dstree ) ;`

   This method returns the stages for the standard multisection ordering. The levels of the domains and separators are obtained via a call to `Tree_setHeightImetric()`. A `stagesIV` `IV` object is created of size `nvtx = mapIV->size`, filled and then returned. If a vertex is found in a domain, its stage is zero. If a vertex is found in a separator, its stage is one.

   Error checking: If `dstree` is `NULL`, or if the object has not been initialized, an error message is printed and the program exits.

4. `IV * DSTree_MS3stages ( DSTree *dstree ) ;`

   This method returns the stages for a three-stage variant of the multisection ordering. The levels of the domains and separators are obtained via a call to `Tree_setHeightImetric()`. A `stagesIV` `IV` object is created of size `nvtx = mapIV->size`, filled and then returned. If a vertex is found in a domain, its stage is zero. The levels of the separators are split into two sets, the lower levels and the upper levels. The stage of a vertex that is found in a separator is either one (if the separator is in the lower levels) or two (if the separator is in the upper levels).

   Error checking: If `dstree` is `NULL`, or if the object has not been initialized, an error message is printed and the program exits.

5. `IV * DSTree_stagesViaDomainWeight ( DSTree *dstree, int *vwghts, DV *cutoffDV ) ;`

   This method sets the stages vector based on subtree (or domain) weights. Each vertex is mapped to a node in the tree. We generate the *subtree weights* for each subtree, the fraction of the total vertex weight (based on `vwghts[]`) that is contained in the subtree. For each node in the tree, its fraction of the node weights lies between two consecutive values in the `cutoff[]` vector, and that is the stage for all vertices contained in the node.

   Error checking: If `dstree` or `cutoffDV` is `NULL`, or if the object has not been initialized, an error message is printed and the program exits.

17.2.5 Utility methods

There is one utility method that returns the number of bytes taken by the object.

1. `int DSTree_sizeOf ( DSTree *dstree ) ;`

   If `dstree` is `NULL`, an error message is printed and the program exits. Otherwise, the number of bytes taken by this object is returned.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

2. `void DSTree_renumberViaPostOT ( DSTree *dstree ) ;`

   This method renumbers the fronts in the tree via a post-order traversal.

   Error checking: If `dstree` is `NULL`, or if the object has not been initialized, an error message is printed and the program exits.

3. `int DSTree_domainWeight ( DSTree *dstree, int vwghts[] ) ;`

   This method returns the weight of the vertices in the domains. If `vwghts` is `NULL`, the vertices have unit weight.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

4. `int DSTree_separatorWeight ( DSTree *dstree, int vwghts[] ) ;`

   This method returns the weight of the vertices in the separators. If `vwghts` is `NULL`, the vertices have unit weight.

   Error checking: If `dstree` is `NULL`, an error message is printed and the program exits.

17.2.6 IO methods

There are the usual eight IO routines. The file structure of a dstree object is simple: the structure for a `Tree` object followed by the structure for an `IV` object.

1. `int DSTree_readFromFile ( DSTree *dstree, char *fn ) ;`

   This method reads a `DSTree` object from a file. It tries to open the file and if it is successful, it then calls `DSTree_readFromFormattedFile()` or `DSTree_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `dstree` or `fn` are `NULL`, or if `fn` is not of the form `*.dstreef` (for a formatted file) or `*.dstreeb` (for a binary file), an error message is printed and the method returns zero.

2. `int DSTree_readFromFormattedFile ( DSTree *dstree, FILE *fp ) ;`

   This method reads in a `DSTree` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `dstree` or `fp` is `NULL`, an error message is printed and zero is returned.

3. `int DSTree_readFromBinaryFile ( DSTree *dstree, FILE *fp ) ;`

   This method reads in a `DSTree` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `dstree` or `fp` is `NULL`, an error message is printed and zero is returned.

4. `int DSTree_writeToFile ( DSTree *dstree, char *fn ) ;`

   This method writes a `DSTree` object to a file. It tries to open the file and if it is successful, it then calls `DSTree_writeFromFormattedFile()` or `DSTree_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `dstree` or `fn` are `NULL`, or if `fn` is not of the form `*.dstreef` (for a formatted file) or `*.dstreeb` (for a binary file), an error message is printed and the method returns zero.

5. `int DSTree_writeToFormattedFile ( DSTree *dstree, FILE *fp ) ;`

   This method writes a `DSTree` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `dstree` or `fp` is `NULL`, an error message is printed and zero is returned.

6. `int DSTree_writeToBinaryFile ( DSTree *dstree, FILE *fp ) ;`

   This method writes a `DSTree` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `dstree` or `fp` is `NULL`, an error message is printed and zero is returned.

7. `int DSTree_writeForHumanEye ( DSTree *dstree, FILE *fp ) ;`

   This method writes a `DSTree` object to a file in a human readable format. The method `DSTree_writeStats()` is called to write out the header and statistics. Then the tree structure is printed via a call to `Tree_writeForHumanEye`, followed by the map structure via a call to `IV_writeForHumanEye`. The value `1` is returned.

   Error checking: If `dstree` or `fp` is `NULL`, an error message is printed and zero is returned.

8. `int DSTree_writeStats ( DSTree *dstree, FILE *fp ) ;`

   This method write the header and statistics to a file. The value `1` is returned.

   Error checking: If `dstree` or `fp` is `NULL`, an error message is printed and zero is returned.

17.3 Driver programs for the `DSTree` object

This section contains brief descriptions of the driver programs.

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads and write `DSTree` files, useful for converting formatted files to binary files and vice versa. One can also read in a `DSTree` file and print out just the header information (see the `DSTree_writeStats()` method).

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `DSTree` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inFile` parameter is the input file for the `DSTree` object. It must be of the form `*.dinpmtxf` or `*.dinpmtxb`. The `DSTree` object is read from the file via the `DSTree_readFromFile()` method.
- The `outFile` parameter is the output file for the `DSTree` object. If `outFile` is `none` then the `DSTree` object is not written to a file. Otherwise, the `DSTree_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.dinpmtxf`), or a binary file (if `outFile` is of the form `*.dinpmtxb`).

2. `writeStagesIV msglvl msgFile inFile type outFile`

   This driver program reads in a `DSTree` from a file, creates a stages `IV` object and writes it to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `DSTree` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `DSTree` object. It must be of the form `*.dstreef` or `*.dstreeb`. The `DSTree` object is read from the file via the `DSTree_readFromFile()` method.
   - The `type` parameter specifies which type of stages vector to create. There are presently four supported types : `ND`, `ND2`, `MS2` and `ND3`. See the stage methods in Section 17.2.4.
   - The `outFile` parameter is the output file for the stages `IV` object. If `outFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.ivf`), or a binary file (if `outFile` is of the form `*.ivb`).

3. `testDomWeightStages msglvl msgFile`
   `inDSTreeFile inGraphFile inCutoffDVfile outFile`

   This driver program is used to create a stages vector based on subtree weight. It reads in three objects from files: a `DSTree` object, a `Graph` object and a `DV` object that contains the cutoff vector, then creates a stages `IV` object and writes it to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `DSTree` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inDSTreeFile` parameter is the input file for the `DSTree` object. It must be of the form `*.dstreef` or `*.dstreeb`. The `DSTree` object is read from the file via the `DSTree_readFromFile()` method.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `inCutoffDVfile` parameter is the input file for the cutoff `DV` object. It must be of the form `*.dvf` or `*.dvb`. The `DV` object is read from the file via the `DV_readFromFile()` method.
   - The `outFile` parameter is the output file for the stages `IV` object. If `outFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.ivf`), or a binary file (if `outFile` is of the form `*.ivb`).

Chapter 18

EGraph: Element Graph Object

The EGraph object is used to model a graph that has a natural element structure (as from finite elements) or a natural covering clique structure (e.g., the rows of A are natural cliques for the graph of $A^T A$).

Translating an element graph EGraph object into an adjacency list Graph object is an easy task — we provide a method to do so — but the process in reverse is much more difficult. Given a Graph object, it is simple to construct a trivial element graph object, simply take each (i, j) edge to be an element. Constructing an element graph with a smaller number of elements is more difficult.

Element graphs, when they arise naturally or are constructed from an adjacency graph, have great potential. The element model for sparse elimination *appears* to be more powerful than the vertex adjacency list model in the sense that concepts like indistinguishability, outmatching and deficiency are more naturally defined with elements. An element graph might be a more natural vehicle for partitioning graphs, because if one consider elements as the "nodes" in a Kernighan-Lin type algorithm, then the "edge" separators are formed of vertices of the original graph.

18.1 Data Structure

The EGraph object has five fields.

- `int type` : type of graph. When `type = 0`, the vertices have unit weight When `type = 1`, the vertices have possibly non-unit weight and the `vwghts` field is not `NULL`.
- `int nelem` : number of elements in the graph
- `int nvtx` : number of vertices in the graph
- `IVL *adjIVL` : pointer to a `IVL` structure that holds the vertex lists for the elements.
- `int *vwghts` : when `type = 1`, `vwghts` points to an `int` vector of size `nvtx` that holds the node weights.

A correctly initialized and nontrivial EGraph object will have positive `nelem` and `nvtx` values, a valid `adjIVL` field. If `type = 1`, the `vwghts` will be non-`NULL`.

18.2 Prototypes and descriptions of EGraph methods

This section contains brief descriptions including prototypes of all methods that belong to the EGraph object.

18.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `EGraph * EGraph_new ( void ) ;`

   This method simply allocates storage for the `EGraph` structure and then sets the default fields by a call to `EGraph_setDefaultFields()`.

2. `void EGraph_setDefaultFields ( EGraph *egraph ) ;`

   This method sets the structure's fields are set to default values: `type = nelem = nvtx = 0`, `adjIVL = vwghts = NULL`.

   Error checking: If `egraph` is `NULL` an error message is printed and the program exits.

3. `void EGraph_clearData ( EGraph *egraph ) ;`

   This method clears data and releases any storage allocated by the object. If `egraph->adjIVL` is not `NULL`, then `IVL_free(egraph->adjIVL)` is called to free the `IVL` object. If `egraph->vwghts` is not `NULL`, then `IVfree(egraph->vwghts)` is called to free the `int` vector. It then sets the structure's default fields with a call to `EGraph_setDefaultFields()`.

   Error checking: If `egraph` is `NULL` an error message is printed and the program exits.

4. `void EGraph_free ( EGraph *egraph ) ;`

   This method releases any storage by a call to `EGraph_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `egraph` is `NULL` an error message is printed and the program exits.

18.2.2 Initializer methods

1. ```
   void EGraph_init ( EGraph *egraph, int type, int nelem, int nvtx,
                      int IVL_type ) ;
   ```

   This method initializes an `EGraph` object given the type of vertices, number of elements, number of vertices, and storage type for the `IVL` element list object. It then clears any previous data with a call to `EGraph_clearData()`. The `IVL` object is initialized by a call to `IVL_init1()`. If `type = 1`, the `vwghts` is initialized via a call to `IVinit()`. See the `IVL` object for a description of the `IVL_type` parameter.

   Error checking: If `egraph` is `NULL` or `type` is not zero or one, or if either `nelem` or `nvtx` are nonpositive, an error message is printed and the program exits.

18.2.3 Utility methods

1. `Graph EGraph_mkAdjGraph ( EGraph *egraph ) ;`

   This method creates and returns a `Graph` object with vertex adjacency lists from the element graph object.

   Error checking: If `egraph` is `NULL`, an error message is printed and the program exits.

2. `EGraph * EGraph_make9P ( int n1, int n2, int ncomp ) ;`

   This method creates and returns a `EGraph` object for a `n1` $\times$ `n2` grid for a 9-point operator matrix. Each element is a linear quadrilateral finite element with `ncomp` degrees of freedom at the grid points. The resulting graph has `n1*n2*ncomp` vertices and `(n1-1)*(n2-1)` elements.

   Error checking: If `n1`, `n2` or `ncomp` is less than or equal to zero, an error message is printed and the program exits.
   ```

3. `EGraph * EGraph_make27P ( int n1, int n2, int n3, int ncomp ) ;`

   This method creates and returns a `EGraph` object for a `n1` $\times$ `n2` $\times$ `n3` grid for a 27-point operator matrix. Each element is a linear hexahedral finite element with `ncomp` degrees of freedom at the grid points. The resulting graph has `n1*n2*n3*ncomp` vertices and `(n1-1)*(n2-1)*(n3-1)` elements.

   Error checking: If `n1`, `n2`, `n3` or `ncomp` is less than or equal to zero, an error message is printed and the program exits.

18.2.4 IO methods

There are the usual eight IO routines. The file structure of a EGraph object is simple: `type`, `nelem`, `nvtx`, an `IVL` object, and an `int` vector if `vwghts` is not `NULL`.

1. `int EGraph_readFromFile ( EGraph *egraph, char *fn ) ;`

   This method reads an `EGraph` object from a file. It tries to open the file and if it is successful, it then calls `EGraph_readFromFormattedFile()` or `EGraph_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `egraph` or `fn` are `NULL`, or if `fn` is not of the form `*.egraphf` (for a formatted file) or `*.egraphb` (for a binary file), an error message is printed and the method returns zero.

2. `int EGraph_readFromFormattedFile ( EGraph *egraph, FILE *fp ) ;`

   This method reads in an `EGraph` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `egraph` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int EGraph_readFromBinaryFile ( EGraph *egraph, FILE *fp ) ;`

   This method reads in an `EGraph` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `egraph` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int EGraph_writeToFile ( EGraph *egraph, char *fn ) ;`

   This method writes an `EGraph` object to a file. It tries to open the file and if it is successful, it then calls `EGraph_writeFromFormattedFile()` or `EGraph_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `egraph` or `fn` are `NULL`, or if `fn` is not of the form `*.egraphf` (for a formatted file) or `*.egraphb` (for a binary file), an error message is printed and the method returns zero.

5. `int EGraph_writeToFormattedFile ( EGraph *egraph, FILE *fp ) ;`

   This method writes an `EGraph` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `egraph` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int EGraph_writeToBinaryFile ( EGraph *egraph, FILE *fp ) ;`

   This method writes an `EGraph` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `egraph` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int EGraph_writeForHumanEye ( EGraph *egraph, FILE *fp ) ;`

   This method writes an `EGraph` object to a file in a human readable format. The method `EGraph_writeStats()` is called to write out the header and statistics. Then the `adjIVL` object is written out using `IVL_writeForHumanEye()`. If the `vwghts` vector is present, the vertex weights are written out. The value `1` is returned.

   Error checking: If `egraph` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int EGraph_writeStats ( EGraph *egraph, FILE *fp ) ;`

   This method writes a header and statistics to a file. The value `1` is returned.

   Error checking: If `egraph` or `fp` are `NULL` an error message is printed and zero is returned.

18.3 Driver programs for the EGraph object

This section contains brief descriptions of the driver programs.

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads and writes `EGraph` files, useful for converting formatted files to binary files and vice versa. One can also read in a `EGraph` file and print out just the header information (see the `EGraph_writeStats()` method).

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `EGraph` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `EGraph` object. It must be of the form `*.egraphf` or `*.egraphb`. The `EGraph` object is read from the file via the `EGraph_readFromFile()` method.
   - The `outFile` parameter is the output file for the `EGraph` object. If `outFile` is `none` then the `EGraph` object is not written to a file. Otherwise, the `EGraph_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.egraphf`), or a binary file (if `outFile` is of the form `*.egraphb`).

2. `mkGraph msglvl msgFile inEGraphFile outGraphFile`

   This driver program reads in an `EGraph` object and creates a `Graph` object, which is then optionally written out to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `EGraph` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inEGraphFile` parameter is the input file for the `EGraph` object. It must be of the form `*.egraphf` or `*.egraphb`. The `EGraph` object is read from the file via the `EGraph_readFromFile()` method.
   - The `outGraphFile` parameter is the output file for the `Graph` object. If `outGraphFile` is `none` then the `Graph` object is not written to a file. Otherwise, the `Graph_writeToFile()` method is called to write the object to a formatted file (if `outGraphFile` is of the form `*.graphf`), or a binary file (if `outGraphFile` is of the form `*.graphb`).

3. `mkGridEGraph msglvl msgFile n1 n2 n3 ncomp outEGraphFile`

   This driver program creates an element graph for linear quadrilateral elements if `n3 = 1` or for linear hexahedral elements if `n3 > 1`. There are `ncomp` degrees of freedom at each grid point. The `EGraph` object is optionally written out to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - `n1` is the number of grid points in the first direction, must be greater than one.
   - `n2` is the number of grid points in the second direction, must be greater than one.
   - `n3` is the number of grid points in the third direction, must be greater than or equal to one.
   - `ncomp` is the number of components (i.e., the number of degrees of freedom) at each grid point, must be greater than or equal to one.
   - The `outEGraphFile` parameter is the output file for the `EGraph` object. If `outEGraphFile` is `none` then the `EGraph` object is not written to a file. Otherwise, the `EGraph_writeToFile()` method is called to write the object to a formatted file (if `outEGraphFile` is of the form `*.egraphf`), or a binary file (if `outEGraphFile` is of the form `*.egraphb`).

Chapter 19

ETree: Elimination and Front Trees

The ETree object is used to model an elimination tree or a front tree for a sparse factorization with symmetric structure. The tree is defined over a set of vertices in a graph — the graph can be unit weight or non-unit weight. A "node" in the tree can be a single vertex (in the context of an elimination tree) or a group of vertices (as for a front tree).

The tree information is stored as a Tree object. In addition there are three IV objects. One stores the total size of the nodes in the fronts, one stores the size of the boundaries of the fronts, and one stores the map from the vertices to the fronts.

There is a great deal of functionality embodied into the ETree object. Given an elimination tree or a front tree, one can extract the permutation vectors (for the fronts or the vertices), extract a multisector based on several criteria, compress the front tree in several ways, justify the tree (order children of a node in meaningful ways), evaluate metric vectors on the tree (heights, depths, subtree accumulators).

The front tree we obtain from a low-fill matrix ordering is usually not the front tree that drives the factorization. We provide three methods that transform the former into the latter. One method merges the fronts together in a way that adds logical zeros to their structure. One method splits large fronts into smaller fronts. One method combines these two functionalities.

19.1 Data Structure

The ETree object has six fields.

- `int nfront` : number of fronts in the tree
- `int nvtx` : number of vertices in the tree
- `Tree *tree` : pointer to a Tree structure
- `IV *nodwghtsIV` : pointer to an IV object to hold front weights, size nfront
- `IV *bndwghtsIV` : pointer to an IV object to hold the weights of the fronts' boundaries, size nfront
- `IV *vtxToFrontIV` : pointer to an IV object to hold the map from vertices to fronts, size nfront

A correctly initialized and nontrivial ETree object will have positive nfront and nvtx values, a valid tree field and non-NULL nodwghtsIV, bndwghtsIV and vtxToFrontIV pointers.

19.2 Prototypes and descriptions of ETree methods

This section contains brief descriptions including prototypes of all methods that belong to the ETree object.

19.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `ETree * ETree_new ( void ) ;`

   This method simply allocates storage for the ETree structure and then sets the default fields by a call to `ETree_setDefaultFields()`.

2. `void ETree_setDefaultFields ( ETree *etree ) ;`

   This method sets the structure's fields are set to default values: `nfront = nvtx = 0`, `tree = nodwghtsIV = bndwghtsIV = vtxToFrontIV = NULL`.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

3. `void ETree_clearData ( ETree *etree ) ;`

   This method clears data and releases any storage allocated by the object. If `tree` is not `NULL`, then `Tree_free(tree)` is called to free the `Tree` object. It releases any storage held by the `nodwghtsIV`, `bndwghtsIV` and `vtxToFrontIV` `IV` objects via calls to `IV_free()`. It then sets the structure's default fields with a call to `ETree_setDefaultFields()`.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

4. `void ETree_free ( ETree *etree ) ;`

   This method releases any storage by a call to `ETree_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

19.2.2 Instance methods

1. `int ETree_nfront ( ETree *etree ) ;`

   This method returns the number of fronts.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

2. `int ETree_nvtx ( ETree *etree ) ;`

   This method returns the number of vertices.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

3. `Tree * ETree_tree ( ETree *etree ) ;`

   This method returns a pointer to the `Tree` object.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

4. `int  ETree_root ( ETree *etree ) ;`

   This method returns the id of the root node.

   Error checking: If `etree` or `etree->tree` is `NULL`, an error message is printed and the program exits.

5. `int *  ETree_par ( ETree *etree ) ;`

   This method returns the pointer to the parent vector.

   Error checking: If `etree` or `etree->tree` is `NULL`, an error message is printed and the program exits.

6. `int *  ETree_fch ( ETree *etree ) ;`

   This method returns the pointer to the first child vector.

   Error checking: If `etree` or `etree->tree` is `NULL`, an error message is printed and the program exits.

7. `int *  ETree_sib ( ETree *etree ) ;`

   This method returns the pointer to the sibling vector.

   Error checking: If `etree` or `etree->tree` is `NULL`, an error message is printed and the program exits.

8. `IV * ETree_nodwghtsIV ( ETree *etree ) ;`

   This method returns a pointer to the `nodwghtsIV` object.

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

9. `int * ETree_nodwghts ( ETree *etree ) ;`

   This method returns a pointer to the `nodwghts` vector.

   Error checking: If `etree` or `etree->nodwghtsIV` is `NULL`, an error message is printed and the program exits.

10. `IV * ETree_bndwghtsIV ( ETree *etree ) ;`

    This method returns a pointer to the `bndwghtsIV` object.

    Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

11. `int * ETree_bndwghts ( ETree *etree ) ;`

    This method returns a pointer to the `bndwghts` vector.

    Error checking: If `etree` or `etree->bndwghtsIV` is `NULL`, an error message is printed and the program exits.

12. `IV * ETree_vtxToFrontIV ( ETree *etree ) ;`

    This method returns a pointer to the `vtxToFrontIV` object.

    Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

13. `int * ETree_vtxToFront ( ETree *etree ) ;`

    This method returns a pointer to the `vtxToFront` vector.

    Error checking: If `etree` or `etree->vtxToFrontIV` is `NULL`, an error message is printed and the program exits.

14. `int ETree_frontSize ( ETree *etree, int J ) ;`

    This method returns the number of internal degrees of freedom in front `J`.

    Error checking: If `etree` is `NULL`, or if `J` is out of range, an error message is printed and the program exits.

15. `int ETree_frontBoundarySize ( ETree *etree, int J ) ;`

    This method returns the number of external or boundary degrees of freedom in front `J`.

    Error checking: If `etree` is `NULL`, or if `J` is out of range, an error message is printed and the program exits.

16. `void ETree_maxNindAndNent ( ETree *etree, int symflag,`
    `int *pmaxnind, int *pmaxnent ) ;`

    This method fills `*pmaxnind` with the maximum number of indices for a front (just column indices if symmetric front, row and column indices if nonsymmetric front) and `*pmaxnent` with the maximum number of entries for a front (just upper entries if symmetric front, all entries if nonsymmetric front). The `symflag` parameter must be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`. The entries in the (2,2) block of the front are not counted.

    Error checking: If `etree` is `NULL`, or if `symflag` is invalid, an error message is printed and the program exits.

19.2.3 Initializer methods

There are four initializer methods.

1. `void ETree_init1 ( ETree *etree, int nfront, int nvtx ) ;`

   This method initializes an `ETree` object given the number of fronts and number of vertices. Any previous data is cleared with a call to `ETree_clearData()`, The `Tree` object is initialized with a call to `Tree_init1()`. The `nodwghtsIV`, `bndwghtsIV` and `vtxToFrontIV` objects are initialized with calls to `IV_init()`. The entries in `nodwghtsIV` and `bndwghtsIV` are set to `0`, while the entries in `vtxToFrontIV` are set to `-1`.

   Error checking: If `etree` is `NULL`, or if `nfront` is negative, or if `nvtx < nfront`, an error message is printed and the program exits.

2. `void ETree_initFromGraph ( ETree *etree, Graph *g ) ;`

   This method generates an elimination tree from a graph. The `nodwghtsIV` vector object is filled with the weights of the vertices in the graph. The `tree->par` vector and `bndwghtsIV` vector object are filled using the simple $O(|L|)$ algorithm from [16]. The `fch[]`, `sib[]` and `root` fields of the included `Tree` object are then set. `vtxToFrontIV`, the `IV` object that holds the map from vertices to fronts, is set to the identity.

   Error checking: If `etree` or `g` is `NULL` or `g->nvtx` is negative, an error message is printed and the program exits.

3. `void ETree_initFromGraphWithPerms ( ETree *etree, Graph *g ) ;`
   `int newToOld[], int oldToNew[] ) ;`

   This method generates an elimination tree from a graph using two permutation vectors. The behavior of the method is exactly the same as the initializer `ETree_initFromGraph()`, with the exception that `vtxToFrontIV`, the `IV` object that holds the map from vertices to fronts, is set to the `oldToNew[]` map.

   Error checking: If `etree` or `g` is `NULL` or `g->nvtx` is negative, an error message is printed and the program exits.

4. `void ETree_initFromDenseMatrix ( ETree *etree, int n, int option, int param ) ;`

   This method initializes a front tree to factor a `n x n` dense matrix. If `option == 1`, then all fronts (save possibly the last) have the same number of internal vertices, namely `param`. If `option == 2`, then we try to make all fronts have the same number of entries in their (1,1), (1,2) and (2,1) blocks, namely `param` entries.

   Error checking: If `etree` is `NULL` or if `n` $<= 0$, or if `option` < 1, or if $2 <$ `option` , or if `param` ≤ 0, an error message is printed and the program exits.

5. `IV * ETree_initFromFile ( ETree *etree, char *inETreeFileName, int msglvl, FILE *msgFile ) ;`

   This method reads in an `ETree` object from a file, gets the old-to-new vertex permutation, permutes to vertex-to-front map, and returns an `IV` object that contains the old-to-new permutation.

   Error checking: If `etree` is `NULL` or `inETreeFileName` is "`none`", an error message is printed and the program exits.

6. `int ETree_initFromSubtree ( ETree *subtree, IV *nodeidsIV, ETree *etree, IV *vtxIV ) ;`

   This method initializes `subtree` from `tree` using the nodes of `etree` that are found in `nodeidsIV`. The map from nodes in `subtree` to nodes in `etree` is returned in `vtxIV`.

   Return code: 1 for a normal return, -1 if `subtree` is `NULL`, -2 if `nodeidsIV` is `NULL`, -3 if `etree` is `NULL`, -4 if `nodeidsIV` is invalid, -5 if `vtxIV` is `NULL`.

19.2.4 Utility methods

The utility methods return the number of bytes taken by the object, or the number of factor indices, entries or operations required by the object.

1. `int ETree_sizeOf ( ETree *etree ) ;`

   This method returns the number of bytes taken by this object (which includes the bytes taken by the internal `Tree` structure).

   Error checking: If `etree` is `NULL`, an error message is printed and the program exits.

2. `int ETree_nFactorIndices ( ETree *etree ) ;`

   This method returns the number of indices taken by the factor matrix that the tree represents. Note, if the `ETree` object is a vertex elimination tree, the number of indices is equal to the number of entries. If the number of compressed indices is required, create an `ETree` object to represent the tree of fundamental supernodes and then call this method with this compressed tree.

   Error checking: If `etree` or `tree` is `NULL` or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `int ETree_nFactorEntries ( ETree *etree, int symflag ) ;`

   This method returns the number of entries taken by the factor matrix that the tree represents. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `symflag` is invalid, an error message is printed and the program exits.

4. `double ETree_nFactorOps ( ETree *etree, int type, int symflag ) ;`

   This method returns the number of operations taken by the factor matrix that the tree represents. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `type` or `symflag` is invalid, an error message is printed and the program exits.

5. `double ETree_nFactorEntriesInFront ( ETree *etree, int symflag, int J ) ;`

   This method returns the number of entries in front `J` for an *LU* factorization. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `symflag` is invalid, or if `J` < 0, or if `J` $\geq$ `nfront`, an error message is printed and the program exits.

6. `double ETree_nInternalOpsInFront ( ETree *etree, int type, int symflag, int J ) ;`

   This method returns the number of internal operations performed by front `J` on its $(1,1)$, $(2,1)$, and $(1,2)$ blocks during a factorization. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. `symflag` must be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `type` or `symflag` is invalid, or if `J` < 0, or if `J` $\geq$ `nfront`, an error message is printed and the program exits.

7. `double ETree_nExternalOpsInFront ( ETree *etree, int type, int symflag, int J ) ;`

   This method returns the number of operations performed by front `J` on its $(2,2)$ block for an LU factorization. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. `symflag` must be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `type` or `symflag` is invalid, or if `J` < 0, or if `J` $\geq$ `nfront`, an error message is printed and the program exits.

8. `IV * ETree_factorEntriesIV ( ETree *etree, int symflag ) ;`

   This method creates and returns an `IV` object that is filled with the number of entries for the fronts. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` is `NULL`, or if `symflag` is invalid, an error message is printed and the program exits.

9. ```
   DV * ETree_backwardOps ( ETree *etree, int type, int symflag,
                            int vwghts[], IV *symbfacIVL ) ;
   ```

   This method creates and returns a `DV` object that is filled with the backward operations (left-looking) for the fronts. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. `symflag` must be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `symbfacIVL` is `NULL`, or if `type` or `symflag` is invalid, an error message is printed and the program exits.

10. `DV * ETree_forwardOps ( ETree *etree, int type, int symflag ) ;`

    This method creates and returns a `DV` object that is filled with the forward operations (right-looking) for the fronts. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. `symflag` must be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

    Error checking: If `etree` is `NULL`, or if `type` or `symflag` is invalid, an error message is printed and the program exits.

11. `ETree * ETree_expand ( ETree *etree, IV *eqmapIV ) ;`

    This method creates and returns an `ETree` object for an uncompressed graph. The map from compressed vertices to uncompressed vertices is found in the `eqmapIV` object.

    Error checking: If `etree` or `eqmapIV` is `NULL`, an error message is printed and the program exits.

12. `ETree * ETree_spliceTwoEtrees ( ETree *etree0, Graph *graph, IV *mapIV, ETree *etree1 ) ;`

    This method creates and returns an `ETree` object that is formed by splicing together two front trees, `etree0` for the vertices the eliminated domains, `etree1` for the vertices the Schur complement. The `mapIV` object maps vertices to domains or the Schur complement — if the entry is `0`, the vertex is in the Schur complement, otherwise it is in a domain.

    Error checking: If `etree0`, `graph`, `mapIV` or `etree1` is `NULL`, an error message is printed and the program exits.
   ```

19.2.5 Metrics methods

Many operations need to know some *metric* defined on the nodes in a etree. Here are three examples:

- the weight of each front in the tree (this is just the `nodwghtsIV` object);
- the number of factor entries in each front
- the number of factor operations associated with each front in a forward looking factorization.

Other metrics based on height, depth or subtree accumulation can be evaluated using the `Tree` metric methods on the `Tree` object contained in the `ETree` object.

1. `IV * ETree_nvtxMetric ( ETree *etree ) ;`

   An `IV` object of size `nvtx` is created, filled with the entries from `etree->nodwghtsIV`, and returned.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

2. `IV * ETree_nentMetric ( ETree *etree, int symflag ) ;`

   An `IV` object of size `nfront` is created and returned. Each entry of the vector is filled with the number of factor entries associated with the corresponding front. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `DV * ETree_nopsMetric ( ETree *etree, int type, int symflag ) ;`

   An `DV` object of size `nfront` is created and returned. Each entry of the vector is filled with the number of factor operations associated with the corresponding front. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `type` or `symflag` is invalid, an error message is printed and the program exits.

19.2.6 Compression methods

Frequently an `ETree` object will need to be compressed in some manner. Elimination trees usually have long chains of vertices at the higher levels, where each chain of vertices corresponds to a supernode. Liu's generalized row envelope methods partition the vertices by longest chains [17]. In both cases, we can construct a map from each node to a set of nodes to define a smaller, more compact `ETree` object. Given such a map, we construct the smaller etree.

A fundamental chain is a set of vertices $v_1, \dots, v_m$ such that

1. v_1 is a leaf or has two or more children,
2. v_i is the only child of v_{i+1} for $1 \leq i < m$,
3. v_m is either a root or has a sibling.

The set of fundamental chains is uniquely defined. In the context of elimination etrees, a fundamental chain is very close to a fundamental supernode, and in many cases, fundamental chains can be used to contruct the fronts with little added fill and factor operations.

A fundamental supernode [4] is a set of vertices $v_1, \dots, v_m$ such that

1. v_1 is a leaf or has two or more children,
2. v_i is the only child of v_{i+1} for $1 \le i < m$,
3. v_m is either a root or has a sibling, and
4. the structures of v_i and v_{i+1} are nested, i.e., `bndwght[`v_i`] = nodwght[`v_{i+1}`] + bndwght[`v_{i+1}`]` for $1 \le i < m$.

The set of fundamental supernodes is uniquely defined.

Once a map from the nodes in a tree to nodes in a compressed tree is known, the compressed tree can be created using the `ETree_compress()` method. In this way, a vertex elimination tree can be used to generate a front tree.

1. `IV * ETree_fundChainMap ( ETree *etree ) ;`

   An `IV` object of size `nfront` is created, filled via a call to `Tree_fundChainMap`, then returned.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

2. `IV * ETree_fundSupernodeMap ( ETree *etree ) ;`

   An `IV` object of size `nfront` is created, filled with the map from vertices to fundamental supernodes, then returned.

   Error checking: If `etree` or `tree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `ETree * ETree_compress ( ETree *etree, IV *frontMapIV ) ;`

   Using `frontMapIV`, a new `ETree` object is created and returned. If `frontMapIV` does not define each inverse map of a new node to be connected set of nodes in the old `ETree` object, the new `ETree` object will not be well defined.

   Error checking: If `etree` or `frontMapIV` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

19.2.7 Justification methods

Given an `ETree` object, how should the children of a node be ordered? This "justification" can have a large impact in the working storage for the front etree in the multifrontal algorithm [15]. Justification also is useful when displaying trees. These methods simply check for errors and then call the appropriate `Tree` method.

1. `void ETree_leftJustify ( ETree *etree ) ;`

   If `u` and `v` are siblings, and `u` comes before `v` in a post-order traversal, then the size of the subtree rooted at `u` is as large or larger than the size of the subtree rooted at `v`.

   Error checking: If `etree` or `tree` is `NULL`, an error message is printed and the program exits.

2. `void ETree_leftJustifyI ( ETree *etree, IV *metricIV ) ;`
   `void ETree_leftJustifyD ( ETree *etree, DV *metricDV ) ;`

   Otherwise, if `u` and `v` are siblings, and `u` comes before `v` in a post-order traversal, then the weight of the subtree rooted at `u` is as large or larger than the weight of the subtree rooted at `v`.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `metricIV` is `NULL` or invalid (wrong size or `NULL` vector inside), an error message is printed and the program exits.

19.2.8 Permutation methods

Often we need to extract a permutation from an `ETree` object, e.g., a post-order traversal of a front tree gives an ordering of the fronts for a factorization or forward solve, the inverse gives an ordering for a backward solve.

1. `IV * ETree_newToOldFrontPerm ( ETree *etree ) ;`
   `IV * ETree_oldToNewFrontPerm ( ETree *etree ) ;`

   An `IV` object is created with size `nfront`. A post-order traversal of the `Tree` object fills the new-to-old permutation. A reversal of the new-to-old permutation gives the old-to-new permutation. Both methods are simply wrappers around the respective `Tree` methods.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

2. `IV * ETree_newToOldVtxPerm ( ETree *etree ) ;`
   `IV * ETree_oldToNewVtxPerm ( ETree *etree ) ;`

   An `IV` object is created with size `nvtx`. First we find a new-to-old permutation of the fronts. Then we search over the fronts in their new order to fill the vertex new-to-old permutation vector. The old-to-new vertex permutation vector is found by first finding the new-to-old vertex permutation vector, then inverting it.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `void ETree_permuteVertices ( ETree *etree, IV *vtxOldToNewIV ) ;`

   This method permutes the vertices — the `vtxToFrontIV` map is updated to reflect the new vertex numbering.

   Error checking: If `etree` or `vtxOldToNewIV` is `NULL`, or if `nvtx` < 1, an error message is printed and the program exits.

19.2.9 Multisector methods

One of our goals is to improve a matrix ordering using the multisection ordering algorithm. To do this, we need to extract a multisector from the vertices, i.e., a set of nodes that when removed from the graph, break the remaining vertices into more than one (typically many) components. The following two methods create and return an `IV` integer vector object that contains the nodes in the multisector.

1. `IV * ETree_msByDepth ( ETree *etree, int depth ) ;`

   An `IV` object is created to hold the multisector nodes and returned. Multisector nodes have their component id zero, domain nodes have their component id one. A vertex is in the multisector if the depth of the front to which it belongs is less than or equal to `depth`.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `depth` ≤ 0, an error message is printed and the program exits.

2. `IV * ETree_msByNvtxCutoff ( ETree *etree, double cutoff ) ;`

   An `IV` object is created to hold the multisector nodes and returned. Multisector nodes have their component id zero, domain nodes have their component id one. Inclusion in the multisector is based on the number of vertices in the subtree that a vertex belongs, or strictly speaking, the number of vertices in the subtree of the front to which a vertex belongs. If weight of the subtree is more than `cutoff` times the vertex weight, the vertex is in the multisector.

Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `IV * ETree_msByNentCutoff ( ETree *etree, double cutoff, int symflag ) ;`

   An `IV` object is created to hold the multisector nodes and returned. Multisector nodes have their component id zero, domain nodes have their component id one. Inclusion in the multisector is based on the number of factor entries in the subtree that a vertex belongs, or strictly speaking, the number of factor entries in the subtree of the front to which a vertex belongs. If weight of the subtree is more than `cutoff` times the number of factor entries, the vertex is in the multisector. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `symflag` is invalid, an error message is printed and the program exits.

4. `IV * ETree_msByNopsCutoff ( ETree *etree, double cutoff, int type, int symflag ) ;`

   An `IV` object is created to hold the multisector nodes and returned. Multisector nodes have their component id zero, domain nodes have their component id one. Inclusion in the multisector is based on the number of right-looking factor operations in the subtree that a vertex belongs, or strictly speaking, the number of factor operations in the subtree of the front to which a vertex belongs. If weight of the subtree is more than `cutoff` times the number of factor operations, the vertex is in the multisector. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `type` or `symflag` is invalid, an error message is printed and the program exits.

5. ```
   void ETree_msStats ( ETree *etree, IV *msIV, IV *nvtxIV, IV *nzfIV,
                        DV *opsDV, int type, int symflag ) ;
   ```

   This method is used to generate some statistics about a domain decomposition. On input, `msIV` is a flag vector, i.e., `ms[v] = 0` means that `v` is in the Schur complement, otherwise `v` is in domain. On output, `msIV` is a map from nodes to regions, i.e., `ms[v] = 0` means that `v` is in the Schur complement, otherwise `v` is in domain `ms[v]`. On output, `nvtxIV` contains the number of vertices in each of the regions, `nzfIV` contains the number of factor entries in each of the regions, and `opsIV` contains the number of factor operations in each of the regions. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree`, `msIV`, `nvtxIV`, `nzfIV` or `opsIV` is `NULL`, an error message is printed and the program exits.

6. ```
   IV * ETree_optPart ( ETree *etree, Graph *graph, IVL *symbfacIVL,
                        double alpha, int *ptotalgain, int msglvl, FILE *msgFile ) ;
   ```

   This method is used to find the optimal domain/Schur complement partition for a semi-implicit factorization. The gain of a subtree $\widehat{J}$ is equal to $|L_{\partial J,\widehat{J}}| - |A_{\partial J,\widehat{J}}| - \alpha |L_{\widehat{J},\widehat{J}}|$. When $\alpha = 0$, we minimize active storage, when $\alpha = 1$, we minimize solve operations. On return, `*ptotalgain` is filled with the total gain. The return value is a pointer to `compidsIV`, where `compids[J] = 0` means that `J` is in the Schur complement, and `compids[J] != 0` means that `J` is in domain `compids[J]`.

   Error checking: If `etree`, `graph` or `symbfacIVL` is `NULL`, an error message is printed and the program exits.

19.2.10 Transformation methods

Often the elimination tree or front tree that we obtain from an ordering of the graph is not as appropriate for a factorization as we would like. There are two important cases.

- Near the leaves of the tree the fronts are typically small in size. There is an overhead associated with each front, and though the overhead varies with regard to the factorization algorithm, it can be beneficial to group small subtrees together into one front. The expense is added storage for the logically zero entries and the factor operations on them. In this library, the technique we use to merge fronts together is *node amalgamation* [10], or more specifically *supernode relaxation* [4].
- Near the root of the tree the fronts can be very large, large enough that special techniques are necessary to handle the large dense frontal matrices that might not be able to exist in-core. Another consideration is a parallel setting where the design decision is to have each front be factored by a single thread of computation. Large fronts dictate a long critical path in the factorization task graph. We try to split a large front into two or more smaller fronts that form a chain in the front tree. Breaking the front into smaller fronts will reduce core storage requirements and have better cache reuse and reduce the critical path through the task graph.

We provide three methods to merge fronts together and one method to break fronts apart, and one method that is a wrapper around all these. Let us describe the differences between the methods that merge fronts together. Each method performs a post-order traversal of the front tree. They differ on the decision process when visiting a front.

- The method `ETree_mergeFrontsAny()` is taken from [4]. When visiting a front it tries to merge that front with one of its children if it will not add too many zero entries to that front. If successful, it tries to merge the front with another child. This approach has served well for over a decade in a serial environment, but we discovered that it has a negative effect on nested dissection orderings when we want a parallel factorization. Often it merges the top level separator with *one* of its children, and thus reduces parallelism in the front tree.
- The method `ETree_mergeFrontsOne()` only tries to merge a front when it has only one child. This method is very useful if one has a vertex elimination tree (where the number of fronts is equal to the number of vertices), for the fundamental supernode tree can be created using `maxzeros = 0`. This method has some affect for minimum degree or fill orderings, where chains of nodes can occur in two ways: aggregation (where a vertex is eliminated that is adjacent to only one subtree) or when the indistinguishabilty test fails. In general, this method does not effectively reduce the number of fronts because it has the "parent-only child" restriction.
- The method `ETree_mergeFrontsAll()` tries to merge a front with *all* of its children, if the resulting front does not contain too many zero entries. This has the effect of merging small bushy subtrees, but will not merge a top level separator with one of its children.

For a serial application, `ETree_mergeFrontsAny()` is suitable. For a parallel application, we recommend first using `ETree_mergeFrontsOne()` followed by `ETree_mergeFrontsAll()`. See the driver programs `testTransform` and `mkNDETree` for examples of how to call the methods.

1. `ETree * ETree_mergeFrontsOne ( ETree *etree, int maxzeros, IV *nzerosIV ) ;`

   This method only tries to merge a front with its only child. It returns an `ETree` object where one or more subtrees that contain multiple fronts have been merged into single fronts. The parameter that governs the merging process is `maxzeros`, the number of zero entries that can be introduced by merging a child and parent front together. On input, `nzerosIV` contains the number of zeros presently in each front. It is modified on output to correspond with the new front tree. This method only tries to merge a front with its only child.

Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

2. `ETree * ETree_mergeFrontsAll ( ETree *etree, int maxzeros ) ;`

   This method only tries to merge a front with all of its children. It returns an `ETree` object where a front has either been merged with none or all of its children. The parameter that governs the merging process is `maxzeros`, the number of zero entries that can be introduced by merging the children and parent front together. On input, `nzerosIV` contains the number of zeros presently in each front. It is modified on output to correspond with the new front tree.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `ETree * ETree_mergeFrontsAny ( ETree *etree, int maxzeros ) ;`

   This method only tries to merge a front with any subset of its children. It returns an `ETree` object where a front has possibly merged with any of its children. The parameter that governs the merging process is `maxzeros`, the number of zero entries that can be introduced by merging the children and parent front together. On input, `nzerosIV` contains the number of zeros presently in each front. It is modified on output to correspond with the new front tree.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

4. ```
   ETree * ETree_splitFronts ( ETree *etree, int vwghts[],
                               int maxfrontsize, int seed ) ;
   ```

   This method returns an `ETree` object where one or more large fronts have been split into smaller fronts. Only an interior front (a front that is not a leaf in the tree) can be split. No front in the returned `ETree` object has more than `maxfrontsize` rows and columns. The `vwghts[]` vector stores the number of degrees of freedom associated with a vertex; if `vwghts` is `NULL`, then the vertices have unit weight. The way the vertices in a front to be split are assigned to smaller fronts is random; the `seed` parameter is a seed to a random number generator that permutes the vertices in a front.

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `maxfrontsize` ≤ 0, an error message is printed and the program exits.

5. ```
   ETree * ETree_transform  ( ETree *etree, int vwghts[], int maxzeros,
                              int maxfrontsize, int seed ) ;
   ETree * ETree_transform2 ( ETree *etree, int vwghts[], int maxzeros,
                              int maxfrontsize, int seed ) ;
   ```

   These methods returns an `ETree` object where one or more subtrees that contain multiple fronts have been merged into single fronts and where one or more large fronts have been split into smaller fronts. The two methods differ slightly. `ETree_transform2()` is better suited for parallel computing because it tends to preserve the tree branching properties. (A front is merged with either an only child or all children. `ETree_transform()` can merge a front with any subset of its children.)

   Error checking: If `etree` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `maxfrontsize` ≤ 0, an error message is printed and the program exits.

19.2.11 Parallel factorization map methods

This family of methods create a map from the fronts to processors or threads, used in a parallel factorization.

1. ```
   IV * ETree_wrapMap ( ETree *etree, int type, int symflag, DV *cumopsDV ) ;
   IV * ETree_balancedMap ( ETree *etree, int type,
                            int symflag, DV *cumopsDV ) ;
   IV * ETree_subtreeSubsetMap ( ETree *etree, int type,
                                 int symflag,DV *cumopsDV ) ;
   IV * ETree_ddMap ( ETree *etree, int type, int symflag,
                      DV *cumopsDV, double cutoff ) ;
   IV * ETree_ddMapNew ( ETree *etree, int type, int symflag,
                         IV *msIV, DV *cumopsDV ) ;
   ```

   These methods construct and return an `IV` object that contains the map from fronts to threads. The size of the input `cumopsDV` object is the number of threads or processors. On output, `cumopsDV` contains the number of factor operations performed by the threads or processors for a fan-in factorization. The `type` parameter can be one of `SPOOLES_REAL` or `SPOOLES_COMPLEX`. `symflag` must be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   - The simplest map is the *wrap map*, where front `J` is assigned to thread or processor `J % nthread`.
   - The *balanced map* attempts to balance the computations across the threads or processes, where the fronts are visited in a post-order traversal of the tree and a front is assigned to a thread or processor with the least number of accumulated operations thus far.
   - The *subtree-subset map* is the most complex, where subsets of threads or processors are assigned to subtrees via a pre-order traversal of the tree. (Each root of the tree can be assigned to all processors.) The tree is then visited in a post-order traversal, and each front is assigned to an eligible thread or processor with the least number of accumulated ops so far.
   - The *domain decomposition map* is also complex, where domains are mapped to threads, then the fronts in the schur complement are mapped to threads, both using independent balanced maps. The method `ETree_ddMapNew()` is more robust than `ETree_ddMap()`, and is more general in the sense that it takes a multisector vector as input. The `msIV` object is a map from the vertices to $\{0, 1\}$. A vertex mapped to 0 lies in the Schur complement, a vertex mapped to 1 lies in a domain.

   Error checking: If `etree` or `cumopsDV` is `NULL`, or if `type` or `symflag` is invalid, an error message is printed and the program exits.

19.2.12 Storage profile methods

These methods fill a vector with the total amount of working storage necessary during the factor and solves.

1. ```
   void ETree_MFstackProfile ( ETree *etree, int type, double dvec[] ) ;
   ```

   On return, `dvec[J]` contains the amount of active storage to eliminate `J` using the multifrontal method and the natural post-order traversal. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `dvec` are `NULL`, or if `symflag` is invalid, an error message is printed and the program exits.

2. ```
   void ETree_GSstorageProfile ( ETree *etree, int type, IVL *symbfacIVL,
                                 int *vwghts, double dvec[] ) ;
   ```

   On return, `dvec[J]` contains the amount of active storage to eliminate `J` using the left-looking general sparse method and the natural post-order traversal. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `dvec` are `NULL`, or if `symflag` is invalid, an error message is printed and the program exits.
   ```

3. `void ETree_FSstorageProfile ( ETree *etree, int type, IVL *symbfacIVL,`
   `double dvec[] ) ;`

   On return, `dvec[J]` contains the amount of active storage to eliminate `J` using the right-looking forward sparse method and the natural post-order traversal. The `symflag` parameter can be one of `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

   Error checking: If `etree` or `dvec` are `NULL`, or if `symflag` is invalid, an error message is printed and the program exits.

4. `void ETree_forwSolveProfile ( ETree *etree, double dvec[] ) ;`

   On return, `dvec[J]` contains the amount of stack storage to solve for `J` using the multifrontal-based forward solve.

   Error checking: If `etree` or `dvec` are `NULL`, an error message is printed and the program exits.

5. `void ETree_backSolveProfile ( ETree *etree, double dvec[] ) ;`

   On return, `dvec[J]` contains the amount of stack storage to solve for `J` using the multifrontal-based backward solve.

   Error checking: If `etree` or `dvec` are `NULL`, an error message is printed and the program exits.

19.2.13 IO methods

There are the usual eight IO routines. The file structure of a tree object is simple: `nfront`, `nvtx`, a `Tree` object followed by the `nodwghtsIV`, `bndwghtsIV` and `vtxToFrontIV` objects.

1. `int ETree_readFromFile ( ETree *etree, char *fn ) ;`

   This method reads an `ETree` object from a file whose name is stored in `*fn`. It tries to open the file and if it is successful, it then calls `ETree_readFromFormattedFile()` or `ETree_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `etree` or `fn` are `NULL`, or if `fn` is not of the form `*.etreef` (for a formatted file) or `*.etreeb` (for a binary file), an error message is printed and the method returns zero.

2. `int ETree_readFromFormattedFile ( ETree *etree, FILE *fp ) ;`

   This method reads an `ETree` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `etree` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int ETree_readFromBinaryFile ( ETree *etree, FILE *fp ) ;`

   This method reads an `ETree` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `etree` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int ETree_writeToFile ( ETree *etree, char *fn ) ;`

   This method writes an `ETree` object to a file whose name is stored in `*fn`. An attempt is made to open the file and if successful, it then calls `ETree_writeFromFormattedFile()` for a formatted file, or `ETree_writeFromBinaryFile()` for a binary file. The method then closes the file and returns the value returned from the called routine.

   Error checking: If `etree` or `fn` are `NULL`, or if `fn` is not of the form `*.etreef` (for a formatted file) or `*.etreeb` (for a binary file), an error message is printed and the method returns zero.

5. `int ETree_writeToFormattedFile ( ETree *etree, FILE *fp ) ;`

   This method writes an `ETree` object to a formatted file. Otherwise, the data is written to the file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `etree` or `fp` are `NULL`, an error message is printed and zero is returned.

6. `int ETree_writeToBinaryFile ( ETree *etree, FILE *fp ) ;`

   This method writes an `ETree` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `etree` or `fp` are `NULL`, an error message is printed and zero is returned.

7. `int ETree_writeForHumanEye ( ETree *etree, FILE *fp ) ;`

   This method writes an `ETree` object to a file in a readable format. Otherwise, the method `ETree_writeStats()` is called to write out the header and statistics. Then the parent, first child, sibling, node weight and boundary weight values are printed out in five columns. The value `1` is returned.

   Error checking: If `etree` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int ETree_writeStats ( ETree *etree, FILE *fp ) ;`

   This method write a header and some statistics to a file. The value `1` is returned.

   Error checking: If `etree` or `fp` are `NULL` an error message is printed and zero is returned.

19.3 Driver programs for the `ETree` object

This section contains brief descriptions of the driver programs.

1. `createETree msglvl msgFile inGraphFile inPermFile outIVfile outETreeFile`

   This driver program reads in a `Graph` object and a `Perm` permutation object and creates a front tree `ETree` object. The map from vertices to fronts is optionally written out to `outIVfile`. The `ETree` object is optionally written out to `outETreeFile`.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `inPermFile` parameter is the input file for the `Perm` object. It must be of the form `*.permf` or `*.permb`. The `Perm` object is read from the file via the `Perm_readFromFile()` method.
   - The `outIVfile` parameter is the output file for the vertex-to-front map `IV` object. If `outIVfile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outIVfile` is of the form `*.ivf`), or a binary file (if `outIVfile` is of the form `*.ivb`).
   - The `outETreeFile` parameter is the output file for the `ETree` object. If `outETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outETreeFile` is of the form `*.etreef`), or a binary file (if `outETreeFile` is of the form `*.etreeb`).

2. `extractTopSep msglvl msgFile inETreeFile outIVfile`

   This driver program creates an `IV` object that contains a `compids[]` vector, where `compids[v] = 0` if vertex `v` is in the top level separator and `-1` otherwise. The `IV` object is optionally written out to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
   - The `outIVfile` parameter is the output file for the vertex-to-front map `IV` object. If `outIVfile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outIVfile` is of the form `*.ivf`), or a binary file (if `outIVfile` is of the form `*.ivb`).

3. `mkNDETree msglvl msgFile n1 n2 n3 maxzeros maxsize outFile`

   This program constructs a front tree for a Laplacian operator on a regular grid ordered using nested dissection. When `n3 = 1`, the problem is two dimensional and a 9-point operator is used. When `n3 > 1`, the problem is three dimensional and a 27-point operator is used. A sequence of five ETree objects are produced:

   - vertex elimination tree
   - fundamental supernode front tree
   - front tree after trying to merge with an only child
   - front tree after trying to merge with all children
   - front tree after splitting large fronts

   The merging and splitting process are controlled by the `maxzeros` and `maxsize` parameters. Here is some typical output for a $15 \times 15 \times 15$ grid matrix with `maxzeros = 64` and `maxsize = 32`.

```
vtx tree :  3375 fronts,   367237 indices,   367237 |L|, 63215265 ops
fs tree  :  1023 fronts,    39661 indices,   367237 |L|, 63215265 ops
merge1   :  1023 fronts,    39661 indices,   367237 |L|, 63215265 ops
merge2   :   511 fronts,    29525 indices,   373757 |L|, 63590185 ops
split    :   536 fronts,    34484 indices,   373757 |L|, 63590185 ops
```

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of grid points in the first direction.
   - `n2` is the number of grid points in the second direction.
   - `n3` is the number of grid points in the third direction.
   - The `maxzeros` parameter is an upper bound on the number of logically zero entries that will be allowed in a new front.
   - The `maxsize` parameter is an upper bound on the number of vertices in a front — any original front that contains more than `maxsize` vertices will be broken up into smaller fronts.

- The `outFile` parameter is the output file for the `ETree` object. If `outFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.etreef`), or a binary file (if `outFile` is of the form `*.etreeb`).

4. `mkNDoutput msglvl msgFile n1 n2 n3 maxzeros maxsize nthread maptype cutoff outETreeFile outMapFile`

   This program constructs a front tree for a Laplacian operator on a regular grid ordered using nested dissection. When `n3 = 1`, the problem is two dimensional and a 9-point operator is used. When `n3 > 1`, the problem is three dimensional and a 27-point operator is used. The front tree is generated in the same fashion as done by the `mkNDETree` driver program. Using this front tree, an `IV` object that maps fronts to processors is then created using one of four different kinds of maps.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of grid points in the first direction.
   - `n2` is the number of grid points in the second direction.
   - `n3` is the number of grid points in the third direction.
   - The `maxzeros` parameter is an upper bound on the number of logically zero entries that will be allowed in a new front.
   - The `maxsize` parameter is an upper bound on the number of vertices in a front — any original front that contains more than `maxsize` vertices will be broken up into smaller fronts.
   - The `nthread` parameter is the number of threads.
   - The `maptype` parameter is the type of map.
     - `1` — wrap map
     - `2` — balanced map
     - `3` — subtree-subset map
     - `4` — domain decomposition map
   - The `cutoff` parameter is used by the domain decomposition map only. Try setting `cutoff = 1/nthread` or `cutoff = 1/(2*nthread)`.
   - The `outETreeFile` parameter is the output file for the `ETree` object. If `outETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outETreeFile` is of the form `*.etreef`), or a binary file (if `outETreeFile` is of the form `*.etreeb`).
   - The `outMapFile` parameter is the output file for the `IV` map object. If `outMapFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outMapFile` is of the form `*.ivf`), or a binary file (if `outMapFile` is of the form `*.ivb`).

5. `permuteETree msglvl msgFile inETreeFile inEqmapIVfile outETreeFile outIVfile`

   This driver program is used to get an old-to-new permutation vector from an `ETree` object and permute the vertices in the `ETree` object. The program has the ability to handle an `ETree` object that is defined on a compressed graph. If `inEqmapIVfile` is not `none`, the program reads in an `IV` object that contains the equivalence map, i.e., the map from the degrees of freedom to the vertices in the compressed graph. This map is used to expand the `ETree` object.

- The msglvl parameter determines the amount of output — taking msglvl >= 3 means the ETree object is written to the message file.
- The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The inETreeFile parameter is the input file for the ETree object. It must be of the form *.etreef or *.etreeb. The ETree object is read from the file via the ETree_readFromFile() method.
- The inEqmapIVfile parameter is the input file for the equivalence map IV object. It must be of the form *.ivf, *.ivb, or none. If inEqmapIVfile is not none, the IV object is read from the file via the IV_readFromFile() method.
- The outETreeFile parameter is the output file for the ETree object. If outETreeFile is none then the ETree object is not written to a file. Otherwise, the ETree_writeToFile() method is called to write the object to a formatted file (if outETreeFile is of the form *.etreef), or a binary file (if outETreeFile is of the form *.etreeb).
- The outIVFile parameter is the output file for the old-to-new IV object. If outIVFile is none then the IV object is not written to a file. Otherwise, the IV_writeToFile() method is called to write the object to a formatted file (if outIVFile is of the form *.ivf), or a binary file (if outIVFile is of the form *.ivb).

6. testExpand msglvl msgFile inETreeFile inEqmapFile outETreeFile

   This driver program is used to translate an ETree object for a compressed graph into an ETree object for the unit weight graph.

   - The msglvl parameter determines the amount of output — taking msglvl >= 3 means the ETree object is written to the message file.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The inETreeFile parameter is the input file for the ETree object for the compressed graph. It must be of the form *.etreef or *.etreeb. The ETree object is read from the file via the ETree_readFromFile() method.
   - The inEqmapFile parameter contains the map from vertices in the unit weight graph into vertices in the compressed graph. It must be of the form *.ivf or *.ivb. The IV object is read from the file via the IV_readFromFile() method.
   - The outETreeFile parameter is the output file for the ETree object for the unit weight graph. If outETreeFile is none then the ETree object is not written to a file. Otherwise, the ETree_writeToFile() method is called to write the object to a formatted file (if outETreeFile is of the form *.etreef), or a binary file (if outETreeFile is of the form *.etreeb).

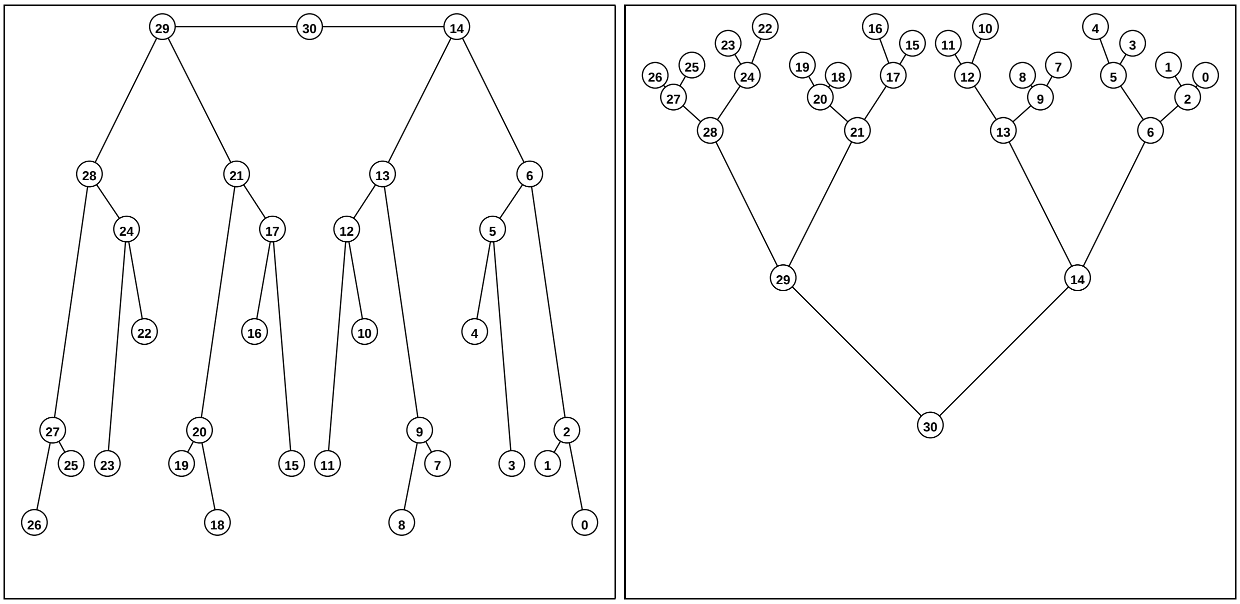
7. testFS msglvl msgFile inETreeFile labelflag radius firstEPSfile secondEPSfile

   This driver program investigates the storage requirements for a limited storage forward sparse factorization. It first reads in a front tree object and for each front J, it determines two quantities: (1) the amount of in-core storage necessary to factor $\widehat{J}$ and its boundary, and (2) the amount of in-core storage necessary to factor J, $\text{par}(J)$, $\text{par}^2(J)$, etc. The program then creates two EPS files, written to firstEPSfile and secondEPSfile. See Figure 19.1 for an example.

   - The msglvl parameter determines the amount of output — taking msglvl >= 3 means the ETree object is written to the message file.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.

- The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- If `labelflag = 1`, the node ids are written on the nodes in the two plots.
- Each node will have a circle with radius `radius`.
- The `firstEPSfile` and `secondEPSfile` parameters is the output EPS file for the two plots.

Figure 19.1: GRD7x7: Working storage for the forward sparse factorization of the nested dissection ordering. On the left is the storage required to factor $\widehat{J}$ and its update matrix. On the right is the storage required to factor J and all of its ancestors. Both plots have the same scale.



8. `testHeight msglvl msgFile inETreeFile`

   This driver program computes the height of the front tree with respect to factor storage. This quantity is the minimum amount of working storage for a forward sparse factorization.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.

9. `testIO msglvl msgFile inFile outFile`

   This driver program reads and writes `ETree` files, useful for converting formatted files to binary files and vice versa. One can also read in a `ETree` file and print out just the header information (see the `ETree_writeStats()` method).

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- The `outFile` parameter is the output file for the `ETree` object. If `outFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.etreef`), or a binary file (if `outFile` is of the form `*.etreeb`).

10. `testMaps msglvl msgFile inETreeFile outIVfile nthread type cutoff`

    This program is used to construct an owners `IV` that maps a front to its owning thread or process. The owners map `IV` object is optionally written to a file.

    - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
    - The `outIVFile` parameter is the output file for the owners map `IV` object. If `outIVFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outIVFile` is of the form `*.ivf`), or a binary file (if `outIVFile` is of the form `*.ivb`).
    - The `nthread` parameter specifies the number of threads or processes to be used.
    - The `type` parameter specifies the type of multisector.
      - `type == 1` — use `ETree_wrapMap()` to compute a wrap mapping.
      - `type == 2` — use `ETree_balancedMap()` to compute a balanced mapping.
      - `type == 3` — use `ETree_subtreeSubset()` to compute a subtree-subset mapping.
      - `type == 4` — use `ETree_ddMap()` to compute a domain decomposition map.
    - `cutoff` is a cutoff value for the multisector used only for the domain decomposition map. The cutoff defines the multisector, $0 \leq$ `cutoff` ≤ 1. If front `J` has a subtree metric based on forward operations that is greater than or equalt to `cutoff` times the total number of operations, then front `J` belongs to the multisector.

11. `testMS msglvl msgFile inETreeFile outIVfile flag cutoff`

    This program is used to extract a multisector from a front tree `ETree` object. It partitions the vertices into domains and a multisector, where each domain is a subtree of the elimination tree and the multisector is the rest of the vertices. The choice of the subtrees depends on the `flag` and `cutoff` parameters — it can be based on depth of a subtree or the number of vertices, factor entries or factor operations associated with the subtree. The component ids `IV` object is optionally written to a file. Here is some sample output for `BCSSTK30` ordered by nested dissection, where the multisector is defined by subtree vertex weight (`flag = 2`) with `cutoff = 0.125`.

```
 region  vertices    entries   operations   metric/(avg domain)
     0       1671     597058    255691396  0.797  2.201  3.967
     1       3104     255341     33205237  1.481  0.941  0.515
     2       3222     457255    116441261  1.537  1.685  1.806
     3       1514     194916     41940202  0.722  0.718  0.651
```

```
  4       2057     333186    100212056  0.981  1.228  1.555
  5         77       5040       356454  0.037  0.019  0.006
  6       1750     266166     62607526  0.835  0.981  0.971
  7       1887     325977    101994905  0.900  1.202  1.582
  8       3405     492662    125496320  1.624  1.816  1.947
  9       3413     501150    141423868  1.628  1.847  2.194
 10       3242     320220     51679456  1.546  1.180  0.802
 11       2118     238011     44427959  1.010  0.877  0.689
 12       1454     136777     18166107  0.694  0.504  0.282
 13         10        106         1168  0.005  0.000  0.000

                  nvtx   %          nzf   %           ops    %
domains          27253 94.22   3526807 85.52    837952519 76.620
schur complement  1671  5.78    597058 14.48    255691396 23.380
total            28924         4123865         1093643915
```

Region 0 is the Schur complement, and there are thirteen domains, eleven of good size. A perfectly balanced tree would have eight domains using `cutoff` equal to 1/8. It is interesting to see that the Schur complment contains only six per cent of the vertices but almost one quarter the number of operations.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- The `outIVFile` parameter is the output file for the `IV` object. If `outIVFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outIVFile` is of the form `*.ivf`), or a binary file (if `outIVFile` is of the form `*.ivb`).
- The `flag` parameter specifies the type of multisector.
  - `flag == 1` — the multisector is based on the depth of the front, i.e., if the front is more than `depth` steps removed from the root, it forms the root of a domain.
  - `flag == 2` — the multisector is based on the number of vertices in a subtree, i.e., if the subtree rooted at a front contains more than `cutoff` times the total number of vertices, it is a domain.
  - `flag == 3` — the multisector is based on the number of factor entries in a subtree, i.e., if the subtree rooted at a front contains more than `cutoff` times the total number of factor entries, it is a domain.
  - `flag == 4` — the multisector is based on the number of factor operations in a subtree, i.e., if the subtree rooted at a front contains more than `cutoff` times the total number of factor operations, it is a domain.
- `cutoff` is a cutoff value for the multisector, see above description when `flag` equals `1`, `2` or `3`.

12. `testStats msglvl msgFile inETreeFile labelflag radius firstEPSfile secondEPSfile`

    This driver program computes one of five metrics associated with a front tree and writes an EPS file that illustrates the metric overlaid on the tree structure. It first reads in a front tree object and a graph file. There are six possible plots:

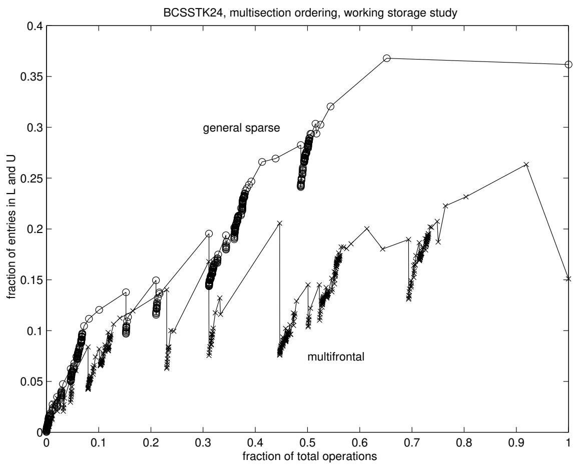
`metricType`	type of metric
0	no metric, just a tree plot
1	# of nodes in a front
2	# of original matrix entries in a front
3	# of factor matrix entries in a front
4	# of forward factor operations in a front
5	# of backward factor operations in a front

The maximum value of the metric creates a circle with radius `rmax`, and all other nodes have circles with their area relative to this largest circle. See Figure 19.2 contains four plots, each used `heightflag = 'D'`, `coordflag = 'P'`, `rmax = 20` and `labelflag = 0`.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `outEPSfile` parameter is the name of the EPS file to hold the tree.
- The `metricType` parameter defines the type of metric to be illustrated. See above for values.
- For information about the `heightflag` and `coordflag` parameters, see Section 25.2.9.
- If `labelflag = 1`, the node ids are written on the nodes in the two plots.
- The `fontscale` parameter is the font size when labels are drawn.

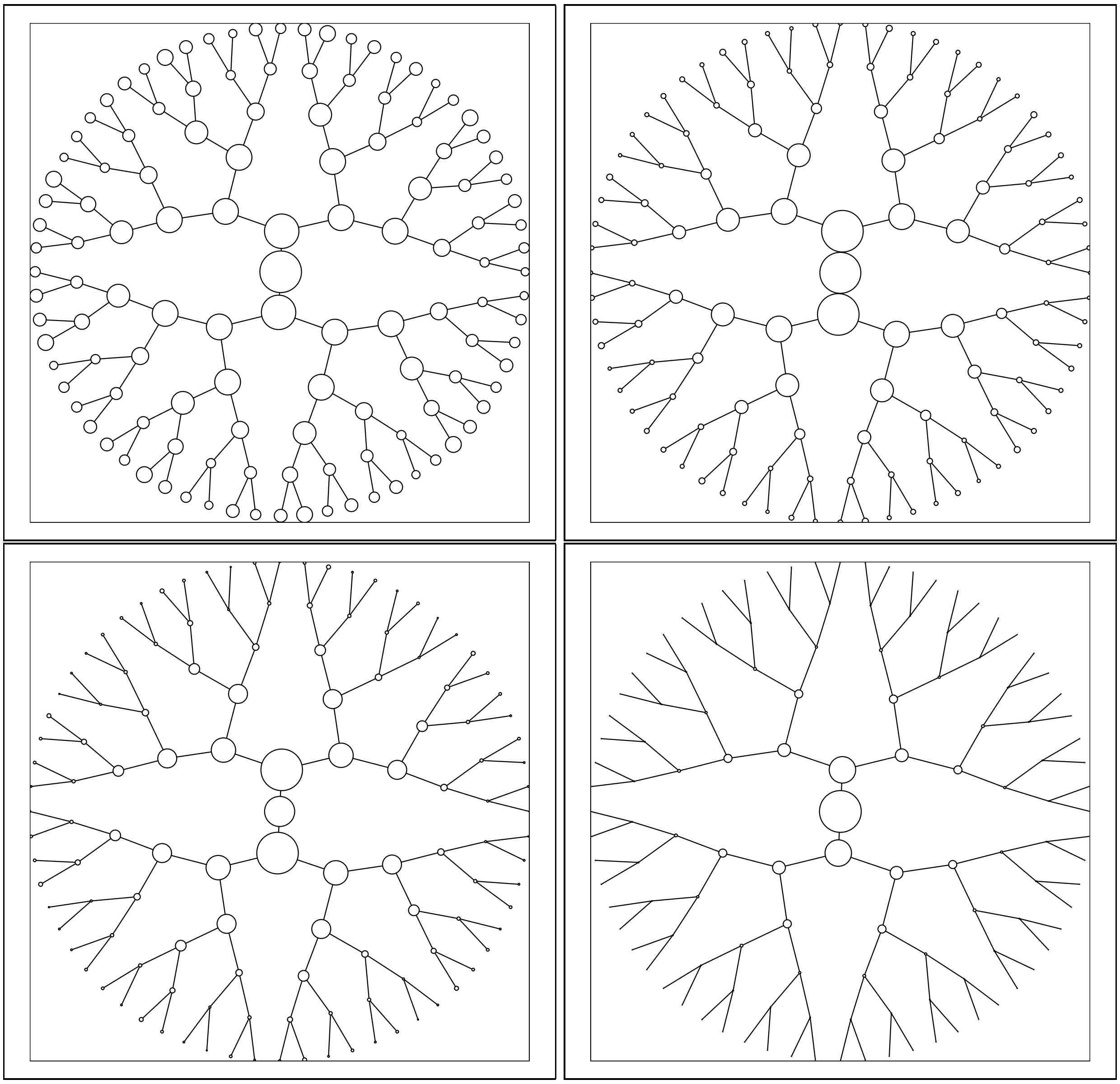
13. `testStorage msglvl msgFile inETreeFile inGraphFile`

    This driver program is used to evaluate the working storage for the left-looking general sparse and multifrontal algorithms using the natural post-order traversal of the front tree. The output is in matlab format to produce a plot. An example is found below.



- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.

Figure 19.2: GRD7x7x7: Four tree plots for a $7 \times 7 \times 7$ grid matrix ordered using nested dissection. The top left tree measure number of original matrix entries in a front. The top right tree measure number of factor matrix entries in a front. The bottom left tree measure number of factor operations in a front for a forward looking factorization, e.g., forward sparse. The bottom right tree measure number of factor operations in a front for a backward looking factorization, e.g., general sparse.



- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.

14. `testTransform msglvl msgFile inETreeFile inGraphFile`
    `outETreeFile maxzeros maxsize seed`

    This driver program is used to transform a front tree `ETree` object into a (possibly) merged and (possibly) split front tree. *Merging* the front tree means combining fronts together that do not introduce more than `maxzeros` zero entries in a front. (See [4] and [10] for a description of this supernode amalgamation or relaxation.) *Splitting* a front means breaking a front up into a chain of smaller fronts; this allows more processors to work on the original front in a straightforward manner. The new front tree is optionally written to a file. Here is some output for the `R3D13824` matrix using `maxzeros = 1000` and `maxsize = 64`.

```
                 CPU  #fronts  #indices  #entries       #ops
original  :              6001    326858   3459359  1981403337
merge one :    0.209     3477    158834   3497139  2000297117
merge all :    0.136      748     95306   3690546  2021347776
merge any :    0.073      597     85366   3753241  2035158539
split     :    0.202      643    115139   3753241  2035158539
final     :    3.216      643    115128   3752694  2034396840
```

Note how the number of fronts, front indices, factor entries and factor operations change after each step. Merging chains (the `merge one` line) halves the number of fronts while increasing operations by 1%. Merging all children when possible (the `merge all` line) reduces the number of fronts by a factor of 5 while increasing operations by another 1%. Merging any other children (the `merge any` line) has another additional effect. Splitting the fronts increases the number of fronts slightly, but appears not to change the factor entries or operation counts. This is false, as the final step computes the symbolic factorization for the last front tree and updates the boundary sizes of the fronts. We see that the number of indices, entries and factor operations actually decrease slightly due to the split fronts.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `ETree` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `outETreeFile` parameter is the output file for the `ETree` object. If `outETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outETreeFile` is of the form `*.etreef`), or a binary file (if `outETreeFile` is of the form `*.etreeb`).
- The `maxzeros` parameter is an upper bound on the number of logically zero entries that will be allowed in a new front.

- The `maxsize` parameter is an upper bound on the number of vertices in a front — any original front that contains more than `maxsize` vertices will be broken up into smaller fronts.
- `seed` is a seed for a random number generator.

Chapter 20

`GPart`: Graph Partitioning Object

The `GPart` object is used to create a partition of a graph. We use an explicit vertex separator to split a graph (or a subgraph) into the separator and two or more connected components. This process proceeds recursively until the subgraphs are too small to split (given by some user-supplied parameter).

At present, there is one path for splitting a graph (or a subgraph).

- Find a *domain decomposition* of the graph. The graph's vertices V are partitioned into *domains*, $\Omega_1, \ldots, \Omega_m$, each a connected component, and the interface vertices Φ. The boundary of a domain Ω_i (those vertices not in the domain but adjacent to a vertex in the domain), written $\text{adj}(\Omega_i)$, are a subset of Φ, the interface vertices. We use the term *multisector* for Φ, for it generalizes the notion of bisector.

  We currently find the domain decomposition by growing domains from random seed vertices. Upper and lower bounds are placed on the weights of the domains.

- Given a domain decomposition of the graph $\langle \Phi, \Omega_1, \ldots, \Omega_m \rangle$, we find a *2-set partition* $[S, B, W]$ of the vertices, where $S \subseteq \Phi$, $\text{Adj}(B) \subseteq S$ and $\text{Adj}(W) \subseteq S$. Note, it may be the case that B and/or W are not connected components.

  We currently find a 2-set partition by forming a *domain-segment* bipartite graph where the segments partition the interface nodes Φ. We use a block Kernighan-Lin method to find an edge separator of this domain-segment graph. Since the "edges" are segments, an edge separator of the domain-segment graph is truly a vertex separator of the original graph.

- Given a 2-set decomposition $[S, B, W]$ of the graph, we improve the partition by *smoothing* S. The goal is to decrease the size of S, or improve the balance of the two sets (minimize $||B| - |W||$, or both. Our present approach is to generate a *wide separator* Y where $S \subseteq Y$ and try to find a separator $\widehat{S} \subseteq Y$ that induces a better partition $[\widehat{S}, \widehat{B}, \widehat{W}]$.

  To do this, we form a network and solve a max flow problem. The nodes in $B \setminus Y$ are condensed into the *source* while the nodes in $W \setminus Y$ are condensed into the *sink*. The rest of the network is formed using the structure of the subgraph induced by Y. Given a *min-cut* of the network we can identify a separator $\widehat{S} \subseteq Y$ that has minimal weight. We examine two (possibly) different min-cuts and evaluate the partitions induced via their minimal weight separators, and accept a better partition if present.

This process we call `DDSEP`, which is short for `D`*omain* `D`*ecomposition* `SEP`*arator*, explained in more detail in [5] and [6].

20.1 Data Structures

The `GPart` structure has a pointer to a `Graph` object and other fields that contain information about the partition of the graph.

The following fields are always active.

- `Graph *graph` : pointer to the `Graph` object
- `int nvtx` : number of internal vertices of the graph
- `int nvbnd` : number of boundary vertices of the graph
- `int ncomp` : number of components in the graph
- `IV compidsIV` : an `IV` object that holds the component ids of the internal vertices — `compids[v] == 0` means that the vertex is in the separator or multisector.
- `IV cweightsIV` : an `IV` object that holds the component weights — `cweights[icomp]` stores the weight of component `icomp`, `cweights[0]` is the separator or multisector weight.
- `int msglvl` : message level parameter. When `msglvl = 0`, no output is produced. When `msglvl = 1`, only "scalar" output is provided, no vectors are printed or any print statements in a loop. When `msglvl > 1`, beware, there can be a fair amount of output.
- `FILE *msgFile` : message file pointer, default value is `stdout`.

The following fields are used when building a domain/separator tree during the recursive dissection process.

- `int id` : id of the partition object
- `GPart *par` : pointer to a parent `GPart` object
- `GPart *fch` : pointer to a first child `GPart` object
- `GPart *sib` : pointer to a sibling `GPart` object
- `IV vtxMapIV` : an `IV` object of size `nvtx + nvbnd`, contains a map from the vertices of the graph to either the vertices of its parent or to the vertices of the root graph

The `DDsepInfo` *helper*-object is used during the `DDSEP` recursive bisection process. It contains input parameters for the different stages of the `DDSEP` algorithm, and collects statistics about the CPU time spent in each stage.

- These parameters are used to generate the domain decomposition.
  - `int minweight`: minimum target weight for a domain
  - `int maxweight`: maximum target weight for a domain
  - `double freeze`: multiplier used to freeze vertices of high degree into the multisector. If the degree of `v` is more than `freeze` times the median degree, `v` is placed into the multisector.
  - `int seed`: random number seed
  - `int DDoption`: If `1`, a new domain decomposition is constructed for each subgraph. If `2`, a domain decomposition is constructed for the original graph, and its projection onto a subgraph is used to define the domain decomposition on the subgraph.
- These parameters are used to find the initial and final bisectors.

- `double alpha`: cost function parameter
- `int seed`: random number seed
- `int nlayer`: number of layers to use to form a wide separator Y from a 2-set partition $[S, B, W]$. If `nlayer = 1` or `2`, $Y = S \cup (Adj(S) \cap B)$ or $Y = S \cup (Adj(S) \cap W)$. When `nlayer = 1` the network is forced to be bipartite. If `nlayer = 3`, $Y_3 = S \cup Adj(S)$, and for `nlayer = 2k+1`, $Y_{2k+1} = Y_{2k-1} \cup Adj(Y_{2k-1})$.

• These parameters accumulate CPU times.

- `double cpuDD`: time to construct the domain decompositions
- `double cpuMap`: time to construct the maps from vertices to domains and segments
- `double cpuBPG`: time to construct the domain/segment bipartite graphs
- `double cpuBKL`: time to find the initial separators via the Block Kernighan-Lin algorithm on the domain/segment graphs
- `double cpuSmooth`: time to smooth the bisectors
- `double cpuSplit`: time to split the subgraphs
- `double cpuTotal`: total cpu time

• Miscellaneous parameters.

- `int maxcompweight`: an attempt is made to split any subgraph that has weight greater than `maxcompweight`.
- `int ntreeobj`: number of tree objects in the tree, used to set `gpart->id` and used to initialize the `DSTree` object.
- `int msglvl` : message level
- `FILE *msgFile` : message file pointer

20.2 Prototypes and descriptions of GPart methods

This section contains brief descriptions including prototypes of all methods that belong to the `GPart` object. There are no IO methods.

20.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `GPart * GPart_new ( void ) ;`

   This method simply allocates storage for the `GPart` structure and then sets the default fields by a call to `GPart_setDefaultFields()`.

2. `void GPart_setDefaultFields ( GPart *gpart ) ;`

   This method sets the structure's fields to default values: `id = -1`, `nvtx = nvbnd = ncomp = 0`, `g` = `par` = `fch` = `sib` = `NULL`, and the default fields for `compidsIV`, `cweightsIV` and `vtxMapIV` are set via calls to `IV_setDefaultFields()`.

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

3. `void GPart_clearData ( GPart *gpart ) ;`

   The `IV_clearData()` method is called for the `compidsIV`, `cweightsIV` and `vtxMapIV` objects. The structure's fields are then set with a call to `GPart_setDefaultFields()`. Note, storage for the `Graph` object `gpart->graph` is **not** free'd. The `GPart` object does not own its `Graph` object, it only uses it.

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

4. `void GPart_free ( GPart *gpart ) ;`

   This method releases any storage by a call to `GPart_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

20.2.2 Initializer methods

There are two initializer methods.

1. `void GPart_init ( GPart *gpart, Graph *graph ) ;`

   This method initializes the `Gpart` object given a `Graph` object as input. Any previous data is cleared with a call to `GPart_clearData()`. The `graph`, `nvtx`, `nvbnd` fields are set. The `compidsIV` and `cweightsIV` `IV` objects are initialized. The remaining fields are not changed from their default values.

