Fermat User's Guide

for

Linux and Mac OSX

32 bit and 64 bit versions

Robert H. Lewis

http://home.bway.net/lewis/

July 25, 2011 v3.9.99, 4.19 (32 bit)
Small revisions, 2016, 5.22 (64 bit); 2017, 6.6; 2021; 2023

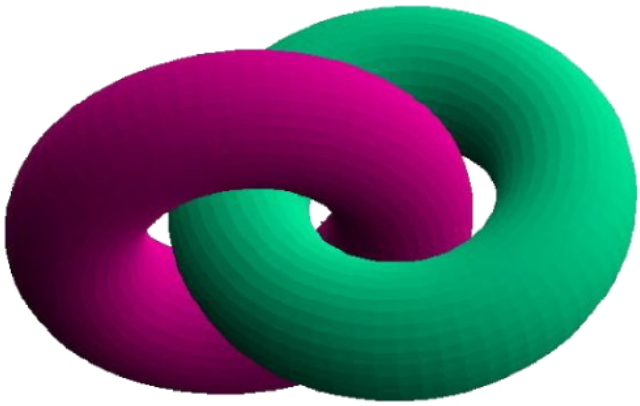


Table of Contents

Titlepage: Two linked toruses. An FFermat generated image. You will find more images scattered here and there in the manual. [Unfortunately the graphics of FFermat no longer work on Mac OS.]

***** As of January 2021 all 32 bit versions are obsolete. Most of**
***** This manual still applies to them, but there are no guarantees.**

Fermat User's Guide

Robert H. Lewis

Fermat is an interactive system for mathematical experimentation. It is a super calculator – computer algebra system, in which items being computed can be rational numbers, real numbers, complex numbers, modular numbers, finite field elements, multivariable polynomials, rational functions ("quolys"), or polynomials modulo other polynomials ("polymods").

There are several versions of Fermat, in different senses of the word "version." Originally, in the late 1980s, Fermat was only for Macintosh and ran under Apple's MPW development system, which provides a command-line interface. Later, stand-alone versions were produced for Macs and Windows95/98/NT/etc. with Metrowerks CodeWarrior. These versions were written in Pascal. Then versions written in C were created for Linux, Unix, and Mac OSX. The other sense of "version" involves the basic "type" one is computing with. All recent versions (since 1999) work over the rationals, in which the ground ring of numerical values is ratios of integers of unlimited size, or over finite fields. There is an old (circa 1999) version that works over the reals, in which case numerical values are the classic "floats" of about 18 significant digits. That version is called FFermat and has graphics capabilities, but is only for Mac OS 9 "classic". For a while the rational version was called "QFermat" when it was necessary to differentiate it from FFermat, but I no longer bother.

Choosing rational or modular "mode" establishes the *ground field* (or *ground ring*) F. On top of this may be attached any number of unevaluated variables $t_1, t_2, \ldots, t_n$, thereby creating the polynomial ring $F[t_1, t_2, \ldots, t_n]$ and its quotient field, the field of rational functions, whose elements are called *quolynomials* or *quolys*. Further, some polynomials $p, q, \ldots$ can be chosen to mod out with, creating the quotient ring $F(t_1, t_2, \ldots) \; / < p, q, \ldots >$, whose elements are called *polymods*. Finally, it is possible to allow *Laurent polynomials*, those with negative as well as positive exponents. Once the computational ring is established in this way, all computations are of elements of this ring.

Fermat has extensive built-in primitives for array and matrix manipulations, such as submatrix, sparse matrix, determinant, normalize, column reduce, Smith form, and matrix inverse. It is consistently faster than some well known computer algebra systems – orders of magnitude faster in some cases.

Fermat provides a complete programming language. Programs are called *functions*, and their use is typical of that in many languages. Programs and data can be saved to an ordinary text file that can be examined as any other file, read during a later session, or read by some other software system.

Fermat has several unique features that enhance debugging. It is possible to interrupt Fermat by moving the cursor to the left edge of the screen (with the mouse) (old obsolete Windows version) or typing control-c (Linux, Unix, OSX). One can then examine the state

of the computation, and later resume it. Error diagnostics are very specific – much more so than in some other systems, which struggle to tell you that an error has occurred somewhere before the last semicolon.

Via arrays, Fermat allows for string processing. The built-in array primitives permit substring and concatenation operations.

