NAME

view — General-purpose data munching program

SYNOPSIS

view [-nv] [-u *codefile*] [-s *commandfile*] [-l *logfile*] [*datafile*] ...

DESCRIPTION

View is a program for storing, manipulating, and finally graphing numerical data. It provides a set of simple fundamental commands which can be combined in many useful ways. It uses the *mamplot(3)* plotting package, developed by MaryAnn Maher, for data plotting to a variety of devices. Through the *viewmod(3)* module, it can be extended by the user in many ways.

To run *View*, just give the command `view`. The program will create a file called something like `view.log` in the current directory into which it stores a transcript of your session. When you are done, type `quit` or control-`D` to exit.

CONCEPTS

View stores data in variables; each variable has a name and a value, which is either a number, a string, or a curve. A *curve* is a vector of x-y pairs of numbers, for example, a table of numbers and their square roots. Numbers and curves can also possess *units*, as in "5 volts" or ".1A". The units are just a string which is carried along with the value; View does not know how to combine units in any automatic ways. Curves have two units strings, one for x and one for y. If you do not wish to use the *units* feature, you can leave all the units strings blank.

For historical reasons, *variables* are sometimes called *curves,* even if they actually contain numbers or strings. Every variable in *View*'s memory has a unique name. Variable names are case-sensitive.

There are also *computed* variables, whose values are determined by some arithmetic expression involving other variables. This allows a spreadsheet-like network of dependencies among variables. A computed variable may at any time be have a number, string, or curve value depending on the type of its defining expression.

When dealing with curves, the set of x values and the x units are together called the *basis* of the curve. Many variables often share the same basis. For example, you might have variables which are plots of various functions evaluated at the same set of points. These variables would all have the same basis.

Basis vectors are typically sorted into increasing order, with no duplicate values. Bases are not in general required to be sorted, but certain commands only work on curves with sorted bases.

The y-vector of a curve may contain any values; in addition, it may contain "undefined" points. For example, after the command

```
view> a = 1/b
```

any values which were zero in `b` will be "undefined" in `a`. Undefined points are usually written as "`x`" or "`(none)`".

WILDCARDS

Many commands accept wildcard characters in variable names. By default, the wildcards are "`*`" matching zero or more characters, and "`?`" matching exactly one character. For example, the command

```
view> plot foo* ?bar
```

will plot all variables whose names start with `foo`, including `foo` itself, and all variables with four-letter names ending in `bar`.

The `wild` command lets you choose instead to have no wildcards, in case you are manipulating variables with these special characters in their names, or regular-expression style wildcards. Regular expressions are described in the manual entry for *regex(2)*. They are much more flexible, but somewhat harder to use.

Many commands create new variables out of old ones. In these commands, wildcards may be used in both the "source" and "destination" names. For example, the command

```
view> copy foo* = *bar
```

would copy all variables with names of the form *xyz*`bar` into corresponding variables with names of the form `foo`*xyz*.

In any arithmetic expression involving variable names, wildcards are not allowed in the variable names. In fact, only variables whose names contain letters, digits, and underscores, starting with letters, may be used in expressions. In other situations, variable names may contain any characters.

COMMANDS

Commands are typed, one per line, to the "`view>`" prompt. The first word on the line identifies the command, with subsequent words and punctuation as the command needs. Command names are case-insensitive, and may be abbreviated as long as they are unambiguous. For example, you could write `rename` as `ren`, but not as `re` as this would be ambiguous with the `regress` command. Also, many commands have handy Shell-like synonyms; for example, `rm` is another word for `delete`.

If you press the [STOP] key while any command is executing, the command is aborted and you return to the *View* prompt. If you press [STOP] from this prompt, you exit the program.

Some commands differ from this syntax, namely assignments (`=` and `==`), shell escapes (`!`), and help (`?`). See below for details.

The full *View* system has many more commands than are listed here. For a complete listing, type `??`. There is another document, *Programming in View,* which describes the rest of the capabilities of *View*.

`<variable> = <expression>`
This is the basic assignment command for variables. In general, the righthand side is any arithmetic expression involving names of other variables. If any curve-valued variables are mentioned, they must all share the same basis, and the result is a curve of the same basis, and whose y-values are computed elementwise from the values of the other curves. If there are no curves on the righthand side, the result will be a single number or string.

