LAPACK++ V. 1.1

High Performance Linear Algebra
Class Reference Guide

April 1996

Jack Dongarra
Roldan Pozo
David Walker

National Institute of Standards and Technology
Oak Ridge National Laboratory
University of Tennessee, Knoxville

Contents

Name

VectorDouble —- lightweight vector class

Description

One-dimensional storage class with minimal overhead. It is one step above a C array, only in that utilizes share-semantics (similar to C++ `string` classes) for optimizing memory usage. Not intended for mathematical denotations, but rather used as building block for other LAPACK++ matrix classes.

- only a container class, no mathematical operations defined
- share semantics and reference counting scheme for dynamic garbage collection
- unit stride (elements are in contiguous memory locations)
- fixed 0-based offset
- $A(i)$ declared `inline` for efficiency
- auto conversion to basic pointer type e.g. `double*`

Declaration

#include ⟨lafnames.h⟩
#include VECTOR_DOUBLE_H

VectorDouble (int $n = 0$)

Construct a vector of length n, ($n \geq 0$). A vector of length zero is perfectly legal and usually termed a *null* vector. Vector elements are UNITIALIZED.

VectorDouble (int n, double s)

Construct a vector of length n and initialize all elements to the scalar value s.

VectorDouble (double * x, int n)

Construct a n-length vector as a view (share semantics) of an existing C/C++ array. Further changes to elements of `x` will be reflected in `VectorDouble`.

VectorDouble (const VectorDouble& V)

Create a new n-length vector as a view (share semantics) of an existing `VectorDouble`. Further changes to `V` will be be reflected in `VectorDouble`.

~VectorDouble ( )

Reclaim vector memory space if this the only structure using it.

Assignments

VectorDouble& **operator=** (const VectorDouble& V)

Release left-hand side (reclaiming memory space if possible) and construct a new view of `V`. Further changes to `V` will be reflected in left-hand side. Return reference to new view.

VectorDouble& **operator=** (double s)

Set elements of left-hand size to the scalar value s.

VectorDouble& **inject** (const VectorDouble& V)

Copy elements of `V` into the memory space referenced by the left-hand side, without first releasing it. The effect is that if other vectors share memory with left-hand side, they too will be affected. Note that the length of `V` must be same be the same as that of the left-hand side vector.

VectorDouble& **copy** (VectorDouble& V)

Release left-hand side and copy elements of `V`. Unlike `VectorDouble::inject()` it does not require conformity, and previous references of left-hand side are unaffected.

int **resize** (int n)

Resize to a *new* vector of length n. The element values are UNINITALIZED, even if n is less than the current vector length.

Access Functions

double& **operator** ( )(inti)

Return ith element of vector, with zero-based offset. Optional runtime bounds checking ($0 \leq i \leq n$) set by compile time macro `VECTOR_DOUBLE_BOUNDS_CHECK`.

double& **operator** [ ] (int i)

Identical to `VectorDouble::operator()` above. Included only for compatibility to C/C++ `[]` syntax.

Information Functions

int **size** ( )

Return the length, n, of the vector.

int **ref_count** ( )

Return the number of the data objects to utilize the same (or portions of the same) memory space used by this vector.

int **null** ( )

Shorthand to test if zero-length vector. Identical to `return size()==0;`.

Conversions

operator **double *** ( )

Return address of first element; equilavent to `&V(0)`. Used to mimic use of C arrays, (e.g. as in parameters to functions).

Macros

VECTOR_DOUBLE_BOUNDS_CHECK

Compile time macro, either defined or undefined to perform bounds checking ($0 \leq i \leq n$) in a n-length vector. The default is NOT to perform this check (this is consistent with C/C++); however, it is **highly recommended** to utilize this check – particularly during initial phases of development, testing, and debugging. There is a performance penalty for this, (essentially a boolean test at each element reference), so it can turned off for production runs where performance may be critical.

Debugging

int **debug** (int d)

Set global debug flag to value specified by d. If d is nonzero, then all entry and exit points of methods and operators of `LaGenMatDouble` are traced.

int **debug** ( )

Read the global debug flag.

I/O Functions

friend **ostream& operator** $<<$ (ostream& s, const VectorDouble& V)

Print vector (on one line), with elements separated by white space.