The features in Fermat reflect my own interests. It began as a kind of arbitrary precision APL. Surprised that it was much faster in manipulating matrices than the computer algebra system that I had access to at the time (on a VAX), I extended it to help in my research in low dimensional algebraic topology. Polynomial variables were added for the same reason. Research in computational group theory and algebraic number theory suggested several new features and changes in old ones (see Appendix 3, Example 1).

I envision the users of Fermat to be rather sophisticated in both mathematics and programming. (However, you don't have to do *any* programming to use Fermat.) At many places in the design and implementation of Fermat I had to balance the conflicting goals of flexibilty and safety. That is, whether to allow the user certain freedoms or language features that might perhaps be abused, or to circumscribe the user in the name of safety. Since I regard the users as sophisticated, I have usually chosen freedom.

$*$ *This is the Fermat manual for Linux and OSX, 32 or 64 bit* $*$

Basic features - a simple example:

When the user invokes the system he sees the prompt character $>$; the interpreter is waiting for a command of some sort. The user could enter: $8+23$ <*return*>, and the system responds immediately with 31, and the prompt character again. The user could enter any arithmetic expression following the usual syntax, such as $(8-3)*(178-96)/2$ <*return*>.

Fermat always responds with a number after every user command. If there is no obvious number associated with a command, it responds with 0. For example, the creation of an array or function yields a 0.

The result of a computation may be stored for later use. The syntax of "set x equal to 8+23" is

$> x := 8 + 23$ [blanks may be inserted for clarity]

31

Fermat ignores blanks inside long integer constants, as 222 333 444 = 222333444.

The name x may or may not have been used before. The user can now compute expressions in x, like

$> y := (x-29)*x-60$

2

If the formula is going to be used frequently, the user will want to give it a name, say F:

$> Function\ \ F(t) = (t-29)*t-60.$

0 [0 is the computed result of a function definition]

The t is called a *formal parameter* or just *parameter*. Any number of parameters is allowed.

The user can now enter commands like

$> y := F(x)$

2

Notice that F(31) has been computed, stored in y, and the answer displayed.

$> F(27 + y + 1)$

-30

Here $F(30)$ has been computed and displayed, but not stored anywhere by the user. The latest computed result arising from a terminal command is stored in the *system variable*, where it can be accessed via the symbol <*control-q*> (hold control and press q), which looks like a left facing arrow.

In rational mode, adding 1/3 + 1/6 produces 1/2. Note well: rational numbers are computed and there is no roundoff of any sort. Values not whole integers are also displayed as decimals, to a precision (called the display constant) selected by the user. For example, with display constant 3, 1/3 will be displayed as

1/3 or 0.333333333333333333333333

(3 blocks of 8 digits). Setting the display constant to 0 turns off this feature.

Computed values and function definitions can be saved from one session to the next. They are stored in an ordinary ASCII file that can be edited with an ordinary word processor. This file is called the *save file* or, when being read, the *input file*.

Fermat uses the ordinary character set of most languages. These symbols will be described in this manual as they are encountered. They are summarized in Appendix 2.

Fermat is command-line oriented. Type a line, then press the *return* key at the end. You may enter long commands that occupy more than one line by putting the *continuation character* ‘ (upper left key on keyboard) on the end of each line (except the last, of course). This continuation character essentially means “ignore the invisible end-of-line character.” There are two occasions when it is not needed, inside multi-line comments and inside multi-line character strings (also called *literals*). See the later chapters on “Comments” and “Character Strings.”

Let’s continue the example of a Fermat session. Suppose the user has a file of previously saved data called “stuff”. He uses the imperative read command

>&(R = stuff) or >&(R = ‘stuff.txt’)

to open the file for reading and bring the data into this session. Suppose there is a 3×3 matrix $[x]$. To see it, he can use the *short form of array display*,

>![x

which results in

```
> [x] := [[  1,      3,      12,
           2,      7,      -5,
           0,      24,     -4   ]]
```

To compute the determinant the command is

$> Det[x]$ [the result is:]

692

The user decides to compute the characteristic polynomial of $[x]$. First he adjoins a polynomial variable t:

>&J
Change of polynomial variable. [Fermat prompts for the name.]
Enter variable name:

$> t$

The user subtracts the variable t from every element in the main diagonal. This is done with:

$> [y] := [x] - [t]$ [$[t]$ means a diagonal matrix, 3×3 since $[x]$ is.]