Some examples:

```
view> basis b = 0:3:1
view> foo = b*2
view> bar = sqr(b) + foo
view> four = 2*2
view> spam = bar/four
```

```
view> stuff = four/b
Warning: division by zero
```

The result will be five vector-valued curves and one constant-valued curve. The vector-valued curves all share the same basis, and their values will be:

(basis)	b	foo	bar	spam	stuff
0	0	0	0	0	X
1	1	2	3	.75	4
2	2	4	8	2	2
3	3	6	15	3.75	1.333

where "X" represents an undefined value.

At each point in the vector, if any of the curves' values were undefined, the result will be undefined. For example, computing `stuff+1` would produce `(X,5,3,2.333)`.

You can also refer to basis values either by the notation *curvename*`_x`. For example, `foo+foo_x` = `foo+b` = `(0,3,6,9)`.

If the variable named on the lefthand side already existed, it is overwritten. It is legal to write, for example, `x=x+1` to increment a number variable `x`, or every y-value of a curve variable `x`.

Expressions may use the following operators, borrowed in many cases from the C language:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`\`	integer division
`%`	modulo
`&`	bitwise AND
`|`	bitwise OR
`@`	bitwise XOR
`= <>`	equals, not equal to
`< <=`	less than, less than or equal to
`> >=`	greater than, greater than or equal to
`<< >>`	shift left, shift right
`&&`	logical AND
`||`	logical OR
`!!`	default
`?:`	if-then-else

The comparison and logical operators use 1 for "true" and 0 for "false." Logical AND evaluates its left argument; if the result is zero, the null string, or "undefined" (all of which are considered false), then that value is returned; otherwise, the right argument is evaluated and its value is used. Logical OR works similarly. The "default" operator evaluates and uses its left argument unless that value is "undefined," in which case it evaluates and uses its right argument. The if-then-else operator, written "`a?b:c`", evaluates `a` and, if `a` is true, evaluates and uses `b`, otherwise evaluates and uses `c`.

There are also a number of functions, including:

```
log ln exp abs sqr sqrt pi
sin cos tan arcsin arccos arctan arctan(y,x)
sinh cosh tanh arcsinh arccosh arctanh
```

The `lfuncs` command writes out a complete list of available functions.

Note that `pi` is really a pre-defined constant. `log` is a base-10 logarithm; `ln` is a base-e logarithm. The two-argument `arctan` function computes `arctan(y/x)`, in the range [-pi,pi), whereas the result of regular `arctan` is always in the range [-pi/2,pi/2).

In all of the above functions and arithmetic operations, any illegal or undefined argument (e.g., sqrt(-1)) will cause the corresponding point in the result to be undefined ("X").

<variable> == <expression>
This command creates a computed variable. The result is superficially the same as for `=`, except that the expression itself is saved with the variable. If any of the constituent variables are later changed, this variable will be recomputed according to the new values. For example,

```
view> x = 5
view> y1 = x+2          (now y1 = 7)
view> y2 == x+2         (now y2 = 7)
view> x = 10
```

After the last command, `y2` will change to 12 but `y1` remains the same. This works in exactly the same way for vector-valued curves.

Computed curves are not actually computed until their values are needed. For example,

```
view> a == b
view> b = 3
view> print a
Recomputing curve a
```

The first command is legal even though `b` has not yet been defined. What matters is that it has been defined by the time the curve is used, as in the third command.

If a computed curve is used and its value cannot be calculated, for example, if we had asked to print `a` before we had defined `b`, then the operation involving that curve is aborted. However, a computation resulting in an "X" value prints a warning message but does not abort the operation on the curve.

Another reason a curve might not be computable is because it has a self-reference, for example, if we had done the command `b==a` after the above session.

Computed curves are not recomputed unless one of the constituent curves is again changed.

load <filename> <filename> ...
This command loads curve data from a file or files. This command understands two formats: *View*'s own file format, with a default extension of `.dat`, and AnaLOG dump format (a subset of Sivilotti's infamous data format), with a default extension of `.text`.