See also

LaGenMatDouble, LaVectorDouble

Name

LaGenMatDouble —- general dense matrix, a two-dimensional array

Description

Basic LAPACK++ matrix:

- uses (deep) copy semantics
- optimizes assignment of temporaries via shallow assignments
- indexing via `A(i,j)` where i, j are either integers or `LaIndex` indices.
- non-unit strides in both directions
- utilizes `VectorDouble` container class
- has basic BLAS++ math functionality
- `cout` $<<$ `A.info()` prints out internal states of A
- optimized to avoid memory copies when returning temporary `LaGenMatDouble` results by value from functions
- auto conversion to basic pointer type e.g. `double*`

Declaration

#include ⟨lafnames.h⟩
#include LA_GEN_MAT_DOUBLE_H

LaGenMatDouble ( )

Construct a null 0×0 matrix.

LaGenMatDouble (int m, int n)

Construct a column-major matrix of size $m \times n$, $(m, n \geq 0)$. Matrix elements are UNITIALIZED.

LaGenMatDouble (int m, int n, double s)

Construct a column-major matrix of size $m \times n$, $(m, n \geq 0)$, and initialize matrix elments to the scalar s.

LaGenMatDouble (double * v, int m, int n)

Construct an $m \times n$ matrix by copying the values from a one-dimensional C/C++ array of length mn.

LaGenMatDouble (const LaGenMatDouble& V)

Create a new matrix from an existing one by copying.

int **resize** (int m, int n)

Resize to a *new* matrix of size $m \times n$. The element values are UNINITALIZED, even if resizing to a smaller matrix.

~LaGenMatDouble ( )

Destroy matrix and reclaim vector memory space if this the only structure using it.

Assignments

LaGenMatDouble& **operator=** (const LaGenMatDouble& V)

Release left-hand side (reclaiming memory space if possible) and construct a new copy of `V`. Return reference to new copy.

LaGenMatDouble& **operator=** (double s)

Set elements of left-hand size to the scalar value s. No new matrix is created, so that if there other matrices that reference this memory space, they will also be affected.

LaGenMatDouble& **inject** (const LaGenMatDouble& V)

Copy elements of `V` into the memory space referenced by the left-hand side, without first releasing it. The effect is that if other matrices share memory with left-hand side, they too will be affected. Note that the size of `V` must be same be the same as that of the left-hand side matrix.

LaGenMatDouble& **copy** (LaGenMatDouble& V)

Release left-hand side and copy elements of `V`. Unlike `LaGenMatDouble::inject()` it does not require conformity, and previous references of left-hand side are unaffected.

LaGenMatDouble& **shallow_assign** ( )

This is an optimization for returning temporary matrices from functions, without copying. Consider the following function

```
LaGenMatDouble foo(LaGenMatDouble &A, LaGenMatDouble &B)
{
    return A+B;
}
```

A temporary matrix is constructed by the compiler to hold the result of `A+B` and the result is returned from the function call by the `X::X(&X)` copy constructor. This may cause unnecessary copying, depending on the degree of optimization employed by the particular compiler implementation. By adjusting this by hand, as in

```
return (A+B).shallow_assign();
```

one can avoid typical data copying. The `shallow_assign()` function essentially sets an internal flag which instructs the `X::X(&X)` copy constructor to avoid the copying.

Access Functions

double& **operator** (int i, int j)

Return (i, j)th element of vector, with zero-based offset. Optional runtime bounds checking $(0 \leq i \leq m)$, $(0 \leq j \leq n)$, set by compile time macro `LA_BOUNDS_CHECK`.

double& **operator** (LaIndex I, LaIndex J)

Return submatrix view specified by indices `I` and `J`. (See `LaIndex` class.) These indices specify start, increment, and ending offsets, similar to triplet notation of Matlab or Fortran 90. For example, if `B` is a 10×10 matrix, `I` is `(0 : 2 : 2)` and `J` is `(3: 1: 4)`, then `B(I,J)` denotes the 2×2 matrix

$$\begin{bmatrix} b_{0,3} & b_{2,3} \\ b_{0,4} & b_{4,4} \end{bmatrix}$$

.

Information Functions

double * **addr** ( )

Returns the memory address of the first element of the matrix. `G.addr()` is equivalent to `&G(0,0)`.

int **size** (int d)

Return the length, n, of the dth dimension, i.e. for an $M \times N$ matrix `size(0)` returns M and `size(1)` returns N.

int **inc** (int d)

Return the distance between memory locations (in terms of # of elements) between consecutive elements along dimension d. For example, if `inc(d)` returns 1, then elements along the dth dimension are contiguous in memory.

int **gdim** (int d)

Return the global dimensions of the (possibly larger) matrix owning this space. This will only differ from `size(d)` if the current matrix is actually a submatrix view of some larger matrix.

int **start** (int d)

If the memory space used by this matrix is viewed as a linear array, `start(d)` returns the starting offset of the first element in dimension d. (See `LaIndex` class.)

int **end** (int d)

If the memory space used by this matrix is viewed as a linear array, `start(d)` return the starting offset of the last element in dimension d. (See `LaIndex` class.)

int **ref_count** ( )

Returns the number of data objects which utilize the same (or portions of the same) memory space used by this matrix.