Just to look at it, he displays $[y]$:

>![y
> [y] := [[-t + 1 , 3, 12,
2, -t + 7 , -5,
0, 24, -t - 4]]

Then invoking determinant computes the desired polynomial:

$> w := Det[y]$

$-t^3 + 4t^2 - 89t + 692$

(There is a built-in function *Chpoly* to more easily compute characteristic polynomials.)

To check the Cayley-Hamilton Theorem, the user decides to evaluate this polynomial at the matrix $[x]$, using the built-in command # for polynomial evaluation:

$> w\#[x]$

[[0, 0, 0,
0, 0, 0,
0, 0, 0]]

The user decides to change the ground ring from the present $\boldsymbol{Z}[t]$ to $\boldsymbol{Z}[t]/ < t^2 + 1 >$. He gives the imperative form of the mod-out-by-polynomial command

$> \&(P = t^2 + 1, 1)$

The extra "1" tells Fermat that $t^2 + 1$ is irreducible, so an integral domain results. Then the command

$> 1/(t + 1)$ yields:

$(-t + 1)/2$

Everything will be saved to the file "stuff2", via the save command(s):

> &(S = ‘stuff2’)

> &s

The user now exits:

> &q

bye

NOTE: the command to quit is &q.

***** As of January 2021 all 32 bit versions are obsolete. Most of *** This manual still applies to them, but there are no guarantees.**

***** The July 2023 revision of Fermat (version 7.0) was significant, mostly involving factoring of polynomials. Look for Pre 2023 and Post 2023 to see the corresponding revisions in this manual.**

Interpreter commands

Interpreter commands affect system-wide *globals*, perform I/O, or display variables or functions. Most begin with the symbol & and have the syntax &<*symbol*>. There are also the cancel commands @<*name*>, and the terminal I/O commands ! and ? . (Note: The brackets < and > are not entered by the user – they designate that <*symbol*> and <*name*> are elements of syntax of the language. A <*symbol*> is a key-stroke. A <*name*> is a legal Fermat name, a sequence of up to ten digits or letters of the alphabet, beginning with a letter.) Many of the & commands can be used as *toggle switches* to flip a global back and forth between two possibilities, or as *imperative commands* to force the global into a certain position. There is a later paragraph on the imperative form of switches.

A Word to the Wise

Don't leave a toggle switch on unless you wish to use it. Don't adjoin a polynomial variable unless you are going to use it. Every capability costs a little in performance.

Cancel

@ : delete a variable, array, or function; erase it; cancel.

To delete a list of items, say variables x, y, array $[x]$, and functions F and G, enter $@(x, y, [x], F, G)$. The list may be in any order.

Occasionally, after many computations, one develops a large number of now useless arrays, often many of the same name. (This can't happen with ordinary variables.) To delete all arrays named $[x]$, enter $@[*x]$. To delete *all arrays* (a powerful command!) enter $@[**]$.

Similarly, during a session of active experimentation and creation of functions, one often develops many versions of the same function. To erase all but the latest version, the delete command has a *purge* option, indicated syntactically by angle brackets, < and > (less than and greater than). To purge the function G, enter $@<G>$. Mnemonically, you are erasing a "vector full" of G's. This can also be done with arrays, and can be done as part of a list of deletions.

@ can also be used to break out of certain modes, if you change your mind. See under &r, &s, and ?.

The &<*symbol*> commands

&a: Toggle switch to change array initial indexing value. By default it is 1, which means the first entry in array $[x]$ is $x[1]$, uniformly for all arrays. For two-dimensional arrays, the first element is $x[1, 1]$. Entering this command changes the constant to 0, so the first element of array $[x]$ is $x[0]$ or $x[0, 0]$. Note that the *creation* of arrays is unaffected – indeed the arrays themselves are unaffected, it is only the *accessing* of arrays that changes. See the later chapter on "Variables and Arrays" for more information.

&b: Toggle switch to control closing of input file when a syntax error occurs while reading. When on and an error occurs, the file is closed. This makes it easier to use a word processor to edit the file. See &r.