View-format files are textual files divided into multi-line "records." Each record has a header line identifying the type of record, and some additional data lines often terminated by a blank line. The possible record types are as follows:

```
<version>
```

A line containing an integer is a version-number record. Version 1 is the only number currently recognized. Every view-format file must begin with a version number record on the very first line of the file; any other version-number records are ignored. This is done to allow the use of Unix `cat` to combine *View* data files.

```
pairs
<curve name>
```

```
<x units>
<y units>
<x> <y>
<x> <y> ...
<blank line or end-of-file>
```

This record defines a vector-valued curve. The x-y pairs must be written one pair per line, separated by blanks. If either the basis or the values are unitless, the corresponding units line should contain the single character "`*`". (A blank line is currently also understood, but this is a bad idea compatibility-wise since a blank line is supposed to terminate the record!)

```
const
<curve name>
<y units>
<value>
<ignored extra lines> ...
<blank line or end-of-file>
```

This record defines a constant-valued curve. Note that extra lines are allowed in the format but never interpreted or written by *View*.

```
expr
<curve name>
<expression>
<ignored extra lines> ...
<blank line or end-of-file>
```

This record defines a computed curve. The extra lines are ignored during reading; *View* uses these lines to write the current value(s) of the curve at the time the curve was saved, purely for your own information. The curve is always recomputed when it is read in.

Any number of blank lines, or comment lines beginning with "`#`", are ignored between (but not within!) records.

The other form, AnaLOG dump files, is not described here.

save <filename> <variable> <variable> ...
append <filename> <variable> <variable> ...
The `save` command writes the specified curve or curves, or all curves in memory, to the specified file. A default `.dat` extension is supplied. The curve names may include wildcards. The following two commands are identical (assuming the default wildcard mode):

```
view> save spam.dat *
view> save spam
```

The `append` command is similar, except that if the specified file already exists, it appends to the file rather than overwriting it. If you append a curve onto a file that already has a definition for the curve, then when the file is read, the most recently appended definition is kept.

source <filename>
Read commands from *filename*`.text`. The commands are interpreted exactly as if they were entered from the keyboard. If any command gets an error, the entire script is aborted. `Source` commands may be nested. If the requested file can not be found in the current directory, *View* looks in `/lib/view`. Generally useful command files should be kept here so that everybody can find them.

When *View* starts up, it always reads commands from `view.cnf`, first in `/lib/view` and then in the current directory, if these files exist. Also, you can give a `-s` command-line option to cause *View* to source a particular file rather than reading commands from the keyboard.

Blank lines, and lines beginning with "#", are ignored. Note that the `view.log` file is formatted so that you can normally read it back in with a `source` command and all but the echoed command lines will be ignored, effectively re-running the previous session.

use <filename>
use <filename> <toolname>
Read the *View* tool in *filename*`.code`. *View* tools are Pascal modules that define additional commands and features for *View*. They are created with the help of *viewmod(3)*; see the documentation for that module for details.

If the file is not found in the current directory, *View* looks in the `/lib/view` directory. A given tool file is never loaded more than once. Each `-u` command line option also specifies a tool to be loaded during startup. Any tools which have been `perm`ed at the time *View* is started are found automatically.

If a `<toolname>` is specified, and a tool by this name has already been initialized, nothing is done. Also, the command checks to make sure such a tool *is* initialized by the loaded file, and prints an error message if not. If no `<toolname>` is present, the file is simply loaded, any tools in it are initialized, and execution proceeds.

list
list <variable> <variable> ...
ls
ls <variable> <variable> ...
List information about the curves in memory. The first form just lists the names of all curves. Curves with different bases are listed on different lines.

The second form briefly describes each curve. If it is computed, the relevent expression is shown. If it is constant, its value is shown. Units are shown. Vector values are *not* shown.

dump <variable> <variable> ...
cat <variable> <variable> ...
Write detailed information about various curves. This command does list vector-valued curves in full.

delete <variable> <variable> ...
rm <variable> <variable> ...
Delete the specified curves.

copy <dest> = <src>
Copy the specified curve(s). Note that `<src>` must be one word, but it may use wildcards to specify a set of curves. The resulting curves will be constant-valued, vector-valued, or computed curves identical to the original curves.