LaGenMatDouble& **info** ( )

Used in conjunction with an `ostream` to print diagnostics about the matrix. For example, if A is a 10×10 matrix that does not share its space with any other matrix, then `cout << A.info();` could produce

```
(10x10) Indices: (0,9) (0,9) #ref: 1 shallow:0
```

This indicates a 10×10 matrix with 0-based offset, a reference count of 1, and the `shallow` flag turned off.

Conversions

operator **double *** ( )

Returns the address of the first element; equivalent to `&V(0,0)`. Used to mimic use of C arrays, (e.g. as parameters passed to functions).

Macros

LA_BOUNDS_CHECK

Compile time macro, either defined or undefined to perform bounds checking on matrix indexing operations. The default is NOT to perform this check (this is consistent with C/C++); however, it is **highly recommended** to utilize this check – particularly during initial phases of development, testing, and debugging. There is a performance penalty for this, (essentially a boolean test at each element reference) so it can turned off for production runs, where performance may be critical.

I/O Functions

friend **ostream& operator** $<<$ (ostream& s, const LaGenMatDouble& V)

Print vector (on one line), with elements separated by white space.

See also

LaVectorDouble, VectorDouble

Name

LaIndex —- indexing into vectors and matrices

Description

`LaIndex` is an integer triplet denoting the start, increment, and ending indices of a vector view. For example, if

$$B = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$$

then

```
B(LaIndex(1,2,5)) = 9.9
```

sets B to

$$B = \{0.0, 0.1, 9.9, 0.3, 9.9, 0.4, 9.9, 0.6\}$$

and is equivalent to

```
for (i=1; i<=5, i += 2)
     B(i) = 9.9;
```

Declaration

#include ⟨lafnames.h⟩
#include LA_INDEX_H

LaIndex ( )

Construct a null index (start=inc=end=0). This special value of `LaIndex` defaults to the complete vector size. For example,

```
B(Index()) = A;
```

will set all elements of B to those of A. This is equivalent to writing

```
B(LaIndex(0,B.size()-1) = A;
```

or

```
for (int i=0; i<N; i++)
     B(i) = A(i);
```

Name

LaColVectorDouble – a column vector, i.e. an $n \times 1$ matrix.

Description

`LaColVectorDouble` represents a column vector. It is identical to an $n \times 1$ `LaGenMatDouble`, except that it can be indexed by a single integer.

Declaration

#include ⟨lafnames.h⟩
#include LA_COL_VECTOR_DOUBLE_H

LaColVectorDouble ( )

Construct a null 0×1 matrix.

LaColVectorDouble (int n)

Construct a column-major matrix of size $n \times 1$, ($n \geq 0$). Vector elements are UNITIALIZED.

LaColVectorDouble (int n, double s)

Construct a column-major matrix of size $n \times 1$, ($n \geq 0$), and initalize matrix elements to the scalar s.

LaColVectorDouble (double * v, int n)

Construct an $n \times 1$ matrix by copying the values from a one-dimensional C/C++ array of length n.

LaColVectorDouble (const LaColVectorDouble& V)

Create a new column vector from an existing one by copying.

~LaColVectorDouble ( )

Destroy column vector and reclaim memory space if this the only structure using it.

Name

LaRowVectorDouble – a row vector, i.e. an $1 \times n$ matrix.

Description

`LaRowVectorDouble` represents a row vector. It is identical to an $1 \times n$ `LaGenMatDouble`, except that it can be indexed by a single integer.

Declaration

#include ⟨lafnames.h⟩
#include LA_COL_VECTOR_DOUBLE_H

LaRowVectorDouble ( )

Construct a null 1×0 matrix.

LaRowVectorDouble (int n)

Construct a row-major matrix of size $1 \times n$, $(n \geq 0)$. Vector elements are UNITIALIZED.

LaRowVectorDouble (int n, double s)

Construct a row-major matrix of size $1 \times n$, $(n \geq 0)$, and initialize matrix elements to the scalar s.

LaRowVectorDouble (double * v, int n)

Construct an row vector by copying the values from a one-dimensional C/C++ array of length n.