   Error checking: If `gpart` or `g` is `NULL`, or if `g->nvtx` ≤ 0, an error message is printed and the program exits.

2. `void GPart_setMessageInfo ( GPart *gpart, int msglvl, FILE *msgFile ) ;`

   This method sets the `msglvl` and `msgFile` fields.

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

20.2.3 Utility methods

1. `void GPart_setCweights ( GPart *gpart ) ;`

   This method sets the component weights vector `cweightsIV`. We assume that the `compidsIV` vector has been set prior to entering this method. The weight of a component is not simply the sum of the weights of the vertices with that component's id. We accept the separator or multisector vertices (those `v` with `compids[v] == 0`) but then find the connected components of the remaining vertices, renumbering the `compidsIV` vector where necessary. Thus, `ncomp` and `compidsIV` may be updated, and `cweightsIV` is set.

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

2. `int GPart_sizeOf ( GPart *gpart ) ;`

   This method returns the number of bytes owned by the object. This includes the structure itself, the `compidsIV`, `cweightsIV` and `vtxMapIV` arrays (if present), but `not` the `Graph` object.

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

3. `int GPart_validVtxSep ( GPart *gpart ) ;`

   This method returns 1 if the partition defined by the `compidsIV` vector has a valid vertex separator and zero otherwise. When there is a valid vertex separator, there are no adjacent vertices not in the multisector that belong to different components (as defined by the `compidsIV` vector).

   Error checking: If `gpart` is `NULL`, an error message is printed and the program exits.

4. `void GPart_split ( GPart *gpart ) ;`

   This method is used to split a subgraph during the nested dissection process that builds a tree of `GPart` objects. We first generate a valid partition via the `GPart_setCweights()` method, and then split the graph into its component subgraphs. Each subgraph is assigned to a new child `GPart` object. The `Graph` object for each subgraph is formed from the parent graph using the `Graph_subGraph()` method. This means that the storage for the adjacency lists of the subgraph is taken from the storage for the parent graph — the lists are mapped into the local ordering via the `vtxMap` vector. After `GPart_split(gpart)` is called, the adjacency lists for the vertices in `gpart->g` are no longer valid.

   Error checking: If `gpart` or `g` is `NULL`, or if `gpart->fch` is not `NULL` (meaning that the subgraph has already been split), an error message is printed and the program exits.

5. `int GPart_vtxIsAdjToOneDomain ( GPart *gpart, int v, int *pdomid ) ;`

   This method determines whether the vertex `v` is adjacent to just one domain or not. We use this method to make a separator or multisector minimal. If the vertex is adjacent to only one domain, the return value is `1` and `*pdomid` is set to the domain's id. If a vertex is adjacent to zero or two or more domains, the return value is zero. If a vertex belongs to a domain, it is considered adjacent to that domain.

   Error checking: If `gpart`, `g` or `domid` is `NULL`, or if `v` is out of range (i.e., $\texttt{v} < 0$ or $\texttt{nvtx} \le \texttt{v}$), an error message is printed and the program exits.

6. `IV * GPart_bndWeightsIV ( GPart *gpart ) ;`

   This method returns an `IV` object that contains the weights of the vertices on the boundaries of the components.

   Error checking: If `gpart` or `g` is `NULL`, an error message is printed and the program exits.

20.2.4 Domain decomposition methods

There are presently two methods that create a domain decomposition of a graph or a subgraph.

1. ```
   void GPart_DDviaFishnet ( GPart *gpart, double frac, int minweight,
                             int maxweight, int seed ) ;
   ```

   This method generates a domain decomposition of a graph using the *fishnet* algorithm (see [5] for details). On return, the `compidsIV` vector is filled with component ids and `ncomp` is set with the number of domains. The `frac` parameter governs the exclusion of nodes of high degree from the domain sets. We have found this to be useful for some graphs. Nodes of very high degree (relative to the average or mean degree) can severely distort a domain decomposition. We have found that setting `frac` to four works well in practice. The `minweight` and `maxweight` parameters are the minimum target weight and maximum target weight for a domain. The `seed` parameter is used to insert a degree of randomness into the algorithm. This allows us to make several runs and take the best partition.

   Error checking: If `gpart` or `g` is `NULL`, or if $\texttt{freeze} \le 0.0$, or if $\texttt{minweight} < 0$, or if $\texttt{maxweight} < 0$, or if $\texttt{minweight} \ge \texttt{maxweight}$, an error message is printed and the program exits.

2. `void GPart_DDviaProjection ( GPart *gpart, IV *DDmapIV ) ;`

   This method generates a domain decomposition for a subgraph by projecting an existing domain decoposition for the original graph onto the subgraph. Using this method (as opposed to generating a domain decomposition for each subgraph) can typically save 15% of the overall time to find the graph decomposition, though the resulting partition is usually not as good.

   Error checking: If `gpart` or `DDmapIV` is `NULL`, an error message is printed and the program exits.
   ```

20.2.5 Methods to generate a 2-set partition

These two methods are used to generate a 2-set partition $[S, B, W]$ from a domain decomposition.

1. ```
   double GPart_TwoSetViaBKL ( GPart *gpart, double alpha, int seed,
                               double cpus[] ) ;
   ```

   This method takes a domain decomposition $\{\Phi, \Omega_1, \ldots, \Omega_m\}$ defined by the `compidsIV` vector and generates a two set partition $[S, B, W]$. We first compute the map from vertices to domains and segments (the segments partition the interface nodes Φ). We then construct the bipartite graph that represents the connectivity of the domains and segments. Each segment is an "edge" that connects two "adjacent" domains. This allows us to use a variant of the Kernighan-Lin algorithm to find an "edge" separator formed of segments, which is really a vertex separator, a subset of Φ. The `alpha` parameter is used in the cost function evaluation for the partition, $\mathrm{cost}([S, B, W]) = |S| \left(1 + \alpha \frac{\max\{|B|, |W|\}}{\min\{|B|, |W|\}}\right)$. The `seed` parameter is used to randomize the algorithm. One can make several runs with different seeds and chose the best partition. The `cpus[]` array is used to store execution times for segments of the algorithm: `cpus[0]` stores the time to compute the domain/segment map; `cpus[2]` stores the time to create the domain/segment bipartite graph; `cpus[3]` stores the time to find the bisector using the block Kernighan-Lin algorithm.

   Error checking: If `gpart` or `cpus` is `NULL`, an error message is printed and the program exits.

2. `IV * GPart_domSegMap ( GPart *gpart, int *pndom, int *pnseg ) ;`

   This method takes a domain decomposition as defined by the `compidsIV` vector and generates the map from the vertices to the domains and segments that are used in the Block Kernighan-Lin procedure to find an initial separator. The map is returned in an `IV` object, and the numbers of domains and segments are set in the `pndom` and `pnseg` addresses. This method is called by `GPart_TwoSetViaBKL`.

   Error checking: If `gpart`, `g`, `pndom` or `pnseg` is `NULL`, an error message is printed and the program exits.

20.2.6 Methods to improve a 2-set partition

These methods are used to improve a 2-set partition $[S, B, W]$. They hinge on finding a *wide separator* Y and constructing a better separator $\widehat{S} \subseteq Y$. The `alpha` parameter is used in the cost function $\mathrm{cost}([S, B, W]) = |S| \left(1 + \alpha \frac{\max\{|B|, |W|\}}{\min\{|B|, |W|\}}\right)$.

1. `IV * GPart_identifyWideSep ( GPart *gpart, int nlayer1, int nlayer2 ) ;`

   This method takes a 2-set partition $[S, B, W]$ and identifies a *wide separator* Y that contains S. The portions of B and W that are included in Y are specified using the `nlayer1` and `nlayer2` parameters. If both are zero, then Y is simply S. If `nlayer1 > 0`, then Y contains all vertices in the first component whose distance is `nlayer1` or less from S, and similarly for `nlayer2 > 0`. The vertices in Y are placed in an `IV` object which is then returned.

   Error checking: If `gpart` or `g` is `NULL`, or if `nlevel1` < 0 or `nlevel2` < 0, an error message is printed and the program exits.

2. `IV * GPart_makeYCmap ( GPart *gpart, IV *YVmapIV ) ;`
   ```

This method contructs and returns an IV object that is the blueprint used to form the network. The wide separator Y can be partitioned into four disjoint sets (though some may be empty):

$$\begin{aligned} Y_0 &= \{y \in Y \mid y \notin Adj(B \setminus Y) \text{ and } y \notin Adj(W \setminus Y)\} \\ Y_1 &= \{y \in Y \mid y \in Adj(B \setminus Y) \text{ and } y \notin Adj(W \setminus Y)\} \\ Y_2 &= \{y \in Y \mid y \notin Adj(B \setminus Y) \text{ and } y \in Adj(W \setminus Y)\} \\ Y_3 &= \{y \in Y \mid y \in Adj(B \setminus Y) \text{ and } y \in Adj(W \setminus Y)\} \end{aligned}$$

The YVmapIV object contains the list of vertices in the wide separator Y. The IV object that is returned, (called YCmapIV in the calling method) contains the subscripts of the Y_0, Y_1, Y_2 or Y_3 sets that contains each vertex.

Error checking: If gpart, g or YVmapIV is NULL, or if nvtx ≤ 0, or if YVmapIV is empty, an error message is printed and the program exits.

3. `void * GPart_smoothBy2layers ( GPart *gpart, int bipartite, float alpha ) ;`

   This method forms the wide separator Y from two layers of vertices, either $Y_B = S \cup (Adj(S) \cap B)$ or $Y_W = S \cup (Adj(S) \cap W)$. (If $|B| \ge |W|$, we first look at Y_B and if no improvement can be made we look at Y_W, and the reverse if $|W| > |B|$.) The bipartite parameter defines the type of network problem we solve. The network induced by the wide separator Y need not be bipartite, and will not be bipartite if $Y_0 \neq \emptyset$ or $Y_3 \neq \emptyset$, (Y_0 and Y_3 are defined immediately above). The Y_3 set is not important, since it must be included in any separator $\widehat{S} \subseteq Y$. When Y_0 is not empty, it forms a layer *between* Y_1 and Y_2, and so the network is not bipartite. We can force the network to be bipartite (set bipartite to 1) by moving all nodes in Y_0 to the appropriate Y_1 or Y_2. When the graph is unit-weight and the network is bipartite, we can use the Dulmage-Mendelsohn decomposition to find one or more separators of minimum weight. In general, forcing a non-bipartite network to be bipartite results in possibly a larger separator, so we do not recommend this option. The capability is there to compare the Dulmage-Mendelsohn decomposition smoothers with the more general (and robust) max flow smoothers.

   Error checking: If gpart is NULL, or if alpha < 0.0, an error message is printed and the program exits.

4. ```
   float * GPart_smoothYSep ( GPart *gpart, IV *YVmapIV,
                              IV *YCmapIV, float alpha ) ;
   ```

   This methods takes as input a 2-set partition $[S, B, W]$ (defined by gpart->compidsIV), a wide separator Y (defined by YVmapIV) and a $\langle Y_0, Y_1, Y_2, Y_3\rangle$ partition of Y (defined by YCmapIV) and attempts to find a better partition. A max flow problem is solved on a network induced by $\langle Y_0, Y_1, Y_2, Y_3\rangle$. Two min-cuts and the partitions they induce are examined and the better partition is accepted if better than $[S, B, W]$. The parameter alpha is used in the partition's cost function, and the cost of the best partition is returned.

   Error checking: If gpart, YVmapIV or YCmapIV is NULL, or if alpha < 0.0, an error message is printed and the program exits.

5. `float * GPart_smoothBisector ( GPart *gpart, int nlayer, float alpha ) ;`

   This method takes a two-set partition $[S, B, W]$ as defined by the compidsIV vector and improves it (if possible). The methods returns the cost of a (possibly) new two-set partition $[\widehat{S}, \widehat{B}, \widehat{W}]$ defined by the compidsIV vector. The wide separator Y that is constructed is *centered* around S, i.e., Y includes all nodes in B and W that are nlayer distance or less from S. This method calls GPart_smoothYSep().

   Error checking: If gpart is NULL, or if nlevel < 0, or if alpha < 0.0, an error message is printed and the program exits.
   ```

20.2.7 Recursive Bisection method

There is presently one method to construct the domain/separator tree.

1. `DSTree * GPart_RBviaDDsep ( GPart *gpart, DDsepInfo *info ) ;`

   This method performs a recursive bisection of the graph using the `DDSEP` algorithm and returns a `DSTree` object that represents the domain/separator tree and the map from vertices to domains and separators. The `DDsepInfo` structure contains all the parameters to the different steps of the `DDSEP` algorithm (the fishnet method to find the domain decomposition, the Block Kernighan-Lin method to find an initial separator, and solves a max flow problem to improve the separator). An attempt is made to split a subgraph if the weight of the internal vertices of the subgraph exceeds `info->maxcompweight`. The cpu times for the different segments of the algorithm are accumulated in fields of the `DDsepInfo` object.

   Error checking: If `gpart` or `info` is `NULL`, or if `nvtx` ≤ 0, an error message is printed and the program exits.

20.2.8 `DDsepInfo` methods

The `DDsepInfo` *helper*-object is used during the `DDSEP` recursive bisection process. It stores the necessary input parameters for the different stages of the `DDSEP` algorithm and collects statistics about the resulting partition.

1. `DDsepInfo * DDsepInfo_new ( void ) ;`

   This method simply allocates storage for the `DDsepInfo` structure and then sets the default fields by a call to `DDsepInfo_setDefaultFields()`.

2. `void DDsepInfo_setDefaultFields ( DDsepInfo *info ) ;`

   This method checks to see whether `info` is `NULL`. If so, an error message is printed and the program exits. Otherwise, the structure's fields are set to the following default values.

   ```
   info->seed          =   1 ; info->cpuDD     =    0.0 ;
   info->minweight     =  40 ; info->cpuMap    =    0.0 ;
   info->maxweight     =  80 ; info->cpuBPG    =    0.0 ;
   info->frac          = 4.0 ; info->cpuBKL    =    0.0 ;
   info->alpha         = 1.0 ; info->cpuSmooth =    0.0 ;
   info->maxcompweight = 500 ; info->cpuSplit  =    0.0 ;
   info->ntreeobj      =   0 ; info->cpuTotal  =    0.0 ;
   info->DDoption      =   1 ; info->msglvl    =      0 ;
   info->nlayer        =   3 ; info->msgFile   = stdout ;
   ```

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

3. `void DDsepInfo_clearData ( DDsepInfo *info ) ;`

   This method checks to see whether `info` is `NULL`. `DDsepInfo_setDefaultFields()` is called to set the default values.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

4. `void DDsepInfo_free ( DDsepInfo *info ) ;`

   This method checks to see whether `info` is `NULL`. If so, an error message is printed and the program exits. Otherwise, it releases any storage by a call to `DDsepInfo_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

5. `void DDsepInfo_writeCpuTimes ( DDsepInfo *info, FILE *msgFile ) ;`

   This method writes a breakdown of the CPU times in a meaningful format. Here is sample output.

```
CPU breakdown for graph partition
              raw CPU   per cent
misc       :     1.61    1.2%
Split      :    24.68   17.7%
find DD    :    12.13    8.7%
DomSeg Map :    13.09    9.4%
DomSeg BPG :     4.66    3.3%
BKL        :     5.68    4.1%
Smooth     :    77.83   55.7%
Total      :   139.67  100.0%
```

   Error checking: If `info` or `msgFile` is `NULL`, an error message is printed and the program exits.

20.3 Driver programs for the `GPart` object

This section contains brief descriptions of four driver programs.

1. `testDDviaFishnet msglvl msgFile inGraphFile freeze minweight maxweight seed outIVfile`

   This driver program constructs a domain decomposition via the *fishnet* algorithm [5]. It reads in a `Graph` object from a file, finds the domain decomposition using the four input parameters, then optionally writes out the map from vertices to components to a file.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the output file — if `msgFile` is `stdout`, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `freeze` parameter is used to place nodes of high degree into the multisector. If the external degree of a vertex is `freeze` times the average degree, then it is placed in the multisector.
   - The *target* minimum weight for a domain is `minweight`.
   - The *target* maximum weight for a domain is `maxweight`.
   - The `seed` parameter is a random number seed.
   - The `outIVfile` parameter is the output file for the `IV` object that contains the map from vertices to components. If `outIVfile` is `"none"`, then there is no output, otherwise `outIVfile` must be of the form `*.ivf` or `*.ivb`.

2. `testTwoSetViaBKL msglvl msgFile inGraphFile inIVfile seed alpha outIVfile`

   This driver program constructs a two-set partition via the Block Kernighan-Lin algorithm [5]. It reads in a `Graph` object and an `IV` object that holds the map from vertices to components (e.g., the output from the driver program `testDDviaFishet`) from two files, constructs the domain-segment graph and finds an initial separator, then optionally writes out the new map from vertices to components to a file.

- The msglvl parameter determines the amount of output.
- The msgFile parameter determines the output file — if msgFile is stdout, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The inGraphFile parameter is the input file for the Graph object. It must be of the form *.graphf or *.graphb. The Graph object is read from the file via the Graph_readFromFile() method.
- The inIVfile parameter is the input file for the IV object that contains the map from vertices to domains and multisector. It inIVfile must be of the form *.ivf or *.ivb.
- The seed parameter is a random number seed.
- The alpha parameter controls the partition evaluation function.
- The outIVfile parameter is the output file for the IV object that contains the map from vertices to separator and the two components. If outIVfile is "none", then there is no output, otherwise outIVfile must be of the form *.ivf or *.ivb.

3.
```
testSmoothBisector msglvl msgFile inGraphFile inIVfile
                   option alpha outIVfile
```

   This driver program smooths a bisector of a graph by solving a sequence of max-flow network problems. It reads in a Graph object and an IV object that holds the map from vertices to components (e.g., the output from the driver program testTwoSetViaBKL) from two files, smooths the separator and then optionally writes out the new component ids map to a file.

   - The msglvl parameter determines the amount of output.
   - The msgFile parameter determines the output file — if msgFile is stdout, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The inGraphFile parameter is the input file for the Graph object. It must be of the form *.graphf or *.graphb. The Graph object is read from the file via the Graph_readFromFile() method.
   - The inIVfile parameter is the input file for the IV object that contains the map from vertices to domains and multisector. It inIVfile must be of the form *.ivf or *.ivb.
   - The option parameter specifies the type of network optimization problem that will be solved.
     - option = 1 — each network has two layers and is bipartite.
     - option = 2 — each network has two layers but need not be bipartite.
     - option = 2 — each network has option/2 layers on each side of the separator.
   - The alpha parameter controls the partition evaluation function.
   - The outIVfile parameter is the output file for the IV object that contains the map from vertices to separator and the two components. If outIVfile is "none", then there is no output, otherwise outIVfile must be of the form *.ivf or *.ivb.

4.
```
testRBviaDDsep msglvl msgFile inGraphFile seed minweight maxweight
               freeze alpha maxdomweight DDoption nlayer
testRBviaDDsep2 msglvl msgFile inGraphFile nruns seed minweight maxweight
               freeze alpha maxdomweight DDoption nlayer
```

   These driver programs construct a multisector via recursive bisection and orders the graph using nested dissection and multisection using the multisector. testRBviaDDsep executes only one run while testRBviaDDsep2 executes nruns runs with random permutations of the graph.

   - The msglvl parameter determines the amount of output.
   - The msgFile parameter determines the output file — if msgFile is stdout, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.

- The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `nruns` parameter is the number of runs made with the graph randomly permuted.
- The `seed` parameter is a random number seed.
- The *target* minimum weight for a domain is `minweight`.
- The *target* maximum weight for a domain is `maxweight`.
- The `freeze` parameter is used to place nodes of high degree into the multisector. If the external degree of a vertex is `freeze` times the average degree, then it is placed in the multisector.
- The `alpha` parameter controls the partition evaluation function.
- The `maxdomweight` parameter controls the recursive bisection — no subgraph with weight less than `maxdomweight` is further split.
- The `DDoption` parameter controls the initial domain/segment partition on each subgraph. When `DDDoption = 1` we use the fishnet algorithm for each subgraph. When `DDDoption = 1` we use the fishnet algorithm once for the entire graph and this is then projected down onto each subgraph.
- The `nlayer` parameter governs the smoothing process by specifying the type of network optimization problem that will be solved.
  - `nlayer = 1` — each network has two layers and is bipartite.
  - `nlayer = 2` — each network has two layers but need not be bipartite.
  - `nlayer > 2` — each network has `option/2` layers on each side of the separator.

5. ```
   mkDSTree msglvl msgFile inGraphFile seed minweight maxweight
            freeze alpha maxdomweight DDoption nlayer outDSTreeFile
   ```

   This driver program constructs a domain/separator tree using recursive bisection. The `DSTree` object is optionally written to a file.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the output file — if `msgFile` is `stdout`, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `seed` parameter is a random number seed.
   - The *target* minimum weight for a domain is `minweight`.
   - The *target* maximum weight for a domain is `maxweight`.
   - The `freeze` parameter is used to place nodes of high degree into the multisector. If the external degree of a vertex is `freeze` times the average degree, then it is placed in the multisector.
   - The `alpha` parameter controls the partition evaluation function.
   - The `maxdomweight` parameter controls the recursive bisection — no subgraph with weight less than `maxdomweight` is further split.
   - The `DDoption` parameter controls the initial domain/segment partition on each subgraph. When `DDDoption = 1` we use the fishnet algorithm for each subgraph. When `DDDoption = 1` we use the fishnet algorithm once for the entire graph and this is then projected down onto each subgraph.
   - The `nlayer` parameter governs the smoothing process by specifying the type of network optimization problem that will be solved.
     - `nlayer = 1` — each network has two layers and is bipartite.
   ```

  - `nlayer = 2` — each network has two layers but need not be bipartite.
  - `nlayer > 2` — each network has `option/2` layers on each side of the separator.
- The `outDSTreeFile` parameter is the output file for the `DSTree` object. It must be of the form `*.dstreef` or `*.dstreeb`. If `outDSTreeFile` is not `"none"`, the `DSTree` object is written to the file via the `DSTree_writeToFile()` method.

Chapter 21

`Graph`: A Graph object

The `Graph` object is used to represent the graph of a matrix. The representation uses a set of adjacency lists, one edge list for each vertex in the graph, and is implemented using an `IVL` object.[1] For the `Graph` object, the vertices and the edges can be either unit weight or non-unit weight independently. None of the algorithms in the package *at present* use weighted edges, though most use weighted vertices. The weighted edges capability is there, and the weighted edges are also stored using an `IVL` object.

The `Graph` object is not too sophisticated, i.e., we chose **not** to implement a method to find a separator of a graph inside this object. Such complex functionality is best left to higher level objects, and our method based on domain decomposition [5] is found in the `GPart` object.

A graph can also be a subgraph of another graph — nested dissection is the natural recursive partition of a graph — and it pays to use the knowledge of the boundary of a subgraph. We chose not to implement a "sub"-graph object separately from a graph object, thus our `Graph` object can have a boundary. One specifies `nvtx`, the number of internal vertices, and `nvbnd`, the number of external or boundary vertices. The labels for internal vertices are found in `[0, nvtx)` and those for boundary vertices are found in `[nvtx, nvtx+nvbnd)`.

It is easy to create a `Graph` object: one specifies the number of internal and boundary vertices, the type of graph (weighted or unit weight vertices and edges), and then uses the methods for the `IVL` object to add adjacency lists and (possibly) lists of edge weights. The `Graph` object relies strongly on the `IVL` object.

Weighted graphs are commonly used in partitioning and ordering algorithm, and they normally arise from *compressing* the graph in some manner. Let us write the unit weight graph as $G(V, E)$ and the weighted graph as $\mathbf{G}(\mathbf{V}, \mathbf{E})$, and let $\phi : V \mapsto \mathbf{V}$ be the map from unit weight vertices to weighted vertices. Let u and v be vertices and (u, v) be an edge in $G(V, E)$, and let $\mathbf{u}$ and $\mathbf{v}$ be vertices and $(\mathbf{u}, \mathbf{v})$ be an edge in $\mathbf{G}(\mathbf{V}, \mathbf{E})$. The weight of a vertex is $w(\mathbf{u})$, the number of unit weight vertices in the weighted vertex. The weight of an edge is $w(\mathbf{u}, \mathbf{v})$, the number of (u, v) edges in the unit weight graph where $u \in \mathbf{u}$ and $v \in \mathbf{v}$.

The natural compressed graph [3], [9] is very important for many matrices from structral analysis and computational fluid mechanics. This type of graph has one special property:

$$w(\mathbf{u}, \mathbf{v}) = w(\mathbf{u}) \cdot w(\mathbf{v})$$

and it is the smallest graph with this property. The compression is *loss-less*, for given $\mathbf{G}(\mathbf{V}, \mathbf{E})$ and ϕ, we can reconstruct the unit weight graph $G(V, E)$. In effect, we can work with the natural compressed graph to find separators and orderings and map back to the unit weight graph. The savings in time and space can be considerable.

The `Graph` object has a method to find the ϕ map for the natural compressed graph; it requires $O(|V|)$ space and $O(|E|)$ time. There is a method to compress a graph (i.e., given $G(V, E)$ and an arbitrary ϕ,

[1] The `EGraph` object represents a graph of the matrix, but stores a list of covering cliques in an `IVL` object.

construct $\mathbf{G}(\mathbf{V},\mathbf{E})$) and a method to expand a graph (i.e., given $\mathbf{G}(\mathbf{V},\mathbf{E})$ and an arbitrary ϕ, construct $G(V,E)$).

There are several utility methods to return information about the memory in use by the `Graph object`, to access adjacency lists and edge weight lists, and to provide information about the connected components of a graph.

21.1 Data Structure

The `Graph` structure has nine fields.

- `int type` : type of graph

`type`	vertices weighted?	edges weighted?
0	no	no
1	yes	no
2	no	yes
3	yes	yes

- `int nvtx` : number of internal vertices
- `int nvbnd` : number of boundary vertices
- `int nedges` : number of edges
- `int totvwght` : total vertex weight
- `int totewght` : total edge weight
- `IVL *adjIVL` : pointer to `IVL` object to hold adjacency lists
- `int *vwghts` : pointer to a vertex to hold vertex weights non-`NULL` if `type % 2 == 1`
- `IVL *ewghtIVL` : pointer to `IVL` object to hold edge weight lists, non-`NULL` if `type / 2 == 1`

21.2 Prototypes and descriptions of `Graph` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Graph` object.

21.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Graph * Graph_new ( void ) ;`

   This method simply allocates storage for the `Graph` structure and then sets the default fields by a call to `Graph_setDefaultFields()`.

2. `void Graph_setDefaultFields ( Graph *graph ) ;`

   This method sets the structure's fields to default values: `type`, `nvtx`, `nvbnd`, `nedges`, `totwght` and `totewght` are all zero, and `adjIVL`, `vwghts` and `ewghtIVL` are all `NULL`.

   Error checking: If `graph` is `NULL`, an error message is printed and the program exits.

3. `void Graph_clearData ( Graph *graph ) ;`

   This method clears the data for the object. If `adjIVL` is not `NULL`, then `IVL_free(adjIVL)` is called to free the `IVL` object. If `ewghtIVL` is not `NULL`, then `IVL_free(ewghtIVL)` is called to free the `IVL` object. If `vwghts` is not `NULL`, then `IVfree(vwghts)` is called to free the `int` vector. The structure's fields are then set to their default values with a call to `Graph_setDefaultFields()`.

   Error checking: If `graph` is `NULL`, an error message is printed and the program exits.

4. `void Graph_free ( Graph *graph ) ;`

   This method releases any storage by a call to `Graph_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `graph` is `NULL`, an error message is printed and the program exits.

21.2.2 Initializer methods

There are three initializer methods. The first is most commonly used, the second is used within the IO routines, and the third is used to create a `Graph` object from the `offsets[]/adjncy[]` format for the adjacency structure.

1. ```
   void Graph_init1 ( Graph *graph, int type, int nvtx, int nvbnd, int nedges,
                      int adjType, int ewghtType ) ;
   ```

   This is the basic initializer method. Any previous data is cleared with a call to `Graph_clearData()`. Then the scalar fields are set and the `adjIVL` object is initialized. If `type` is `1` or `3`, the `vwghts` vector is initialized to zeros. If `type` is `2` or `3`, the `ewghtIVL` object is initialized.

   Error checking: If `graph` is `NULL`, `type` is invalid (`type` must be in `[0,3]`), `nvtx` is non-positive, `nvbnd` or `nedges` is negative, or `adjType` of `ewghtType` is invalid (they must be `IVL_CHUNKED`, `IVL_SOLO` or `IVL_UNKNOWN`). an error message is printed and the program exits.

2. ```
   void Graph_init2 ( Graph *graph, int type, int nvtx, int nvbnd, int nedges,
         int totvwght, int totewght, IVL *adjIVL, int *vwghts, IVL *ewghtIVL)
   ```

   This method is used by the IO read methods. When a `Graph` object is read from a file, the `IVL` object(s) must be initialized and then read in from the file. Therefore, we need an initialization method that allows us to set pointers to the `IVL` objects and the `vwghts` vector. Note, `adjIVL`, `vwghts` and `ewghtIVL` are owned by the `Graph` object and will be free'd when the `Graph` object is free'd.

   Error checking: If `graph` or `adjIVL` is `NULL`, `type` is invalid (`type` must be in `[0,3]`), `nvtx` is non-positive, `nvbnd` or `nedges` is negative, or if `type % 2 = 1` and `vwghts` is `NULL`, or if `type` ≥ 2 and `ewghtIVL` is `NULL`, an error message is printed and the program exits.

3. ```
   void Graph_fillFromOffsets ( Graph *graph, int neqns, int offsets[],
                                int adjncy[], int flag )
   ```

   This method initializes the `Graph` object using an adjacency structure, as is the storage format for a Harwell-Boeing matrix. The entries in list `v` are found in `adjncy[i1:i2]`, where `i1 = offsets[v]` and `i2 = offsets[v+1]-1`. The `offsets[]` and `adjncy[]` arrays are assumed to be zero-based (as are C-arrays), not one-based (as are Fortran arrays). If `flag == 0` then the lists are simply loaded into the `Graph` object. If `flag == 1`, the adjacency structure must be upper, meaning that the list for `v` contains entries that are greater than or equal to `v`. The `Graph` will have a full adjacency structure, including the `(v,v)` edges.

   Error checking: If `graph`, `offsets` or `adjncy` is `NULL`, or if `neqns` ≤ 0, or if `flag` < 0 or if `flag` > 1, an error message is printed and the program exits.
   ```

4. `void Graph_setListsFromOffsets ( Graph *graph, int neqns,`
   `int offsets[], int adjncy[] ) ;`

   This method initializes the `Graph` object using a *full* adjacency structure. The entries in list `v` are found in `adjncy[i1:i2]`, where `i1 = offsets[v]` and `i2 = offsets[v+1]-1`. The `offsets[]` and `adjncy[]` arrays are assumed to be zero-based (as are C-arrays), not one-based (as are Fortran arrays). Use this method with caution — the adjacency list for vertex `v` must contain `v` and *all* vertices it is adjacent to. Note, new storage for the adjacency lists is not allocated, the `Graph` object's `IVL` object points into the storage in `adjncy[]`.

   Error checking: If `graph`, `offsets` or `adjncy` is `NULL`, or if `neqns` ≤ 0, an error message is printed and the program exits.

21.2.3 Compress and Expand methods

These three methods find an equivalence map for the natural compressed graph, compress a graph, and expand a graph.

1. `IV * Graph_equivMap ( Graph *graph ) ;`

   This method constructs the equivalence map from the graph to its natural compressed graph. The map $\phi : V \mapsto \mathbf{V}$ is then constructed (see the Introduction in this section) and put into an `IV` object that is then returned.

   Error checking: If `graph` is `NULL` or `nvtx <= 0`, an error message is printed and the program exits.

2. `Graph * Graph_compress  ( Graph *graph, int map[], int coarseType ) ;`
   `Graph * Graph_compress2 ( Graph *graph, IV *mapIV, int coarseType ) ;`

   This `Graph` and map objects (`map[]` or `mapIV`) are checked and if any errors are found, the appropriate message is printed and the program exits. The compressed graph object is constructed and returned. Note, the compressed graph does not have a boundary, even though the original graph may have one.

   Error checking: If `graph`, `map` or `mapIV` is `NULL`, or if `nvtx` ≤ 0, or if `coarseType` < 0, or if $3 <$ `coarseType`, an error message is printed and the program exits.

3. `Graph * Graph_expand  ( Graph *graph, int nvtxbig, int map[] ) ;`
   `Graph * Graph_expand2 ( Graph *graph, IV *mapIV ) ;`

   This `Graph` and map objects (`map[]` or `mapIV`) are checked and if any errors are found, the appropriate message is printed and the program exits. The expanded unit weight graph object is constructed and returned.

   Error checking: If `graph`, `map` or `mapIV` is `NULL`, or if `nvtxbig` ≤ 0, an error message is printed and the program exits.

21.2.4 Wirebasket domain decomposition ordering

1. `void Graph_wirebasketStages ( Graph *graph, IV *stagesIV, int radius ) ;`

   This method is used to group the vertices into stages that is suitable for a wirebasket domain decomposition of a general graph. On input, `stages[v] = 0` means that `v` is in a domain. On output, `stages[v]` contains the stage of elimination — zero is for all vertices in the domains. If `stages[v] > 0`, then it is the number of domains that are found within `radius` edges of `v`.

   Error checking: If `graph` or `stagesIV` is `NULL`, or if `radius < 0`, an error message is printed and the program exits.

21.2.5 Utility methods

1. `int Graph_sizeOf ( Graph *graph ) ;`

   This method returns the number of bytes taken by this object.

   Error checking: If `graph` is `NULL`, an error message is printed and the program exits.

2. `Graph_externalDegree ( Graph *graph, int v ) ;`

   This method returns the weight of adj(v).

   Error checking: If `graph` is `NULL`, or `v` is out of range, an error message is printed and the program exits.

3. `int Graph_adjAndSize ( Graph *graph, int u, int *pusize, int **puadj) ;`

   This method fills `*pusize` with the size of the adjacency list for `u` and `*puadj` points to the start of the list vector.

   Error checking: If `graph` is `NULL`, or if `u < 0` or `u >= nvtx` or if `pusize` or `puadj` is `NULL`, an error message is printed and the program exits.

4. ```
   int Graph_adjAndEweights ( Graph *graph, int u, int *pusize,
                              int **puadj, int **puewghts) ;
   ```

   This method fills `*psize` with the size of the adjacency list, `*puadj` points to the start of the list vector and `*puewghts` points to the start of the edge weights vector.

   Error checking: If `graph` is `NULL`, or if `u < 0` or `u >= nvtx` or if `pusize`, `puadj` or `puewghts` is `NULL`, an error message is printed and the program exits.

5. `IV * Graph_componentMap ( Graph *graph ) ;`

   This method computes and returns an IV object that holds a map from vertices to components. The values of the map vector are in the range `[0, number of components)`.

   Error checking: If `graph` is `NULL` then an error message is printed and the program exits.

6. ```
   void Graph_componentStats ( Graph *graph, int map[],
                               int counts[], int weights[] ) ;
   ```

   This method computes some statistics about the components. The length of `map` is `nvtx`. The number of components is `1 + max(map)`, and the length of `counts[]` and `weights[]` must be as large as the number of components. On return, `counts[icomp]` and `weights[icomp]` are filled with the number of vertices and weight of the vertices in component `icomp`, respectively.

   Error checking: If `graph`, `map`, `counts` or `weights` is `NULL`, then an error message is printed and the program exits.

7. `Graph * Graph_subGraph ( Graph *graph, int icomp, int compids[], int **pmap ) ;`

   This method is used by the graph partitioning methods. For a graph $G(V, E)$, a vertex separator $S \subset V$ is found which separates the subgraph induced by $V \setminus S$ into two or more connected components. We construct a new graph object for each component using this method. The `compids[]` vector maps the internal vertices of the parent graph into components. This method extracts the subgraph associated with component `icomp`.

   There is one key design feature. *Most of the storage for the adjacency lists of the subgraph is the same as its parent graph.* This keeps us from replicating too much storage. The subgraph has internal vertices and boundary vertices (the latter contain at least part of S.) Each adjacency list for an internal vertex of the subgraph points to the corresponding adjacency list for the vertex in the parent graph. Each adjacency list for a boundary vertex of the subgraph is new storage, and only these lists are free'd when the subgraph is free'd. A map vector is created that maps the subgraphs's vertices (both internal and boundary) into the parent graph's vertices; the address of the map vector is put into `*pmap`. The adjacency lists for the subgraph are overwritten by the `map[]` vector. This renders the graph object invalid. The graph partitioning methods map the vertices back to their original values. Presently, only graphs with unit edge weights are allowed as input.

Error checking: If `graph` is `NULL` or `icomp < 0` or `compids` or `pmap` is `NULL`, an error message is printed and the program exits.

8. `int Graph_isSymmetric ( Graph *graph ) ;`

   This method returns `1` if the graph is symmetric (i.e., edge `(i,j)` is present if and only if edge `(j,i)` is present) and `0` otherwise.

   Error checking: If `graph` is `NULL`, an error message is printed and the program exits.

21.2.6 IO methods

There are the usual eight IO routines. The file structure of a `Graph` object is simple: The six scalar fields come first: `type`, `nvtx`, `nvbnd`, `nedges`, `totvwght` and `totewght`. The adjacency `IVL` structure `adjIVL` follows. If the graph has non-unit vertex weights, i.e., `type % 2 == 1`, the `vwghts` vector follows. If the graph has non-unit edge weights, i.e., `type / 2 == 1`, the `IVL` structure `ewghtIVL` follows.

1. `int Graph_readFromFile ( Graph *graph, char *fn ) ;`

   This method reads a `Graph object` from a file. It tries to open the file and if it is successful, it then calls `Graph_readFromFormattedFile()` or `Graph_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `graph` or `fn` are `NULL`, or if `fn` is not of the form `*.graphf` (for a formatted file) or `*.graphb` (for a binary file), an error message is printed and the method returns zero.

2. `int Graph_readFromFormattedFile ( Graph *graph, FILE *fp ) ;`

   This method reads a `Graph` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int Graph_readFromBinaryFile ( Graph *graph, FILE *fp ) ;`

   This method reads a `Graph` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int Graph_writeToFile ( Graph *graph, char *fn ) ;`

   This method writes a `Graph object` to a file. It tries to open the file and if it is successful, it then calls `Graph_writeFromFormattedFile()` or `Graph_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `graph` or `fn` are `NULL`, or if `fn` is not of the form `*.graphf` (for a formatted file) or `*.graphb` (for a binary file), an error message is printed and the method returns zero.

5. `int Graph_writeToFormattedFile ( Graph *graph, FILE *fp ) ;`

   This method writes a `Graph` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int Graph_writeToBinaryFile ( Graph *graph, FILE *fp ) ;`

   This method writes a `Graph` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int Graph_writeForHumanEye ( Graph *graph, FILE *fp ) ;`

   This method writes a `Graph` object to a file in a human readable format. The method `Graph_writeStats()` is called to write out the header and statistics. The value `1` is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int Graph_writeStats ( Graph *graph, FILE *fp ) ;`

   The header and statistics are written to a file. The value `1` is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

9. `int Graph_writeToMetisFile ( Graph *graph, FILE *fp ) ;`

   This method writes a `Graph` object to a file in the format of the **METIS** or **CHACO** packages. The value `1` is returned.

   Error checking: If `graph` or `fp` are `NULL` an error message is printed and zero is returned.

21.3 Driver programs for the `Graph` object

This section contains brief descriptions of six driver programs.

1. `checkComponents msglvl msgFile inGraphFile`

   This driver program reads in a `Graph` object from a file, and prints out information about the number of vertices and weights of the vertices in the components.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.

2. `compressGraph msglvl msgFile inGraphFile coarseType outMapFile outGraphFile`

   This driver program reads in a `Graph` object from a file, computes the equivalence map to its natural compressed graph (the first graph need not be unit weight), and constructs the natural compressed graph. The equivalence map and compressed graph are optionally written out to files.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `coarseType` parameter defines the type of compressed graph; valid values are in `[0,3]`.

- The `outMapFile` parameter is the output file for the `IV` object that holds the equivalence map. If `outMapFile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the `IV` object to a formatted file (if `outMapFile` is of the form `*.ivf`), or a binary file (if `outMapFile` is of the form `*.ivb`).
- The `outGraphFile` parameter is the output file for the compressed `Graph` object. If `outGraphFile` is `none` then the `Graph` object is not written to a file. Otherwise, the `Graph_writeToFile()` method is called to write the graph to a formatted file (if `outGraphFile` is of the form `*.graphf`), or a binary file (if `outGraphFile` is of the form `*.graphb`).

3. `expandGraph msglvl msgFile inGraphFile inMapFile outGraphFile`

   This driver program reads in a `Graph` object and a map `IV` object from two files. It then creates a new `Graph` object which is the original graph "expanded" by the map, and optionally writes this object to a file. The program `expandGraph` is the inverse of `compressGraph`.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `inMapFile` parameter is the input file for the `IV` object that holds the expansion map. The `IV_readFromFile()` method is called to read the map from a formatted file (if `inMapFile` is of the form `*.ivf`), or a binary file (if `inMapFile` is of the form `*.ivb`).
   - The `outGraphFile` parameter is the output file for the compressed `Graph` object. If `outGraphFile` is `none` then the `Graph` object is not written to a file. Otherwise, the `Graph_writeToFile()` method is called to write the graph to a formatted file (if `outGraphFile` is of the form `*.graphf`), or a binary file (if `outGraphFile` is of the form `*.graphb`).

4. `mkGridGraph msglvl msgFile stencil n1 n2 n3 outFile`

   This driver program creates a Graph object for a finite difference operator on a `n1` × `n2` × `n3` regular grid.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - Valid `stencil` values are `5` for a 2-D 5-point operator, `7` for a 3-D 7-point operator, `9` for a 2-D 9-point operator, `13` for a 2-D 13-point operator and `27` for a 3-D 27-point operator.
   - `n1` is the number of points in the first direction.
   - `n2` is the number of points in the second direction.
   - `n3` is the number of points in the third direction, ignored for `stencil` = `5`, `9` and `13`.
   - The `Graph` object is written to file `outFile`. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is written to the file via the `Graph_writeToFile()` method.

5. `testIO msglvl msgFile inFile outFile`

   This driver program reads in a `Graph` object from `inFile` and writes out the object to `outFile`

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Graph` object is written to the message file.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `outFile` parameter is the output file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is written to the file via the `Graph_writeToFile()` method.

6. `testIsSymmetric msglvl msgFile inFile`

   This driver program reads in a `Graph` object and tests whether it is symmetric using the `Graph_isSymmetric()` method. This was useful in one application where the `Graph` object was constructed improperly.

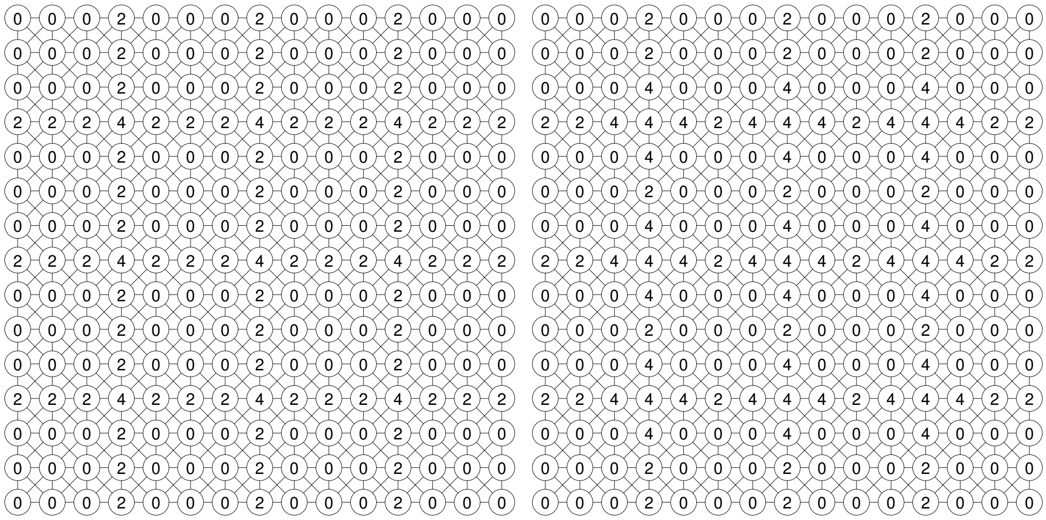
   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Graph` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.

7. `testWirebasket msglvl msgFile inGraphFile inStagesFile`
   `outStagesFile radius`

   This driver program reads in a `Graph` object and and a file that contains the stages ids of the vertices, (stage equal to zero means the vertex is in the Schur complement), and overwrites the stages vector to specify the stage that the vertex lies for a wirebasket domain decomposition of the graph. For a Schur complement vertex, its stage is precisely the number of domains that lie within `radius` edges of it. The new stages vector is written to the `outStagesFile` file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Graph` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `inStagesFile` parameter is the input file for the `IV` object that holds the component ids. It must be of the form `*.ivf` or `*.ivb`. The `IV` object is read from the file via the `IV_readFromFile()` method.
   - The `outStagesFile` parameter is the output file for the stages `IV` object. It must be of the form `*.ivf` or `*.ivb`. The `IV` object is written to the file via the `IV_writeToFile()` method.
   - The `radius` parameter is used to define the stage of a Schur complement vertex, namely the stage is the number of domains that are found within `radius` edges of the vertex.

   The two plots below illustrate the wirebasket stages for a 15×15 grid. They show the stages for `radius = 1` on the left and `radius = 2` on the right. The domains are 3×3 subgrids whose vertices have labels equal to zero.



8. `writeMetisFile msglvl msgFile inGraphFile outMetisFile`

   This driver program reads in an `Graph` object and write it out to a file in the format required by the **METIS** software package.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `outMetisFile` parameter is the outfile file for the **METIS** graph object.

Chapter 22

MSMD:
Multi-Stage Minimum Degree Object

We need an ordering for a sparse matrix. The MSMD object will provide one of three orderings:

- a minimum degree ordering, or
- a multisection ordering if given with a domain/Schur-complement partition, or
- a nested dissection ordering if given a domain/separator tree.

But in what form do we want our ordering? If all we want is a permutation vector, there is a MSMD method that will fill them. If we want more information, a method returns the ETree object that is a front tree[1] for the ordering.

The MSMD object is complex, at least in its functionality. However, its component methods are simple; one can put them together in different ways to get a wide variety of algorithms.

There are methods to eliminate a vertex or set of vertices in one of three ways:

- a single vertex, or
- a *step* of vertices, i.e., an independent set of vertices, or
- a *stage* of vertices, i.e., a set of vertices defined in a number of consecutive steps.

How to choose a vertex to eliminate is based on a *priority*, currently one of:

- external degree, or
- approximate external degree, ($\hat{d}$ from [1]) and [12], or
- half external and half approximate, ($\tilde{d}$ from [1]), or
- a constant priority (to induce maximal independent set elimination).

[1]The ETree object has the Tree object that defines the connectivity of the fronts, knows the internal and external size of each front, and has a map from the vertices to the fronts.

We intend to add more priorities, e.g., approximate deficiency from [18], [19] and [20].

Choose a priority, then specify the definition of a *step*, how to choose an independent set of vertices to eliminate at a time. Then provide a map from each vertex to the *stage* at which it will be eliminated.

Presently there is one ordering method, `MSMD_order()`. It orders the vertices by *stages*, i.e. vertices in stage k will be ordered before vertices in stage $k+1$. Inside each *stage* the vertices are ordered by *steps*. At each step an independent set of vertices is eliminated, and the choice is based on their priorities. When the ordering is finished one can extract permutation vectors of a front tree.

Here are three examples of how stages define an ordering method. (These methods are supported by the present `MSMD` object).

- Set the stage of each vertex to be zero and we have a simple minimum degree (priority) ordering.
- Given a domain/Schur complement partition or a domain/separator tree, we can find a multisection ordering by setting the stage of a vertex to be zero if it is a domain or one if it is in a separator.
- Given a domain/separator tree, we can find an incomplete nested dissection ordering by specifying the stage of a vertex to be the level of the separator or domain that contains it.

Here are three slightly more complicated examples.

- Order the vertices in the domains, then order the Schur complement graph both by nested dissection and minimum degree, and then splice the better of the two orderings together with the ordering of the domain vertices.
- Apply the above algorithm to the Schur complement graph recursively.
- Since multisection is nothing more than applying minimum degree to the Schur complement graph, randomly permute the graph and apply the minimum degree ordering. Repeat several times and take the best ordering. (Ordering the Schur complement graph is much, much less time consuming than ordering the vertices in the domains.)

Any of these three algorithms is bound to be better than both nested dissection and multisection. The tools are largely written so any of these three algorithms can be prototyped in a small amount of time and effort.

22.1 Data Structure

There are four typed objects.

- `MSMD` : the main object.
- `MSMDinfo` : an object that communicate parameter choices from the caller to the `MSMD` object and information and statistics from the `MSMD` object to the caller.
- `MSMDstageInfo` : an object that contains statistics for a stage of elimination, e.g., number of steps, number of vertices eliminated, weight of vertices eliminated, etc.
- `MSMDvtx` : an object that models a vertex.

A user needs to understand the `MSMDinfo` object, so this is where we will start our description.

22.1.1 `MSMDinfo` : define your algorithm

- `int compressFlag` – define initial and subsequent compressions of the graph.

  We compress a graph using a checksum technique. At some point in the elimination, vertices in the reach set (those adjacent to vertices just eliminated) have a checksum based on their adjacencies computed, and then vertices with the same checksum are compared to see if they are indistinguishable. This operation has a cost, and there are classes of vertices for which there is a large amount of compression, and for other classes there is little. Compression is a powerful tool, but we need a way to control it.

  - `compressFlag % 4 == 0` — do not perform any compression after each elimination step.
  - `compressFlag % 4 == 1` — after each elimination step, consider only those nodes that are *2-adjacent*, adjacent to two eliminated subtrees and having no uncovered adjacent edges.
  - `compressFlag % 4 == 2` — after each elimination step, consider all nodes.
  - `compressFlag / 4 >= 1` — compress at stage zero before any elimination.

  The default value is `1`, no initial compression and consider only 2-adjacent nodes after each elimination step.

- `int prioType` — define the priority to choose a vertex to eliminate.

  - `prioType == 0` — zero priority
  - `prioType == 1` — exact external degree for each vertex
  - `prioType == 2` — approximate external degree for each vertex ($\hat{d}$ from [1])
  - `prioType == 3` — exact external degree for 2-adjacent vertices, approximate external degree for the others

  The default value is `1`, exact external degree for each vertex.

- `double stepType` — define the elimination steps.

  - `stepType == 0` — only one vertex of minimum priority is eliminated at each step, e.g., as used in SPARSPAK's `GENQMD`, YSMP's ordering, and `AMD` [1].
  - `stepType == 1` — an independent set of vertices of minimum priority is eliminated at each step, e.g., as used in GENMMD, multiple minimum degree.
  - `stepType > 1` — an independent set of vertices is eliminated whose priorities lie between the minimum priority and the minimum priority multiplied by `stepType`.

  The default value is `1`, multiple elimination of vertices with minimum priority.

- `int seed` — a seed used for a random number generator, this introduces a necessary random element to the ordering.

- `int msglvl` – message level for statistics, diagnostics and monitoring. The default value is zero, no statistics. Set `msglvl` to one and get elimination monitoring. Increase `msglvl` slowly to get more mostly debug information.

- `FILE *msgFile` – message file, default is `stdout`.

- `int maxnbytes` – maximum number of bytes used during the ordering.

- `int nbytes` – present number of bytes used during the ordering.

- `int istage` – present stage of elimination.

- `int nstage` – number of stages of elimination.
- `MSMDstageInfo *stageInfo` – pointer to vector of `MSMDstageInfo` structures that hold information about each stage of the elimination.
- `double totalCPU` – total CPU to find the ordering.

22.1.2 `MSMD` : driver object

A user need not know anything about the internals of this object, just the methods to initialize it, order the graph, and extract the permutation vectors and/or a front tree.

- `int nvtx` — number of internal vertices in the graph.
- `IIheap *heap` – pointer to a `IIheap` object that maintains the priority queue.
- `IP *baseIP` – pointer to the base `IP` objects, used to hold subtree lists
- `IP *freeIP` – pointer to the list of free `IP` objects
- `int incrIP` – integer that holds the increment factor for the `IP` objects.
- `MSMDvtx *vertices` – pointer to vector of `MSMDvtx` objects that represent the vertices.
- `IV ivtmpIV` – `IV` object that holds an integer temporary vector.
- `IV reachIV` – `IV` object that holds the reach vector.

22.1.3 `MSMDstageInfo` : statistics object for a stage of the elimination

This object stores information about the elimination process at a stage of the elimination.

- `int nstep` — number of elimination steps in this stage
- `int nfront` — number of fronts created at this stage
- `int welim` — weight of the vertices eliminated at this stage
- `int nfind` — number of front indices
- `int nzf` — number of factor entries (for a Cholesky factorization)
- `double ops` — number of factor operations (for a Cholesky factorization)
- `int nexact2` — number of exact degree updates to 2-adjacent vertices
- `int nexact3` — number of exact degree updates to non-2-adjacent vertices
- `int napprox` — number of approximate degree updates
- `int ncheck` — number of comparisons of vertices with the same checksum during the process to find indistinguishable nodes
- `int nindst` — number of indistinguishable nodes that were detected.
- `int noutmtch` — number of nodes that were outmatched
- `double cpu` — elapsed CPU time for this stage of the elimination.

22.1.4 MSMDvtx : vertex object

This object stores information for a vertex during the elimination.

- `int id` — id of the vertex, in range `[0,nvtx)`
- `char mark` — character mark flag, `'O'` or `'X'`
- `char status` — character status of the vertex
  - `'L'` – eliminated leaf vertex
  - `'E'` – eliminated interior vertex
  - `'O'` – outmatched vertex
  - `'D'` – vertex on degree (priority) heap
  - `'R'` – vertex on reach set
  - `'I'` – vertex found to be indistinguishable to another
  - `'B'` – boundary vertex, to be eliminated in another stage
- `int stage` — stage of the vertex. Stage 0 nodes are eliminated before stage 1 nodes, etc.
- `int wght` — weight of the vertex
- `int nadj` — size of the `adj` vector
- `int *adj` — for an uneliminated vertex, `adj` points to a list of uncovered adjacent edges; for an eliminated vertex, `adj` points points to a list of its boundary vertices (only valid when the vertex is a leaf of the elimination tree or a root of a subtree of uneliminated vertices).
- `int bndwght` — for an eliminated vertex, the weight of the vertices on its boundary.
- `MSMDvtx *par` — for an eliminated vertex, `par` points to its parent vertex in the front tree (`NULL` if the vertex is the root of a subtree). For an indistinguishable vertex, `par` points to its representative vertex (which may have also been found to be indistinguishable to another).
- `IP *subtrees` — pointer to a list of `IP` objects to store the adjacent subtrees, valid only for uneliminated vertices.

22.2 Prototypes and descriptions of MSMDinfo methods

This section contains brief descriptions including prototypes of all methods that belong to the `MSMDinfo` object.

22.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `MSMDinfo * MSMDinfo_new ( void ) ;`

   This method simply allocates storage for the `MSMDinfo` structure and then sets the default fields by a call to `MSMDinfo_setDefaultFields()`.

2. `void MSMDinfo_setDefaultFields ( MSMDinfo *info ) ;`

   This method sets the structure's fields to default values.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

3. `void MSMDinfo_clearData ( MSMDinfo *info ) ;`

   This method clears any data owned by the object and then sets the structure's default fields with a call to `MSMDinfo_setDefaultFields()`.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

4. `void MSMDinfo_free ( MSMDinfo *info ) ;`

   This method releases any storage by a call to `MSMDinfo_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

22.2.2 Utility methods

There are two utility methods, one to print the object, one to check to see if it is valid.

1. `void MSMDinfo_print ( MSMDinfo *info, FILE *fp ) ;`

   This method prints out the information to a file.

   Error checking: If `info` or `fp` is `NULL`, an error message is printed and the program exits.

2. `int MSMDinfo_isValid ( MSMDinfo *info ) ;`

   This method checks that the object is valid. The return value is `1` for a valid object, `0` for an invalid object.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

22.3 Prototypes and descriptions of `MSMD` methods

This section contains brief descriptions including prototypes of all methods that belong to the `MSMD` object. The methods are loosely classified as *public* and *private*. Since the C language does not support private methods (with the exception of `static` methods within a file), specifying a method as public or private is advisory.

22.3.1 Basic methods — public

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `MSMD * MSMD_new ( void ) ;`

   This method simply allocates storage for the `MSMD` structure and then sets the default fields by a call to `MSMD_setDefaultFields()`.

2. `void MSMD_setDefaultFields ( MSMD *msmd ) ;`

   This method sets the structure's fields to default values.

   Error checking: If `msmd` is `NULL`, an error message is printed and the program exits.

3. `void MSMD_clearData ( MSMD *msmd ) ;`

   This method clears any data owned by the object, then sets the structure's default fields with a call to `MSMD_setDefaultFields()`.

   Error checking: If `msmd` is `NULL`, an error message is printed and the program exits.

4. `void MSMD_free ( MSMD *msmd ) ;`

   This method releases any storage by a call to `MSMD_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `msmd` is `NULL`, an error message is printed and the program exits.

22.3.2 Initialization methods — public

There is one initialization method.

1. `void MSMD_init ( MSMD *msmd, Graph *graph, int stages[], MSMD *info ) ;`

   This method initializes the `MSMD` object prior to an ordering. It is called by `MSMD_order()` method, and so it is currently a *private* method for the object. However, when designing more complicated ordering methods, this object is necessary to set up the data structures. There are two input arguments: `graph` is a pointer to a `Graph` object that holds the adjacency lists and weights of the vertices, and `stages` is a map from each vertex to the stage at which it is to be eliminated. (If `stages == NULL`, then all vertices will be eliminated in one stage, i.e., we order the graph using minimum degree.) Unlike much other ordering software, we do **not** destroy the adjacency structure of the graph — however we might shuffle the entries in each adjacency list.

   Error checking: If `msmd`, `graph` or `info` is `NULL`, an error message is printed and the program exits.

22.3.3 Ordering methods — public

There is currently one ordering method.

1. `void MSMD_order ( MSMD *msmd, Graph *graph, int stages[], MSMD *info ) ;`

   This method orders the vertices in the graph and maintains the `MSMDvtx` objects in a suitable representation to generate permutation vectors and/or a front tree. The input is the same as for the `MSMD_init()` method defined above.

   The method first checks that the input is valid, i.e., that `msmd`, `graph` and `info` are not `NULL` and that the `info` structure is valid by calling `MSMD_isValid()`. The `msmd` is then initialized by calling `MSMD_init()`. If called for, the graph is compressed prior to any elimination. The vertices are then eliminated by their stages via calls to `MSMD_eliminateStage()`. The overall statistics for the elimination are then computed, and then the working storage is then released, save for the `MSMDvtx` structures.

   Error checking: If `msmd`, `graph` or `info` is `NULL`, an error message is printed and the program exits.

22.3.4 Extraction methods — public

There are two methods to extract the ordering. The first fills one or two `IV` objects with the permutation vector(s). The second returns an `ETree` object that holds the front tree for the ordering.

1. `void MSMD_fillPerms ( MSMD *msmd, IV *newToOldIV, IV *oldToNewIV ) ;`

   If `newToOldIV` is not `NULL`, this method fills the `IV` object with the new-to-old permutation of the vertices, resizing the `IV` object if necessary. If `oldToNewIV` is not `NULL`, this method fills the `IV` object with the old-to-new permutation of the vertices, resizing the `IV` object if necessary.

   Error checking: If `msmd` is `NULL`, or if `newToOldIV` and `oldToNewIV` is `NULL`, an error message is printed and the program exits.

2. `ETree * MSMD_frontETree ( MSMD *msmd ) ;`

   This method constructs and returns a `ETree` object that contains the front tree for the ordering.

   Error checking: If `msmd` is `NULL`, an error message is printed and the program exits.

22.3.5 Internal methods — private

The following methods are used internally to order the graph. the user should never have any cause to call them.

1. `void MSMD_eliminateStage ( MSMD *msmd, MSMD *info ) ;`

   This method eliminates the vertices in the present stage.

   Error checking: If `msmd` or `info` is `NULL`, an error message is printed and the program exits.

2. `int MSMD_eliminateStep ( MSMD *msmd, MSMD *info ) ;`

   This method eliminates one *step* of vertices, an independent set of vertices. The return value is the weight of the vertices eliminated at this step.

   Error checking: If `msmd` or `info` is `NULL`, an error message is printed and the program exits.

3. `void MSMD_eliminateVtx ( MSMD *msmd, MSMDvtx *v, MSMD *info ) ;`

   This method eliminates vertex `v`.

   Error checking: If `msmd`, `v` or `info` is `NULL`, an error message is printed and the program exits.

4. `void MSMD_findInodes ( MSMD *msmd, MSMD *info ) ;`

   This method examines nodes in the reach set to detect indistinguishability.

   - If `info->compressFlag % 4 == 0`, there is a simple return.
   - If `info->compressFlag % 4 == 1`, only 2-adjacent nodes are examined.
   - If `info->compressFlag % 4 == 2`, all nodes are examined.

   The order of the nodes in the reach set may be permuted, but any indistinguishable nodes in the reach set are not purged from the reach set.

   Error checking: If `msmd` or `info` is `NULL`, an error message is printed and the program exits.

5. `void MSMD_cleanReachSet ( MSMD *msmd, MSMD *info ) ;`

   This method cleans the nodes in the reach set by calling `MSMD_cleanSubtreeList()` and `MSMD_clearEdgeList()`.

   Error checking: If `msmd` or `info` is `NULL`, an error message is printed and the program exits.

6. `void MSMD_cleanSubtreeList ( MSMD *msmd, MSMDvtx *v, MSMD *info ) ;`

   This method cleans the list of subtrees for vertex `v`, removing any node which is no longer the root of a subtree of eliminated nodes.

   Error checking: If `msmd`, `v` or `info` is `NULL`, an error message is printed and the program exits.

7. `void MSMD_cleanEdgeList ( MSMD *msmd, MSMDvtx *v, MSMD *info ) ;`

   This method cleans the list of uncovered edges for vertex `v`, removing any edge `(v,w)` where `v` and `w` share a common adjacent subtree.

   Error checking: If `msmd`, `v` or `info` is `NULL`, an error message is printed and the program exits.

8. `void MSMD_update ( MSMD *msmd, MSMD *info ) ;`

   This method updates the priorities of all nodes in the reach set.

   Error checking: If `msmd` or `info` is `NULL`, an error message is printed and the program exits.

9. `int MSMD_exactDegree2 ( MSMD *msmd, MSMDvtx *v, MSMD *info ) ;`

   This method computes and returns the exact external degree for vertex `v`.

   Error checking: If `msmd`, `v` or `info` is `NULL`, an error message is printed and the program exits.

10. `int MSMD_exactDegree3 ( MSMD *msmd, MSMDvtx *v, MSMD *info ) ;`

    This method computes and returns the exact external degree for vertex `v`.

    Error checking: If `msmd`, `v` or `info` is `NULL`, an error message is printed and the program exits.

11. `int MSMD_approxDegree ( MSMD *msmd, MSMDvtx *v, MSMD *info ) ;`

    This method computes and returns the approximate external degree for vertex `v`.

    Error checking: If `msmd`, `v` or `info` is `NULL`, an error message is printed and the program exits.

12. `void MSMD_makeSchurComplement ( MSMD *msmd, Graph *schurGraph, IV *VtoPhiIV ) ;`

    This method fills `schurGraph` with the graph of the Schur complement matrix (the fill graph of the uneliminated vertices) and fills `VtoPhiIV` with a map from the vertices of the original graph to the vertices of the Schur complement graph. (The mapped value of an eliminated vertex is `-1`.)

    Error checking: If `msmd`, `schurGraph` or `VtoPhiIV` is `NULL`, an error message is printed and the program exits.

22.4 Prototypes and descriptions of MSMDvtx methods

The `MSMDvtx` object is private so would not normally be accessed by the user. There is one method to print out the object.

1. `void MSMDvtx_print ( MSMDvtx *v, FILE *fp ) ;`

   This method prints a human-readable representation of a vertex, used for debugging.

   Error checking: If `v` or `fp` is `NULL`, an error message is printed and the program exits.

22.5 Driver programs for the MSMD object

This section contains brief descriptions of four driver programs.

1. ```
   orderViaMMD msglvl msgFile inGraphFile seed compressFlag prioType
      stepType outOldToNewIVfile outNewToOldIVfile outETreeFile
   ```

   This driver program orders a graph using the multiple minimum degree algorithm — exactly which algorithm is controlled by the input parameters.
   ```

- The `msglvl` parameter determines the amount of output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `seed` parameter is a random number seed.
- The `compressFlag` parameter controls the compression of the graph (identifying indistinguishable nodes) before and during the elimination process.
  - `compressFlag / 4 >= 1` — a compression step is done before elimination.
  - `compressFlag % 4 == 2` — compress after each elimination step, consider all nodes.
  - `compressFlag % 4 == 1` — compress after each elimination step, consider only 2-adjacent nodes (the most likely to form indistinguishable nodes).
  - `compressFlag % 4 == 0` — do no compression.
- The `prioType` parameter controls the type of priority that is used to choose nodes to eliminate.
  - `prioType == 1` — true external degree.
  - `prioType == 2` — approximate external degree.
  - `prioType == 3` — true external degree for 2-adjacent nodes, approximate external degree for the others.
  - `prioType == 4` — priority of each node is zero; this implies random elimination.
- The `stepType` parameter controls the type of multiple elimination to be done.
  - `stepType == 0` — one vertex eliminated at each step, like YSMP, and QMD from SPARSPAK.
  - `stepType == 1` — regular multiple elimination, e.g., GENMMD.
  - `stepType > 1` — vertices whose priority lies between the minimum priority and `stepType` times the minimum priority are eligible for elimination at a step.
- The `outOldToNewIVfile` parameter is the output file for the `IV` object that contains the old-to-new permutation vector. If `outOldToNewIVfile` is `"none"`, then there is no output, otherwise `outOldToNewIVfile` must be of the form `*.ivf` or `*.ivb`.
- The `outNewToOldIVfile` parameter is the output file for the `IV` object that contains the new-to-old permutation vector. If `outNewToOldIVfile` is `"none"`, then there is no output, otherwise `outNewToOldIVfile` must be of the form `*.ivf` or `*.ivb`.
- The `outETreeFile` parameter is the output file for the `ETree` object that contains the front tree for the ordering. If `outETreeFile` is `"none"`, then there is no output, otherwise `outETreeFile` must be of the form `*.etreef` or `*.etreeb`.

2. `orderViaND msglvl msgFile inGraphFile inDSTreeFile seed compressFlag`
   `prioType stepType outOldToNewIVfile outNewToOldIVfile outETreeFile`

   This driver program orders a graph using the incomplete nested dissection algorithm. The stages of elimination are generated by a `DSTree` domain/separator tree object that is read in from the `inDSTreeFile` file. All the other parameters are the same as for the `orderViaMMD` driver program.

3. `orderViaMS msglvl msgFile inGraphFile inDSTreeFile seed compressFlag`
   `prioType stepType outOldToNewIVfile outNewToOldIVfile outETreeFile`

   This driver program orders a graph using the multisection algorithm. The stages of elimination are generated by a `DSTree` domain/separator tree object that is read in from the `inDSTreeFile` file. All the other parameters are the same as for the `orderViaMMD` driver program.

4. `orderViaStages msglvl msgFile inGraphFile inStagesIVfile seed compressFlag prioType stepType outOldToNewIVfile outNewToOldIVfile outETreeFile`

   This driver program orders a graph using the multi-stage minimum degree algorithm. The stages of elimination are found in an `IV` object that is read in from the `inStagesIVfile` file. All the other parameters are the same as for the `orderViaMMD` driver program.

Chapter 23

`Network`: Simple Max-flow solver

First, some background on how the `Network` object is used to find a minimal weight separator. The process is rather complex.

We are given a partition of the vertices V into three disjoint sets, B, Y and W, where Y is a "wide" separator (i.e., not a minimal separator). We construct a network from this vertex partition, solve a max flow problem on this network, and then find one or more mincuts that correspond to a separator $S \subset Y$ with minimal vertex weight.

Here are the steps by which the `GPart` object contructs the network.

- All nodes in B are collapsed into the source s.
- All nodes in W are collapsed into the sink t.
- Y is partitioned into four sets:
  - Y_B are those nodes adjacent to B but not adjacent to W.
  - Y_W are those nodes adjacent to W but not adjacent to B.
  - Y_I are those nodes adjacent to neither W nor B.
  - $Y_{B,W}$ are those nodes adjacent to both W and B.

  Normally, by construction, $Y_{B,W} = \emptyset$, but the code should work fine otherwise.
- Each $y \in Y_B$ becomes one node y in the network, and the edge (s, y) has capacity $weight(y)$.
- Each $y \in Y_W$ becomes one node y in the network, and the edge (y, t) has capacity $weight(y)$.
- Each $y \in Y_I$ becomes two nodes in the network, y^- and y^+. The edge (y^-, y^+) has capacity $weight(y)$.
- An edge (x, y) where $x \in Y_B$ and $y \in Y_B$ is not found in the network. (It is not necessary.) Similarly, an edge (x, y) where $x \in Y_W$ and $y \in Y_W$ is not found in the network.
- An edge (x, y) where $x \in Y_B$ and $y \in Y_I$ becomes two edges, (x, y^-) and (y^+, x), both with infinite capacity.
- An edge (y, z) where $y \in Y_I$ and $z \in Y_W$ becomes two edges, (y^+, z) and (z, y^-), both with infinite capacity.
- An edge (x, y) where $x \in Y_I$ and $y \in Y_I$ becomes two edges, (x^+, y^-) and (y^+, x^-), both with infinite capacity.

The `Network` object can be constructed fairly simply. It is initialized by specifying the number of nodes in the network, including the source and sink. Arcs can be added one at a time and it is not necessary to know the total number of arcs ahead of time. To specify an arc one needs to provide the first and second vertices, the capacity and the present flow.

Once we have constructed the network, we solve the max flow problem in a very simple manner, basically the Ford-Fulkerson algorithm that generates augmenting paths. No doubt this can be improved, and it would be welcome because for large three dimensional finite element graphs, up to sixty per cent of the time is spent smoothing the separators, and most of this time is spent solving a max flow problem.

However, the network we generate in practice have two special properties:

- The networks are very shallow, i.e., the distance from the source to the sink is generally 3-6 in practice. This reduces the potential improvement of a pre-push algorithm.
- The maximum capacity of an edge is small, usually 6-12. Therefore scaling algorithms have little applicability.

Finding a minimal separator gives rise to networks of a special nature and that may require specialized solution techniques. In fact, there is a more straightforward approach that generates a network where each vertex in Y becomes *one* node in the network (as opposed to two network nodes for a vertex in Y_I). For this special network, all edges have infinite capacity and it is the vertices that have finite capacity. In any case, the `Network` object is but a naive and straightforward implementation of the simplest max flow solution scheme and will no doubt be improved.

23.1 Data Structure

There are three structures associated with the `Network` object.

- `Network` – the main object
- `Arc` – a structure that represents an edge in the network.
- `ArcChunk` – a structure that holds the storage for a number of arcs. Since we do not require the number of arcs to be known in advance when initializing the `Network` object, we allocate chunks of space to hold the arcs as necessary. Each chunks holds space for `nnode` arc structures.

The `Network` object has six fields.

- `int nnode` — the number of nodes in the network, including the source (node 0) and the sink (node `nnode-1`).
- `int narc` — the number of arcs in the network
- `int ntrav` — the number of arc traversals that we made to find a max flow.
- `Arc **inheads` — pointer to a vector of pointers to `Arc`, `inheads[v]` points to the first arc in the in-list for node `v`.
- `Arc **outheads` — pointer to a vector of pointers to `Arc`, `outheads[v]` points to the first arc in the out-list for node `v`.
- `ArcChunk *chunk` — pointer to the first `ArcChunk` structure.
- `int msglvl` — message level for debugging and diagnostics. Setting `msglvl = 0` means no output.

- `FILE *msgFile` — message file for debugging and diagnostics.

A correctly initialized and nontrivial `Network` object will have positive `nnode` and `narc` values, and non-`NULL` `inheads`, `outheads` and `chunk` fields.

The `Arc` object has six fields.

- `int first` — the first node in the arc.
- `int second` — the second node in the arc.
- `int capacity` — the capacity of the arc.
- `int flow` — the flow along the arc.
- `Arc *nextOut` — a pointer to the next `Arc` structure in the out-list for node `first`.
- `Arc *nextIn` — a pointer to the next `Arc` structure in the in-list for node `second`.

The `ArcChunk` object has four fields.

- `int size` — the total number of `Arc` structures in this chunk.
- `int inuse` — the number of active `Arc` structures in this chunk.
- `Arc *base` — pointer to the first `Arc` structure in this chunk.
- `ArcChunk *next` — pointer to the next `ArcChunk` structure in the list of chunks.

23.2 Prototypes and descriptions of `Network` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Network` object.

23.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Network * Network_new ( void ) ;`

   This method simply allocates storage for the `Network` structure and then sets the default fields by a call to `Network_setDefaultFields()`.

2. `void Network_setDefaultFields ( Network *network ) ;`

   This method sets the structure's fields to default values.

   Error checking: If `network` is `NULL`, an error message is printed and the program exits.

3. `void Network_clearData ( Network *network ) ;`

   This method releases any storage held by the object, e.g., it free's the `inheads` and `outheads` vectors and one by one it releases the storage held in the list of `ArcChunk` structures. It then sets the structure's default fields with a call to `Network_setDefaultFields()`.

   Error checking: If `network` is `NULL`, an error message is printed and the program exits.

4. `void Network_free ( Network *network ) ;`

   Thismethod releases any storage by a call to `Network_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `network` is `NULL`, an error message is printed and the program exits.

23.2.2 Initializer methods

There are three initializer methods.

1. `void Network_init ( Network *network, int nnode, int narc ) ;`

   This method initializes an `Network` object given the number of nodes and number of arcs. (The latter may be zero since we allow the storage for the arcs to grow dynamically.)

   Error checking: If `network` is `NULL`, or if `nnode` ≤ 2, or if `narc` < 0, an error message is printed and the program exits.

2. `void Network_setMessageInfo ( Network *network, int msglvl, FILE *msgFile) ;`

   This method sets the message level and message file pointer for the object.

   Error checking: If `network` is `NULL`, an error message is printed and the program exits.

3. ```
   void Network_addArc ( Network *network, int firstNode, secondNode,
                         int capacity, int flow ) ;
   ```

   This method adds an arc from `firstNode` to `secondNode` with flow `flow` and capacity `capacity`. The arc is inserted in the out-list for `firstNode` and the in-list for `secondNode`.

   Error checking: If `network` is `NULL`, or if `nnode` ≤ 0, or if `firstNode` ≤ 0, or if `nnode` $\leq$ `firstNode`, or if `secondNode` ≤ 0, or if `nnode` $\leq$ `secondNode`, or if `capacity` ≤ 0, an error message is printed and the program exits.

23.2.3 Utility methods

1. `void Network_findMaxFlow ( Network *network ) ;`

   This method finds a maximum flow over the network by repeatedly calling the method to find an augmenting path and then the method to augment the path. It uses an `Ideq` object to maintain a priority dequeue.

   Error checking: If `network` is `NULL`, or if `nnode` ≤ 0, an error message is printed and the program exits.

2. ```
   int Network_findAugmentingPath ( Network *network, int node, int delta,
       int tag, Ideq *deq, int tags[], int deltas[], int pres[] ) ;
   ```

   This methods tries to find an augmenting path. If successful, the return value is the additional flow that can flow down the path. The start node is `node`, adjacent to the source and for which the edge `(source, node)` is not saturated. The input parameter `delta` is the difference between the capacity and the flow along this initial edge. The `Ideq` object holds the priority dequeue to store the nodes ids that are visited during the search. The `tags[]` vector is used to tag nodes that have been visited — if `tags[v] = tag`, then `v` has been visited. The `deltas[v]` value maintains the largest admissible flow in the path from the source to `v`. The `pred[]` vector holds the tree links for the nodes.

   Error checking: If `network`, `deq`, `tags`, `deltas` or `pred` is `NULL`, or if `nnode` ≤ 0, or if `node` ≤ 0, or if `nnode` $- 1 \leq$ `node`, an error message is printed and the program exits.

3. `void Network_augmentPath ( Network *network, int delta, int pred[] ) ;`

   This method augments the flow along the path defined by the `pred[]` vector by `delta` units.

   Error checking: If `network` or `pred` is `NULL`, or if `nnode` ≤ 0, or if `delta` ≤ 0, an error message is printed and the program exits.

4. `void Network_findMincutFromSource ( Network *network, Ideq deq, int mark[]) ;`

   This method finds the min-cut closest to the source by traversing a tree of flow-alternating paths from the source. On return, `mark[v] = 1` if the node `v` is in the component that contains the source. If the node `v` is in the component that contains the sink, then `mark[v] = 2`.

   Error checking: If `network`, `deq` or `mark` is `NULL`, or if `nnode` ≤ 0, an error message is printed and the program exits.

5. `void Network_findMincutFromSink ( Network *network, Ideq deq, int mark[]) ;`

   This method finds the min-cut closest to the sink by traversing a tree of flow-alternating paths into the sink. On return, `mark[v] = 1` if the node `v` is in the component that contains the source. If the node `v` is in the component that contains the sink, then `mark[v] = 2`.

   Error checking: If `network`, `deq` or `mark` is `NULL`, or if `nnode` ≤ 0, an error message is printed and the program exits.

23.2.4 IO methods

There are two IO routines for debugging purposes.

1. `void Network_writeForHumanEye ( Network *network, FILE *fp ) ;`

   This method writes the network to a file in a human readable format. The method `Network_writeStats()` is called to write out the header and statistics. Then the in-list and out-lists for the nodes in the network are printed.

   Error checking: If `network` or `fp` is `NULL`, an error message is printed and the program exits.

2. `void Network_writeStats ( Network *network, FILE *fp ) ;`

   This method writes a header and statistics to a file.

   Error checking: If `network` or `fp` is `NULL`, an error message is printed and the program exits.

Chapter 24

SolveMap: Forward and Backsolve Map

The SolveMap object is to assign submatrix operations to threads or processors in a forward and backsolve.

A front is *owned* by a single process, and this ownership is defined by an owners[] vector. If process myid owns front J, then $L_{J,J}$, $D_{J,J}$ and $U_{J,J}$ are owned by myid. The off-diagonal submatrices in the upper block and their owners are stored as triples in three vectors. The ii'th submatrix in the upper triangle has row id rowidsUpper[ii], column id colidsUpper[ii], and is owned by thread or process mapUpper[ii]. A similar situation holds for the lower triangle when the factorization is nonsymmetric.

24.1 Data Structure

The SolveMap structure has the following fields.

- int symmetryflag : symmetry flag
  - SPOOLES_SYMMETRIC – symmetric $(U^T + I)D(I + U)$ factorization
  - SPOOLES_HERMITIAN – hermitian $(U^H + I)D(I + U)$ factorization
  - SPOOLES_NONSYMMETRIC – nonsymmetric $(L + I)D(I + U)$ factorization
- int nfront – number of fronts
- int nproc – number of threads or processes
- int *owners – vector mapping fronts to owning threads or processes
- int nblockUpper – number of submatrices in the upper triangle
- int *rowidsUpper – vector of row ids for the upper triangle
- int *colidsUpper – vector of column ids for the upper triangle
- int *mapUpper – map from submatrices to threads or processes
- int nblockLower – number of submatrices in the lower triangle
- int *rowidsLower – vector of row ids for the lower triangle
- int *colidsLower – vector of column ids for the lower triangle
- int *mapLower – map from submatrices to threads or processes processes

24.2 Prototypes and descriptions of SolveMap methods

This section contains brief descriptions including prototypes of all methods that belong to the SolveMap object.

24.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `SolveMap * SolveMap_new ( void ) ;`

   This method simply allocates storage for the SolveMap structure and then sets the default fields by a call to SolveMap_setDefaultFields().

2. `void SolveMap_setDefaultFields ( SolveMap *solvemap ) ;`

   This method sets the default fields of the object — symmetryflag = SPOOLES_SYMMETRIC, nfront, nproc, nblockUpper and nblockLower are set to zero, and owners, rowidsUpper, colidsUpper, mapUpper, rowidsLower, colidsLower and mapLower are set to NULL.

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

3. `void SolveMap_clearData ( SolveMap *solvemap ) ;`

   This method clears any data allocated by this object and then sets the default fields with a call to SolveMap_setDefaultFields().

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

4. `void SolveMap_free ( SolveMap *solvemap ) ;`

   This method releases any storage by a call to SolveMap_clearData() then free's the storage for the structure with a call to free().

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

24.2.2 Instance methods

1. `int SolveMap_symmetryflag ( SolveMap *solvemap ) ;`

   This method returns symmetryflag, the symmetry flag.

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

2. `int SolveMap_nfront ( SolveMap *solvemap ) ;`

   This method returns nfront, the number of fronts.

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

3. `int SolveMap_nproc ( SolveMap *solvemap ) ;`

   This method returns nproc, the number of threads or processes.

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

4. `int SolveMap_nblockUpper ( SolveMap *solvemap ) ;`

   This method returns nblockUpper, the number of off-diagonal submatrices in the upper triangle.

   Error checking: If solvemap is NULL, an error message is printed and the program exits.

5. `int SolveMap_nblockLower ( SolveMap *solvemap ) ;`

   This method returns `nblockLower`, the number of off-diagonal submatrices in the lower triangle.

   Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

6. `int * SolveMap_owners ( SolveMap *solvemap ) ;`

   This method returns `owners`, a pointer to the map from fronts to owning threads or processes.

   Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

7. `int * SolveMap_rowidsUpper ( SolveMap *solvemap ) ;`

   This method returns `rowidsUpper`, a pointer to the vector of row ids of the submatrices in the upper triangle.

   Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

8. `int * SolveMap_colidsUpper ( SolveMap *solvemap ) ;`

   This method returns `colidsUpper`, a pointer to the vector of column ids of the submatrices in the upper triangle.

   Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

9. `int * SolveMap_mapUpper ( SolveMap *solvemap ) ;`

   This method returns `mapUpper`, a pointer to the vector that maps the submatrices in the upper triangle to threads or processes.

   Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

10. `int * SolveMap_rowidsLower ( SolveMap *solvemap ) ;`

    This method returns `rowidsLower`, a pointer to the vector of row ids of the submatrices in the lower triangle.

    Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

11. `int * SolveMap_colidsLower ( SolveMap *solvemap ) ;`

    This method returns `colidsLower`, a pointer to the vector of column ids of the submatrices in the upper triangle.

    Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

12. `int * SolveMap_mapLower ( SolveMap *solvemap ) ;`

    This method returns `mapLower`, a pointer to the vector that maps the submatrices in the upper triangle to threads or processes.

    Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

24.2.3 Initialization method

1. ```
   void SolveMap_init ( SolveMap *solvemap, int symmetryflag, int nfront,
                        int nproc, int nblockUpper, nblockLower ) ;
   ```

   Any previously owned data is cleared via a call to `SolveMap_clearData()`. The five scalars are then set and the vectors are allocated and initialized.

   Error checking: If `solvemap` is `NULL`, or `symmetryflag` is invalid, or `nfront`, `nblockUpper`, `nblockLower` or `nproc` is negative, an error message is printed and the program exits.
   ```

24.2.4 Map creation methods

1. ```
   void SolveMap_randomMap ( SolveMap *solvemap, int symmetryflag,
                             IVL *upperBlockIVL, IVL *lowerBlockIVL, int nproc,
                             IV *ownersIV, int seed, int msglvl, FILE *msgFile) ;
   ```
   This method maps offdiagonal submatrices to threads or processes in a random fashion.

   Error checking: If `solvemap`, `upperBlockIVL` or `ownersIV` is `NULL`, or if `symmetryflag` is invalid, an error message is printed and the program exits.

2. ```
   void SolveMap_ddMap ( SolveMap *solvemap, int symmetryflag,
                         IVL *upperBlockIVL, IVL *lowerBlockIVL, int nproc,
                         IV *ownersIV, int seed, int msglvl, FILE *msgFile) ;
   ```
   This method maps offdiagonal submatrices to threads or processes in a domain decomposition fashion. A *domain* is a subtree of fronts that are owned by the same thread or process. Furthermore, a domain is *maximal*, i.e., the parent of the root domain (if it exists) is owned by a different process. If J belongs to a domain, then for all K, $L_{K,J}$ and $U_{J,K}$ are owned by the thread or process that owns the domain. All other submatrices are mapped to threads or processes in a random fashion.

   Error checking: If `solvemap`, `upperBlockIVL` or `ownersIV` is `NULL`, or if `symmetryflag` is invalid, an error message is printed and the program exits.

24.2.5 Solve setup methods

1. ```
   IP ** SolveMap_forwardSetup ( SolveMap *solvemap, int myid,
                                 int msglvl, FILE *msgFile ) ;
   IP ** SolveMap_backwardSetup ( SolveMap *solvemap, int myid,
                                  int msglvl, FILE *msgFile ) ;
   ```
   These two methods return a vector of pointers to `IP` objects that contain the list of submatrices that thread or process `myid` will use during the forward or backward solves.

   Error checking: If `solvemap` is `NULL`, or if `myid < 0` or `myid >= solvemap->nproc`, an error message is printed and the program exits.

24.2.6 Utility methods

1. ```
   int SolveMap_owners ( SolveMap *solvemap, int rowid, int colid ) ;
   ```
   If `rowid = colid`, this method returns the owner of front `rowid`. Otherwise, this method returns the thread or process of the owner of $L_{\texttt{rowid},\texttt{colid}}$ if `rowid` $\geq$ `colid` or $U_{\texttt{rowid},\texttt{colid}}$ if `rowid` $<$ `colid`.

   Error checking: If `solvemap` is `NULL`, an error message is printed and the program exits.

2. ```
   IVL * SolveMap_upperSolveIVL ( SolveMap *solvemap, int myid,
                                  int msglvl, FILE *msgFile ) ;
   ```
   This method returns an IVL object whose list `K` contains all processes that do not own `K` but who own an $U_{\texttt{J},\texttt{K}}$ for some `J` $<$ `K`.

   Error checking: If `solvemap` is `NULL` then an error message is printed and the program exits.

3. ```
   IVL * SolveMap_lowerSolveIVL ( SolveMap *solvemap, int myid,
                                  int msglvl, FILE *msgFile ) ;
   ```
   This method returns an IVL object whose list `J` contains all processes that do not own `J` but who own an $L_{\texttt{K},\texttt{J}}$ for some `K` $>$ `J`.

   Error checking: If `solvemap` is `NULL` then an error message is printed and the program exits.

4. `IV * SolveMap_upperAggregateIV ( SolveMap *solvemap, int myid int msglvl, FILE *msgFile ) ;`

   This method returns an `IV` object that contains the aggregate count for a backward solve. If `myid` owns front `J`, then entry `J` of the returned `IV` object contains the number of processes (other than `myid`) that own an $U_{\texttt{J},\texttt{K}}$ submatrix, and so is the number of incoming aggregate submatrices process `myid` expects for front `J`.

   Error checking: If `solvemap` is `NULL` or `nlist < 0` then an error message is printed and the program exits.

5. `IV * SolveMap_lowerAggregateIV ( SolveMap *solvemap, int myid int msglvl, FILE *msgFile ) ;`

   This method returns an `IV` object that contains the aggregate count for a forward solve. If `myid` owns front `J`, then entry `J` of the returned `IV` object contains the number of processes (other than `myid`) that own an $L_{\texttt{J},\texttt{I}}$ submatrix, (or $U_{\texttt{I},\texttt{J}}$ submatrix if symmetric or hermitian) and so is the number of incoming aggregate submatrices process `myid` expects for front `J`.

   Error checking: If `solvemap` is `NULL` or `nlist < 0` then an error message is printed and the program exits.

24.2.7 IO methods

There are the usual eight IO routines. The file structure of a `SolveMap` object is simple: `symmetryflag`, `nfront`, `nproc`, `nblockUpper` and `nblockLower`, followed by `owners[*]`, `rowidsUpper[*]`, `colidsUpper[*]` and `mapidsUpper[*]`, and if `symmetryflag = SPOOLES_NONSYMMETRIC`, followed by `rowidsLower[*]`, `colidsLower[*]` and `mapidsLower[*]`.

1. `int SolveMap_readFromFile ( SolveMap *solvemap, char *fn ) ;`

   This method reads an `SolveMap` object from a file. If the the file can be opened successfully, the method calls `SolveMap_readFromFormattedFile()` or `SolveMap_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `solvemap` or `fn` are `NULL`, or if `fn` is not of the form `*.solvemapf` (for a formatted file) or `*.solvemapb` (for a binary file), an error message is printed and the method returns zero.

2. `int SolveMap_readFromFormattedFile ( SolveMap *solvemap, FILE *fp ) ;`

   This method reads an `SolveMap` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `solvemap` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int SolveMap_readFromBinaryFile ( SolveMap *solvemap, FILE *fp ) ;`

   This method reads an `SolveMap` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `solvemap` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int SolveMap_writeToFile ( SolveMap *solvemap, char *fn ) ;`

   This method writes an `SolveMap` object to a file. If the the file can be opened successfully, the method calls `SolveMap_writeFromFormattedFile()` or `SolveMap_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `solvemap` or `fn` are `NULL`, or if `fn` is not of the form `*.solvemapf` (for a formatted file) or `*.solvemapb` (for a binary file), an error message is printed and the method returns zero.

5. `int SolveMap_writeToFormattedFile ( SolveMap *solvemap, FILE *fp ) ;`

   This method writes an `SolveMap` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `solvemap` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int SolveMap_writeToBinaryFile ( SolveMap *solvemap, FILE *fp ) ;`

   This method writes an `SolveMap` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `solvemap` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int SolveMap_writeForHumanEye ( SolveMap *solvemap, FILE *fp ) ;`

   This method writes an `SolveMap` object to a file in an easily readable format. The method `SolveMap_writeStats()` is called to write out the header and statistics. The value `1` is returned.

   Error checking: If `solvemap` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int SolveMap_writeStats ( SolveMap *solvemap, FILE *fp ) ;`

   This method writes some statistics about an `SolveMap` object to a file. The value `1` is returned.

   Error checking: If `solvemap` or `fp` are `NULL`, an error message is printed and zero is returned.

Chapter 25

Tree: A Tree Object

The `Tree` object has very simple functionality, it represents the graph of a *tree* data structure of fixed size. (In reality, it is a "forest" object, for the graph need not be connected.) Trees are used throughout sparse matrix computations. The elimination tree [16] is the most common example, though assembly trees [10], element merge trees [11] and front trees are also common.

The `Tree` object is very simple — there is a size, a root, and parent, first child and sibling vectors. No information is stored for a node except for its tree connections. For an elimination tree, each vertex needs to know the number of ancestors adjacent in the factor graph. For a front tree, each front needs to know the dimensions of the front matrix. This extra information cannot be stored in the `Tree` object. See the `ETree` object in Chapter 19; each `ETree` object contains a `Tree` object. (In a language that supports inheritance, `ETree` could be a subclass of `Tree`.)

25.1 Data Structure

The `Tree` object has a very simple data structure. The value `-1` is used to denote a null pointer for the parent, first child and sibling fields.

- `int n` : size of the tree
- `int root` : root of the tree, in range `[0,n-1]`, in the range `[-1,n-1]`
- `int *par` : pointer to parent vector, size `n`, entries in the range `[-1,n-1]`
- `int *fch` : pointer to first child vector, size `n`, entries in the range `[-1,n-1]`
- `int *sib` : pointer to sibling vector, size `n`, entries in the range `[-1,n-1]`

The user should rarely if ever change these five fields. In particular, throughout the code we assume that the `Tree` object was correctly initialized using one of the three initializer methods. Inside almost every method we check to ensure $n > 0$. If $n > 0$ then we assume that the structure was intialized correctly and that the `par`, `fch` and `sib` fields point to storage that was allocated by the initializer method.

25.2 Prototypes and descriptions of `Tree` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Tree` object.

25.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Tree * Tree_new ( void ) ;`

   This method simply allocates storage for the `Tree` structure and then sets the default fields by a call to `Tree_setDefaultFields()`.

2. `void Tree_setDefaultFields ( Tree *tree ) ;`

   This method sets the structure's fields to default values: `n` is zero, `root` is `-1`, and `par`, `fch` and `sib` are all `NULL`.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

3. `void Tree_clearData ( Tree *tree ) ;`

   This method releases any storage held by the parent, first child and sibling vectors, then sets the structure's default fields with a call to `Tree_setDefaultFields()`.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

4. `void Tree_free ( Tree *tree ) ;`

   This method releases any storage by a call to `Tree_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

25.2.2 Instance methods

1. `int Tree_nnodes ( Tree *tree ) ;`

   This method returns the number of nodes in the tree.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

2. `int Tree_root ( Tree *tree ) ;`

   This method returns the root of the tree.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

3. `int * Tree_par ( Tree *tree ) ;`

   This method returns a pointer to the parent vector.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

4. `int * Tree_fch ( Tree *tree ) ;`

   This method returns a pointer to the first child vector.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

5. `int * Tree_sib ( Tree *tree ) ;`

   This method returns a pointer to the sibling vector.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

25.2.3 Initializer methods

There are three initializers and two helper functions to set the dimensions of the tree, allocate the three vectors, and fill the information.

1. `void Tree_init1 ( Tree *tree, int size ) ;`

   This is the basic initializer method. Any previous data is cleared with a call to `Tree_clearData()`. The size is set and storage allocated for the three tree vectors using `IVinit()`. All entries in the three vectors are set to `-1`.

   Error checking: If `tree` is `NULL` or `size` is negative, an error message is printed and the program exits.

2. `void Tree_init2 ( Tree *tree, int size, int par[] ) ;`

   The simple initializer `Tree_init1()` is called and the entries in `par[]` are copied into the parent vector. The helper method `Tree_setFchSibRoot()` is then called to set the other fields.

   Error checking: If `tree` or `par` is `NULL`, or if `size` is negative, an error message is printed and the program exits.

3. `void Tree_init3 ( Tree *tree, int size, int par[], int fch[], int sib[] ) ;`

   The simple initializer `Tree_init1()` is called and the entries in `par[]`, `fch[]` and `sib[]` are copied into their respective vectors. The helper method `Tree_setRoot()` is then called to set the root field.

   Error checking: If `tree`, `par`, `fch` or `sib` is `NULL`, or if `size` is negative, an error message is printed and the program exits.

4. `int Tree_initFromSubtree ( Tree *subtree, IV *nodeidsIV, Tree *tree ) ;`

   The `subtree` object is initialized from the `tree` object, the nodes that are included are those found in `nodeidsIV`. A parent-child link in the subtree means that the two nodes have a parent-child link in the tree.

   Return codes:

1	normal return	-3	`tree` is `NULL`
-1	`subtree` is `NULL`	-4	`nodeidsIV` is invalid
-2	`nodeidsIV` is `NULL`		

5. `void Tree_setFchSibRoot ( Tree *tree ) ;`

   The root and the entries in the `fch[]` and `sib[]` vectors are set using the entries in the `par[]` vector.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

6. `void Tree_setRoot ( Tree *tree ) ;`

   The vertices that are roots in the tree are linked by their `sib[]` field and the root of the tree is set to the head of the list.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

25.2.4 Utility methods

The utility methods return the number of bytes taken by the object, aid in performing pre-order and post-order traversals, and return statistics about the tree (e.g., the number of roots or leaves in the tree, or the number of children of a node in the tree). This functionality can be easily had by direct manipulation or inquiry of the object, but these methods insulate the user from the internals and allow us to change and improve the internals if necessary.

1. `int Tree_sizeOf ( Tree *tree ) ;`

   This method returns the number of bytes taken by this object.

   Error checking: If `tree` is `NULL`, an error message is printed and the program exits.

2. `int Tree_postOTfirst ( Tree *tree ) ;`

   This method returns the first node in a post-order traversal.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

3. `int Tree_postOTnext ( Tree *tree, int v ) ;`

   This method returns the node that follows `v` in a post-order traversal.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1` or `v` is not in `[0,tree->n-1]`, an error message is printed and the program exits.

4. `int Tree_preOTfirst ( Tree *tree ) ;`

   This method returns the first node in a pre-order traversal.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

5. `int Tree_preOTnext ( Tree *tree, int v ) ;`

   This method returns the node that follows `v` in a pre-order traversal.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, or `v` is not in `[0,tree->n-1]`, an error message is printed and the program exits.

6. `int Tree_nleaves ( Tree *tree ) ;`

   This method returns the number of leaves of the tree.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

7. `int Tree_nroots ( Tree *tree ) ;`

   This method returns the number of roots of the tree (really a forest).

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

8. `int Tree_nchild ( Tree *tree, int v ) ;`

   This method returns the number of children of `v`.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, or `v` is not in `[0,tree->n-1]`, an error message is printed and the program exits.

9. `IV * Tree_nchildIV ( Tree *tree ) ;`

   This method creates an `IV` object that holds the number of children for each of the nodes, i.e., entry `v` of the returned `IV` object contains the number of children of node `v`.

   Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

10. `int Tree_maxNchild ( Tree *tree ) ;`

    This method returns the maximum number of children of any vertex.

    Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

11. `int Tree_height ( Tree *tree ) ;`

    This method returns the height of the tree.

    Error checking: If `tree` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

12. `IV * Tree_maximizeGainIV ( Tree *tree, IV *gainIV, int *ptotalgain, int msglvl, FILE *msgFile ) ;`

    Given a gain value assigned to each node, find a set of nodes, no two in a child-ancestor relationship, that maximizes the total gain. This problem arises in finding the optimal domain/Schur complement partition for a semi-implicit factorization.

    Error checking: If `tree`, `gainIV` or `ptotalgain` is `NULL`, an error message is printed and the program exits.

25.2.5 Metrics methods

Many operations need to know some *metric* defined on the nodes in a tree. Here are three examples: the height of a node (the minimum distance from a descendant leaf), the depth of a node (the distance from its root ancestor), or the weight associated with a subtree rooted at a node. Of course, a weight could be associated with each node, so the height or depth becomes the weight of the nodes on the path.

Metrics can be `int` or `double`. Because of the limitations of C, we need two separate methods for each of the height, depth and subtree functions. Each pair of methods differs only in the type of the vector object argument.

1. `IV * Tree_setSubtreeImetric ( Tree *tree, IV *vmetricIV ) ;`
   `DV * Tree_setSubtreeDmetric ( Tree *tree, DV *vmetricDV ) ;`

   These methods create and return `IV` or `DV` objects that contain subtree metrics using as input an `IV` or `DV` object that contains the metric for each of the nodes. If `tmetric[]` is the vector in the returned `IV` or `DV` object, then

   ```
   tmetric[v] = vmetric[v] + sum_{par[u] = v} tmetric[u].
   ```

   Error checking: If `tree` or `vmetric{I,D}V` is `NULL`, an error message is printed and the program exits.

2. `IV * Tree_setDepthImetric ( Tree *tree, IV * vmetricIV ) ;`
   `DV * Tree_setDepthDmetric ( Tree *tree, DV * vmetricDV ) ;`

   These methods create and return `IV` or `DV` objects that contain depth metrics using as input an `IV` or `DV` object that contains the metric for each of the nodes. If `dmetric[]` is the vector in the returned `IV` or `DV` object, then

   ```
   dmetric[v] = vmetric[v] if par[v] == -1
              = vmetric[v] + dmetric[par[v]] if par[v] != -1
   ```

   Error checking: If `tree` or `vmetric{I,D}V` is `NULL`, an error message is printed and the program exits.

3. `IV * Tree_setHeightImetric ( Tree *tree, IV * vmetricIV ) ;`
   `DV * Tree_setHeightDmetric ( Tree *tree, DV * vmetricDV ) ;`

   These methods create and return `IV` or `DV` objects that contain height metrics using as input an `IV` or `DV` object that contains the metric for each of the nodes. If `hmetric[]` is the vector in the returned `IV` or `DV` object, then

   ```
   hmetric[v] = vmetric[v] if fch[v] == -1
              = vmetric[v] + max_{par[u] = v} hmetric[par[v]]
   ```

   Error checking: If `tree` or `vmetric{I,D}V` is `NULL`, an error message is printed and the program exits.

25.2.6 Compression methods

Frequently a tree will need to be compressed in some manner. Elimination trees usually have long chains of nodes at the higher levels, where each chain of nodes corresponds to a supernode. Liu's generalized row envelope methods partition the vertices by longest chains [17]. In both cases, we can construct a map from each node to a set of nodes to define a smaller, more compact tree. Given such a map, we construct the smaller tree.

A fundamental chain is a set of nodes $v_1, \dots, v_m$ such that (1) v_1 is a leaf or has two or more children, (2) v_{i+1} is the parent of v_i for $1 \leq i < m$, and (3) v_m is either a root or has a sibling. The set of fundamental chains is uniquely defined. In the context of elimination trees, a fundamental chain is very close to a fundamental supernode, and in many cases, fundamental chains can be used to contruct the fronts with little added fill and factor operations.

1. `IV * Tree_fundChainMap ( Tree *tree ) ;`

   This method creates and returns an `IV` object that contains the map a vertex to the fundamental chain to which it belongs, i.e., `map[v]` contains the id of the fundamental chain that contains `v`. If `u` is a descendant of `v`, then `map[u] <= map[v]`. The number of fundamental chains is returned.

   Error checking: If `tree` is `NULL`, or if `n < 1`, an error message is printed and the program exits.

2. `Tree * Tree_compress ( Tree *tree, IV  *mapIV ) ;`

   This method creates and returns a new `Tree` object formed by compressing `tree` using the `mapIV` object. The compressed tree is constructed and returned.

   Error checking: If `tree` or `mapIV` is `NULL`, or if `n < 1`, an error message is printed and the program exits.

25.2.7 Justification methods

Given a tree, how should the children of a node be ordered? This "justification" can have a large impact in the working storage for the front tree in the multifrontal algorithm. Justification also is useful when displaying trees.

1. `void Tree_leftJustify ( Tree *tree ) ;`

   This method justifies the tree, reordering the children of each node as necessary. If `u` and `v` are siblings, and `u` comes before `v` in a post-order traversal, then the size of the subtree rooted at `u` is as large or larger than the size of the subtree rooted at `v`.

   Error checking: If `tree` or `map` is `NULL`, or if `n < 1`, an error message is printed and the program exits.

2. `void Tree_leftJustifyI ( Tree *tree, IV *metricIV ) ;`
   `void Tree_leftJustifyD ( Tree *tree, DV *metricIV ) ;`

   This method justifies the tree, reordering the children of each node as necessary. If `u` and `v` are siblings, and `u` comes before `v` in a post-order traversal, then the weight of the subtree rooted at `u` is as large or larger than the weight of the subtree rooted at `v`.

   Error checking: If `tree` or `metricIV` is `NULL`, or if `n < 1`, or if `n` is not the size of `metricIV`, an error message is printed and the program exits.

25.2.8 Permutation methods

Often we need to extract a permutation from a tree, e.g., a post-order traversal of an elimination tree gives an ordering for a sparse matrix. On other occasions, we need to permute a tree, i.e. re-label the nodes.

1. `void Tree_fillNewToOldPerm ( Tree *tree, int newToOld[] ) ;`
   `void Tree_fillOldToNewPerm ( Tree *tree, int oldToNew[] ) ;`
   `void Tree_fillBothPerms ( Tree *tree, int newToOld[], int oldToNew[] ) ;`

   If `tree` is `NULL`, `tree->n < 1` or a permutation vector is `NULL`, an error message is printed and the program exits. Otherwise, the permutation vector(s) is (are) filled with the ordering of the nodes in a post-order traversal.

   Error checking: If `tree` or a permutation vector is `NULL`, or if `n < 1`, an error message is printed and the program exits.

2. `Tree * Tree_permute ( Tree *tree, int newToOld[], int oldToNew[] ) ;`

   A new tree is created with the same connectivity as the old but the nodes are relabeled.

   Error checking: If `tree`, `newToOld` or `oldToNew` is `NULL`, or if `tree->n < 1`, an error message is printed and the program exits.

25.2.9 Drawing method

1. ```
   int Tree_getSimpleCoords ( Tree *tree, char heightflag, int coordflag,
                              DV *xDV, DV *yDV) ;
   ```

   This method fills the `xDV` and `yDV` vector objects with coordinates of the nodes in the tree. When `coordflag = 'C'`, we create Cartesian coordinates, where the leaves are at the bottom and the root(s) at the top. When `coordflag = 'P'`, we create polar coordinates, where the leaves are found on the outside and the root(s) in the center. The height of a node is the distance from the bottom for Cartesian coordinates, and the distance from the outermost circle for polar coordinates. When `heightflag = 'H'`, the height of a node is one unit more than that of its highest child. When `heightflag = 'D'`, the height of a node is one unit less than that of its parent.

   `Return codes:`

1	normal return	-3	`coordflag` is invalid
-1	`tree` is `NULL`	-3	`xDV` is `NULL`
-2	`heightflag` is invalid	-4	`yDV` is `NULL`

2. ```
   int Tree_drawToEPS ( Tree *tree, FILE *filename, DV *xDV, DV *yDV,
                        double rscale, DV *radiusDV, int labelflag,
                        double fontscale, IV *labelsIV, double bbox[],
                        double frame[], double bounds[] ) ;
   ```

   This method draws a tree. The coordinates of the nodes are found in the `xDV` and `yDV` vectors.

   The nodes will have circles of constant radius (if `radiusDV` is `NULL`) or each circle can have a different radius found in `radiusDV` when `radiusDV` is not `NULL`. The value `rscale` is used to scale all the radii. (If `radiusDV` is `NULL`, then all radii are equal to one point — there are 72 points to the inch.)

   If `labelflag = 1`, the nodes will have a numeric label. If `labelsIV` is `NULL`, then the label will be the node id. Otherwise, the labels are taken from the `labelsIV` vector. The size of the fonts for the labels is found in `fontscale`, e.g., `fontscale = 10` implies using a 10 point font. `bbox[4]` and `frame[4]` define the bounding box and frame, respectively.

   If `bounds[]` is `NULL`, the tree is sized to fit inside the frame. Note, when the radii of the nodes are non-constant, determining the local coordinates is a non-linear process that may not converge for a large radius with respect to the frame. If this occurs, an error message is printed and the program exits. If `bounds[]` is not `NULL`, then the nodes are mapped to local coordinates within the frame. This is useful when we have two or more trees that need a common reference frame. (See the `testFS` driver program in the `ETree/drivers` directory.)

See the `drawTree` driver program in the next section.

`Return codes:`

1	normal return	-5	`rscale` is negative
-1	`tree` is NULL	-6	`fontscale` is negative
-2	`filename` is NULL	-7	`bbox` is NULL
-3	xDV is NULL	-8	`frame` is NULL
-4	yDV is NULL		

25.2.10 IO methods

There are the usual eight IO routines. The file structure of a tree object is simple: `size`, `root`, `par[size]`, `fch[size]` and `sib[size]`.

1. `int Tree_readFromFile ( Tree *tree, char *fn ) ;`

   This method reads in a `Perm` object from a file. It tries to open the file and if it is successful, it then calls `Tree_readFromFormattedFile()` or `Tree_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `tree` or `fn` are `NULL`, or if `fn` is not of the form `*.treef` (for a formatted file) or `*.treeb` (for a binary file), an error message is printed and the method returns zero.

2. `int Tree_readFromFormattedFile ( Tree *tree, FILE *fp ) ;`

   This method reads in a `Perm` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `tree` or `fp` are `NULL`, an error message is printed and zero is returned.

3. `int Tree_readFromBinaryFile ( Tree *tree, FILE *fp ) ;`

   This method reads in a `Perm` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `tree` or `fp` are `NULL`, an error message is printed and zero is returned.

4. `int Tree_writeToFile ( Tree *tree, char *fn ) ;`

   This method writes a `Perm` object to a file. It tries to open the file and if it is successful, it then calls `Tree_writeFromFormattedFile()` or `Tree_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `tree` or `fn` are `NULL`, or if `fn` is not of the form `*.treef` (for a formatted file) or `*.treeb` (for a binary file), an error message is printed and the method returns zero.

5. `int Tree_writeToFormattedFile ( Tree *tree, FILE *fp ) ;`

   This method writes a `Perm` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `tree` or `fp` are `NULL`, an error message is printed and zero is returned.

6. `int Tree_writeToBinaryFile ( Tree *tree, FILE *fp ) ;`

   This method writes a `Perm` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `tree` or `fp` are `NULL`, an error message is printed and zero is returned.

7. `int Tree_writeForHumanEye ( Tree *tree, FILE *fp ) ;`

   This method writes a `Perm` object to a file in a human readable format. The method `Tree_writeStats()` is called to write out the header and statistics. Then the parent, first child and sibling values are printed out in three columns. The value `1` is returned.

   Error checking: If `tree` or `fp` are `NULL`, an error message is printed and zero is returned.

8. `int Tree_writeStats ( Tree *tree, FILE *fp ) ;`

   This method writes the header and statistics to a file. The value `1` is returned.

   Error checking: If `tree` or `fp` are `NULL`, an error message is printed and zero is returned.

25.3 Driver programs for the `Tree` object

1. `drawTree msglvl msgFile inTreeFile inTagsFile outEPSfile`
   `heightflag coordflag radius bbox[4] frame[4] tagflag fontsize`

   This driver program reads in a `Tree` file and optionally a tags `IV` file and creates an EPS file with a simple picture of a tree.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the output file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inTreeFile` parameter is the input file for the `Tree` object. It must be of the form `*.treef` or `*.treeb`. The `Tree` object is read from the file via the `Tree_readFromFile()` method.
   - The `inTagsFile` parameter is the input file for the `IV` vector object than holds the tags for the nodes. It must be of the form `*.ivf` or `*.ivb` or `none`. The `IV` object is read from the file via the `IV_readFromFile()` method.
   - The `outEPSfile` parameter is name of the encapsulated Postscript file to be written.
   - The `heightflag` parameter is `'D'` to use a depth metric, (i.e., parent and child are in adjacent levels), and `'H'` to use a height metric (i.e., a leaf is on the outermost level).
   - The `coordflag` parameter is `'C'` to put the tree in a Cartesian coordinate system and `'P'` for a polar coordinate system.
   - The `radius` parameter is the radius of each node in the tree.
   - The `bbox` parameter a sequence of four numbers that form the bounding box: lower left x value, lower left y value, width and height.
   - The `frame` parameter a sequence of four numbers that form the frame of the plot within the bounding box: lower left x value, lower left y value, width and height.
   - When `tagflag = 1`, tags are drawn on the nodes. If `tagsFile` is `NULL`, then node ids will be drawn on the nodes. Otherwise, node ids will be taken from the `tagsIV` object.
   - The `fontsize` parameter is the size of the font to be used to draw the node labels.

   Use the `doDraw` script file as an example. Four plots of a tree for the `R2D100` matrix ordered by nested dissection are found below.

Figure 25.1: R2D100: domain/separator tree. On the left `heightflag = 'H'` and `coordflag = 'C'`, on the right `heightflag = 'D'` and `coordflag = 'C'`.

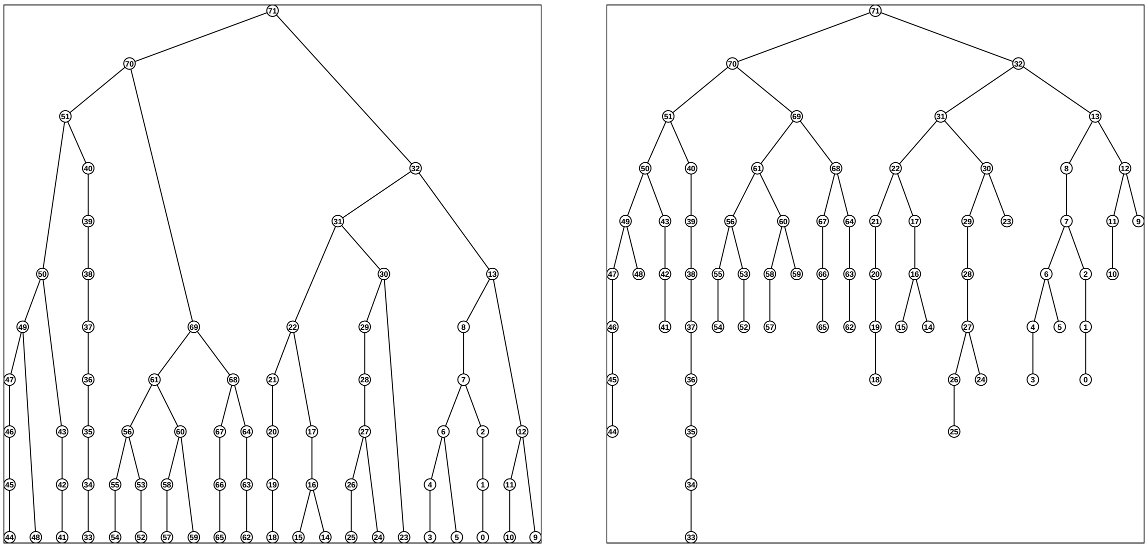
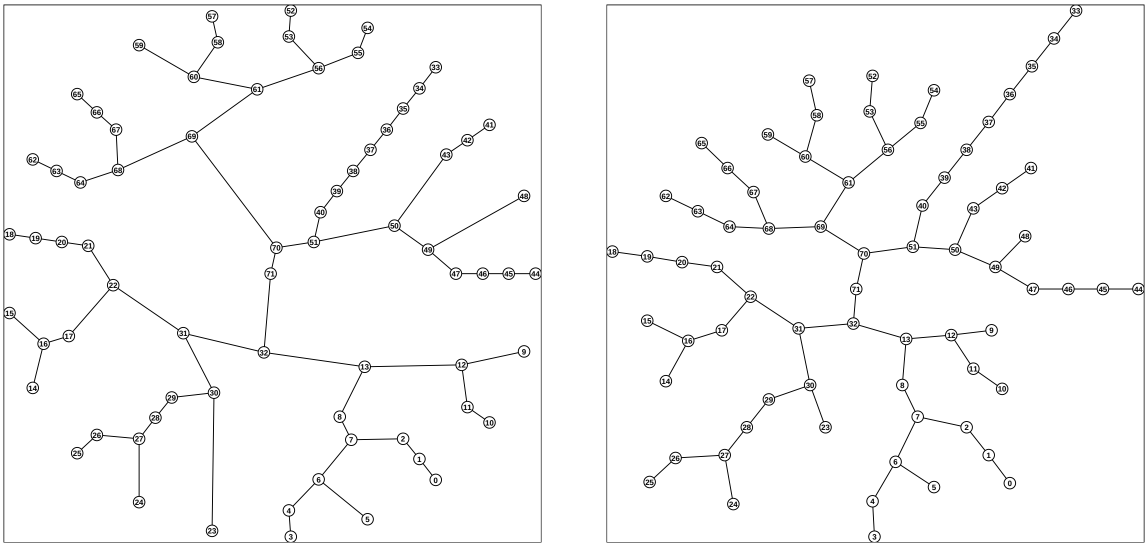


Figure 25.2: R2D100: domain/separator tree. On the left `heightflag = 'H'` and `coordflag = 'P'`, on the right `heightflag = 'D'` and `coordflag = 'P'`.



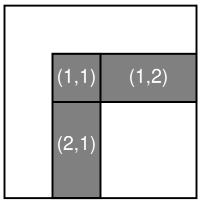
Part IV

Numeric Objects and Methods

Chapter 26

Chv: Block chevron

The `Chv` object is used to store and operate on a *front* during a sparse factorization. The `Chv` object can contain either double precision real or complex data. A front is a portion of a matrix, shaded grey in the diagram below.



We use the word "chevron" to describe the front, because if you rotate the figure 45° clockwise the shaded region resembles the chevron insignia of enlisted personnel in the armed forces. Similar matrices are also known as "arrowhead" matrices, but we have found the "chevron" has a simpler abbreviation. We use the adjective "block" to emphasize that the chevron object may have multiple entries of the diagonal of the matrix. A "single" chevron (which is one way we assemble entries from a matrix into this data structure) contains a single entry from the diagonal of the matrix.

The shaded region in the diagram above will normally be sparse, i.e., many of the entries might be zero. There are three logical blocks to the `Chv` object: a nonempty square (1,1) block in the upper left corner, and (possibly empty) (1,2) and (2,1) blocks in the upper right and lower left corners. To conserve space and minimize work on logically zero elements, we store only rows of the lower part and columns of the upper part that have (or may have) nonzero elements. (Note, a particular row or column may have zero elements, but normally there will be nonzeros in each row and column that we store.)

`Chv` objects come in three types — symmetric, Hermitian and nonsymmetric. When an object is symmetric or Hermitian, we only store the upper triangle. There is one limitation, perhaps unnecessary, that we put on the `Chv` object — the number of rows in the (2,1) block and number of columns in the (1,2) block are equal. The `Chv` object is used within the context of a factorization of a sparse matrix that is assumed to have symmetric structure. If we ever extend the code to handle a true nonsymmetric structure factorization (e.g., UMFPACK and SUPERLU), then we can modify the `Chv` object to handle unequal rows and columns.

During a factorization, a front has to take part in four distinct operations.

1. Assemble entries from the original matrix (or matrix pencil). (See the `Chv_addChevron()` method.)

2. Accumulate updates from descendant fronts. (See the `Chv_update{S,H,N}()` methods.)

3. Assemble any postponed data from its children fronts. (See the `Chv_assemblePostponedData()` method.)

4. Compute the factorization of the completely assembled front. (See the `Chv_factor()` method.)

The implementor of a front object has a great deal of freedom to design the underlying data structures. We have chosen to store the entries in each single chevron in contiguous memory — the first entry of a chevron is in the last row of the front, the last entry of a chevron is in the last column of the front. The figure below shows the storage locations for the entries — on the left is a nonsymmetric chevron, on the right is a symmetric or hermitian chevron.

8	9	10	11	12	13	14	15	16
7	24	25	26	27	28	29	30	31
6	23	38	39	40	41	42	43	44
5	22	37						
4	21	36						
3	20	35						
2	19	34						
1	18	33						
0	17	32						

0	1	2	3	4	5	6	7	8
	9	10	11	12	13	14	15	16
		17	18	19	20	21	22	23

Any storage format has advantages and disadvantages.

con Moving along the diagonal is a nonconstant stride through memory. The same holds for moving along a row in the lower part and along a column in the upper part. While the strides are nonconstant, they are easily determined, particularly when starting in the first chevron. This affects the search methods that look for a pivot in the (1,1) block, the method that evaluates a pivot, the swap methods that swap rows and columns, and the methods that extract the entries from the chevron to be stored in the factor matrix.

pro Moving along a row in the upper part and along a column in the lower part uses a unit stride. This is useful when performing an update to the remaining part of the front after a pivot element has been selected.

pro The assembly of data, be it from the original matrix stored by chevrons, aggregate update fronts from other processes in a parallel factorization, or postponed data when pivoting for stability is used can be done in a straightforward manner.

The chevron object exists within the context of a larger global matrix, and so needs indices to define its rows and columns. For a symmetric or Hermitian matrix, we only store the column indices. For a nonsymmetric matrix, we store the both the row and column indices. This second case may seem unnecessary, since we assume that the larger global matrix has symmetric structure. However, during a factorization with pivoting enabled, a pivot element may be chosen from anywhere in the (1,1) block, so the row indices and column indices may no longer be identical.

A `Chv` object is inherently a serial, single threaded object, meaning it is designed so that only one thread or process "owns" or operates on a particular `Chv` object. A `Chv` object is an "atom" of communication. It stores postponed rows and columns to be assembled in a parent front. It might have to be written to and read from a file in an out-of-core implementation. In a distributed environment, it is communicated between processes. For these reasons, we designed the object so that its data (the scalars that describe its dimensions, id and type, the row and column indices, and its entries) are found in contiguous storage managed by a `DV` object. A file operation can be done with a single read or write, a message can be sent without packing and unpacking data, or defining a new datatype. Managing working storage for a number of `Chv` objects is now simpler.

When the `Chv` object contains double precision *complex* data, it stores and operates on them as double precision entries. We follow the FORTRAN convention that the real and imaginary part of a complex number are stored consecutively, the real part first followed by the imaginary number. In the above complex nonsymmetric matrix, the third diagonal entry is found at location 38 *in terms of the complex numbers*, but its real and imaginary parts are found in locations 2*38 = 76 and 2*38+1 = 77 of the double precision vector that stores the entries. Computations are done in a mix of subroutine calls (see `Utilites/ZV.h`) and by expanding the complex arithmetic into real arithmetic.

The `Chv` object "knows" about the `IV`, `DV` and `ZV` vector objects (for `int`, `double` and `double complex` data types), the `A2` object for dense 2-D arrays, and the `SubMtx` object for dense or sparse 2-D submatrices. These `IV`, `DV`, `ZV`, `A2` and `SubMtx` objects are subordinate to the `Chv` object.

26.1 Data Structure

The `Chv` structure has the following fields.

- `int id` : object's id, default value is `-1`.
- `int nD` : number of rows and columns in the (1,1) block
- `int nL` : number of rows in the (2,1) block
- `int nU` : number of columns in the (1,2) block
- `int type` : type of entries
  - `SPOOLES_REAL` $\Longrightarrow$ real entries
  - `SPOOLES_COMPLEX` $\Longrightarrow$ complex entries
- `int symflag` : symmetry flag
  - `SPOOLES_SYMMETRIC` $\Longrightarrow$ symmetric entries
  - `SPOOLES_HERMITIAN` $\Longrightarrow$ Hermitian entries
  - `SPOOLES_NONSYMMETRIC` $\Longrightarrow$ nonsymmetric entries
- `int *rowind` : pointer to the base address of the `int` vector that contains row indices.
- `int *colind` : pointer to the base address of the `int` vector that contains column indices.
- `double *entries` : pointer to the base address of the `double` vector that contains the entries.
- `DV wrkDV` : object that manages the owned working storage.
- `Chv *next` : link to a next object in a singly linked list.

One can query the type and symmetry of the object using these simple macros.

- `CHV_IS_REAL(chv)` is `1` if `chv` has real entries and `0` otherwise.
- `CHV_IS_COMPLEX(chv)` is `1` if `chv` has complex entries and `0` otherwise.
- `CHV_IS_SYMMETRIC(chv)` is `1` if `chv` is symmetric and `0` otherwise.
- `CHV_IS_HERMITIAN(chv)` is `1` if `chv` is Hermitian and `0` otherwise.
- `CHV_IS_NONSYMMETRIC(chv)` is `1` if `chv` is nonsymmetric and `0` otherwise.

26.2 Prototypes and descriptions of `Chv` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Chv` object.

26.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Chv * Chv_new ( void ) ;`

   This method simply allocates storage for the `Chv` structure and then sets the default fields by a call to `Chv_setDefaultFields()`.

2. `void Chv_setDefaultFields ( Chv *chv ) ;`

   The structure's fields are set to default values: `id` = `-1`, `nD` = `nL` = `nU` = 0, `type` = `SPOOLES_REAL`, `symflag` = `SPOOLES_SYMMETRIC`, and `rowind` = `colind` = `entries` = `next` = `NULL` . The `wrkDV` object has its default fields set via a call to `DV_setDefaultFields()`.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

3. `void Chv_clearData ( Chv *chv ) ;`

   This method clears the object and free's any owned data by invoking the `_clearData()` methods for its internal `DV` object. There is a concluding call to `Chv_setDefaultFields()`.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

4. `void Chv_free ( Chv *chv ) ;`

   This method releases any storage by a call to `Chv_clearData()` and then free the space for `chv`.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

26.2.2 Instance methods

1. `int Chv_id ( Chv *chv ) ;`

   This method returns the *id* of the object.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

2. `int Chv_type ( Chv *chv ) ;`

   This method returns the *type* of the object.

   - `SPOOLES_REAL` $\Longrightarrow$ real entries
   - `SPOOLES_COMPLEX` $\Longrightarrow$ complex entries

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

3. `int Chv_symmetryFlag ( Chv *chv ) ;`

   This method returns the *symmetry flag* of the object.

   - `SPOOLES_SYMMETRIC` $\Longrightarrow$ symmetric entries, i.e., $a_{i,j} = a_{j,i}$.
   - `SPOOLES_HERMITIAN` $\Longrightarrow$ hermitian entries, i.e., $a_{i,j} = \overline{a_{j,i}}$.
   - `SPOOLES_NONSYMMETRIC` $\Longrightarrow$ nonsymmetric entries.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

4. `void Chv_dimensions ( Chv *chv, int *pnD, int *pnL, *pnU ) ;`

   This method fills `*pnD`, `*pnL` and `*pnU` with `nD`, `nL` and `nU`.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

5. `void Chv_rowIndices ( Chv *chv, int *pnrow, **prowind ) ;`

   This method fills `*pnrow` with the number of rows (`nD + nL`) and `*prowind` with a pointer to the row indices.

   Error checking: If `chv`, `pnrow` or `prowind` is `NULL`, an error message is printed and zero is returned.

6. `void Chv_columnIndices ( Chv *chv, int *pncol, **pcolind ) ;`

   This method fills `*pncol` with the number of columns (`nD + nU`) and `*pcolind` with a pointer to the column indices.

   Error checking: If `chv`, `pncol` or `pcolind` is `NULL`, an error message is printed and zero is returned.

7. `int  Chv_nent ( Chv *chv ) ;`

   This method returns number of matrix entries that the object contains. Note, for a complex chevron, this is the number of *double precision complex* entries, equal to one half the number of double precision entries that are stored.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

8. `double * Chv_entries ( Chv *chv ) ;`

   This method returns the *entries* field of the object, a pointer to the base location of the double precision array that stores the complex data.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

9. `double * Chv_diagLocation ( Chv *chv, int ichv ) ;`

   This method returns a pointer to the address of the entry in the `ichv`'th diagonal location. For a real chevron, to find the entry `k` places to the right of the diagonal entry, add `k` to the address. To find an entry `k` places below the diagonal entry, subtract `k` from the address. For a complex chevron, to find the entry `k` places to the right of the diagonal entry, add `2*k` to the address. To find an entry `k` places below the diagonal entry, subtract `2*k` from the address.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

10. `void * Chv_workspace ( Chv *chv ) ;`

    This method returns a pointer to the base address of the workspace.

    Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

11. `void  Chv_realEntry ( Chv *chv, int irow, int jcol, double *pValue ) ;`

    This method fills `*pValue` with the entry in row `irow` and column `jcol`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq \texttt{irow} < \texttt{nD} + \texttt{nL}$ and $0 \leq \texttt{jcol} < \texttt{nD} + \texttt{nU}$.

    Error checking: If `chv` or `pValue` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

12. `Chv_locationOfRealEntry ( Chv *chv, int irow, int jcol, double **ppValue ) ;`

    This method fills `*ppValue` with a pointer to the entry in row `irow` and column `jcol`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq \texttt{irow} < \texttt{nD} + \texttt{nL}$ and $0 \leq \texttt{jcol} < \texttt{nD} + \texttt{nU}$.

    Error checking: If `chv` or `ppValue` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

13. `void  Chv_setRealEntry ( Chv *chv, int irow, int jcol, double value ) ;`

    This method sets the entry in row `irow` and column `jcol` to be `value`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq \texttt{irow} < \texttt{nD} + \texttt{nL}$ and $0 \leq \texttt{jcol} < \texttt{nD} + \texttt{nU}$.

    Error checking: If `chv` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

14. `void Chv_complexEntry ( Chv *chv, int irow, int jcol, double *pReal, double *pImag ) ;`

    This method fills `*pReal` with the real part and `*pImag` with the imaginary part of the the entry in row `irow` and column `jcol`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \le$ `irow` $<$ `nD` $+$ `nL` and $0 \le$ `jcol` $<$ `nD` $+$ `nU`.

    Error checking: If `chv`, `pReal` or `pImag` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

15. `Chv_locationOfComplexEntry ( Chv *chv, int irow, int jcol, double **ppReal, double **ppImag ) ;`

    This method fills `*ppReal` with a pointer to the real part and `*ppImag` with a pointer to the imaginary part of the entry in row `irow` and column `jcol`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \le$ `irow` $<$ `nD` $+$ `nL` and $0 \le$ `jcol` $<$ `nD` $+$ `nU`.

    Error checking: If `chv`, `ppReal` or `ppImag` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

16. `void Chv_setComplexEntry ( Chv *chv, int irow, int jcol, double real, double imag ) ;`

    This method sets the real and imaginary parts and the entry in row `irow` and column `jcol` to be `real` and `imag`, respectively. Note, `irow` and `jcol` are *local* indices, i.e., $0 \le$ `irow` $<$ `nD` $+$ `nL` and $0 \le$ `jcol` $<$ `nD` $+$ `nU`.

    Error checking: If `chv` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

26.2.3 Initialization methods

There are three initializer methods.

1. `void Chv_init( Chv *chv, int id, int nD, int nL, int nU, int type, int symflag ) ;`

   This is the initializer method used when the `Chv` object is to use its owned workspace to store indices and entries. The number of indices and entries is computed, the work space is set up via calls to `Chv_nbytesNeeded()` and `Chv_setNbytesInWorkspace()`, and the scalars, pointers and buffer are set up via a call to `Chv_setFields()`.

   Error checking: If `chv` is `NULL`, or if `nD` ≤ 0, or if `nL` or `nU` < 0, or if `type` or if `symflag` is not valid, an error message is printed and zero is returned.

2. `void Chv_initWithPointers ( Chv *chv, int id, int nD, int nL, int nU, int type, int symflag, int *rowind, int *colind, double *entries ) ;`

   This initializer method is used when the `Chv` object does not own the storage for its indices and entries, but points into some other storage.

   Error checking: If `chv` is `NULL`, or if `nD` ≤ 0, or if `nL` or `nU` < 0, or if `type` or if `symflag` is not valid, or if `entries` or `colind` is `NULL`, or if `symflag = SPOOLES_NONSYMMETRIC` and `rowind` is `NULL`, an error message is printed and zero is returned.

3. `void Chv_initFromBuffer ( Chv *chv ) ;`

   This initializer method is used to set the scalar and pointer fields when the object's buffer is already preloaded. This functionality is used in the MPI factorization where a `Chv` object is sent and received, more precisely, the workspace buffer owned by the `Chv` object is sent and received.

   Error checking: If `chv` is `NULL`, an error message is printed and zero is returned.

26.2.4 Search methods

1. `int Chv_maxabsInDiagonal11 ( Chv *chv, int mark[], int tag, double *pmaxval ) ;`

   This method returns the location of the first tagged element with the largest magnitude in the diagonal of the (1,1) block. Element `jj` must have `mark[jj] = tag` to be eligible. Its magnitude is returned in `*pmaxval`. Note, if the chevron is complex, the location is in terms of the complex entries, not in the real entries, i.e., if `k = Chv_maxabsDiagonal11(chv,...)`, then the complex entry is found in `chv->entries[2*kk:2*kk+1]`.

   Error checking: If `chv`, `mark` or `pmaxval` is `NULL`, an error message is printed and the program exits.

2. ```
   int Chv_maxabsInRow11 ( Chv *chv, int irow, int colmark[],
                           int tag, double *pmaxval ) ;
   ```

   This method returns the location of the first element with the largest magnitude in row `irow` of the (1,1) block. Element `jj` must have `colmark[jj] = tag` to be eligible. Its magnitude is returned in `*pmaxval`. Note, if the chevron is complex, the location is in terms of the complex entries, not in the real entries, i.e., if `k = Chv_maxabsRow11(chv,...)`, then the complex entry is found in `chv->entries[2*kk:2*kk+1]`.

   Error checking: If `chv` is `NULL` or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

3. ```
   int Chv_maxabsInColumn11 ( Chv *chv, int jcol, int rowmark[],
                              int tag, double *pmaxval ) ;
   ```

   This method returns the location of the first element with the largest magnitude in column `jcol` of the (1,1) block. Element `jj` must have `rowmark[jj] = tag` to be eligible. Its magnitude is returned in `*pmaxval`. Note, if the chevron is complex, the location is in terms of the complex entries, not in the real entries, i.e., if `k = Chv_maxabsColumn11(chv,...)`, then the complex entry is found in `chv->entries[2*kk:2*kk+1]`.

   Error checking: If `chv` is `NULL` or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

4. ```
   int Chv_maxabsInRow ( Chv *chv, int irow, int colmark[],
                         int tag, double *pmaxval ) ;
   ```

   This method returns the location of the first element with the largest magnitude in row `irow`. Element `jj` must have `colmark[jj] = tag` to be eligible. Its magnitude is returned in `*pmaxval`. Note, if the chevron is complex, the location is in terms of the complex entries, not in the real entries, i.e., if `k = Chv_maxabsRow(chv,...)`, then the complex entry is found in `chv->entries[2*kk:2*kk+1]`.

   Error checking: If `chv` is `NULL` or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

5. ```
   int Chv_maxabsInColumn ( Chv *chv, int jcol, int rowmark[],
                            int tag, double *pmaxval ) ;
   ```

   This method returns the location of the first element with the largest magnitude in column `jcol`. Element `jj` must have `rowmark[jj] = tag` to be eligible. Its magnitude is returned in `*pmaxval`. Note, if the chevron is complex, the location is in terms of the complex entries, not in the real entries, i.e., if `k = Chv_maxabsColumn11(chv,...)`, then the complex entry is found in `chv->entries[2*kk:2*kk+1]`.

   Error checking: If `chv` is `NULL` or `irow` is not in `[0,n1-1]`, an error message is printed and the program exits.

6. ```
   double Chv_quasimax ( Chv *chv, int rowmark[], int colmark[]
                         int tag, int *pirow, int *pjcol ) ;
   ```
   ```

This method searches for a *quasimax* entry in the $(1,1)$ block, an entry $a_{i,j}$ that has largest magnitude of the tagged entries in row i and column j. An entry $a_{i,j}$ is *tagged* when `rowmark[i] = tag` and `colmark[j] = tag`. On return, `*pirow` is filled with the row id and `*pjcol` is filled with the column id of the quasimax entry. The return value is the magnitude of the entry.

Error checking: If `chv`, `rowmark`, `colmark`, `pirow` or `pjcol` is `NULL`, an error message is printed and the program exits.

7. ```
   void Chv_fastBunchParlettPivot ( Chv *chv, int mark[], int tag,
                                    int *pirow, int *pjcol ) ;
   ```

   This method is used only for a symmetric or hermitian object and finds a 1×1 or 2×2 pivot that is suitable for elimination. Only pivots from the $(1,1)$ block can be chosen. A diagonal element $a_{r,r}$ with maximum magnitude is first found using the `Chv_maxabsInDiagonal11()` method. We then find the element $a_{r,k}$ in that row that has a maximum magnitude. If $|a_{r,r}| > 0.6404|a_{r,k}|$ then we accept the 1×1 pivot element. Otherwise we look for an offdiagonal element that is largest in its row and column and return it as a 2×2 pivot.

   Error checking: If `chv`, `mark`, `pirow` or `pjcol` is `NULL`, an error message is printed and the method returns.

26.2.5 Pivot methods

1. ```
   int Chv_findPivot ( Chv *chv, DV *workDV, double tau, int ndelay,
                       int *pirow, int *pjcol, int *pntest ) ;
   ```

   This method finds and tests a pivot, where if it were used at the next elimination step, each entry in L and U would have magnitude less than or equal to `tau`. The `workDV` object is used for workspace, it is resized as necessary. The `ndelay` parameter allows one to specify the number of leading rows and columns to ignore, useful when delayed rows and columns have been placed in the leading portion of the chevron. The `pirow`, `pjcol` and `pntest` addresses are filled with the pivot row, pivot column, and number of pivot tests performed to find the pivot. If no pivot was found, `pirow` and `pjcol` are filled with `-1`. The return value is the size of the pivot. If the chevron is symmetric, we can find a 1×1 or 2×2 pivot. If the chevron is nonsymmetric, we only find a 1×1 pivot. A return value of zero means that no pivot was found.

   Error checking: If `chv`, `workDV`, `pirow`, `pjcol` or `pntest` is `NULL`, or if `tau` < 1.0, or if `ndelay` < 0, an error message is printed and the program exits.

26.2.6 Update methods

1. ```
   void Chv_updateS ( Chv *chv, SubMtx *mtxD, SubMtx *mtxU, DV *tempDV ) ;
   void Chv_updateH ( Chv *chv, SubMtx *mtxD, SubMtx *mtxU, DV *tempDV ) ;
   void Chv_updateN ( Chv *chv, SubMtx *mtxL, SubMtx *mtxD, SubMtx *mtxU,
                      DV *tempDV ) ;
   ```

   These methods perform an update to a chevron during the factorization. For a symmetric chevron, we compute

   $$\begin{aligned} T_{J\cap\partial I,J\cap\partial I} &:= T_{J\cap\partial I,J\cap\partial I} - U^T_{I,J\cap\partial I} D_{I,I} U_{I,J\cap\partial I} \\ T_{J\cap\partial I,\partial J\cap\partial I} &:= T_{J\cap\partial I,\partial J\cap\partial I} - U^T_{I,J\cap\partial I} D_{I,I} U_{I,\partial J\cap\partial I} \end{aligned}$$

   where D is diagonal or block diagonal with 1×1 and/or symmetric 2×2 pivots. U is stored by sparse or dense columns. For a Hermitian chevron, we compute

   $$\begin{aligned} T_{J\cap\partial I,J\cap\partial I} &:= T_{J\cap\partial I,J\cap\partial I} - U^H_{I,J\cap\partial I} D_{I,I} U_{I,J\cap\partial I} \\ T_{J\cap\partial I,\partial J\cap\partial I} &:= T_{J\cap\partial I,\partial J\cap\partial I} - U^H_{I,J\cap\partial I} D_{I,I} U_{I,\partial J\cap\partial I} \end{aligned}$$
   ```

where D is diagonal or block diagonal with 1×1 and/or Hermitian 2×2 pivots. U is stored by sparse or dense columns. For a nonsymmetric chevron, we compute

$$\begin{aligned} T_{J\cap\partial I, J\cap\partial I} &:= T_{J\cap\partial I, J\cap\partial I} - L_{J\cap\partial I, I} D_{I,I} U_{I, J\cap\partial I} \\ T_{J\cap\partial I, \partial J\cap\partial I} &:= T_{J\cap\partial I, \partial J\cap\partial I} - L_{J\cap\partial I, I} D_{I,I} U_{I, \partial J\cap\partial I} \\ T_{\partial J\cap\partial I, J\cap\partial I} &:= T_{\partial J\cap\partial I, J\cap\partial I} - L_{\partial J\cap\partial I, I} D_{I,I} U_{I, J\cap\partial I} \end{aligned}$$

where D is diagonal, L is stored by sparse or dense rows, and U is stored by sparse or dense columns. `tempDV` is a temporary working vector whose storage is resized as necessary.

Error checking: If `chvT`, `mtxL`, `mtxD`, `mtxU` or `tempDV` is `NULL`, an error message is printed and the program exits.

26.2.7 Assembly methods

1. `void Chv_addChevron ( Chv *chv, double alpha[], int ichv, int chvsize,`
   `int chvind[], double chvent[] ) ;`

   This method is used to assemble entries from the matrix pencil $A + \sigma B$ into the block chevron object. Typically the entries from A or B will come from a `InpMtx` object, one of whose modes of storage is by single `chevrons`. The value `ichv` is the row and column location of the diagonal entry. The indices found in `chvind[]` are *offsets*. Let `off = chvind[ii]` be the offset for one of the chevron's entries. If `off` ≥ 0, then the entry is found in location `(ichv, ichv+off)` of the matrix. If `off` < 0, then the entry is found in location `(ichv-off, ichv)` of the matrix. The value(s) in `alpha[]` form a scalar used to scale the entire chevron for its assembly. A call to assemble entries in A (from the pencil $A + \sigma B$) would have `alpha[] = (1.0,0.0)`; to assemble entries in B (from the pencil $A + \sigma B$) would have `alpha[]` $= (Real(\sigma), Imag(\sigma))$.

   Error checking: If `chv`, `chvind`, `chvent` or `alpha` is `NULL`, or if `ichv` or `chvsize` are less than zero, an error message is printed and the program exits.

2. `void Chv_assembleChv ( Chv *chvJ, Chv *chvI ) ;`

   This method is used to assemble entries from one `Chv` object into another. The application is during a factorization with pivoting, postponed entries from the children are stored in the `chvI Chv` object and need to be assembled into the final working front, along with all updates from the descendents (which are stored in the `chvJ Chv` object. Note, the row and column indices of `chvI` *must nest* with those of `chvJ`.

   Error checking: If `chvI` or `chvJ` is `NULL`, or if their `symflag` fields are not identical, an error message is printed and the program exits.

3. `int Chv_assemblePostponedData ( Chv *newchv, Chv *oldchv, Chv *firstchild ) ;`

   This method is used to assemble a `Chv` object for a front (`oldchv`) along with any postponed data from the children (objects are held in a list where `firstchild` is the head) into a `Chv` object `newchv`. The return value is the number of delayed rows and columns from the children fronts which are found in the leading rows and columns of the chevron.

   Error checking: If `newchv`, `oldchv` or `firstchild` is `NULL`, an error message is printed and the program exits.

26.2.8 Factorization methods

1. `int Chv_factorWithPivoting ( Chv *chv, int ndelay, int pivotflag,`
   `IV *pivotsizesIV, DV *workDV, double tau, int *pntest ) ;`

This method factors a front using pivoting for numerical stability. The number of rows and columns that have been delayed (assembled from the children) is given by `ndelay`, this allows the method that finds the pivots to skip over these rows and columns since no pivot can be found there. When pivoting is enabled (`pivotflag` is `SPOOLES_PIVOTING`), the `workDV` object used during the search process for pivots must be non-`NULL`, `tau` is the upper bound on factor entries, and `pivotsizesIV` must be non-`NULL` when the front is symmetric or Hermitian. The address `pntest` is incremented with the number of pivot tests by the `Chv_findPivot()` method. The return value is the number of eliminated rows and columns.

Error checking: If `chv` is `NULL`, or if `pivotflag` is not valid, or if `ndelay` is negative, or if `pivotflag == SPOOLES_PIVOTING` and `workDV` is `NULL` or `tau` is less than `1.0`, or if the chevron is symmetric or Hermitian, `pivotflag == SPOOLES_PIVOTING` and `pivotsizesIV` is `NULL`, an error message is printed and the program exits.

2. `int Chv_factorWithNoPivoting ( Chv *chv, PatchAndGoInfo *info ) ;`

   This method factors a front without using pivoting for numerical stability. It does support "patch-and-go" functionality, where if a small or zero entry is found in the diagonal element that is to be eliminated, some action can be taken. The return value is the number of eliminated rows and columns.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

3. `int Chv_r1upd ( Chv *chv ) ;`

   This method is used during the factorization of a front, performing a rank-one update of the chevron. The return value is `1` if the pivot is nonzero, `0` otherwise.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

4. `int Chv_r2upd ( Chv *chv ) ;`

   This method is used during the factorization of a front, performing a rank-two update of the chevron. The return value is `1` if the pivot is nonsingular, `0` otherwise.

   Error checking: If `chv` is `NULL`, or if the chevron is nonsymmetric, an error message is printed and the program exits.

5. ```
   void Chv_maxabsInChevron ( Chv *chv, int ichv,
                              double *pdiagmaxabs, *prowmaxabs, *pcolmaxabs ) ;
   ```

   This method is used during the factorization of a front with a "patch-and-go" strategy. On return, `*pdiagmaxabs` contains the magnitude of the diagonal entry for the chevron, `*prowmaxabs` contains the maximum magnitude of the entries in the row of the chevron, and `*pcolmaxabs` contains the maximum magnitude of the entries in the column of the chevron.

   Error checking: If `chv`, `pdiagmaxabs`, `prowmaxabs` or `pcolmaxabs` is `NULL`, or if `ichv` is out of range, an error message is printed and the program exits.

6. `void Chv_zeroOffdiagonalOfChevron ( Chv *chv, int ichv ) ;`

   This method is used during the factorization of a front with a "patch-and-go" strategy. On return, the offdiagonal entries of chevron `ichv` have been set to zero.

   Error checking: If `chv` is `NULL`, or if `ichv` is out of range, an error message is printed and the program exits.

26.2.9 Copy methods

1. ```
   int Chv_countEntries ( Chv *chv, int npivot, int pivotsizes[],
                          int countflag ) ;
   ```

This method counts the number of entries in the chevron that are larger in magnitude than `droptol`. `countflag` has the following meaning.

- `CHV_STRICT_LOWER` $\Longrightarrow$ count strict lower entries
- `CHV_DIAGONAL` $\Longrightarrow$ count diagonal entries
- `CHV_STRICT_UPPER` $\Longrightarrow$ count strict upper entries
- `CHV_STRICT_LOWER_11` $\Longrightarrow$ count strict lower entries in the (1,1) block
- `CHV_LOWER_21` $\Longrightarrow$ count lower entries in the (2,1) block
- `CHV_STRICT_UPPER_11` $\Longrightarrow$ count strict upper entries in the (1,1) block
- `CHV_UPPER_12` $\Longrightarrow$ count upper entries in the (1,2) block

This method is used to compute the necessary storage to store a chevron as a dense front.

Error checking: If `chv` is `NULL` or if `countflag` is not valid, an error message is printed and the program exits.

2. ```
   int Chv_countBigEntries ( Chv *chv, int npivot, int pivotsizes[],
                             int countflag, double droptol ) ;
   ```

   This method counts the number of entries in the chevron that are larger in magnitude than `droptol`. `countflag` has the following meaning.

   - `CHV_STRICT_LOWER` $\Longrightarrow$ count strict lower entries
   - `CHV_STRICT_UPPER` $\Longrightarrow$ count strict upper entries
   - `CHV_STRICT_LOWER_11` $\Longrightarrow$ count strict lower entries in the (1,1) block
   - `CHV_LOWER_21` $\Longrightarrow$ count lower entries in the (2,1) block
   - `CHV_STRICT_UPPER_11` $\Longrightarrow$ count strict upper entries in the (1,1) block
   - `CHV_UPPER_12` $\Longrightarrow$ count upper entries in the (1,2) block

   This method is used to compute the necessary storage to store a chevron as a sparse front.

   Error checking: If `chv` is `NULL` or if `countflag` is not valid, an error message is printed and the program exits.

3. ```
   int Chv_copyEntriesToVector ( Chv *chv, int npivot, int pivotsizes[],
                  int length, double dvec[], int copyflag, int storeflag) ;
   ```

   This method copies some entries the chevron object into a double precision vector. This method is called after a front has been factored and is used to store the factor entries into the storage for the factor matrix. If the front is nonsymmetric, the front contains entries of L, D and U, where D is diagonal. If the front is symmetric or Hermitian, the front contains entries of D and U, and D is diagonal if `pivotsizesIV` is `NULL` or may contain a mixture of 1×1 and 2×2 pivots otherwise. `copyflag` has the following meaning.

   - `CHV_STRICT_LOWER` $\Longrightarrow$ copy strict lower entries
   - `CHV_DIAGONAL` $\Longrightarrow$ copy diagonal entries
   - `CHV_STRICT_UPPER` $\Longrightarrow$ copy strict upper entries
   - `CHV_STRICT_LOWER_11` $\Longrightarrow$ copy strict lower entries in the (1,1) block
   - `CHV_LOWER_21` $\Longrightarrow$ copy lower entries in the (2,1) block
   - `CHV_STRICT_UPPER_11` $\Longrightarrow$ copy strict upper entries in the (1,1) block
   - `CHV_UPPER_12` $\Longrightarrow$ copy upper entries in the (1,2) block

If storeflag is CHV_BY_ROWS, the entries are stored by rows and if storeflag is CHV_BY_COLUMNS, the entries are stored by columns.

Error checking: If chv or dvec is NULL or if length is less than the number of entries to be copied, or if copyflag or storeflag is valid, an error message is printed and the program exits.

4. `int Chv_copyBigEntriesToVector ( Chv *chv, int npivot, int pivotsizes[], int sizes[], int ivec[], double dvec[], int copyflag, int storeflag, double droptol ) ;`

   This method also copies some entries the chevron object into a double precision vector, but only those entries whose magnitude is greater than or equal to droptol are copied. This method is called after a front has been factored and is used to store the factor entries of large magnitude into the storage for the factor matrix. If the front is nonsymmetric, the front contains entries of L, D and U, where D is diagonal. If the front is symmetric, the front contains entries of D and U, and D is diagonal if pivotsizesIV is NULL or may contain a mixture of 1×1 and 2×2 pivots otherwise. copyflag has the following meaning.

   - CHV_STRICT_LOWER $\Longrightarrow$ copy strict lower entries
   - CHV_STRICT_UPPER $\Longrightarrow$ copy strict upper entries
   - CHV_STRICT_LOWER_11 $\Longrightarrow$ copy strict lower entries in the (1,1) block
   - CHV_LOWER_21 $\Longrightarrow$ copy lower entries in the (2,1) block
   - CHV_STRICT_UPPER_11 $\Longrightarrow$ copy strict upper entries in the (1,1) block
   - CHV_UPPER_12 $\Longrightarrow$ copy upper entries in the (1,2) block

   If storeflag is CHV_BY_ROWS, the entries are stored by rows and if storeflag is CHV_BY_COLUMNS, the entries are stored by columns.

   When we store the large entries in the columns of U, sizes[jcol] contains the number of large entries in column jcol. The vectors ivec[] and dvec[] contain the row indices and the entries that are stored. When we store the large entries in the rows of L, sizes[irow] contains the number of large entries in column irow. The vectors ivec[] and dvec[] contain the column indices and the entries that are stored. Presently there is no checking that sizes[], ivec[] and dvec[] are large enough to store the sizes, indices and entries. The large entry count can be obtained using the method Chv_countBigEntries().

   Error checking: If chv or dvec is NULL or if length is less than the number of entries to be copied, or if copyflag or storeflag is not valid, an error message is printed and the program exits.

5. `void Chv_copyTrailingPortion ( Chv *chvI, Chv *chvJ, int offset ) ;`

   This method copies the trailing portion of chvJ into chvI. The first offsets chevrons are not copied, the remainder are copied. This method is used to extract the delayed entries from a front which has been factored.

   Error checking: If chvI or chvJ is NULL, or if offset < 0 or offset is greater than the number of chevrons in chvJ, an error message is printed and the program exits.

26.2.10 Swap methods

1. `void Chv_swapRows ( Chv *chv, int irow, int jrow ) ;`

   This method swaps rows irow and jrow of the chevron. Both rows must be less than the width nD of the chevron. The row ids of the two rows are also swapped. If the chevron is symmetric, then the method Chv_swapRowsAndColumns() is called.

Error checking: If `chv` is `NULL` or if `irow` or `jrow` are less than 0 or greater than or equal to `nD`, an error message is printed and the program exits.

2. `void Chv_swapColumns ( Chv *chv, int icol, int jcol ) ;`

   This method swaps columns `icol` and `jcol` of the chevron. Both columns must be less than the width `nD` of the chevron. The column ids of the two columns are also swapped. If the chevron is symmetric, then the method `Chv_swapRowsAndColumns()` is called.

   Error checking: If `chv` is `NULL` or if `icol` or `jcol` are less than 0 or greater than or equal to `nD`, an error message is printed and the program exits.

3. `void Chv_swapRowsAndColumns ( Chv *chv, int ii, int jj ) ;`

   This method swaps rows and columns `ii` and `jj` of the chevron. Both must be less than the width `nD` of the chevron. The row and/or column ids are also swapped.

   Error checking: If `chv` is `NULL` or if `ii` or `jj` are less than 0 or greater than or equal to `nD`, an error message is printed and the program exits.

26.2.11 Utility methods

1. `int Chv_nbytesNeeded ( int nD, int nL, int nU, int type, int symflag ) ;`

   This method returns the number of bytes necessary to store an object with the given dimensions and type in its workspace.

   Error checking: If `nD`, `nL`, or `nU` is less than zero, or if `type` or `symflag` are not valid, an error message is printed and the program exits.

2. `int Chv_nbytesInWorkspace ( Chv *chv ) ;`

   This method returns the number of bytes in the workspace.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

3. `void Chv_setNbytesInWorkspace ( Chv *chv, int nbytes ) ;`

   This method sets the number of bytes in the workspace. If `nbytes` is less than the number of present bytes in the workspace, the workspace is not shrunk.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

4. ```
   void Chv_setFields ( Chv *chv, int id, int nD, int nL, int nU,
                        int type, int symflag ) ;
   ```

   This method sets the scalar fields and `rowind`, `colind` and `entries` pointers.

   Error checking: If `chv` is `NULL`, or if `nD` ≤ 0, or if `nL` or `nU` are less than zero, or if `type` or `symflag` are not valid, an error message is printed and the program exits.

5. `void Chv_shift ( Chv *chv, int shift ) ;`

   This method is used to shift the base of the entries and adjust dimensions of the `Chv` object. If `shift` is positive, the first `shift` chevrons are removed from the chevron. If `shift` is negative, the `shift` previous chevrons are prepended to the chevron. This is a dangerous method as it changes the state of the object. We use it during the factorization of a front, where one `Chv` object points to the entire chevron in order to swap rows and columns, while another chevron points to the uneliminated rows and columns of the front. It is the latter chevron that is shifted during the factorization.

   Error checking: If `chv` is `NULL` an error message is printed and the program exits.
   ```

6. `void Chv_fill11block ( Chv *chv, A2 *mtx ) ;`

   This method is used to fill a `A2` dense matrix object with the entries in the $(1,1)$ block of the chevron.

   Error checking: If `chv` or `mtx` is `NULL`, an error message is printed and the program exits.

7. `void Chv_fill12block ( Chv *chv, A2 *mtx ) ;`

   This method is used to fill a `A2` dense matrix object with the entries in the $(1,2)$ block of the chevron.

   Error checking: If `chv` or `mtx` is `NULL`, an error message is printed and the program exits.

8. `void Chv_fill21block ( Chv *chv, A2 *mtx ) ;`

   This method is used to fill a `A2` dense matrix object with the entries in the $(2,1)$ block of the chevron.

   Error checking: If `chv` or `mtx` is `NULL`, an error message is printed and the program exits.

9. `double Chv_maxabs ( Chv *chv ) ;`

   This method returns the magnitude of the entry of largest magnitude in the object.

   Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

10. `double Chv_frobNorm ( Chv *chv ) ;`

    This method returns the Frobenius norm of the chevron.

    Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

11. `void Chv_sub ( Chv *chvJ, Chv *chvI ) ;`

    This method subtracts `chvI` from `chvJ`.

    Error checking: If `chvJ` or `chvI` is `NULL`, or if their dimensions are not the same, or if either of their `entries` fields are `NULL`, an error message is printed and the program exits.

12. `void Chv_zero ( Chv *chv ) ;`

    This method zeroes the entries in the chevron.

    Error checking: If `chv` is `NULL`, an error message is printed and the program exits.

26.2.12 IO methods

1. `void Chv_writeForHumanEye ( Chv *chv, FILE *fp ) ;`

   This method writes a `Chv` object to a file in an easily readable format.

   Error checking: If `chv` or `fp` are `NULL`, an error message is printed and zero is returned.

2. `void Chv_writeForMatlab ( Chv *chv, char *chvname, FILE *fp ) ;`

   This method writes a `Chv` object to a file in a matlab format. For a real chevron, a sample line is

   ```
   a(10,5) =  -1.550328201511e-01 ;
   ```

   where chvname = `"a"`. For a complex chevron, a sample line is

   ```
   a(10,5) =  -1.550328201511e-01 +   1.848033378871e+00*i;
   ```

   where chvname = `"a"`. The matrix indices come from the `rowind[]` and `colind[]` vectors, and are incremented by one to follow the Matlab and FORTRAN convention.

   Error checking: If `chv`, `chvname` or `fp` are `NULL`, an error message is printed and zero is returned.

26.3 Driver programs for the `Chv object`

1. `test_addChevron msglvl msgFile nD nU type symflag seed alphareal alphaimag`

   This driver program tests the `Chv_addChevron` method. Use the script file `do_addChevron` for testing. When the output file is loaded into matlab, the last line to the screen is the error of the assembly.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nD` parameter is the number of rows and columns in the (1,1) block.
   - The `nU` parameter is the number of columns in the (1,2) block.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
   - The `seed` parameter is a random number seed.
   - The `alphareal` and `alphaimag` parameters form a complex number that is a scaling parameter. Normally `alpha` is (1.0,0.0), when we are just loading matrix entries into a front. However, when we factor $A + \alpha B$, the entries of B will be loaded with `alpha` set equal to $\alpha[0:1]$.

2. `test_assmbChv msglvl msgFile nDJ nUJ nDI nUI type symflag seed`

   This driver program tests the `Chv_assembleChv` method. It assembles a chevron T_I into T_J, as is done during the assembly of postponed rows and columns during the factorization when pivoting is enabled. Use the script file `do_assmbChv` for testing. When the output file is loaded into matlab, the last line to the screen is the error of the assembly.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nDJ` parameter is the number of rows and columns in the (1,1) block of T_J.
   - The `nUJ` parameter is the number of columns in the (1,2) block of T_J.
   - The `nDI` parameter is the number of rows and columns in the (1,1) block of T_I.
   - The `nUI` parameter is the number of columns in the (1,2) block of T_I.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
   - The `seed` parameter is a random number seed.

3. `test_copyEntriesToVector msglvl msgFile nD nU type symflag pivotingflag storeflag seed`

   This driver program tests the `Chv_copyEntriesToVector` method which is used when after a front has been factored to store the entries into dense L and U submatrices. Use the script file `do_copyEntriesToVector` for testing. When the output file is loaded into matlab, the last line to the screen is a matrix that contains two entries. If the program executes correctly, these two entries should be exactly zero.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `nD` parameter is the number of rows and columns in the (1,1) block.
- The `nU` parameter is the number of columns in the (1,2) block.
- The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
- The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
- The `pivotingflag` parameter is the pivoting flag — `SPOOLES_NO_PIVOTING` for no pivoting, `SPOOLES_PIVOTING` for pivoting.
- The `storeflag` parameter is the storage flag, to store by rows, use `SPOOLES_BY_ROWS`, to store by columns, use `SPOOLES_BY_COLUMNS`.
- The `seed` parameter is a random number seed.

4. `test_copyBigEntriesToVector msglvl msgFile nD nU type symflag pivotingflag storeflag seed droptol`

   This driver program tests the `Chv_copyBigEntriesToVector` method which is used when after a front has been factored to store the entries into sparse L and U submatrices. Use the script file `do_copyBigEntriesToVector` for testing. When the output file is loaded into matlab, the last line to the screen is a matrix that contains three entries. The first two are the maximum magnitudes of the entries that were not copied (two different ways), and the third is the drop tolerance. If the program executes correctly, the third term is larger than the first two.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nD` parameter is the number of rows and columns in the (1,1) block.
   - The `nU` parameter is the number of columns in the (1,2) block.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
   - The `pivotingflag` parameter is the pivoting flag — `SPOOLES_NO_PIVOTING` for no pivoting, `SPOOLES_PIVOTING` for pivoting.
   - The `storeflag` parameter is the storage flag, to store by rows, use `SPOOLES_BY_ROWS`, to store by columns, use `SPOOLES_BY_COLUMNS`.
   - The `seed` parameter is a random number seed.
   - The `droptol` parameter is a drop tolerance parameters, entries whose magnitude is smaller than `droptol` are not copied.

5. `test_factor msglvl msgFile nD nU type symflag pivotingflag seed tau`

   This driver program tests the `Chv_factor` method. Use the script file `do_factor` for testing. When the output file is loaded into matlab, the last line to the screen is a matrix that contains three entries. The first entry is the error in the factorization. The second and third entries are the maximum magnitudes of the entries in L and U, respectively.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `nD` parameter is the number of rows and columns in the (1,1) block.
- The `nU` parameter is the number of columns in the (1,2) block.
- The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
- The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
- The `pivotingflag` parameter is the pivoting flag — `SPOOLES_NO_PIVOTING` for no pivoting, `SPOOLES_PIVOTING` for pivoting.
- The `seed` parameter is a random number seed.
- The `tau` parameter is used when pivoting is enabled. All entries in L and U will have magnitudes less than `tau`.

6. `test_findPivot msglvl msgFile nD nU type symflag seed tau`

   This driver program tests the `Chv_findPivot` method. Use the script file `do_findPivot` for testing. When the output file is loaded into matlab, look on the screen for the variables `maxerrupd` (the error in the factor and update), `ubound` (the maximum magnitude of the entries in U), and if nonsymmetric `lbound` (the maximum magnitude of the entries in L).

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nD` parameter is the number of rows and columns in the (1,1) block.
   - The `nU` parameter is the number of columns in the (1,2) block.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
   - The `seed` parameter is a random number seed.
   - The `tau` parameter is used when pivoting is enabled. All entries in L and U will have magnitudes less than `tau`.

7. `test_maxabs msglvl msgFile nD nU type symflag seed`

   This driver program tests the `Chv_maxabsInRow()`, `Chv_maxabsInRow11()`, `Chv_maxabsInColumn()`, `Chv_maxabsInColumn11()` and `Chv_maxabsInDiagonal11()` methods. Use the script file `do_maxabs` for testing. When the output file is loaded into matlab, look on the screen for the variables `rowerror`, `colerror`, `rowerror11`, `colerror11` and `diag11error`. All should be zero.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nD` parameter is the number of rows and columns in the (1,1) block.
   - The `nU` parameter is the number of columns in the (1,2) block.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

- The `pivotingflag` parameter is the pivoting flag — `SPOOLES_NO_PIVOTING` for no pivoting, `SPOOLES_PIVOTING` for pivoting.
- The `seed` parameter is a random number seed.

8. `test_r1upd msglvl msgFile nD nU type symflag seed`

   This driver program tests the `Chv_r1upd()` method. Use the script file `do_r1upd` for testing. When the output file is loaded into matlab, the last line is the error of the update.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nD` parameter is the number of rows and columns in the (1,1) block.
   - The `nU` parameter is the number of columns in the (1,2) block.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
   - The `seed` parameter is a random number seed.

9. `test_r2upd msglvl msgFile nD nU type symflag seed`

   This driver program tests the `Chv_r2upd()` method. Use the script file `do_r2upd` for testing. When the output file is loaded into matlab, the last line is the error of the update.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nD` parameter is the number of rows and columns in the (1,1) block.
   - The `nU` parameter is the number of columns in the (1,2) block.
   - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
   - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
   - The `seed` parameter is a random number seed.

10. `test_swap msglvl msgFile nD nU type symflag seed`

    This driver program tests three methods: `Chv_swapRowsAndColumns()`, `Chv_swapRows()` and `Chv_swapColumns()`. Use the script file `do_swap` for testing. When the output file is loaded into matlab, look for the `maxerrrowswap1`, `maxerrcolswap1`, `maxerrswap`, `maxerrsymswap1` and `maxerrsymswap2` values. All should be zero.

    - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `nD` parameter is the number of rows and columns in the (1,1) block.
    - The `nU` parameter is the number of columns in the (1,2) block.
    - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
    - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.

- The `seed` parameter is a random number seed.

11. `test_update msglvl msgFile type symflag sparsityflag`
    `ncolT ncolU nrowD nentU offset seed`

    This driver program tests the `Chv_updateH()`, `Chv_updateS()` and `Chv_updateN()` methods. The `Chv` object T is updated by $-U^T DU$, $-U^H DU$ or $-LDU$, depending on whether T is symmetric, hermitian or nonsymmetric. Use the script file `do_update` for testing. When the output file is loaded into matlab, the last line is the error in the update which should be zero.

    - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `type` parameter denotes the type of entries — `SPOOLES_REAL` or `SPOOLES_COMPLEX`
    - The `symflag` parameter is the symmetry flag — `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`.
    - The `sparsityflag` parameter should be zero for dense U and L, or 1 for sparse U and L.
    - The `ncolT` parameter is the number of columns in the (1,1) and (1,2) blocks of T.
    - The `nDT` parameter is the number of rows and columns in the (1,1) block of T.
    - The `ncolU` parameter is the number of columns in U.
    - The `nrowD` parameter is the number of rows and columns in D.
    - The `nentU` parameter is the number entries in U, ignored if `sparsityflag = 0`.
    - The `offset` parameter is the offset of first index in T from the last index in D.
    - The `seed` parameter is a random number seed.

Chapter 27

ChvList: Chv list object

This object was created to handle a list of lists of Chv objects during a matrix factorization. Its form and function is very close to the SubMtxList object that handles lists of lists of SubMtx objects during the forward and backsolves.

Here are the main properties.

1. There are a fixed number of lists, set when the ChvList object is initialized.
2. For each list there is an expected count, the number of times an object will be added to the list. (Note, a NULL object can be added to the list. In this case, nothing is added to the list, but its count is decremented.)
3. There is one lock for all the lists, but each list can be flagged as necessary to lock or not necessary to lock before an insertion, count decrement, or an extraction is made to the list.

The ChvList object manages a number of lists that may require handling critical sections of code. For example, one thread may want to add an object to a particular list while another thread is removing objects. The critical sections are hidden inside the ChvList object. Our factorization code does not know about any mutual exclusion locks that govern access to the lists.

There are four functions of the ChvList object.

- Is the incoming count for a list nonzero?
- Is a list nonempty?
- Add an object to a list (possibly a NULL object) and decrement the incoming count.
- Remove a subset of objects from a list.

The first two operations are queries, and can be done without locking the list. The third operation needs a lock only when two or more threads will be inserting objects into the list. The fourth operation requires a lock only when one thread will add an object while another thread removes the object and the incoming count is not yet zero.

Having a lock associated with a ChvList object is optional, for example, it is not needed during a serial factorization nor a MPI factorization. In the latter case there is one ChvList per process. For a multithreaded factorization there is one ChvList object that is shared by all threads. The mutual exclusion lock that is (optionally) embedded in the ChvList object is a Lock object from this library. It is inside the Lock object that we have a mutual exclusion lock. Presently we support the Solaris and POSIX thread packages. Porting the multithreaded codes to another platform should be simple if the POSIX thread package is present. Another type of thread package will require some modifications to the Lock object, but none to the ChvList objects.