&d: Change the display constant. In rational mode, the display constant is the number of blocks of 8 significant digits to the right of the decimal point displayed by the interpreter

when a non-integer is displayed. If the display constant is 0, this feature is disabled. This can also be affected by the options after a display "!" command (see below). If you enter &d, you will be prompted to enter the new display constant. Alternatively, use the imperative form described below.

&D: Set the determinant cutoff. Two basic ways to compute the determinant are expansion by minors and Gaussian elimination. The first is an $O(n!)$ method and the second is an $O(n^3)$ method, so is "of course" superior. But $5! = 120$ and $5^3 = 125$. Furthermore, it is far easier to multiply polynomials than to divide them. Fermat runs a hybrid of these two. The determinant cutoff controls the switchoff point. Setting $\&D = -1$ lets Fermat decide where to switch off. Setting $\&D = n$ means begin with Gaussian elimination and do the last n rows by minors. $\&D = 0$ means Gaussian elimination all the way. Type &D, and you will be prompted for the value. The imperative form $\&(D = ...)$ can be placed inside a function. The default cutoff is not always the best if there are lots of complex quolynomials in the matrix. This does not work the same way with sparse arrays. See "determinant cutoff, sparse". There are other ways to compute determinant; see appendix four.

&e: Toggle switch to change error handling in higher command levels. When turned on, Fermat will attempt to recover from certain errors in certain situations. See the later chapter "Popping, Pushing, Debugging, and Panic Stops". Entering &e again turns this feature off.

&E: Toggle switch to eliminate the extra blank line that Fermat puts on the screen between lines of input and output.

&f, &F: List all current functions. Only the names and parameter lists of each will be displayed. To list the complete body of the functions as well, enter &F∗ or &f∗. To list the body of only the function named G, type &F,G , &f,G , &F<G , or &f<G. These all have the same effect. The choices exist so that the user doesn't have to release the shift key when entering the command. (< and comma are on the same key.)

To facilitate the editing of functions, Fermat will display the function with the *continuation character* ‘ (upper left key on keyboard) at the end of each line. You can easily change part of the function on the screen, cancel the function with @, and then enter the amended version from the screen. Alternatively, you don't have to cancel the old version – the new version is stacked on top of the old. [You can also *purge* the old version(s), keeping only the most recent, as explained in the chapter "Functions."] **Note well:** this character ‘ should not appear in an input file and is not part of the function definition.

&g: Report the status of the system-wide "globals". If you can't remember the ground ring, what file you are saving to, what the array initial index is, etc., enter this command. Several globals, such as error-push (&e) and random divisor (&_r), are not mentioned unless they have been changed from the default.

&h: On Linux, Unix, and OSX, reports the heap size – the number of bytes of memory that have not yet been allocated.

&m: Change the command interrupt level. This is now obsolete; just hit cntl-c. See below under "Popping, Pushing, Debugging, and Panic Stops".

&M: Change the prompt. If you don't like > for a prompt, change it to something else.

&M will ask you for the new prompt. If you wish no visible prompt, change it to either the empty string, by hitting <*enter*> immediately, or change it to the end-of-line character (ASCII 13) by typing one blank and then hitting <*enter*>. These two will seem equivalent until the interpreter enters a higher command mode (see the chapter on "Popping, Pushing, Debugging, and Panic Stops".)

&n: Toggle switch for "modpolyreadin." See polynomial read-in below.

&N: Toggle switch to cut off "noise," suppress the displaying of *all* results (until &N is entered again). Displaying of *individual* results can be suppressed by typing colon before <*enter*>. For example, $x := 1000!$: will correctly assign x, but nothing will be displayed on the terminal screen.

&o: Used to save specified variables to the output file. For example, !(&o, x, y) writes the value of x and y to the previously defined output file. See &s, below.

&p: Change into modular ground ring. See the later chapter "Arithmetic Modes".

&P: Polynomial mod-out. You will be prompted to enter the polynomial to mod out by. For example, you could enter $t^2 + 1$ to simulate the field of complex numbers. &P can also be used to stop modding out, by entering $-t$ after the prompt. See the chapter on "Polymods".

&q, &Q: Quit.

&r, &R: read from input file, loading previously saved functions and variables. An input file may be created using an editor (save as ASCII or raw text) or the save command (see &s below). Each line of text will be treated as if the user had entered it during a Fermat session (except that end-of-line will be ignored). There must be a semicolon after each complete command, except after comments or multiline literals in functions. **Fermat will not read beyond the first semicolon it sees on the line.** Reading ceases when the &x command is met. The file should end with the &x command (see &x below).