LaRowVectorDouble (const LaRowVectorDouble& V)

Create a new row vector from an existing one by copying.

~LaRowVectorDouble ( )

Destroy row vector and reclaim memory space if this the only structure using it.

Name

LaSymmMatDouble —- square symmetric matrix

Description

- uses (deep) copy semantics
- optimizes assignment of temporaries via shallow assignments
- non-unit strides in both directions
- utilizes `VectorDouble` container class
- has basic BLAS++ math functionality
- `cout << A.info()` prints out internal states of A

Declaration

#include ⟨lafnames.h⟩
#include LA_SYMM_MAT_DOUBLE_H

LaSymmMatDouble ( )

Construct a null 0×0 matrix.

LaSymmMatDouble (int m, int n)

Construct a column-major matrix of size $m \times n$, $(m, n \geq 0)$. Matrix elements are UNITIALIZED.

LaSymmMatDouble (int m, int n, double s)

Construct a column-major matrix of size $m \times n$, $(m, n \geq 0)$, and initialize matrix elments to the scalar s.

LaSymmMatDouble (double * v, int m, int n)

Construct an $m \times n$ $(m = n)$ symmetric matrix by copying the values from a one-dimensional C/C++ array of length mn.

LaSymmMatDouble (const LaSymmMatDouble& V)

Create a new symmetric matrix from an existing one by copying.

int **resize** (int m, int n)

Resize to a *new* matrix of size $m \times n$. The element values are UNINITALIZED, even if resizing to a smaller matrix.

~LaSymmMatDouble ( )

Destroy matrix and reclaim vector memory space if this the only structure using it.

Assignments

LaSymmMatDouble& **operator=** (const LaSymmMatDouble& V)

Release left-hand side (reclaiming memory space if possible) and construct a new copy of `V`. Return reference to new copy.

LaSymmMatDouble& **operator=** (double s)

Set elements of left-hand size to the scalar value s. No new matrix is created, so that if there other matrices that reference this memory space, they will also be affected.

LaSymmMatDouble& **copy** (LaSymmMatDouble& V)

Release left-hand side and copy elements of `V`. Unlike `LaSymmMatDouble::inject()` it does not require conformity, and previous references of left-hand side are unaffected.

Access Functions

double& **operator** (int i, int j)

Return (i, j)th element of vector, with zero-based offset. Optional runtime bounds checking $(0 \leq i \leq m)$, $(0 \leq j \leq n)$, set by compile time macro `LA_BOUNDS_CHECK`.

Information Functions

double * **addr** ( )

Returns the memory address of the first element of the matrix. `G.addr()` is equivalent to `&G(0,0)`.

int **size** (int d)

Return the length, n, of the dth dimension, i.e. for an $M \times N$ matrix `size(0)` returns M and `size(1)` returns N.

int **inc** (int d)

Return the distance between memory locations (in terms of # of elements) between consecutive elements along dimension d. For example, if `inc(d)` returns 1, then elements along the dth dimension are contiguous in memory.

int **gdim** (int d)

Return the global dimensions of the (possibly larger) matrix owning this space. This will only differ from `size(d)` if the current matrix is actually a submatrix view of some larger matrix.

int **ref_count** ( )

Returns the number of data objects which utilize the same (or portions of the same) memory space used by this matrix.

LaSymmMatDouble& **info** ( )

Used in conjunction with an `ostream` to print diagnostics about the matrix. For example, if A is a 10×10 matrix that does not share its space with any other matrix, then `cout << A.info();` could produce

```
(10x10) Indices: (0,9) (0,9) #ref: 1 shallow:0
```

This indicates a 10×10 matrix with 0-based offset, a reference count of 1, and the `shallow` flag turned off.

Conversions

operator **double *** ( )

Returns the address of the first element; equivalent to `&V(0,0)`. Used to mimic use of C arrays, (e.g. as parameters passed to functions).

I/O Functions

friend **ostream& operator** < < (ostream& s, const LaSymmMatDouble& V)

Print vector (on one line), with elements separated by white space.

See also

LaGenMat, LaVector, Vector

Name

LaUpperTriangMatDouble —- upper triangular matrix

Description

- uses (deep) copy semantics
- optimizes assignment of temporaries via shallow assignments
- non-unit strides in both directions
- utilizes `VectorDouble` container class
- has basic BLAS++ math functionality
- `cout` << `A.info()` prints out internal states of A

Declaration

#include ⟨lafnames.h⟩
#include LA_UPPER_TRIANG_MAT_DOUBLE_H

LaUpperTriangMatDouble ( )

Construct a null 0×0 matrix.