27.1 Data Structure

The ChvList structure has the following fields.

- int nlist : number of lists.
- Chv **heads : vector of pointers to the heads of the list of Chv objects.
- int *counts : vector of incoming counts for the lists.
- Lock *lock : mutual exclusion lock.
- char *flags : vector of lock flags for the lists. If flags[ilist] == 'N', the list does not need to be locked. If flags[ilist] == 'Y', the list does need to be locked. Used only when lock is not NULL.
- int nlocks : total number of locks made on the mutual exclusion lock.

27.2 Prototypes and descriptions of ChvList methods

This section contains brief descriptions including prototypes of all methods that belong to the ChvList object.

27.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. ChvList * ChvList_new (void) ;

   This method simply allocates storage for the ChvList structure and then sets the default fields by a call to ChvList_setDefaultFields().

2. void ChvList_setDefaultFields (ChvList *list) ;

   The structure's fields are set to default values: nlist and nlocks set to zero, and heads, counts, lock and flags are set to NULL .

   Error checking: If list is NULL, an error message is printed and the program exits.

3. void ChvList_clearData (ChvList *list) ;

   This method clears the object and free's any owned data by calling Chv_free() for each object on the free list. If heads is not NULL, it is free'd. If counts is not NULL, it is free'd via a call to IVfree(). If flags is not NULL, it is free'd via a call to CVfree(). If the lock is not NULL, it is destroyed via a call to Lock_free(). There is a concluding call to ChvList_setDefaultFields().

   Error checking: If list is NULL, an error message is printed and the program exits.

4. void ChvList_free (ChvList *list) ;

   This method releases any storage by a call to ChvList_clearData() and then free the space for list.

   Error checking: If list is NULL, an error message is printed and the program exits.

27.2.2 Initialization methods

There are three initializer methods.

1. `void ChvList_init( ChvList *list, int nlist, int counts[],`
   `int lockflag, char flags[] ) ;`

   Any data is cleared via a call to `ChvList_clearData()`. The number of lists is set and the `heads[]` vector is initialized. If `counts` is not `NULL`, the object's `counts[]` vector is allocated and filled with the incoming entries. If `lockflag` is zero, the lock is not initialized. If `lockflag` is `1`, the lock is initialized to be able to synchronize threads with the calling process. If `lockflag` is `2`, the lock is initialized to be able to synchronize threads across processes. If `flags` is not `NULL`, the object's `flags[]` vector is allocated and filled with the incoming entries.

   Error checking: If `list` is `NULL`, or if `nlist` ≤ 0, or if `lockflag` is not in `[0,2]`, an error message is printed and zero is returned.

27.2.3 Utility methods

1. `int ChvList_isListNonempty ( ChvList *list, int ilist ) ;`

   If list `ilist` is empty, the method returns 0. Otherwise, the method returns 1.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

2. `int ChvList_isCountZero ( ChvList *list, int ilist ) ;`

   If `counts` is `NULL`, or if `counts[ilist]` equal to zero, the method returns 1. Otherwise, the method returns 0.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

3. `Chv * ChvList_getList ( ChvList *list, int ilist ) ;`

   If list `ilist` is empty, the method returns `NULL`. Otherwise, if the list needs to be locked, the lock is locked. The head of the list is saved to a pointer and then the head is set to `NULL`. If the list was locked, the number of locks is incremented and the lock unlocked. The saved pointer is returned.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

4. `void ChvList_addObjectToList ( ChvList *list, Chv *chv, int ilist ) ;`

   If the list needs to be locked, the lock is locked. If `chv` is not `NULL`, it is added to the head of the list. If `counts` is not `NULL`, then `counts[ilist]` is decremented. If the lock was locked, the number of locks is incremented and it is now unlocked.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

27.2.4 IO methods

1. `void ChvList_writeForHumanEye ( ChvList *list, FILE *fp ) ;`

   This method writes the list to a file in user readable form.

   Error checking: If `list` or `fp` are `NULL`, an error message is printed and zero is returned.

Chapter 28

ChvManager: Chv manager object

This object was created to handle a number of instances of Chv objects. Our codes are heavily dependent on dynamic memory management. This is partly due to the pivoting capability during the factorization and partly to the nondeterministic nature of parallel computation — we may not know ahead of time just what data structures will exist during the computations.

We wanted to be able to generate and re-use Chv objects, and we wanted to make the process somewhat transparent to other sections of the code. Towards this aim, there are two simple functions.

- Ask the manager object for a Chv object that has a certain amount of workspace.
- Return to the manager object a Chv object or list of objects that are no longer needed.

Where the manager object gets an instance, or what the manager does with the instance objects when they are returned to it, is of no concern to the user of the manager object — unless the process takes too much time or storage. We support two *modes* of behavior.

- *catch-and-release*

  In this mode the ChvManager object is just a front to malloc() and free() calls. The user asks for an object of a certain size, and the manager creates one using a call to malloc(). When the user returns an object, the manager releases the storage via a call to free().

- *recycle*

  In this mode the ChvManager object keeps a free pool of Chv objects. When the user requests a Chv object of a certain size, the manager searches the pool and finds one of that size or larger, removes the object from the pool, and returns the object to the user. (Our implementation finds *a* smallest object of that size or larger.) If there is no object on the free pool of sufficient size, one is created and returned. When the user releases an object to the manager, the object is placed on the free pool.

For the factorization, serial, multithreaded or MPI, we recommend using the recycling mode.

A multithreaded environment creates some difficulties. Should there be one manager object per thread, or should all the threads share one object? We have chosen the latter course, but this requires that a lock be present to guard the critical section of code where one searches or adds an object to the list. The lock we use is a Lock object, and so the ChvManager code is completely independent of the thread package. Porting to a new system might require some modification to the Lock, but none to the manager object.

Each manager object keeps track of certain statistics, bytes in their workspaces, the total number of bytes requested, the number of requests for a Chv objects, the number of releases, and the number of locks and unlocks.

28.1 Data Structure

The `ChvList` structure has the following fields.

- `Chv *head` : vector of pointers to the heads of the list of `Chv` objects.
- `Lock *lock` : mutual exclusion lock.
- `int mode` : behavior mode. When `mode = 0`, the object calls `SubMtx_new()` and `SubMtx_free()` to create and release objects. When `mode = 1`, the object recycles the objects.
- `int nactive` : number of active instances.
- `int nbytesactive` : number of bytes that are active.
- `int nbytesrequested` : number of bytes that have been requested.
- `int nbytesalloc` : number of bytes that have been allocated.
- `int nrequests` : number of requests for instances.
- `int releases` : number of instances that have been released.
- `int nlocks` : total number of locks made on the mutual exclusion lock. `int nunlocks` : total number of unlocks made on the mutual exclusion lock.

28.2 Prototypes and descriptions of `ChvManager` methods

This section contains brief descriptions including prototypes of all methods that belong to the `ChvManager` object.

28.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `ChvManager * ChvManager_new ( void ) ;`

   This method simply allocates storage for the `ChvManager` structure and then sets the default fields by a call to `ChvManager_setDefaultFields()`.

2. `void ChvManager_setDefaultFields ( ChvManager *manager ) ;`

   The structure's fields are set to default values: `mode`, `nactive`, `nbytesactive`, `nbytesrequested`, `nbytesalloc`, `nrequests`, `nreleases`, `nlocks` and `nunlocks` set to zero, and `head` and `lock` are set to `NULL` .

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

3. `void ChvManager_clearData ( ChvManager *manager ) ;`

   This method clears the object and free's any owned data by calling `Chv_free()` for each object on the free list. If the lock is not `NULL`, it is destroyed via a call to `mutex_destroy()` and then free'd. There is a concluding call to `ChvManager_setDefaultFields()`.

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

4. `void ChvManager_free ( ChvManager *manager ) ;`

   This method releases any storage by a call to `ChvManager_clearData()` and then free the space for `manager`.

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

28.2.2 Initialization methods

1. `void ChvManager_init( ChvManager *manager, int lockflag, int mode ) ;`

   Any data is cleared via a call to `ChvManager_clearData()`. If `lockflag` is zero, the lock is not initialized. If `lockflag` is `1`, the lock is initialized to be able to synchronize threads with the calling process. If `lockflag` is `2`, the lock is initialized to be able to synchronize threads across processes. The behavior mode is set to `mode`.

   Error checking: If `manager` is `NULL`, or if `lockflag` is not in `[0,2]`, or if `mode` is not in `[0,1]`, an error message is printed and the program exits.

28.2.3 Utility methods

1. ```
   Chv * ChvManager_newObjectOfSizeNbytes ( ChvManager *manager,
                                            int nbytesNeeded ) ;
   ```

   This method returns a `Chv` object whose workspace contains at least `nbytesNeeded` bytes.

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

2. `void ChvManager_releaseObject ( ChvManager *manager, Chv *chv ) ;`

   This method releases the `chv` instance into the free pool of objects.

   Error checking: If `manager` is `NULL`, an error message is printed and zero is returned.

3. `void ChvManager_releaseListOfObjects ( ChvManager *manager, Chv *chv ) ;`

   This method releases a list of `Chv` objects into the free pool of objects. The head of the list is the `chv` instance.

   Error checking: If `manager` is `NULL`, an error message is printed and zero is returned.

28.2.4 IO methods

1. `void ChvManager_writeForHumanEye ( ChvManager *manager, FILE *fp ) ;`

   This method writes the statistics to a file in user readable form.

   Error checking: If `manager` or `fp` are `NULL`, an error message is printed and zero is returned.
   ```

Chapter 29

DenseMtx: Dense matrix object

The DenseMtx object contains a dense matrix along with row and column indices. The entries in the matrix can be double precision real or double precision complex. It needs to be able to manage its own storage, much like the Chv and SubMtx objects that are used during the factor and solves, so we include this capability via a contained DV object. A DenseMtx object may also be found in a list, so there is a **next** field that points to another DenseMtx object.

The DenseMtx object also exists in an MPI environment, where it holds the solution and right hand side matrices. Since each of these two matrices is distributed, a processor *owns* only part of the global matrix, and so the need for row and column indices to specify which rows and columns are present on which processor.

29.1 Data Structure

The DenseMtx structure has the following fields.

- **int type** : type of entries, SPOOLES_REAL or SPOOLES_COMPLEX.
- **int rowid** : object's row id, default value is -1.
- **int colid** : object's column id, default value is -1.
- **int nrow** : number of rows
- **int ncol** : number of columns
- **int inc1** : row increment, difference in addresses between entries in the same column
- **int inc2** : column increment, difference in addresses between entries in the same row
- **int *rowind** : pointer to the base address of the int vector that contains row indices.
- **int *colind** : pointer to the base address of the int vector that contains column indices.
- **double *entries** : pointer to the base address of the double vector that contains the entries.
- **DV wrkDV** : object that manages the owned working storage.
- **DenseMtx *next** : link to a next object in a singly linked list.

One can query the type of entries via two macros.

- DENSEMTX_IS_REAL(mtx) returns 1 if the matrix has real entries, and 0 otherwise.
- DENSEMTX_IS_COMPLEX(mtx) returns 1 if the matrix has complex entries, and 0 otherwise.

29.2 Prototypes and descriptions of DenseMtx methods

This section contains brief descriptions including prototypes of all methods that belong to the DenseMtx object.

29.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `DenseMtx * DenseMtx_new ( void ) ;`

   This method simply allocates storage for the DenseMtx structure and then sets the default fields by a call to DenseMtx_setDefaultFields().

2. `void DenseMtx_setDefaultFields ( DenseMtx *mtx ) ;`

   The structure's fields are set to default values: type = SPOOLES_REAL, rowid = colid = -1, nrow = ncol = inc1 = inc2 = 0 and rowind = colind = entries = next = NULL . The wrkDV object has its default fields set via a call to DV_setDefaultFields().

   Error checking: If mtx is NULL, an error message is printed and the program exits.

3. `void DenseMtx_clearData ( DenseMtx *mtx ) ;`

   This method clears the object and free's any owned data by invoking the _clearData() methods for its internal DV object. There is a concluding call to DenseMtx_setDefaultFields().

   Error checking: If mtx is NULL, an error message is printed and the program exits.

4. `void DenseMtx_free ( DenseMtx *mtx ) ;`

   This method releases any storage by a call to DenseMtx_clearData() and then free the space for mtx.

   Error checking: If mtx is NULL, an error message is printed and the program exits.

29.2.2 Instance methods

1. `int DenseMtx_rowid ( DenseMtx *mtx ) ;`

   This method returns the *rowid* field of the object.

   Error checking: If mtx is NULL, an error message is printed and the program exits.

2. `int DenseMtx_colid ( DenseMtx *mtx ) ;`

   This method returns the *colid* field of the object.

   Error checking: If mtx is NULL, an error message is printed and the program exits.

3. `void DenseMtx_dimensions ( DenseMtx *mtx, int *pnrow, int *pncol ) ;`

   This method fills *pnrow and *pncol with nrow and ncol.

   Error checking: If mtx is NULL, an error message is printed and the program exits.

4. `int DenseMtx_columnIncrement ( DenseMtx *mtx ) ;`

   This method returns the row increment of the object, the difference in memory locations of two entries in consecutive columns in the same row.

   Error checking: If mtx is NULL, an error message is printed and the program exits.

5. `int DenseMtx_rowIncrement ( DenseMtx *mtx ) ;`

   This method returns the row increment of the object, the difference in memory locations of two entries in consecutive rows in the same column.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

6. `void DenseMtx_rowIndices ( DenseMtx *mtx, int *pnrow, **prowind ) ;`

   This method fills `*pnrow` with `nrow`, the number of rows, and `*prowind` with `rowind`, a pointer to the row indices.

   Error checking: If `mtx`, `pnrow` or `prowind` is `NULL`, an error message is printed and the program exits.

7. `void DenseMtx_columnIndices ( DenseMtx *mtx, int *pncol, **colind ) ;`

   This method fills `*pncol` with `ncol`, the number of columns, and `*pcolind` with `colind`, a pointer to the column indices.

   Error checking: If `mtx`, `pncol` or `pcolind` is `NULL`, an error message is printed and the program exits.

8. `double * DenseMtx_entries ( DenseMtx *mtx ) ;`

   This method returns the *entries* field of the object.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

9. `void * DenseMtx_workspace ( DenseMtx *mtx ) ;`

   This method returns a pointer to the base address of the object's workspace.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

10. `void  DenseMtx_realEntry ( DenseMtx *mtx, int irow, int jcol, double *pValue ) ;`

    This method fills `*pValue` with the entry in row `irow` and column `jcol`.

    Error checking: If `mtx` or `pValue` is `NULL`, or if the matrix is not real, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

11. ```
    void  DenseMtx_complexEntry ( DenseMtx *mtx, int irow, int jcol,
                                  double *pReal, double *pImag ) ;
    ```

    This method fills `*pReal` with the real part and `*pImag` with the imaginary part of the the entry in row `irow` and column `jcol`.

    Error checking: If `mtx`, `pReal` or `pImag` is `NULL`, or if the matrix is not complex, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

12. `void DenseMtx_setRealEntry ( DenseMtx *mtx, int irow, int jcol, double value) ;`

    This method sets the entry in row `irow` and column `jcol` to be `value`.

    Error checking: If `mtx` is `NULL`, or if the matrix is not real, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

13. ```
    void DenseMtx_setComplexEntry ( DenseMtx *mtx, int irow, int jcol,
                                    double real, double imag ) ;
    ```

    This method sets the real and imaginary parts of the entry in row `irow` and column `jcol` to be `(real,imag)`.

    Error checking: If `mtx` is `NULL`, or if the matrix is not complex, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

14. `int DenseMtx_row ( DenseMtx *mtx, int irow, double **prowent ) ;`

    This method fills `*prowent` with the first location of the entries in row `irow`.

    Return codes: `1` is a normal return, `-1` means `mtx` is `NULL`, `-2` means invalid type for `mtx`, `-3` means `irow` is out-of-range, `-4` means `prowent` is `NULL`.

15. `int DenseMtx_column ( DenseMtx *mtx, int jcol, double **pcolent ) ;`

    This method fills `*pcolent` with the first location of the entries in column `jcol`.

    Return codes: `1` is a normal return, `-1` means `mtx` is `NULL`, `-2` means invalid type for `mtx`, `-3` means `jcol` is out-of-range, `-4` means `pcolent` is `NULL`.

29.2.3 Initialization methods

There are three initializer methods.

1. ```
   void DenseMtx_init( DenseMtx *mtx, int type, int rowid, int colid,
                       int nrow, int ncol, int inc1, int inc2 ) ;
   ```

   This is the initializer method used when the `DenseMtx` object is to use its workspace to store indices and entries. The number of bytes required in the workspace is computed, the workspace is resized if necessary, and the scalar and pointer fields are set.

   Error checking: If `mtx` is `NULL`, or if `type` is neither `SPOOLES_REAL` nor `SPOOLES_COMPLEX`, or if `nrow`, `ncol`, `inc1` or `inc2` is less than or equal to zero, or if neither `inc1` nor `inc2` are `1`, an error message is printed and the program exits.

2. ```
   void DenseMtx_initWithPointers ( DenseMtx *mtx, int type, int rowid, int colid,
            int nD, int nL, int nU, int *rowind, int *colind, double *entries ) ;
   ```

   This is the initializer method used when the `DenseMtx` object does not own the storage for its indices and entries, but points into some other storage.

   Error checking: If `mtx` is `NULL`, or if `type` is neither `SPOOLES_REAL` nor `SPOOLES_COMPLEX`, or if `nrow`, `ncol`, `inc1` or `inc2` is less than or equal to zero, or if neither `inc1` nor `inc2` are `1`, or if `rowind`, `colind` or `entries` is `NULL`, an error message is printed and the program exits.

3. ```
   int DenseMtx_initAsSubmatrix ( DenseMtx *B, DenseMtx *A, int firstrow, int lastrow,
                                  int firstcol, int lastcol ) ;
   ```

   This method initializes `B` to contain rows `firstrow:lastrow` and columns `firstcol:lastcol` of `A`. Note, the `rowind`, `colind` and `entries` fields of `B` point into the indices and entries for `A`.

   Return codes: `1` is the normal return, `-1` means `B` is `NULL`, `-2` means `A` is `NULL`, `-3` means `A` has invalid type

1	normal return	-3	`A` has invalid type
-1	`B` is `NULL`	-4	requested rows are out-of-range
-2	`A` is `NULL`	-5	requested columns are out-of-range

4. `void DenseMtx_initFromBuffer ( DenseMtx *mtx ) ;`

   This method initializes the object using information present in the workspace buffer. This method is used to initialize the `DenseMtx` object when it has been received as an MPI message.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

5. `void DenseMtx_setA2 ( DenseMtx *mtx, A2 *a2 ) ;`

   This method initializes the `a2` object to point into the entries of the matrix.

   Error checking: If `mtx` or `a2` is `NULL`, an error message is printed and the program exits.
   ```

29.2.4 Utility methods

1. `int DenseMtx_nbytesNeeded ( int type, int nrow, int ncol ) ;`

   This method returns the number of bytes required to store the object's information in its buffer.

   Error checking: If `type` is neither `SPOOLES_REAL` nor `SPOOLES_COMPLEX`, or if `nrow` or `ncol` is less than zero, an error message is printed and the program exits.

2. `int DenseMtx_nbytesInWorkspace ( DenseMtx *mtx ) ;`

   This method returns the number of bytes in the workspace owned by this object.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `void DenseMtx_setNbytesInWorkspace ( DenseMtx *mtx, int nbytes ) ;`

   This method sets the number of bytes in the workspace of this object. If `nbytes` is less than the present number of bytes, the workspace is not resized.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. ```
   void DenseMtx_setFields( DenseMtx *mtx, int type, int rowid, int colid,
                            int nrow, int ncol, int inc1, int inc2 ) ;
   ```

   This method sets the scalar and pointer fields.

   Error checking: If `mtx` is `NULL`, or if `type` is neither `SPOOLES_REAL` nor `SPOOLES_COMPLEX`, or if `nrow`, `ncol`, `inc1` or `inc2` is less than or equal to zero, or if neither `inc1` nor `inc2` are `1`, an error message is printed and the program exits.

5. ```
   void DenseMtx_permuteRows ( DenseMtx *mtx, IV *oldToNewIV ) ;
   void DenseMtx_permuteColumns ( DenseMtx *mtx, IV *oldToNewIV ) ;
   ```

   These methods permute the rows or columns using an old-to-new permutation vector. The row or column ids are overwritten using the permutation vector, and then the rows or columns are sorted into ascending order.

   Error checking: If `mtx` or `oldToNewIV` is `NULL`, an error message is printed and the program exits.

6. `void DenseMtx_sort ( DenseMtx *mtx ) ;`

   This method sort the rows so the row ids are in ascending order and sorts the columns so the column ids are in ascending order.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

7. `void DenseMtx_copyRow ( DenseMtx *mtxB, int irowB, DenseMtx *mtxA, int irowA ) ;`

   This method copies row `irowA` from matrix `mtxA` into row `irowB` of matrix `mtxB`.

   Error checking: If `mtxB` is `NULL`, or if `irowB` is out of range, or if `mtxA` is `NULL`, or if `irowA` is out of range, or if the number of columns in `mtxB` and `mtxA` are not the same, an error message is printed and the program exits.

8. ```
   void DenseMtx_copyRowAndIndex ( DenseMtx *mtxB, int irowB,
                                   DenseMtx *mtxA, int irowA ) ;
   ```

   This method copies row `irowA` from matrix `mtxA` into row `irowB` of matrix `mtxB`, and copies the index of row `irowA` of `mtxA` into location `irowB` of the row indices for `mtxB`.

   Error checking: If `mtxB` is `NULL`, or if `irowB` is out of range, or if `mtxA` is `NULL`, or if `irowA` is out of range, or if the number of columns in `mtxB` and `mtxA` are not the same, an error message is printed and the program exits.
   ```

9. `void DenseMtx_addRow ( DenseMtx *mtxB, int irowB, DenseMtx *mtxA, int irowA ) ;`

   This method adds row `irowA` from matrix `mtxA` into row `irowB` of matrix `mtxB`.

   Error checking: If `mtxB` is `NULL`, or if `irowB` is out of range, or if `mtxA` is `NULL`, or if `irowA` is out of range, or if the number of columns in `mtxB` and `mtxA` are not the same, an error message is printed and the program exits.

10. `void DenseMtx_zero ( DenseMtx *mtx ) ;`

    This method zeros the entries in the matrix.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

11. `void DenseMtx_fillRandomEntries ( DenseMtx *mtx, Drand *drand ) ;`

    This method the entries in the matrix with random numbers using the `drand` object.

    Error checking: If `mtx` or `drand` is `NULL`, an error message is printed and the program exits.

12. `void DenseMtx_checksums ( DenseMtx *mtx, double sums[3] ) ;`

    This method fills `sums[0]` with the sum of the row indices, `sums[1]` with the sum of the column indices, and `sums[2]` with the sum of the magnitudes of the entries. This method is used to check the MPI method where a distributed matrix is re-distributed.

    Error checking: If `mtx` or `sums` is `NULL`, an error message is printed and the program exits.

13. `int DenseMtx_scale ( DenseMtx *mtx, double alpha[] ) ;`

    This method scales the entries in `mtx` by `alpha`.

    Return values: 1 for a normal return, -1 if `mtx` is NULL, -2 if `mtx` has an invalid type, -3 if `alpha` is NULL.

14. `double DenseMtx_maxabs ( DenseMtx *mtx ) ;`

    This method returns the entry of maximum magnitude of the entries.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

15. `double DenseMtx_sub ( DenseMtx *mtxB, *DenseMtx *mtxA ) ;`

    This method subtracts matrix `mtxA` from `mtxB` .

    Error checking: If `mtxA` or `mtxB` is `NULL`, an error message is printed and the program exits.

16. `double DenseMtx_copyRowIntoVector ( DenseMtx *mtx, int irow, double vec[] ) ;`

    This method copies row `irow` of matrix `mtx` into vector `vec[]`.

    Error checking: If `mtx` or `vec` is `NULL`, or if `irow` < 0 or `irow` $\geq$ `nrow`, an error message is printed and the program exits.

17. `double DenseMtx_copyVectorIntoRow ( DenseMtx *mtx, int irow, double vec[] ) ;`

    This method copies vector `vec[]` into row `irow` of matrix `mtx`.

    Error checking: If `mtx` or `vec` is `NULL`, or if `irow` < 0 or `irow` $\geq$ `nrow`, an error message is printed and the program exits.

18. `double DenseMtx_addVectorIntoRow ( DenseMtx *mtx, int irow, double vec[] ) ;`

    This method adds vector `vec[]` into row `irow` of matrix `mtx`.

    Error checking: If `mtx` or `vec` is `NULL`, or if `irow` < 0 or `irow` $\geq$ `nrow`, an error message is printed and the program exits.

29.2.5 IO methods

The file structure of a DenseMtx object is simple. First comes seven scalars, type, rowid, colid, nrow, ncol, inc1 and inc2, followed by the row indices, followed by the column indices, and then followed by the matrix entries.

1. `int DenseMtx_readFromFile ( DenseMtx *mtx, char *fn ) ;`

   This method reads an DenseMtx object from a file. If the the file can be opened successfully, the method calls DenseMtx_readFromFormattedFile() or DenseMtx_readFromBinaryFile(), closes the file and returns the value returned from the called routine.

   Error checking: If mtx or fn are NULL, or if fn is not of the form *.densemtxf (for a formatted file) or *.densemtxb (for a binary file), an error message is printed and the method returns zero.

2. `int DenseMtx_readFromFormattedFile ( DenseMtx *mtx, FILE *fp ) ;`

   This method reads an DenseMtx object from a formatted file. If there are no errors in reading the data, the value 1 is returned. If an IO error is encountered from fscanf, zero is returned.

   Error checking: If mtx or fp are NULL an error message is printed and zero is returned.

3. `int DenseMtx_readFromBinaryFile ( DenseMtx *mtx, FILE *fp ) ;`

   This method reads an DenseMtx object from a binary file. If there are no errors in reading the data, the value 1 is returned. If an IO error is encountered from fread, zero is returned.

   Error checking: If mtx or fp are NULL an error message is printed and zero is returned.

4. `int DenseMtx_writeToFile ( DenseMtx *mtx, char *fn ) ;`

   This method writes an DenseMtx object to a file. If the the file can be opened successfully, the method calls DenseMtx_writeFromFormattedFile() or DenseMtx_writeFromBinaryFile(), closes the file and returns the value returned from the called routine.

   Error checking: If mtx or fn are NULL, or if fn is not of the form *.densemtxf (for a formatted file) or *.densemtxb (for a binary file), an error message is printed and the method returns zero.

5. `int DenseMtx_writeToFormattedFile ( DenseMtx *mtx, FILE *fp ) ;`

   This method writes an DenseMtx object to a formatted file. If there are no errors in writing the data, the value 1 is returned. If an IO error is encountered from fprintf, zero is returned.

   Error checking: If mtx or fp are NULL an error message is printed and zero is returned.

6. `int DenseMtx_writeToBinaryFile ( DenseMtx *mtx, FILE *fp ) ;`

   This method writes an DenseMtx object to a binary file. If there are no errors in writing the data, the value 1 is returned. If an IO error is encountered from fwrite, zero is returned.

   Error checking: If mtx or fp are NULL an error message is printed and zero is returned.

7. `int DenseMtx_writeStats ( DenseMtx *mtx, FILE *fp ) ;`

   This method writes out a header and statistics to a file. The value 1 is returned.

   Error checking: If mtx or fp are NULL, an error message is printed and zero is returned.

8. `void DenseMtx_writeForHumanEye ( DenseMtx *mtx, FILE *fp ) ;`

   This method writes a DenseMtx object to a file in an easily readable format.

   Error checking: If mtx or fp are NULL, an error message is printed and zero is returned.

9. `void DenseMtx_writeForMatlab ( DenseMtx *mtx, char *mtxname, FILE *fp ) ;`

   This method writes out a `DenseMtx` object to a file in a Matlab format. A sample line is

   ```
   a(10,5) =  -1.550328201511e-01 +   1.848033378871e+00*i ;
   ```

   for complex matrices, or

   ```
   a(10,5) =  -1.550328201511e-01 ;
   ```

   for real matrices, where mtxname = `"a"`. The matrix indices come from the `rowind[]` and `colind[]` vectors, and are incremented by one to follow the Matlab and FORTRAN convention.

   Error checking: If `mtx`, `mtxname` or `fp` are `NULL`, an error message is printed and zero is returned.

Chapter 30

FrontMtx: Front matrix

The `FrontMtx` object is used to solve linear systems of equations by computing and using an LU or U^TDU factorization of a matrix or matrix pencil. The "front" in its name refers to a multifrontal formulation of the factor matrices. We don't actually use the multifrontal factorization method, (rather a left-looking block general sparse algorithm), but the storage of the factors and the computations are based on "fronts".

There are four orthogonal axes that describe a front matrix.

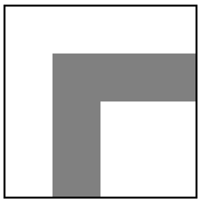
- The entries of the matrix can be double precision real or double precision complex.
- The factorization could be from a real or complex symmetric matrix, from a Hermitian matrix, or from a real or complex nonsymmetric matrix. In addition, the matrix can be represented as $A+\sigma B$, a linear combination of two matrices.
- The factorization can be performed with or without pivoting for numerical stability.
- The factorization can be *direct* or *approximate*. In the former case, the submatrices of the factors are stored as dense matrices. In the latter case, a user supplied drop tolerance is used to decide which entries to keep in the factorization.

The front matrix can exist in three different environments: serial, shared memory with parallelism enabled using Solaris or POSIX threads, and distributed memory using MPI.

This object computes, stores and solves linear systems using three types of factorizations:

1. $(A+\sigma B) = P(U^T+I)D(I+U)P^T$, where A and B are symmetric or Hermitian matrices. If pivoting is not enabled, D is a diagonal matrix. If pivoting is enabled, D has 1×1 and 2×2 blocks on its diagonal. U is strictly upper triangular, and the nonzero structures of U and D are disjoint. P is a permutation matrix. If pivoting is not used, P is the identity.
2. $(A+\sigma B) = P(L+I)D(I+U)Q^T$ for a square nonsymmetric matrix A with symmetric structure. D is a diagonal matrix. U is strictly upper triangular. L is strictly lower triangular. P and Q are permutation matrices. If pivoting is not used, P and Q are the identity.
3. $A = QR$ for square or rectangular A. Q is an orthogonal matrix that is not explicitly computed or stored. R is upper triangular.

The factorization is performed using a one dimensional decomposition of the global sparse matrix. A typical *front* of the matrix is found the shaded portion of the figure below.



A front is indivisible, it is found on one processor, and one processor or one thread is responsible for its internal computations. This is extremely important if we want to support pivoting for stability, for deciding how to choose the pivot elements in the front requires continuous up-to-date information about all the entries in the front. If a front were partitioned among threads or processors, the cost of the communication to select pivot elements would be intolerable.

Solving a nonsymmetric linear system $(A + \sigma B)X = B$ is done in the following steps.

- Factor $(A + \sigma B) = P(L + I)D(I + U)Q^T$.
- Solve $(L + I)Y = P^T B$
- Solve $DZ = Y$
- Solve $(I + U)W = Z$
- $X = QW$.

Release 1.0 used a one-dimensional data decomposition for the solves. Release 2.0 has changed to a two-dimensional data decomposition to increase the available parallelism. After the factorization is computed using a one-dimensional data decomposition, we post-process the matrix to obtain the two-dimensional decomposition and then perform the forward and backsolves.

To use the front matrix object, the user need know about only the initialization, factor, postprocess and solve methods. Here are the objects that a front matrix interacts with from the user's or "external" perspective.

- A sparse matrix A that is to be factored is contain in a `InpMtx` object. This object has been designed to be easy to use, to assemble and permute matrix entries, and to be put into a convenient form to be assembled into the front matrix. It contains real or complex matrix entries.
- The linear combination $A + \sigma B$ is found in a `Pencil` object.
- The `ETree` object contains the front tree that governs the factorization and solve. Inside this object are the dimensions of each front (the number of internal and external rows and columns), the tree connectivity of the fronts, and a map from each vertex to the front that contains it as an internal row and column. The `FrontMtx` object contains a pointer to an `ETree` object, but it does not modify the object, nor does it own the storage for the `ETree` object. Thus multiple front matrices can all point to the same `ETree` object simultaneously.
- An `IVL` object (`I`nteger `V`ector `L`ist), contains the symbolic factorization. For each front, it gives the list of internal and external rows and columns, used to initialize a front prior to its factorization. For a factorization without pivoting, this object stores the index information for the factors, and so is used during the forward and backsolves. For a factorization with pivoting, the index information for a front may change, so this object is not used during the solves. As for the `ETree` object, the symbolic factorization is neither modified or owned by the front matrix object.
- Working storage is necessary during the factor and solves. Instead of forcing one way of managing working storage, (e.g., simple `malloc` and `free's` or a complex management of one large work array), we have abstracted this behavior into two objects.

- The `SubMtxManager` object manages instances of the `SubMtx` object, used to store submatrices of the factors and working storage during the solves. The `FrontMtx` object contains a pointer to this manager object, set upon initialization.
- The `ChvManager` object manages instances of the `Chv` object, used to store fronts during the factorization. This manager object is passed to the front matrix object in a call to the factorization methods.

The user can easily override the behavior of these two manager objects. Our default supplied object are simple in their functionality — they are either wrappers around `malloc()` and `free()` calls, or they manage a pool of available objects. We measure their overhead and storage requirements during the factorizations and solve.

- The right hand side B and solution X are stored in `DenseMtx` objects. This object is a very simple wrapper around a dense matrix stored either column major or row major. (Our solves presently require the storage to be column major.) The matrices B and X can be either global (as in a serial or shared memory environment) or partitioned into local matrices (as in a distributed implementation).
- A parallel factorization requires a map from fronts to threads or processors, and this functionality is supplied by an `IV` (`Integer Vector`) object.
- The parallel solve requires a map from the submatrices to the threads or processors. This two-dimensional map is embodied in the `SolveMap` object.

To see how the front matrix object interacts with the other objects in the **SPOOLES** library, here is a brief description of the objects "internal" to the front matrix, its factorization and solve.

- The `Chv` object stores a front as a block *chevron*. Updates to the front, its assembly of postponed data (when pivoting is enabled) or aggregate data (in a parallel factorization), and the factorization of the fully assembled front, take place within the context of this object.
- The `SubMtx` object is used to store a submatrix of the factor matrices D, L and U. Once a front is factored it is split into one or more of these submatrix objects. After the factorization is complete, the data structures are postprocessed to yield submatrices that contain the coupling between fronts. The working storage during the solves is also managed by `SubMtx` objects.
- Each submatrix represents the coupling between two fronts, I and J. To enable rapid random access to these submatrices, we use a `I2Ohash` object that is a hash table whose keys are two integers and whose data is a `void *` pointer.
- The set of nonzero submatrices, i.e., the nonzero couplings between two fronts, is kept in one or two `IVL` objects. This information is necessary for the factorization and forward and backsolves.
- The factorization and solves require *lists* of fronts and submatrices to manage assembly of data and synchronization. We encapsulate these functions in the `ChvList` and `SubMtxList` objects that operate in serial, multithreaded and MPI environments.
- For a factorization with pivoting, the composition of a front (its dimensions and the row and column indices) may change, so we need additional data structures to store this information. We use an `IV` object to store the front size — the number of rows and columns that were eliminated when the front was factored. We use an `IVL` object to store the column indices — internal and external — and if the matrix is nonsymmetric, another `IVL` object to store the row indices.
- If we have a multithreaded factorization and use pivoting or an approximate factorization, we need exclusive access to the `IV` object that stores the final front size, and the `IVL` object(s) that store the final row and column indices for the front. Therefore we use a `Lock` object to govern exclusive access to these objects.

30.1 Data Structures

The `FrontMtx` structure has the following fields.

- `int nfront` : number of fronts.
- `int neqns` : number of rows and columns in the factor matrix.
- `int symmetryflag` : flag to denote the type of symmetry of $A + \sigma B$.
  - `SPOOLES_SYMMETRIC` — A and/or B are symmetric.
  - `SPOOLES_HERMITIAN` — A and/or B are hermitian.
  - `SPOOLES_NONSYMMETRIC` — A and/or B are nonsymmetric.
- `int pivotingflag` : flag to specify pivoting for stability,
  - `SPOOLES_NO_PIVOTING` — pivoting not used
  - `SPOOLES_PIVOTING` — pivoting used
- `int sparsityflag` : flag to specify storage of factors.
  - `0` — each front is dense
  - `1` — a front may be sparse due to entries dropped because they are below a drop tolerance.
- `int dataMode` : flag to specify data storage.
  - `1` — one-dimensional, used during the factorization.
  - `2` — two-dimensional, used during the solves.
- `int nentD` : number of entries in D
- `int nentL` : number of entries in L
- `int nentU` : number of entries in U
- `Tree *tree` : Tree object that holds the tree of fronts. Note, normally this is `frontETree->tree`, but we leave this here for later enhancements where we change the tree after the factorization, e.g., merge/drop fronts.
- `ETree *frontETree` : elimination tree object that holds the front tree.
- `IVL *symbfacIVL` : `IVL` object that holds the symbolic factorization.
- `IV *frontsizesIV` : `IV` object that holds the vector of front sizes, i.e., the number of internal rows and columns in a front.
- `IVL *rowadjIVL` : `IVL` object that holds the row list for the fronts, used only for a nonsymmetric factorization with pivoting enabled.
- `IVL *coladjIVL` : `IVL` object that holds the column list for the fronts, used only for a symmetric or nonsymmetric factorization with pivoting enabled.
- `IVL *lowerblockIVL` : `IVL` object that holds the front-to-front coupling in L, used only for a nonsymmetric factorization.
- `IVL *upperblockIVL` : `IVL` object that holds the front-to-front coupling in U.

- `SubMtx **p_mtxDJJ` : a vector of pointers to diagonal submatrices.
- `SubMtx **p_mtxUJJ` : a vector of pointers to submatrices in U that are on the block diagonal, used only during the factorization.
- `SubMtx **p_mtxUJN` : a vector of pointers to submatrices in U that are off the block diagonal, used only during the factorization.
- `SubMtx **p_mtxLJJ` : a vector of pointers to submatrices in L that are on the block diagonal, used only during a nonsymmetric factorization.
- `SubMtx **p_mtxLNJ` : a vector of pointers to submatrices in L that are off the block diagonal, used only during a nonsymmetric factorization.
- `I2Ohash *lowerhash` : pointer to a `I2Ohash` hash table for submatrices in L, used during the solves.
- `I2Ohash *upperhash` : pointer to a `I2Ohash` hash table for submatrices in U, used during the solves.
- `SubMtxManager *manager` : pointer to an object that manages the instances of submatrices during the factors and solves.
- `Lock *lock` : pointer to a `Lock` lock used in a multithreaded environment to ensure exlusive access while allocating storage in the `IV` and `IVL` objects. This is not used in a serial or MPI environment.
- `int nlocks` : number of times the lock has been locked.
- `PatchAndGo *info` : this is a pointer to an object that is used by the `Chv` object during the factorization of a front.

One can query the properties of the front matrix object using these simple macros.

- `FRONTMTX_IS_REAL(frontmtx)` is `1` if `frontmtx` has real entries and `0` otherwise.
- `FRONTMTX_IS_COMPLEX(frontmtx)` is `1` if `frontmtx` has complex entries and `0` otherwise.
- `FRONTMTX_IS_SYMMETRIC(frontmtx)` is `1` if `frontmtx` comes from a symmetric matrix or linear combination of symmetric matrices, and `0` otherwise.
- `FRONTMTX_IS_HERMITIAN(frontmtx)` is `1` if `frontmtx` comes from a Hermitian matrix or linear combination of Hermitian matrices, and `0` otherwise.
- `FRONTMTX_IS_NONSYMMETRIC(frontmtx)` is `1` if `frontmtx` comes from a nonsymmetric matrix or linear combination of nonsymmetric matrices, and `0` otherwise.
- `FRONTMTX_IS_DENSE_FRONTS(frontmtx)` is `1` if `frontmtx` comes from a direct factorization and so stores dense submatrices, and `0` otherwise.
- `FRONTMTX_IS_SPARSE_FRONTS(frontmtx)` is `1` if `frontmtx` comes from an approximate factorization and so stores sparse submatrices, and `0` otherwise.
- `FRONTMTX_IS_PIVOTING(frontmtx)` is `1` if pivoting was used during the factorization, and `0` otherwise.
- `FRONTMTX_IS_1D_MODE(frontmtx)` is `1` if the factor are still stored as a one-dimensional data decomposition (i.e., the matrix has not yet been post-processed), and `0` otherwise.
- `FRONTMTX_IS_2D_MODE(frontmtx)` is `1` if the factor are stored as a two-dimensional data decomposition (i.e., the matrix has been post-processed), and `0` otherwise.

30.2 Prototypes and descriptions of FrontMtx methods

This section contains brief descriptions including prototypes of all methods that belong to the FrontMtx object.

30.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `FrontMtx * FrontMtx_new ( void ) ;`

   This method simply allocates storage for the FrontMtx structure and then sets the default fields by a call to `FrontMtx_setDefaultFields()`.

2. `void FrontMtx_setDefaultFields ( FrontMtx *frontmtx ) ;`

   The structure's fields are set to default values: `nfront`, `neqns`, `nentD`, `nentL`, `nentU` and `nlocks` are set to zero. Five scalars are set to their default values,

type	=	SPOOLES_REAL
symmetryflag	=	SPOOLES_SYMMETRIC
sparsityflag	=	FRONTMTX_DENSE_FRONTS
pivotingflag	=	SPOOLES_NO_PIVOTING
dataMode	=	FRONTMTX_1D_MODE

   and the structure's pointers are set to NULL.

   Error checking: If `frontmtx` is NULL, an error message is printed and the program exits.

3. `void FrontMtx_clearData ( FrontMtx *frontmtx ) ;`

   This method clears the object and free's any owned data by invoking the `_clearData()` methods for its internal IV and IVL objects, (*not* including the `frontETree` and `symbfacIVL` objects that are not owned by this FrontMtx object). If the `lock` pointer is not NULL, the lock is destroyed via a call to `Lock_free()` and its storage is then free'd. There is a concluding call to `FrontMtx_setDefaultFields()`.

   Error checking: If `frontmtx` is NULL, an error message is printed and the program exits.

4. `void FrontMtx_free ( FrontMtx *frontmtx ) ;`

   This method releases any storage by a call to `FrontMtx_clearData()` and then free the space for `frontmtx`.

   Error checking: If `frontmtx` is NULL, an error message is printed and the program exits.

30.2.2 Instance methods

1. `int FrontMtx_nfront ( FrontMtx *frontmtx ) ;`

   This method returns the number of fronts in the matrix.

   Error checking: If `frontmtx` is NULL, an error message is printed and the program exits.

2. `int FrontMtx_neqns ( FrontMtx *frontmtx ) ;`

   This method returns the number of equations in the matrix.

   Error checking: If `frontmtx` is NULL, an error message is printed and the program exits.

3. `Tree * FrontMtx_frontTree ( FrontMtx *frontmtx ) ;`

   This method returns the `Tree` object for the fronts.

   Error checking: If `frontmtx` is `NULL`, an error message is printed and the program exits.

4. ```
   void FrontMtx_initialFrontDimensions ( FrontMtx *frontmtx, int J,
                               int *pnD, int *pnL, int *pnU, int *pnbytes ) ;
   ```

   This method fills the four pointer arguments with the number of internal rows and columns, number of rows in the lower block, number of columns in the upper block, and number of bytes for a `Chv` object to hold the front. in front `J`.

   Error checking: If `frontmtx` is `NULL`, or if `J` is not in `[0,nfront)`, or if any of the four pointer arguments are NULL, an error message is printed and the program exits.

5. `int FrontMtx_frontSize ( FrontMtx *frontmtx, int J ) ;`

   This method returns the number of internal rows and columns in front `J`.

   Error checking: If `frontmtx` or `frontsizesIV` is `NULL`, or if `J` is not in `[0,nfront)`, an error message is printed and the program exits.

6. `void FrontMtx_setFrontSize ( FrontMtx *frontmtx, int J, int size ) ;`

   This method sets the number of internal rows and columns in front `J` to be `size`. This method is used during factorizations with pivoting enabled since we cannot tell ahead of time how many rows and columns in a front will be eliminated.

   Error checking: If `frontmtx` or `frontsizesIV` is `NULL`, or if `J` is not in `[0,nfront)`, or if `size` < 0, an error message is printed and the program exits.

7. ```
   void FrontMtx_columnIndices ( FrontMtx *frontmtx, int J,
                                 int *pncol, int **pindices ) ;
   ```

   This method fills `*pncol` with the number of columns and `*pindices` with a pointer to the column indices for front `J`.

   Error checking: If `frontmtx`, `pncol` or `pindices` is `NULL`, or if `J` is not in `[0,nfront)`, an error message is printed and the program exits.

8. ```
   void FrontMtx_rowIndices ( FrontMtx *frontmtx, int J,
                              int *pnrow, int **pindices ) ;
   ```

   This method fills `*pnrow` with the number of rows and `*pindices` with a pointer to the row indices for front `J`.

   Error checking: If `frontmtx`, `pnrow` or `pindices` is `NULL`, or if `J` is not in `[0,nfront)`, an error message is printed and the program exits.

9. `SubMtx * FrontMtx_diagMtx ( FrontMtx *frontmtx, int J ) ;`

   This method returns a pointer to the object that contains submatrix $D_{J,J}$.

   Error checking: If `frontmtx` is `NULL`, or if `J` is not in `[0,nfront)`, an error message is printed and the program exits.

10. `SubMtx * FrontMtx_upperMtx ( FrontMtx *frontmtx, int J, int K ) ;`

    This method returns a pointer to the object that contains submatrix $U_{J,K}$. If $K = nfront$, then the object containing $U_{J,\partial J}$ is returned.

    Error checking: If `frontmtx` is `NULL`, or if `J` is not in `[0,nfront)`, or if `K` is not in `[0,nfront]`, an error message is printed and the program exits.
   ```

11. `SubMtx * FrontMtx_lowerMtx ( FrontMtx *frontmtx, int K, int J ) ;`

    This method returns a pointer to the object that contains submatrix $L_{K,J}$. If $K = nfront$, then the object containing $L_{\partial J,J}$ is returned.

    Error checking: If `frontmtx` is `NULL`, or if `J` is not in `[0,nfront)`, or if `K` is not in `[0,nfront]`, an error message is printed and the program exits.

12. ```
    void FrontMtx_lowerAdjFronts ( FrontMtx *frontmtx, int J,
                                   int *pnadj, int **padj ) ;
    ```

    This method fills `*pnadj` with the number of fronts adjacent to `J` in L and fills `*padj` with a pointer to the first entry of a vector containing the ids of the adjacent fronts.

    Error checking: If `frontmtx`, `pnadj` or `ppadj` is `NULL`, or if `J` is not in `[0,nfront)`, an error message is printed and the program exits.

13. ```
    void FrontMtx_upperAdjFronts ( FrontMtx *frontmtx, int J,
                                   int *pnadj, int **padj ) ;
    ```

    This method fills `*pnadj` with the number of fronts adjacent to `J` in U and fills `*padj` with a pointer to the first entry of a vector containing the ids of the adjacent fronts.

    Error checking: If `frontmtx`, `pnadj` or `ppadj` is `NULL`, or if `J` is not in `[0,nfront)`, an error message is printed and the program exits.

14. `int FrontMtx_nLowerBlocks ( FrontMtx *frontmtx ) ;`

    This method returns the number of nonzero $L_{K,J}$ submatrices.

    Error checking: If `frontmtx` is `NULL`, an error message is printed and the program exits.

15. `int FrontMtx_nUpperBlocks ( FrontMtx *frontmtx ) ;`

    This method returns the number of nonzero $U_{J,K}$ submatrices.

    Error checking: If `frontmtx` is `NULL`, an error message is printed and the program exits.

16. `IVL * FrontMtx_upperBlockIVL ( FrontMtx *frontmtx ) ;`

    This method returns a pointer to the `IVL` object that holds the upper blocks.

    Error checking: If `frontmtx` is `NULL`, an error message is printed and the program exits.

17. `IVL * FrontMtx_lowerBlockIVL ( FrontMtx *frontmtx ) ;`

    This method returns a pointer to the `IVL` object that holds the lower blocks.

    Error checking: If `frontmtx` is `NULL`, an error message is printed and the program exits.

30.2.3 Initialization methods

1. ```
   void FrontMtx_init ( FrontMtx *frontmtx, ETree *frontETree,
             IVL *symbfacIVL, int type, int symmetryflag, int sparsityflag,
             int pivotingflag, int lockflag, int myid, IV *ownersIV,
             SubMtxManager *manager, int msglvl, FILE *msgFile ) ;
   ```

   This method initializes the object, allocating and initializing the internal objects as necessary. See the previous section on data structures for the meanings of the `type`, `symmetryflag`, `sparsityflag` and `pivotingflag` parameters. The `lockflag` parameter has the following meaning.

   - `0` — the `Lock` object is not allocated or initialized.
   - `1` — the `Lock` object is allocated and initialized to synchronize only threads in this process.
   ```

- 2 — the `Lock` object is allocated and initialized to synchronize threads in this and other processes.

If `lockflag` is not `0`, the lock is allocated and initialized.

This method allocates as much storage as possible. When pivoting is not enabled and dense fronts are stored the structure of the factor matrix is fixed and given by the `frontETree` object. The diagonal $D_{J,J}$, upper triangular $U_{J,J}$ and $U_{J,\partial J}$ matrices, and lower triangular $L_{J,J}$ and $L_{\partial J,J}$ matrices are allocated.

The `myid` and `ownersIV` parameters are used in a distributed environment where we specify which process owns each front. When we can preallocate data structures (when there is no pivoting and dense fronts are stored) we need each process to determine what parts of the data it can allocate and set up. In a serial or multithreaded environment, use `ownersIV = NULL`.

Error checking: If `frontmtx`, `frontETree` or `symbfacIVL` is `NULL`, or if `type`, `symmetryflag`, `sparsityflag` or `pivotingflag` are not valid, or if `lockflag` is not `0`, `1` or `2`, or if `ownersIV` is not `NULL` and `myid < 0`, an error message is printed and the program exits.

30.2.4 Utility Factorization methods

The following methods are called by all the factor methods — serial, multithreaded and MPI.

1. `void FrontMtx_initializeFront ( FrontMtx *frontmtx, Chv *frontJ, int J ) ;`

   This method is called to initialize a front. The number of internal rows and columns is found from the front `ETree` object and the row and column indices are obtained from the symbolic factorization `IVL` object. The front `Chv` object is initialized via a call to `Chv_init()`, and the column indices and row indices (when nonsymemtric) are copied. Finally the front's entries are zeroed via a call to `Chv_zero()`.

   Error checking: None presently.

2. ```
   char FrontMtx_factorVisit ( FrontMtx *frontmtx, Pencil *pencil, int J,
      int myid, int owners[], Chv *fronts[], int lookahead, double tau,
      double droptol, char status[], IP *heads[], IV *pivotsizesIV, DV *workDV,
      int parent[], ChvList *aggList, ChvList *postList, ChvManager *chvmanager,
      int stats[], double cpus[], int msglvl, FILE *msgFile ) ;
   ```

   This method is called during the serial, multithreaded and MPI factorizations when front `J` is visited during the bottom-up traversal of the tree.

   Error checking: None presently.

3. ```
   Chv * FrontMtx_setupFront ( FrontMtx *frontmtx, Pencil *pencil, int J,
                               int myid, int owners[], ChvManager *chvmanager,
                               double cpus[], int msglvl, FILE *msgFile ) ;
   ```

   This method is called by `FrontMtx_visitFront()` to initialize the front's `Chv` object and load original entries if applicable.

   Error checking: None presently.

4. ```
   IP ** FrontMtx_factorSetup ( FrontMtx *frontmtx, IV *frontOwnersIV,
                                int myid, int msglvl, FILE *msgFile ) ;
   ```

   This method is called by the serial, multithreaded and MPI factorizations methods to initialize a data structure that contains the front-to-front updates that this thread or processor will perform. The data structure is a vector of pointers to `IP` objects that holds the heads of list of updates for each front.

   Error checking: None presently.
   ```

5. `int * FrontMtx_nactiveChild ( FrontMtx *frontmtx, char *status, int myid ) ;`

   This method is called by the multithreaded and MPI factorizations to create an integer vector that contains the number of active children of each front with respect to this thread or processor.

   Error checking: If `frontmtx` or `status` is `NULL`, or if `myid` < 0, an error message is printed and the program exits.

6. ```
   Ideq * FrontMtx_setUpDequeue ( FrontMtx *frontmtx, int owners[], int myid,
                                  char status[], IP *heads[], char activeFlag,
                                  char inactiveFlag, int msglvl, FILE *msgFile ) ;
   ```

   This method is called by the multithreaded and MPI factorizations to create and return an integer dequeue object to schedule the bottom-up traversal of the front tree.

   Error checking: If `frontmtx`, `owners` or `status` is `NULL`, or if `myid` < 0, an error message is printed and the program exits.

7. ```
   void FrontMtx_loadActiveLeaves ( FrontMtx *frontmtx, char status[],
                                    char activeFlag, Ideq *dequeue ) ;
   ```

   This method is called by the multithreaded and MPI factor and solve methods to load the dequeue with the active leaves in the front tree with respect to the thread or processor.

   Error checking: None presently.

8. ```
   ChvList * FrontMtx_postList ( FrontMtx *frontmtx, IV *frontOwnersIV,
                                 int lockflag ) ;
   ```

   This method is called by the multithreaded and MPI factor methods to create and return a list object to hold postponed chevrons and help synchronize the factorization.

   Error checking: None presently.

9. ```
   ChvList * FrontMtx_aggregateList ( FrontMtx *frontmtx,
                                      IV *frontOwnersIV, int lockflag ) ;
   ```

   This method is called by the multithreaded factor methods to create and return a list object to hold aggregate fronts and help synchronize the factorization. There is an analogous `FrontMtx_MPI_aggregateList()` method for the MPI environment.

   Error checking: If `frontmtx` or `frontOwnersIV` is `NULL`, or if `lockflag` is invalid, an error message is printed and the program exits.

10. ```
    void FrontMtx_loadEntries ( Chv *frontJ, DPencil *pencil,
                                int msglvl, FILE *msgFile) ;
    ```

    This method is called to load the original entries into a front.

    Error checking: If `frontJ` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

11. ```
    void FrontMtx_update ( FrontMtx *frontmtx, Chv *frontJ, IP *heads[],
                           char status[], DV *tempDV, int msglvl, FILE *msgFile) ;
    ```

    This method is called to update the current front stored in `frontJ` from all descendent fronts. (For the multithreaded and MPI factorizations, updates come from all owned descendent fronts.) The `heads[]` vector maintains the linked list of completed fronts that still have ancestors to update. The `tempDV` object is used as working storage by the `Chv` update methods, its size is automatically resized. When pivoting is disabled, the maximum size of the `tempDV` object is three times the maximum number of internal rows and columns in a front.

    Error checking: None presently.

12. `Chv * FrontMtx_assemblePostponedData ( FrontMtx *frontmtx, Chv *frontJ,`
    `ChvList *postponedlist, ChvManager *chvmanager, int *pndelay) ;`

    This method is called to assemble any postponed data from its children fronts into the current front. `frontJ` contains the updates from the descendents. Any postponed data is found in the list in `postponedlist`. If this list is empty, a new front is created to hold the aggregate updates and the postponed data, and the `chvmanager` object receives the aggregate and postponed `Chv` objects. The number of delayed rows and columns is returned in `*pndelay` — this is used during the factorization of the front that follows immediately.

    Error checking: None presently.

13. `FrontMtx_storePostponedData ( FrontMtx *frontmtx, Chv *frontJ,`
    `int npost, int K, ChvList *postponedlist, ChvManager *chvmanager ) ;`

    This method is used to store any postponed rows and columns from the current front `frontJ` into a `Chv` object obtained from the `chvmanager` object and place it into the list of postponed objects for `K`, its parent, found in the `postponedlist` object. The `frontJ` object is unchanged by this method.

    Error checking: None presently.

14. `FrontMtx_storeFront ( FrontMtx *frontmtx, Chv *frontJ, IV *pivotsizesIV,`
    `double droptol, int msglvl, FILE *msgFile ) ;`

    This method is used to store the eliminated rows and columns of the current front `frontJ` into the factor matrix storage.

    Error checking: None presently.

30.2.5 Serial Factorization method

There are two factorization methods: the first is for factoring a matrix A stored in a `DInpMtx` object, the second factors a linear combination $A + \sigma B$ stored in a `DPencil` object.

1. ```
   Chv * FrontMtx_factorInpMtx ( FrontMtx *frontmtx, InpMtx *inpmtx, double tau,
                       double droptol, ChvManager *chvmanager, int *perror,
                       double cpus[], int stats[],  int msglvl, FILE *msgFile ) ;
   Chv * FrontMtx_factorPencil ( FrontMtx *frontmtx, Pencil *pencil, double tau,
                       double droptol, ChvManager *chvmanager, int *perror,
                       double cpus[], int stats[],  int msglvl, FILE *msgFile ) ;
   ```

   These two serial factorization methods factor a matrix A (stored in `inpmtx`) or a matrix pencil $A+\sigma B$ (stored in `pencil`). The `tau` parameter is used when pivoting is enabled, each entry in U and L (when nonsymmetric) will have magnitude less than or equal to `tau`. The `droptol` parameter is used when the fronts are stored in a sparse format, each entry in U and L (when nonsymmetric) will have magnitude greater than or equal to `droptol`.

   The return value is a pointer to the first element in a list of `Chv` objects that contain the rows and columns that were not able to be eliminated. In all present cases, this should be `NULL`; we have left this return value as a hook to future factorizations via stages. The `perror` parameter is an address that is filled with an error code on return. If the factorization has completed, then `*perror` is a negative number. If `*perror` is in the range `[0,nfront)`, then an error has been detected at front `*perror`. On return, the `cpus[]` vector is filled with the following information.

   - `cpus[0]` — time spent initializing the fronts.
   - `cpus[1]` — time spent loading the original entries.
   - `cpus[2]` — time spent accumulating updates from descendents.
   ```

- `cpus[3]` — time spent assembling postponed data.
- `cpus[4]` — time spent to factor the fronts.
- `cpus[5]` — time spent to extract postponed data.
- `cpus[6]` — time spent to store the factor entries.
- `cpus[7]` — miscellaneous time.
- `cpus[8]` — total time in the method.

On return, the `stats[]` vector is filled with the following information.

- `stats[0]` — number of pivots.
- `stats[1]` — number of pivot tests.
- `stats[2]` — number of delayed rows and columns.
- `stats[3]` — number of entries in D.
- `stats[4]` — number of entries in L.
- `stats[5]` — number of entries in U.

Error checking: If `frontmtx`, `pencil`, `cpus` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

30.2.6 QR factorization utility methods

1. ```
   void FrontMtx_QR_setup ( FrontMtx *frontmtx, InpMtx *mtxA, IVL **prowsIVL,
                            int **pfirstnz, int msglvl, FILE *msgFile ) ;
   ```

   This method sets up the `rowsIVL` and `firstnz[]` data structures. The address of `rowsIVL` is placed in `*prowsIVL` and the address of `firstnz` is placed in `*pfirstnz`. List `J` of `rowsIVL` contains the rows of A that will be assembled into front `J`. The leading column with a nonzero entry in row `j` is found in `firstnz[j]`.

   Error checking: If `frontmtx`, `mtxA`, `prowsIVL` or `pfirstnz` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

2. ```
   void FrontMtx_QR_factorVisit ( FrontMtx *frontmtx, int J, InpMtx *mtxA,
        IVL *rowsIVL, int firstnz[], ChvList *updList, ChvManager *chvmanager,
        char status[], int colmap[], DV *workDV, double cpus[],
        double *pfacops, int msglvl, FILE *msgFile ) ;
   ```

   This method visits front `J` during the QR factorization. The number of operations to reduce the staircase matrix to upper trapezoidal or triangular form is incremented in `*pfacops`.

   Error checking: If `frontmtx`, `mtxA`, `rowsIVL`, `firstnz`, `updlist`, `chvmanager`, `status`, `colmap`, `workDV`, `cpus` or `pfacops` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

3. ```
   A2 * FrontMtx_QR_assembleFront ( FrontMtx *frontmtx, int J, InpMtx *mtxA,
               IVL *rowsIVL, int firstnz[], int colmap[], Chv *firstchild,
               DV *workDV, int msglvl, FILE *msgFile ) ;
   ```

   This method creates an `A2` object to hold the front, assembles any original rows of A and any update matrices from the children into the front, and then returns the front. The rows and update matrices are assembled into staircase form, so no subsequent permutations of the rows is necessary.

   Error checking: If `frontmtx`, `mtxA`, `rowsIVL`, `firstnz`, `colmap` or `workDV` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.
   ```

4. `void FrontMtx_QR_storeFront ( FrontMtx *frontmtx, int J, A2 *frontJ,`
   `int msglvl, FILE *msgFile ) ;`

   This method takes as input `frontJ`, the front in trapezoidal or triangular form. It scales the strict upper triangle or trapezoid by the diagonal entries, then squares the diagonal entries. (This transforms $R^T R$ into $(U^T + I)D(I + U)$ or $R^H R$ into $(U^H + I)D(I + U)$ for our solves.) It then stores the entries into the factor matrix.

   Error checking: If `frontmtx` or `frontJ` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

5. `Chv * FrontMtx_QR_storeUpdate ( FrontMtx *frontmtx, int J, A2 *frontJ,`
   `ChvManager *chvmanager, int msglvl, FILE *msgFile ) ;`

   This method takes as input `frontJ`, the front in trapezoidal or triangular form. It extracts the update matrix, stores the entries in a `Chv` object, and returns the `Chv` object. entries, then squares the diagonal entries.

   Error checking: If `frontmtx`, `frontJ` or `chvmanager` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

30.2.7 Serial QR Factorization method

1. `void FrontMtx_QR_factor ( FrontMtx *frontmtx, InpMtx *mtxA,`
   `ChvManager *chvmanager, double cpus[],`
   `double *pfacops,  int msglvl, FILE *msgFile ) ;`

   This method computes the $(U^T + I)D(I + U)$ factorization of $A^T A$ if A is real or $(U^H + I)D(I + U)$ factorization of $A^H A$ if A is complex. The `chvmanager` object manages the working storage. On return, the `cpus[]` vector is filled as follows.

   - `cpus[0]` – setup time, time to compute the `rowsIVL` and `firstnz[]` objects
   - `cpus[1]` – time to initialize and load the staircase matrices
   - `cpus[2]` – time to factor the matrices
   - `cpus[3]` – time to scale and store the factor entries
   - `cpus[4]` – time to store the update entries
   - `cpus[5]` – miscellaneous time
   - `cpus[6]` – total time

   On return, `*pfacops` contains the number of floating point operations done by the factorization.

   Error checking: If `frontmtx`, `frontJ` or `chvmanager` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

30.2.8 Postprocessing methods

1. `void FrontMtx_postProcess ( FrontMtx *frontmtx, int msglvl, FILE *msgFile ) ;`

   This method does post-processing chores after the factorization is complete. If pivoting was enabled, the method permutes the row and column adjacency objects, permutes the lower and upper matrices, and updates the block adjacency objects. The chevron submatrices $L_{\partial J,J}$ and $U_{J,\partial J}$ are split into $L_{K,J}$ and $U_{J,K}$ where $K \cap \partial J \neq \emptyset$.

   Error checking: If `frontmtx` is `NULL`, or if `msglvl` ¿ 0 and `msgFile` is `NULL`, an error message is printed and the program exits.

2. `void FrontMtx_permuteUpperAdj ( FrontMtx *frontmtx,`
   `int msglvl, FILE *msgFile ) ;`
   `void FrontMtx_permuteLowerAdj ( FrontMtx *frontmtx,`
   `int msglvl, FILE *msgFile ) ;`

   These methods are called during the postprocessing step, where they permute the upper and lower adjacency structures so that vertices in ∂J are in ascending order with respect to the indices in $K \cup \partial K$, where K is the parent of J.

   Error checking: If `frontmtx` is `NULL`, or if `msglvl` ¿ 0 and `msgFile` is `NULL`, an error message is printed and the program exits.

3. `void FrontMtx_permuteUpperMatrices ( FrontMtx *frontmtx,`
   `int msglvl, FILE *msgFile ) ;`
   `void FrontMtx_permuteLowerMatrices ( FrontMtx *frontmtx,`
   `int msglvl, FILE *msgFile ) ;`

   These methods are called during the postprocessing step, where they permute the upper $U_{J,\partial J}$ and lower $L_{\partial J,J}$ submatrices so that the columns in $U_{J,\partial J}$ and rows in $L_{\partial J,J}$ are in ascending order with the columns and rows of the final matrix.

   Error checking: If `frontmtx` is `NULL`, or if `msglvl` ¿ 0 and `msgFile` is `NULL`, an error message is printed and the program exits.

4. `void FrontMtx_splitUpperMatrices ( FrontMtx *frontmtx, int msglvl, FILE *msgFile ) ;`
   `void FrontMtx_splitLowerMatrices ( FrontMtx *frontmtx, int msglvl, FILE *msgFile ) ;`

   These methods are called during the postprocessing step, where they split the chevron submatrices $L_{\partial J,J}$ and $U_{J,\partial J}$ into $L_{K,J}$ and $U_{J,K}$ where $K \cap \partial J \neq \emptyset$.

   Error checking: If `frontmtx` is `NULL`, or if `msglvl` ¿ 0 and `msgFile` is `NULL`, an error message is printed and the program exits.

30.2.9 Utility Solve methods

The following methods are called by all the solve methods — serial, multithreaded and MPI.

1. `SubMtx ** FrontMtx_loadRightHandSide ( FrontMtx *frontmtx, DenseMtx *mtxB,`
   `int owners[], int myid, SubMtxManager *mtxmanager,`
   `int msglvl, FILE *msgFile ) ;`

   This method creates and returns a vector of pointers to `SubMtx` objects that hold pointers to the right hand side submatrices owned by the thread or processor.

   Error checking: None presently.

2. `void FrontMtx_forwardVisit ( FrontMtx *frontmtx, int J, int nrhs,`
   `int *owners, int myid, SubMtxManager *mtxmanager, SubMtxList *aggList,`
   `SubMtx *p_mtx[], char frontIsDone[], IP *heads[], SubMtx *p_agg[],`
   `char status[], int msglvl, FILE *msgFile) ;`

   This method is used to visit front `J` during the forward solve, $(U^T + I)Y = B$, $(U^H + I)Y = B$ or $(L + I)Y = B$.

   Error checking: None presently.

3. `void FrontMtx_diagonalVisit ( FrontMtx *frontmtx, int J, int owners[],`
   `int myid, SubMtx *p_mtx[], char frontIsDone[], SubMtx *p_agg[],`
   `int msglvl, FILE *msgFile ) ;`

This method is used to visit front `J` during the diagonal solve, $DZ = Y$.

Error checking: None presently.

4. `void FrontMtx_backwardVisit ( FrontMtx *frontmtx, int J, int nrhs, int *owners, int myid, SubMtxManager *mtxmanager, SubMtxList *aggList, SubMtx *p_mtx[], char frontIsDone[], IP *heads[], SubMtx *p_agg[], char status[], int msglvl, FILE *msgFile ) ;`

   This method is used to visit front `J` during the backward solve, $(U + I)Y = B$.

   Error checking: None presently.

5. `void FrontMtx_storeSolution ( FrontMtx *frontmtx, int owners[], int myid, SubMtxManager *mtxmanager, SubMtx *p_mtx[], DenseMtx *mtxX, int msglvl, FILE *msgFile ) ;`

   This method stores the solution in the `solmtx` dense matrix object.

   Error checking: None presently.

6. `IP ** FrontMtx_forwardSetup ( FrontMtx *frontmtx, int msglvl, FILE *msgFile ) ;`

   This method is used to set up a data structure of `IP` objects that hold the updates of the form $Y_J := Y_J - U_{I,J}^T X_I$, $Y_J := Y_J - U_{I,J}^H X_I$ or $Y_J := Y_J - L_{J,I} X_I$ that will be performed by this thread or processor.

   Error checking: None presently.

7. `IP ** FrontMtx_backwardSetup ( FrontMtx *frontmtx, int msglvl, FILE *msgFile ) ;`

   This method is used to set up a data structure of `IP` objects that hold the updates of the form $Z_J := Z_J - U_{J,K} X_K$ that will be performed by this thread or processor.

   Error checking: None presently.

8. `void FrontMtx_loadActiveRoots ( FrontMtx *frontmtx, char status[], char activeFlag, Ideq *dequeue ) ;`

   This method loads the active roots for a thread or a processor into the dequeue for the backward solve.

   Error checking: None presently.

30.2.10 Serial Solve method

1. `void FrontMtx_solve ( FrontMtx *frontmtx, DenseMtx *mtxX, DenseMtx *mtxB, SubMtxManager *mtxmanager, double cpus[], int msglvl, FILE *msgFile ) ;`

   This method is used to solve one of three linear systems of equations — $(U^T + I)D(I + U)X = B$, $(U^H + I)D(I + U)X = B$ or $(L + I)D(I + U)X = B$. Entries of B are *read* from `mtxB` and entries of X are written to `mtxX`. Therefore, `mtxX` and `mtxB` can be the same object. (Note, this does not hold true for an MPI factorization with pivoting.) The `mtxmanager` object manages the working storage using the solve. On return the `cpus[]` vector is filled with the following.

   - `cpus[0]` — set up the solves
   - `cpus[1]` — fetch right hand side and store solution
   - `cpus[2]` — forward solve
   - `cpus[3]` — diagonal solve
   - `cpus[4]` — backward solve

- `cpus[5]` — total time in the method.

Error checking: If `frontmtx`, `mtxB` or `cpus` is `NULL`, or if `msglvl` ¿ 0 and `msgFile` is `NULL`, an error message is printed and the program exits.

30.2.11 Serial QR Solve method

1. `void FrontMtx_QR_solve ( FrontMtx *frontmtx, InpMtx *mtxA, DenseMtx *mtxX, DenseMtx *mtxB, SubMtxManager *mtxmanager, double cpus[], int msglvl, FILE *msgFile ) ;`

   This method is used to minimize $\|B - AX\|_F$, where A is stored in `mtxA`, B is stored in `mtxB`, and X will be stored in `mtxX`. The `frontmtx` object contains a $(U^T + I)D(I + U)$ factorization of $A^T A$ if A is real or $(U^H + I)D(I + U)$ factorization of $A^H A$ if A is complex. We solve the seminormal equations $(U^T + I)D(I + U)X = A^T B$ or $(U^H + I)D(I + U)X = A^H B$ for X. The `mtxmanager` object manages the working storage used in the solves. On return the `cpus[]` vector is filled with the following.

   - `cpus[0]` — set up the solves
   - `cpus[1]` — fetch right hand side and store solution
   - `cpus[2]` — forward solve
   - `cpus[3]` — diagonal solve
   - `cpus[4]` — backward solve
   - `cpus[5]` — total time in the solve method.
   - `cpus[6]` — time to compute $A^T B$ or $A^H B$.
   - `cpus[7]` — total time.

   Error checking: If `frontmtx`, `mtxA`, `mtxX`, `mtxB` or `cpus` is `NULL`, or if `msglvl` ¿ 0 and `msgFile` is `NULL`, an error message is printed and the program exits.

30.2.12 Utility methods

1. `IV * FrontMtx_colmapIV ( FrontMtx *frontmtx ) ;`
   `IV * FrontMtx_rowmapIV ( FrontMtx *frontmtx ) ;`

   These methods construct and return an `IV` object that map the rows and columns to the fronts that contains them.

   Error checking: None presently.

2. `IV * FrontMtx_ownedRowsIV ( FrontMtx *frontmtx, int myid, IV *ownersIV, int msglvl, FILE *msgFile ) ;`
   `IV * FrontMtx_ownedColumnsIV ( FrontMtx *frontmtx, int myid, IV *ownersIV, int msglvl, FILE *msgFile ) ;`

   These methods construct and return `IV` objects that contain the ids of the rows and columns that belong to fronts that are owned by processor `myid`. If `ownersIV` is `NULL`, an `IV` object is returned that contains {`0,1,2,3, ..., nfront-1`}.

   Error checking: If `frontmtx` is `NULL`, an error message is printed and the program exits.

3. `IVL * FrontMtx_makeUpperBlockIVL ( FrontMtx *frontmtx, IV *colmapIV ) ;`
   `IVL * FrontMtx_makeLowerBlockIVL ( FrontMtx *frontmtx, IV *rowmapIV ) ;`

These methods construct and return `IVL` objects that contain the submatrix structure of the lower and upper factors. The `IV` objects map the rows and columns of the matrix to the fronts in the factor matrix that contain them.

Error checking: If `frontmtx`, `colmapIV` or `rowmapIV` are `NULL`, an error message is printed and the program exits.

4. `void FrontMtx_inertia ( FrontMtx *frontmtx, int *pnneg, int *pnzero, int *pnpos ) ;`

   This method determines the inertia of a symmetric matrix based on the $(U^T+I)D(I+U)$ factorization. The number of negative eigenvalues is returned in `*pnneg`, the number of zero eigenvalues is returned in `*pnzero`, and the number of positive eigenvalues is returned in `*pnpos`.

   Error checking: If `frontmtx`, `pnneg`, `pnzero` or `pnpos` is `NULL`, or if `symmetryflag` $\neq 0$ an error message is printed and the program exits.

5. `int FrontMtx_nSolveOps ( FrontMtx *frontmtx ) ;`

   This method computes and return the number of floating point operations for a solve with a single right hand side.

   Error checking: If `frontmtx` is `NULL`, or if `type` or `symmetryflag` are invalid, an error message is printed and the program exits.

30.2.13 IO methods

1. `int FrontMtx_readFromFile ( FrontMtx *frontmtx, char *fn ) ;`

   This method reads a `FrontMtx object` from a file. It tries to open the file and if it is successful, it then calls `FrontMtx_readFromFormattedFile()` or `FrontMtx_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `frontmtx` or `fn` are `NULL`, or if `fn` is not of the form `*.frontmtxf` (for a formatted file) or `*.frontmtxb` (for a binary file), an error message is printed and the method returns zero.

2. `int FrontMtx_readFromFormattedFile ( FrontMtx *frontmtx, FILE *fp ) ;`

   This method reads a `FrontMtx` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `frontmtx` or `fp` are `NULL` an error message is printed and zero is returned.

3. `int FrontMtx_readFromBinaryFile ( FrontMtx *frontmtx, FILE *fp ) ;`

   This method reads a `FrontMtx` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `frontmtx` or `fp` are `NULL` an error message is printed and zero is returned.

4. `int FrontMtx_writeToFile ( FrontMtx *frontmtx, char *fn ) ;`

   This method writes a `FrontMtx object` to a file. It tries to open the file and if it is successful, it then calls `FrontMtx_writeFromFormattedFile()` or `FrontMtx_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `frontmtx` or `fn` are `NULL`, or if `fn` is not of the form `*.frontmtxf` (for a formatted file) or `*.frontmtxb` (for a binary file), an error message is printed and the method returns zero.

5. `int FrontMtx_writeToFormattedFile ( FrontMtx *frontmtx, FILE *fp ) ;`

   This method writes a `FrontMtx` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `frontmtx` or `fp` are `NULL` an error message is printed and zero is returned.

6. `int FrontMtx_writeToBinaryFile ( FrontMtx *frontmtx, FILE *fp ) ;`

   This method writes a `FrontMtx` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `frontmtx` or `fp` are `NULL` an error message is printed and zero is returned.

7. `int FrontMtx_writeForHumanEye ( FrontMtx *frontmtx, FILE *fp ) ;`

   This method writes a `FrontMtx` object to a file in a human readable format. The method `FrontMtx_writeStats()` is called to write out the header and statistics. The value `1` is returned.

   Error checking: If `frontmtx` or `fp` are `NULL` an error message is printed and zero is returned.

8. `int FrontMtx_writeStats ( FrontMtx *frontmtx, FILE *fp ) ;`

   The header and statistics are written to a file. The value `1` is returned.

   Error checking: If `frontmtx` or `fp` are `NULL` an error message is printed and zero is returned.

9. ```
   int FrontMtx_writeForMatlab ( FrontMtx *frontmtx, char *Lname, char *Dname,
                                 char *Uname, FILE *fp ) ;
   ```

   This method writes out the factor matrix entries in a Matlab-readable form. `Lname` is a string for the lower triangular matrix, `Dname` is a string for the diagonal matrix, and `Uname` is a string for the upper triangular matrix.

   Error checking: If `frontmtx`, `Lname`, `Dname`, `Uname` or `fp` are `NULL`, an error message is printed and zero is returned.

30.3 Driver programs for the `DFrontMtx` object

1. ```
   testGrid msglvl msgFile n1 n2 n3 maxzeros maxsize seed type
            symmetryflag sparsityflag pivotingflag tau droptol
            lockflag nrhs
   ```

   This driver program tests the serial `FrontMtx_factor()` and `FrontMtx_solve()` methods for the linear system $AX = B$. Use the script file `do_grid` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of points in the first grid direction.
   - `n2` is the number of points in the second grid direction.
   - `n3` is the number of points in the third grid direction.
   - `maxzeros` is used to merge small fronts together into larger fronts. Look at the `ETree` object for the `ETree_mergeFronts{One,All,Any}()` methods.
   - `maxsize` is used to split large fronts into smaller fronts. See the `ETree_splitFronts()` method.
   - The `seed` parameter is a random number seed.
   - The `type` parameter specifies a real or complex linear system.
     - `type = 1 (SPOOLES_REAL)` for real,
     - `type = 2 (SPOOLES_COMPLEX)` for complex.
   - The `symmetryflag` parameter specifies the symmetry of the matrix.
     - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,

- `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
- `type = 2 (SPOOLES_NONSYMMETRIC)`

for A real or complex nonsymmetric.

- The `sparsityflag` parameter signals a direct or approximate factorization.
  - `sparsityflag = 0 (FRONTMTX_DENSE_FRONTS)` implies a direct factorization, the fronts will be stored as dense submatrices.
  - `sparsityflag = 1 (FRONTMTX_SPARSE_FRONTS)` implies an approximate factorization. The fronts will be stored as sparse submatrices, where the entries in the triangular factors will be subjected to a drop tolerance test — if the magnitude of an entry is `droptol` or larger, it will be stored, otherwise it will be dropped.
- The `pivotingflag` parameter signals whether pivoting for stability will be enabled or not.
  - If `pivotingflag = 0 (SPOOLES_NO_PIVOTING)`, no pivoting will be done.
  - If `pivotingflag = 1 (SPOOLES_PIVOTING)`, pivoting will be done to ensure that all entries in U and L have magnitude less than `tau`.
- The `tau` parameter is an upper bound on the magnitude of the entries in L and U when pivoting is enabled.
- The `droptol` parameter is a lower bound on the magnitude of the entries in L and U when the approximate factorization is enabled.
- When `lockflag` is zero, the mutual exclusion lock for the factor matrix is not enabled. When `lockflag` is not zero, the mutual exclusion lock is set. This capability is here to test the overhead for the locks for a serial factorization.
- The `nrhs` parameter is the number of right hand sides to solve as one block.

2. `testQRgrid msglvl msgFile n1 n2 n3 seed nrhs type`

   This driver program tests the serial `FrontMtx_QR_factor()` and `FrontMtx_QR_solve()` methods for the least squares problem $\min_X \|F - AX\|_F$.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of points in the first grid direction.
   - `n2` is the number of points in the second grid direction.
   - `n3` is the number of points in the third grid direction.
   - The `seed` parameter is a random number seed.
   - The `nrhs` parameter is the number of right hand sides to solve as one block.
   - The `type` parameter specifies a real or complex linear system.
     - `type = 1 (SPOOLES_REAL)` for real,
     - `type = 2 (SPOOLES_COMPLEX)` for complex.

Chapter 31

ILUMtx: Incomplete LU Matrix Object

The ILUMtx object represents and approximate (incomplete) $(L+I)D(I+U)$, $(U^T+I)D(I+U)$ or $(U^H+I)D(I+U)$ factorization. It is a very simple object, rows and columns of L and U are stored as single vectors. All computations to compute the factorization and to solve linear systems are performed with sparse BLAS1 kernels. Presently, the storage scheme is very simple minded, we use malloc() and free() to handle the individual vectors of the rows and columns of L and U.

At present we have one factorization method. No pivoting is performed. Rows of U are stored, along with columns of L if the matrix is nonsymmetric. If a zero pivot is encountered on the diagonal during the factorization, the computation stops and returns a nonzero error code. (Presently, there is no "patch-and-go" functionality.) An $L_{j,i}$ entry is kept if $|L_{j,i}D_{i,i}| \geq \sigma\sqrt{|D_{i,i}|\ |A_{j,j}|}$, where σ is a user supplied drop tolerance, and similarly for $U_{i,j}$. Note, if $A_{j,j} = 0$, as is common for KKT matrices, all $L_{j,i}$ and $U_{i,j}$ entries will be kept. It is simple to modify the code to use another drop tolerance criteria, e.g., an absolute tolerance, or one based only on $|D_{i,i}|$. We intend to write other factorization methods that will conform to a user-supplied nonzero structure for the factors.

31.1 Data Structure

The ILUMtx structure has the following fields.

- `int neqns` : number of equations.
- `int type` : type of entries, SPOOLES_REAL or SPOOLES_COMPLEX.
- `int symmetryflag` : type of matrix symmetry, SPOOLES_SYMMETRIC, SPOOLES_HERMITIAN or SPOOLES_NONSYMMETRIC.
- `int UstorageMode` : type of storage for U, SPOOLES_BY_ROWS or SPOOLES_BY_COLUMNS.
- `int LstorageMode` : type of storage for L, SPOOLES_BY_ROWS or SPOOLES_BY_COLUMNS.
- `double *entD` : vector of diagonal entries.
- `int *sizesL` : vector of sizes of the off-diagonal vectors of L, not used if the matrix is symmetric or Hermitian.
- `int **p_indL` : vector of pointers to the indicies vectors of L, not used if the matrix is symmetric or Hermitian.
- `double **p_entL` : vector of pointers to the entries vectors of L, not used if the matrix is symmetric or Hermitian.

- `int *sizesU` : vector of sizes of the off-diagonal vectors of U.
- `int **p_indU` : vector of pointers to the indicies vectors of U.
- `double **p_entU` : vector of pointers to the entries vectors of U.

One can query the attributes of the object with the following macros.

- `ILUMTX_IS_REAL(mtx)` returns `1` if the entries are real, and `0` otherwise.
- `ILUMTX_IS_COMPLEX(mtx)` returns `1` if the entries are complex, and `0` otherwise.
- `ILUMTX_IS_SYMMETRIC(mtx)` returns `1` if the factorization is symmetric, and `0` otherwise.
- `ILUMTX_IS_HERMITIAN(mtx)` returns `1` if the factorization is Hermitian, and `0` otherwise.
- `ILUMTX_IS_NONSYMMETRIC(mtx)` returns `1` if the factorization is nonsymmetric, and `0` otherwise.
- `ILUMTX_IS_L_BY_ROWS(mtx)` returns `1` if L is stored by rows, and `0` otherwise.
- `ILUMTX_IS_L_BY_COLUMNS(mtx)` returns `1` if L is stored by columns, and `0` otherwise.
- `ILUMTX_IS_U_BY_ROWS(mtx)` returns `1` if U is stored by rows, and `0` otherwise.
- `ILUMTX_IS_U_BY_COLUMNS(mtx)` returns `1` if U is stored by columns, and `0` otherwise.