The following commands are similar but not identical:

```
view> foo = bar
view> copy foo = bar
```

The latter command always creates the same kind of curve as `bar`, whereas the former will take a "snapshot" of a computed curve. Also, the assignment command does not allow wildcards.

rename <dest> = <src>
mv <dest> = <src>

Change the name of a curve. If the `<dest>` name specifies a curve that already exists, the command is aborted.

edit <variable> <variable> ...
Save the specified curve(s) in a temporary data file, then edit this file with the text editor, Caged. When the editor returns, do a `load` command on the file. Normally this is done to edit the values of vectors in ways inconvenient in *View* proper. The temporary file is deleted automatically.

setunits <variable> <units>
su <variable> <units>
Change the y-units for the specified curve(s) to the specified value. If no `<units>` string is given, this makes the curve unitless. There is no direct way to change the x-units for vector-valued curves. The `<variable>` name must be one word, but it may contain wildcards.

basis <curve> = <min> : <max>
basis <curve> = <min> : <max> : <incr>
basis <curve> = <min> : <max> : <incr> : <units>
Create a vector-valued curve with the specified name, which is a "basis" curve. That is, the x- and y-values of the curve are the same. A basis curve forms a useful starting point for creating curves from equations. For example,

```
view> basis x = 0:3
view> squares = sqr(x)
```

The curve `x` will contain a set of points about evenly spaced between (0,0) and (3,3). The curve `squares` will be a graph of the `sqr` function, containing points from (0,0) to (3,9).

You may write any expressions for `<min>`, `<max>`, and `<incr>`. However, these expressions may not contain spaces except between parentheses.

If `<incr>` is specified, the basis values are min, min+incr, min+2*incr, etc. For example, `0:1:.3` would generate basis values of 0, .3, .6, .9, and 1. If `<incr>` is not specified, a suitable increment is chosen to generate about twenty points in the basis.

The basis has x- and y-units of `<units>`. If this is omitted, the basis is unitless.

The `basis` command is also handy for manual data entry. You can create a curve with `basis`, then use `edit` to key in your own y-values.

range <new> = <old> [<min> : <max>]
Create a new curve which is a subrange of a vector-valued curve. The set of data points whose x-values are between `<min>` and `<max>`, inclusive, are copied into the curve `<new>`. Note that the new curve will have a different basis unless the entire basis is contained between the two limits.

```
view> basis a = 0:10:1
view> range b = a [ 2 : (2 + 2) ]
```

The curve `b` will contain data points at values 2, 3, and 4 only. Note that all of the spaces in the above `range` command were optional except the one between `range` and `b`.

Wildcards may be used in `<dest>` and `<src>`.

map <dest> = <basis> : <src>

map [<interp>] <dest> = <basis> : <src>
Create a new vector-valued curve `<dest>` with the same basis as `<basis>`, and the same y-values as `<src>`. (The y-values of `<basis>` are ignored.) For x-values which do not appear in `<src>`, some sort of interpolation or extrapolation is performed.

View defines two interpolation functions. The default is `[linear]`, which does linear interpolation between x-values, and creates undefined points for x-values beyond the range present in the original curve. The other is `[lextr]`, which also does linear interpolation, and does linear extrapolation beyond the original range.

```
view> basis x = 1:3
view> squares = sqr(x)
view> basis z = 0:5
view> map s1 = z : squares
view> map [lextr] s2 = z : squares
```

After these commands, curve `s1` contains an approximate graph of `sqr` bewteen x=1 and x=3, and undefined points outside of that range. Curve `s2` contains the same values as `s1` between x=1 and x=3, and is linear in the ranges x¡1 and x¿3, with slopes matching the slope of `squares` at the endpoints.

View tools can define other initialization functions, as described in *viewmod(3)*.

interp
Print a list of all available interpolators, with a brief description. The default list contains only `linear` and `lextr`.

regress <src>
regress <src> [<min> : <max>]
regress <dest> = <src>
regress <dest> = <src> [<min> : <max>]
Fit a straight line to the vector-valued curve `<src>`. In the simplest case, this command simply prints the slope and intercepts of the line. If a `<dest>` curve is specified, the command creates a new curve with the same basis as `<src>`, and whose y-values are a graph of the computed line. This curve is suitable for plotting along with the original curve to illustrate the regression.