LaUpperTriangMatDouble (int m, int n)

Construct an upper triangular matrix of size $m \times n$, $(m, n \geq 0)$. Matrix elements are UNITIALIZED.

LaUpperTriangMatDouble (int m, int n, double s)

Construct an upper triangular matrix of size $m \times n$, $(m, n \geq 0)$, and initialize matrix elments to the scalar s.

LaUpperTriangMatDouble (double * v, int m, int n)

Construct an $m \times n$ $(m = n)$ upper triangular matrix by copying the values from a one-dimensional C/C++ array of length mn.

LaUpperTriangMatDouble (const LaUpperTriangMatDouble& V)

Create a new upper triangular matrix from an existing one by copying.

int **resize** (int m, int n)

Resize to a *new* matrix of size $m \times n$. The element values are UNINITALIZED, even if resizing to a smaller matrix.

~LaUpperTriangMatDouble ( )

Destroy matrix and reclaim vector memory space if this the only structure using it.

Assignments

LaUpperTriangMatDouble& **operator=** (
const LaUpperTriangMatDouble& V)

Release left-hand side (reclaiming memory space if possible) and construct a new copy of `V`. Return reference to new copy.

LaUpperTriangMatDouble& **operator=** (double s)

Set elements of left-hand size to the scalar value s. No new matrix is created, so that if there other matrices that reference this memory space, they will also be affected.

LaUpperTriangMatDouble& **copy** (LaUpperTriangMatDouble& V)

Release left-hand side and copy elements of `V`. Unlike `LaUpperTriangMatDouble::inject()` it does not require conformity, and previous references of left-hand side are unaffected.

Access Functions

double& **operator** (int i, int j)

Return (i, j)th element of vector, with zero-based offset. Optional runtime bounds checking $(0 \leq i \leq m)$, $(0 \leq j \leq n)$, set by compile time macro `LA_BOUNDS_CHECK`.

Information Functions

double * **addr** ( )

Returns the memory address of the first element of the matrix. `G.addr()` is equivalent to `&G(0,0)`.

int **size** (int d)

Return the length, n, of the dth dimension, i.e. for an $M \times N$ matrix `size(0)` returns M and `size(1)` returns N.

int **inc** (int d)

Return the distance between memory locations (in terms of # of elements) between consecutive elements along dimension d. For example, if `inc(d)` returns 1, then elements along the dth dimension are contiguous in memory.

int **gdim** (int d)

Return the global dimensions of the (possibly larger) matrix owning this space. This will only differ from `size(d)` if the current matrix is actually a submatrix view of some larger matrix.

int **ref_count** ( )

Returns the number of data objects which utilize the same (or portions of the same) memory space used by this matrix.

LaUpperTriangMatDouble& **info** ( )

Used in conjunction with an `ostream` to print diagnostics about the matrix. For example, if A is a 10×10 matrix that does not share its space with any other matrix, then `cout << A.info();` could produce

```
(10x10) Indices: (0,9) (0,9) #ref: 1 shallow:0
```

This indicates a 10×10 matrix with 0-based offset, a reference count of 1, and the `shallow` flag turned off.

Conversions

operator **double *** ( )

Returns the address of the first element; equivalent to `&V(0,0)`. Used to mimic use of C arrays, (e.g. as parameters passed to functions).

I/O Functions

friend **ostream& operator** < < (ostream& *s*,
const LaUpperTriangMatDouble& V)

Print vector (on one line), with elements separated by white space.

See also

LaGenMat, LaVector, Vector

Name

lautil.h – various Lapack++ utilities

Description

Contains various support functions for Lapack++. Some of these are internal to the Lapack++/Lapack interface. `LaEnvBlockSize()`, for example, is used to determine the size of temporary workspace required for some Lapack routines.

int **LaEnvBlockSize** (char * *fname*, LaGenMatDouble *A*)

Returns the block size used by Lapack for given subroutine name `fname`. (This string corresponds to the f77 name as documented in the Lapack Users' Guide.)

int **LaEnvBlockSize** (char * *fname*, LaSymmMatDouble *A*)

Returns the block size used by Lapack for given subroutine name `fname`. (This string corresponds to the f77 name as documented in the Lapack Users' Guide.)

double **Mach_eps_double** ( )

Returns machine epsilon for double precision floating point format. (On most machines this should be in the range of 10^−14 or so.)

void **LaSwap** (LaGenMatDouble *&A*, LaVectorLongInt *&ipiv*)

Swaps rows of matrix `A` according to pivot vector `ipiv`.