31.2 Prototypes and descriptions of ILUMtx methods

This section contains brief descriptions including prototypes of all methods that belong to the `ILUMtx` object.

31.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `ILUMtx * ILUMtx_new ( void ) ;`

   This method simply allocates storage for the `ILUMtx` structure and then sets the default fields by a call to `ILUMtx_setDefaultFields()`.

2. `int ILUMtx_setDefaultFields ( ILUMtx *mtx ) ;`

   This method sets the structure's fields to default values: `neqns = 0`, `type` = `SPOOLES_REAL`, `symmetryflag` = `SPOOLES_SYMMETRIC`, `UstorageMode` = `SPOOLES_BY_ROWS`, `LstorageMode` = `SPOOLES_BY_COLUMNS`, and `entD`, `sizesL`, `p_indL`, `p_entL`, `sizesU`, `p_indU` and `p_entU` are all set to `NULL`.

   Return codes: `1` means a normal return, `-1` means `mtx` is `NULL`.

3. `int ILUMtx_clearData ( ILUMtx *mtx ) ;`

   This method releases all storage held by the object.

   Return codes: `1` means a normal return, `-1` means `mtx` is `NULL`.

4. `int ILUMtx_free ( ILUMtx *mtx ) ;`

   This method releases all storage held by the object via a call to `ILUMtx_clearData()`, then free'd the storage for the object.

   Return codes: `1` means a normal return, `-1` means `mtx` is `NULL`.

31.2.2 Initialization Methods

1. `int ILUMtx_init ( ILUMtx *mtx, int neqns, int type, int symmetryflag, int LstorageMode, int UstorageMode ) ;`

   This is the initializer method that should be called immediately after `ILUMtx_new()`. It first clears any previous data with a call to `ILUMtx_clearData()`. The object's scalar fields are then set. The `sizesU` (and `sizesL` if nonsymmetric) vector(s) are then initialized and filled with zeros. The `p_indU`, `p_entU` (and `p_indL` and `p_entL` if nonsymmetric) vectors of pointers are initialized and filled with `NULL` values. The `entD` vector is initialized and filled with zeros.

   Return codes:

1	normal return	-4	`symmetryflag` is invalid
-1	`mtx` is `NULL`	-5	`LstorageMode` is invalid
-2	`neqns` <= 0	-6	`UstorageMode` is invalid
-3	`type` is invalid	-7	`type` and storage modes do not match

31.2.3 Factorization Methods

1. `int ILUMtx_factor ( ILUMtx *mtx, InpMtx *mtxA, double sigma, double *pops, int msglvl, FILE *msgFile ) ;`

   This methods computes a drop tolerance $A = (L+I)D(I+U)$, $A = (U^T+I)D(I+U)$ or $A = (U^H+I)D(I+U)$ factorization. An $L_{j,i}$ entry is kept if $|L_{j,i}D_{i,i}| \geq \sigma\sqrt{|D_{i,i}|\ |A_{j,j}|}$, where σ is a user supplied drop tolerance, and similarly for $U_{i,j}$. If `pops` is not `NULL`, then on return `*pops` holds the number of floating point operations that was performed during the factorization.

   Return codes:

1	normal return	-10	`p_indU` is `NULL`
-1	`mtx` is `NULL`	-11	`entD` is `NULL`
-2	`neqns` <= 0	-12	`p_entL` is `NULL`
-3	`type` is invalid	-13	`p_entU` is `NULL`
-4	`symmetryflag` is invalid	-14	`mtxA` is `NULL`
-5	`LstorageMode` is invalid	-15	types of `mtxLDU` and `mtxA` do not match
-6	`UstorageMode` is invalid	-16	`mtxA` is not in chevron mode
-7	`sizesL` is `NULL`	-17	$sigma < 0$
-8	`sizesU` is `NULL`	-18	`msglvl > 0` and `msgFile` is `NULL`
-9	`p_indL` is `NULL`	-19	singular pivot found

31.2.4 Solve Methods

1. `int ILUMtx_solveVector ( ILUMtx *mtx, DV *X, DV *B, DV *workDV, double *pops, int msglvl, FILE *msgFile ) ;`

   This methods solves a linear system $(L+I)D(I+U)x = b$, $(U^T+I)D(I+U)x = b$ or $(U^H+I)D(I+U)x = b$. `workDV` is a work vector. If `workDV` is different that `B`, then `B` is unchanged on return. One can have `X`, `B` and `workDV` point to the same object. If `pops` is not `NULL`, then on return `*pops` holds the number of floating point operations that was performed during the solve.

   Return codes:

1	normal return	-12	`p_entL` is `NULL`
-1	`mtx` is `NULL`	-13	`p_entU` is `NULL`
-2	`neqns` <= 0	-14	`X` is `NULL`
-3	`type` is invalid	-15	size of `X` is incorrect
-4	`symmetryflag` is invalid	-16	entries of `X` are `NULL`
-5	`LstorageMode` is invalid	-17	`B` is `NULL`
-6	`UstorageMode` is invalid	-18	size of `B` is incorrect
-7	`sizesL` is `NULL`	-19	entries of `B` are `NULL`
-8	`sizesU` is `NULL`	-20	`workDV` is `NULL`
-9	`p_indL` is `NULL`	-21	size of `workDV` is incorrect
-10	`p_indU` is `NULL`	-22	entries of `workDV` are `NULL`
-11	`entD` is `NULL`	-23	`msglvl > 0` and `msgFile` is `NULL`

31.2.5 Utility methods

1. `int ILUMtx_fillRandom ( ILUMtx *mtx, int seed ) ;`

   This method fills the `mtx` object with a random nonzero pattern and random matrix entries. The matrix must have already been initialized using the `ILUMtx_init()` method.

   Return codes:

1	normal return	-5	`LstorageMode` is invalid	-10	`p_indU` is `NULL`
-1	`mtx` is `NULL`	-6	`UstorageMode` is invalid	-11	`entD` is `NULL`
-2	`neqns` <= 0	-7	`sizesL` is `NULL`	-12	`p_entL` is `NULL`
-3	`type` is invalid	-8	`sizesU` is `NULL`	-13	`p_entU` is `NULL`
-4	`symmetryflag` is invalid	-9	`p_indL` is `NULL`		

31.2.6 IO methods

1. ```
   int ILUMtx_writeForMatlab ( ILUMtx *mtx, char *Lname, char *Dname,
                               char *Uname, FILE *fp ) ;
   ```

   This method writes out a `ILUMtx` object to a file in a Matlab format. The entries in L use the `Lname` string, the entries in D use the `Dname` string, and the entries in U use the `Uname` string, A sample line is

   ```
   L(10,5) =  -1.550328201511e-01 +   1.848033378871e+00*i ;
   ```

   for complex matrices, or

   ```
   L(10,5) =  -1.550328201511e-01 ;
   ```

   for real matrices, where `Lname` = `"L"`. The matrix indices are incremented by one to follow the Matlab and FORTRAN convention.

   Return codes:

1	normal return	-6	`UstorageMode` is invalid	-12	`p_entL` is `NULL`
-1	`mtx` is `NULL`	-7	`sizesL` is `NULL`	-13	`p_entU` is `NULL`
-2	`neqns` <= 0	-8	`sizesU` is `NULL`	-14	`Lname` is `NULL`
-3	`type` is invalid	-9	`p_indL` is `NULL`	-15	`Dname` is `NULL`
-4	`symmetryflag` is invalid	-10	`p_indU` is `NULL`	-16	`Uname` is `NULL`
-5	`LstorageMode` is invalid	-11	`entD` is `NULL`	-17	`fp` is `NULL`
   ```

31.3 Driver programs for the ILUMtx object

This section contains brief descriptions of the driver programs.

1. `testFactor msglvl msgFile type symflag`
   `neqns nitem seed sigma matlabFile`

   This driver program generates a random matrix A stored in an `InpMtx` object. It then factors $A = (L+I)D(I+U)$, $A = (U^T+I)D(I+U)$ or $A = (U^H+I)D(I+U)$ (depending on `type` and `symflag`). If `matlabFile` is not `"none"`, it writes A, L, D and U to a Matlab file, which can then be run through matlab to compute the error in the factorization. The CPU, number of operations and megaflops for the factorization are printed to `msgFile`.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `type` must be either `SPOOLES_REAL` or `SPOOLES_COMPLEX`.
   - `symflag` must be either `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_SYMMETRIC`.
   - `neqns` is the number of equations, must be positive.
   - `nitem` is the number of off-diagonal entries, must be nonnegative.
   - `seed` is a random number seed.
   - `sigma` is the drop tolerance.
   - `matlabFile` is the name of the Matlab file for the matrices. If `"none"` then no output is written.

2. `testSolve msglvl msgFile neqns type symflag`
   `LstorageMode UstorageMode seed matlabFile`

   This driver program solve a linear system $(L+I)D(I+U)X = B$, $(U^T+I)D(I+U)X = B$ or $(U^H+I)D(I+U)X = B$, depending on `type` and `symflag`. L, D and L are random sparse matrices and B is a random vector. If `matlabFile` is not `"none"`, it writes L, D, U, B and the computed solution X to a Matlab file, which can then be run through matlab to compute the error in the solve. The CPU, number of operations and megaflops for the factorization are printed to `msgFile`.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `neqns` is the number of equations, must be positive.
   - `type` must be either `SPOOLES_REAL` or `SPOOLES_COMPLEX`.
   - `symflag` must be either `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_SYMMETRIC`.
   - `LstorageMode` must be either `SPOOLES_BY_ROWS` or `SPOOLES_BY_COLUMNS`.
   - `UstorageMode` must be either `SPOOLES_BY_ROWS` or `SPOOLES_BY_COLUMNS`.
   - `seed` is a random number seed.
   - `matlabFile` is the name of the Matlab file for the matrices. If `"none"` then no output is written.

Chapter 32

InpMtx: Input Matrix Object

The InpMtx object has two functions:

- It is used to assemble a sparse matrix (or just its structure) from individual entries, rows, columns or dense submatrices (or any combination of these) that may overlap.
- It is used to communicate entries of a matrix into a front during the factorization.

We have designed this object to be easy to use, but it has one significant drawback — it is an in-core implementation, and this is a disadvantage in situations where memory is limited. Extending this object to work out-of-core is not difficult, but we leave that *value-added* function to others in the future.

The InpMtx object has three faces. It can just manipulate (i, j) pairs, where it assembles just the nonzero structure of a matrix. We use this functionality to generate a Graph object that is needed as input to the ordering software. Alternatively, it can assemble and manipulate $(i, j, a_{i,j}$ triples where $a_{i,j}$ is either a real or complex number. (At any one time, the object works with either no numbers, real numbers or complex numbers but not mixtures of the three.) The normal input to the InpMtx object is a collection of matrix entries in some form, e.g., single entries, (partial) rows or columns, or dense submatrices.

Here is a common sequence of events to use this object when we want to build the structure of a sparse matrix.

1. Create an instance of a InpMtx object using the InpMtx_new() method.
2. Initialize the InpMtx object using the InpMtx_init() method; set the input mode to indices only, maximum number of entries for the workspace, and the number of vectors. (The latter two quantities may be zero, for the object resizes its storage as required.)
3. Call the method InpMtx_changeCoordType() to set the coordinate type to rows.
4. Load data into the object using one or more of the five input methods: InpMtx_inputEntry(), InpMtx_inputRow(), InpMtx_inputColumn(), InpMtx_inputMatrix() and InpMtx_inputTriples() methods. Each time the workspace fills up, the raw data is sorted and compressed and then the workspace is resized. If the input data overlaps, e.g., elemental matrices are being assembled, it would be efficient to have sufficient elbow room to minimize the number of sorts and compressions. In this case, a tight upper bound on the necessary storage is the sum of the sizes of the elemental matrices. The entries are assembled by a call to InpMtx_changeStorageMode().
5. Create an IVL object that contains the full adjacency of $A+A^T$ by calling the InpMtx_fullAdjacency() method.

6. Create a `Graph` object using the `Graph_init2()` method and the `IVL` object as an input argument.

A similar functionality exists for creating a `Graph` object from a linear combination of two `InpMtx` objects that contains the matrices A and B. The `InpMtx_fullAdjacency2()` method returns an `IVL` object with the full adjacency of $(A+B)+(A+B)^T$. These two methods are called by the `DPencil_fullAdjacency()` methods to return the full adjacency of a matrix pencil.

Here is a common sequence of events to use this object when we want to assemble the entries of a sparse matrix.

1. Create an instance of a `InpMtx` object using the `InpMtx_new()` method.
2. Initialize the `InpMtx` object using the `InpMtx_init()` method; set the input mode to real or complex entries, maximum number of entries for the workspace, and the number of vectors. (The latter two quantities may be zero, for the object resizes its storage as required.)
3. Call the method `InpMtx_changeCoordType()` to set the coordinate type to rows.
4. Load data into the object using one or more of the five input methods: `InpMtx_inputEntry()`, `InpMtx_inputRow()`, `InpMtx_inputColumn()`, `InpMtx_inputMatrix()` and `InpMtx_inputTriples()` methods. Each time the workspace fills up, the raw data is sorted and compressed and then the workspace is resized. If the input data overlaps, e.g., elemental matrices are being assembled, it would be efficient to have sufficient elbow room to minimize the number of sorts and compressions. In this case, a tight upper bound on the necessary storage is the sum of the sizes of the elemental matrices. The entries are assembled by a call to `InpMtx_changeStorageMode()`.

The `InpMtx` object is now ready to be permuted, take part in a matrix-vector multiply, become part of a `Pencil` matrix pencil object, or serve as input to a numeric factorization.

NOTE: to improve performance we have changed the `InpMtx_fullAdjacency()` method. The `InpMtx` object must be in the chevron coordinate type and have its storage mode be by vectors. Previously, this was done if necessary inside the method.

32.1 Data Structure

The `InpMtx` structure has the following fields.

- `int coordType` : coordinate type. The following types are supported.
  - `INPMTX_BY_ROWS` — row triples, the coordinates for $a_{i,j}$ is (i,j).
  - `INPMTX_BY_COLUMNS` — column triples, the coordinates for $a_{i,j}$ is (j,i).
  - `INPMTX_BY_CHEVRONS` — chevron triples, the coordinates for $a_{i,j}$ is $(\min(i,j), j-i)$. (Chevron j contains $a_{j,j}$, $a_{j,k} \neq 0$ and $a_{k,j} \neq 0$ for $k > j$.)
  - `INPMTX_CUSTOM` — custom coordinates.
- `int storageMode` : mode of storage
  - `INPMTX_RAW_DATA` — data is raw pairs or triples, two coordinates and (optionally) one or two double precision values.
  - `INPMTX_SORTED` — data is sorted and distinct triples, the primary key is the first coordinate, the secondary key is the second coordinate.

- INPMTX_BY_VECTORS — data is sorted and distinct vectors. All entries in a vector share something in common. For example, when coordType is INPMTX_BY_ROWS, INPMTX_BY_COLUMNS or INPMTX_BY_CHEVRONS, row vectors, column vectors, or chevron vectors are stored, respectively. When coordType is INPMTX_CUSTOM, a custom type, entries in the same vector have something in common but it need not be a common row, column or chevron coordinate.

- int inputMode : mode of data input
  - INPMTX_INDICES_ONLY — only indices are stored, not entries.
  - SPOOLES_REAL — indices and real entries are stored.
  - SPOOLES_COMPLEX — indices and complex entries are stored.
- int maxnent – present maximum number of entries in the object. This quantity is initialized by the InpMtx_init() method, but will be changed as the object resizes itself as necessary.
- int nent – present number of entries in the object. This quantity changes as data is input or when the raw triples are sorted and compressed.
- double resizeMultiple – governs how the workspace grows as necessary. The default value is 1.25.
- IV ivec1IV – an IV vector object of size mxnent that holds first coordinates.
- IV ivec2IV – an IV vector object of size mxnent that holds second coordinates.
- DV dvecDV – a DV vector object of size mxnent that holds double precision entries. Used only when inputMode is SPOOLES_REAL or SPOOLES_COMPLEX.
- int maxnvector – present maximum number of vectors. This quantity is initialized by the InpMtx_init() method, but will be changed as the object resizes itself as necessary. Used only when storageMode is INPMTX_BY_VECTORS.
- int nvector – present number of vectors. Used only when storageMode is INPMTX_BY_VECTORS.
- IV vecidsIV – an IV vector object of size nvector to hold the id of each vector. Used only when storageMode is INPMTX_BY_VECTORS.
- IV sizesIV – an IV vector object of size nvector to hold the size of each vector. Used only when storageMode is INPMTX_BY_VECTORS.
- IV offsetsIV – an IV vector object of size nvector to hold the offset of each vector into the ivec1IV, ivec2IV and dvecDV vector objects. Used only when storageMode is INPMTX_BY_VECTORS.

One can query the attributes of the object with the following macros.

- INPMTX_IS_BY_ROWS(mtx) returns 1 if the entries are stored by rows, and 0 otherwise.
- INPMTX_IS_BY_COLUMNS(mtx) returns 1 if the entries are stored by columns, and 0 otherwise.
- INPMTX_IS_BY_CHEVRONS(mtx) returns 1 if the entries are stored by chevrons, and 0 otherwise.
- INPMTX_IS_BY_CUSTOM(mtx) returns 1 if the entries are stored by some custom coordinate, and 0 otherwise.
- INPMTX_IS_RAW_DATA(mtx) returns 1 if the entries are stored as unsorted pairs or triples, and 0 otherwise.
- INPMTX_IS_SORTED(mtx) returns 1 if the entries are stored as sorted pairs or triples, and 0 otherwise.

- INPMTX_IS_BY_VECTORS(mtx) returns 1 if the entries are stored as vectors, and 0 otherwise.
- INPMTX_IS_INDICES_ONLY(mtx) returns 1 if the entries are not stored, and 0 otherwise.
- INPMTX_IS_REAL_ENTRIES(mtx) returns 1 if the entries are real, and 0 otherwise.
- INPMTX_IS_COMPLEX_ENTRIES(mtx) returns 1 if the entries are complex, and 0 otherwise.

32.2 Prototypes and descriptions of InpMtx methods

This section contains brief descriptions including prototypes of all methods that belong to the InpMtx object.

32.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `InpMtx * InpMtx_new ( void ) ;`

   This method simply allocates storage for the InpMtx structure and then sets the default fields by a call to InpMtx_setDefaultFields().

2. `void InpMtx_setDefaultFields ( InpMtx *inpmtx ) ;`

   This method sets the structure's fields to default values: coordType = INPMTX_BY_ROWS, storageMode = INPMTX_RAW_DATA, inputMode = SPOOLES_REAL, resizeMultiple = 1.25, and maxnent = nent = maxnvector = nvector = 0. The IV and DV objects have their fields set to their default values via calls to IV_setDefaultFields() and DV_setDefaultFields().

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

3. `void InpMtx_clearData ( InpMtx *inpmtx ) ;`

   This method releases all storage held by the object.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

4. `void InpMtx_free ( InpMtx *inpmtx ) ;`

   This method releases all storage held by the object via a call to InpMtx_clearData(), then free'd the storage for the object.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

32.2.2 Instance Methods

1. `int InpMtx_coordType ( InpMtx *inpmtx ) ;`

   This method returns the coordinate type.

   - INPMTX_NO_TYPE – none specified
   - INPMTX_BY_ROWS – storage by row triples
   - INPMTX_BY_COLUMNS – storage by column triples
   - INPMTX_BY_CHEVRONS – storage by chevron triples
   - INPMTX_CUSTOM – custom type

Error checking: If inpmtx is NULL, an error message is printed and the program exits.

2. `int InpMtx_storageMode ( InpMtx *inpmtx ) ;`

   This method returns the storage mode.

   - INPMTX_NO_MODE – none specified
   - INPMTX_RAW_DATA – raw triples
   - INPMTX_SORTED – sorted and distinct triples
   - INPMTX_BY_VECTORS – vectors by the first coordinate

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

3. `int InpMtx_inputMode ( InpMtx *inpmtx ) ;`

   This method returns the input mode.

   - INPMTX_INDICES_ONLY – indices only
   - SPOOLES_REAL – indices and real entries
   - SPOOLES_COMPLEX – indices and complex entries

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

4. `int InpMtx_maxnent ( InpMtx *inpmtx ) ;`

   This method returns the maximum number of entries.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

5. `int InpMtx_nent ( InpMtx *inpmtx ) ;`

   This method returns the present number of entries.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

6. `int InpMtx_maxnvector ( InpMtx *inpmtx ) ;`

   This method returns the maximum number of vectors.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

7. `int InpMtx_nvector ( InpMtx *inpmtx ) ;`

   This method returns the present number of vectors.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

8. `double InpMtx_resizeMultiple ( InpMtx *inpmtx ) ;`

   This method returns the present resize multiple for the storage of entries.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

9. `int * InpMtx_ivec1 ( InpMtx *inpmtx ) ;`

   This method returns the base address of the ivec1[] vector.

   Error checking: If inpmtx is NULL, an error message is printed and the program exits.

10. `int * InpMtx_ivec2 ( InpMtx *inpmtx ) ;`

    This method returns the base address of the ivec2[] vector.

    Error checking: If inpmtx is NULL, an error message is printed and the program exits.

11. `double * InpMtx_dvec ( InpMtx *inpmtx ) ;`

    This method returns the base address of the `dvec[]` vector.

    Error checking: If `inpmtx` is `NULL`, an error message is printed and the program exits.

12. `int * InpMtx_vecids ( InpMtx *inpmtx ) ;`

    This method returns the base address of the `vecids[]` vector.

    Error checking: If `inpmtx` is `NULL`, an error message is printed and the program exits.

13. `int * InpMtx_sizes ( InpMtx *inpmtx ) ;`

    This method returns the base address of the `sizes[]` vector.

    Error checking: If `inpmtx` is `NULL`, an error message is printed and the program exits.

14. `int * InpMtx_offsets ( InpMtx *inpmtx ) ;`

    This method returns the base address of the `offsets[]` vector.

    Error checking: If `inpmtx` is `NULL`, an error message is printed and the program exits.

15. ```
    void InpMtx_vector ( InpMtx *inpmtx, int id, int *pnent, int **pindices ) ;
    void InpMtx_realVector ( InpMtx *inpmtx, int id, int *pnent,
                             int **pindices, double **pentries ) ;
    void InpMtx_complexVector ( InpMtx *inpmtx, int id, int *pnent,
                                int **pindices, double **pentries ) ;
    ```

    This methods fills `*pnent` with the number of entries in vector `id` and sets `*pindices` to the base address of the indices. When the object stores real or complex matrix entries, the methods sets `*pentries` to the base address of the entries.

    Error checking: If `inpmtx` is `NULL`, or if `storageMode` $\neq$ `INPMTX_BY_VECTORS`, or if `id` is out of range, or if `pnent`, `pindices` or `pentries` is `NULL`, an error message is printed and the program exits.

16. ```
    int InpMtx_range ( InpMtx *inpmtx, int *pmincol, int *pmaxcol,
                                       int *pminrow, int *pmaxrow ) ;
    ```

    This method computes and returns the minimum and maximum rows and columns in the matrix. If `pmincol` is not `NULL`, on return `*pmincol` is filled with the minimum column id. If `pmaxcol` is not `NULL`, on return `*pmaxcol` is filled with the maximum column id. If `pminrow` is not `NULL`, on return `*pminrow` is filled with the minimum row id. If `pmaxrow` is not `NULL`, on return `*pmaxrow` is filled with the maximum row id.

    Return codes:

1	normal return	-2	no entries in the matrix
-1	`mtx` is `NULL`	-3	invalid coordinate type

17. `void InpMtx_setMaxnent ( InpMtx *inpmtx, int newmaxnent ) ;`

    This method sets the maxinum number of entries in the indices and entries vectors.

    Error checking: If `inpmtx` is `NULL`, or if `newmaxnent` < 0, an error message is printed and the program exits.

18. `void InpMtx_setNent ( InpMtx *inpmtx, int newnent ) ;`

    This method sets the present number of entries in the indices and entries vectors.

    Error checking: If `inpmtx` is `NULL`, or if `newnent` < 0, an error message is printed and the program exits.

19. `void InpMtx_setMaxnvector ( InpMtx *inpmtx, int newmaxnvector ) ;`

    This method sets the maxinum number of vectors.

    Error checking: If `inpmtx` is `NULL`, or if `newmaxnvector` < 0, an error message is printed and the program exits.

20. `void InpMtx_setNvector ( InpMtx *inpmtx, int newnvector ) ;`

    This method sets the present number of vectors.

    Error checking: If `inpmtx` is `NULL`, or if `newnvector` < 0, an error message is printed and the program exits.

21. `void InpMtx_setResizeMultiple ( InpMtx *inpmtx, double resizeMultiple ) ;`

    This method sets the present number of vectors.

    Error checking: If `inpmtx` is `NULL`, or if `resizeMultiple` < 0, an error message is printed and the program exits.

22. `void InpMtx_setCoordType ( InpMtx *inpmtx, int type ) ;`

    This method sets a custom coordinate type, so type must be greater than or equal to `4`. To change from one of the three supported types to another, use `InpMtx_changeCoordType()`.

    Error checking: If `inpmtx` is `NULL`, or if `coordType` $<= 3$, an error message is printed and the program exits.

32.2.3 Methods to initialize and change state

1. ```
   void InpMtx_init ( InpMtx *inpmtx, int coordType, int inputMode,
                      int maxnent, int maxnvector) ;
   ```

   This is the initializer method that should be called immediately after `InpMtx_new()`. It first clears any previous data with a call to `InpMtx_clearData()`. The `coordType` and `inputMode` fields are set. If `maxnent > 0` then the `ivec1IV` and `ivec2IV` objects are initialized to have size `maxnent`. If `maxnent > 0` and `inputMode = SPOOLES_REAL` or `inputMode = SPOOLES_COMPLEX` , then the `dvecDV` object is initialized to have size `maxnent`. If `maxnvector > 0` then the `sizesIV` and `offsetsIV` objects are initialized to have size `maxnvector`.

   Error checking: If `inpmtx` is `NULL` or if `coordType` or `inputMode` is invalid, or if `maxnent` or `maxnvector` are less than zero, an error message is printed and the program exits.

2. `void InpMtx_changeCoordType ( InpMtx *inpmtx, int newType ) ;`

   This method changes the coordinate type. If `coordType` = `newType`, the program returns. If `coordType` $\geq$ 4, then the triples are held in some unknown custom type and cannot be translated, so an error message is printed and the program exits. If `newType` ≥ 4, then some custom coordinate type is now present; the `coordType` field is set and the method returns. If `newType` is one of `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, a translation is made from the old coordinate type to the new type.

   Error checking: If `inpmtx` is `NULL` or `newType` is invalid, an error message is printed and the program exits.

3. `void InpMtx_changeStorageMode ( InpMtx *inpmtx, int newMode ) ;`

   If `storageMode` = `newMode`, the method returns. Otherwise, a translation between the three valid modes is made by calling `InpMtx_sortAndCompress()` and `InpMtx_convertToVectors()`, as appropriate.

   Error checking: If `inpmtx` is `NULL` or `newMode` is invalid, an error message is printed and the program exits.
   ```

32.2.4 Input methods

1. `void InpMtx_inputEntry ( InpMtx *inpmtx, int row, int col ) ;`
   `void InpMtx_inputRealEntry ( InpMtx *inpmtx, int row, int col, double value ) ;`
   `void InpMtx_inputComplexEntry ( InpMtx *inpmtx, int row, int col, double real, double imag ) ;`

   This method places a single entry into the matrix object. The coordinate type of the object must be `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`. The triple is formed and inserted into the vectors, which are resized if necessary.

   Error checking: If `inpmtx` is `NULL` or `row` or `col` are negative, an error message is printed and the program exits.

2. `void InpMtx_inputRow ( InpMtx *inpmtx, int row, int rowsize, int rowind[] ) ;`
   `void InpMtx_inputRealRow ( InpMtx *inpmtx, int row, int rowsize, int rowind[], double rowent[] ) ;`
   `void InpMtx_inputComplexRow ( InpMtx *inpmtx, int row, int rowsize, int rowind[], double rowent[] ) ;`

   This method places a row or row fragment into the matrix object. The coordinate type of the object must be `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`. The individual entries of the row are placed into the vector storage as triples, and the vectors are resized if necessary.

   Error checking: If `inpmtx` is `NULL`, or `row` or `rowsize` are negative, or `rowind` or `rowent` are `NULL`, an error message is printed and the program exits.

3. `void InpMtx_inputColumn ( InpMtx *inpmtx, int col, int colsize, int colind[] ) ;`
   `void InpMtx_inputRealColumn ( InpMtx *inpmtx, int col, int colsize, int colind[], double colent[] ) ;`
   `void InpMtx_inputComplexColumn ( InpMtx *inpmtx, int col, int colsize, int colind[], double colent[] ) ;`

   This method places a column or column fragment into the matrix object. The coordinate type of the object must be `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`. The individual entries of the column are placed into the vector storage as triples, and the vectors are resized if necessary.

   Error checking: If `inpmtx` is `NULL`, or `col` or `colsize` are negative, or `colind` or `colent` are `NULL`, an error message is printed and the program exits.

4. `void InpMtx_inputChevron ( InpMtx *inpmtx, int chv, int chvsize, int chvind[] ) ;`
   `void InpMtx_inputRealChevron ( InpMtx *inpmtx, int chv, int chvsize, int chvind[], double chvent[] ) ;`
   `void InpMtx_inputComplexChevron ( InpMtx *inpmtx, int chv, int chvsize, int chvind[], double chvent[] ) ;`

   This method places a chevron or chevron fragment into the matrix object. The coordinate type of the object must be `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`. The individual entries of the chevron are placed into the vector storage as triples, and the vectors are resized if necessary.

   Error checking: If `inpmtx` is `NULL`, or `chv` or `chvsize` are negative, or `chvind` or `chvent` are `NULL`, an error message is printed and the program exits.

5. `void InpMtx_inputMatrix ( InpMtx *inpmtx, int nrow, int col, int rowstride, int colstride, int rowind[], int colind[] ) ;`
   `void InpMtx_inputRealMatrix ( InpMtx *inpmtx, int nrow, int col, int rowstride, int colstride, int rowind[], int colind[], double mtxent[] ) ;`
   `void InpMtx_inputComplexMatrix ( InpMtx *inpmtx, int nrow, int col,`

```
int rowstride, int colstride, int rowind[], int colind[], double mtxent[] ) ;
```

This method places a dense submatrix into the matrix object. The coordinate type of the object must be INPMTX_BY_ROWS, INPMTX_BY_COLUMNS or INPMTX_BY_CHEVRONS. The individual entries of the matrix are placed into the vector storage as triples, and the vectors are resized if necessary.

Error checking: If inpmtx is NULL, or col or row are negative, or rowstride or colstride are less than 1, or rowind, colind or mtxent are NULL, an error message is printed and the program exits.

6. ```
   void InpMtx_inputTriples ( InpMtx *inpmtx, int ntriples,
                  int rowids[], int colids[]) ;
   void InpMtx_inputRealTriples ( InpMtx *inpmtx, int ntriples,
                  int rowids[], int colids[], double entries[] ) ;
   void InpMtx_inputComplexTriples ( InpMtx *inpmtx, int ntriples,
                  int rowids[], int colids[], double entries[] ) ;
   ```

   This method places a vector of (row,column,entry) triples into the matrix object. The coordinate type of the object must be INPMTX_BY_ROWS, INPMTX_BY_COLUMNS or INPMTX_BY_CHEVRONS.

   Error checking: If inpmtx, rowids, colids is NULL, or ntriples are negative, or if inputMode = 2 and entries is NULL, an error message is printed and the program exits.

32.2.5 Permutation, map and support methods

These methods find the *support* of a matrix, map the indices from one numbering to another, and permute the rows and/or columns of the matrix.

1. ```
   void InpMtx_supportNonsym ( InpMtx *A, IV *rowsupIV, IV *colsupIV ) ;
   void InpMtx_supportNonsymT ( InpMtx *A, IV *rowsupIV, IV *colsupIV ) ;
   void InpMtx_supportNonsymH ( InpMtx *A, IV *rowsupIV, IV *colsupIV ) ;
   ```

   These methods are used to set up sparse matrix-matrix multiplies of the form $Y := Y + \alpha AX$, $Y := Y + \alpha A^T X$ or $Y := Y + \alpha A^H X$, where A is a nonsymmetric matrix. These methods fill rowsupIV with the rows of Y that will be updated, and colsupIV with the rows of X that will be accessed. In a distributed environment, A, X and Y will be distributed, and A will contain only part of the larger global matrix A. Finding the row an column support enables one to construct local data structures for X and the product αAX.

   Error checking: If A, rowsupIV or colsupIV is NULL, an error message is printed and the program exits.

2. ```
   void InpMtx_supportSym ( InpMtx *A, IV *supIV ) ;
   void InpMtx_supportSymH ( InpMtx *A, IV *supIV ) ;
   ```

   These methods are used to set up sparse matrix-matrix multiplies of the form $Y := Y + \alpha AX$ where A is a symmetric or Hermitian matrix. These methods fill supIV with the rows of Y that will be updated. Since A has symmetric nonzero structure, the rows of Y that will be updated are exactly the same as the rows of X that will be accessed. In a distributed environment, A, X and Y will be distributed, and A will contain only part of the larger global matrix A. Finding the row an column support enables one to construct local data structures for X and the product αAX.

   Error checking: If A or supIV is NULL, an error message is printed and the program exits.

3. ```
   void InpMtx_mapEntries ( InpMtx *A, IV *rowmapIV, IV *colmapIV ) ;
   ```

   These methods are used to map a matrix from one numbering system to another. The primary use of this method is to map a part of a distributed matrix between the global and local numberings.

   Error checking: If A, rowmapIV or colmapIV is NULL, an error message is printed and the program exits.

4. `void InpMtx_permute ( InpMtx *inpmtx, int rowOldToNew[], int colOldToNew[] ) ;`

   This method permutes the rows and or columns of the matrix. If `rowOldToNew` and `colOldToNew` are both `NULL`, or if there are no entries in the matrix, the method returns. Note, either `rowOldToNew` or `colOldToNew` can be `NULL`. If `coordType == INPMTX_BY_CHEVRONS`, then the coordinates are changed to row coordinates. The coordinates are then mapped to their new values. The `storageMode` is set to `1`, (raw triples).

   Error checking: If `inpmtx` is `NULL`, an error message is printed and the program exits.

32.2.6 Matrix-matrix multiply methods

There are four families of matrix-vector and matrix-matrix multiply methods. The `InpMtx_*_mmm*()` methods compute

$$Y := Y + \alpha A X, \qquad Y := Y + \alpha A^T X \qquad \text{and} \qquad Y := Y + \alpha A^H X,$$

where A is an `InpMtx` object, and X and Y are column major `DenseMtx` objects. The `InpMtx_*_mmmVector*()` methods compute

$$y := y + \alpha A x, \qquad y := y + \alpha A^T x \qquad \text{and} \qquad y := y + \alpha A^H x,$$

where A is an `InpMtx` object, and x and y are vectors. The `InpMtx_*_gmmm*()` methods compute

$$Y := \beta Y + \alpha A X, \qquad Y := \beta Y + \alpha A^T X \qquad \text{and} \qquad Y := \beta Y + \alpha A^H X,$$

where A is an `InpMtx` object, and X and Y are column major `DenseMtx` objects. The `InpMtx_*_gmvm*()` methods compute

$$y := \beta y + \alpha A x, \qquad y := \beta y + \alpha A^T x \qquad \text{and} \qquad y := \beta y + \alpha A^H x,$$

where A is an `InpMtx` object, and x and y are `double` vectors. The code notices if α and/or β are zero or 1 and takes special action.

1.
```
void InpMtx_nonsym_mmm ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;
void InpMtx_sym_mmm ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;
void InpMtx_herm_mmm ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;
void InpMtx_nonsym_mmm_T ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;
void InpMtx_nonsym_mmm_H ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;
```

   These five methods perform the following computations.

`InpMtx_nonsym_mmm()`	$Y := Y + \alpha A X$	nonsymmetric	real or complex
`InpMtx_sym_mmm()`	$Y := Y + \alpha A X$	symmetric	real or complex
`InpMtx_herm_mmm()`	$Y := Y + \alpha A X$	Hermitian	complex
`InpMtx_nonsym_mmm_T()`	$Y := Y + \alpha A^T X$	nonsymmetric	real or complex
`InpMtx_nonsym_mmm_H()`	$Y := Y + \alpha A^H X$	nonsymmetric	complex

   `A`, `X` and `Y` must all be real or all be complex. When `A` is real, then α = `alpha[0]`. When `A` is complex, then α = `alpha[0]` + i* `alpha[1]`. The values of α must be loaded into an array of length 1 or 2.

   Error checking: If `A`, `Y` or `X` are `NULL`, or if `coordType` is not `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, or if `storageMode` is not one of `INPMTX_RAW_DATA`, `INPMTX_SORTED` or `INPMTX_BY_VECTORS`, or if `inputMode` is not `SPOOLES_REAL` or `SPOOLES_COMPLEX`, an error message is printed and the program exits.

2. `void InpMtx_nonsym_mmmVector ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;`
   `void InpMtx_sym_mmmVector ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;`
   `void InpMtx_herm_mmmVector ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;`
   `void InpMtx_nonsym_mmmVector_T ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;`
   `void InpMtx_nonsym_mmmVector_H ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X ) ;`

   These five methods perform the following computations.

`InpMtx_nonsym_mmm()`	$y := y + \alpha A x$	nonsymmetric	real or complex
`InpMtx_sym_mmm()`	$y := y + \alpha A x$	symmetric	real or complex
`InpMtx_herm_mmm()`	$y := y + \alpha A x$	Hermitian	complex
`InpMtx_nonsym_mmm_T()`	$y := y + \alpha A^T x$	nonsymmetric	real or complex
`InpMtx_nonsym_mmm_H()`	$y := y + \alpha A^H x$	nonsymmetric	complex

   `A`, `x` and `y` must all be real or all be complex. When `A` is real, then α = `alpha[0]`. When `A` is complex, then α = `alpha[0]` + i* `alpha[1]`. The values of α must be loaded into an array of length 1 or 2.

   Error checking: If `A`, `x` or `x` are `NULL`, or if `coordType` is not `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, or if `storageMode` is not one of `INPMTX_RAW_DATA`, `INPMTX_SORTED` or `INPMTX_BY_VECTORS`, or if `inputMode` is not `SPOOLES_REAL` or `SPOOLES_COMPLEX`, an error message is printed and the program exits.

3. ```
   int InpMtx_nonsym_gmmm ( InpMtx *A, double beta[], DenseMtx *Y,
                            double alpha[], DenseMtx *X ) ;
   int InpMtx_sym_gmmm ( InpMtx *A, double beta[], DenseMtx *Y,
                         double alpha[], DenseMtx *X ) ;
   int InpMtx_herm_gmmm ( InpMtx *A, double beta[], DenseMtx *Y,
                          double alpha[], DenseMtx *X ) ;
   int InpMtx_nonsym_gmmm_T ( InpMtx *A, double beta[], DenseMtx *Y,
                              double alpha[], DenseMtx *X ) ;
   int InpMtx_nonsym_gmmm_H ( InpMtx *A, double beta[], DenseMtx *Y,
                              double alpha[], DenseMtx *X ) ;
   ```

   These five methods perform the following computations.

`InpMtx_nonsym_gmmm()`	$Y := \beta Y + \alpha A X$	nonsymmetric	real or complex
`InpMtx_sym_gmmm()`	$Y := \beta Y + \alpha A X$	symmetric	real or complex
`InpMtx_herm_gmmm()`	$Y := \beta Y + \alpha A X$	Hermitian	complex
`InpMtx_nonsym_gmmm_T()`	$Y := \beta Y + \alpha A^T X$	nonsymmetric	real or complex
`InpMtx_nonsym_gmmm_H()`	$Y := \beta Y + \alpha A^H X$	nonsymmetric	complex

   `A`, `X` and `Y` must all be real or all be complex. When `A` is real, then β = `beta[0]` and α = `alpha[0]`. When `A` is complex, then β = `beta[0]` + i*`beta[1]` and α = `alpha[0]` + i*`alpha[1]`. The values of β and α must be loaded into an array of length 1 or 2.

   Return codes:

1	normal return	-8	entries of `Y` are `NULL`
-1	`A` is `NULL`	-9	`alpha` is `NULL`
-2	type of `A` is invalid	-10	`X` is `NULL`
-3	indices of entries of `A` are `NULL`	-11	type of `X` is invalid
-4	`beta` is `NULL`	-12	bad dimensions and strides for `X`
-5	`Y` is `NULL`	-13	entries of `X` are `NULL`
-6	type of `Y` is invalid	-14	types of `A`, `X` and `Y` are not identical
-7	bad dimensions and strides for `Y`	-15	number of columns in `X` and `Y` are not equal
   ```

4. `int InpMtx_nonsym_gmvm ( InpMtx *A, double beta[], int ny, double y[], double alpha[], int nx, double x[] ) ;`
   `int InpMtx_sym_gmvm ( InpMtx *A, double beta[], int ny, double y[], double alpha[], int nx, double x[] ) ;`
   `int InpMtx_herm_gmvm ( InpMtx *A, double beta[], int ny, double y[], double alpha[], int nx, double x[] ) ;`
   `int InpMtx_nonsym_gmvm_T ( InpMtx *A, double beta[], int ny, double y[], double alpha[], int nx, double x[] ) ;`
   `int InpMtx_nonsym_gmvm_H ( InpMtx *A, double beta[], int ny, double y[], double alpha[], int nx, double x[] ) ;`

   These five methods perform the following computations.

`InpMtx_nonsym_gmvm()`	$y := \beta y + \alpha A x$	nonsymmetric	real or complex
`InpMtx_sym_gmvm()`	$y := \beta y + \alpha A x$	symmetric	real or complex
`InpMtx_herm_gmvm()`	$y := \beta y + \alpha A x$	Hermitian	complex
`InpMtx_nonsym_gmvm_T()`	$y := \beta y + \alpha A^T x$	nonsymmetric	real or complex
`InpMtx_nonsym_gmvm_H()`	$y := \beta y + \alpha A^H x$	nonsymmetric	complex

   When `A` is real, then β = `beta[0]` and α = `alpha[0]`. When `A` is complex, then β = `beta[0]` + i*`beta[1]` and α = `alpha[0]` + i*`alpha[1]`. The values of β and α must be loaded into an array of length 1 or 2.

   Return codes:

1	normal return	-5	`ny` ≤ 0
-1	`A` is `NULL`	-6	`y` is `NULL`
-2	type of `A` is invalid	-7	`alpha` is `NULL`
-3	indices of entries of `A` are `NULL`	-8	`nx` ≤ 0
-4	`beta` is `NULL`	-9	`x` is `NULL`

32.2.7 Graph construction methods

Often we need to construct a graph object from a matrix, e.g., when we need to find an ordering of the rows and columns. We don't construct a `Graph` object directly, but create a full adjacency structure that is stored in an `IVL` object, a lower level object than the `Graph` object.

1. `IVL * InpMtx_fullAdjacency ( InpMtx *inpmtxA ) ;`

   This method creates and returns an `IVL` object that holds the full adjacency structure of $A + A^T$, where `inpmtxA` contains the entries in A.

   Error checking: If `inpmtxA` is `NULL`, or if the coordinate type is not `INPMTX_BY_ROWS` or `INPMTX_BY_COLUMNS`, or if the storage mode is not `INPMTX_BY_VECTORS`, an error message is printed and the program exits.

2. `IVL * InpMtx_fullAdjacency2 ( InpMtx *inpmtxA, InpMtx *inpmtxB ) ;`

   This method creates and returns an `IVL` object that holds the full adjacency structure of $(A + B) + (A + B)^T$, where `inpmtxA` contains the entries in A and `inpmtxB` contains the entries in B.

   Error checking: If `inpmtxA` is `NULL`, or if the coordinate type is not `INPMTX_BY_ROWS` or `INPMTX_BY_COLUMNS`, or if the storage mode is not `INPMTX_BY_VECTORS`, an error message is printed and the program exits.

3. `IVL * InpMtx_adjForATA ( InpMtx *inpmtxA ) ;`

   This method creates and returns an `IVL` object that holds the full adjacency structure of $A^T A$, where `inpmtxA` contains the entries in A.

   Error checking: If `inpmtxA` is `NULL`, an error message is printed and the program exits.

32.2.8 Submatrix extraction method

1. `int InpMtx_initFromSubmatrix ( InpMtx *B, InpMtx *A, IV *BrowsIV, IV *BcolsIV, int symmetryflag, int msglvl, FILE *msgFile ) ;`

   This method fills `B` with the submatrix formed from the rows and columns of `A` found in `BrowsIV` and `BcolsIV`. The row and column indices in `B` are local with respect to `BrowsIV` and `BcolsIV`.

   When `symmetryflag` is `SPOOLES_SYMMETRIC` or `SPOOLES_HERMITIAN`, then we assume that when $i \neq j$, $A_{i,j}$ or $A_{j,i}$ is stored, but not both. (A could be stored by rows of its upper triangle, or by columns of its lower triangle, or a mixture.) In this case, if `BrowsIV` and `BcolsIV` are identical, then just the upper triangular part of `B` is stored. Otherwise `B` contains all entries of A for rows in `rowsIV` and columns in `colsIV`.

   Return codes:

1	normal return	-5	invalid input mode for `A`
-1	`B` is `NULL`	-6	invalid coordinate type for `A`
-2	`BcolsIV` is `NULL`	-7	invalid `symmetryflag`
-3	`BrowsIV` is `NULL`	-8	Hermitian `symmetryflag` but not complex
-4	`A` is `NULL`	-9	`msglvl` > 0 and `msgFile` is `NULL`

32.2.9 Utility methods

1. `void InpMtx_sortAndCompress ( InpMtx *inpmtx ) ;`

   This method sorts the triples first by their primary key and next by their secondary key. At this point any two triples with identical first and second coordinates lie in consecutive locations, so it is easy to add all entries together that are associated with a triple and thus compress the vectors.

   Error checking: If `inpmtx` is `NULL`, or if `storageMode` is not 1, an error message is printed and the program exits.

2. `void InpMtx_convertToVectors ( InpMtx *inpmtx ) ;`

   This method fills the `sizes[]` and `offsets[]` arrays to generate a set of vectors of triples whose first coordinate is identical. The method requires that `storageMode = INPMTX_SORTED`, i.e., that the triples have been sorted and compressed. The sizes of the two arrays are changed as necessary.

   Error checking: If `inpmtx` is `NULL`, or if `storageMode` is not 2, an error message is printed and the program exits.

3. `void InpMtx_dropOffDiagonalEntries ( InpMtx *inpmtx ) ;`
   `void InpMtx_dropLowerTriangle ( InpMtx *inpmtx ) ;`
   `void InpMtx_dropUpperTriangle ( InpMtx *inpmtx ) ;`

   These methods purge entries based on structure.

   Error checking: If `inpmtx` is `NULL`, or if `coordType` is not `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, an error message is printed and the program exits.

4. `void InpMtx_mapToLowerTriangle  ( InpMtx *inpmtx ) ;`
   `void InpMtx_mapToUpperTriangle  ( InpMtx *inpmtx ) ;`
   `void InpMtx_mapToUpperTriangleH ( InpMtx *inpmtx ) ;`

   If the `InpMtx` object holds only the lower or upper triangle of a matrix (as when the matrix is symmetric or Hermitian), and is then permuted, it is not likely that the permuted object will only have entries in the lower or upper triangle. The first method moves $a_{i,j}$ for $i < j$ to $a_{j,i}$. The second method moves $a_{i,j}$ for $i > j$ to $a_{j,i}$, (If the matrix is Hermitian, the sign of the imaginary part of an entry is dealt with in the correct fashion.) In other words, using these methods will restore the lower or upper triangular structure after a permutation.

Error checking: If `inpmtx` is `NULL`, or if `coordType` is invalid, an error message is printed and the program exits.

5. `void InpMtx_log10profile ( InpMtx *inpmtx, int npts, DV *xDV, DV *yDV, double tausmall, double taubig, int *pnzero, int *pnsmall, int *pnbig ) ;`

   This method fills the `xDV` and `yDV` objects with with an approximate density profile of the magnitudes of the entries in the matrix. Only values whose $\log 10(a_{i,j})$ is in the range `[tausmall, taubig]` contribute to the profile. The range is divided up into `npts` buckets. The `x` value is the $\log 10$ of a average magnitude of a bucket, and the `y` value is the number of entries found in that bucket. On return, `*pnzero` returns the number of zero entries in the matrix, `*pnsmall` returns the number of entries whose $\log 10$ magnitude is smaller than `tausmall`, and `*pnbig` returns the number of entries whose $\log 10$ magnitude is larger than `taubig`. The `DVL_log10profile()` method is used to find the profile.

   Error checking: If `inpmtx`, `xDV`, `yDV`, `pnzero`, `pnsmall` or `pnbig` is `NULL`, or if `inputMode` is not `SPOOLES_REAL` or `SPOOLES_COMPLEX`, or if `npts`, `taubig` or `tausmall` ≤ 0, or if `tausmall > taubig`, an error message is printed and the program exits.

6. `void InpMtx_checksums ( InpMtx *inpmtx, double sums[] ) ;`

   This method fills `sums[0]` with the sum of the absolute values of the first coordinates, `sums[1]` with the sum of the absolute values of the second coordinates, and if entries are present, it fills `sums[2]` with the sum of the magnitudes of the entries.

   Error checking: If `inpmtx` is `NULL`, or if `inputMode` is not valid, an error message is printed and the program exits.

7. `int InpMtx_randomMatrix ( InpMtx *inpmtx, int inputMode, int coordType, int storageMode, int nrow, int ncol, int symflag, int nonzerodiag, int nitem, int seed ) ;`

   This methods fills `mtx` with random entries. `inputMode` can be indices only, real or complex. `coordType` can be rows, columns or chevrons. `storageMode` can be raw, sorted or vectors. `nrow` and `ncol` must be positive. `symflag` can be symmetric, Hermitian or nonsymmetric. if `nonzerodiag` is `1`, the diagonal of the matrix is filled with nonzeros. `nitem` numbers (or `nitem + min(nrow,ncol)` if `nonzerodiag = 1`) are placed into the matrix. `seed` is used for the random number generator.

   Error checking: If `inpmtx` is `NULL`, `-1` is returned. If `inputMode` is invalid, `-2` is returned. If `coordType` is invalid, `-3` is returned. If `storageMode` is invalid, `-4` is returned. If `nrow` or `ncol` is not positive, `-5` is returned. If `symflag` is invalid, `-5` is returned. If `symflag` is Hermitian but `inputMode` is not complex, `-7` is returned. If `symflag` is symmetric or Hermitian but `nrow` is not equal to `ncol`, `-8` is returned. If `nitem` is not positive, `-9` is returned. Otherwise, `1` is returned.

   Return codes:

1	normal return	-5	`nrow` or `ncol` negative
-1	`inpmtx` is `NULL`	-6	`symflag` is invalid
-2	`inputMode` invalid	-7	`(symflag,inputMode)` invalid
-3	`coordType` invalid	-8	`(symflag,nrow,ncol)` invalid
-4	`storageMode` invalid	-9	`nitem` negative

32.2.10 IO methods

There are the usual eight IO routines. The file structure of a `InpMtx` object is simple: The first entries in the file are `coordType`, `storageMode`, `inputMode`, `nent` and `nvector`. If `nent > 0`, then the `ivec1IV` and `ivec2IV` vectors follow, If `nent > 0` and `inputMode = SPOOLES_REAL` or `SPOOLES_COMPLEX`, the `dvecDV` vector follows. If `storageMode = INPMTX_BY_VECTORS` and `nvector > 0`, the `vecidsIV`, `sizesIV` and `offsetsIV` vectors follow.

1. `int InpMtx_readFromFile ( InpMtx *inpmtx, char *fn ) ;`

   This method reads the object from a formatted or binary file. It tries to open the file and if successful, it then calls `InpMtx_readFromBinaryFile()` or `InpMtx_readFromFormattedFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `inpmtx` or `fn` is `NULL`, or if `fn` is not of the form `*.inpmtxf` (for a formatted file) or `*.inpmtxb` (for a binary file), or if the file cannot be opened, an error message is printed and the method returns zero.

2. `int InpMtx_readFromFormattedFile ( InpMtx *inpmtx, FILE *fp ) ;`

   This method reads in the object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned.

   Error checking: If `inpmtx` or `fp` is `NULL`, an error message is printed and the method returns zero.

3. `int InpMtx_readFromBinaryFile ( InpMtx *inpmtx, FILE *fp ) ;`

   This method reads in the object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned.

   Error checking: If `inpmtx` or `fp` is `NULL`, an error message is printed and the method returns zero.

4. `int InpMtx_writeToFile ( InpMtx *inpmtx, char *fn ) ;`

   This method writes the object to a formatted or binary file. It tries to open the file and if successful, it then calls `InpMtx_writeToBinaryFile()` or `InpMtx_writeToFormattedFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `inpmtx` of `fn` is `NULL`, or if `fn` is not of the form `*.inpmtxf` (for a formatted file) or `*.inpmtxb` (for a binary file), or if the file cannot be opened, an error message is printed and the method returns zero.

5. `int InpMtx_writeToFormattedFile ( InpMtx *inpmtx, FILE *fp ) ;`

   This method writes the object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `inpmtx` or `fp` is `NULL`, an error message is printed and the method returns zero.

6. `int InpMtx_writeToBinaryFile ( InpMtx *inpmtx, FILE *fp ) ;`

   This method writes the object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `inpmtx` or `fp` is `NULL`, an error message is printed and the method returns zero.

7. `int InpMtx_writeForHumanEye ( InpMtx *inpmtx, FILE *fp ) ;`

   This method writes the object to a file suitable for reading by a human. The method `InpMtx_writeStats()` is called to write out the header and statistics. The data is written out in the appropriate way, e.g., if the storage mode is by triples, triples are written out. The value `1` is returned.

   Error checking: If `inpmtx` or `fp` are `NULL`, an error message is printed and zero is returned.

8. `int InpMtx_writeStats ( InpMtx *inpmtx, FILE *fp ) ;`

   This method writes the statistics about the object to a file. human. The value `1` is returned.

   Error checking: If `inpmtx` or `fp` are `NULL`, an error message is printed and zero is returned.

9. `void InpMtx_writeForMatlab ( InpMtx *mtx, char *mtxname, FILE *fp ) ;`

   This method writes out a `InpMtx` object to a file in a Matlab format. A sample line is

   ```
   a(10,5) =  -1.550328201511e-01 +   1.848033378871e+00*i ;
   ```

   for complex matrices, or

   ```
   a(10,5) =  -1.550328201511e-01 ;
   ```

   for real matrices, where mtxname = `"a"`. The matrix indices come from the `rowind[]` and `colind[]` vectors, and are incremented by one to follow the Matlab and FORTRAN convention.

   Error checking: If `mtx`, `mtxname` or `fp` are `NULL`, an error message is printed and zero is returned.

10. `int InpMtx_readFromHBFile ( InpMtx *inpmtx, char *fn ) ;`

    This method reads the object from a Harwell-Boeing file. This method calls `readHB_info()` and `readHB_mat_double()` from the Harwell-Boeing C IO routines from NIST[1], found in the `misc/src/iohb.c` file.

    Error checking: If `inpmtx` or `fn` is `NULL`, or if the file cannot be opened, an error message is printed and the method returns zero.

32.3 Driver programs for the `InpMtx` object

This section contains brief descriptions of the driver programs.

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads and write `InpMtx` files, useful for converting formatted files to binary files and vice versa. One can also read in a `InpMtx` file and print out just the header information (see the `InpMtx_writeStats()` method).

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `InpMtx` object. It must be of the form `*.inpmtxf` or `*.inpmtxb`. The `InpMtx` object is read from the file via the `InpMtx_readFromFile()` method.
   - The `outFile` parameter is the output file for the `InpMtx` object. If `outFile` is `none` then the `InpMtx` object is not written to a file. Otherwise, the `InpMtx_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.inpmtxf`), or a binary file (if `outFile` is of the form `*.inpmtxb`).

2. `testFullAdj msglvl msgFile nvtx nent seed`

   This driver program tests the `InpMtx_fullAdjacency()` method. If first generates a `InpMtx` object filled with random entries of a matrix A and then constructs an `IVL` object that contains the full adjacency structure of $A + A^T$, diagonal edges included.

[1] `http://math.nist.gov/mcsd/Staff/KRemington/harwell_io/harwell_io.html`

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `nvtx` parameter is the number of rows and columns in A.
- The `nent` parameter is an upper bound on the number of entries in A. (Since the locations of the entries are generated via random numbers, there may be duplicate entries.)
- The `seed` parameter is random number seed.

3. `testFullAdj2 msglvl msgFile nvtx nentA nentB seed`

   This driver program tests the `InpMtx_fullAdjacency2()` method. If first generates two `InpMtx` object filled with random entries — one for a matrix A and one for a matrix B. It then constructs an `IVL` object that contains the full adjacency structure of $(A + B) + (A + B)^T$, diagonal edges included.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `nvtx` parameter is the number of rows and columns in A.
   - The `nentA` parameter is an upper bound on the number of entries in A. (Since the locations of the entries are generated via random numbers, there may be duplicate entries.)
   - The `nentB` parameter is an upper bound on the number of entries in B. (Since the locations of the entries are generated via random numbers, there may be duplicate entries.)
   - The `seed` parameter is random number seed.

4. `createGraph msglvl msgFile inFile outFile`

   This driver program reads in `InpMtx` object from the file `inFile` that holds a matrix A. It then creates a `Graph` object for $B = A + A^T$ and writes it to the file `outFile`. Recall, a `Graph` object must be symmetric, so if the `InpMtx` object only holds the lower or upper triangular part of the matrix, the other portion will be added. Also, a `Graph` object has edges of the form `(v,v)`, and if these entries are missing from the `InpMtx` object, they will be added.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `InpMtx` object. It must be of the form `*.inpmtxf` or `*.inpmtxb`. The `InpMtx` object is read from the file via the `InpMtx_readFromFile()` method.
   - The `outFile` parameter is the output file for the `InpMtx` object. If `outFile` is `none` then the `InpMtx` object is not written to a file. Otherwise, the `InpMtx_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.inpmtxf`), or a binary file (if `outFile` is of the form `*.inpmtxb`).

5. `createGraphForATA msglvl msgFile inFile outFile`

   This driver program reads in `InpMtx` object from the file `inFile` that holds a matrix A. It then creates a `Graph` object for $B = A^T A$ and writes it to the file `outFile`.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inFile` parameter is the input file for the `InpMtx` object. It must be of the form `*.inpmtxf` or `*.inpmtxb`. The `InpMtx` object is read from the file via the `InpMtx_readFromFile()` method.
- The `outFile` parameter is the output file for the `InpMtx` object. If `outFile` is `none` then the `InpMtx` object is not written to a file. Otherwise, the `InpMtx_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.inpmtxf`), or a binary file (if `outFile` is of the form `*.inpmtxb`).

6. `adjToGraph msglvl msgFile inAdjacencyFile outGraphFile flag`

   This driver program was used to generate a `type 0 Graph` object (unit weight vertices and edges) from a file that contained the adjacency structure of a matrix in the following form.

   ```
   nvtx nadj
   offsets[nvtx+1]
   indices[nadj]
   ```

   There are `nvtx` vertices in the graph and the adjacency vector has `nadj` entries. It was not known whether the adjacency structure contained `(v,v)` entries or if it was only the upper or lower triangle. Our `Graph` object is symmetric with loops, i.e., `(u,v)` is present if and only if `(v,u)` is present, and `(v,v)` is present.

   This program reads in the adjacency structure, decrements the offsets and indices by one if specified by the flag parameter (our application came from a Fortran code with 1-indexing), then loads the entries into a `InpMtx` object where they are assembled and sorted by rows. The (v, v) entries are loaded, and each vector of the adjacency structure is loaded as both a column and as a row, so in effect we are constructing the graph of $(A + A^T)$. Recall, multiple entries are collapsed during the sort and merge step.

   A `Graph` object is then created using the `Graph_fillFromOffsets()` method using the vectors in the `InpMtx` object. The `Graph` object is then optionally written to a file.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inAdjacencyFile` parameter is the input file for the adjacency structure as defined above. It must be a formatted file.
   - The `outGraphFile` parameter is the output file for the `Graph` object. If `outGraphFile` is `none` then the `Graph` object is not written to a file. Otherwise, the `Graph_writeToFile()` method is called to write the object to a formatted file (if `outGraphFile` is of the form `*.graphf`), or a binary file (if `outGraphFile` is of the form `*.graphb`).
   - The `flag` parameter is used to specify whether the offsets and indices are 0-indexed (as in C) or 1-indexed (as in Fortran). If they are 1-indexed, the offsets and indices are decremented prior to loading into the `InpMtx` object.

7. `weightedAdjToGraph msglvl msgFile inAdjacencyFile outGraphFile flag`

   This driver program was used to generate a `type 1 Graph` object (weighted vertices, unit weight edges) from a file that contained the adjacency structure of a matrix in the following form.

```
nvtx nadj
vwghts[nvtx]
offsets[nvtx+1]
indices[nadj]
```

There are `nvtx` vertices in the graph and the adjacency vector has `nadj` entries. It was not known whether the adjacency structure contained `(v,v)` entries or if it was only the upper or lower triangle. Our `Graph` object is symmetric with loops, i.e., `(u,v)` is present if and only if `(v,u)` is present, and `(v,v)` is present.

This program reads in the adjacency structure, decrements the offsets and indices by one if specified by the flag parameter (our application came from a Fortran code with 1-indexing), then loads the entries into a `InpMtx` object where they are assembled and sorted by rows. The (v, v) entries are loaded, and each vector of the adjacency structure is loaded as both a column and as a row, so in effect we are constructing the graph of $(A + A^T)$. Recall, multiple entries are collapsed during the sort and merge step.

A `Graph` object is then created using the `Graph_fillFromOffsets()` method using the vectors in the `InpMtx` object. The `Graph` object is then optionally written to a file.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inAdjacencyFile` parameter is the input file for the adjacency structure as defined above. It must be a formatted file.
- The `outGraphFile` parameter is the output file for the `Graph` object. If `outGraphFile` is `none` then the `Graph` object is not written to a file. Otherwise, the `Graph_writeToFile()` method is called to write the object to a formatted file (if `outGraphFile` is of the form `*.graphf`), or a binary file (if `outGraphFile` is of the form `*.graphb`).
- The `flag` parameter is used to specify whether the offsets and indices are 0-indexed (as in C) or 1-indexed (as in Fortran). If they are 1-indexed, the offsets and indices are decremented prior to loading into the `InpMtx` object.

8. `testR2D msglvl msgFile EGraphFile CoordsFile coordType seed outInpMtxFile`

   This driver program reads in an `EGraph` element graph and a `Coords` grid point coordinate object for one of the `R2D*` randomly triangulated 2-D grids. It then generates the finite element matrices for each of the triangular elements and assembles the matrices into a `InpMtx` object, which is then optionally written out to a file. A matrix-vector product is computed using the unassembled matrix and the assembled matrix and compared to detect errors. The `InpMtx` object is then permuted and a matrix-vector multiply again computed and checked for errors.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - The `EGraphFile` is the file that holds the `EGraph` object — must be of the form `*.egraphf` or `*.egraphb`.
   - The `CoordsFile` is the file that holds the `Coords` object — must be of the form `*.coordsf` or `*.coordsb`.
   - The `coordType` determines the coordinate type for the `InpMtx` object.

- `1` — storage of entries by rows
- `2` — storage of entries by columns
- `3` — storage of entries by chevrons

- The `seed` parameter is used as a random number seed to determine the row and column permutations for the matrix-vector multiply.
- The `outInpMtxFile` parameter is the output file for the `InpMtx` object. If `outInpMtxFile` is `none` then the `InpMtx` object is not written to a file. Otherwise, the `InpMtx_writeToFile()` method is called to write the object to a formatted file (if `outInpMtxFile` is of the form `*.inpmtxf`), or a binary file (if `outInpMtxFile` is of the form `*.inpmtxb`).

9. `readAIJ msglvl msgFile inputFile outInpMtxFile flag`

   This driver program reads $(i, j, a_{i,j})$ triples from a file, loads them into a `InpMtx` object, and optionally writes the object out to a file. The input file has the form:

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means that all objects are written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any message data.
   - The `inputFile` is the file that holds the triples. It has the following form.

     ```
     nrow ncol nentries
     irow jcol value
      ...
     irow jcol value
     ```

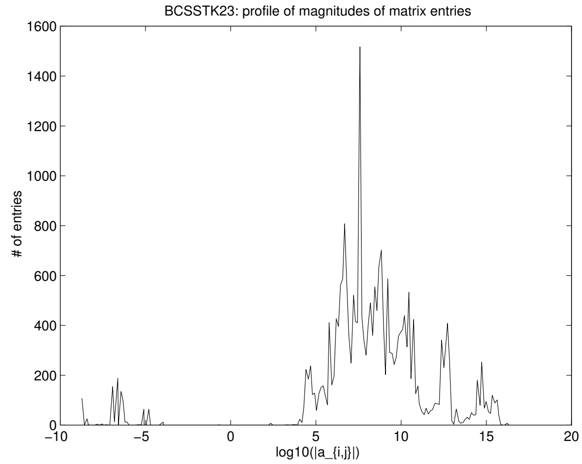
     Note, `nrow` and `ncol` are not used by the `InpMtx` object — each (irow, jcol, value) triple is loaded.
   - The `outInpMtxFile` parameter is the output file for the `InpMtx` object. If `outInpMtxFile` is `none` then the `InpMtx` object is not written to a file. Otherwise, the `InpMtx_writeToFile()` method is called to write the object to a formatted file (if `outInpMtxFile` is of the form `*.inpmtxf`), or a binary file (if `outInpMtxFile` is of the form `*.inpmtxb`).
   - The `flag` parameter is used to specify whether the indices are 0-indexed (as in C) or 1-indexed (as in Fortran). If they are 1-indexed, the indices are decremented prior to loading into the `InpMtx` object.

10. `getProfile msglvl msgFile inInpMtxFile npts tausmall taubig`

    This driver program produces a profile of the magnitudes of the matrix entries in a format that is suitable for plotting by Matlab. The `npts` parameter specifies how many points to be used in the profile plot. The message file will contain line of the form.

    ```
    data = [ ...
        x1    y1
        ...
        xnpts ynpts ] ;
    ```

    which can be used to generate the following matlab plot. An example is given below for the BCSSTK23 matrix, where `npts = 200`, `tausmall = 1.e-10` and `taubig = 1.e100`.



The number of entries that are zero, the number whose magnitude is less than `tausmall`, and the number whose magnitude is larger than `taubig` are printed to `msgFile`.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inInpMtxFile` parameter is the input file for the `InpMtx` object that holds the matrix. It must be of the form `*.inpmtxf` or `*.inpmtxb`. The `InpMtx` object is read from the file via the `InpMtx_readFromFile()` method.
- The `npts` parameter determines the number of points to use in the plot.
- The `tausmall` parameter is a lower cutoff for putting entries in the profile plot.
- The `taubig` parameter is an upper cutoff for putting entries in the profile plot.

11. `mkNaturalFactorMtx msglvl msgFile n1 n2 n3 seed outFile`

    This driver program generates rectangular matrix that would arise from a natural factor representation of the Laplacian operator on a regular grid. If `n3 = 1`, we have a `n1`×`n2` grid. There are `(n1-1)*(n2-1)` elements and each element gives rise to four equations, so the resulting matrix has `4(n1-1)*(n2-1)` rows and `n1*n2` columns. If `n3 > 1`, we have a `n1`×`n2`×`n3` grid. There are `(n1-1)*(n2-1)*(n3-1)` elements and each element gives rise to eight equations, so the resulting matrix has `8(n1-1)*(n2-1)*(n3-1)` rows and `n1*n2*n3` columns.

    - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - `n1` is the number of points in the first direction.
    - `n2` is the number of points in the second direction.
    - `n3` is the number of points in the third direction.
    - The `seed` parameter is a random number seed used to fill the matrix entries with random numbers.
    - The `outFile` parameter is the output file for the `InpMtx` object that holds the matrix. It must be of the form `*.inpmtxf` or `*.inpmtxb`. The `InpMtx` object is written to the file via the `InpMtx_writeToFile()` method.

12. `testMMM msglvl msgFile dataType symflag coordType transpose`
    `nrow ncol nitem nrhs seed alphaReal alphaImag`

    This driver program tests the matrix-matrix multiply methods. This driver program generates A, a `nrow` $\times$ `ncol` matrix using `nitem` input entries, X and Y, `nrow` $\times$ `nrhs` matrices, and all are filled with random numbers. It then computes $Y := Y + \alpha AX$, $Y := Y + \alpha A^T X$ or $Y := Y + \alpha A^H X$. The program's output is a file which when sent into Matlab, outputs the error in the computation.

    - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - `dataType` is the type of entries, `0` for real, `1` for complex.
    - `symflag` is the symmetry flag, `0` for symmetric, `1` for Hermitian, `2` for nonsymmetric.
    - `coordType` is the storage mode for the entries, `1` for by rows, `2` for by columns, `3` for by chevrons.
    - `transpose` determines the equation, `0` for $Y := Y + \alpha AX$, `1` for $Y := Y + \alpha A^H X$ or `2` for $Y := Y + \alpha A^T X$.
    - `nrowA` is the number of rows in A
    - `ncolA` is the number of columns in A
    - `nitem` is the number of matrix entries that are assembled into the matrix.
    - `nrhs` is the number of columns in X and Y.
    - The `seed` parameter is a random number seed used to fill the matrix entries with random numbers.
    - `alphaReal` and `alphaImag` form the scalar in the multiply.

13. `testGMMM msglvl msgFile dataType symflag coordType transpose`
    `nrow ncol nitem nrhs seed alphaReal alphaImag betaReal betaImag`

    This driver program tests the generalized matrix-matrix multiply methods. It generates A, a `nrow` $\times$ `ncol` matrix using `nitem` input entries, X and Y, `nrow` $\times$ `nrhs` matrices, and all are filled with random numbers. It then computes $Y := \beta Y + \alpha AX$, $Y := \beta Y + \alpha A^T X$ or $Y := \beta Y + \alpha A^H X$. The program's output is a file which when sent into Matlab, outputs the error in the computation.

    - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - `dataType` is the type of entries, `0` for real, `1` for complex.
    - `symflag` is the symmetry flag, `0` for symmetric, `1` for Hermitian, `2` for nonsymmetric.
    - `coordType` is the storage mode for the entries, `1` for by rows, `2` for by columns, `3` for by chevrons.
    - `transpose` determines the equation, `0` for $Y := \beta Y + \alpha AX$, `1` for $Y := \beta Y + \alpha A^H X$ or `2` for $Y := \beta Y + \alpha A^T X$.
    - `nrowA` is the number of rows in A
    - `ncolA` is the number of columns in A
    - `nitem` is the number of matrix entries that are assembled into the matrix.
    - `nrhs` is the number of columns in X and Y.
    - The `seed` parameter is a random number seed used to fill the matrix entries with random numbers.

- `alphaReal` and `alphaImag` form the α scalar in the multiply.
- `betaReal` and `betaImag` form the β scalar in the multiply.

14. `testGMVM msglvl msgFile dataType symflag coordType transpose nrow ncol nitem seed alphaReal alphaImag betaReal betaImag`

    This driver program tests the generalized matrix-vector multiply methods. It generates A, a `nrow`×`ncol` matrix using `nitem` input entries, x and y, and fills the matrices with random numbers. It then computes $y := \beta y + \alpha A x$, $y := \beta y + \alpha A^T x$ or $y := \beta y + \alpha A^H x$. The program's output is a file which when sent into Matlab, outputs the error in the computation.

    - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - `dataType` is the type of entries, `0` for real, `1` for complex.
    - `symflag` is the symmetry flag, `0` for symmetric, `1` for Hermitian, `2` for nonsymmetric.
    - `coordType` is the storage mode for the entries, `1` for by rows, `2` for by columns, `3` for by chevrons.
    - `transpose` determines the equation, `0` for $y := \beta y + \alpha A x$, `1` for $y := \beta y + \alpha A^T x$ or `2` for $y := \beta y + \alpha A^H x$.
    - `nrowA` is the number of rows in A
    - `ncolA` is the number of columns in A
    - `nitem` is the number of matrix entries that are assembled into the matrix.
    - The `seed` parameter is a random number seed used to fill the matrix entries with random numbers.
    - `alphaReal` and `alphaImag` form the α scalar in the multiply.
    - `betaReal` and `betaImag` form the β scalar in the multiply.

15. `testHBIO msglvl msgFile inFile outFile`

    This driver program read in a matrix from a Harwell-Boeing file, and optionally writes it to a formatted or binary `InpMtx` file.

    - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `inFile` parameter is the Harwell-Boeing file.
    - The `outFile` parameter is the output file for the `InpMtx` object. If `outFile` is `none` then the `InpMtx` object is not written to a file. Otherwise, the `InpMtx_writeToFile()` method is called to write the object to a formatted file (if `outFile` is of the form `*.inpmtxf`), or a binary file (if `outFile` is of the form `*.inpmtxb`).

Chapter 33

Iter: Iterative Methods

Iter is composed of 5 Krylov space iterative methods, PCG (Preconditioned Conjugate Gradients), BiCGStab, TFQMR, and BGMRES (Block GMRES), and MLBiCGStab. (For references, see top comments in codes.) The intent of these methods is to provide the user of **SPOOLES** with an easy way to evaluate the effectiveness of the approximate factorizations belonging to the FrontMtx object. To further facilitate the evaluation we have included a single call driver that can run anyone of the methods we have provided with the type of preconditioner desired. For each iterative method we allow for left and right preconditioning. Also, for each method, except BGMRES, we allow for real or complex matrices.

Because our intent was to provide a simple means to test the effectiveness of the preconditioners, these implementations are not parallel (neither shared or distributed memory). However, they were intentionally written to be consistent in style and form so that they could be easily adapted to exploit the parallelism that is in **SPOOLES**. All iterative methods use the basic structure DenseMtx for handling the intermediate vectors and performing the matrix multiplications and system solves. By doing this we have also anticipated the eventual movement to block iterative methods and the DenseMtx structure can remain the basic structure. There are a few basic utilities that have been added, which are discribed in this section, upon which the iterative methods were built. These are provided to aid the experienced **SPOOLES** user with an ability to develop additional iterative methods, as seen fit.

33.1 Data Structure

The methods in Iter solve a linear system $AX = B$, where A is an InpMtx object and X and B are DenseMtx objects. The preconditioner is a FrontMtx object obtained via frontal method which uses several other objects. See header file Iter.h for further information.

33.2 Prototypes and descriptions of Iter methods

This section contains brief descriptions including prototypes of all methods found in the Iter source directory.

33.2.1 Utility methods

1. `double DenseMtx_frobNorm ( DenseMtx *mtx ) ;`

   This method returns the Frobenius norm of the matrix.

   Error checking: If mtx is NULL, an error message is printed and the program exits.

2. `double DenseMtx_twoNormOfColumn ( DenseMtx *mtx, int jcol ) ;`

   This method returns the two-norm of column `jcol` of the matrix.

   Error checking: If `mtx` is `NULL`, or `jcol` is not in `[0,ncol-1]`, an error message is printed and the program exits.

3. ```
   void DenseMtx_colCopy ( DenseMtx *mtxB, int jcol,
                           DenseMtx *mtxA, int icol ) ;
   ```

   This method copies the column `icol` of the matrix `mtxA` to the column `jcol` of the matrix `mtxB`.

   Error checking: If `mtxA` or `mtxB` is `NULL`, `jcol` is not in `[0,ncolB-1]`, or `icol` is not in `[0,ncolA-1]` an error message is printed and the program exits.

4. ```
   void DenseMtx_colDotProduct ( DenseMtx *mtxA, int icol,
                                 DenseMtx *mtxB, int jcol, double *prod ) ;
   ```

   This method computes dot product of column `icol` of the matrix `mtxA` and column `jcol` of the matrix `mtxB`. Note that the column `icol` of the matrix `mtxA` will be transported and conjugated for complex entries.

   Error checking: If `mtxA` or `mtxB` is `NULL`, `jcol` is not in `[0,ncolB-1]`, or `icol` is not in `[0,ncolA-1]` an error message is printed and the program exits.

5. ```
   void DenseMtx_colGenAxpy ( double *alpha, DenseMtx *mtxA, int icol,
                              double *beta,  DenseMtx *mtxB, int jcol ) ;
   ```

   This method replaces column `icol` of the matrix `mtxA` by `alpha` times itself plus `beta` times column `jcol` of `mtxB`.

   Error checking: If `mtxA` or `mtxB` is `NULL`, `jcol` is not in `[0,ncolB-1]`, or `icol` is not in `[0,ncolA-1]` an error message is printed and the program exits.

6. ```
   int DenseMtx_mmm ( char *A_opt, char *B_opt, double *beta, DenseMtx *mtxC,
      double *alpha, DenseMtx *mtxA, DenseMtx *mtxB );
   ```

   This method computes the matrix-matrix multiplication $C := \beta C + \alpha AB$, where A, B and C are found in the `C DenseMtx` object, β and α are real or complex in `beta[]` and `alpha[]`. If any of the input objects are `NULL`, an error message is printed and the program exits. `A`, `B` and `C` must all be real or all be complex. When `A and B` are real, then α = `alpha[0]`. When `A and B` are complex, then α = `alpha[0]` + i* `alpha[1]`. When `C` is real, then β = `beta[0]`. When `C` is complex, then β = `beta[0]` + i* `beta[1]`. This means that one cannot call the method with a constant as the third and fifth parameter, e.g., `DenseMtx_mmm(a_opt, b_opt, beta, C, alpha, A, B)`, for this may result in a segmentation violation. The values of α and β must be loaded into an array of length 1 or 2 .

   Error checking: If `beta`, `alpha`, `C`, `A`, `B` are `NULL`, or if `C`, `A` and `B` do not have the same data type (`SPOOLES_REAL` or `SPOOLES_COMPLEX`), or if `A_opt` or `B_opt` is invalid, or the number of column of `A` and the number of row of `B` is not match, an error message is printed and the program exits.

7. ```
   void FrontMtx_solveOneColumn ( FrontMtx *frontmtx, DenseMtx *solmtx,
      int jcol, DenseMtx *rhsmtx, int icol, SubMtxManager *mtxmanager,
      double cpus[], int msglvl, FILE *msgFile ) ;
   ```

   This method is used to solve one of three linear systems of equations — $(U^T + I)D(I + U)X = B$, $(U^H + I)D(I + U)X = B$ or $(L + I)D(I + U)X = B$. Entries of B are read from column `icol` of `rhsmtx` and entries of X are written to column `jcol` of `solmtx`. Therefore, `rhsmtx` and `solmtx` can be the same object. (Note, this does not hold true for an MPI factorization with pivoting.) The `mtxmanager` object manages the working storage using the solve. On return the `cpus[]` vector is filled with the following.
   ```

- cpus[0] — set up the solves
- cpus[1] — fetch right hand side and store solution
- cpus[2] — forward solve
- cpus[3] — diagonal solve
- cpus[4] — backward solve
- cpus[5] — total time in the method.

Error checking: If frontmtx, rhsmtx or cpus is NULL, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

33.2.2 Iterative methods

A collection of iterative methods is provided to solve a sparse linear system $AX = B$, where A is an InpMtx object and X and B are DenseMtx objects. This includes left and right preconditioning BiCGStab, MLBiCGStab, TFQMR, PCG, and BGMRES. All methods have similar input arguments:

- n_matrixSize is order of the matrix A.
- type is the type of entries, 0 for real, 1 for complex.
- The symmetryflag parameter specifies the symmetry of the matrix A.
  - type = 0 (SPOOLES_SYMMETRIC) for A real or complex symmetric,
  - type = 1 (SPOOLES_HERMITIAN) for A complex Hermitian,
  - type = 2 (SPOOLES_NONSYMMETRIC) for A real or complex nonsymmetric.
- mtxA is the matrix A.
- Precond is the preconditioner.
- mtxX is the solution vectors X saved as a DenseMtx object.
- mtxB is the right-hand-side vectors B saved as a DenseMtx object.
- itermax is the maximum iterations number.
- convergetol parameter is a stop criterion for iterative algorithms.
- maxninner is the maximum number of inner iterations in BGMRES method.
- maxnouter is the maximum number of outer iterations in BGMRES method.
- pninner is last number of inner iterations executed in BGMRES method.
- pnouter is last number of outer iterations executed in BGMRES method.
- mtxQ is the starting vectors saved as a DenseMtx object for MLBiCGStab method.
- The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The msglvl parameter determines the amount of output — taking msglvl >= 3 means most of the objects are written to the message file.

1. `int bicgstabr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using BiCGStab algorithm with right preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

2. `int bicgstabl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using BiCGStab algorithm with left preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

3. `int mlbicgstabr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxQ, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using MLBiCGStab algorithm with right preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

4. `int mlbicgstabl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxQ, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using MLBiCGStab algorithm with left preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

5. `int tfqmrr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using TFQMR algorithm with right preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

6. `int tfqmrl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using TFQMR algorithm with left preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

7. `int pcgr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real symmetric position definite linear system using PCG algorithm with right preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

8. `int pcgl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real symmetric position definite linear system using PCG algorithm with left preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

9. `int bgmresr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int maxnouter, int maxninner, int *pnouter, int *pninner, double convergetol, int msglvl, FILE *msgFile ) ;`

   This method solves a real linear system using BGMRES algorithm with right preconditioner.

   Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

10. `int bgmresl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int maxnouter, int maxninner, int *pnouter, int *pninner, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a real linear system using BGMRES algorithm with left preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

11. `int zbicgstabr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex linear system using BiCGStab algorithm with right preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

12. `int zbicgstabl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex linear system using BiCGStab algorithm with left preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

13. `int zmlbicgstabr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxQ, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex linear system using MLBiCGStab algorithm with right preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

14. `int zmlbicgstabl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxQ, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex linear system using MLBiCGStab algorithm with left preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

15. `int ztfqmrr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex linear system using TFQMR algorithm with right preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

16. `int ztfqmrl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex linear system using TFQMR algorithm with left preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

17. `int zpcgr ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex hermitian position definite linear system using PCG algorithm with right preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

18. `int zpcgl ( int n_matrixSize, int type, int symmetryflag, InpMtx *mtxA, FrontMtx *Precond, DenseMtx *mtxX, DenseMtx *mtxB, int itermax, double convergetol, int msglvl, FILE *msgFile ) ;`

    This method solves a complex hermitian position definite linear system using PCG algorithm with left preconditioner.

    Return codes: `1` is a normal return. Otherwise, an error message is printed and the program exits.