If range limits are specified, then only those points whose x-values lie between the limits, inclusively, are used in the computation. This is helpful if there is noise near the ends of a curve.

Wildcards may be used in `<dest>` and `<src>`.

deriv <dest> = <src>
deriv <dest> = <src> : <basis>
Compute the derivative of the vector-valued curve `<src>`. This is done using the simple two-point rule for numerical differentiation. Note that this rule will produce only n-1 points given an n-point curve, so the first point in `<dest>` will always be undefined, as will any point whose nearby values are undefined.

If a `<basis>` curve is specified, then the y-values of `<src>` are differentiated with respect to the y-values of `<basis>`. Otherwise, the curve is differentiated with respect to its own basis values.

Wildcards may be used in `<dest>` and `<src>`.

splice <dest> = <basis> : <src>

Create a new curve by splicing together two other curves. The y-values of <basis> form the x-values of the new curve, and the y-values of <src> form the y-values of that curve. The two input curves do not need to have the same bases, but they must contain the same number of points.

If the y-values of <basis> are not in increasing order, the resulting curve will be sorted. If the new basis contains duplicate values, the resulting curve will have fewer points than the original curves.

Wildcards may be used in <dest> and <src>. All curves will be spliced onto the same basis.

plot <curve> <curve> ... : <basis>
fplot <curve> <curve> ... : <basis>
These are the curve plotting commands. They take a set of vector-valued curves and plot them against their bases. The curves do not need to have the same basis.

If the curve <basis> is specified, the curves are plotted against the y-values of this curve. In this case, all curves must have the same basis. This format is useful for making x-y plots in which the x-values are not necessarily sorted.

If the curves in the <curve> list all have the same y-units, they are plotted against a single y-axis on the lefthand side of the graph. If they have two different y-units strings among them, *View* generates both a lefthand and a righthand y-axis. If they have more than two kinds of units among them, these units are randomly assigned to the two axes.

The fplot command simply plots the curves with little fanfare. The y-axes are labelled by the names of the curves, the x-axis is unlabelled or labelled by the name of the <basis> curve, and there is no title. All axes are linear by default. When you are done viewing the plot, you may want to push the [ALPHA] key on a 9836C to clear it off the screen.

The plot command prompts for a title, axis labels, and log-linear information. It then provides the option of generating a plot on a local pen plotter, or the Apple LaserWriter, or in a PostScript (.ps) or Until (.ff) file.

By default, a line plot of each curve is generated. Undefined points are shown as missing line segments. A valid point between two undefined points is plotted as an asterisk. For log plots, negative values are shown as dashed lines or hash marks. The style allows you to change these defaults.

xlimits <option>
ylimits <option>
y2limits <option>
These commands control the labelling of the bottom, lefthand, and optional righthand axes. If you give the command with no additional arguments, it displays the current settings for a particular axis.

The <option> must be either a log-linear option, a notation option, or a range option.

A log-linear option is either log, for a logarithmic scale on that axis, or lin, for a linear scale. This setting is used by the fplot command, and by plot if you type a blank line to the corresponding prompt.

A notation option is one of sci, for the normal scientific notation, or eng, for engineering notation, or int, for integer notation. For example, a label with value .01 and units of "V" would would be displayed as 1 (10^-2 V) in scientific notation, 10 (mV) in engineering notation, or .01 (V) in integer notation.

A range option must be either none or :, to specify auto-ranging on the axis (this is the default), or <min>:<max>, to force a particular range on the axis. Explicitly setting the range is useful if a series of plots must be generated that may be superimposed. When the range is fixed for an axis, any points that lie beyond that range are clipped and may not be visible.

style <style-code> = <options>
A style-code is a number from 0 to 9 preceded by an "@" sign. Each style-code number has a associated with it a line style, and a point style. By default, the points are plotted in style 0 except for negative points on logarithmic axes, which are plotted in style 1. You can override this in the `plot` or `fplot` command by attaching one or two style-codes to any `<curve>` name. The first style code replaces the default of 0; the second replaces the default of 1.