33.3 Driver programs

1. `test_colCopy msglvl msgFile type n1 n2 inc1 inc2 icol jcol seed`

   This driver program generates a `DenseMtx` object whose column `icol` is copied to column `jcol`.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `DenseMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `type` is the type of entries, `0` for real, `1` for complex.
   - `n1` is the row dimension of the test matrix.
   - `n2` is the column dimension of the test matrix.
   - `inc1` is the row increment.
   - `inc2` is the column increment.
   - `icol` is the column number to be copied. $0 \leq$`icol`$<$`n2`.
   - `jcol` is the column number to be replaced. $0 \leq$`jcol`$<$`n2`.
   - `seed` parameter is random number seed.

2. `test_colDotProduct msglvl msgFile type n1 n2 inc1 inc2 icol jcol seed`

   This driver program generates a `DenseMtx` object object, and computes the dot product of column `icol` and column `jcol` of the matrix.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `DenseMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `type` is the type of entries, `0` for real, `1` for complex.
   - `n1` is the row dimension of the test matrix.
   - `n2` is the column dimension of the test matrix.
   - `inc1` is the row increment.

- inc2 is the column increment.
- icol is the first column number. $0 \leq$icol<n2.
- jcol is the second column number. $0 \leq$jcol<n2.
- seed parameter is random number seed.

3. test_colGenAxpy msglvl msgFile type n1 n2 inc1 inc2 icol jcol
   ralpha, ialpha, rbeta, ibeta, seed

   This driver program generates a DenseMtx object whose column icol is replaced by α times column icol plus β times column jcol.

   - The msglvl parameter determines the amount of output — taking msglvl >= 3 means the DenseMtx object is written to the message file.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - type is the type of entries, 0 for real, 1 for complex.
   - n1 is the row dimension of the test matrix.
   - n2 is the column dimension of the test matrix.
   - inc1 is the row increment.
   - inc2 is the column increment.
   - icol is the column number to be replaced. $0 \leq$icol<n2.
   - jcol is the column number to be added. $0 \leq$jcol<n2.
   - ralpha is the real part of the scalar α.
   - ialpha is the imaginary part of the scalar α.
   - rbeta is the real part of the scalar β.
   - ibeta is the imaginary part of the scalar β.
   - seed parameter is random number seed.

4. test_frobNorm msglvl msgFile type n1 n2 inc1 inc2 seed

   This driver program generates a DenseMtx object and computes the Frobenius norm of this matrix.

   - The msglvl parameter determines the amount of output — taking msglvl >= 3 means the DenseMtx object is written to the message file.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - type is the type of entries, 0 for real, 1 for complex.
   - n1 is the row dimension of the test matrix.
   - n2 is the column dimension of the test matrix.
   - inc1 is the row increment.
   - inc2 is the column increment.
   - seed parameter is random number seed.

5. test_frobNorm msglvl msgFile type n1 n2 inc1 inc2 jcol seed

   This driver program generates a DenseMtx object and computes two norm of column jcol of this matrix.

- The msglvl parameter determines the amount of output — taking msglvl >= 3 means the DenseMtx object is written to the message file.
- The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- type is the type of entries, 0 for real, 1 for complex.
- n1 is the row dimension of the test matrix.
- n2 is the column dimension of the test matrix.
- inc1 is the row increment.
- inc2 is the column increment.
- jcol is the column number whose two norm is required. $0 \leq$jcol<n2.
- seed parameter is random number seed.

6. `test_DenseMtx_mmm msglvl msgFile type nrow nk ncol ainc1 ainc2 binc1 binc2 cinc1 cinc2 a_opt b_opt ralpha ialpha rbeta ibeta seed`

   This driver program tests the matrix-matrix multiply method. The program generates DenseMtx objects A, ,B and C. It returns the matrix C whose elements are replaced by β times matrix C plus α times matrix A times matrix B.

   - The msglvl parameter determines the amount of output — taking msglvl >= 3 means the DenseMtx object is written to the message file.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - type is the type of entries, 0 for real, 1 for complex.
   - nrow is the row dimension of the test matrix A.
   - nk is the column dimension of the test matrix A and the row dimension of the test matrix B.
   - ncol is the column dimension of the test matrix B.
   - ainc1 is the row increment for the test matrix A.
   - ainc2 is the column increment for the test matrix A.
   - binc1 is the row increment for the test matrix B.
   - binc2 is the column increment for the test matrix B.
   - cinc1 is the row increment for the test matrix C.
   - cinc2 is the column increment for the test matrix C.
   - a_opt specifies the computation of the test matrix A to be performed. "n" or "N" is No transpose. "t" or "T" is Transpose. "c" or "C" is Conjugate transpose.
   - b_opt specifies the computation of the test matrix B to be performed. "n" or "N" is No transpose. "t" or "T" is Transpose. "c" or "C" is Conjugate transpose.
   - ralpha is the real part of the scalar α.
   - ialpha is the imaginary part of the scalar α.
   - rbeta is the real part of the scalar β.
   - ibeta is the imaginary part of the scalar β.
   - seed parameter is random number seed.

7. `iter inFile`

   This driver program reads required parameters from the `inFile` to solve a sparse linear system $AX = B$, where A is an `InpMtx` object and X and B are `DenseMtx` objects, using selected methods with left or right preconditioner. The preconditioner is obtained via applying frontal method to the matrix A. In the `inFile`, the required parameters are in a layout as

```
srcMtxFormat
srcMtxFile
InpMtxFile
ETreeFormat
ETreeFile
rhsFile
slnFile
msgFile
msglvl  seed  nrhs  Ik  itermax  iterout
symmetryflag  sparsityflag  pivotingflag
tau  droptol  convtol
methods
```

   All comment lins should start with a start (`*`) and the lines order of the required parameters should not be changed.

   - `srcMtxFormat` is the file format of source matrix A, `0` for InpMtx, `1` for HBF, and `2` for AIJ2.
   - `srcMtxFile` is the file name saved the source matrix A.
   - `InpMtxFile` is the file name to save InpMtx object if the original input matrix is in HBF or AIJ2 format. It should be with extension `.inpmtxb` or `.inpmtxf`. If `InpMtxFile` is `none`, the converted InpMtx object will not be written to file.
   - `ETreeFormat` is the source format for ETree object. `0` for reading from file, `1` for obtaining via the best of a nested dissection and a multisection ordering, `2` for obtaining via a multiple minimum degree ordering, `3` for obtaining via a multisection ordering, and `4` for obtaining via a nested dissection ordering.
   - `ETreeFile` is the name of file from which ETree object is read if `ETreeFormat` is `0`. Otherwise, it is the file name to save the computed ETree object. It should be with extension `.etreeb` or `.etreef`. If `ETreeFile` is `none`, the computed ETree object will not be written to file.
   - `rhsFile` is the name of file from which right-hand-side vectors B is read. It should be with extension `.densemtxb` or `.densemtxf`. If `rhsFile` is `none`, the right-hand-side B is generated by random numbers.
   - `slnFile` is the name of file from which solution vectors X is saved. It should be with extension `.densemtxb` or `.densemtxf`. If `rhsFile` is `none`, the solution X is not saved.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means most of the objects are written to the message file.
   - `seed` parameter is random number seed.
   - `nrhs` is the number of columns of right-hand-side B.
   - `Ik` is a block parameter for MLBiCGStab method.
   - `itermax` is the maximum iterations number. (inner iterations number for GMRES method)

- `iterout` is the maximum outer number of iterations for GMRES method.
- The `symmetryflag` parameter specifies the symmetry of the matrix A.
  - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,
  - `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
  - `type = 2 (SPOOLES_NONSYMMETRIC)`

  for A real or complex nonsymmetric.
- The `sparsityflag` parameter signals a direct or approximate factorization.
  - `sparsityflag = 0 (FRONTMTX_DENSE_FRONTS)` implies a direct factorization, the fronts will be stored as dense submatrices.
  - `sparsityflag = 1 (FRONTMTX_SPARSE_FRONTS)` implies an approximate factorization. The fronts will be stored as sparse submatrices, where the entries in the triangular factors will be subjected to a drop tolerance test — if the magnitude of an entry is `droptol` or larger, it will be stored, otherwise it will be dropped.
- The `pivotingflag` parameter signals whether pivoting for stability will be enabled or not.
  - If `pivotingflag = 0 (SPOOLES_NO_PIVOTING)`, no pivoting will be done.
  - If `pivotingflag = 1 (SPOOLES_PIVOTING)`, pivoting will be done to ensure that all entries in U and L have magnitude less than `tau`.
- The `tau` parameter is an upper bound on the magnitude of the entries in L and U when pivoting is enabled.
- The `droptol` parameter is a lower bound on the magnitude of the entries in L and U when the approximate factorization is enabled.
- `convtol` parameter is a stop criterion for iterative algorithms.
- `methods` parameters are choices of iterative algorithms, `0` for BiCGStabR, `1` for BiCGStabL, `2` for MLBiCGStabR, `3` for MLBiCGStabL, `4` for TFQMRR, `5` for TFQMRL, `6` for PCGR `7`, for PCGL `8` for BGMRESR, and `9` for BGMRESL.

Chapter 34

`PatchAndGoInfo`: Pivot Modification Object

On occasion, an application will demand specific behavior during a factorization. We have written the `PatchAndGoInfo` object to communicate information to the `Chv` object during a factorization of a front. Most users can ignore this object. However, if a different type of behavior is required, one could extend this object by adding a new strategy to it and modifying the `Chv` methods that factor a front.

Let us describe two strategies that we presently support.

- Primal-dual linear programming may require repeated factorizations of matrices of the form AD^2A^T, where A comes from constraint equations and D is a diagonal matrix. As the optimization proceeds, AD^2A^T becomes increasingly ill-conditioned because the entries in D go to zero or infinity. Normally, when a small or zero pivot element is detected, we would either signal an error (if we expected the matrix to be positive definite) or pivot for stability. However, in the primal-dual pivot context, a small or zero element on the diagonal is not a calamity. It signals that the variable associated with the small entry can be "skipped" in the solution process. There are several ways to implement this behavior. We have chosen a simple way: the diagonal entry is set to 1.0 and all off-diagonal entries in the corresponding column of L are set to zero.

- In structural analysis, "multi-point constraints" are often applied to a linear system. At times, applying these constraints generates a matrix that is essentially singular. The singularity may be benign, as in the following case.
$$\begin{bmatrix} A_{1,1} & 0 \\ 0 & A_{2,2} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 0 \\ B_2 \end{bmatrix}$$
If $A_{1,1}$ is singular, the solution $X_1 = 0$ and $X_2 = A_{2,2}^{-1}B_2$ is perfectly acceptable. In other cases, the location of the singularity can be communicated back to the user to supply useful information about the finite element model. One common practice is to not use pivoting, but to check the magnitude of the diagonal entry as a row and column is to be eliminated. If the magnitude is smaller than a user-supplied parameter, the diagonal entry is set to some multiple of the largest offdiagonal entry in that row and column of the front, the location and perturbation is noted, and the factorization proceeds.

Other strategies can be added to the `PatchAndGoInfo` object. For example, if a matrix is being factored that is believed to be positive definite, and a negative value is found in a pivot element, one could abort the factorization, or perturb the element so that it is positive.

34.1 Data Structure

The `PatchAndGoInfo` structure has five fields.

- `int strategy` : type of patch-and-go strategy
  - `1` — used with optimization matrices, if $|A_{i,i}| \leq$ `toosmall` then set $A_{i,i} = 1$ and $L_{j,i} = 0$ for $j > i$.
  - `2` — used with structural analysis matrices, if $|A_{i,i}| \leq$ `fudge` then set $A_{i,i} =$ `fudge`$\cdot \max\{1, \max_{j>i}\{|A_{i,j}|, |A_{j,i}|\}\}$
- `double toosmall` : cutoff for diagonal entry magnitude
- `double fudge` : pertubation multiplier for modification
- `IV *fudgeIV` : vector to collect locations of perturbations, may be `NULL`.
- `DV *fudgeDV` : vector to collect perturbations, may be `NULL`.

34.2 Prototypes and descriptions of `PatchAndGoInfo` methods

This section contains brief descriptions including prototypes of all methods that belong to the `PatchAndGoInfo` object.

34.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `PatchAndGoInfo * PatchAndGoInfo_new ( void ) ;`

   This method simply allocates storage for the `PatchAndGoInfo` structure and then sets the default fields by a call to `PatchAndGoInfo_setDefaultFields()`.

2. `void PatchAndGoInfo_setDefaultFields ( PatchAndGoInfo *info ) ;`

   This method sets the structure's fields to default values: `strategy` = -1, `toosmall` = `fudge` = 0.0, and `fudgeIV` = `fudgeDV` = `NULL` .

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

3. `void PatchAndGoInfo_clearData ( PatchAndGoInfo *info ) ;`

   This method clears any data owned by the object. If `fudgeIV` is not `NULL` it is free'd by a call to `IV_free()`. If `fudgeDV` is not `NULL` it is free'd by a call to `DV_free()`. The structure's default fields are then set with a call to `PatchAndGoInfo_setDefaultFields()`.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

4. `void PatchAndGoInfo_free ( PatchAndGoInfo *info ) ;`

   This method releases any storage by a call to `PatchAndGoInfo_clearData()` then free's the storage for the structure with a call to `free()`.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

34.2.2 Initializer methods

1. `void PatchAndGoInfo_init ( PatchAndGoInfo *info, int strategy, double toosmall, double fudge, int storeids, int storevalues ) ;`

   This method initializes the object. Presently, two strategies are supported: `strategy = 1` for optimization matrices and `strategy = 2` for structural analysis matrices. `toosmall` is the cutoff for diagonal entry modification, if an entry has magnitude less than `toosmall` some action is taken. For the second strategy, the `fudge` parameter contributes to the perturbation. When `storeids` is not zero, the `fudgeIV` object is created to accumulate the locations of the perturbations. When `storevalues` is not zero, the `fudgeDV` object is created to accumulate information on the perturbations themselves.

   Error checking: If `info` is `NULL` or `strategy` is not 1 or 2, or `toosmall` or `fudge` are less than zero, an error message is printed and the program exits.

Chapter 35

Pencil: Matrix pencil

This object stores a matrix pencil $A + \sigma B$. A and B are both stored as InpMtx objects. Many of the Pencil methods simply call the equivalent InpMtx method.

35.1 Data Structure

The Pencil structure has the following fields.

- int type : type of matrix entries,
  - SPOOLES_REAL for real entries
  - SPOOLES_COMPLEX for complex entries
- int symflag : type of symmetry present in the matrices
  - SPOOLES_SYMMETRIC for real or complex symmetric matrices
  - SPOOLES_HERMITIAN for complex Hermitian matrices
  - SPOOLES_NONSYMMETRIC for real or complex nonsymmetric matrices
- InpMtx *inpmtxA : pointer to the matrix object for A. If inpmtxA is NULL, then A is the identity matrix.
- InpMtx *inpmtxB : pointer to the matrix object for B. If inpmtxB is NULL, then B is the identity matrix.
- double sigma[2] : real or complex scalar shift value.

35.2 Prototypes and descriptions of Pencil methods

This section contains brief descriptions including prototypes of all methods that belong to the Pencil object.

35.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Pencil * Pencil_new ( void ) ;`

   This method simply allocates storage for the `Pencil` structure and then sets the default fields by a call to `Pencil_setDefaultFields()`.

2. `void Pencil_setDefaultFields ( Pencil *pencil ) ;`

   The structure's fields are set to default values: `sigma[2] = {0,0}`, `type` = `SPOOLES_REAL`, `symflag` = `SPOOLES_SYMMETRIC`, and `inpmtxA` = `inpmtxB` = `NULL` .

   Error checking: If `pencil` is `NULL`, an error message is printed and the program exits.

3. `void Pencil_clearData ( Pencil *pencil ) ;`

   This method clears the object and free's any owned data by invoking the `InpMtx_free()` method for the `inpmtxA` and `inpmtxB` objects. There is a concluding call to `Pencil_setDefaultFields()`.

   Error checking: If `pencil` is `NULL`, an error message is printed and the program exits.

4. `void Pencil_free ( Pencil *pencil ) ;`

   This method releases any storage by a call to `Pencil_clearData()` and then free the space for `pencil`.

   Error checking: If `pencil` is `NULL`, an error message is printed and the program exits.

35.2.2 Initialization methods

1. ```
   void Pencil_init( Pencil *pencil, int type, int symflag,
                     InpMtx *inpmtxA, double sigma[], InpMtx *inpmtxB ) ;
   ```

   The fields of the pencil object are set to the input parameters.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

35.2.3 Utility methods

1. `void Pencil_changeCoordType ( Pencil *pencil, int newType ) ;`

   This method simply calls the `InpMtx_changeCoordType()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

2. `void Pencil_changeStorageMode ( Pencil *pencil, int newMode ) ;`

   This method simply calls the `InpMtx_changeStorageMode()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

3. `void Pencil_sortAndCompress ( Pencil *pencil ) ;`

   This method simply calls the `InpMtx_sortAndCompress()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

4. `void Pencil_convertToVectors ( Pencil *pencil ) ;`

   This method simply calls the `InpMtx_sortAndCompress()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

5. `void Pencil_mapToLowerTriangle ( Pencil *pencil ) ;`

   This method simply calls the `InpMtx_mapToLowerTriangle()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.
   ```

6. `void Pencil_mapToUpperTriangle ( Pencil *pencil ) ;`

   This method simply calls the `InpMtx_mapToUpperTriangle()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

7. `void Pencil_permute ( Pencil *pencil, IV *rowOldToNewIV, IV *colOldToNewIV ) ;`

   This method simply calls the `InpMtx_permute()` method for each of its two matrices.

   Error checking: If `pencil` is `NULL`, an error message is printed and zero is returned.

8. `void Pencil_mmm ( Pencil *pencil, DenseMtx *Y, DenseMtx *X ) ;`

   This method is used to compute $X = (A + \sigma B)X$.

   Error checking: If `pencil`, `X` or `Y` is `NULL` an error message is printed and the program exits.

9. `IVL * Pencil_fullAdjacency ( Pencil *pencil ) ;`

   This method returns an IVL object that holds the full adjacency structure of $(A + \sigma B) + (A + \sigma B)^T$.

   Error checking: If `pencil` is `NULL`, an error message is printed and the program exits.

35.2.4 IO methods

1. ```
   Pencil * Pencil_setup ( int myid, int symflag, char *inpmtxAfile,
      double sigma[], char *inpmtxBfile, int randomflag, Drand *drand,
      int msglvl, FILE *msgFile ) ;
   ```

   This method is used to read in the matrices from two files and initialize the objects. If the file name is "none", then no matrix is read. If `symflag` is `SPOOLES_SYMMETRIC` or `SPOOLES_HERMITIAN`, entries in the lower triangle are dropped. If `randomflag` is one, the entries are filled with random numbers using the `Drand` random number generator `drand`.

   Note: this method was created for an MPI application. If `myid` is zero, then the files are read in, otherwise just stubs are created for the internal matrix objects. In our MPI drivers, process zero reads in the matrices and then starts the process to distribute them to the other processes.

   Error checking: If `pencil` or `fp` are `NULL`, an error message is printed and zero is returned.

2. `int Pencil_readFromFiles ( Pencil *pencil, char *fnA, char *fnB ) ;`

   This method reads the two `InpMtx` objects from two files. If `fnA` is "`none`", then A is not read. If `fnB` is "`none`", then B is not read.

   Error checking: If `pencil` or `fp` are `NULL`, an error message is printed and zero is returned.

3. `void Pencil_writeForHumanEye ( Pencil *pencil, FILE *fp ) ;`

   This method writes a `Pencil` object to a file in an easily readable format.

   Error checking: If `pencil` or `fp` are `NULL`, an error message is printed and zero is returned.

4. `void Pencil_writeStats ( Pencil *pencil, FILE *fp ) ;`

   This method writes statistics for `Pencil` object to a file.

   Error checking: If `pencil` or `fp` are `NULL`, an error message is printed and zero is returned.
   ```

Chapter 36

SemiImplMtx: Semi-Implicit Factorization

The `SemiImplMtx` object contains a semi-implicit representation of a sparse matrix factorization. Assume that the matrix A has been factored as $PAQ = LDU$, where L is unit lower triangular and U is unit upper triangular. Now consider PAQ (and so L, D and U) partitioned as follows.

$$\widehat{A} = PAQ = \begin{bmatrix} \widehat{A}_{1,1} & \widehat{A}_{1,2} \\ \widehat{A}_{2,1} & \widehat{A}_{2,2} \end{bmatrix} = \begin{bmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{bmatrix} \begin{bmatrix} D_{1,1} & 0 \\ 0 & D_{2,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & U_{2,2} \end{bmatrix}$$

After some algebra we can arrive at the following identities.

$$L_{2,1} = \widehat{A}_{2,1} D_{1,1}^{-1} \qquad \text{and} \qquad U_{1,2} = D_{1,1}^{-1} \widehat{A}_{1,2}$$

The straightforward solution of $AX = B$ can be done as follows, as we solve the permuted linear system $\widehat{A}\widehat{X} = \widehat{B}$, where $\widehat{X} = Q^T X$ and $\widehat{B} = PB$.

- solve $L_{1,1} Y_1 = \widehat{B}_1$.
- solve $L_{2,2} Y_2 = \widehat{B}_2 - L_{2,1} Y_1$.
- solve $D_{1,1} Z_1 = Y_1$.
- solve $D_{2,2} Z_2 = Y_2$.
- solve $U_{2,2} \widehat{X}_2 = Z_2$.
- solve $U_{1,1} \widehat{X}_1 = Z_1 - U_{1,2} Z_2$.

An equivalent process does not requires $L_{2,1}$ and $U_{1,2}$, but instead uses the $\widehat{A}_{1,2}$ and $\widehat{A}_{2,1}$ matrices.

- solve $L_{1,1} D_{1,1} U_{1,1} T_1 = \widehat{B}_1$.
- solve $L_{2,2} D_{2,2} U_{2,2} \widehat{X}_2 = \widehat{B}_2 - A_{2,1} T_1$.
- solve $L_{1,1} D_{1,1} U_{1,1} \widehat{X}_1 = \widehat{B}_1 - A_{1,2} \widehat{X}_2$.

In effect, we have traded multiplies with $L_{2,1}$ and $U_{1,2}$ for multiplies with $A_{1,2}$ and $A_{2,1}$ and two extra solves with $D_{1,1}$. In some cases this *semi-implicit* procedure (so named because $L_{2,1}$ and $U_{1,2}$ are stored in a semi-implicit form) can pay off — storage can be saved when the number of entries in $L_{2,1}$ and $U_{1,2}$ are larger than the number of entries in $A_{2,1}$ and $A_{1,2}$. The number of solve operations is reduced by $|L_{2,1}| + |U_{1,2}| - 2|D_{1,1}| - |A_{2,1}| - |A_{1,2}|$, where $|\cdot|$ denotes the number of nonzeroes in a matrix.

36.1 Data Structure

The SemiImplMtx structure has the following fields.

- **int neqns** : number of equations.
- **int type** : type of entries, SPOOLES_REAL or SPOOLES_COMPLEX.
- **int symmetryflag** : type of matrix symmetry, SPOOLES_SYMMETRIC, SPOOLES_HERMITIAN or SPOOLES_NONSYMMETRIC.
- **int ndomeqns** : number of equations in the domains, or (1,1) block.
- **int nschureqns** : number of equations in the Schur complement, or (2,2) block.
- **FrontMtx *domainMtx** : matrix object for $L_{1,1}$, $D_{1,1}$ and $U_{1,1}$.
- **FrontMtx *schurMtx** : matrix object for $L_{2,2}$, $D_{2,2}$ and $U_{2,2}$.
- **InpMtx *A21** : matrix object for $\widehat{A}_{2,1}$.
- **InpMtx *A12** : matrix object for $\widehat{A}_{1,2}$.
- **IV *domRowsIV** : object that holds the global ids of the rows in $\widehat{A}_{1,1}$.
- **IV *schurRowsIV** : object that holds the global ids of the rows in $\widehat{A}_{2,2}$.
- **IV *domColumnsIV** : object that holds the global ids of the columns in $\widehat{A}_{1,1}$.
- **IV *schurColumnsIV** : object that holds the global ids of the columns in $\widehat{A}_{2,2}$.

36.2 Prototypes and descriptions of SemiImplMtx methods

This section contains brief descriptions including prototypes of all methods that belong to the SemiImplMtx object.

36.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `SemiImplMtx * SemiImplMtx_new ( void ) ;`

   This method simply allocates storage for the SemiImplMtx structure and then sets the default fields by a call to SemiImplMtx_setDefaultFields().

2. `int SemiImplMtx_setDefaultFields ( SemiImplMtx *mtx ) ;`

   This method sets the structure's fields to default values: neqns = 0, type = SPOOLES_REAL, symmetryflag = SPOOLES_SYMMETRIC, ndomeqns = nschureqns = 0, and domainMtx, schurMtx, A21, A12, domRowsIV, schurRowsIV, domColumnsIV and schurColumnsIV are all set to NULL.

   Return codes: 1 means a normal return, -1 means mtx is NULL.

3. `int SemiImplMtx_clearData ( SemiImplMtx *mtx ) ;`

   This method releases all storage held by the object.

   Return codes: 1 means a normal return, -1 means mtx is NULL.

4. `int SemiImplMtx_free ( SemiImplMtx *mtx ) ;`

   This method releases all storage held by the object via a call to `SemiImplMtx_clearData()`, then free'd the storage for the object.

   Return codes: `1` means a normal return, `-1` means `mtx` is `NULL`.

36.2.2 Initialization Methods

1. `int SemiImplMtx_initFromFrontMtx ( SemiImplMtx *semimtx, FrontMtx *frontmtx, InpMtx *inpmtx, IV *frontmapIV, int msglvl, FILE *msgFile) ;`

   This initializer is used after the `FrontMtx` object for the factorization has been computed. The `frontmapIV` object defines which fronts map to domains and which to the Schur complement. If entry `J` of the `frontmapIV` object is zero, then front `J` belongs in the Schur complement, otherwise it belongs to the domains' matrix. The $A_{1,2}$ and $A_{2,1}$ (if nonsymmetric) matrices are extracted from the `InpMtx` object.

   The `semimtx` object removes submatrices from the `frontmtx` object, i.e., after the return of this method, the `frontmtx` no longer owns (and so cannot free) the submatrices from the $(1,1)$ and $(2,2)$ blocks. On return, the `frontmtx` object can safely be free'd without affecting the `semimtx` object.

   Return codes:

1	normal return	-4	`frontmapIV` is `NULL`
-1	`semimtx` is `NULL`	-5	`frontmapIV` is invalid
-2	`frontmtx` is `NULL`	-6	unable to create (1,1) front matrix
-3	`inpmtx` is `NULL`	-7	unable to create (2,2) front matrix

2. `int FrontMtx_initFromSubMtx ( FrontMtx *submtx, FrontMtx *frontmtx, IV *frontidsIV, IV *rowsIV, IV *colsIV, int msglvl, FILE *msgFile) ;`

   This initializer is used to initialize the `submtx` `FrontMtx` object from a global `FrontMtx` object, i.e., to initialize the `domainMtx` and `schurMtx` objects. The fronts of the `frontmtx` that will be included into the `submtx` object are given in the `frontidsIV` vector object. The `submtx` object extracts the submatrices from the `frontmtx` object, i.e., after the return of this method, the `frontmtx` no longer owns (and so cannot free) its submatrices. The `submtx` front matrix has *local* numbering, its global row ids are placed in `rowsIV` and its global column ids are placed in `colsIV`.

   Return codes:

1	normal return	-7	`colsIV` is `NULL`
-1	`submtx` is `NULL`	-8	unable to create the front tree
-2	`frontmtx` is `NULL`	-9	unable to create the symbolic factorization
-3	`frontmtx` is not in 2-d mode	-10	unable to create the column adjacency
-4	`frontidsIV` is `NULL`	-11	unable to create the row adjacency
-5	`frontidsIV` is invalid	-12	unable to create the upper block `IVL`
-6	`rowsIV` is `NULL`	-13	unable to create the lower block `IVL`

36.2.3 Solve Methods

1. `int SemiImplMtx_solve ( SemiImplMtx *mtx, DenseMtx *X, DenseMtx *B, SubMtxManager *mtxmanager, double cpus[], int msglvl, FILE *msgFile ) ;`

   This methods solves a linear system $(L+I)D(I+U)X = B$, $(U^T+I)D(I+U)X = B$ or $(U^H+I)D(I+U)X = B$, where `X` and `B` are `DenseMtx` objects. `mtxmanager` is an object to handle the working `SubMtx` objects during the solve. One can have `X` and `B` point to the same object, for entries are read from `B` and written to `X`. On return, the `cpus[]` vector contains the following information.

`cpus[0]`	initialize working matrices	`cpus[5]`	compute domains' right hand side
`cpus[1]`	load right hand side	`cpus[6]`	second solve with domains
`cpus[2]`	first solve with domains	`cpus[7]`	store solution
`cpus[3]`	compute Schur right hand side	`cpus[8]`	miscellaneous time
`cpus[4]`	Schur solve	`cpus[9]`	total time

Return codes:

1	normal return	-3	`B` is `NULL`
-1	`mtx` is `NULL`	-4	`mtxmanager` is `NULL`
-2	`X` is `NULL`	-5	`cpus` is `NULL`

36.2.4 Utility methods

1. `int SemiImplMtx_stats ( SemiImplMtx *mtx, int stats[] ) ;`

   This method fills the `stats[]` vector with some statistics.

`stats[0]`	# of equations	`stats[7]`	# of entries in $D_{2,2}$
`stats[1]`	# of equations in the $(1,1)$ block	`stats[8]`	# of entries in $U_{2,2}$
`stats[2]`	# of equations in the $(2,2)$ block	`stats[9]`	# of entries in $A_{1,2}$
`stats[3]`	# of entries in $L_{1,1}$	`stats[10]`	# of entries in $A_{2,1}$
`stats[4]`	# of entries in $D_{1,1}$	`stats[11]`	total # of entries
`stats[5]`	# of entries in $U_{1,1}$	`stats[12]`	# of operations for a solve
`stats[6]`	# of entries in $L_{2,2}$		

   Return values:

   `1` for a normal return, `-1` if `mtx` is `NULL`, `-2` if `stats` is `NULL`.

36.2.5 IO methods

1. `int SemiImplMtx_writeForHumanEye ( SemiImplMtx *mtx, FILE *fp ) ;`

   This method writes out a `SemiImplMtx` object to a file in a human readable format.

   Return codes:

1	normal return	-3	`symmetryflag` is invalid
-1	`mtx` is `NULL`	-4	`fp` is `NULL`
-2	`type` is invalid		

36.3 Driver programs for the `SemiImplMtx` object

This section contains brief descriptions of the driver programs.

1. `testGrid msglvl msgFile n1 n2 n3 maxzeros maxsize seed type symmetryflag sparsityflag pivotingflag tau droptol nrhs depth`

   This driver program tests the `SemiImplMtx` creation and solve methods for a matrix from a regular 2-D or 3-D grid. The matrix can be real or complex and is loaded with random entries. The linear system $AX = B$ is solved as follows.

   - First A is factored, and a `FrontMtx` object is created to hold the factorization.

- The system is solved using the `FrontMtx` object.
- A `SemiImplMtx` matrix object is constructed from the `FrontMtx` object and A.
- The system is solved using the `SemiImplMtx` object.

Various statistics and CPU timings are written to the message file to compare the two solution processes. Use the `do_grid` shell script for testing.

- The `msglvl` parameter determines the amount of output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- `n1` is the number of grid points in the first direction.
- `n2` is the number of grid points in the second direction.
- `n3` is the number of grid points in the third direction.
- `maxzeros` is the maximum number of zeroes to place into a front.
- `maxsize` is the maximum number of internal rows and columns in a front.
- `type` must be either `SPOOLES_REAL` or `SPOOLES_COMPLEX`.
- `symmetryflag` must be either `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or
- `sparsityflag` must be either `FRONTMTX_DENSE_FRONTS` or `FRONTMTX_SPARSE_FRONTS`.
- `pivotingflag` must be either `SPOOLES_PIVOTING`, `SPOOLES_NO_PIVOTING` or
- `tau` is used when pivoting is enabled, it is an upper bound on the magnitude of the entries in L and U.
- `droptol` is used when an approximate factorization is called for, (i.e., when `sparsityflag` is `FRONTMTX_SPARSE_FRONTS`). It is a lower bound on the magnitude of the entries in L and U that are stored and used in computations.
- `nrhs` is the number of right hand sides.
- `depth` is used to specify the schur complement. It is based on separators, not on fronts. (Recall that large separators can be split into smaller fronts for efficiency reasons.) All fronts found in separators lower than `depth` in depth (the top level separator has depth zero) belong in domains.

2. `testSimple msglvl msgFile inFrontMtxFile inInpMtxFile inIVfile`

   This driver program is used to construct a `SemiImplMtx` object. It reads in a `FrontMtx` and `InpMtx` from files. It also reads in an `IV` object that specifies whether a front is to be in the domains (the (1,1) block) or the Schur complement (the (2,2) block). It then creates the `SemiImplMtx` object and writes it to the message file. Use the `do_simple` script file for testing.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `FrontMtx` object is read from the `inFrontMtxFile` file, which must be of the form `*.frontmtxf` or `*.frontmtxb`.
   - The `InpMtx` object is read from the `inInpMtxFile` file, which must be of the form `*.inpmtxf` or `*.inpmtxb`.
   - The map vector `IV` object is read from the `inIVfile` file, which must be of the form `*.ivf` or `*.ivb`.

Chapter 37

SubMtx: Submatrix object

The `SubMtx` object was created to hold the data for and operate with a submatrix of a sparse matrix. The entries in a submatrix can be either double precision real or complex.

For example, the lower and upper triangular matrices L and U that are created during the factorization are stored as submatrices, e.g., $L_{I,I}$ and $L_{J,I}$ where I and J are index sets. To be more precise, I and J are index sets associated with fronts `I` and `J`. We do not necessarily represent $L_{J,I}$, because some of the rows in the submatrix may be zero. Instead we keep $L_{\partial I \cap J,I}$, where $\partial I \cap J$ are precisely those rows that may have nonzeros. The situation is similar for U where we keep $U_{I,\partial I \cap J}$.

The submatrices for L and U may be dense or sparse. (A direct factorization typically generates dense submatrices while a drop tolerance factorization produces sparse submatrices.) We also use `SubMtx` objects to represent submatrices of the D matrix, where D is either diagonal or has 1×1 and 2×2 blocks on its diagonal. In the latter case, we support $D_{I,I}$ to be either real symmetric, complex symmetric or complex Hermitian.

The `SubMtx` object has the following attributes.

- A `SubMtx` object has a row id and column id to identify itself within the context of a larger block matrix.
- Each row and column of the block matrix corresponds to a certain index set. A `SubMtx` object associated with block row `J` and block column `I` has row indices J and column indices I.
- Matrix entries stored in one of the following ways.
  - dense by rows, i.e., dense and row major
  - dense by columns, i.e., dense and column major
  - sparse using dense subrows
  - sparse using dense subcolumns
  - sparse using sparse rows
  - sparse using sparse columns
  - sparse using $(i, j, a_{i,j})$ triples
  - a diagonal matrix
  - a block diagonal symmetric matrix where the blocks are 1×1 or 2×2, used in the symmetric indefinite factorization.
  - a block diagonal Hermitian matrix where the blocks are 1×1 or 2×2, used in the hermitian indefinite factorization.

- The `SubMtx` object can be self-contained, in the sense that its structure contains a `DV` object that manages a contiguous vector of workspace that is used to store all information about the `SubMtx` object — its scalar parameters, any integer index or dimension information, and all matrix entries. In a distributed environment, this allows a `SubMtx` object to be sent between processors as one message, no copying to an internal buffer is needed, nor any custom data type needs to be defined as for MPI. In an out-of-core environment, a `SubMtx` object can be read from or written to a file by a single operation.

The `SubMtx` object is a superset of the `DenseMtx` object in terms of data structure and functionality. If we were working in a language that supports inheritance, `SubMtx` would be an abstract class and `DenseMtx` would be a subclass where entries would be stored by dense rows or columns. At some point in the future we may deprecate the `DenseMtx` object in this library, replacing it with the `SubMtx` object.

Because the `SubMtx` object wears so many hats, i.e., it supports nine different storage formats, it has to be flexible in how it responds to its environment. For example, how we access the data is different depending on which storage format. Instead of accessing structure fields directly, e.g., let `mtx->entries` point to the start of the matrix entries, we follow a convention that *instance* methods return information. For example, the function call

```
SubMtx_columnIndices(mtx, &nrow, &rowind) ;
```

is an instance method that fills `nrow` with the number of rows and `rowind` with the first location of the row indices. A more complex example is for the sparse storage by rows format,

```
SubMtx_sparseRowsInfo(mtx, &nrow, &nent, &sizes, &indices, &entries) ;
```

where the number of rows and entries are returned in `nrow` and `nent`, the number of nonzero entries in each row is contained in `sizes[]`, and the column indices and nonzero entries are found in `indices[]` and `entries[]`, respectively. This convention of using instance methods to return information is better than using explicit structure fields. For example, if we want to extend the object by allowing another storage format, we do not need to increase the size of the structure at all — it is only necessary to provide one or more instance methods to return the new information.

37.1 Data Structure

The `SubMtx` structure has the following fields.

- `int type` : type of entries.
  - `SPOOLES_REAL` : double precision real entries.
  - `SPOOLES_COMPLEX` : double precision complex entries.
- `int mode` : storage mode.
  - `SUBMTX_DENSE_ROWS` : dense, storage by rows.
  - `SUBMTX_DENSE_COLUMNS` : dense, storage by columns.
  - `SUBMTX_SPARSE_ROWS` : sparse, storage by rows.
  - `SUBMTX_SPARSE_COLUMNS` : sparse, storage by columns.
  - `SUBMTX_SPARSE_TRIPLES` : sparse, storage by $(i, j, a_{i,j})$ triples.
  - `SUBMTX_DENSE_SUBROWS` : sparse, storage by dense subrows.
  - `SUBMTX_DENSE_SUBCOLUMNS` : sparse, storage by dense subcolumns.

- `SUBMTX_DIAGONAL` : a diagonal matrix.
- `SUBMTX_BLOCK_DIAGONAL_SYM` : a symmetric block diagonal matrix with 1×1 and 2×2 blocks.
- `SUBMTX_BLOCK_DIAGONAL_HERM` : a hermitian block diagonal matrix with 1×1 and 2×2 blocks.

- `int rowid` : object's row id, default value is `-1`.
- `int colid` : object's column id, default value is `-1`.
- `int nrow` : number of rows
- `int ncol` : number of columns
- `int nent` : number of stored matrix entries.
- `DV wrkDV` : object that manages the owned working storage.
- `SubMtx *next` : link to a next object in a singly linked list.

One can query the type of the object using these simple macros.

- `SUBMTX_IS_REAL(mtx)` is `1` if `mtx` has real entries and `0` otherwise.
- `SUBMTX_IS_COMPLEX(mtx)` is `1` if `mtx` has complex entries and `0` otherwise.
- `SUBMTX_IS_DENSE_ROWS(mtx)` is `1` if `mtx` has dense rows as its storage format, and `0` otherwise.
- `SUBMTX_IS_DENSE_COLUMNS(mtx)` is `1` if `mtx` has dense columns as its storage format, and `0` otherwise.
- `SUBMTX_IS_SPARSE_ROWS(mtx)` is `1` if `mtx` has sparse rows as its storage format, and `0` otherwise.
- `SUBMTX_IS_SPARSE_COLUMNS(mtx)` is `1` if `mtx` has sparse columns as its storage format, and `0` otherwise.
- `SUBMTX_IS_SPARSE_TRIPLES(mtx)` is `1` if `mtx` has sparse triples as its storage format, `0` otherwise.
- `SUBMTX_IS_DENSE_SUBROWS(mtx)` is `1` if `mtx` has dense subrows as its storage format, `0` otherwise.
- `SUBMTX_IS_DENSE_SUBCOLUMNS(mtx)` is `1` if `mtx` has dense subcolumns as its storage format, `0` otherwise.
- `SUBMTX_IS_DIAGONAL(mtx)` is `1` if `mtx` is diagonal, `0` otherwise.
- `SUBMTX_IS_BLOCK_DIAGONAL_SYM(mtx)` is `1` if `mtx` is block diagonal and symmetric, `0` otherwise.
- `SUBMTX_IS_BLOCK_DIAGONAL_HERM(mtx)` is `1` if `mtx` is block diagonal and hermitian, `0` otherwise.

37.2 Prototypes and descriptions of `SubMtx` methods

This section contains brief descriptions including prototypes of all methods that belong to the `SubMtx` object.

37.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `SubMtx * SubMtx_new ( void ) ;`

   This method simply allocates storage for the `SubMtx` structure and then sets the default fields by a call to `SubMtx_setDefaultFields()`.

2. `void SubMtx_setDefaultFields ( SubMtx *mtx ) ;`

   The structure's fields are set to default values: `type` = `SPOOLES_REAL`, `mode` = `DENSEMTX_DENSE_COLUMNS`, `rowid` = `colid` = `-1`, `type` = `nrow` = `ncol` = `nent` = 0 and `next` = `NULL` . The `wrkDV` object has its default fields set via a call to `DV_setDefaultFields()`.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `void SubMtx_clearData ( SubMtx *mtx ) ;`

   This method clears the object and free's any owned data by invoking the `_clearData()` methods for its internal `DV` object. There is a concluding call to `SubMtx_setDefaultFields()`.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. `void SubMtx_free ( SubMtx *mtx ) ;`

   This method releases any storage by a call to `SubMtx_clearData()` and then frees the space for `mtx`.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

37.2.2 Instance methods

1. `void SubMtx_ids ( SubMtx *mtx, int *prowid, int *pcolid ) ;`

   This method fills `*prowid` with the row id and `*pcolid` with the column id of the object.

   Error checking: If `mtx`, `prowid` or `pcolid` is `NULL`, an error message is printed and the program exits.

2. `void SubMtx_setIds ( SubMtx *mtx, int rowid, int colid ) ;`

   This method sets the row and column id's of the matrix.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `void SubMtx_dimensions ( SubMtx *mtx, int *pnrow, int *pncol, int *pnent ) ;`

   This method fills `*pnrow`, `*pncol` and `*pnent` with the number of rows, columns and matrix entries, respectively.

   Error checking: If `mtx`, `pnrow` or `pncol` is `NULL`, an error message is printed and the program exits.

4. `void SubMtx_rowIndices ( SubMtx *mtx, int *pnrow, **prowind ) ;`

   This method fills `*pnrow` with the number of rows. If `prowind` is not `NULL`, `*prowind` is filled with a pointer to the row indices.

   Error checking: If `mtx` or `pnrow` is `NULL`, an error message is printed and the program exits.

5. `void SubMtx_columnIndices ( SubMtx *mtx, int *pncol, **colind ) ;`

   This method fills `*pncol` with the number of columns. If `pcolind` is not `NULL`, `*pcolind` is filled with a pointer to the column indices.

   Error checking: If `mtx`, `pncol` or `pcolind` is `NULL`, an error message is printed and the program exits.

6. `void SubMtx_denseInfo ( SubMtx *mtx, int *pnrow, int *pncol, int *pinc1, int *pinc2, double **pentries ) ;`

   This method is used when the storage mode is dense rows or columns. It fills `*pnrow` with the number of rows, `*pncol` with the number of columns, `*pinc1` with the row increment, `*pinc2` with the column increment, and `*pentries` with the base address of entries vector.

   Error checking: If `mtx`, `pnrow`, `pncol`, `pinc1`, `pinc2` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_DENSE_ROWS` or `SUBMTX_DENSE_COLUMNS`, an error message is printed and the program exits.

7. `void SubMtx_sparseRowsInfo ( SubMtx *mtx, int *pnrow, int *pnent, int **psizes, int **pindices, double **pentries ) ;`

   This method is used when the storage mode is sparse rows. It fills `*pnrow` with the number of rows, `*pnent` with the number of matrix entries, `*psizes` with the base address of the `sizes[nrow]` vector that contains the number of entries in each row, `*indices` with the base address of the `indices[nent]` vector that contains the column index for each entry, and `*pentries` with the base address of `entries[nent]` vector. The indices and entries for the rows are stored contiguously.

   Error checking: If `mtx`, `pnrow`, `pnent`, `psizes`, `pindices` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_SPARSE_ROWS`, an error message is printed and the program exits.

8. `void SubMtx_sparseColumnsInfo ( SubMtx *mtx, int *pncol, int *pnent, int **psizes, int **pindices, double **pentries ) ;`

   This method is used when the storage mode is sparse columns. It fills `*pncol` with the number of columns, `*pnent` with the number of matrix entries, `*psizes` with the base address of the `sizes[ncol]` vector that contains the number of entries in each column, `*indices` with the base address of the `indices[nent]` vector that contains the row index for each entry, and `*pentries` with the base address of `entries[nent]` vector. The indices and entries for the columns are stored contiguously.

   Error checking: If `mtx`, `pncol`, `pnent`, `psizes`, `pindices` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_SPARSE_COLUMNS`, an error message is printed and the program exits.

9. `void SubMtx_sparseTriplesInfo ( SubMtx *mtx, int *pnent, int **prowids, int **pcolids, double **pentries ) ;`

   This method is used when the storage mode is sparse triples. It fills `*pnent` with the number of matrix entries, `*prowids` with the base address of the `rowids[nent]` vector that contains the row id of each entry, `*pcolids` with the base address of the `colids[nent]` vector that contains the column id of each entry, and `*pentries` with the base address of `entries[nent]` vector.

   Error checking: If `mtx`, `pnent`, `prowids`, `pcolids` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_SPARSE_TRIPLES`, an error message is printed and the program exits.

10. `void SubMtx_denseSubrowsInfo ( SubMtx *mtx, int *pnrow, int *pnent, int **pfirstlocs, int **plastlocs, double **pentries ) ;`

    This method is used when the storage mode is dense subrows. It fills `*pnrow` with the number of rows, `*pnent` with the number of matrix entries, `*pfirstlocs` with the base address of the `firstlocs[nrow]` vector, `*plastlocs` with the base address of the `lastlocs[nrow]` vector, and `*pentries` with the base address of `entries[nent]` vector. For row `irow`, the nonzero entries are found in columns `[firstlocs[irow],lastlocs[irow]]` when `firstlocs[irow]` ≥ 0 and `firstlocs[irow]` $\leq$ `lastlocs[irow]`. The entries for the rows are stored contiguously.

    Error checking: If `mtx`, `pnrow`, `pnent`, `pfirstlocs`, `plastlocs` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_DENSE_SUBROWS`, an error message is printed and the program exits.

11. `void SubMtx_denseSubcolumnsInfo ( SubMtx *mtx, int *pncol, int *pnent,`
    `int **pfirstlocs, int **plastlocs, double **pentries ) ;`

    This method is used when the storage mode is dense subcolumns. It fills `*pncol` with the number of columns, `*pnent` with the number of matrix entries, `*pfirstlocs` with the base address of the `firstlocs[ncol]` vector, `*plastlocs` with the base address of the `lastlocs[ncol]` vector, and `*pentries` with the base address of `entries[nent]` vector. For column `jcol`, the nonzero entries are found in rows `[firstlocs[jcol],lastlocs[jcol]]` when `firstlocs[jcol]` ≥ 0 and `firstlocs[jcol]` $\leq$ `lastlocs[jcol]`. The entries for the columns are stored contiguously.

    Error checking: If `mtx`, `pnrow`, `pnent`, `pfirstlocs`, `plastlocs` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_DENSE_SUBCOLUMNS`, an error message is printed and the program exits.

12. `void SubMtx_diagonalInfo ( SubMtx *mtx, int *pncol, double **pentries ) ;`

    This method is used when the storage mode is diagonal. It fills `*pncol` with the number of columns and `*pentries` with the base address of `entries[]` vector.

    Error checking: If `mtx`, `pncol` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_DIAGONAL`, an error message is printed and the program exits.

13. `void SubMtx_blockDiagonalInfo ( SubMtx *mtx, int *pncol, int *pnent,`
    `int **ppivotsizes, double **pentries ) ;`

    This method is used when the storage mode is block diagonal. It fills `*pncol` with the number of columns, `*pnent` with the number of entries, `*ppivotsizes` with the base address of the pivot sizes vector, and `*pentries` with the base address of `entries[]` vector.

    Error checking: If `mtx`, `pncol`, `pnent`, `ppivotsizes` or `pentries` is `NULL`, or if the matrix type is not `SUBMTX_BLOCK_DIAGONAL_SYM`, or `SUBMTX_BLOCK_DIAGONAL_HERM`, an error message is printed and the program exits.

14. `int  SubMtx_realEntry ( SubMtx *mtx, int irow, int jcol, double *pValue ) ;`

    This method fill `*pValue` with the entry in row `irow` and column`jcol`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq$ `irow` $\leq$ `nrow` and $0 \leq$ `jcol` $\leq$ `ncol`. If the `(irow,jcol)` entry is present, the return value is the offset from the start of the `entries` vector. Otherwise, -1 is returned.

    Error checking: If `mtx` or `pValue` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

15. `int  SubMtx_complexEntry ( SubMtx *mtx, int irow, int jcol,`
    `double *pReal, double *pImag ) ;`

    This method fill `*pReal` with the real part and `*pImag` with the imaginary part of the the entry in row `irow` and column`jcol`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq$ `irow` $\leq$ `nrow` and $0 \leq$ `jcol` $\leq$ `ncol`. If the `(irow,jcol)` entry is present, the return value is the offset from the start of the `entries` vector. (The offset is in terms of complex entries, not double entries.) Otherwise, -1 is returned.

    Error checking: If `mtx`, `pReal` or `pImag` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

16. `void  SubMtx_locationOfRealEntry ( SubMtx *mtx, int irow, int jcol,`
    `double **ppValue ) ;`

    If the `(irow,jcol)` entry is present, this method fills `*ppValue` with a pointer to the entry in row `irow` and column`jcol`. Otherwise, `*ppValue` is set to `NULL`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq$ `irow` $\leq$ `nrow` and $0 \leq$ `jcol` $\leq$ `ncol`.

    Error checking: If `mtx` or `ppValue` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

17. `void  SubMtx_locationOfComplexEntry ( SubMtx *mtx, int irow, int jcol, double **ppReal, double **ppImag ) ;`

    If the `(irow,jcol)` entry is present, this method fills `*ppReal` with a pointer to the real part and `*ppImag` with a pointer to the imaginary part of the the entry in row `irow` and column`jcol`. Otherwise, `*ppImag` and `*ppReal` are set to `NULL`. Note, `irow` and `jcol` are *local* indices, i.e., $0 \leq$ `irow` $\leq$ `nrow` and $0 \leq$ `jcol` $\leq$ `ncol`.

    Error checking: If `mtx`, `ppReal` or `ppImag` is `NULL`, or if `irow` or `jcol` is out of range, an error message is printed and the program exits.

37.2.3 Initialization methods

There are three initializer methods.

1. `void SubMtx_init( SubMtx *mtx, int type, int mode, int rowid, int colid, int nrow, int ncol, int nent ) ;`

   This is the initializer method used when the `SubMtx` object is to use its workspace to store indices and entries. The number of bytes required in the workspace is computed, the workspace is resized if necessary, the scalar fields are set, and the row and column indices are set to `[0,nrow)` and `[0,ncol)`, respectively.

   Error checking: If `mtx` is `NULL`, or if `nrow`, `ncol`, `inc1` or `inc2` is less than or equal to zero, or if neither `inc1` nor `inc2` are `1`, an error message is printed and the program exits.

2. `void SubMtx_initFromBuffer ( SubMtx *mtx ) ;`

   This method initializes the object using information present in the workspace buffer. This method is used to initialize the `SubMtx` object when it has been received as an MPI message.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `void SubMtx_initRandom ( SubMtx *mtx, int type, int mode, int rowid, int colid, int nrow, int ncol, int nent, int seed ) ;`

   This is used to initialize an object to have random entries and (possibly) random structure. The object is first initialized via a call to `SubMtx_init()`. Its matrix entries are then filled with random numbers. If the matrix is sparse, its sparsity pattern is sparse and random, using `nent` when applicable. The row and column indices are ascending starting from zero.

   Error checking: If `mtx` is `NULL`, or if `nrow`, `ncol`, `inc1` or `inc2` is less than or equal to zero, or if neither `inc1` nor `inc2` are `1`, an error message is printed and the program exits.

4. `void SubMtx_initRandomLowerTriangle ( SubMtx *mtx, int type, int mode, int rowid, int colid, int nrow, int ncol, int nent, int seed, int strict ) ;`
   `void SubMtx_initRandomUpperTriangle ( SubMtx *mtx, int type, int mode, int rowid, int colid, int nrow, int ncol, int nent, int seed, int strict ) ;`

   This is used to initialize an object to have random entries and (possibly) random structure. The matrix type may not be diagonal, block diagonal, or triples. If `strict = 1`, the matrix will be strict lower or upper triangular. The object is first initialized via a call to `SubMtx_init()`. Its matrix entries are then filled with random numbers. If the matrix is sparse, its sparsity pattern is sparse and random, using `nent` when applicable. The row and column indices are ascending starting from zero.

   Error checking: If `mtx` is `NULL`, or if `nrow`, `ncol`, `inc1` or `inc2` is less than or equal to zero, or if neither `inc1` nor `inc2` are `1`, an error message is printed and the program exits.

37.2.4 Vector scaling methods

These methods are used during the factorization when we compute products of the form $-U^T DU$, $-U^H DU$ and $-LDU$.

1. ```
   void SubMtx_scale1vec ( SubMtx *mtxD, double y0[], double x0[] ) ;
   void SubMtx_scale2vec ( SubMtx *mtxD, double y0[], double y1[],
                           double x0[], double x1[] ) ;
   void SubMtx_scale3vec ( SubMtx *mtxD, double y0[], double y1[], double y2[],
                           double x0[], double x1[], double x2[]
   ```
   These methods compute one of the following
   $$y_0 = Dx_0, \qquad \left[\begin{array}{cc} y_0 & y_1 \end{array}\right] = D \left[\begin{array}{cc} x_0 & x_1 \end{array}\right] \qquad \text{or} \qquad \left[\begin{array}{ccc} y_0 & y_1 & y2 \end{array}\right] = D \left[\begin{array}{ccc} x_0 & x_1 & x2 \end{array}\right]$$
   where D is stored in the `SubMtx` object `mtxD`, and the y_0, y_1, y_2, x_0, x_1 and x_2 vectors are stored as simple real or complex vectors. This method is only used when `mtxD` is diagonal or block diagonal (symmetric or Hermitian).

   Error checking: If `mtxD`, `y0`, `y1`, `y2`, `x0`, `x1` or `x2` is `NULL`, an error message is printed and the program exits.

37.2.5 Solve methods

These methods are used during the forward and backward solves.

1. `void SubMtx_solve ( SubMtx *mtxA, SubMtx *mtxB ) ;`

   This method is used to solve $(I + A)X = B$ (if A is strict lower or upper triangular) or $AX = B$ (if A is diagonal or block diagonal). The solution X overwrites B, and `mtxB` must have dense columns. If A is strict lower triangular, then `mtxA` must have dense subrows or sparse rows. If A is strict upper triangular, then `mtxA` must have dense subcolumns or sparse columns.

   Error checking: If `mtxA` or `mtxB` is `NULL`, an error message is printed and the program exits.

2. `void SubMtx_solveH ( SubMtx *mtxA, SubMtx *mtxB ) ;`

   This method is used to solve $(I + A^H)X = B$, where A is strict lower or upper triangular. The solution X overwrites B, and `mtxB` must have dense columns. If A is strict lower triangular, then `mtxA` must have dense subrows or sparse rows. If A is strict upper triangular, then `mtxA` must have dense subcolumns or sparse columns.

   Error checking: If `mtxA` or `mtxB` is `NULL`, an error message is printed and the program exits.

3. `void SubMtx_solveT ( SubMtx *mtxA, SubMtx *mtxB ) ;`

   This method is used to solve $(I + A^T)X = B$, where A is strict lower or upper triangular. The solution X overwrites B, and `mtxB` must have dense columns. If A is strict lower triangular, then `mtxA` must have dense subrows or sparse rows. If A is strict upper triangular, then `mtxA` must have dense subcolumns or sparse columns.

   Error checking: If `mtxA` or `mtxB` is `NULL`, an error message is printed and the program exits.

4. `void SubMtx_solveupd ( SubMtx *mtxY, SubMtx *mtxA, SubMtx *mtxX ) ;`

   This method is used to update $Y := Y - A * X$, where A has dense or sparse rows or columns. `mtxY` and `mtxX` must have dense columns.

   Error checking: If `mtxY`, `mtxA` or `mtxX` is `NULL`, an error message is printed and the program exits.
   ```

5. `void SubMtx_solveupdH ( SubMtx *mtxY, SubMtx *mtxA, SubMtx *mtxX ) ;`

   This method is used to update $Y := Y - A^H * X$, where A has dense or sparse rows or columns. `mtxY` and `mtxX` must have dense columns.

   Error checking: If `mtxY`, `mtxA` or `mtxX` is `NULL`, an error message is printed and the program exits.

6. `void SubMtx_solveupdT ( SubMtx *mtxY, SubMtx *mtxA, SubMtx *mtxX ) ;`

   This method is used to update $Y := Y - A^T * X$, where A has dense or sparse rows or columns. `mtxY` and `mtxX` must have dense columns.

   Error checking: If `mtxY`, `mtxA` or `mtxX` is `NULL`, an error message is printed and the program exits.

37.2.6 Utility methods

1. `int SubMtx_nbytesNeeded ( int type, int mode, int nrow, int ncol, int nent ) ;`

   This method returns the number of bytes required to store the object's information in its buffer.

   Error checking: If `nrow` or `ncol` is less than or equal to zero, or if `nent` is less than to zero, or if `type` is invalid, an error message is printed and the program exits.

2. `int SubMtx_nbytesInUse ( SubMtx *mtx ) ;`

   This method returns the actual number of bytes that are used in the workspace owned by this object.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

3. `int SubMtx_nbytesInWorkspace ( SubMtx *mtx ) ;`

   This method returns the number of bytes in the workspace owned by this object.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

4. `void SubMtx_setNbytesInWorkspace ( SubMtx *mtx, int nbytes ) ;`

   This method sets the number of bytes in the workspace of this object. If `nbytes` is less than the present number of bytes, the workspace is not resized.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

5. `void * SubMtx_workspace ( SubMtx *mtx ) ;`

   This method returns a pointer to the base address of the workspace.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

6. ```
   void SubMtx_setFields( SubMtx *mtx, int type, int mode, int rowid,
                          int colid, int nrow, int ncol, int nent ) ;
   ```

   This method sets the scalar fields.

   Error checking: If `mtx` is `NULL`, or if `nrow`, `ncol`, `nent` is less than or equal to zero, or if `type` or `mode` is invalid, an error message is printed and the program exits.

7. `void SubMtx_sortRowsUp ( SubMtx *mtx ) ;`

   This method sort the rows so the row ids are in ascending order.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

8. `void SubMtx_sortColumnsUp ( SubMtx *mtx ) ;`

   This method sort the rows so the column ids are in ascending order.

   Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.
   ```

9. `void SubMtx_fillRowDV ( SubMtx *mtx, int irow, DV *rowDV ) ;`

   This method is used for real submatrices. It copies the entries in row `irow` of the `mtx` object into the `rowDV` vector object.

   Error checking: If `mtx` or `rowDV` is `NULL`, or if `irow` is out of range, an error message is printed and the program exits.

10. `void SubMtx_fillColumnDV ( SubMtx *mtx, int jcol, DV *rowDV ) ;`

    This method is used for real submatrices. It copies the entries in column `jcol` of the `mtx` object into the `colDV` vector object.

    Error checking: If `mtx` or `colDV` is `NULL`, or if `jcol` is out of range, an error message is printed and the program exits.

11. `void SubMtx_fillRowZV ( SubMtx *mtx, int irow, ZV *rowZV ) ;`

    This method is used for complex submatrices. It copies the entries in row `irow` of the `mtx` object into the `rowZV` vector object.

    Error checking: If `mtx` or `rowZV` is `NULL`, or if `irow` is out of range, an error message is printed and the program exits.

12. `void SubMtx_fillColumnZV ( SubMtx *mtx, int jcol, ZV *rowZV ) ;`

    This method is used for complex submatrices. It copies the entries in column `jcol` of the `mtx` object into the `colZV` vector object.

    Error checking: If `mtx` or `colZV` is `NULL`, or if `jcol` is out of range, an error message is printed and the program exits.

13. `double SubMtx_maxabs ( SubMtx *mtx ) ;`

    This method returns the magnitude of the element in the matrix with the largest magnitude.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

14. `void SubMtx_zero ( SubMtx *mtx ) ;`

    This method zeros the entries of the submatrix.

    Error checking: If `mtx` is `NULL`, an error message is printed and the program exits.

37.2.7 IO methods

The file structure of a `SubMtx` object is exactly that of its internal workspace buffer. See the source code for more details.

1. `int SubMtx_readFromFile ( SubMtx *mtx, char *fn ) ;`

   This method reads a `SubMtx` object from a file. It tries to open the file and if it is successful, it then calls `SubMtx_readFromFormattedFile()` or `SubMtx_readFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `mtx` or `fn` are `NULL`, or if `fn` is not of the form `*.mtxf` (for a formatted file) or `*.mtxb` (for a binary file), an error message is printed and the method returns zero.

2. `int SubMtx_readFromFormattedFile ( SubMtx *mtx, FILE *fp ) ;`

   This method reads in a `SubMtx` object from a formatted file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fscanf`, zero is returned. Note, if the mtxutation vectors are one-based (as for Fortran), they are converted to zero-based vectors.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

3. `int SubMtx_readFromBinaryFile ( SubMtx *mtx, FILE *fp ) ;`

   This method reads in a `SubMtx` object from a binary file. If there are no errors in reading the data, the value `1` is returned. If an IO error is encountered from `fread`, zero is returned. Note, if the mtxutation vectors are one-based (as for Fortran), they are converted to zero-based vectors.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

4. `int SubMtx_writeToFile ( SubMtx *mtx, char *fn ) ;`

   This method writes a `SubMtx` object to a file. It tries to open the file and if it is successful, it then calls `SubMtx_writeFromFormattedFile()` or `SubMtx_writeFromBinaryFile()`, closes the file and returns the value returned from the called routine.

   Error checking: If `mtx` or `fn` are `NULL`, or if `fn` is not of the form `*.mtxf` (for a formatted file) or `*.mtxb` (for a binary file), an error message is printed and the method returns zero.

5. `int SubMtx_writeToFormattedFile ( SubMtx *mtx, FILE *fp ) ;`

   This method writes out a `SubMtx` object to a formatted file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fprintf`, zero is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

6. `int SubMtx_writeToBinaryFile ( SubMtx *mtx, FILE *fp ) ;`

   This method writes out a `SubMtx` object to a binary file. If there are no errors in writing the data, the value `1` is returned. If an IO error is encountered from `fwrite`, zero is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

7. `int SubMtx_writeForHumanEye ( SubMtx *mtx, FILE *fp ) ;`

   This method writes out a `SubMtx` object to a file in a human readable format. The method `SubMtx_writeStats()` is called to write out the header and statistics. The value `1` is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

8. `int SubMtx_writeStats ( SubMtx *mtx, FILE *fp ) ;`

   This method writes out a header and statistics to a file. The value `1` is returned.

   Error checking: If `mtx` or `fp` are `NULL`, an error message is printed and zero is returned.

9. `void SubMtx_writeForMatlab ( SubMtx *mtx, char *mtxname, FILE *fp ) ;`

   This method writes out a `SubMtx` object to a file in a Matlab format. A sample line is

   ```
   a(10,5) =  -1.550328201511e-01 +   1.848033378871e+00*i ;
   ```

   for complex matrices, or

   ```
   a(10,5) =  -1.550328201511e-01 ;
   ```

   for real matrices, where mtxname = `"a"`. The matrix indices come from the `rowind[]` and `colind[]` vectors, and are incremented by one to follow the Matlab and FORTRAN convention.

   Error checking: If `mtx`, `mtxname` or `fp` are `NULL`, an error message is printed and zero is returned.

37.3 Driver programs for the `SubMtx` object

1. `testIO msglvl msgFile inFile outFile`

   This driver program reads in a `SubMtx` object from `inFile` and writes out the object to `outFile`

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `SubMtx` object is written to the message file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inFile` parameter is the input file for the `SubMtx` object. It must be of the form `*.submtxf` or `*.submtxb`. The `SubMtx` object is read from the file via the `SubMtx_readFromFile()` method.
   - The `outFile` parameter is the output file for the `SubMtx` object. It must be of the form `*.submtxf` or `*.submtxb`. The `SubMtx` object is written to the file via the `SubMtx_writeToFile()` method.

2. `test_scalevec msglvl msgFile type mode nrowA seed`

   This driver program tests the `SubMtx_scalevec{1,2,3}()` methods. Use the script file `do_scalevec` for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter must be one of 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
   - The `mode` parameter must be one of 7 (`SUBMTX_DIAGONAL`), 8 (`SUBMTX_BLOCK_DIAGONAL_SYM`) or 9 (`SUBMTX_BLOCK_DIAGONAL_HERM`).
   - The `nrowA` parameter is the number of rows in the matrix.
   - The `seed` parameter is a random number seed.

3. `test_solve msglvl msgFile type mode nrowA nentA ncolB seed`

   This driver program tests the `SubMtx_solve()` method which tests the solve $AX = B$ when A is diagonal or block diagonal, and $(I + A)X = B$ otherwise (A is strict upper or lower triangular). Use the script file `do_solve` for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter must be one of 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
   - The `mode` parameter must be one of 2 (`SUBMTX_SPARSE_ROWS`), 3 (`SUBMTX_SPARSE_COLUMNS`), 5 (`SUBMTX_DENSE_SUBROWS`), 6 (`SUBMTX_DENSE_SUBCOLUMNS`), 7 (`SUBMTX_DIAGONAL`), 8 (`SUBMTX_BLOCK_DIAGONAL_SYM`) or 9 (`SUBMTX_BLOCK_DIAGONAL_HERM`).
   - The `nrowA` parameter is the number of rows in the matrix.
   - The `nentA` parameter is the number of nonzero entries in the submatrix, when appropriate.
   - The `ncolB` parameter is the number of columns in B.
   - The `seed` parameter is a random number seed.

4. `test_solveH msglvl msgFile type mode nrowA nentA ncolB seed`

   This driver program tests the `SubMtx_solve()` method which tests the solve $(I + A^H)X = B$ when A is strict upper or lower triangular and has dense subrows, dense subcolumns, sparse rows, or sparse columns. Use the script file `do_solveH` for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter must be 2 (`SPOOLES_COMPLEX`).
   - The `mode` parameter must be one of 2 (`SUBMTX_SPARSE_ROWS`), 3 (`SUBMTX_SPARSE_COLUMNS`), 5 (`SUBMTX_DENSE_SUBROWS`) or 6 (`SUBMTX_DENSE_SUBCOLUMNS`).
   - The `nrowA` parameter is the number of rows in the matrix.
   - The `nentA` parameter is the number of nonzero entries in the submatrix, when appropriate.
   - The `ncolB` parameter is the number of columns in B.
   - The `seed` parameter is a random number seed.

5. `test_solveT msglvl msgFile type mode nrowA nentA ncolB seed`

   This driver program tests the `SubMtx_solve()` method which tests the solve $(I + A^T)X = B$ when A is strict upper or lower triangular and has dense subrows, dense subcolumns, sparse rows, or sparse columns. Use the script file `do_solveT` for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter must be one of 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
   - The `mode` parameter must be one of 2 (`SUBMTX_SPARSE_ROWS`), 3 (`SUBMTX_SPARSE_COLUMNS`), 5 (`SUBMTX_DENSE_SUBROWS`) or 6 (`SUBMTX_DENSE_SUBCOLUMNS`).
   - The `nrowA` parameter is the number of rows in the matrix.
   - The `nentA` parameter is the number of nonzero entries in the submatrix, when appropriate.
   - The `ncolB` parameter is the number of columns in B.
   - The `seed` parameter is a random number seed.

6. `test_solveupd msglvl msgFile type mode nrowA nentA ncolB seed`

   This driver program tests the `SubMtx_solveupd()` method which tests the update $Y := Y - A * X$, used in the forward solve. X and Y have dense columns, and A has dense rows or columns or sparse rows or columns. Use the script file `do_solveupd` for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter must be one of 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
   - The `mode` parameter must be one of 0 (`SUBMTX_DENSE_ROWS`), 1 (`SUBMTX_DENSE_COLUMNS`), 2 (`SUBMTX_SPARSE_ROWS`) or 3 (`SUBMTX_SPARSE_COLUMNS`).

- The nrowY parameter is the number of rows in Y.
- The ncolY parameter is the number of columns in Y.
- The nrowA parameter is the number of rows in A, nrowA $\leq$ nrowY.
- The ncolA parameter is the number of columns in A, ncolA $\leq$ nrowX.
- The nentA parameter is the number of nonzero entries in the submatrix, when appropriate.
- The nrowX parameter is the number of rows in X, nrowA $\leq$ nrowY.
- The seed parameter is a random number seed.

7. test_solveupdH msglvl msgFile type mode nrowA nentA ncolB seed

   This driver program tests the SubMtx_solveupd() method which tests the update $Y := Y - A^H * X$, used in the forward solve of a hermitian factorization. X and Y have dense columns, and A has dense rows or columns or sparse rows or columns. Use the script file do_solveupdH for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The msglvl parameter determines the amount of output. Use msglvl = 1 for just timing output.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The type parameter must be 2 (SPOOLES_COMPLEX).
   - The mode parameter must be one of 0 (SUBMTX_DENSE_ROWS), 1 (SUBMTX_DENSE_COLUMNS), 2 (SUBMTX_SPARSE_ROWS) or 3 (SUBMTX_SPARSE_COLUMNS).
   - The nrowY parameter is the number of rows in Y.
   - The ncolY parameter is the number of columns in Y.
   - The nrowA parameter is the number of rows in A, nrowA $\leq$ nrowY.
   - The ncolA parameter is the number of columns in A, ncolA $\leq$ nrowX.
   - The nentA parameter is the number of nonzero entries in the submatrix, when appropriate.
   - The nrowX parameter is the number of rows in X, nrowA $\leq$ nrowY.
   - The seed parameter is a random number seed.

8. test_solveupdT msglvl msgFile type mode nrowA nentA ncolB seed

   This driver program tests the SubMtx_solveupd() method which tests the update $Y := Y - A^T * X$, used in the forward solve of a symmetric factorization. X and Y have dense columns, and A has dense rows or columns or sparse rows or columns. Use the script file do_solveupdT for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The msglvl parameter determines the amount of output. Use msglvl = 1 for just timing output.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The type parameter must be one of 1 (SPOOLES_REAL) or 2 (SPOOLES_COMPLEX).
   - The mode parameter must be one of 0 (SUBMTX_DENSE_ROWS), 1 (SUBMTX_DENSE_COLUMNS), 2 (SUBMTX_SPARSE_ROWS) or 3 (SUBMTX_SPARSE_COLUMNS).
   - The nrowY parameter is the number of rows in Y.
   - The ncolY parameter is the number of columns in Y.
   - The nrowA parameter is the number of rows in A, nrowA $\leq$ nrowY.
   - The ncolA parameter is the number of columns in A, ncolA $\leq$ nrowX.

- The `nentA` parameter is the number of nonzero entries in the submatrix, when appropriate.
- The `nrowX` parameter is the number of rows in X, `nrowA` $\leq$ `nrowY`.
- The `seed` parameter is a random number seed.

9. `test_sort msglvl msgFile type mode nrowA ncolA nentA seed`

   This driver program tests the `SubMtx_sortRowsUp()` and `SubMtx_sortColumnsUp()` methods. Use the script file `do_sort` for testing. When the output file is loaded into matlab, the last lines to the screen contain the errors.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter must be one of 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
   - The `mode` parameter must be one of 0 (`SUBMTX_DENSE_ROWS`), 1 (`SUBMTX_DENSE_COLUMNS`), 2 (`SUBMTX_SPARSE_ROWS`) or 3 (`SUBMTX_SPARSE_COLUMNS`).
   - The `nrowA` parameter is the number of rows in A.
   - The `ncolA` parameter is the number of columns in A.
   - The `nentA` parameter is the number of nonzero entries in the submatrix, when appropriate.
   - The `seed` parameter is a random number seed.

Chapter 38

SubMtxList: SubMtx list object

This object was created to handle a list of lists of SubMtx objects during a matrix solve. Its form and function is very close to the ChvList object that handles lists of lists of Chv objects during the factorization.

Here are the main properties.

1. There are a fixed number of lists, set when the SubMtxList object is initialized.
2. For each list there is an expected count, the number of times an object will be added to the list. (Note, a NULL object can be added to the list. In this case, nothing is added to the list, but its count is decremented.)
3. There is one lock for all the lists, but each list can be flagged as necessary to lock or not necessary to lock before an insertion, count decrement, or an extraction is made to the list.

The SubMtxList object manages a number of lists that may require handling critical sections of code. For example, one thread may want to add an object to a particular list while another thread is removing objects. The critical sections are hidden inside the SubMtxList object. Our solve code do not know about any mutual exclusion locks that govern access to the lists.

There are four functions of the SubMtxList object.

- Is the incoming count for a list nonzero?
- Is a list nonempty?
- Add an object to a list (possibly a NULL object) and decrement the incoming count.
- Remove a subset of objects from a list.

The first two operations are queries, and can be done without locking the list. The third operation needs a lock only when two or more threads will be inserting objects into the list. The fourth operation requires a lock only when one thread will add an object while another thread removes the object and the incoming count is not yet zero.

Having a lock associated with a SubMtxList object is optional, for example, it is not needed during a serial factorization nor a MPI solve. In the latter case there is one SubMtxList per process. For a multithreaded solve there is one SubMtxList object that is shared by all threads. The mutual exclusion lock that is (optionally) embedded in the SubMtxList object is a Lock object from this library. It is inside the Lock object that we have a mutual exclusion lock. Presently we support the Solaris and POSIX thread packages. Porting the multithreaded codes to another platform should be simple if the POSIX thread package is present. Another type of thread package will require some modifications to the Lock object, but none to the SubMtxList objects.

38.1 Data Structure

The `SubMtxList` structure has the following fields.

- `int nlist` : number of lists.
- `SubMtx **heads` : vector of pointers to the heads of the list of `SubMtx` objects.
- `int *counts` : vector of incoming counts for the lists.
- `Lock *lock` : mutual exclusion lock.
- `char *flags` : vector of lock flags for the lists. If `flags[ilist] == 'N'`, the list does not need to be locked. If `flags[ilist] == 'Y'`, the list does need to be locked. Used only when `lock` is not `NULL`.
- `int nlocks` : total number of locks made on the mutual exclusion lock.

38.2 Prototypes and descriptions of `SubMtxList` methods

This section contains brief descriptions including prototypes of all methods that belong to the `SubMtxList` object.

38.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `SubMtxList * SubMtxList_new ( void ) ;`

   This method simply allocates storage for the `SubMtxList` structure and then sets the default fields by a call to `SubMtxList_setDefaultFields()`.

2. `void SubMtxList_setDefaultFields ( SubMtxList *list ) ;`

   The structure's fields are set to default values: `nlist` and `nlocks` set to zero, and `heads`, `counts`, `lock` and `flags` are set to `NULL` .

   Error checking: If `list` is `NULL`, an error message is printed and the program exits.

3. `void SubMtxList_clearData ( SubMtxList *list ) ;`

   This method clears the object and free's any owned data by calling `SubMtx_free()` for each object on the free list. If `heads` is not `NULL`, it is free'd. If `counts` is not `NULL`, it is free'd via a call to `IVfree()`. If `flags` is not `NULL`, it is free'd via a call to `CVfree()`. If the lock is not `NULL`, it is destroyed via a call to `mutex_destroy()` and then free'd. There is a concluding call to `SubMtxList_setDefaultFields()`.

   Error checking: If `list` is `NULL`, an error message is printed and the program exits.

4. `void SubMtxList_free ( SubMtxList *list ) ;`

   This method releases any storage by a call to `SubMtxList_clearData()` and then free the space for `list`.

   Error checking: If `list` is `NULL`, an error message is printed and the program exits.

38.2.2 Initialization methods

There are three initializer methods.

1. `void SubMtxList_init( SubMtxList *list, int nlist, int counts[], int lockflag, char flags[] ) ;`

   Any data is cleared via a call to `SubMtxList_clearData()`. The number of lists is set and the `heads[]` vector is initialized. If `counts` is not `NULL`, the object's `counts[]` vector is allocated and filled with the incoming entries. If `lockflag` is zero, the lock is not initialized. If `lockflag` is `1`, the lock is initialized to be able to synchronize threads with the calling process. If `lockflag` is `2`, the lock is initialized to be able to synchronize threads across processes. If `flags` is not `NULL`, the object's `flags[]` vector is allocated and filled with the incoming entries.

   Error checking: If `list` is `NULL`, or if `nlist` ≤ 0, or if `lockflag` is not in `[0,2]`, an error message is printed and zero is returned.

38.2.3 Utility methods

1. `int SubMtxList_isListNonempty ( SubMtxList *list, int ilist ) ;`

   If list `ilist` is empty, the method returns 0. Otherwise, the method returns 1.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

2. `int SubMtxList_isCountZero ( SubMtxList *list, int ilist ) ;`

   If `counts` is `NULL`, or if `counts[ilist]` equal to zero, the method returns 1. Otherwise, the method returns 0.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

3. `SubMtx * SubMtxList_getList ( SubMtxList *list, int ilist ) ;`

   If list `ilist` is empty, the method returns `NULL`. Otherwise, if the list needs to be locked, the lock is locked. The head of the list is saved to a pointer and then the head is set to `NULL`. If the list was locked, the number of locks is incremented and the lock unlocked. The saved pointer is returned.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

4. `void SubMtxList_addObjectToList ( SubMtxList *list, SubMtx *mtx, int ilist ) ;`

   If the list needs to be locked, the lock is locked. If `mtx` is not `NULL`, it is added to the head of the list. If `counts` is not `NULL`, then `counts[ilist]` is decremented. If the lock was locked, the number of locks is incremented and it is now unlocked.

   Error checking: If `list` is `NULL`, or if `ilist` is not in the range `[0,nlist)`, an error message is printed and zero is returned.

38.2.4 IO methods

1. `void SubMtxList_writeForHumanEye ( SubMtxList *list, FILE *fp ) ;`

   This method write the list to a file in user readable form.

   Error checking: If `list` or `fp` are `NULL`, an error message is printed and zero is returned.

Chapter 39

SubMtxManager: SubMtx object manager

This object was created to manage a number of instances of SubMtx double precision matrix objects. Its form and functionality is almost identical to that of the ChvManager object.

The SubMtxManager object is very simple. It has two functions.

- When asked for a SubMtx object of a certain size, it returns one.
- When given a SubMtx object (or a list of objects connected via their next fields) that is (are) no longer necessary for the calling program, it takes some action with it (them).

There are presently two *modes* of behavior : the first is a wrapper around calls to SubMtx_new() and SubMtx_free() (which contain calls to malloc() and free()), the second can *recycle* instances to be used later.

Both behaviors are appropriate in certain circumstances. When one needs a large number of objects (though not all at the same time) whose workspace requirements are roughly equal, recycling the objects can be cost effective. On the other hand, consider a scenario which arises in the factorization of FrontMtx objects. At first one needs a moderate number of large SubMtx objects which store the $U_{J,\partial J}$ and $L_{\partial J,J}$ submatrices. We then replace them with a larger number of smaller objects that store the $U_{J,K}$ and $L_{K,J}$ matrices. In this case recycling is *not* cost-effective for the large objects are recycled as smaller objects and much of their workspace is inactive and therefore wasted. The total storage footprint can be almost twice as large as necessary.

Our recycling mode is a very simple implementation. The manager object maintains a free list of objects, sorting in ascending order of the number of bytes in their workspace. When asked for an object with a certain amount of workspace, the manager performs a linear search of the list and returns the first object that has sufficient space. If no such object exists, i.e., if the list is empty or there is no object large enough, the manager allocates a new SubMtx object, initializes it with sufficient work space, and returns a pointer to the object. When a SubMtx object is no longer necessary, it is *released* to the manager object, which then inserts it into the free list. A list of SubMtx objects can be released in one call.

One can specify whether the object is to be locked via a mutual exclusion lock. This is not necessary for a serial or MPI factorization or solve (where there is one SubMtxManager object for each processor), but it is necessary for in a multithreaded environment.

Each manager object keeps track of certain statistics, bytes in their workspaces, the total number of bytes requested, the number of requests for a SubMtx objects, the number of releases, and the number of locks and unlocks.

39.1 Data Structure

The `SubMtxManager` structure has the following fields.

- `SubMtx *head` : head of the free list of `SubMtx` objects.
- `Lock *lock` : mutual exclusion lock.
- `int mode` : behavior mode. When `mode = 0`, the object calls `SubMtx_new()` and `SubMtx_free()` to create and release objects. When `mode = 1`, the object recycles the objects.
- `int nactive` : number of active `SubMtx` objects.
- `int nbytesactive` : number of bytes in the active `SubMtx` objects.
- `int nbytesrequested` : total number of bytes in the requested `SubMtx` objects.
- `int nbytesalloc` : total number of bytes that were actually allocated in the workspace of the `SubMtx` objects.
- `int nrequests` : total number of requests for `SubMtx` objects.
- `int nreleases` : total number of releases of `SubMtx` objects.
- `int nlocks` : total number of locks made on the mutual exclusion lock.
- `int nunlocks` : total number of unlocks made on the mutual exclusion lock.

39.2 Prototypes and descriptions of `SubMtxManager` methods

This section contains brief descriptions including prototypes of all methods that belong to the `SubMtxManager` object.

39.2.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `SubMtxManager * SubMtxManager_new ( void ) ;`

   This method simply allocates storage for the `SubMtxManager` structure and then sets the default fields by a call to `SubMtxManager_setDefaultFields()`.

2. `void SubMtxManager_setDefaultFields ( SubMtxManager *manager ) ;`

   The structure's fields are set to default values: `mode`, `nactive`, `nbytesactive`, `nbytesrequested`, `nbytesalloc`, `nrequests`, `nreleases`, `nlocks` and `nunlocks` are set to zero, and `head` and `lock` are set to `NULL` .

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

3. `void SubMtxManager_clearData ( SubMtxManager *manager ) ;`

   This method clears the object and free's any owned data by calling `SubMtx_free()` for each object on the free list. If the lock is not `NULL`, it is destroyed via a call to `mutex_destroy()` and then free'd. There is a concluding call to `SubMtxManager_setDefaultFields()`.

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

4. `void SubMtxManager_free ( SubMtxManager *manager ) ;`

   This method releases any storage by a call to `SubMtxManager_clearData()` and then free the space for `manager`.

   Error checking: If `manager` is `NULL`, an error message is printed and the program exits.

39.2.2 Initialization methods

1. `void SubMtxManager_init( SubMtxManager *manager, int lockflag, int mode ) ;`

   Any data is cleared via a call to `SubMtxManager_clearData()`. If `lockflag` is zero, the lock is not initialized. If `lockflag` is `1`, the lock is initialized to be able to synchronize threads with the calling process. If `lockflag` is `2`, the lock is initialized to be able to synchronize threads across processes. The behavior mode is set to `mode`.