The `<options>` may contain line-style options, and point-style options. Only one of a particular kind of option may be given.

The line-style option is one of `line`, `dots`, `dashes`, `longdash`, `dashdot`, or `dashdot2`. If no line-style is specified, the points will not be connected by any lines.

The point-style option is one of `stars`, `box`, `circ`, `tri`, `fbox`, `fcirc`, or `ftri`. The "f" forms are filled symbols; the others are outlines. One such symbol is drawn at each (valid) point on the curve. They will be approximately centered, but various output devices may not be able to accomplish this centering very precisely.

If neither a line-style nor a point-style is given, the default is `line`, with no point-style.

```
view> style 1 =
view> style 2 = dashes ftri
view> style 3 = dots tri
view> plot first@2@3 second
```

In this example, `first` will be plotted in dashed lines with filled triangles at the vertices, except for negative log points, which are plotted with dotted lines and open triangles. The curve `second` is plotted with lines in all situations, as are all future curves by default.

wild unix
wild full
wild none
Select a format for wildcard characters in curve names. The default is `unix`, for wildcards similar to filenames in the Shell: the "*" wildcard means zero or more characters, and "?" means exactly one character.

The `full` mode uses the full regular expression syntax recognized by *regex(2)*. The documentation for that module describes the format of regular expressions; the entry for `re_compile` describes searching for curves, and `re_replace` describes wildcards in `<dest>` curve names.

The `none` mode disables wildcarding. All characters are taken verbatim in curve names, though some (such as spaces) may require you to enclose the curve name in quotes. Beware, if you use strange characters in a curve name you will not be able to use that curve in an expression.

print <expr> <expr> ...
This command evaluates and prints the expressions, one after another. The expressions may be quoted strings, in which case the strings are printed, or they may involve arithmetic on numbers and constant-valued curves. Vector-valued curve names are not allowed.

debug
This command turns debugging mode on or off. This is useful mostly to the developers of new *View* tools; it controls the global variable called `v_debugging`.

quit

Exit from *View*. If this command is read by the `source` command, it exits only from the command file in which it appears.

help
?
? <command>
Print help information. With no arguments, this prints a list of available commands, with brief descriptions. With a command name, this prints a detailed description of the command.

!
! <shell-command>
This is the shell-escape command. A "`!`" by itself breaks into a sub-Shell, from which you can give regular shell commands. For example, you might want to do an `ls *.dat` to search for *View* data files. When you give a `quit` or control-D command to the Shell, you return to *View*.

If you write a Shell command line after the "`!`" symbol, that single command is done and then control returns to *View*.

INITIALIZATION

The sequence of actions taken by *View* on initialization is as follows:

First, the *View* internals are initialized. A log file is opened. The default name is `view.log` in the current diectory. (If a file by that name is already open by another machine, *View* tries `view1.log`, etc.) You can give a `-l name` option on the command line to specify the name of a log file, or a `-n` option for no log file.

Second, any tools which were already `perm`ed are initialized. See *viewmod(3)* for details about *View* tools.

Third, if the file `/lib/view/view.cnf` exists, it is read as by a `source` command. You can give a `-v` ("vanilla") option to suppress reading of configuration files.

Fourth, if the file `view.cnf` exists, and you did not use the `-v` option, this command file is read.

Fifth, if any `-u name` options are given, these tools are loaded and initialized as by a `use` command. Tools are loaded in the same order as they appear on the command line.

Sixth, if any data file names are given on the command line, these files are loaded as by a `load` command.

Finally, commands are read either from the keyboard, or from another command file if the `-s name` option is used. When it encounters a `quit` command or end-of-file, *View* sends "exit" signals to all initialized tools, closes the log file, and exits.

FILES

`*.dat`	`Data files`
`view.log`	`Log file`
`view.cnf`	`Startup command file`
`/lib/view/view.cnf`	`System-wide startup file`
`/lib/view/*.text`	`Public command files`
`/lib/view/*.code`	`Public tool files`
`/lib/view/src/*`	`Sources for public tools`

SEE ALSO

viewmod(3)

AUTHOR

Dave Gillespie [DAVE]