   Error checking: If `manager` is `NULL`, or if `lockflag` is not in `[0,2]`, or if `mode` is not in `[0,1]`, an error message is printed and zero is returned.

39.2.3 Utility methods

1. ```
   SubMtx * SubMtxManager_newObjectOfSizeNbytes ( SubMtxManager *manager,
                                                  int nbytesNeeded ) ;
   ```

   This method returns a pointer to a `SubMtx` object that has at least `nbytesNeeded` bytes in its workspace.

   Error checking: If `manager` is `NULL`, or if `nbytesNeeded` ≤ 0, an error message is printed and zero is returned.

2. `void SubMtxManager_releaseObject ( SubMtxManager *manager, SubMtx *mtx ) ;`

   This method releases the `mtx` instance, either free'ing it (if `mode = 0`), or returning it to the free list (if `mode = 1`).

   Error checking: If `manager` or `mtx` is `NULL`, an error message is printed and zero is returned.

3. `void SubMtxManager_releaseListOfObjects ( SubMtxManager *manager, SubMtx *first ) ;`

   This method releases a list of `SubMtx` objects whose head is `first`, either free'ing them (if `mode = 0`), or returning them to the free list (if `mode = 1`).

   Error checking: If `manager` or `head` is `NULL`, an error message is printed and zero is returned.

39.2.4 IO methods

1. `void SubMtxManager_writeForHumanEye ( SubMtxManager *manager, FILE *fp ) ;`

   This method writes a `SubMtxManager` object to a file in an easily readable format.

   Error checking: If `manager` or `fp` are `NULL`, an error message is printed and zero is returned.
   ```

Chapter 40

SymbFac: Symbolic Factorization

This object is really a collection of methods — there is no struct associated with it, and therefore no data. The reason for its existence is that a symbolic factorization can be produced using an ETree object and one of several different inputs, e.g., a Graph object, a InpMtx object, and a Pencil object. Possibly there could be others, all that is necessary is to be able to communicate the nonzero structure of a chevron.

The symbolic factorization methods used to belong to the ETree object. It was a natural location for this functionality. We first generated a symbolic factorization using a Graph object as input, and since the ETree object used a Graph object to initialize itself, this was acceptable. Then we started to bypass the Graph object and use a InpMtx object as input, and this forced the *vision* of the ETree object (the other objects it must know about) to grow. By the time we started using the Pencil matrix pencil object to find the symbolic factorization, we knew things were out of hand. By creating a new object to handle the symbolic factorization, we can remove the InpMtx and Pencil objects from the vision of the ETree object.

The symbolic factorization is stored in an IVL object. The vertices in $J \cup \partial J$ are stored in the J'th list and can be accessed via a call to

```
IVL_listAndSize(symbfacIVL, J, &size, &indices) ;
```

where on return, the int vector indices[size] contains the vertices.

NOTE: The SymbFac_initFromInpMtx() and SymbFac_initFromPencil() methods have been changed slightly to make them more efficient. The InpMtx objects that are input are now required to have chevron coordinate type and storage mode must be by vectors.

40.1 Data Structure

There is no struct or data associated with the SymbFac object.

40.2 Prototypes and descriptions of SymbFac methods

This section contains brief descriptions including prototypes of all methods that belong to the SymbFac object.

40.2.1 Symbolic factorization methods

1. `IVL * SymbFac_initFromGraph ( ETree *etree, Graph *graph ) ;`

This symbolic factorization method takes a `Graph` object as input. This method constructs an `IVL` object that contains one list per front. List `ilist` contains the internal and external vertices for front `ilist`. If the input `graph` is a compressed graph, then the lists of compressed vertices make little sense; they must be converted to original vertices. To do this, see the `IVL_expand()` method. The `nodwghtsIV` and `bndwghtsIV` objects for the `ETree` object are updated using information from the symbolic factorization.

Error checking: If `etree` or `graph` is `NULL`, or if `nfront` < 1, or if `nvtx` < 1, or if `graph->nvtx` $\neq$ `nvtx`, an error message is printed and the program exits.

2. `IVL * SymbFac_initFromInpMtx ( ETree *etree, InpMtx *inpmtx ) ;`

   This symbolic factorization method takes a `InpMtx` object as input. This method constructs an `IVL` object that contains one list per front. List `ilist` contains the internal and external vertices for front `ilist`. We assume that both the `ETree` and `InpMtx` objects have had been permuted into their final ordering. The `nodwghtsIV` and `bndwghtsIV` objects for the `ETree` object are updated using information from the symbolic factorization.

   Error checking: If `etree` or `inpmtx` is `NULL`, or if the coordinate type of `inpmtx` is not `INPMTX_BY_CHEVRONS`, or if the storage mode of `inpmtx` is not `INPMTX_BY_VECTORS`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

3. `IVL * SymbFac_initFromPencil ( ETree *etree, Pencil *pencil ) ;`

   This first symbolic factorization method takes a `Pencil` object as input and is used to compute the symbolic factorization for a matrix pencil $A - \sigma B$. This method constructs an `IVL` object that contains one list per front. List `ilist` contains the internal and external vertices for front `ilist`. We assume that both the `ETree` and `InpMtx` objects have had been permuted into their final ordering. The `nodwghtsIV` and `bndwghtsIV` objects for the `ETree` object are updated using information from the symbolic factorization.

   Error checking: If `etree` or `inpmtxA` is `NULL`, or if the coordinate type of either internal `InpMtx` objects is not `INPMTX_BY_CHEVRONS`, or if the storage mode of either internal `InpMtx` objects is not `INPMTX_BY_VECTORS`, or if `nfront` < 1, or if `nvtx` < 1, an error message is printed and the program exits.

40.3 Driver programs

1. ```
   testSymbFacInpMtx msglvl msgFile inETreeFile inDInpMtxFile
                     outETreeFile outIVfile outIVLfile
   ```

   This driver program reads in an `ETree` object and a `InpMtx` object and computes the symbolic factorization. The `ETree` object is updated (the front sizes and boundary sizes may change) and is optionally written out to `outETreeFile`. The old-to-new `IV` object is optionally written to `outIVfile`. The `IVL` object that contains the symbolic factorization is optionally written to `outIVLfile`.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
   - The `inInpMtxFile` parameter is the input file for the `InpMtx` object. It must be of the form `*.inpmtxf` or `*.inpmtxb`. The `InpMtx` object is read from the file via the `InpMtx_readFromFile()` method.
   ```

- The `outETreeFile` parameter is the output file for the `ETree` object. If `outETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outETreeFile` is of the form `*.etreef`), or a binary file (if `outETreeFile` is of the form `*.etreeb`).
- The `outIVfile` parameter is the output file for the vertex-to-front map `IV` object. If `outIVfile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outIVfile` is of the form `*.ivf`), or a binary file (if `outIVfile` is of the form `*.ivb`).
- The `outIVLfile` parameter is the output file for the symbolic factorization `IVL` object. If `outIVLfile` is `none` then the `IVL` object is not written to a file. Otherwise, the `IVL_writeToFile()` method is called to write the object to a formatted file (if `outIVLfile` is of the form `*.ivlf`), or a binary file (if `outIVLfile` is of the form `*.ivlb`).

2. `testSymbFacGraph msglvl msgFile inETreeFile inGraphFile`
   `outETreeFile outIVfile outIVLfile`

   This driver program reads in an `ETree` object and a `Graph` object and computes the symbolic factorization. The `ETree` object is updated (the front sizes and boundary sizes may change) and is optionally written out to `outETreeFile`. The old-to-new `IV` object is optionally written to `outIVfile`. The `IVL` object that contains the symbolic factorization is optionally written to `outIVLfile`.

   - The `msglvl` parameter determines the amount of output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `outETreeFile` parameter is the output file for the `ETree` object. If `outETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `outETreeFile` is of the form `*.etreef`), or a binary file (if `outETreeFile` is of the form `*.etreeb`).
   - The `outIVfile` parameter is the output file for the vertex-to-front map `IV` object. If `outIVfile` is `none` then the `IV` object is not written to a file. Otherwise, the `IV_writeToFile()` method is called to write the object to a formatted file (if `outIVfile` is of the form `*.ivf`), or a binary file (if `outIVfile` is of the form `*.ivb`).
   - The `outIVLfile` parameter is the output file for the symbolic factorization `IVL` object. If `outIVLfile` is `none` then the `IVL` object is not written to a file. Otherwise, the `IVL_writeToFile()` method is called to write the object to a formatted file (if `outIVLfile` is of the form `*.ivlf`), or a binary file (if `outIVLfile` is of the form `*.ivlb`).

Part V

Miscellaneous Methods

Chapter 41

Misc directory

This directory contains a number of miscellaneous functions and driver programs that don't really fit anywhere else. There are functions to generate nested dissection orderings on regular 2-D and 3-D grids — the usual nested dissection, with double wide separators, and local nested dissection [8]. There are wrapper methods for minimum degree, nested dissection and multisection orderings for general graphs. There is also a driver program to produce a postscript file for a 2-D graph, very useful for visualizing graph partitionings and orderings.

41.1 Prototypes and descriptions of methods in the Misc directory

This section contains brief descriptions including prototypes of all methods in the `Misc` directory.

41.1.1 Theoretical nested dissection methods

1. `void mkNDperm ( int n1, int n2, int n3, int newToOld[], int west,`
   `                int east, int south, int north, int bottom, int top ) ;`

   This method this vector fills a permutation vector with the nested dissection new-to-old ordering of the vertices for the subgrid defined by nodes whose coordinates lie in

   `[west, east] x [south, north] x [bottom, top].`

   The method calls itself recursively. To find the permutation for an `n1 x n2 x n3` grid, call

   `mkNDperm(n1, n2, n3, newToOld, 0, n1-1, 0, n2-1, 0, n3-1) ;`

   from a driver program.

   Error checking: If `n1`, `n2` or `n3` are less than or equal to zero, or if `newToOld` is `NULL`, or if `west`, `south` or `bottom` are less than or equal to zero, of if `east` $\geq$ `n1`, of if `north` $\geq$ `n2`, of if `top` $\geq$ `n3`, an error message is printed and the program exits.

2. `void mkNDperm2 ( int n1, int n2, int n3, int newToOld[], int west,`
   `                 int east, int south, int north, int bottom, int top ) ;`

   This method this vector fills a permutation vector with the nested dissection new-to-old ordering of the vertices for the subgrid defined by nodes whose coordinates lie in `[west, east] x [south, north] x [bottom, top].`

There is one important difference between this method and `mkNDperm()` above; this method finds *double-wide* separators, necessary for an operator with more than nearest neighbor grid point coupling. The method calls itself recursively. To find the permutation for an `n1 x n2 x n3` grid, call

```
mkNDperm(n1, n2, n3, newToOld, 0, n1-1, 0, n2-1, 0, n3-1) ;
```

from a driver program.

Error checking: If `n1`, `n2` or `n3` are less than or equal to zero, or if `newToOld` is `NULL`, or if `west`, `south` or `bottom` are less than or equal to zero, of if `east` $\geq$ `n1`, of if `north` $\geq$ `n2`, of if `top` $\geq$ `n3`, an error message is printed and the program exits.

3. ```
   void localND2D ( int n1, int n2, int p1, int p2,
                    int dsizes1[], int dsizes2[], int oldToNew[] ) ;
   ```

   This method finds a local nested dissection ordering [8] for an `n1 x n2` 2-D grid. There are `p1 x p2` domains in the grid. The `dsizes1[]` and `dsizes2[]` vectors are optional; they allow the user to explicitly input domain sizes. If `dsizes1[]` and `dsizes2[]` are not `NULL`, the `q = q1 + q2*p1`'th domain contains a `dsizes1[q1] x dsizes2[q2]` subgrid of points.

   Error checking: If `n1` or `n2` are less than or equal to zero, or if `p1` or `p2` are less than or equal to zero, or if $2\texttt{p1} - 1 > \texttt{n1}$, or if $2\texttt{p2} - 1 > \texttt{n2}$, or if `oldToNew` is `NULL`, or if `dsizes1[]` and `dsizes2[]` are not `NULL` but have invalid entries (all entries must be positive, entries in `dsizes1[]` must sum to `n1 - p1 + 1`, and entries in `dsizes2[]` must sum to `n2 - p2 + 1`, an error message is printed and the program exits.

4. ```
   void localND3D ( int n1, int n2, int n3, int p1, int p2, int p3,
                    int dsizes1[], int dsizes2[], int dsizes3[],
                    int oldToNew[] ) ;
   ```

   This method finds a local nested dissection ordering [8] for an `n1 x n2 x n3` 3-D grid. There are `p1 x p2 x p3` domains in the grid. The `q`'th domain contains a `dsizes1[q] x dsizes2[q] x dsizes3[q]` subgrid of points. The `dsizes1[]`, `dsizes2[]` and `dsizes3[]` vectors are optional; they allow the user to explicitly input domain sizes. If `dsizes1[]`, `dsizes2[]` and `dsizes3[]` are not `NULL`, the `q = q1 + q2*p1+ q3*p1*p2`'th domain contains a `dsizes1[q1] x dsizes2[q2] x disizes3[q3]` subgrid of points.

   Error checking: If `n1`, `n2` or `n3` are less than or equal to zero, or if `p1`, `p2` or `p3` are less than or equal to zero, or if $2\texttt{p1} - 1 > \texttt{n1}$, or if $2\texttt{p2} - 1 > \texttt{n2}$, or if $2\texttt{p3} - 1 > \texttt{n3}$, or if `oldToNew` is `NULL`, or if `dsizes1[]`, `disizes2[]` and `dsizes3[]` are not `NULL` but have invalid entries (all entries must be positive, entries in `dsizes1[]` must sum to `n1 - p1 + 1`, entries in `dsizes2[]` must sum to `n2 - p2 + 1`, and entries in `dsizes3[]` must sum to `n3 - p3 + 1`, an error message is printed and the program exits.

5. ```
   void fp2DGrid ( int n1, int n2, int ivec[], FILE *fp ) ;
   ```

   This method writes the `ivec[]` vector onto an `n1 x n2` grid to file `fp`. This is useful to visualize an ordering or a metric on a grid.

   Error checking: If `n1` or `n2` are less than or equal to zero, or if `ivec` or `fp` are `NULL`, an error message is printed and the program exits.

6. ```
   void fp3DGrid ( int n1, int n2, int n3, int ivec[], FILE *fp ) ;
   ```

   This method writes the `ivec[]` vector onto an `n1 x n2 x n3` grid to file `fp`. This is useful to visualize an ordering or a metric on a grid.

   Error checking: If `n1`, `n2` or `n3` are less than or equal to zero, or if `ivec` or `fp` are `NULL`, an error message is printed and the program exits.

41.1.2 Multiple minimum degree, Nested dissection and multisection wrapper methods

There are three simple methods to find minimum degree, nested dissection and multisection orderings. In addition, there is one method that finds the better of two methods – nested dissection and multisection. (Much of the work to find either nested dissection or multisection is identical, so this method takes little more time than either of the two separately.)

To properly specify these methods there are many parameters — these three wrapper methods insulate the user from all but one or two of the parameters. As a result, the quality of the ordering may not be as good as can be found by using non-default settings of the parameters.

One wrapper method computes a minimum degree ordering — the only input parameter is a random number seed. Two wrappers methods compute the nested dissection and multisection orderings — in addition to a random number seed there is a upper bound on the subgraph size used during the graph partition. This is the most sensitive of the parameters.

The user interested in more customized orderings should consult the chapters on the the `GPart`, `DSTree` and `MSMD` objects that perform the three steps of the ordering process: perform an incomplete nested dissection of the graph, construct the map from vertices to stages in which they will be eliminated, and perform the multi-stage minimum degree ordering. The driver programs in the `GPart` and `MSMD` directories fully exercise the graph partition and ordering strategies by giving the user access to all input parameters.

1. `ETree * orderViaMMD ( Graph *graph, int seed, int msglvl, FILE *msgFile ) ;`

   This method returns a front tree `ETree` object for a multiple minimum degree ordering of the graph `graph`. The `seed` parameter is a random number seed. The `msglvl` and `msgFile` parameters govern the diagnostics output. Use `msglvl = 0` for no output, `msglvl = 1` for timings and scalar statistics, and use `msglvl > 1` with care, for it can generate huge amounts of output.

   Error checking: If `graph` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

2. ```
   ETree * orderViaND ( Graph *graph, int maxdomainsize, int seed,
                        int msglvl, FILE *msgFile ) ;
   ```

   This method returns a front tree `ETree` object for a nested dissection ordering of the graph `graph`. If a subgraph has more vertices than the `maxdomainsize` parameter, it is split. The `seed` parameter is a random number seed. The `msglvl` and `msgFile` parameters govern the diagnostics output. Use `msglvl = 0` for no output, `msglvl = 1` for timings and scalar statistics, and use `msglvl > 1` with care, for it can generate huge amounts of output.

   Error checking: If `graph` is `NULL`, or if `maxdomainsize` ≤ 0, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

3. ```
   ETree * orderViaMS ( Graph *graph, int maxdomainsize, int seed,
                        int msglvl, FILE *msgFile ) ;
   ```

   This method returns a front tree `ETree` object for a multisection ordering of the graph `graph`. If a subgraph has more vertices than the `maxdomainsize` parameter, it is split. The `seed` parameter is a random number seed. The `msglvl` and `msgFile` parameters govern the diagnostics output. Use `msglvl = 0` for no output, `msglvl = 1` for timings and scalar statistics, and use `msglvl > 1` with care, for it can generate huge amounts of output.

   Error checking: If `graph` is `NULL`, or if `maxdomainsize` ≤ 0, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

4. ```
   ETree * orderViaBestOfNDandMS ( Graph *graph, int maxdomainsize, int maxzeros,
                                   int maxsize, int seed, int msglvl, FILE *msgFile ) ;
   ```
   ```

This method returns a front tree `ETree` object for a better of two orderings, a nested dissection and multisection ordering. If a subgraph has more vertices than the `maxdomainsize` parameter, it is split. The `seed` parameter is a random number seed. This method also transforms the front tree using the `maxzeros` and `maxsize` parameters. See the `ETree_transform()` method in Section 19.2.10. The `msglvl` and `msgFile` parameters govern the diagnostics output. Use `msglvl = 0` for no output, `msglvl = 1` for timings and scalar statistics, and use `msglvl > 1` with care, for it can generate huge amounts of output.

Error checking: If `graph` is `NULL`, or if `maxdomainsize` ≤ 0, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

41.1.3 Graph drawing method

1. `void drawGraphEPS ( Graph *graph, Coords *coords, IV *tagsIV, double bbox[4], double rect[4], double linewidth1, double linewidth2, double radius, char *epsFileName, int msglvl, FILE *msgFile ) ;`

```
void drawGraphEPS ( Graph *graph, Coords *coords, IV *tagsIV,
                    double bbox[4], double rect[4], double linewidth1,
                    double linewidth2, double radius, char *epsFileName,
                    int msglvl, FILE *msgFile ) ;
```

This method is used to create an EPS (Encapsulated Postscript) file that contains a picture of a graph in two dimensions. We use this to visualize separators and domain decompositions, mostly of regular grids and triangulations of a planar region.

The `graph` object defines the connectivity of the vertices. The `coords` object defines the locations of the vertices. The `tagsIV` object is used to define whether or not an edge is drawn between two vertices adjacent in the graph. When `tagsIV` is not `NULL`, if there is an edge `(u,v)` in the graph and `tags[u] = tags[v]`, then the edge with width `linewidth1` is drawn. For edges `(u,v)` in the graph and `tags[u] != tags[v]`, then the edge with width `linewidth2` is drawn, assuming `linewidth2` > 0. If `tagsIV` is `NULL`, than all edges are drawn with width `linewidth1`. Each vertex is draw with a filled circle with radius `radius`.

The graph and its `Coords` object occupy a certain area in 2-D space. We try to plot the graph inside the area defined by the `rect[]` array in such a manner that the relative scales are preserved (the graph is not stretched in either the x or y direction) and that the larger of the width and height of the graph fills the area defined by the `rect[]` rectangle. *Note*: hacking postscript is *not* an area of expertise of either author. Some Postscript viewers give us messages that we are not obeying the format conventions (this we do not doubt), but we have never failed to view or print one of these files.

Error checking: If the method is unable to open the file, an error message is printed and the program exits.

41.1.4 Linear system construction

Our driver programs test linear systems where the matrices come from regular grids using nested dissection orderings. There are two methods that generate linear systems of this form along with the front tree and symbolic factorization.

1.
```
void mkNDlinsys ( int n1, int n2, int n3, int maxzeros, int maxsize,
                  int type, int symmetryflag, int nrhs, int seed, int msglvl,
                  FILE *msgFile, ETree **pfrontETree, IVL **psymbfacIVL,
                  InpMtx **pmtxA, DenseMtx **pmtxX, DenseMtx **pmtxB ) ;
```

This method creates a linear system $AX = B$ for a `n1` $\times$ `n2` $\times$ `n3` grid. The entries in A and X are random numbers, B is computed as the product of A with X. A can be real (`type = 1`) or complex (`type = 2`), and can be symmetric (`symmetryflag = 0`), Hermitian (`symmetryflag = 1`) or nonsymmetric (`symmetryflag = 2`). The number of columns of X is given by `nrhs`. The linear system is ordered using theoretical nested dissection, and the front tree is transformed using the `maxzeros` and `maxsize` parameters. The addresses of the front tree, symbolic factorization, and three matrix objects are returned in the last five arguments of the calling sequence.

Error checking: None presently.

2. ```
void mkNDlinsysQR (int n1, int n2, int n3, int type, int nrhs, int seed,
                    int msglvl, FILE *msgFile, ETree **pfrontETree, IVL **psymbfacIVL,
                    InpMtx **pmtxA, DenseMtx **pmtxX, DenseMtx **pmtxB) ;
```

   This method creates a linear system $AX = B$ for a natural factor formulation of a `n1` $\times$ `n2` $\times$ `n3` grid. If `n1`, `n2` and `n3` are all greater than 1, the grid is formed of linear hexahedral elements and the matrix $A$ has `8*n1*n2*n3` rows. If one of `n1`, `n2` and `n3` is equal to 1, the grid is formed of linear quadrilateral elements and the matrix $A$ has `4*n1*n2*n3` rows. The entries in $A$ and $X$ are random numbers, $B$ is computed as the product of $A$ with $X$. $A$ can be real (`type = 1`) or complex (`type = 2`). The number of columns of $X$ is given by `nrhs`. The linear system is ordered using theoretical nested dissection, and the front tree is transformed using the `maxzeros` and `maxsize` parameters. The addresses of the front tree, symbolic factorization, and three matrix objects are returned in the last five arguments of the calling sequence.

   *Error checking:* None presently.

## 41.2 Driver programs found in the `Misc` directory

This section contains brief descriptions of the driver programs.

1. `testNDperm msglvl msgFile n1 n2 n3 outPermFile`

   This driver program generates a `Perm` object that contains a nested dissection ordering for a `n1 x n2 x n3` regular grid.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Perm` object is written to the output file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of points in the first direction.
   - `n2` is the number of points in the second direction.
   - `n3` is the number of points in the third direction.
   - The `outPermFile` parameter is the output file for the `Perm` object. If `outPermFile` is `none` then the `Perm` object is not written to a file. Otherwise, the `Perm_writeToFile()` method is called to write the object to a formatted file (if `outPermFile` is of the form `*.permf`), or a binary file (if `outPermFile` is of the form `*.permb`).

2. `testOrderViaMMD msglvl msgFile GraphFile seed ETreeFile`

   This program reads in a `Graph` object from a file and computes a multiple minimum degree ordering of the graph.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Perm` object is written to the output file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
```

- The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `seed` parameter is a random number seed.
- The `ETreeFile` parameter is the output file for the `ETree` object. If `ETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `ETreeFile` is of the form `*.etreef`), or a binary file (if `ETreeFile` is of the form `*.etreeb`).

3. `testOrderViaND msglvl msgFile GraphFile maxdomainsize seed ETreeFile`

   This program reads in a `Graph` object from a file and computes a generalized nested dissection ordering of the graph.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Perm` object is written to the output file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `maxdomainsize` parameter governs the partition of a graph. If a subgraph has more than `maxdomainsize` vertices, it is split.
   - The `seed` parameter is a random number seed.
   - The `ETreeFile` parameter is the output file for the `ETree` object. If `ETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `ETreeFile` is of the form `*.etreef`), or a binary file (if `ETreeFile` is of the form `*.etreeb`).

4. `testOrderViaMS msglvl msgFile GraphFile maxdomainsize seed ETreeFile`

   This program reads in a `Graph` object from a file and computes a multisection ordering of the graph.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Perm` object is written to the output file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `inGraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
   - The `maxdomainsize` parameter governs the partition of a graph. If a subgraph has more than `maxdomainsize` vertices, it is split.
   - The `seed` parameter is a random number seed.
   - The `ETreeFile` parameter is the output file for the `ETree` object. If `ETreeFile` is `none` then the `ETree` object is not written to a file. Otherwise, the `ETree_writeToFile()` method is called to write the object to a formatted file (if `ETreeFile` is of the form `*.etreef`), or a binary file (if `ETreeFile` is of the form `*.etreeb`).

5. ```
   drawGraph msglvl msgFile inGraphFile inCoordsFile inTagsIVfile
             outEPSfile linewidth1 linewidth2 bbox[4] rect[4] radius
   ```

   This driver program generates a Encapsulated Postscript file `outEPSfile` of a 2-D graph using a `Graph` object, a `Coords` object and a tags `IV` object that contains the component ids of the vertices.

   See the `doDraw` script file in this directory for an example calling sequence.
   ```

- The msglvl parameter determines the amount of output — taking msglvl >= 3 means that all objects are written to the output file.
- The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The inGraphFile parameter is the input file for the Graph object. It must be of the form *.graphf or *.graphb. The Graph object is read from the file via the Graph_readFromFile() method.
- The inCoordsFile parameter is the input file for the Coords object. It must be of the form *.coordsf or *.coordsb. The Coords object is read from the file via the Coords_readFromFile() method.
- The inTagsIVfile parameter is the input file for the tags IV object. It must be of the form 'none', *.ivf or *.ivb. The IV object is read from the file via the IV_readFromFile() method.
- The outEPSfile parameter is the output file for the Encapsulated Postscript file.
- The linewidth1 parameter governs the linewidth of edges between vertices in the same component.
- The linewidth2 parameter governs the linewidth of edges between vertices in different components.
- The bbox[4] array is the bounding box for the plot. In Postscript the coordinates are in *points*, where there are 72 points per inch. For example, a bounding box of 0 0 200 300 will create a plot whose size is 2.78 inches by 4.17 inches.
- The rect[4] array is the enclosing rectangle for the plot. To put a 20 point margin around the plot, set rect[0] = bbox[0] + 20, rect[1] = bbox[1] + 20, rect[2] = bbox[2] - 20 and rect[3] = bbox[3] - 20.
- The radius parameter governs the size of the filled circle that is centered on each vertex. The dimension is in points.

See Figure 41.1 for a plot of the graph of R2D100, a randomly triangulated grid with 100 vertices with linewidth1 = 3. Figure 41.2 illustrates a domain decomposition obtained from the fishnet algorithm of Chapter 20 with linewidth1 = 3 and linewidth2 = 0.1.

6. testSemi msglvl msgFile GraphFile ETreeFile mapFile

This program is used to compute the effect of using a semi-implicit factorization to solve

$$AX = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} B_0 \\ B_1 \end{bmatrix} = B.$$

A is factored as

$$\begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} = \begin{bmatrix} L_{0,0} & 0 \\ L_{1,0} & L_{1,1} \end{bmatrix} \begin{bmatrix} U_{0,0} & U_{0,1} \\ 0 & U_{1,1} \end{bmatrix},$$

and to solve $AX = B$, we do the following steps.

- solve $L_{0,0}Y_0 = B_0$
- solve $L_{1,1}U_{1,1}X_1 = B_1 - L_{1,0}Y_0$
- solve $U_{0,0}X_0 = Y_0 - U_{0,1}X_1$

An alternative factorization is

$$A = \begin{bmatrix} L_{0,0} & 0 \\ A_{1,0}U_{0,0}^{-1} & L_{1,1} \end{bmatrix} \begin{bmatrix} U_{0,0} & L_{0,0}^{-1}U_{0,1} \\ 0 & U_{1,1} \end{bmatrix}.$$

To solve $AX = B$, we do the following *semi-implicit solve*.

Figure 41.1: R2D100

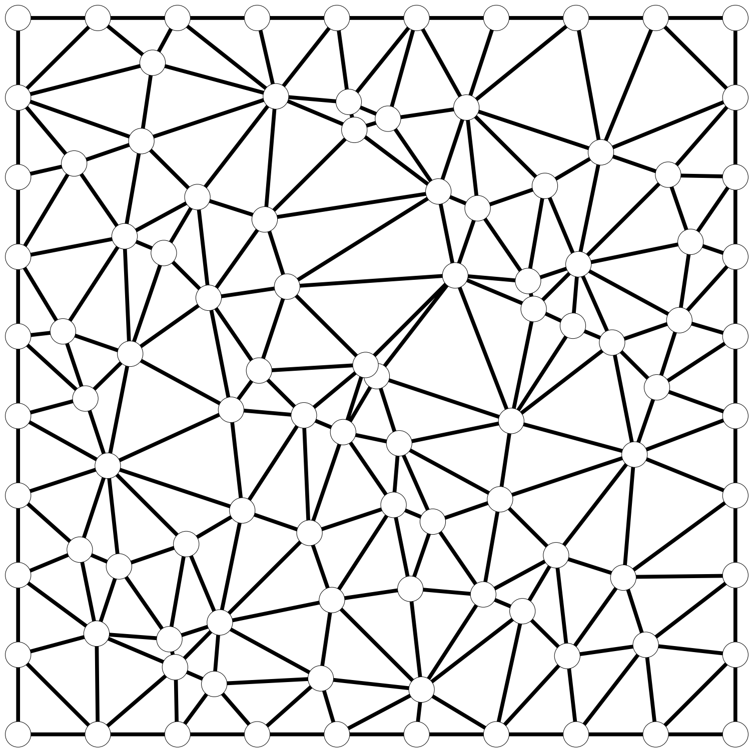
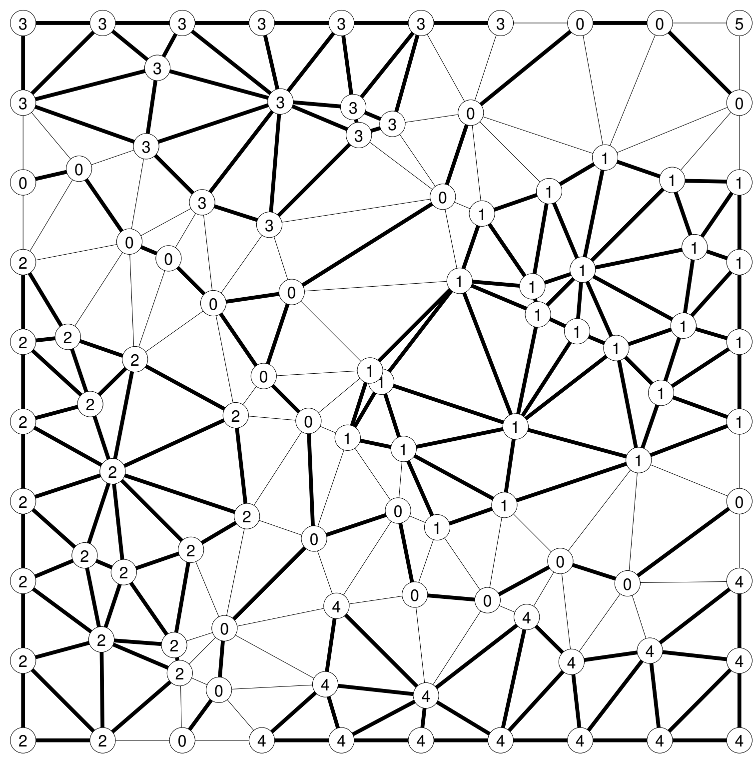


Figure 41.2: R2D100: fishnet domain decomposition



- solve $L_{0,0}U_{0,0}Z_0 = B_0$
- solve $L_{1,1}U_{1,1}X_1 = B_1 - A_{1,0}Z_0$
- solve $L_{0,0}U_{0,0}X_0 = B_0 - A_{0,1}X_1$

When we compare the semi-implicit solve against the explicit solve, we see that the former needs $A_{0,1}$ and $A_{1,0}$ but not $L_{1,0}$ or $A_{0,1}$. and executes two solves with $L_{0,0}$ and $U_{0,0}$ (instead of one) and performs a matrix-matrix multiply with $A_{0,1}$ and $A_{1,0}$ instead of $L_{1,0}$ and $U_{0,1}$. In situations where the numbers of entries in $L_{1,0}$ and $U_{0,1}$ are much larger than those in $A_{1,0}$ and $A_{0,1}$, and the numbers of entries in $L_{0,0}$ and $U_{0,0}$ are not too large, the semi-implicit factorization can be more efficient.

This program reads in three objects: a `Graph` object, an `ETree` object to specify the ordering, and an `IV` map object that tells which vertices are in the which blocks of the matrix. The map from vertices to blocks follows the same convention as the *component map* from the `GPart` object. If `map[v] = 0`, then vertex `v` belongs to the Schur complement $(1, 1)$ block. Otherwise, `v` belongs to a domain (the domain number is `map[v]`) and so belongs to the $(0, 0)$ block. The output of the program gives statistics for storage and operation count for the two types of solves. For example,

```
storage: explicit = 1404, semi-implicit = 1063, ratio = 1.321
opcount: explicit = 2808, semi-implicit = 2742, ratio = 1.024
```

is the output using the `do_testSemi` driver program for the `R2D100` matrix.

- The `msglvl` parameter determines the amount of output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `GraphFile` parameter is the input file for the `Graph` object. It must be of the form `*.graphf` or `*.graphb`. The `Graph` object is read from the file via the `Graph_readFromFile()` method.
- The `ETreeFile` parameter is the input file for the `ETree` object. It must be of the form `*.etreef` or `*.etreeb`. The `ETree` object is read from the file via the `ETree_readFromFile()` method.
- The `mapFile` parameter is the input file for the map `IV` object. It must be of the form `*.ivf` or `*.ivb`. The `IV` object is read from the file via the `IV_readFromFile()` method.

7. ```
   allInOne msglvl msgFile type symmetryflag pivotingflag
            matrixFileName rhsFileName seed
   ```

   This *all-in-one* driver program is an example that tests the serial $U^T DU$, $U^H DU$ or LU factorization and solve. Matrix entries are read in from a file, and then the matrix is assembled and factored. The right hand side entries are read in from a file, and the system is solved. Three input parameters specify the type of system (real or complex), the type of factorization (symmetric, Hermitian or nonsymmetric) and whether pivoting is to be used for numerical stability.

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Perm` object is written to the output file.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `type` is the type of entries
     - `1` — (`SPOOLES_REAL`) for real entries
     - `2` — (`SPOOLES_COMPLEX`) for complex entries
   - `symmetryflag` defines the factorization
     - `0` — (`SPOOLES_SYMMETRIC`) for a real or complex $U^T DU$ factorization
   ```

  - 1 — (SPOOLES_SYMMETRIC) for a complex $U^H DU$ factorization
  - 2 — (SPOOLES_SYMMETRIC) for a real or complex LU factorization

- `pivotingflag` defines pivoting or not for numerical stability
  - 0 — (SPOOLES_NO_PIVOTING) for no pivoting
  - 1 — (SPOOLES_PIVOTING) for pivoting

  Note, the code has a pivoting threshold `tau = 100` hardwired into the code.
- The `matrixFileName` parameter is the name of the input file for the matrix entries. For a real matrix, this file must have the following form.

  ```
  nrow ncol nent
  ...
  irow jcol value
  ...
  ```

  where the first line has the number of rows, columns and entries. (Note, for this driver program `nrow` must be equal to `ncol` since we are factoring a square matrix.) Each of the `nent` following lines contain one nonzero entry. For a complex matrix, the file has this structure.

  ```
  nrow ncol nent
  ...
  irow jcol real_value imag_value
  ...
  ```

  For both real and complex entries, the entries need not be disjoint, i.e., entries with the same `irow` and `jcol` values are *summed*.
- The `rhsFileName` parameter is the name of the input file for the right hand side matrix. It has the following structure

  ```
  nrow nrhs
  ...
  irow value_0 value_1 ... value_\{nrhs-1\}
  ...
  ```

  Note, `nrow` need not be the number of equations, here it is the number of nonzero right hand side entries. This allows us to input sparse right hand sides without specifying the zeroes. In contrast to the input for the matrix entries, the nonzero rows *must* be unique. The right hand side entries are not assembled into a dense matrix object, but placed into the object.
- `seed` is a random number seed used for the ordering process.

8. `patchAndGo msglvl msgFile type symmetryflag patchAndGoFlag fudge toosmall`

   `storeids storevalues matrixFileName rhsFileName seed`

   This driver program is used to test the "patch-and-go" functionality for a factorization without pivoting. When small diagonal pivot elements are found, one of three actions are taken. See the `PatchAndGoInfo` object for more information.

   The program reads in a matrix A and right hand side B, generates the graph for A and orders the matrix, factors A and solves the linear system $AX = B$ for X using multithreaded factors and solves. Use the script file `do_patchAndGo` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.

- The `type` parameter specifies a real or complex linear system.
  - `type = 1 (SPOOLES_REAL)` for real,
  - `type = 2 (SPOOLES_COMPLEX)` for complex.
- The `symmetryflag` parameter specifies the symmetry of the matrix.
  - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,
  - `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
  - `type = 2 (SPOOLES_NONSYMMETRIC)`

  for A real or complex nonsymmetric.
- The `patchAndGoFlag` specifies the "patch-and-go" strategy.
  - `patchAndGoFlag = 0` — if a zero pivot is detected, stop computing the factorization, set the error flag and return.
  - `patchAndGoFlag = 1` — if a small or zero pivot is detected, set the diagonal entry to 1 and the offdiagonal entries to zero.
  - `patchAndGoFlag = 2` — if a small or zero pivot is detected, perturb the diagonal entry.
- The `fudge` parameter is used to perturb a diagonal entry.
- The `toosmall` parameter is judge when a diagonal entry is small.
- If `storeids = 1`, then the locations where action was taken is stored in an `IV` object.
- If `storevalues = 1`, then the perturbations are stored in an `DV` object.
- The `matrixFileName` parameter is the name of the files where the matrix entries are read from. The file has the following structure.

  ```
  neqns neqns nent
  irow jcol entry
  ...  ...  ...
  ```

  where `neqns` is the global number of equations and `nent` is the number of entries in this file. There follows `nent` lines, each containing a row index, a column index and one or two floating point numbers, one if real, two if complex.
- The `rhsFileName` parameter is the name of the files where the right hand side entries are read from. The file has the following structure.

  ```
  nrow nrhs
  irow entry ... entry
  ...  ...   ... ...
  ```

  where `nrow` is the number of rows in this file and `nrhs` is the number of rigght and sides. There follows `nrow` lines, each containing a row index and either `nrhs` or `2*nrhs` floating point numbers, the first if real, the second if complex.
- The `seed` parameter is a random number seed.

9. `QRallInOne msglvl msgFile type matrixFileName rhsFileName seed`

   This *all-in-one* driver program is an example that tests the serial QR factorization and solve. Matrix entries are read in from a file, and then the matrix is assembled and factored. The right hand side entries are read in from a file, and the system is solved. One input parameter specifies the type of system (real or complex).

   - The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `Perm` object is written to the output file.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- `type` is the type of entries
  - `1` — (`SPOOLES_REAL`) for real entries
  - `2` — (`SPOOLES_COMPLEX`) for complex entries
- The `matrixFileName` parameter is the name of the input file for the matrix entries. For a real matrix, this file must have the following form.

  ```
  nrow ncol nent
  ...
  irow jcol value
  ...
  ```

  where the first line has the number of rows, columns and entries. Each of the `nent` following lines contain one nonzero entry. For a complex matrix, the file has this structure.

  ```
  nrow nrhs nent
  ...
  irow jcol real_value imag_value
  ...
  ```

  For both real and complex entries, the entries need not be disjoint, i.e., entries with the same `irow` and `jcol` values are *summed*.
- The `rhsFileName` parameter is the name of the input file for the right hand side matrix. It has the following structure

  ```
  nrow nrhs
  ...
  irow value_0 value_1 ... value_\{nrhs-1\}
  ...
  ```

  Note, `nrow` need not be the number of equations, here it is the number of nonzero right hand side entries. This allows us to input sparse right hand sides without specifying the zeroes. In contrast to the input for the matrix entries, the nonzero rows *must* be unique. The right hand side entries are not assembled into a dense matrix object, but placed into the object.
- `seed` is a random number seed used for the ordering process.

Part VI

Multithreaded Methods

Chapter 42

`MT` directory

All methods that use multithreaded function calls are found in this directory. Three functionalities are presently supported: matrix-matrix multiplies, sparse factorizations, and solves.

The multithreaded methods to compute $Y := Y + \alpha AX$, $Y := Y + \alpha A^T X$ and $Y := Y + \alpha A^H X$ are simple. Their calling sequences are almost identical to their serial counterparts: global data structures for Y, α, A and X are followed by the number of threads, a message level and file. Thread q accesses part of A, part of X, and computes its own $Y^q = \alpha AX$ using those entries of A that it is responsible for. This work is done independently by all threads. The global summation $Y := Y + \sum_q Y^q$ is done in serial mode by the calling process.

This approach is not scalable. A better approach would be to explicitly partition A into local A^q matrices, and use local X^q and Y^q to hold rows of X and Y that have support with A^q, as is done with the distributed MPI matrix-matrix multiplies. (With MPI there is added complexity since X and Y are distributed among processors.)

A matrix-matrix multiply does not exist in isolation. For example, a block shifted eigensolver requires factorizations of $A - \sigma B$ and multiplies using A or B. The data structure for the matrix that takes part in the multiply needs to toggle back and for between its forms for the factor and multiply. Managing this in a distributed environment is actually easier than a multithreaded environment, for A and B are already distributed. Our multithreaded factorization expects A and B in global form. Insisting that A and B be partitioned as A^q and B^q matrices is too great a burden for the user that has no need for a multithreaded matrix-matrix multiply. Allowing the A^q matrices to overlap or point into the global A matrix in a persistent fashion is not cleanly possible, but requires changes to the `InpMtx` object.

In the future we intend to provide a scalable multithreaded matrix-matrix multiply. It requires a more in-depth consideration of the issues involved than we are able to give it at the present time.

The multithreaded factorizations $A = LU$ and $A = QR$ are very similar to the serial factorizations, in both the calling sequence visible to the user and in the underlying code structure. The only additional parameters in the calling sequence is a map from the fronts to the threads that defines who does what computation, and a *lookahead* parameter that allows some ability to control and reduce the idle time during the factorization. Inside the code, the deterministic post-order traversal of the serial factorization is replaced by independent topological traversals of the front tree. It is the list and working storage data structures (the `ChvList`, `ChvManager` and `SubMtxManager` objects) that have locks. *What* is done is common code between the serial and multithreaded environments, it is the choreography, i.e., *who* does what, that differs.

Most of these same comments apply to the multithreaded solve methods. The calling sequences between the serial and multithreaded solves differs by one parameter, a `SolveMap` object that maps the submatrices of the factor matrix to the threads that will compute with them.

42.1 Data Structure

There are no multithreaded specific data structures. See the `Lock` object which is used to hide the particular mutual exclusion device used by a thread library.

42.2 Prototypes and descriptions of `MT` methods

This section contains brief descriptions including prototypes of all methods found in the `MT` source directory.

42.2.1 Matrix-matrix multiply methods

There are five methods to multiply a vector times a dense matrix. The first three methods, called `InpMtx_MT_nonsym_mmm*()`, are straightforward, $y := y + \alpha Ax$, where A is nonsymmetric, and α is real (if A is real) and complex (if A is complex). The fourth method, `InpMtx_MT_sym_mmm()`, is used when the matrix is real symmetric or complex symmetric, though it is not necessary that only the lower or upper triangular entries are stored. (If one fills the `InpMtx` object with only the entries in the lower triangle of A, and then permute the matrix PAP^T, the entries will not generally be found in only the lower or upper triangle. However, the code is still correct.) The last method, `InpMtx_MT_herm_mmm()`, is used when the matrix is complex hermitian.

1. ```
   void InpMtx_MT_nonsym_mmm ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X,
                               int nthread, int msglvl, int msgFile ) ;
   void InpMtx_MT_sym_mmm ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X,
                               int nthread, int msglvl, int msgFile ) ;
   void InpMtx_MT_herm_mmm ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X,
                               int nthread, int msglvl, int msgFile ) ;
   ```

   These methods compute the matrix-vector product $y := y + \alpha Ax$, where y is found in the `Y DenseMtx` object, α is real or complex in `alpha[]`, A is found in the `A Inpmtx` object, and x is found in the `X DenseMtx` object. If any of the input objects are `NULL`, an error message is printed and the program exits. `A`, `X` and `Y` must all be real or all be complex. When `A` is real, then α = `alpha[0]`. When `A` is complex, then α = `alpha[0]` + i* `alpha[1]`. This means that one cannot call the methods with a constant as the third parameter, e.g., `InpMtx_MT_nonsym_mmm(A, Y, 3.22, X, nthread, msglvl, msgFile)`, for this may result in a segmentation violation. The values of α must be loaded into an array of length 1 or 2. The number of threads is specified by the `nthread` parameter; if, `nthread` is `1`, the serial method is called. The `msglvl` and `msgFile` parameters are used for diagnostics during the creation of the threads' individual data structures.

   Error checking: If `A`, `Y` or `X` are `NULL`, or if `coordType` is not `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, or if `storageMode` is not one of `INPMTX_RAW_DATA`, `INPMTX_SORTED` or `INPMTX_BY_VECTORS`, or if `inputMode` is not `SPOOLES_REAL` or `SPOOLES_COMPLEX`, an error message is printed and the program exits.

2. ```
   void InpMtx_MT_nonsym_mmm_T ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X,
                               int nthread, int msglvl, int msgFile ) ;
   ```

   This method computes the matrix-vector product $y := y + \alpha A^T x$, where y is found in the `Y DenseMtx` object, α is real or complex in `alpha[]`, A is found in the `A Inpmtx` object, and x is found in the `X DenseMtx` object. If any of the input objects are `NULL`, an error message is printed and the program exits. `A`, `X` and `Y` must all be real or all be complex. When `A` is real, then α = `alpha[0]`. When `A` is complex, then α = `alpha[0]` + i* `alpha[1]`. This means that one cannot call the methods with a constant as the third parameter, e.g., `InpMtx_MT_nonsym_mmm(A, Y, 3.22, X, nthread, msglvl, msgFile)`, for this may result in a segmentation violation. The values of α must be loaded into an array of length 1 or 2. The number of threads is specified by the `nthread` parameter; if, `nthread` is `1`, the serial method is called. The `msglvl` and `msgFile` parameters are used for diagnostics during the creation of the threads' individual data structures.

Error checking: If `A`, `Y` or `X` are `NULL`, or if `coordType` is not `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, or if `storageMode` is not one of `INPMTX_RAW_DATA`, `INPMTX_SORTED` or `INPMTX_BY_VECTORS`, or if `inputMode` is not `SPOOLES_REAL` or `SPOOLES_COMPLEX`, an error message is printed and the program exits.

3. `void InpMtx_MT_nonsym_mmm_H ( InpMtx *A, DenseMtx *Y, double alpha[], DenseMtx *X, int nthread, int msglvl, int msgFile ) ;`

   This method computes the matrix-vector product $y := y + \alpha A^H x$, where y is found in the `Y DenseMtx` object, α is complex in `alpha[]`, A is found in the `A Inpmtx` object, and x is found in the `X DenseMtx` object. If any of the input objects are `NULL`, an error message is printed and the program exits. `A`, `X` and `Y` must all be complex. The number of threads is specified by the `nthread` parameter; if, `nthread` is `1`, the serial method is called. The `msglvl` and `msgFile` parameters are used for diagnostics during the creation of the threads' individual data structures.

   Error checking: If `A`, `Y` or `X` are `NULL`, or if `coordType` is not `INPMTX_BY_ROWS`, `INPMTX_BY_COLUMNS` or `INPMTX_BY_CHEVRONS`, or if `storageMode` is not one of `INPMTX_RAW_DATA`, `INPMTX_SORTED` or `INPMTX_BY_VECTORS`, or if `inputMode` is not `SPOOLES_COMPLEX`, an error message is printed and the program exits.

42.2.2 Multithreaded Factorization methods

1. ```
   Chv * FrontMtx_MT_factorInpMtx ( FrontMtx *frontmtx, InpMtx *inpmtx,
                double tau, double droptol, ChvManager *chvmanager,
                IV *ownersIV, int lookahead, double cpus[], int stats[],
                int msglvl, FILE *msgFile ) ;
   Chv * FrontMtx_MT_factorPencil ( FrontMtx *frontmtx, Pencil *pencil,
                double tau, double droptol, ChvManager *chvmanager,
                IV *ownersIV, int lookahead, double cpus[], int stats[],
                int msglvl, FILE *msgFile ) ;
   ```

   These two methods compute a multithreaded factorization for a matrix A (stored in `inpmtx`) or a matrix pencil $A + \sigma B$ (stored in `pencil`). The `tau` parameter is used when pivoting is enabled, each entry in U and L (when nonsymmetric) will have magnitude less than or equal to `tau`. The `droptol` parameter is used when the fronts are stored in a sparse format, each entry in U and L (when nonsymmetric) will have magnitude greater than or equal to `droptol`. The map from fronts to owning processes is found in `ownersIV`. The `lookahead` parameter governs the "upward–looking" nature of the computations. Choosing `lookahead = 0` is usually the most conservative with respect to working storage, while positive values increase the working storage and sometimes decrease the factorization time. On return, the `cpus[]` vector is filled with the following information.

   - `cpus[0]` — time spent managing working storage.
   - `cpus[1]` — time spent initializing the fronts and loading the original entries.
   - `cpus[2]` — time spent accumulating updates from descendents.
   - `cpus[3]` — time spent inserting aggregate fronts.
   - `cpus[4]` — time spent removing and assembling aggregate fronts.
   - `cpus[5]` — time spent assembling postponed data.
   - `cpus[6]` — time spent to factor the fronts.
   - `cpus[7]` — time spent to extract postponed data.
   ```

- cpus[8] — time spent to store the factor entries.
- cpus[9] — miscellaneous time.

On return, the stats[] vector is filled with the following information.

- stats[0] — number of pivots.
- stats[1] — number of pivot tests.
- stats[2] — number of delayed rows and columns.
- stats[3] — number of entries in D.
- stats[4] — number of entries in L.
- stats[5] — number of entries in U.
- stats[6] — number of locks of the FrontMtx object.
- stats[7] — number of locks of aggregate list.
- stats[8] — number of locks of postponed list.

Error checking: If frontmtx, inpmtxA, cpus or stats is NULL, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

42.2.3 Multithreaded QR Factorization method

1.
```
void FrontMtx_MT_QR_factor ( FrontMtx *frontmtx, InpMtx *mtxA,
                             ChvManager *chvmanager, IV *ownersIV, double cpus[],
                             double *pfacops,  int msglvl, FILE *msgFile ) ;
```

   This method computes the $(U^T + I)D(I + U)$ factorization of $A^T A$ if A is real or $(U^H + I)D(I + U)$ factorization of $A^H A$ if A is complex. The chvmanager object manages the working storage. The map from fronts to threads is found in ownersIV. On return, the cpus[] vector is filled as follows.

   - cpus[0] – time to set up the factorization.
   - cpus[1] – time to set up the fronts.
   - cpus[2] – time to factor the matrices.
   - cpus[3] – time to scale and store the factor entries.
   - cpus[4] – time to store the update entries
   - cpus[5] – miscellaneous time
   - cpus[6] – total time

   On return, *pfacops contains the number of floating point operations done by the factorization.

   Error checking: If frontmtx, frontJ or chvmanager is NULL, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

42.2.4 Multithreaded Solve method

1.
```
void FrontMtx_MT_solve ( FrontMtx *frontmtx, DenseMtx *mtxX, DenseMtx *mtxB,
                         SubMtxManager *mtxmanager, SolveMap *solvemap,
                         double cpus[], int msglvl, FILE *msgFile ) ;
```

   This method is used to solve one of three linear systems of equations using a multithreaded solve — $(U^T + I)D(I + U)X = B$, $(U^H + I)D(I + U)X = B$ or $(L + I)D(I + U)X = B$. Entries of B are *read* from mtxB and entries of X are written to mtxX. Therefore, mtxX and mtxB can be the same object. (Note, this does not hold true for an MPI factorization with pivoting.) The submatrix manager object manages the working storage. The solvemap object contains the map from submatrices to threads. The map from fronts to processes that own them is given in the ownersIV object. On return the cpus[] vector is filled with the following. The stats[] vector is not currently used.

- cpus[0] — set up the solves
- cpus[1] — fetch right hand side and store solution
- cpus[2] — forward solve
- cpus[3] — diagonal solve
- cpus[4] — backward solve
- cpus[5] — total time in the method.

Error checking: If frontmtx, rhsmtx, mtxmanager, solvemap, cpus or stats is NULL, or if msglvl ¿ 0 and msgFile is NULL, an error message is printed and the program exits.

42.2.5 Multithreaded QR Solve method

1.
```
void FrontMtx_MT_QR_solve ( FrontMtx *frontmtx, InpMtx *mtxA, DenseMtx *mtxX,
              DenseMtx *mtxB, SubMtxManager *mtxmanager, SolveMap *solvemap,
              double cpus[], int msglvl, FILE *msgFile ) ;
```

   This method is used to minimize $\|B - AX\|_F$, where A is stored in mtxA, B is stored in mtxB, and X will be stored in mtxX. The frontmtx object contains a $(U^T + I)D(I + U)$ factorization of $A^T A$ if A is real or $(U^H + I)D(I + U)$ factorization of $A^H A$ if A is complex. We solve the seminormal equations $(U^T + I)D(I + U)X = A^T B$ or $(U^H + I)D(I + U)X = A^H B$ for X. On return the cpus[] vector is filled with the following.

   - cpus[0] — set up the solves
   - cpus[1] — fetch right hand side and store solution
   - cpus[2] — forward solve
   - cpus[3] — diagonal solve
   - cpus[4] — backward solve
   - cpus[5] — total time in the solve method.
   - cpus[6] — time to compute $A^T B$ or $A^H B$.
   - cpus[7] — total time.

   Only the solve is presently done in parallel.

   Error checking: If frontmtx, mtxA, mtxX, mtxB, mtxmanager, solvemap or cpus is NULL, or if msglvl ¿ 0 and msgFile is NULL, an error message is printed and the program exits.

42.3 Driver programs for the multithreaded functions

1.
```
allInOneMT msglvl msgFile type symmetryflag pivotingflag
           matrixFileName rhsFileName seed nthread
```

   This driver program reads in a matrix A and right hand side B, generates the graph for A and orders the matrix, factors A and solves the linear system $AX = B$ for X using multithreaded factors and solves. Use the script file do_gridMT for testing.

- The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `type` parameter specifies a real or complex linear system.
  - `type = 1 (SPOOLES_REAL)` for real,
  - `type = 2 (SPOOLES_COMPLEX)` for complex.
- The `symmetryflag` parameter specifies the symmetry of the matrix.
  - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,
  - `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
  - `type = 2 (SPOOLES_NONSYMMETRIC)`

  for A real or complex nonsymmetric.
- The `pivotingflag` parameter signals whether pivoting for stability will be enabled or not.
  - If `pivotingflag = 0 (SPOOLES_NO_PIVOTING)`, no pivoting will be done.
  - If `pivotingflag = 1 (SPOOLES_PIVOTING)`, pivoting will be done to ensure that all entries in U and L have magnitude less than `tau`.
- The `matrixFileName` parameter is the name of the files where the matrix entries are read from. The file has the following structure.

  ```
  neqns neqns nent
  irow jcol entry
  ...  ...  ...
  ```

  where `neqns` is the global number of equations and `nent` is the number of entries in this file. There follows `nent` lines, each containing a row index, a column index and one or two floating point numbers, one if real, two if complex.
- The `rhsFileName` parameter is the name of the files where the right hand side entries are read from. The file has the following structure.

  ```
  nrow nrhs
  irow entry ... entry
  ...  ...   ... ...
  ```

  where `nrow` is the number of rows in this file and `nrhs` is the number of rigght and sides. There follows `nrow` lines, each containing a row index and either `nrhs` or `2*nrhs` floating point numbers, the first if real, the second if complex.
- The `seed` parameter is a random number seed.
- The `nthread` parameter is the number of threads.

2. `patchAndGoMT msglvl msgFile type symmetryflag patchAndGoFlag fudge toosmall storeids storevalues matrixFileName rhsFileName seed nthread`

   This driver program is used to test the "patch-and-go" functionality for a factorization without pivoting. When small diagonal pivot elements are found, one of three actions are taken. See the `PatchAndGoInfo` object for more information.

   The program reads in a matrix A and right hand side B, generates the graph for A and orders the matrix, factors A and solves the linear system $AX = B$ for X using multithreaded factors and solves. Use the script file `do_patchAndGo` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.

- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `type` parameter specifies a real or complex linear system.
  - `type = 1 (SPOOLES_REAL)` for real,
  - `type = 2 (SPOOLES_COMPLEX)` for complex.
- The `symmetryflag` parameter specifies the symmetry of the matrix.
  - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,
  - `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
  - `type = 2 (SPOOLES_NONSYMMETRIC)`

  for A real or complex nonsymmetric.
- The `patchAndGoFlag` specifies the "patch-and-go" strategy.
  - `patchAndGoFlag = 0` — if a zero pivot is detected, stop computing the factorization, set the error flag and return.
  - `patchAndGoFlag = 1` — if a small or zero pivot is detected, set the diagonal entry to 1 and the offdiagonal entries to zero.
  - `patchAndGoFlag = 2` — if a small or zero pivot is detected, perturb the diagonal entry.
- The `fudge` parameter is used to perturb a diagonal entry.
- The `toosmall` parameter is judge when a diagonal entry is small.
- If `storeids = 1`, then the locations where action was taken is stored in an `IV` object.
- If `storevalues = 1`, then the perturbations are stored in an `DV` object.
- The `matrixFileName` parameter is the name of the files where the matrix entries are read from. The file has the following structure.

  ```
  neqns neqns nent
  irow jcol entry
  ...   ...   ...
  ```

  where `neqns` is the global number of equations and `nent` is the number of entries in this file. There follows `nent` lines, each containing a row index, a column index and one or two floating point numbers, one if real, two if complex.
- The `rhsFileName` parameter is the name of the files where the right hand side entries are read from. The file has the following structure.

  ```
  nrow nrhs
  irow entry ... entry
  ...   ...    ... ...
  ```

  where `nrow` is the number of rows in this file and `nrhs` is the number of rigght and sides. There follows `nrow` lines, each containing a row index and either `nrhs` or `2*nrhs` floating point numbers, the first if real, the second if complex.
- The `seed` parameter is a random number seed.
- The `nthread` parameter is the number of threads.

3. `testMMM msglvl msgFile dataType symflag storageMode transpose`
   `nrow ncol nitem nrhs seed alphaReal alphaImag nthread`

   This driver program generates A, a `nrow`×`ncol` matrix using `nitem` input entries, X and Y, `nrow`×`nrhs` matrices, is filled with random numbers. It then computes $Y + \alpha * A * X$, $Y + \alpha * A^T * X$ or $Y + \alpha * A^H * X$. The program's output is a file which when sent into Matlab, outputs the error in the computation.

- The `msglvl` parameter determines the amount of output — taking `msglvl >= 3` means the `InpMtx` object is written to the message file.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- `dataType` is the type of entries, `0` for real, `1` for complex.
- `symflag` is the symmetry flag, `0` for symmetric, `1` for Hermitian, `2` for nonsymmetric.
- `storageMode` is the storage mode for the entries, `1` for by rows, `2` for by columns, `3` for by chevrons.
- `transpose` determines the equation, `0` for $Y + \alpha * A * X$, `1` for $Y + \alpha * A^T * X$ or `2` for $Y + \alpha * A^H * X$.
- `nrowA` is the number of rows in A
- `ncolA` is the number of columns in A
- `nitem` is the number of matrix entries that are assembled into the matrix.
- `nrhs` is the number of columns in X and Y.
- The `seed` parameter is a random number seed used to fill the matrix entries with random numbers.
- `alphaReal` and `alphaImag` form the scalar in the multiply.
- `nthread` is the number of threads to use.

4.
```
testGridMT msglvl msgFile n1 n2 n3 maxzeros maxsize seed type
         symmetryflag sparsityflag pivotingflag tau droptol
         nrhs nthread maptype cutoff lookahead
```

   This driver program tests the serial `FrontMtx_MT_factor()` and `FrontMtx_MT_solve()` methods for the linear system $AX = B$. The factorization and solve are done in parallel. Use the script file `do_gridMT` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of points in the first grid direction.
   - `n2` is the number of points in the second grid direction.
   - `n3` is the number of points in the third grid direction.
   - `maxzeros` is used to merge small fronts together into larger fronts. Look at the `ETree` object for the `ETree_mergeFronts{One,All,Any}()` methods.
   - `maxsize` is used to split large fronts into smaller fronts. See the `ETree_splitFronts()` method.
   - The `seed` parameter is a random number seed.
   - The `type` parameter specifies a real or complex linear system.
     - `type = 1 (SPOOLES_REAL)` for real,
     - `type = 2 (SPOOLES_COMPLEX)` for complex.
   - The `symmetryflag` parameter specifies the symmetry of the matrix.
     - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,
     - `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
     - `type = 2 (SPOOLES_NONSYMMETRIC)`

     for A real or complex nonsymmetric.
   - The `sparsityflag` parameter signals a direct or approximate factorization.

- `sparsityflag = 0 (FRONTMTX_DENSE_FRONTS)` implies a direct factorization, the fronts will be stored as dense submatrices.
- `sparsityflag = 1 (FRONTMTX_SPARSE_FRONTS)` implies an approximate factorization. The fronts will be stored as sparse submatrices, where the entries in the triangular factors will be subjected to a drop tolerance test — if the magnitude of an entry is `droptol` or larger, it will be stored, otherwise it will be dropped.

- The `pivotingflag` parameter signals whether pivoting for stability will be enabled or not.
  - If `pivotingflag = 0 (SPOOLES_NO_PIVOTING)`, no pivoting will be done.
  - If `pivotingflag = 1 (SPOOLES_PIVOTING)`, pivoting will be done to ensure that all entries in U and L have magnitude less than `tau`.
- The `tau` parameter is an upper bound on the magnitude of the entries in L and U when pivoting is enabled.
- The `droptol` parameter is a lower bound on the magnitude of the entries in L and U when the approximate factorization is enabled.
- The `nrhs` parameter is the number of right hand sides to solve as one block.
- The `nthread` parameter is the number of threads.
- The `maptype` parameter determines the type of map from fronts to processes to be used during the factorization
  - 1 – wrap map
  - 2 – balanced map
  - 3 – subtree-subset map
  - 4 – domain decomposition map
  - 5 – improved domain decomposition map

  See the `ETree` methods for constructing maps.
- The `cutoff` parameter is used for domain decomposition maps. We try to construct domains (each domain is owned by a single thread) that contain $0 \leq$ `cutoff` ≤ 1 of the rows and columns of the matrix. Try to choose `cutoff` to be `1/nthread` or `1/(2*nthread)`.
- The `lookahead` parameter controls the degree that a thread will look past a stalled front in order to do some useful work. lookahead = 0 implies a thread will not look ahead, while `lookahead = k` implies a thread will look `k` ancestors up the front tree to find useful work. Bewarned, while a thread is doing useful work further up the tree, the stalled front may be ready, so large values of lookahead can be detrimental to a fast computation. In addition, a positive value of `lookahead` means a larger storage footprint taken by the factorization.

5. `testQRgridMT msglvl msgFile n1 n2 n3 seed nrhs type nthread maptype cutoff`

   This driver program tests the serial `FrontMtx_QR_factor()` and `FrontMtx_QR_solve()` methods for the least squares problem $\min_X \|F - AX\|_F$. The factorization and solve are done in parallel.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - `n1` is the number of points in the first grid direction.
   - `n2` is the number of points in the second grid direction.
   - `n3` is the number of points in the third grid direction.

- The `seed` parameter is a random number seed.
- The `nrhs` parameter is the number of right hand sides to solve as one block.
- The `type` parameter specifies a real or complex linear system.
  - `type = 1 (SPOOLES_REAL)` for real,
  - `type = 2 (SPOOLES_COMPLEX)` for complex.
- The `nthread` parameter is the number of threads.
- The `maptype` parameter determines the type of map from fronts to processes to be used during the factorization
  - 1 – wrap map
  - 2 – balanced map
  - 3 – subtree-subset map
  - 4 – domain decomposition map
  - 5 – improved domain decomposition map

  See the `ETree` methods for constructing maps.
- The `cutoff` parameter is used for domain decomposition maps. We try to construct domains (each domain is owned by a single thread) that contain $0 \leq$ `cutoff` ≤ 1 of the rows and columns of the matrix. Try to choose `cutoff` to be `1/nthread` or `1/(2*nthread)`.

Part VII

MPI Methods

Chapter 43

MPI directory

All methods that use MPI constructs are found in this directory. There are a remarkably small number when one considers that the numeric functionality of this library has been extended to a distributed memory system. Most of the necessary data structures exist equally well as a *global* object for a serial or multithreaded application or as a *distributed* object for a distributed memory application.

There is very little new numeric code in this directory. The "chores" — what is done and how — is unchanged from the serial codes. The "choreography" — who does what when — is unchanged from the multithreaded codes. All that was necessary to add the explicit message passing demanded by the MPI environment.

All communication is "safe", meaning that the programs will complete when using an MPI implementation that conforms to the standard. We use non-blocking communication, i.e., communication calls that are guaranteed to complete, namely `MPI_Alltoall()`, `MPI_Sendrecv()`, `MPI_Bcast()`, `MPI_Allgather()`, `MPI_Irecv()` and `MPI_Isend()`.

43.1 Data Structure

There is one MPI specific data structure, used in the distributed matrix-matrix multiply.

43.1.1 `MatMulInfo` : Matrix-matrix multiply information object

The distributed matrix-matrix multiply is a very complex operation. We want to compute $Y := Y + \alpha AX$, where Y, A and X are distributed matrices. Processor q owns the local matrices Y^q, A^q and X^q. The entries of A^q do not travel among the processors, it is the entries of X and/or the partial entries of the product αAX that are communicated. Each processor performs the local computation $Y^q_{supp} = \alpha A^q X^q_{supp}$, where the rows of X^q_{supp} correspond to the columns of A^q with a nonzero entry, and the rows of Y^q_{supp} correspond to the rows of A^q with a nonzero entry. (Something similar holds for the operations $Y := Y + \alpha A^T X$ and $Y := Y + \alpha A^T X$.) This requires entries of X to be *gathered* into X^q_{supp} and the entries of Y^q_{supp} be *scatter/added* into Y.

The `MatMulInfo` object stores all the necessary information to make this happen. There is one `MatMulInfo` object per processor. It has the following fields.

- `symflag` — symmetry flag for A
  - 0 (`SPOOLES_SYMMETRIC`) – symmetric matrix
  - 1 (`SPOOLES_HERMITIAN`) – hermitian matrix

  - 2 (SPOOLES_NONSYMMETRIC) – nonsymmetric matrix

- `opflag` — operation flag for the multiply
  - 0 (MMM_WITH_A) — perform $Y := Y + \alpha AX$
  - 1 (MMM_WITH_AT) — perform $Y := Y + \alpha A^T X$
  - 2 (MMM_WITH_AH) — perform $Y := Y + \alpha A^H X$
- `IV *XownedIV` — list of rows of X that are owned by this processor, these form the rows of X^q.
- `IV *XsupIV` — list of rows of X that are accessed by this processor, these form the rows of X^q_{supp}
- `IV *XmapIV` — a map from the global ids of the rows of X^q_{supp} to their local ids within X^q_{supp}
- `IVL *XsendIVL` — list r holds the local row ids of the owned rows of X^q that must be sent from this processor to processor r
- `IVL *XrecvIVL` — list r holds the local row ids of the supported rows of X^q_{supp} that will be received from processor r.
- `IV *YownedIV` — list of rows of Y that are owned by this processor, these form the rows of Y^q.
- `IV *YsupIV` — list of rows of Y that are updated by this processor, these form the rows of Y^q_{supp}
- `IV *YmapIV` — a map from the global ids of the rows of Y^q_{supp} to their local ids within Y^q_{supp}
- `IVL *YsendIVL` — list r holds the local row ids of the supported rows of Y^q_{supp} that must be sent from this processor to processor r
- `IVL *YrecvIVL` — list r holds the local row ids of the owned rows of Y^q that will be received from processor r.
- `DenseMtx *Xsupp` — a temporary data structure to hold X^q_{supp}.
- `DenseMtx *Ysupp` — a temporary data structure to hold Y^q_{supp}.

See the methods `MatMul_MPI_setup()`, `MatMul_setLocalIndices()`, `MatMul_setGlobalIndices()`, `MatMul_MPI_mmm()` and `MatMul_cleanup()` which use the `MatMulInfo` data object.

43.2 Prototypes and descriptions of MPI methods

This section contains brief descriptions including prototypes of all methods found in the `MPI` source directory.

43.2.1 Split and redistribution methods

In a distributed environment, data must be distributed, and sometimes during a computation, data must be re-distributed. These methods split and redistribute four data objects.

1.
```
void DenseMtx_MPI_splitByRows ( DenseMtx *mtx, IV *mapIV, int stats[], int msglvl,
                                FILE *msgFile, int firsttag, MPI_Comm comm ) ;
```
   This method splits and redistributes the `DenseMtx` object based on the `mapIV` object that maps rows to processes. The messages that will be sent require `nproc` consecutive tags — the first is the parameter `firsttag`. On return, the `stats[]` vector contains the following information.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

Note, the values in `stats[]` are *incremented*, i.e., the `stats[]` vector is not zeroed at the start of the method, and so can be used to accumulated information with multiple calls.

Error checking: If `mtx` or `rowmapIV` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `firsttag < 0` or `firsttag + nproc` is larger than the largest available tag, an error message is printed and the program exits.

2. `DenseMtx * DenseMtx_MPI_splitFromGlobalByRows ( DenseMtx *Xglobal, DenseMtx *Xlocal, IV *rowmapIV, int root, int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   This method is used when the `Xglobal DenseMtx` matrix object is owned by processor `root` and redistributed to the other processors.

   `Xglobal` is pertinent only to processor `root`. If the local matrix `Xlocal` is `NULL`, and if the local matrix will be nonempty, then it is created. If the local matrix is not `NULL`, then it will be returned. The remaining input arguments are the same as for the `DenseMtx_MPI_splitByRows()` method.

   Error checking: Processor `root` does a fair amount of error checking — it ensures that `Xglobal` is valid, that `firsttag` is valid, and that the `rowmapIV` object is valid. The return code is broadcast to the other processors. If an error is found, the processors call `MPI_Finalize()` and exit.

3. `DenseMtx * DenseMtx_MPI_mergeToGlobalByRows ( DenseMtx *Xglobal, DenseMtx *Xlocal, IV *rowmapIV, int root, int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   This method is used when the processors own a partitioned `DenseMtx` object and it must be assembled onto the `root` processor. Each processor owns a `Xlocal` matrix (which may be `NULL`). The global matrix will be accumulated in the `Xglobal` object.

   `Xglobal` is pertinent only to processor `root`. If the global matrix `Xglobal` is `NULL`, and if the global matrix will be nonempty, then it is created. If the global matrix is not `NULL`, then it will be returned. The remaining input arguments are the same as for the `DenseMtx_MPI_splitByRows()` method.

   Error checking: Each processor does a fair amount of error checking — they ensure that `firsttag` is valid, that the types of the local matrices are identical, and that the number of columns of the local matrices are identical, If there is any error detected by any of the processors, they call `MPI_Finalize()` and exit.

4. `void InpMtx_MPI_split ( InpMtx *inpmtx, IV *mapIV, int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   This method splits and redistributes the `InpMtx` object based on the `mapIV` object that maps the `InpMtx` object's vectors (rows, columns or chevrons) to processes. The the vectors are defined by the first coordinate of the `InpMtx` object. For the distributed LU, U^TDU and U^HDU factorizations, we use the chevron coordinate type to store the matrix entries. This method will redistribute a matrix by rows if the coordinate type is `1` (for rows) and `mapIV` is a row map. Similarly, this method will redistribute a matrix by columns if the coordinate type is `2` (for columns) and `mapIV` is a column map. See the `InpMtx` object for details. The messages that will be sent require `nproc` consecutive tags — the first is the parameter `firsttag`. On return, the `stats[]` vector contains the following information.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

Note, the values in `stats[]` are *incremented*, i.e., the `stats[]` vector is not zeroed at the start of the method, and so can be used to accumulated information with multiple calls.

Error checking: If `firsttag < 0` or `firsttag + nproc` is larger than the largest available tag, an error message is printed and the program exits.

5. ```
   InpMtx * InpMtx_MPI_splitFromGlobal ( InpMtx *Aglobal, InpMtx *Alocal,
                                          IV *mapIV, int root, int stats[], int msglvl,
                                          FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   ```

   This method is used when the `Aglobal` `InpMtx` matrix object is owned by processor `root` and redistributed to the other processors.

   `Aglobal` is pertinent only to processor `root`. If the local matrix `Alocal` is `NULL`, and if the local matrix will be nonempty, then it is created. If the local matrix is not `NULL`, then it will be returned. The remaining input arguments are the same as for the `InpMtx_MPI_split()` method.

   Error checking: Processor `root` does a fair amount of error checking — it ensures that `Aglobal` is valid, that `firsttag` is valid, and that the `mapIV` object is valid. The return code is broadcast to the other processors. If an error is found, the processors call `MPI_Finalize()` and exit.

6. ```
   void Pencil_MPI_split ( Pencil *pencil, IV *mapIV, int tag, int stats[],
                           int msglvl, FILE *msgFile, MPI_Comm comm ) ;
   ```

   This method splits and redistributes the matrix pencil based on the `mapIV` object that maps rows and columns to processes. This is a simple wrapper around the `InpMtx_MPI_split()` method. The messages that will be sent require `2*nproc` consecutive tags — the first is the parameter `firsttag`. On return, the `stats[]` vector contains the following information.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   Note, the values in `stats[]` are *incremented*, i.e., the `stats[]` vector is not zeroed at the start of the method, and so can be used to accumulated information with multiple calls.

   Error checking: If `firsttag < 0` or `firsttag + 2*nproc` is larger than the largest available tag, an error message is printed and the program exits.

7. ```
   void FrontMtx_MPI_split ( FrontMtx *frontmtx, SolveMap *solvemap, int stats[],
                             int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   ```

   Used after the factorization, this method is used instead of the `FrontMtx_splitUpperMatrices()` and `FrontMtx_splitLowerMatrices()` methods. The method splits and redistributes the `FrontMtx` object based on the `solvemap` object that maps submatrices to processes. The `firsttag` is the first tag that will be used for all messages. Unfortunately, the number of different tags that are necessary is not known prior to entering this method. On return, the `stats[]` vector contains the following information.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   Note, the values in `stats[]` are *incremented*, i.e., the `stats[]` vector is not zeroed at the start of the method, and so can be used to accumulated information with multiple calls.

   Error checking: If `mtx` or `rowmapIV` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `firsttag < 0` is larger than the largest available tag, an error message is printed and the program exits.
   ```

43.2.2 Gather and scatter methods

These method gather and scatter/add rows of `DenseMtx` objects. These operations are performed during the distributed matrix-matrix multiply. The gather operation $X^q_{supp} \leftarrow X$ is performed by `DenseMtx_MPI_gatherRows()`, while the scatter/add operation $Y^q := Y^q + \sum_r Y^r_{supp}$ is performed by `DenseMtx_MPI_scatterAddRows()`.

1.
```
void DenseMtx_MPI_gatherRows ( DenseMtx *Y, DenseMtx *X, IVL *sendIVL,
                               IVL *recvIVL, int stats[], int msglvl,
                               FILE *msgFile, int tag, MPI_Comm comm) ;
```
   This method is used to gather rows of `X`, a globally distributed matrix, into `Y`, a local matrix. List q of `sendIVL` contains the local row ids of the local part of `X` that will be sent to processor q. List q of `recvIVL` contains the local row ids of `Y` that will be received from processor q.

   This method uses tags in the range `[tag,tag+nproc*nproc)`. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   This method is *safe* in the sense that it uses only non-blocking sends and receives, `MPI_Isend()` and `MPI_Irecv()`.

   Error checking: If `Y`, `X`, `sendIVL` or `recvIVL` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `tag < 0` or `tag + nproc*nproc` is larger than the largest available tag, an error message is printed and the program exits.

2.
```
void DenseMtx_MPI_scatterAddRows ( DenseMtx *Y, DenseMtx *X, IVL *sendIVL,
                                   IVL *recvIVL, int stats[], int msglvl,
                                   FILE *msgFile, int tag, MPI_Comm comm) ;
```
   This method is used to scatter/add rows of `X`, a globally distributed matrix, into `Y`, a local matrix. List q of `sendIVL` contains the local row ids of the local part of `X` that will be sent to processor q. List q of `recvIVL` contains the local row ids of `Y` that will be received from processor q.

   This method uses tags in the range `[tag,tag+nproc*nproc)`. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   This method is *safe* in the sense that it uses only non-blocking sends and receives, `MPI_Isend()` and `MPI_Irecv()`.

   Error checking: If `Y`, `X`, `sendIVL` or `recvIVL` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `tag < 0` or `tag + nproc*nproc` is larger than the largest available tag, an error message is printed and the program exits.

43.2.3 Symbolic Factorization methods

1.
```
IVL * SymbFac_MPI_initFromInpMtx ( ETree *etree, IV *frontOwnersIV,
                                   InpMtx *inpmtx, int stats[], int msglvl,
                                   FILE *msgFile, int firsttag, MPI_Comm comm ) ;
IVL * SymbFac_MPI_initFromPencil ( ETree *etree, IV *frontOwnersIV,
                                   Pencil *pencil, int stats[], int msglvl,
                                   FILE *msgFile, int firsttag, MPI_Comm comm ) ;
```

These methods are used in place of the `Symbfac_initFrom{InpMtx,Pencil}()` methods to compute the symbolic factorization. The `ETree` object is assumed to be replicated over the processes. The `InpMtx` and `Pencil` objects are partitioned among the processes. Therefore, to compute the `IVL` object that contains the symbolic factorization is a distributed, cooperative process. At the end of the symbolic factorization, each process will own a portion of the `IVL` object. The `IVL` object is neither replicated nor partitioned (except in trivial cases), but the `IVL` object on each process contains just a portion, usually not much more than what it needs to know for its part of the factorization and solves.

This method uses tags in the range `[tag,tag+nfront)`. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

This method is *safe* in the sense that it uses only non-blocking sends and receives, `MPI_Isend()` and `MPI_Irecv()`.

Error checking: If `etree`, `inpmtx`, `pencil` or `frontOwnersIV` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `tag < 0` or `tag + nfront` is larger than the largest available tag, an error message is printed and the program exits.

43.2.4 Numeric Factorization methods

1. ```
   Chv * FrontMtx_MPI_factorPencil ( FrontMtx *frontmtx, Pencil *pencil, double tau,
                             double droptol, ChvManager *chvmanager, IV *frontOwnersIV,
                             int lookahead, int *perror, double cpus[], int stats[],
                             int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   Chv * FrontMtx_MPI_factorInpMtx ( FrontMtx *frontmtx, InpMtx *inpmtx, double tau,
                             double droptol, ChvManager *chvmanager, IV *frontOwnersIV,
                             int lookahead, int *perror, double cpus[], int stats[],
                             int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   ```

   These methods are used to compute the numeric factorization and are very similar to the multithreaded `FrontMtx_MT_factorPencil()` and `FrontMtx_MT_factorInpMtx()` methods. All that has been added is the code to send and receive the `Chv` messages. The input `firsttag` parameter is used to tag the messages during the factorization. This method uses tags in the range `[firsttag, firsttag + 3*nfront + 3)`.

   On return, `*perror` holds an error flag. If the factorization completed without any error detected, `*perror` will be negative. Otherwise it holds the id of a front where the factorization failed. Currently, this can happen only if pivoting is not enabled and a zero pivot was detected.

   The return value is a pointer to a list of `Chv` objects that hold entries of the matrix that could not be factored. This value should be `NULL` in all cases. We have left this return behavior as a hook for future implementation of a multi-stage factorization.

   On return, the `cpus[]` vector has the following information.

`cpus[0]`	–	initialize fronts	`cpus[7]`	–	extract postponed data
`cpus[1]`	–	load original entries	`cpus[8]`	–	store factor entries
`cpus[2]`	–	update fronts	`cpus[9]`	–	post initial receives
`cpus[3]`	–	insert aggregate data	`cpus[10]`	–	check for received messages
`cpus[4]`	–	assemble aggregate data	`cpus[11]`	–	post initial sends
`cpus[5]`	–	assemble postponed data	`cpus[12]`	–	check for sent messages
`cpus[6]`	–	factor fronts			
   ```

On return, the stats[] vector has the following information.

stats[0]	—	# of pivots
stats[1]	—	# of pivot tests
stats[2]	—	# of delayed rows and columns
stats[3]	—	# of entries in D
stats[4]	—	# of entries in L
stats[5]	—	# of entries in U
stats[6]	—	# of aggregate messages sent
stats[7]	—	# of bytes sent in aggregate messages
stats[8]	—	# of aggregate messages received
stats[9]	—	# of bytes received in aggregate messages
stats[10]	—	# of postponed messages sent
stats[11]	—	# of bytes sent in postponed messages
stats[12]	—	# of postponed messages received
stats[13]	—	# of bytes received in postponed messages
stats[14]	—	# of active Chv objects (working storage)
stats[15]	—	# of active bytes in working storage
stats[16]	—	# of requested bytes for working storage

Error checking: If frontmtx, pencil, frontOwnersIV, cpus or stats is NULL, or if tau < 1.0 or droptol < 0.0, or if firsttag < 0 or firsttag + 3*nfront + 2 is larger than the largest available tag, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

43.2.5 Post-processing methods

1. ```
   void FrontMtx_MPI_postProcess ( FrontMtx *frontmtx, IV *frontOwnersIV, int stats[],
                                   int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   ```
   After the factorization is complete, the factor matrices are split into submatrices. This method replaces the serial FrontMtx_postProcess() method. The messages that will be sent require at most 5*nproc consecutive tags — the first is the parameter firsttag.

   Error checking: If frontmtx, frontOwnersIV or stats is NULL, or if firsttag < 0 or firsttag + 5*nproc, is larger than the largest available tag, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

2. ```
   void FrontMtx_MPI_permuteUpperAdj ( FrontMtx *frontmtx, IV *frontOwnersIV, int stats[],
                                       int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   void FrontMtx_MPI_permuteLowerAdj ( FrontMtx *frontmtx, IV *frontOwnersIV, int stats[],
                                       int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   ```
   If pivoting takes place during the factorization, the off diagonal blocks of the factor matrices must be permuted prior to being split into submatrices. To do this, the final rows and columns of the factor matrix must be made known to the different processors. The messages that will be sent require at most nproc consecutive tags — the first is the parameter firsttag.

   Error checking: If frontmtx, frontOwnersIV or stats is NULL, or if firsttag < 0 or firsttag + nproc, is larger than the largest available tag, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

3. ```
   void IV_MPI_allgather ( IV *iv, IV *ownersIV, int stats[], int msglvl,
                           FILE *msgFile, int firsttag, MPI_Comm comm ) ;
   ```
   After a factorization with pivoting, the frontsizesIV object needs to be made global on each processor. This methods takes the individual entries of an IV object whose owners are specified by the ownersIV
   ```

object, and communicates the entries around the processors until the global IV object is present on each. The messages that will be sent require at most nproc consecutive tags — the first is the parameter firsttag.

Error checking: If iv, ownersIV or stats is NULL, or if firsttag < 0 or firsttag + nproc, is larger than the largest available tag, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

4. `void IVL_MPI_allgather ( IVL *ivl, IV *ownersIV, int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   When the FrontMtx object is split into submatrices, each processor accumulates the structure of the block matrix for the fronts its owns. This structure must be global to all processors before the submatrix map can be computed. This method takes a *partitioned* IVL object and communicates the entries among the processors until the global IVL object is present on each. Which processor owns what lists of the IVL object is given by the ownersIV object. The messages that will be sent require at most nproc consecutive tags — the first is the parameter firsttag.

   Error checking: If ivl, ownersIV or stats is NULL, or if firsttag < 0 or firsttag + nproc, is larger than the largest available tag, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

43.2.6 Numeric Solve methods

1. `void FrontMtx_MPI_solve ( FrontMtx *frontmtx, DenseMtx *mtxX, DenseMtx *mtxB, SubMtxManager *mtxmanager, SolveMap *solvemap, double cpus[], int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   This method is used to compute the forward and backsolves. Its structure is very, very similar to the multithreaded FrontMtx_MT_solve() method. All that has been added is the code to send and receive the SubMtx messages. The method uses tags in the range [firsttag, firsttag + 2*nfront). On return, the cpus[] vector has the following information.

cpus[0]	—	setup the solves	cpus[3]	—	diagonal solve
cpus[1]	—	load rhs and store solution	cpus[4]	—	backward solve
cpus[2]	—	forward solve	cpus[5]	—	miscellaneous

   On return, the following statistics will have been added.

stats[0]	—	# of solution messages sent
stats[1]	—	# of aggregate messages sent
stats[2]	—	# of solution bytes sent
stats[3]	—	# of aggregate bytes sent
stats[4]	—	# of solution messages received
stats[5]	—	# of aggregate messages received
stats[6]	—	# of solution bytes received
stats[7]	—	# of aggregate bytes received

   Error checking: If frontmtx, mtxX, mtxB, mtxmanager, solvemap, cpus or stats is NULL, or if firsttag < 0 or firsttag + 2*nfront is larger than the largest available tag, or if msglvl > 0 and msgFile is NULL, an error message is printed and the program exits.

43.2.7 Matrix-matrix multiply methods

The usual sequence of events is as follows.

- Set up the data structure via a call to `MatMul_MPI_setup()`.
- Convert the local A^q matrix to local indices via a call to `MatMul_setLocalIndices()`.
- Compute the matrix-matrix multiply with a call to `MatMul_MPI_mmm()`. Inside this method, the MPI methods `DenseMtx_MPI_gatherRows()` and `DenseMtx_MPI_scatterAddRows()` are called, along with a serial `InpMtx` matrix-matrix multiply method.
- Clean up and free data structures via a call to `MatMul_cleanup()`.
- Convert the local A^q matrix to global indices via a call to `MatMul_setGlobalIndices()`.

1. ```
   MatMulInfo * MatMul_MPI_setup ( InpMtx *A, int symflag, int opflag,
                                   IV *XownersIV, IV *YownersIV int stats[],
                                   int msglvl, FILE *msgFile, MPI_Comm comm) ;
   ```

   This method is used to set up and return the `MatMulInfo` data structure that stores the information for the distributed matrix-matrix multiply. The `symflag` parameter specifies the symmetry of the matrix.

   - 0 (`SPOOLES_SYMMETRIC`)
   - 1 (`SPOOLES_HERMITIAN`)
   - 2 (`SPOOLES_NONSYMMETRIC`)

   The `opflag` parameter specifies what type of operation will be performed.

   - 0 (`MMM_WITH_A`) — $Y := Y + \alpha AX$
   - 1 (`MMM_WITH_AT`) — $Y := Y + \alpha A^T X$
   - 2 (`MMM_WITH_AH`) — $Y := Y + \alpha A^H X$

   The `XownersIV` object is the map from the rows of X to their owning processors. The `YownersIV` object is the map from the rows of Y to their owning processors.

   On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   This method calls `makeSendRecvIVLs()`.

   Error checking: If `A`, `XownersIV`, `YownersIV` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

2. ```
   void MatMul_setLocalIndices ( MatMulInfo *info, InpMtx *A ) ;
   void MatMul_setGlobalIndices ( MatMulInfo *info, InpMtx *A ) ;
   ```

   The first method maps the indices of `A` (which are assumed to be global) into local indices. The second method maps the indices of `A` (which are assumed to be local) back into global indices. It uses the `XmapIV`, `XsupIV` `YmapIV` and `YsupIV` objects that are contained in the `info` object. These are serial methods, performed independently on each processor.

   Error checking: If `info` or `A` is `NULL`, an error message is printed and the program exits.

3. `void MatMul_MPI_mmm ( MatMulInfo *info, DenseMtx *Yloc, double alpha[], InpMtx *A, DenseMtx *Xloc, int stats[], int msglvl, FILE *msgFile, MPI_Comm comm) ;`

   This method computes a distributed matrix-matrix multiply $Y := Y + \alpha AX$, $Y := Y + \alpha A^T X$ or $Y := Y + \alpha A^H X$, depending on how the `info` object was set up. NOTE: `A` must have local indices, use `MatMul_setLocalIndices()` to convert from global to local indices. `Xloc` and `Yloc` contain the owned rows of X and Y, respectively.

   On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   This method calls `makeSendRecvIVLs()`.

   Error checking: If `info`, `Yloc`, `alpha`, `A`, `Xloc` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

4. `void MatMul_cleanup ( MatMulInfo *info ) ;`

   This method free's the data structures owned by the `info` object, and then free's the object. processor.

   Error checking: If `info` is `NULL`, an error message is printed and the program exits.

43.2.8 Broadcast methods

1. `ETree * ETree_MPI_Bcast ( ETree *etree, int root, int msglvl, FILE *msgFile, MPI_Comm comm ) ;`

   This method is a broadcast method for an `ETree` object. The `root` processor broadcasts its `ETree` object to the other nodes and returns a pointer to its `ETree` object. A node other than `root` free's its `ETree` object (if not `NULL`), receives `root`'s `ETree` object, and returns a pointer to it.

   Error checking: None presently.

2. `Graph * Graph_MPI_Bcast ( Graph *etree, int root, int msglvl, FILE *msgFile, MPI_Comm comm ) ;`

   This method is a broadcast method for an `Graph` object. The `root` processor broadcasts its `Graph` object to the other nodes and returns a pointer to its `Graph` object. A node other than `root`, clears the data in its Graph object, receives the Graph object from the root and returns a pointer to it.

   Error checking: None presently.

3. `IVL * IVL_MPI_Bcast ( IVL *obj, int root, int msglvl, FILE *msgFile, MPI_Comm comm ) ;`

   This method is a broadcast method for an `IVL` object. The `root` processor broadcasts its `IVL` object to the other nodes and returns a pointer to its `IVL` object. A node other than `root`, clears the data in its IVL object, receives the IVL object from the root and returns a pointer to it.

   Error checking: None presently.

4. `IV * IV_MPI_Bcast ( IV *obj, int root, int msglvl, FILE *msgFile, MPI_Comm comm ) ;`

   This method is a broadcast method for an `IV` object. The `root` processor broadcasts its `IV` object to the other nodes and returns a pointer to its `IV` object. A node other than `root`, clears the data in its IV object, receives the IV object from the root and returns a pointer to it.

   Error checking: None presently.

43.2.9 Utility methods

1. `IVL * InpMtx_MPI_fullAdjacency ( InpMtx *inpmtx, int stats[], int msglvl, FILE *msgFile, MPI_Comm comm ) ;`
   `IVL * Pencil_MPI_fullAdjacency ( Pencil *pencil, int stats[], int msglvl, FILE *msgFile, MPI_Comm comm ) ;`

   These methods are used to return an `IVL` object that contains the full adjacency structure of the graph of the matrix or matrix pencil. The matrix or matrix pencil is distributed among the processes, each process has a *local* portion of the matrix or matrix pencil. The returned `IVL` object contains the structure of the global graph. The `stats[]` vector must have at least four fields. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   Error checking: If `inpmtx`, `pencil` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, an error message is printed and the program exits.

2. `ChvList * FrontMtx_MPI_aggregateList ( FrontMtx *frontmtx, IV *frontOwnersIV, int stats[], int msglvl, FILE *msgFile, int tag, MPI_Comm comm ) ;`

   This method is used in place of the `FrontMtx_aggregateList()` method to initialize the aggregate list object. Since the symbolic factorization data is distributed among the processes, the number of incoming aggregates for a front and the number of different processes contributing to a front — information necessary to initialize the list object — must be computed cooperatively. This method uses `tag` as the message tag for all messages communicated during this method. The `stats[]` vector must have at least four fields. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   Error checking: If `frontmtx` or `frontOwnersIV` is `NULL`, or if `tag < 0` or `tag` is larger than the largest available tag, an error message is printed and the program exits.

3. `IV * FrontMtx_MPI_colmapIV ( FrontMtx *frontmtx, IV *frontOwnersIV, int msglvl, FILE *msgFile, MPI_Comm comm ) ;`
   `IV * FrontMtx_MPI_rowmapIV ( FrontMtx *frontmtx, IV *frontOwnersIV, int msglvl, FILE *msgFile, MPI_Comm comm ) ;`

   For a factorization with pivoting, the elimination of some rows and columns may be delayed from the front that initially contains them to an ancestor front. The solution and right hand side entries would therefore need to be redistributed. To do so requires new row and column maps, maps from the row or column to the processor that owns them. These two methods construct that map. The routine uses the `MPI_Allgather()` and `MPI_Bcast()` methods, so no unique tag values are needed.

   Error checking: None at present.

4. `IVL *`
   `IVL_MPI_alltoall ( IVL *sendIVL, IVL *recvIVL, int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   This method is used during the setup for matrix-vector multiplies. Each processor has computed the vertices it needs from other processors, these lists are contained in `sendIVL`. On return, `recvIVL` contains the lists of vertices this processor must send to all others.

   This method uses tags in the range `[tag,tag+nproc-1)`. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

This method is *safe* in the sense that it uses only `MPI_Sendrecv()`.

Error checking: If `sendIVL` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `tag < 0` or `tag + nproc` is larger than the largest available tag, an error message is printed and the program exits.

5. `void * makeSendRecvIVLs ( IV *supportedIV, IV *globalmapIV, IVL *sendIVL, IVL *recvIVL, int stats[], int msglvl, FILE *msgFile, int firsttag, MPI_Comm comm ) ;`

   The purpose of this method to analyze and organize communication. It was written in support of a distributed matrix-vector multiply but can be used for other applications.

   Each processor has a list of items it "supports" or needs found in the `supportedIV` object. The `globalmapIV` object contains the map from items to owning processors. We need to figure out what items this processor will send to and receive from each other processor. This information is found in the `sendIVL` and `recvIVL` objects.

   On return, list `jproc` of `sendIVL` contains the items owned by this processor and needed by `jproc`. On return, list `jproc` of `recvIVL` contains the items needed by this processor and owned by `jproc`.

   This method initializes the `recvIVL` object, and then calls `IVL_MPI_alltoall()` to construct the `sendIVL` object. This method uses tags in the range `[tag,tag+nproc*nproc)`. On return, the following statistics will have been added.

`stats[0]`	—	# of messages sent	`stats[1]`	—	# of bytes sent
`stats[2]`	—	# of messages received	`stats[3]`	—	# of bytes received

   This method is *safe* in the sense that it uses only `MPI_Sendrecv()`.

   Error checking: If `sendIVL` or `stats` is `NULL`, or if `msglvl > 0` and `msgFile` is `NULL`, or if `tag < 0` or `tag + nproc` is larger than the largest available tag, an error message is printed and the program exits.

6. `int maxTagMPI ( MPI_Comm comm) ;`

   This method returns the maximum tag value for the communicator `comm`.

   Error checking: None at present.

43.3 Driver programs

1. `allInOne msglvl msgFile type symmetryflag pivotingflag seed`

   This driver program is an example program for reading in a linear system and right hand side, ordering the matrix, factoring the matrix, and solving the system. Use the script file `do_AllInOne` for testing.

   The files names for the matrix and right hand side entries are hardcoded. Processor q reads in matrix entries from file `matrix.`q`.input` and right hand side entries from file `rhs.`q`.input`. The format for the matrix files is as follows:

   ```
   neqns neqns nent
   irow jcol entry
   ...  ...  ...
   ```

where `neqns` is the global number of equations and `nent` is the number of entries in this file. There follows `nent` lines, each containing a row index, a column index and one or two floating point numbers, one if real, two if complex. The format for the right hand side file is similar:

```
nrow nrhs
irow entry ... entry
...  ...   ... ...
```

where `nrow` is the number of rows in this file and `nrhs` is the number of rigght and sides. There follows `nrow` lines, each containing a row index and either `nrhs` or `2*nrhs` floating point numbers, the first if real, the second if complex.

- The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `type` parameter specifies whether the linear system is real (`type` = 1) or complex (`type` = 2).
- The `symmetryflag` parameter specifies whether the matrix is symmetric (`symmetryflag` = 0), Hermitian (`symmetryflag` = 1) or nonsymmetric (`symmetryflag` = 2)
- The `pivotingflag` parameter specifies whether pivoting will be performed during the factorization, yes (`symmetryflag` = 0) or no (`symmetryflag` = 2). The pivot tolerance is hardcoded as `tau = 100.0`.
- The `seed` parameter is a random number seed.

2. `patchAndGoMPI msglvl msgFile type symmetryflag patchAndGoFlag`
   `fudge toosmall storeids storevalues seed`

   This driver program is used to test the "patch-and-go" functionality for a factorization without pivoting. When small diagonal pivot elements are found, one of three actions are taken. See the `PatchAndGoInfo` object for more information.

   The program reads in a matrix A and right hand side B, generates the graph for A and orders the matrix, factors A and solves the linear system $AX = B$ for X.

   The files names for the matrix and right hand side entries are hardcoded. Processor q reads in matrix entries from file `patchMatrix.`q`.input` and right hand side entries from file `patchRhs.`q`.input`. The format for the matrix files is as follows:

```
neqns neqns nent
irow jcol entry
...  ...  ...
```

   where `neqns` is the global number of equations and `nent` is the number of entries in this file. There follows `nent` lines, each containing a row index, a column index and one or two floating point numbers, one if real, two if complex. The format for the right hand side file is similar:

```
nrow nrhs
irow entry ... entry
...  ...   ... ...
```

   where `nrow` is the number of rows in this file and `nrhs` is the number of rigght and sides. There follows `nrow` lines, each containing a row index and either `nrhs` or `2*nrhs` floating point numbers, the first if real, the second if complex. Use the script file `do_patchAndGo` for testing.

- The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `type` parameter specifies a real or complex linear system.
  - `type = 1 (SPOOLES_REAL)` for real,
  - `type = 2 (SPOOLES_COMPLEX)` for complex.
- The `symmetryflag` parameter specifies the symmetry of the matrix.
  - `type = 0 (SPOOLES_SYMMETRIC)` for A real or complex symmetric,
  - `type = 1 (SPOOLES_HERMITIAN)` for A complex Hermitian,
  - `type = 2 (SPOOLES_NONSYMMETRIC)`

  for A real or complex nonsymmetric.
- The `patchAndGoFlag` specifies the "patch-and-go" strategy.
  - `patchAndGoFlag = 0` — if a zero pivot is detected, stop computing the factorization, set the error flag and return.
  - `patchAndGoFlag = 1` — if a small or zero pivot is detected, set the diagonal entry to 1 and the offdiagonal entries to zero.
  - `patchAndGoFlag = 2` — if a small or zero pivot is detected, perturb the diagonal entry.
- The `fudge` parameter is used to perturb a diagonal entry.
- The `toosmall` parameter is judge when a diagonal entry is small.
- If `storeids = 1`, then the locations where action was taken is stored in an `IV` object.
- If `storevalues = 1`, then the perturbations are stored in an `DV` object.
- The `seed` parameter is a random number seed.

3. `testGather msglvl msgFile type nrow ncol inc1 inc2 seed`

   This driver program test the `DenseMtx_MPI_gatherRows()` method. Each processor creates part of a distributed matrix X and fills its entries with entries known to all processors. ($X_{j,k} = j + k * \text{nproc}$ if real and $X_{j,k} = j + k * \text{nproc} + i * 2 * (j + k * \text{nproc})$) if complex. The mapping from rows of X to processors is random. Each processor then generates a random vector that contains its rows in a local Y, which will be filled with the corresponding rows of X. The rows of X are then gathered into Y, and the local errors are computed. The global error is written to the results file by processor zero.

   Use the script file `do_gather` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter specifies whether the linear system is real (`type` = 1) or complex (`type` = 2).
   - `nrow` is the number of rows in `X`.
   - `ncol` is the number of columns in `X`.
   - `inc1` is the row increment for `X`.
   - `inc2` is the column increment for `X`.
   - The `seed` parameter is a random number seed.

4. `testGraph_Bcast msglvl msgFile type nvtx nitem root seed`

   This driver program tests the distributed `Graph_MPI_Bcast()` method. Processor `root` generates a random graph of type `type` (see the documentation for the `Graph` object in chapter 21) with `nvtx` vertices. The random graph is constructed via an `InpMtx` object using `nitem` edges. Processor `root` then sends its `Graph` object to the other processors. Each processor computes a checksum for its object, and the error are collected on processor 0. Use the script file `do_Graph_Bcast` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `type` parameter specifies the type of the graph, unweighted, weighted vertices, weighted edges, and combinations.
   - The `nvtx` parameter specifies the number of vertices in the graph.
   - The `nitem` parameter is used to specify the number of edges that form the graph. An upper bound on the number of edges is `nvtx + 2*nitem`.
   - `root` is the root processor for the broadcast.
   - The `seed` parameter is a random number seed.

5. `testGridMPI msglvl msgFile n1 n2 n3 maxzeros maxsize seed type`
   `symmetryflag sparsityflag pivotingflag tau droptol lookahead`
   `nrhs maptype cutoff`

   This driver program creates and solves the linear system $AX = Y$ where the structure of A is from a `n1` $\times$ `n2` $\times$ `n3` regular grid operator and is ordered using nested dissection. The front tree is formed allowing `maxzeros` in a front with a maximum of `maxsize` vertices in a non-leaf front. Process `0` generates the linear system and broadcasts the front tree to the other processes. Using `maptype`, the processes generate the owners map for the factorization in parallel. The A, X and Y matrices are then distributed among the processes. The symbolic factorization is then computed in parallel, the front matrix is initialized, and the factorization is computed in parallel. If pivoting has taken place, the solution and right hand side matrices are redistributed as necessary. The matrix is post-processed where it is converted to a submatrix storage format. Each processor computes the identical solve map, and the front matrix is split among the processes. The linear system is then solved in parallel and the error is computed. Use the script file `do_gridMPI` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `n1` parameter is the number of grid points in the first direction.
   - The `n2` parameter is the number of grid points in the second direction.
   - The `n3` parameter is the number of grid points in the third direction.
   - The `maxzeros` parameter is the maximum number of zero entries allowed in a front.
   - The `maxsize` parameter is the maximum number of internal rows and columns allowed in a front.
   - The `seed` parameter is a random number seed.
   - The `type` parameter specifies whether the linear system is real or complex. Use `1` for real and `2` for complex.
   - The `symmetryflag` parameter denotes the presence or absence of symmetry.

- Use `0` for a real or complex symmetric matrix A. A $(U^T + I)D(I + U)$ factorization is computed.
- Use `1` for a complex Hermitian matrix A. A $(U^H + I)D(I + U)$ factorization is computed.
- Use `2` for a real or complex nonsymmetric matrix A. A $(L + I)D(I + U)$ factorization is computed.

- The `sparsityflag` parameter denotes a direct or approximate factorization. Valid values are `0` for a direct factorization and `1` is for an approximate factorization.
- The `pivotingflag` parameter denotes whether pivoting is to be used in the factorization. Valid values are `0` for no pivoting and `1` to enable pivoting.
- The `tau` parameter is used when pivoting is enabled, in which case it is an upper bound on the magnitude of an entry in the triangular factors L and U.
- The `droptol` parameter is used when an approximate factorization is requested, in which it is a lower bound on the magnitude of an entry in L and U.
- The `lookahead` parameter governs the "upward–looking" nature of the factorization. Choosing `lookahead = 0` is usually the most conservative with respect to working storage, while positive values increase the working storage and sometimes decrease the factorization time.
- The `nrhs` parameter is the number of right hand sides.
- The `maptype` parameter is the type of factorization map.
  - `1` — a wrap map via a post-order traversal
  - `2` — a balanced map via a post-order traversal
  - `3` — a subtree–subset map
  - `4` — a domain decomposition map
- The `cutoff` parameter is used with the domain decomposition map, and specifies the maximum fraction of the vertices to be included into a domain. Try `cutoff` = `1/nproc` or `1/(2*nproc)`.

6. `testIV_allgather msglvl msgFile n seed`

   This driver program tests the distributed `IV_MPI_allgather()` method. Each processor generates the same `owners[]` map and fills an `IV` object with random entries for the entries which it owns. The processors all-gather the entries of the vector so each processor has a copy of the global vector. Each processor computes a checksum of the vector to detect errors. Use the script file `do_IVallgather` for testing.

   - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
   - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The `n` parameter is the length of the vector.
   - The `seed` parameter is a random number seed.

7. `testIVL_alltoall msglvl msgFile n seed`

   This driver program tests the distributed `IVL_MPI_alltoall()` method. This is used by the `makeSendRecvIVLs` method when setting up the distributed matrix-matrix multiply. Each processor constructs a "receive" `IVL` object with `nproc` lists. List `iproc` contains a set of ids of items that this processor will receive from processor `iproc`. The processors then call `IVL_MPI_allgather` to create their "send" `IVL` object, where list `iproc` contains a set of ids of items that this processor will send to processor `iproc`. The set of lists in all the "receive" `IVL` objects is exactly the same as the set of lists in all the "send" objects. This is an "all-to-all" scatter/gather operation. Had the lists be stored contiguously or at least in one block of storage, we could have used the `MPI_Alltoallv()` method.

   Use the script file `do_IVL_alltoall` for testing.

- The msglvl parameter determines the amount of output. Use msglvl = 1 for just timing output.
- The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The n parameter is an upper bound on list size and element value.
- The seed parameter is a random number seed.

8. testIVL_allgather msglvl msgFile nlist seed

   This driver program tests the distributed IVL_MPI_allgather() method. Each processor generates the same owners[] map and fills an IVL object with random entries for the lists which it owns. The processors all-gather the entries of the IVL object so each processor has a copy of the global object. Each processor computes a checksum of the lists to detect errors. Use the script file do_IVLallgather for testing.

   - The msglvl parameter determines the amount of output. Use msglvl = 1 for just timing output.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The nlist parameter is the number of lists.
   - The seed parameter is a random number seed.

9. testIVL_Bcast msglvl msgFile nlist maxlistsize root seed

   This driver program tests the distributed IVL_MPI_Bcast() method. Processor root generates a random IVL object with nlist lists. The size of each list is bounded above by maxlistsize. Processor root then sends its IVL object to the other processors. Each processor computes a checksum for its object, and the error are collected on processor 0. Use the script file do_IVL_Bcast for testing.

   - The msglvl parameter determines the amount of output. Use msglvl = 1 for just timing output.
   - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
   - The nlist parameter specifies the number of lists.
   - The maxlist parameter is an upper bound on the size of each list.
   - root is the root processor for the broadcast.
   - The seed parameter is a random number seed.

10. testMMM msglvl msgFile nrowA ncolA nentA ncolX coordType
    inputMode symflag opflag seed real imag

    This driver program tests the distributed matrix-matrix multiply $Y := Y + \alpha AX$, $Y := Y + \alpha A^T X$ or $Y := Y + \alpha A^H X$. Process zero creates Y, A and X and computes $Z = Y + \alpha AX$, $Z = Y + \alpha A^T X$ or $Z = Y + \alpha A^H X$. Using random maps, it distributes A, X and Y among the other processors. The information structure is created using MatMul_MPI_setup(). The local matrix A^q is mapped to local coordinates. The matrix-matrix multiply is computed, and then all the Y^q local matrices are gathered onto processor zero into Y, which is then compared with Z that was computed using a serial matrix-matrix multiply. The error is written to the message file by processor zero. Use the script file do_MMM for testing.

    - The msglvl parameter determines the amount of output. Use msglvl = 1 for just timing output.
    - The msgFile parameter determines the message file — if msgFile is stdout, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.

- The `nrowA` parameter is the number of rows in A.
- The `ncolA` parameter is the number of columns in A.
- The `nentA` parameter is the number of entries to be put into A.
- The `nrowX` parameter is the number of rows in X.
- The `coordType` parameter defines the coordinate type that will be used during the redistribution. Valid values are `1` for rows, `2` for columns and `3` for chevrons.
- The `inputMode` parameter defines the mode of input. Valid values are `1` for real entries and `2` for complex entries.
- The `symflag` parameter specifies whether the matrix is symmetric (`symflag` $= 0$), Hermitian (`symflag` $= 1$) or nonsymmetric (`symflag` $= 2$)
- The `opflag` parameter specifies the type of multiply, `0` for $Y := Y + \alpha AX$, `1` for $Y := Y + \alpha A^T X$ or `2` for $Y := Y + \alpha A^H X$.
- The `seed` parameter is a random number seed.
- The `real` parameter is the real part of the scalar α.
- The `imag` parameter is the imaginary part of the scalar α, ignored for real entries.

11. `testScatterDenseMtx msglvl msgFile nrow ncol inc1 inc2 seed root`

    This driver program exercises the `DenseMtx_MPI_splitFromGlobalByRows()` method to split or redistribute by rows a `DenseMtx` dense matrix object. Process `root` generates the `DenseMtx` object. A random map is generated (the same map on all processes) and the object is redistributed using this random map. The local matrices are then gathered into a second global matrix on processor `root` and the two are compared. Use the script file `do_ScatterDenseMtx` for testing.

    - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `nrow` parameter is the number of rows for the matrix.
    - The `ncol` parameter is the number of columns for the matrix.
    - The `inc1` parameter is the row increment for the matrix. Valid values are `1` for column major and `ncol` for row major.
    - The `inc2` parameter is the column increment for the matrix. Valid values are `1` for row major and `nrow` for column major.
    - The `seed` parameter is a random number seed.
    - The `root` parameter is the root processor for the scatter and gather.

12. `testScatterInpMtx msglvl msgFile neqns seed`
    `coordType inputMode inInpMtxFile root`

    This driver program tests the distributed `InpMtx_MPI_splitFromGlobal()` method to split a `InpMtx` sparse matrix object. Process `root` reads in the `InpMtx` object. A random map is generated (the same map on all processes) and the object is scattered from processor `root` to the other processors. Use the script file `do_ScatterInpMtx` for testing.

    - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.

- The `neqns` parameter is the number of equations for the matrix.
- The `seed` parameter is a random number seed.
- The `coordType` parameter defines the coordinate type that will be used during the redistribution. Valid values are `1` for rows, `2` for columns and `3` for chevrons.
- The `inputMode` parameter defines the mode of input. Valid values are `0` for indices only, `1` for real entries and `2` for complex entries.
- The `inInpMtxFile` parameter is the name of the file that contain the `InpMtx` object.

13. `testSplitDenseMtx msglvl msgFile nrow ncol inc1 inc2 seed`

    This driver program tests the distributed `DenseMtx_MPI_splitByRows()` method to split or redistribute by rows a `DenseMtx` dense matrix object. Process zero generates the `DenseMtx` object. It is then split among the processes using a wrap map. A random map is generated (the same map on all processes) and the object is redistributed using this random map. Use the script file `do_SplitDenseMtx` for testing.

    - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `nrow` parameter is the number of rows for the matrix.
    - The `ncol` parameter is the number of columns for the matrix.
    - The `inc1` parameter is the row increment for the matrix. Valid values are `1` for column major and `ncol` for row major.
    - The `inc2` parameter is the column increment for the matrix. Valid values are `1` for row major and `nrow` for column major.
    - The `seed` parameter is a random number seed.

14. `testSplitInpMtx msglvl msgFile neqns seed coordType inputMode inInpMtxFile`

    This driver program tests the distributed `InpMtx_MPI_split()` method to split or redistribute a `InpMtx` sparse matrix object. Process zero reads in the `InpMtx` object. It is then split among the processes using a wrap map. A random map is generated (the same map on all processes) and the object is redistributed using this random map. Use the script file `do_SplitInpMtx` for testing.

    - The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
    - The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
    - The `neqns` parameter is the number of equations for the matrix.
    - The `seed` parameter is a random number seed.
    - The `coordType` parameter defines the coordinate type that will be used during the redistribution. Valid values are `1` for rows, `2` for columns and `3` for chevrons.
    - The `inputMode` parameter defines the mode of input. Valid values are `0` for indices only, `1` for real entries and `2` for complex entries.
    - The `inInpMtxFile` parameter is the name of the file that contain the `InpMtx` object.

15. `testSymbFac msglvl msgFile inGraphFile inETreeFile seed`

    This driver program tests the distributed `SymbFac_MPI_initFromInpMtx()` method that forms a `IVL` object that contains the necessary parts of a symbolic factorization for each processor. The program reads in the global `Graph` and `ETree` objects. Each processor creates a global `InpMtx` object from the structure of the graph and computes a global symbolic factorization object using the serial `SymbFac_initFromInpMtx()` method. The processors then compute a map from fronts to processors, and each processor throws away the unowned matrix entries from the `InpMtx` object. The processors then compute their necessary symbolic factorizations in parallel. For a check, they compare the two symbolic factorizations for error. Use the script file `do_symbfac` for testing.

- The `msglvl` parameter determines the amount of output. Use `msglvl = 1` for just timing output.
- The `msgFile` parameter determines the message file — if `msgFile` is `stdout`, then the message file is *stdout*, otherwise a file is opened with *append* status to receive any output data.
- The `inGraphFile` parameter is the input file for the `Graph` object.
- The `inETreeFile` parameter is the input file for the `ETree` object.
- The `seed` parameter is a random number seed.

Bibliography

[1] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17:886–905, 1996.

[2] S. L. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review*, 32:221–251, 1990.

[3] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Sci. Comput.*, 16:1404–1411, 1995.

[4] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[5] C. Ashcraft and J. W. H. Liu. Using domain decompositions to find graph bisectors. *BIT*, 37:506–534, 1997.

[6] C. Ashcraft and J. W. H. Liu. Applications of the Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement. *SIAM J. Matrix Analysis and Applic.*, 19:325–354, 1998.

[7] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software – Practice and Experience*, 23(11):1249–1265, 1993.

[8] M. V. Bhat, W. G. Habashi, J. W. H. Liu, V. N. Nguyen, and M. F. Peeters. A note on nested dissection for rectangular grids. *SIAM J. Matrix Analysis and Applic.*, 14:253–258, 1993.

[9] A. C. Damhaug. *Sparse Solution of Finite Element Equations*. PhD thesis, The Norwegian Institute of Technology, 1992.

[10] I. Duff and J. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 6:302–325, 1983.

[11] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Applications of an element model for Gaussian elimination. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 85–96. Academic Press, 1976.

[12] J. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Analysis and Applic.*, 13:335–356, 1992.

[13] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20:468–489, 1998.

[14] G. Karypis and V. Kumar. Metis 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL *http://www.cs.umn.edu/˜metis*.

[15] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. on Math. Software*, 12:249–264, 1986.

[16] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applic.*, 11:134–172, 1990.

[17] J. W. H. Liu. A generalized envelope method for sparse factorization by rows. *ACM Trans. on Math. Software*, 17:112–129, 1991.

[18] E. Ng and P. Raghavan. Minimum deficiency ordering. In *Second SIAM Conference on Sparse Matrices*, 1996. Conference presentation.

[19] E. Rothberg. Ordering sparse matrices using approximate minimum local fill. In *Second SIAM Conference on Sparse Matrices*, 1996. Conference presentation.

[20] E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal.*, 19:682–695, 1998.

Index

A2_applyQT(), 29
A2_clearData(), 26
A2_column(), 27
A2_columnMajor(), 31
A2_complexEntry(), 27
A2_computeQ(), 28
A2_copy(), 32
A2_copyEntriesToVector(), 33
A2_entries(), 26
A2_extractColumn(), 31
A2_extractColumnDV(), 31
A2_extractColumnZV(), 31
A2_extractRow(), 31
A2_extractRowDV(), 31
A2_extractRowZV(), 31
A2_fillRandomNormal(), 32
A2_fillRandomUniform(), 32
A2_fillWithIdentity(), 32
A2_free(), 26
A2_frobNorm(), 29
A2_inc1(), 26
A2_inc2(), 26
A2_infinityNorm(), 29
A2_infinityNormOfColumn(), 29
A2_infinityNormOfRow(), 30
A2_init(), 28
A2_makeStaircase(), 28
A2_maxabs(), 29
A2_ncol(), 26
A2_new(), 26
A2_nrow(), 26
A2_oneNorm(), 29
A2_oneNormOfColumn(), 29
A2_oneNormOfRow(), 29
A2_permuteColumns(), 30
A2_permuteRows(), 30
A2_pointerToComplexEntry(), 27
A2_pointerToRealEntry(), 27
A2_QRreduce(), 28
A2_readFromBinaryFile(), 33
A2_readFromFile(), 33
A2_readFromFormattedFile(), 33
A2_realEntry(), 27
A2_row(), 27
A2_rowMajor(), 31
A2_setColumn(), 32
A2_setColumnDV(), 32
A2_setColumnZV(), 32
A2_setComplexEntry(), 27
A2_setDefaultFields(), 26
A2_setRealEntry(), 27
A2_setRow(), 31
A2_setRowDV(), 32
A2_setRowZV(), 32
A2_shiftBase(), 30
A2_sizeOf(), 30
A2_sortColumnsUp(), 30
A2_sortRowsUp(), 30
A2_sub(), 33
A2_subA2(), 28
A2_swapColumns(), 33
A2_swapRows(), 33
A2_transpose(), 31
A2_twoNormOfColumn(), 29
A2_twoNormOfRow(), 30
A2_writeForHumanEye(), 34
A2_writeForMatlab(), 34
A2_writeStats(), 34
A2_writeToBinaryFile(), 34
A2_writeToFile(), 34
A2_writeToFormattedFile(), 34
A2_zero(), 32

bicgstabl(), 304
bicgstabr(), 304
BKL_clearData(), 118
BKL_domAdjToSep(), 120
BKL_eval(), 120
BKL_evalfcn(), 120
BKL_evalgain(), 120
BKL_exhSearch(), 120
BKL_fidmat(), 121
BKL_flipDomain(), 119
BKL_free(), 118

BKL_greyCodeDomain(), 119
BKL_init(), 118
BKL_new(), 118
BKL_segColor(), 119
BKL_setColorWeights(), 119
BKL_setDefaultFields(), 118
BKL_setInitPart(), 119
BKL_setRandomColors(), 119
BPG_clearData(), 124
BPG_DMdecomposition(), 126
BPG_DMviaMaxFlow(), 126
BPG_free(), 124
BPG_init(), 125
BPG_initFromColoring(), 125
BPG_levelStructure(), 125
BPG_makeGraphXbyX(), 125
BPG_makeGraphYbyY(), 125
BPG_new(), 124
BPG_pseudoperipheralnode(), 125
BPG_readFromBinaryFile(), 127
BPG_readFromFile(), 126
BPG_readFromFormattedFile(), 126
BPG_setDefaultFields(), 124
BPG_writeForHumanEye(), 127
BPG_writeStats(), 127
BPG_writeToBinaryFile(), 127
BPG_writeToFile(), 127
BPG_writeToFormattedFile(), 127

Chv_addChevron(), 229
Chv_assembleChv(), 229
Chv_assemblePostponedData(), 229
Chv_clearData(), 224
Chv_columnIndices(), 225
Chv_complexEntry(), 226
Chv_copyBigEntriesToVector(), 232
Chv_copyEntriesToVector(), 231
Chv_copyTrailingPortion(), 232
Chv_countBigEntries(), 231
Chv_countEntries(), 230
Chv_diagLocation(), 225
Chv_dimensions(), 224
Chv_entries(), 225
Chv_factorWithNoPivoting(), 230
Chv_factorWithPivoting(), 229
Chv_fastBunchParlettPivot(), 228
Chv_fill11block(), 234
Chv_fill12block(), 234
Chv_fill21block(), 234
Chv_findPivot(), 228
Chv_free(), 224
Chv_frobNorm(), 234
Chv_id(), 224
Chv_init(), 226
Chv_initFromBuffer(), 226
Chv_initWithPointers(), 226
Chv_locationOfComplexEntry(), 226
Chv_locationOfRealEntry(), 225
Chv_maxabs(), 234
Chv_maxabsInChevron(), 230
Chv_maxabsInColumn(), 227
Chv_maxabsInColumn11(), 227
Chv_maxabsInDiagonal11(), 227
Chv_maxabsInRow(), 227
Chv_maxabsInRow11(), 227
Chv_nbytesInWorkspace(), 233
Chv_nbytesNeeded(), 233
Chv_nent(), 225
Chv_new(), 224
Chv_quasimax(), 227
Chv_r1upd(), 230
Chv_r2upd(), 230
Chv_realEntry(), 225
Chv_rowIndices(), 225
Chv_setComplexEntry(), 226
Chv_setDefaultFields(), 224
Chv_setFields(), 233
Chv_setNbytesInWorkspace(), 233
Chv_setRealEntry(), 225
Chv_shift(), 233
Chv_sub(), 234
Chv_swapColumns(), 233
Chv_swapRows(), 232
Chv_swapRowsAndColumns(), 233
Chv_symmetryFlag(), 224
Chv_type(), 224
Chv_updateH(), 228
Chv_updateN(), 228
Chv_updateS(), 228
Chv_workspace(), 225
Chv_writeForHumanEye(), 234
Chv_writeForMatlab(), 234
Chv_zero(), 234
Chv_zeroOffdiagonalOfChevron(), 230
ChvList_addObjectToList(), 242
ChvList_clearData(), 241
ChvList_free(), 241
ChvList_getList(), 242
ChvList_init(), 242
ChvList_isCountZero(), 242
ChvList_isListNonempty(), 242
ChvList_new(), 241

ChvList_setDefaultFields(), 241
ChvList_writeForHumanEye(), 242
ChvManager_clearData(), 244
ChvManager_free(), 245
ChvManager_init(), 245
ChvManager_new(), 244
ChvManager_newObjectOfSizeNbytes(), 245
ChvManager_releaseListOfObjects(), 245
ChvManager_releaseObject(), 245
ChvManager_setDefaultFields(), 244
ChvManager_writeForHumanEye(), 245
Coords_clearData(), 37
Coords_free(), 37
Coords_init(), 37
Coords_init27P(), 37
Coords_init9P(), 37
Coords_max(), 38
Coords_min(), 38
Coords_new(), 36
Coords_readFromBinaryFile(), 39
Coords_readFromFile(), 38
Coords_readFromFormattedFile(), 39
Coords_setDefaultFields(), 37
Coords_setValue(), 38
Coords_sizeOf(), 38
Coords_value(), 38
Coords_writeForHumanEye(), 39
Coords_writeStats(), 39
Coords_writeToBinaryFile(), 39
Coords_writeToFile(), 39
Coords_writeToFormattedFile(), 39
CVcopy(), 84
CVfill(), 84
CVfp80(), 84
CVfprintf(), 84
CVfree(), 84
CVfscanf(), 84
CVinit(), 84
CVinit2(), 84

DDsepInfo_clearData(), 172
DDsepInfo_free(), 172
DDsepInfo_new(), 172
DDsepInfo_setDefaultFields(), 172
DDsepInfo_writeCpuTimes(), 173
DenseMtx_addRow(), 251
DenseMtx_addVectorIntoRow(), 251
DenseMtx_checksums(), 251
DenseMtx_clearData(), 247
DenseMtx_colCopy(), 302
DenseMtx_colDotProduct(), 302
DenseMtx_colGenAxpy(), 302
DenseMtx_colid(), 247
DenseMtx_column(), 249
DenseMtx_columnIncrement(), 247
DenseMtx_columnIndices(), 248
DenseMtx_complexEntry(), 248
DenseMtx_copyRow(), 250
DenseMtx_copyRowAndIndex(), 250
DenseMtx_copyRowIntoVector(), 251
DenseMtx_copyVectorIntoRow(), 251
DenseMtx_dimensions(), 247
DenseMtx_entries(), 248
DenseMtx_fillRandomEntries(), 251
DenseMtx_free(), 247
DenseMtx_frobNorm(), 301
DenseMtx_init(), 249
DenseMtx_initAsSubmatrix(), 249
DenseMtx_initFromBuffer(), 249
DenseMtx_initWithPointers(), 249
DenseMtx_maxabs(), 251
DenseMtx_mmm(), 302
DenseMtx_MPI_gatherRows(), 379
DenseMtx_MPI_mergeToGlobalByRows(), 377
DenseMtx_MPI_scatterAddRows(), 379
DenseMtx_MPI_splitByRows(), 376
DenseMtx_MPI_splitFromGlobalByRows(), 377
DenseMtx_nbytesInWorkspace(), 250
DenseMtx_nbytesNeeded(), 250
DenseMtx_new(), 247
DenseMtx_permuteColumns(), 250
DenseMtx_permuteRows(), 250
DenseMtx_readFromBinaryFile(), 252
DenseMtx_readFromFile(), 252
DenseMtx_readFromFormattedFile(), 252
DenseMtx_realEntry(), 248
DenseMtx_row(), 249
DenseMtx_rowid(), 247
DenseMtx_rowIncrement(), 248
DenseMtx_rowIndices(), 248
DenseMtx_scale(), 251
DenseMtx_setA2(), 249
DenseMtx_setComplexEntry(), 248
DenseMtx_setDefaultFields(), 247
DenseMtx_setFields(), 250
DenseMtx_setNbytesInWorkspace(), 250
DenseMtx_setRealEntry(), 248
DenseMtx_sort(), 250
DenseMtx_sub(), 251
DenseMtx_twoNormOfColumn(), 302
DenseMtx_workspace(), 248
DenseMtx_writeForHumanEye(), 252

DenseMtx_writeForMatlab(), 253
DenseMtx_writeStats(), 252
DenseMtx_writeToBinaryFile(), 252
DenseMtx_writeToFile(), 252
DenseMtx_writeToFormattedFile(), 252
DenseMtx_zero(), 251
Drand_clearData(), 49
Drand_fillDvector(), 50
Drand_fillIvector(), 50
Drand_fillZvector(), 50
Drand_free(), 49
Drand_init(), 49
Drand_new(), 49
Drand_setDefaultFields(), 49
Drand_setNormal(), 50
Drand_setSeed(), 49
Drand_setSeeds(), 49
Drand_setUniform(), 50
Drand_value(), 50
drawGraphEPS(), 351
DSTree_clearData(), 130
DSTree_domainWeight(), 132
DSTree_free(), 130
DSTree_init1(), 130
DSTree_init2(), 131
DSTree_mapIV(), 130
DSTree_MS2stages(), 131
DSTree_MS3stages(), 131
DSTree_ND2stages(), 131
DSTree_NDstages(), 131
DSTree_new(), 130
DSTree_readFromBinaryFile(), 133
DSTree_readFromFile(), 132
DSTree_readFromFormattedFile(), 133
DSTree_renumberViaPostOT(), 132
DSTree_separatorWeight(), 132
DSTree_setDefaultFields(), 130
DSTree_sizeOf(), 132
DSTree_stagesViaDomainWeight(), 132
DSTree_tree(), 130
DSTree_writeForHumanEye(), 133
DSTree_writeStats(), 133
DSTree_writeToBinaryFile(), 133
DSTree_writeToFile(), 133
DSTree_writeToFormattedFile(), 133
DV2isortDown(), 103
DV2isortUp(), 103
DV2qsortDown(), 104
DV2qsortUp(), 104
DV_clearData(), 42
DV_copy(), 45
DV_entries(), 43
DV_entry(), 43
DV_fill(), 45
DV_first(), 45
DV_free(), 42
DV_init(), 43
DV_init1(), 43
DV_init2(), 43
DV_log10profile(), 45
DV_max(), 44
DV_size(), 42
DV_min(), 44
DV_new(), 42
DV_next(), 45
DV_owned(), 42
DV_push(), 44
DV_ramp(), 44
DV_readFromBinaryFile(), 46
DV_readFromFile(), 45
DV_readFromFormattedFile(), 46
DV_setDefaultFields(), 42
DV_setEntry(), 43
DV_setMaxsize(), 43
DV_setSize(), 44
DV_shiftBase(), 44
DV_shuffle(), 44
DV_size(), 42
DV_sizeAndEntries(), 43
DV_sizeOf(), 45
DV_sortDown(), 44
DV_sortUp(), 44
DV_sum(), 44
DV_writeForHumanEye(), 46
DV_writeForMatlab(), 46
DV_writeStats(), 46
DV_writeToBinaryFile(), 46
DV_writeToFile(), 46
DV_writeToFormattedFile(), 46
DV_zero(), 45
DVadd(), 85
DVaxpy(), 85
DVaxpy11(), 86
DVaxpy12(), 86
DVaxpy13(), 86
DVaxpy21(), 86
DVaxpy22(), 85
DVaxpy23(), 85
DVaxpy31(), 85
DVaxpy32(), 85
DVaxpy33(), 85
DVaxpyi(), 86

DVcompress(), 86
DVcopy(), 86
DVdot(), 86
DVdot11(), 88
DVdot12(), 88
DVdot13(), 88
DVdot21(), 87
DVdot22(), 87
DVdot23(), 87
DVdot31(), 87
DVdot32(), 87
DVdot33(), 86
DVdoti(), 88
DVfill(), 88
DVfprintf(), 84
DVfree(), 84
DVfscanf(), 85
DVgather(), 88
DVgatherAddZero(), 88
DVgatherZero(), 88
DVinit(), 84
DVinit2(), 84
DVinvPerm(), 88
DVisascending(), 102
DVisdescending(), 102
DVisortDown(), 103
DVisortUp(), 103
DVIVisortDown(), 103
DVIVisortUp(), 103
DVIVqsortDown(), 104
DVIVqsortUp(), 104
DVmax(), 88
DVmaxabs(), 88
DVmin(), 88
DVminabs(), 89
DVperm(), 89
DVqsortDown(), 104
DVqsortUp(), 104
DVramp(), 89
DVscale(), 89
DVscatter(), 89
DVscatterAdd(), 89
DVscatterAddZero(), 89
DVscatterZero(), 89
DVshuffle(), 90
DVsub(), 89
DVsum(), 89
DVsumabs(), 89
DVswap(), 89
DVzero(), 90
EGraph_clearData(), 136
EGraph_free(), 136
EGraph_init(), 136
EGraph_make27P(), 137
EGraph_make9P(), 136
EGraph_mkAdjGraph(), 136
EGraph_new(), 136
EGraph_readFromBinaryFile(), 137
EGraph_readFromFile(), 137
EGraph_readFromFormattedFile(), 137
EGraph_setDefaultFields(), 136
EGraph_writeForHumanEye(), 138
EGraph_writeStats(), 138
EGraph_writeToBinaryFile(), 137
EGraph_writeToFile(), 137
EGraph_writeToFormattedFile(), 137
ETree_backSolveProfile(), 153
ETree_backwardOps(), 145
ETree_balancedMap(), 152
ETree_bndwghts(), 142
ETree_bndwghtsIV(), 142
ETree_clearData(), 141
ETree_compress(), 147
ETree_ddMap(), 152
ETree_ddMapNew(), 152
ETree_expand(), 145
ETree_factorEntriesIV(), 145
ETree_fch(), 142
ETree_forwardOps(), 145
ETree_forwSolveProfile(), 153
ETree_free(), 141
ETree_frontBoundarySize(), 142
ETree_frontSize(), 142
ETree_FSstorageProfile(), 153
ETree_fundChainMap(), 147
ETree_fundSupernodeMap(), 147
ETree_GSstorageProfile(), 152
ETree_init1(), 143
ETree_initFromDenseMatrix(), 143
ETree_initFromFile(), 144
ETree_initFromGraph(), 143
ETree_initFromGraphWithPerms(), 143
ETree_initFromSubtree(), 144
ETree_leftJustify(), 147
ETree_leftJustifyD(), 147
ETree_leftJustifyI(), 147
ETree_maxNindAndNent(), 143
ETree_mergeFrontsAll(), 151
ETree_mergeFrontsAny(), 151
ETree_mergeFrontsOne(), 150
ETree_MFstackProfile(), 152

ETree_MPI_Bcast(), 384
ETree_msByDepth(), 148
ETree_msByNentCutoff(), 149
ETree_msByNopsCutoff(), 149
ETree_msByNvtxCutoff(), 148
ETree_msStats(), 149
ETree_nentMetric(), 146
ETree_new(), 141
ETree_newToOldFrontPerm(), 148
ETree_newToOldVtxPerm(), 148
ETree_nExternalOpsInFront(), 145
ETree_nFactorEntries(), 144
ETree_nFactorEntriesInFront(), 144
ETree_nFactorIndices(), 144
ETree_nFactorOps(), 144
ETree_nfront(), 141
ETree_nInternalOpsInFront(), 145
ETree_nodwghts(), 142
ETree_nodwghtsIV(), 142
ETree_nopsMetric(), 146
ETree_nvtx(), 141
ETree_nvtxMetric(), 146
ETree_oldToNewFrontPerm(), 148
ETree_oldToNewVtxPerm(), 148
ETree_optPart(), 149
ETree_par(), 142
ETree_permuteVertices(), 148
ETree_readFromBinaryFile(), 153
ETree_readFromFile(), 153
ETree_readFromFormattedFile(), 153
ETree_root(), 141
ETree_setDefaultFields(), 141
ETree_sib(), 142
ETree_sizeOf(), 144
ETree_spliceTwoEtrees(), 145
ETree_splitFronts(), 151
ETree_subtreeSubsetMap(), 152
ETree_transform(), 151
ETree_transform2(), 151
ETree_tree(), 141
ETree_vtxToFront(), 142
ETree_vtxToFrontIV(), 142
ETree_wrapMap(), 152
ETree_writeForHumanEye(), 154
ETree_writeStats(), 154
ETree_writeToBinaryFile(), 154
ETree_writeToFile(), 153
ETree_writeToFormattedFile(), 154

fp2DGrid(), 349
fp3DGrid(), 349

FrontMtx_aggregateList(), 263
FrontMtx_assemblePostponedData(), 264
FrontMtx_backwardSetup(), 268
FrontMtx_backwardVisit(), 268
FrontMtx_clearData(), 259
FrontMtx_colmapIV(), 269
FrontMtx_columnIndices(), 260
FrontMtx_diagMtx(), 260
FrontMtx_diagonalVisit(), 267
FrontMtx_factorInpMtx(), 264
FrontMtx_factorPencil(), 264
FrontMtx_factorSetup(), 262
FrontMtx_factorVisit(), 262
FrontMtx_forwardSetup(), 268
FrontMtx_forwardVisit(), 267
FrontMtx_free(), 259
FrontMtx_frontSize(), 260
FrontMtx_frontTree(), 260
FrontMtx_inertia(), 270
FrontMtx_init(), 261
FrontMtx_initFromSubMtx(), 319
FrontMtx_initialFrontDimensions(), 260
FrontMtx_initializeFront(), 262
FrontMtx_loadActiveLeaves(), 263
FrontMtx_loadActiveRoots(), 268
FrontMtx_loadEntries(), 263
FrontMtx_loadRightHandSide(), 267
FrontMtx_lowerAdjFronts(), 261
FrontMtx_lowerBlockIVL(), 261
FrontMtx_lowerMtx(), 261
FrontMtx_makeLowerBlockIVL(), 269
FrontMtx_makeUpperBlockIVL(), 269
FrontMtx_MPI_aggregateList(), 385
FrontMtx_MPI_colmapIV(), 385
FrontMtx_MPI_factorInpMtx(), 380
FrontMtx_MPI_factorPencil(), 380
FrontMtx_MPI_permuteLowerAdj(), 381
FrontMtx_MPI_permuteUpperAdj(), 381
FrontMtx_MPI_postProcess(), 381
FrontMtx_MPI_rowmapIV(), 385
FrontMtx_MPI_solve(), 382
FrontMtx_MPI_split(), 378
FrontMtx_MT_factorInpMtx(), 365
FrontMtx_MT_factorPencil(), 365
FrontMtx_MT_QR_factor(), 366
FrontMtx_MT_QR_solve(), 367
FrontMtx_MT_solve(), 366
FrontMtx_nactiveChild(), 263
FrontMtx_neqns(), 259
FrontMtx_new(), 259
FrontMtx_nfront(), 259

FrontMtx_nLowerBlocks(), 261
FrontMtx_nSolveOps(), 270
FrontMtx_nUpperBlocks(), 261
FrontMtx_ownedColumns(), 269
FrontMtx_ownedRows(), 269
FrontMtx_permuteLowerAdj(), 267
FrontMtx_permuteLowerMatrices(), 267
FrontMtx_permuteUpperAdj(), 267
FrontMtx_permuteUpperMatrices(), 267
FrontMtx_postList(), 263
FrontMtx_postProcess(), 266
FrontMtx_QR_assembleFront(), 265
FrontMtx_QR_factor(), 266
FrontMtx_QR_factorVisit(), 265
FrontMtx_QR_setup(), 265
FrontMtx_QR_solve(), 269
FrontMtx_QR_storeFront(), 266
FrontMtx_QR_storeUpdate(), 266
FrontMtx_readFromBinaryFile(), 270
FrontMtx_readFromFile(), 270
FrontMtx_readFromFormattedFile(), 270
FrontMtx_rowIndices(), 260
FrontMtx_rowmapIV(), 269
FrontMtx_setDefaultFields(), 259
FrontMtx_setFrontSize(), 260
FrontMtx_setupFront(), 262
FrontMtx_solve(), 268
FrontMtx_solveOneColumn(), 302
FrontMtx_splitLowerMatrices(), 267
FrontMtx_splitUpperMatrices(), 267
FrontMtx_storeFront(), 264
FrontMtx_storePostponedData(), 264
FrontMtx_storeSolution(), 268
FrontMtx_update(), 263
FrontMtx_upperAdjFronts(), 261
FrontMtx_upperBlockIVL(), 261
FrontMtx_upperMtx(), 260
FrontMtx_writeForHumanEye(), 271
FrontMtx_writeForMatlab(), 271
FrontMtx_writeStats(), 271
FrontMtx_writeToBinaryFile(), 271
FrontMtx_writeToFile(), 270
FrontMtx_writeToFormattedFile(), 270
FVadd(), 99
FVaxpy(), 99
FVaxpyi(), 99
FVcompress(), 99
FVcopy(), 99
FVdot(), 99
FVfill(), 99
FVfprintf(), 99
FVfree(), 98
FVfscanf(), 99
FVgather(), 99
FVgatherAddZero(), 99
FVgatherZero(), 99
FVinit(), 98
FVinit2(), 98
FVinvPerm(), 99
FVmax(), 99
FVmaxabs(), 100
FVmin(), 100
FVminabs(), 100
FVperm(), 100
FVramp(), 100
FVscale(), 100
FVscatter(), 100
FVscatterAddZero(), 100
FVscatterZero(), 100
FVshuffle(), 100
FVsub(), 100
FVsum(), 100
FVsumabs(), 100
FVswap(), 100
FVzero(), 100

GPart_bndWeightsIV(), 169
GPart_clearData(), 168
GPart_DDviaFishnet(), 169
GPart_DDviaProjection(), 169
GPart_domSegMap(), 170
GPart_free(), 168
GPart_identifyWideSep(), 170
GPart_init(), 168
GPart_makeYCmap(), 170
GPart_new(), 167
GPart_RBviaDDsep(), 172
GPart_setCweights(), 168
GPart_setDefaultFields(), 167
GPart_setMessageInfo(), 168
GPart_sizeOf(), 168
GPart_smoothBisector(), 171
GPart_smoothBy2layers(), 171
GPart_smoothYSep(), 171
GPart_split(), 169
GPart_TwoSetViaBKL(), 170
GPart_validVtxSep(), 168
GPart_vtxIsAdjToOneDomain(), 169
Graph_adjAndEweights(), 181
Graph_adjAndSize(), 181
Graph_clearData(), 179
Graph_componentMap(), 181

Graph_componentStats(), 181
Graph_compress(), 180
Graph_compress2(), 180
Graph_equivMap(), 180
Graph_expand(), 180
Graph_expand2(), 180
Graph_externalDegree(), 181
Graph_fillFromOffsets(), 179
Graph_free(), 179
Graph_init1(), 179
Graph_init2(), 179
Graph_isSymmetric(), 182
Graph_MPI_Bcast(), 384
Graph_new(), 178
Graph_readFromBinaryFile(), 182
Graph_readFromFile(), 182
Graph_readFromFormattedFile(), 182
Graph_setDefaultFields(), 178
Graph_setListsFromOffsets(), 180
Graph_sizeOf(), 181
Graph_subGraph(), 181
Graph_wirebasketStages(), 180
Graph_writeForHumanEye(), 183
Graph_writeStats(), 183
Graph_writeToBinaryFile(), 183
Graph_writeToFile(), 182
Graph_writeToFormattedFile(), 182
Graph_writeToMetisFile(), 183

I2Ohash_clearData(), 53
I2Ohash_free(), 53
I2Ohash_init(), 53
I2Ohash_insert(), 54
I2Ohash_locate(), 54
I2Ohash_measure(), 54
I2Ohash_new(), 53
I2Ohash_remove(), 54
I2Ohash_setDefaultFields(), 53
I2Ohash_writeForHumanEye(), 54
I2OP_fprintf(), 106
I2OP_free(), 106
I2OP_init(), 106
I2OP_initStorage(), 106
Ideq_clear(), 75
Ideq_clearData(), 75
Ideq_free(), 75
Ideq_head(), 75
Ideq_insertAtHead(), 76
Ideq_insertAtTail(), 76
Ideq_new(), 75
Ideq_removeFromHead(), 75
Ideq_removeFromTail(), 76
Ideq_resize(), 75
Ideq_setDefaultFields(), 75
Ideq_tail(), 76
Ideq_writeForHumanEye(), 76
IIheap_clearData(), 57
IIheap_free(), 57
IIheap_init(), 57
IIheap_insert(), 57
IIheap_new(), 56
IIheap_print(), 57
IIheap_remove(), 57
IIheap_root(), 57
IIheap_setDefaultFields(), 57
IIheap_sizeOf(), 57
ILUMtx_clearData(), 274
ILUMtx_factor(), 275
ILUMtx_fillRandom(), 276
ILUMtx_free(), 274
ILUMtx_init(), 275
ILUMtx_new(), 274
ILUMtx_setDefaultFields(), 274
ILUMtx_solveVector(), 275
ILUMtx_writeForMatlab(), 276
InpMtx_adjForATA(), 289
InpMtx_changeCoordType(), 284
InpMtx_changeStorageMode(), 284
InpMtx_checksums(), 291
InpMtx_clearData(), 281
InpMtx_complexVector(), 283
InpMtx_convertToVectors(), 290
InpMtx_coordType(), 281
InpMtx_dropLowerTriangle(), 290
InpMtx_dropOffDiagonalEntries(), 290
InpMtx_dropUpperTriangle(), 290
InpMtx_dvec(), 283
InpMtx_free(), 281
InpMtx_fullAdjacency(), 289
InpMtx_fullAdjacency2(), 289
InpMtx_herm_gmmm(), 288
InpMtx_herm_gmvm(), 289
InpMtx_herm_mmm(), 287
InpMtx_herm_mmmVector(), 288
InpMtx_init(), 284
InpMtx_initFromSubmatrix(), 290
InpMtx_inputChevron(), 285
InpMtx_inputColumn(), 285
InpMtx_inputComplexChevron(), 285
InpMtx_inputComplexColumn(), 285
InpMtx_inputComplexEntry(), 285
InpMtx_inputComplexMatrix(), 286

InpMtx_inputComplexRow(), 285
InpMtx_inputComplexTriples(), 286
InpMtx_inputEntry(), 285
InpMtx_inputMatrix(), 286
InpMtx_inputMode(), 282
InpMtx_inputRealChevron(), 285
InpMtx_inputRealColumn(), 285
InpMtx_inputRealEntry(), 285
InpMtx_inputRealMatrix(), 286
InpMtx_inputRealRow(), 285
InpMtx_inputRealTriples(), 286
InpMtx_inputRow(), 285
InpMtx_inputTriples(), 286
InpMtx_ivec1(), 282
InpMtx_ivec2(), 282
InpMtx_log10profile(), 291
InpMtx_mapEntries(), 286
InpMtx_mapToLowerTriangle(), 290
InpMtx_mapToUpperTriangle(), 290
InpMtx_mapToUpperTriangleH(), 290
InpMtx_maxnent(), 282
InpMtx_maxnvector(), 282
InpMtx_MPI_alltoall, 385
InpMtx_MPI_fullAdjacency(), 385
InpMtx_MPI_split(), 377
InpMtx_MPI_splitFromGlobal(), 378
InpMtx_MT_herm_mmm(), 364
InpMtx_MT_nonsym_mmm(), 364
InpMtx_MT_nonsym_mmm_H(), 365
InpMtx_MT_nonsym_mm_Tm(), 364
InpMtx_MT_sym_mmm(), 364
InpMtx_nent(), 282
InpMtx_new(), 281
InpMtx_nonsym_gmmm(), 288
InpMtx_nonsym_gmmm_H(), 288
InpMtx_nonsym_gmmm_T(), 288
InpMtx_nonsym_gmvm(), 289
InpMtx_nonsym_gmvm_H(), 289
InpMtx_nonsym_gmvm_T(), 289
InpMtx_nonsym_mmm(), 287
InpMtx_nonsym_mmm_H(), 287
InpMtx_nonsym_mmm_T(), 287
InpMtx_nonsym_mmmVector(), 288
InpMtx_nonsym_mmmVector_H(), 288
InpMtx_nonsym_mmmVector_T(), 288
InpMtx_nvector(), 282
InpMtx_offsets(), 283
InpMtx_permute(), 287
InpMtx_randomMatrix(), 291
InpMtx_range(), 283
InpMtx_readFromBinaryFile(), 292
InpMtx_readFromFile(), 292
InpMtx_readFromFormattedFile(), 292
InpMtx_readFromHBFile(), 293
InpMtx_realVector(), 283
InpMtx_resizeMultiple(), 282
InpMtx_setCoordType(), 284
InpMtx_setDefaultFields(), 281
InpMtx_setMaxnent(), 283
InpMtx_setMaxnvector(), 284
InpMtx_setNent(), 283
InpMtx_setNvector(), 284
InpMtx_setResizeMultiple(), 284
InpMtx_sizes(), 283
InpMtx_sortAndCompress(), 290
InpMtx_storageMode(), 282
InpMtx_supportNonsym(), 286
InpMtx_supportNonsymH(), 286
InpMtx_supportNonsymT(), 286
InpMtx_supportSym(), 286
InpMtx_supportSymH(), 286
InpMtx_sym_gmmm(), 288
InpMtx_sym_gmvm(), 289
InpMtx_sym_mmm(), 287
InpMtx_sym_mmmVector(), 288
InpMtx_vecids(), 283
InpMtx_vector(), 283
InpMtx_writeForHumanEye(), 292
InpMtx_writeForMatlab(), 293
InpMtx_writeStats(), 293
InpMtx_writeToBinaryFile(), 292
InpMtx_writeToFile(), 292
InpMtx_writeToFormattedFile(), 292
IP_fp80(), 105
IP_fprintf(), 105
IP_free(), 105
IP_init(), 105
IP_mergeSortUp(), 106
IP_mergeUp(), 105
IP_radixSortDown(), 106
IP_radixSortUp(), 106
IV2DVisortDown(), 102
IV2DVisortUp(), 102
IV2DVqsortDown(), 103
IV2DVqsortUp(), 103
IV2DVsortUpAndCompress(), 105
IV2isortDown(), 102
IV2isortUp(), 102
IV2qsortDown(), 103
IV2qsortUp(), 103
IV2sortUpAndCompress(), 104
IV2ZVisortDown(), 103

IV2ZVisortUp(), 103
IV2ZVqsortDown(), 104
IV2ZVqsortUp(), 104
IV2ZVsortUpAndCompress(), 105
IV_clearData(), 59
IV_copy(), 62
IV_decrement(), 63
IV_entries(), 60
IV_entry(), 60
IV_fill(), 62
IV_filterKeep(), 62
IV_filterPurge(), 62
IV_findValue(), 63
IV_findValueAscending(), 63
IV_findValueDescending(), 63
IV_first(), 62
IV_fp80(), 64
IV_free(), 59
IV_increment(), 62
IV_init(), 60
IV_init1(), 60
IV_init2(), 60
IV_inverseMap(), 63
IV_max(), 61
IV_maxsize(), 60
IV_min(), 61
IV_MPI_allgather(), 381
IV_MPI_Bcast(), 384
IV_new(), 59
IV_next(), 62
IV_owned(), 59
IV_push(), 61
IV_ramp(), 61
IV_readFromBinaryFile(), 64
IV_readFromFile(), 63
IV_readFromFormattedFile(), 64
IV_setDefaultFields(), 59
IV_setEntry(), 60
IV_setMaxsize(), 61
IV_setSize(), 61
IV_shiftBase(), 61
IV_shuffle(), 62
IV_size(), 60
IV_sizeAndEntries(), 60
IV_sizeOf(), 62
IV_sortDown(), 61
IV_sortUp(), 61
IV_targetEntries(), 63
IV_writeForHumanEye(), 64
IV_writeForMatlab(), 64
IV_writeStats(), 64
IV_writeToBinaryFile(), 64
IV_writeToFile(), 64
IV_writeToFormattedFile(), 64
IVcompress(), 97
IVcopy(), 97
IVDVisortDown(), 102
IVDVisortUp(), 102
IVDVqsortDown(), 103
IVDVqsortUp(), 103
IVDVsortUpAndCompress(), 104
IVfill(), 97
IVfp80(), 97
IVfprintf(), 97
IVfree(), 97
IVfscanf(), 97
IVgather(), 97
IVinit(), 96
IVinit2(), 96
IVinverse(), 97
IVinvPerm(), 97
IVisascending(), 102
IVisdescending(), 102
IVisortDown(), 102
IVisortUp(), 102
IVL_absorbIVL(), 71
IVL_clearData(), 67
IVL_equivMap1(), 70
IVL_equivMap2(), 70
IVL_expand(), 71
IVL_firstInList(), 69
IVL_free(), 67
IVL_incr(), 68
IVL_init(), 68
IVL_init2(), 68
IVL_init3(), 68
IVL_initFromSubIVL(), 69
IVL_listAndSize(), 69
IVL_make13P(), 71
IVL_make27P(), 72
IVL_make5P(), 71
IVL_make9P(), 71
IVL_mapEntries(), 71
IVL_max(), 70
IVL_maxListSize(), 70
IVL_maxnlist(), 68
IVL_min(), 70
IVL_MPI_allgather(), 382
IVL_MPI_Bcast(), 384
IVL_new(), 67
IVL_nextInList(), 69
IVL_nlist(), 68

IVL_overwrite(), 71
IVL_readFromBinaryFile(), 72
IVL_readFromFile(), 72
IVL_readFromFormattedFile(), 72
IVL_setDefaultFields(), 67
IVL_setincr(), 68
IVL_setList(), 69
IVL_setMaxnlist(), 69
IVL_setNlist(), 69
IVL_setPointerToList(), 70
IVL_sizeOf(), 70
IVL_sortUp(), 70
IVL_sum(), 70
IVL_tsize(), 68
IVL_type(), 68
IVL_writeForHumanEye(), 72
IVL_writeStats(), 73
IVL_writeToBinaryFile(), 72
IVL_writeToFile(), 72
IVL_writeToFormattedFile(), 72
IVlocateViaBinarySearch(), 97
IVmax(), 97
IVmaxabs(), 97
IVmin(), 98
IVminabs(), 98
IVperm(), 98
IVqsortDown(), 103
IVqsortUp(), 103
IVramp(), 98
IVscatter(), 98
IVshuffle(), 98
IVsortUpAndCompress(), 104
IVsum(), 98
IVsumabs(), 98
IVswap(), 98
IVzero(), 98
IVZVisortDown(), 103
IVZVisortUp(), 103
IVZVqsortDown(), 103
IVZVqsortUp(), 103
IVZVsortUpAndCompress(), 104

localND2D(), 349
localND3D(), 349
Lock_clearData(), 78
Lock_free(), 78
Lock_init(), 78
Lock_lock(), 78
Lock_new(), 78
Lock_setDefaultFields(), 78
Lock_unlock(), 78

makeSendRecvIVLs, 386
MatMul_cleanup(), 384
MatMul_MPI_mmm(), 384
MatMul_MPI_setup(), 383
MatMul_setGlobalIndices(), 383
MatMul_setLocalIndices(), 383
maxTagMPI(), 386
mkNDlinsys(), 351
mkNDlinsysQR(), 352
mkNDperm(), 348
mkNDperm2(), 348
mlbicgstabl(), 304
mlbicgstabr(), 304
MSMD_approxDegree(), 195
MSMD_cleanEdgeList(), 195
MSMD_cleanReachSet(), 194
MSMD_cleanSubtreeList(), 194
MSMD_clearData(), 193
MSMD_eliminateStage(), 194
MSMD_eliminateStep(), 194
MSMD_eliminateVtx(), 194
MSMD_exactDegree2(), 195
MSMD_exactDegree3(), 195
MSMD_fillPerms(), 194
MSMD_findInodes(), 194
MSMD_free(), 193
MSMD_frontETree(), 194
MSMD_init(), 193
MSMD_makeSchurComplement(), 195
MSMD_new(), 192
MSMD_order(), 193
MSMD_setDefaultFields(), 192
MSMD_update(), 195
MSMDinfo_clearData(), 192
MSMDinfo_free(), 192
MSMDinfo_isValid(), 192
MSMDinfo_new(), 191
MSMDinfo_print(), 192
MSMDinfo_setDefaultFields(), 192
MSMDvtx_print(), 195

Network_addArc(), 201
Network_augmentPath(), 201
Network_clearData(), 200
Network_findAugmentingPath(), 201
Network_findMaxFlow(), 201
Network_findMincutFromSink(), 202
Network_findMincutFromSource(), 202
Network_free(), 200
Network_init(), 201
Network_new(), 200

Network_setDefaultFields(), 200
Network_setMessageInfo(), 201
Network_writeForHumanEye(), 202
Network_writeStats(), 202

orderViaBestOfNDandMS(), 350
orderViaMMD(), 350
orderViaMS(), 350
orderViaND(), 350

PatchAndGoInfo_clearData(), 312
PatchAndGoInfo_free(), 312
PatchAndGoInfo_init(), 313
PatchAndGoInfo_new(), 312
PatchAndGoInfo_setDefaultFields(), 312
pcgl(), 304, 305
pcgr(), 304
PCVcopy(), 101
PCVfree(), 101
PCVinit(), 101
PCVsetup(), 101
PDVcopy(), 101
PDVfree(), 101
PDVinit(), 101
PDVsetup(), 101
Pencil_changeCoordType(), 315
Pencil_changeStorageMode(), 315
Pencil_clearData(), 315
Pencil_convertToVectors(), 315
Pencil_free(), 315
Pencil_fullAdjacency(), 316
Pencil_init(), 315
Pencil_mapToLowerTriangle(), 315
Pencil_mapToUpperTriangle(), 316
Pencil_mmm(), 316
Pencil_MPI_fullAdjacency(), 385
Pencil_MPI_split(), 378
Pencil_new(), 315
Pencil_permute(), 316
Pencil_readFromFiles(), 316
Pencil_setDefaultFields(), 315
Pencil_setup(), 316
Pencil_sortAndCompress(), 315
Pencil_writeForHumanEye(), 316
Pencil_writeStats(), 316
Perm_checkPerm(), 80
Perm_clearData(), 80
Perm_compress(), 81
Perm_fillNewToOld(), 80
Perm_fillOldToNew(), 80
Perm_free(), 80
Perm_initWithTypeAndSize(), 80
Perm_new(), 79
Perm_readFromBinaryFile(), 81
Perm_readFromFile(), 81
Perm_readFromFormattedFile(), 81
Perm_releaseNewToOld(), 81
Perm_releaseOldToNew(), 81
Perm_setDefaultFields(), 80
Perm_sizeOf(), 80
Perm_writeForHumanEye(), 82
Perm_writeStats(), 82
Perm_writeToBinaryFile(), 82
Perm_writeToFile(), 81
Perm_writeToFormattedFile(), 82
PFVcopy(), 102
PFVfree(), 102
PFVinit(), 102
PFVsetup(), 102
PIVcopy(), 101
PIVfree(), 101
PIVinit(), 101
PIVsetup(), 101

SemiImplMtx_clearData(), 318
SemiImplMtx_free(), 319
SemiImplMtx_initFromFrontMtx(), 319
SemiImplMtx_new(), 318
SemiImplMtx_setDefaultFields(), 318
SemiImplMtx_solve(), 319
SemiImplMtx_stats(), 320
SemiImplMtx_writeForHumanEye(), 320
SolveMap_backwardSetup(), 206
SolveMap_clearData(), 204
SolveMap_colidsLower(), 205
SolveMap_colidsUpper(), 205
SolveMap_ddMap(), 206
SolveMap_forwardSetup(), 206
SolveMap_free(), 204
SolveMap_init(), 205
SolveMap_lowerAggregateIV(), 207
SolveMap_lowerSolveIVL(), 206
SolveMap_mapLower(), 205
SolveMap_mapUpper(), 205
SolveMap_nblockLower(), 205
SolveMap_nblockUpper(), 204
SolveMap_new(), 204
SolveMap_nfront(), 204
SolveMap_nproc(), 204
SolveMap_owners(), 205, 206
SolveMap_randomMap(), 206
SolveMap_readFromBinaryFile(), 207

SolveMap_readFromFile(), 207
SolveMap_readFromFormattedFile(), 207
SolveMap_rowidsLower(), 205
SolveMap_rowidsUpper(), 205
SolveMap_setDefaultFields(), 204
SolveMap_symmetryflag(), 204
SolveMap_upperAggregateIV(), 207
SolveMap_upperSolveIVL(), 206
SolveMap_writeForHumanEye(), 208
SolveMap_writeStats(), 208
SolveMap_writeToBinaryFile(), 208
SolveMap_writeToFile(), 207
SolveMap_writeToFormattedFile(), 208
SubMtx_blockDiagonalInfo(), 327
SubMtx_clearData(), 325
SubMtx_columnIndices(), 325
SubMtx_complesEntry(), 327
SubMtx_denseInfo(), 326
SubMtx_denseSubcolumnsInfo(), 327
SubMtx_denseSubrowsInfo(), 326
SubMtx_diagonalInfo(), 327
SubMtx_dimensions(), 325
SubMtx_fillColumnDV(), 331
SubMtx_fillColumnZV(), 331
SubMtx_fillRowDV(), 331
SubMtx_fillRowZV(), 331
SubMtx_free(), 325
SubMtx_ids(), 325
SubMtx_init(), 328
SubMtx_initFromBuffer(), 328
SubMtx_initRandom(), 328
SubMtx_initRandomLowerTriangle(), 328
SubMtx_initRandomUpperTriangle(), 328
SubMtx_locationOfComplexEntry(), 328
SubMtx_locationOfRealEntry(), 327
SubMtx_maxabs(), 331
SubMtx_nbytesInUse(), 330
SubMtx_nbytesInWorkspace(), 330
SubMtx_nbytesNeeded(), 330
SubMtx_new(), 325
SubMtx_readFromBinaryFile(), 332
SubMtx_readFromFile(), 331
SubMtx_readFromFormattedFile(), 331
SubMtx_realEntry(), 327
SubMtx_rowIndices(), 325
SubMtx_scale1vec(), 329
SubMtx_scale2vec(), 329
SubMtx_scale3vec(), 329
SubMtx_setDefaultFields(), 325
SubMtx_setFields(), 330
SubMtx_setIds(), 325
SubMtx_setNbytesInWorkspace(), 330
SubMtx_solve(), 329
SubMtx_solveH(), 329
SubMtx_solveT(), 329
SubMtx_solveupd(), 329
SubMtx_solveupdH(), 330
SubMtx_solveupdT(), 330
SubMtx_sortColumnsUp(), 330
SubMtx_sortRowsUp(), 330
SubMtx_sparseColumnsInfo(), 326
SubMtx_sparseRowsInfo(), 326
SubMtx_sparseTriplesInfo(), 326
SubMtx_workspace(), 330
SubMtx_writeForHumanEye(), 332
SubMtx_writeForMatlab(), 332
SubMtx_writeStats(), 332
SubMtx_writeToBinaryFile(), 332
SubMtx_writeToFile(), 332
SubMtx_writeToFormattedFile(), 332
SubMtx_zero(), 331
SubMtxList_addObjectToList(), 339
SubMtxList_clearData(), 338
SubMtxList_free(), 338
SubMtxList_getList(), 339
SubMtxList_init(), 339
SubMtxList_isCountZero(), 339
SubMtxList_isListNonempty(), 339
SubMtxList_new(), 338
SubMtxList_setDefaultFields(), 338
SubMtxList_writeForHumanEye(), 339
SubMtxManager_clearData(), 341
SubMtxManager_free(), 342
SubMtxManager_init(), 342
SubMtxManager_new(), 341
SubMtxManager_newObjectOfSizeNbytes(), 342
SubMtxManager_releaseListOfObjects(), 342
SubMtxManager_releaseObject(), 342
SubMtxManager_setDefaultFields(), 341
SubMtxManager_writeForHumanEye(), 342
SymbFac_initFromGraph(), 343
SymbFac_initFromInpMtx(), 344
Symbfac_initFromPencil(), 344
SymbFac_MPI_initFromInpMtx(), 379
SymbFac_MPI_initFromPencil(), 379

tfqmrl(), 304
tfqmrr(), 304
Tree_clearData(), 210
Tree_compress(), 214
Tree_drawToEPS(), 215
Tree_fch(), 210

Tree_fillBothPerms(), 215
Tree_fillNewToOldPerm(), 215
Tree_fillOldToNewPerm(), 215
Tree_free(), 210
Tree_fundChainMap(), 214
Tree_getSimpleCoords(), 215
Tree_height(), 212
Tree_init1(), 211
Tree_init2(), 211
Tree_init3(), 211
Tree_initFromSubtree(), 211
Tree_leftJustify(), 214
Tree_leftJustifyD(), 214
Tree_leftJustifyI(), 214
Tree_maximizeGainIV(), 213
Tree_maxNchild(), 212
Tree_nchild(), 212
Tree_nchildIV(), 212
Tree_new(), 210
Tree_nleaves(), 212
Tree_nnodes(), 210
Tree_nroots(), 212
Tree_par(), 210
Tree_permute(), 215
Tree_postOTfirst(), 212
Tree_postOTnext(), 212
Tree_preOTfirst(), 212
Tree_preOTnext(), 212
Tree_readFromBinaryFile(), 216
Tree_readFromFile(), 216
Tree_readFromFormattedFile(), 216
Tree_root(), 210
Tree_setDefaultFields(), 210
Tree_setDepthDmetric(), 213
Tree_setDepthImetric(), 213
Tree_setFchSibRoot(), 211
Tree_setHeightDmetric(), 213
Tree_setHeightImetric(), 213
Tree_setRoot(), 211
Tree_setSubtreeDmetric(), 213
Tree_setSubtreeImetric(), 213
Tree_sib(), 210
Tree_sizeOf(), 212
Tree_writeForHumanEye(), 217
Tree_writeStats(), 217
Tree_writeToBinaryFile(), 216
Tree_writeToFile(), 216
Tree_writeToFormattedFile(), 216

Zabs(), 90
zbicgstabl(), 305
zbicgstabr(), 305
zmlbicgstabl(), 305
zmlbicgstabr(), 305
zpcgl(), 306
zpcgr(), 306
Zrecip(), 90
Zrecip2(), 90
ztfqmrl(), 305
ztfqmrr(), 305
ZV_clearData(), 109
ZV_copy(), 112
ZV_entries(), 110
ZV_entry(), 110
ZV_fill(), 112
ZV_free(), 109
ZV_init(), 110
ZV_init1(), 110
ZV_init2(), 111
ZV_log10profile(), 112
ZV_maxabs(), 111
ZV_size(), 110
ZV_minabs(), 111
ZV_new(), 109
ZV_owned(), 109
ZV_pointersToEntry(), 110
ZV_push(), 111
ZV_readFromBinaryFile(), 112
ZV_readFromFile(), 112
ZV_readFromFormattedFile(), 112
ZV_setDefaultFields(), 109
ZV_setEntry(), 110
ZV_setMaxsize(), 111
ZV_setSize(), 111
ZV_shiftBase(), 111
ZV_size(), 109
ZV_sizeAndEntries(), 110
ZV_sizeOf(), 111
ZV_writeForHumanEye(), 113
ZV_writeForMatlab(), 113
ZV_writeStats(), 113
ZV_writeToBinaryFile(), 113
ZV_writeToFile(), 113
ZV_writeToFormattedFile(), 113
ZV_zero(), 112
ZVaxpy(), 90
ZVaxpy11(), 92
ZVaxpy12(), 91
ZVaxpy13(), 91
ZVaxpy21(), 91
ZVaxpy22(), 91
ZVaxpy23(), 91

ZVaxpy31(), 91
ZVaxpy32(), 91
ZVaxpy33(), 90
ZVcopy(), 92
ZVdotC(), 92
ZVdotC11(), 96
ZVdotC12(), 96
ZVdotC13(), 95
ZVdotC21(), 95
ZVdotC22(), 95
ZVdotC23(), 95
ZVdotC31(), 95
ZVdotC32(), 94
ZVdotC33(), 94
ZVdotiC(), 92
ZVdotiU(), 92
ZVdotU(), 92
ZVdotU11(), 94
ZVdotU12(), 94
ZVdotU13(), 94
ZVdotU21(), 93
ZVdotU22(), 93
ZVdotU23(), 93
ZVdotU31(), 93
ZVdotU32(), 92
ZVdotU33(), 92
ZVfprintf(), 90
ZVgather(), 96
ZVinit(), 90
ZVmaxabs(), 96
ZVminabs(), 96
ZVscale(), 96
ZVscatter(), 96
ZVsub(), 96
ZVzero(), 96