The input file must not itself contain a read command.

The *first* time you enter &r, if you do so from the keyboard, you will be prompted for the name of the file to be read from. If the first read command is made from a function, use the *imperative form*: &(R = <*filename*>) . Later calls to &r will take up where the last one stopped – i.e., after the &x. (More information below under "Imperative form of switches".)

It is possible to read from another, different file by entering &R. You will be prompted for the name of the file. The old file will be closed and data will be read from the new file. If you wish to put this command in a function, use the *imperative form* &(R = <*filename*>). For the rules of *filename* formation, see the later chapter on "Names".

To close the file without opening a new one, use the command &(R = @). See &b.

If you entered &R by mistake, just enter "@" as the name of the file, and the read will be aborted.

It is not an error to try to read from an empty file; nothing happens. In practice this occurs when the user mistypes the name of the file. In some systems, the user discovers the problem only much later after a lot of frustration. To prevent this annoying situation, Fermat issues a warning message when it tries to read from an empty file.

&s, &S: Specify and save to an output file. The current variables and functions will be written to (the end of) an ordinary text file, which can be edited and later read during another Fermat session. The previous contents of the file are not erased.

&s and &S are not strict analogs or duals of &r and &R. It is possible to do a "dumb save" that has no analog in reading.

The first time you enter &s, if you do so from the keyboard, you will be prompted for the name of the file to be saved to. *Nothing will be written to the file you name. You have so far only specified its name.* Upon the second &s command, all of the current functions and variables will be written to the file in human readable (and Fermat readable) form. Each succeeding &s likewise saves all the current functions and variables.

If you have been saving data to one file and wish to change to another, enter &S. *Nothing will be written to the file you name. You have so far only specified its name.* You will be prompted for the name of the new file. The old file will be closed (which means that if Fermat crashes, the saved material *should* survive. More on this in the chapter "Hints and Observations."). Subsequent commands &s will save data to the new file. If you wish to put the &S command in a function, use the *imperative form* &(S = <*filename*>). For the rules of *filename* formation, see the later chapter on "Names."

Each call of &s adds more stuff to the previous contents of the file.

To close the file without opening a new one, use the command &(S = @).

Inserted in the saved file will be the commands &p, &a, &J, etc., reflecting the arithmetic mode (ground ring), initial array index, polynomial variables, etc. In this way, if you read the saved file in your next Fermat session, all these globals will be restored to their status at the moment you saved the file. (More information on this below under "Imperative form of switches".)

It is possible to do a "dumb save" in which raw data only is appended to the save file. Use the display command ! along with &o, as in !(&o, $x, y, x + y$), which writes the current values of x, y, and $x + y$ to the file that has already been specified as the save file.

As a time- and space-saving aid, one can add the ^ when saving, as in !!(&o, 'q := ', ^q, ';'). Without the ^, q is duplicated in the course of expression evaluation. That might be a big waste of time or space.

If you entered &S or &s by mistake, just enter "@" as the name of the file, and the command will be aborted.

Note: input and output cannot be to the same file.

&t: Toggle switch to turn automatic timing on/off. When on, the length of time in seconds that each command or calculation takes will be displayed right after the result of that command or calculation is displayed. The measured time does not include the time it takes to display the result – which, for very large numbers, can be substantial. The accuracy depends on the OS, 1/60th - 1/1000 of a second. Before 2009, in OSX the cpu time was displayed, not the actual real time.

The display of elapsed time changed in 2010. When timing is enabled, two numbers are displayed, called "Elapsed CPU time" and "Elapsed real time". CPU time is just the CPU time used by Fermat. This is what has been displayed by Fermat in most previous versions. However, the number is meaningful only up to about an hour. For much longer times, the value shown is meaningless.

Elapsed real time is wall clock time, just as it sounds. If the elapsed real time is more than 5 seconds, then it is also displayed.

&T: Report the time since Fermat was invoked. This command can be used to measure how long part of a function takes to run, by setting a variable equal to &T, then performing the computation, then computing the difference between &T (a second call) and the variable.

&U: Toggle switch to enable ugly display. When on, Fermat will display long integers and polynomials in the style of other computer algebra systems (Maple). This facilitates communication between Fermat and the others.

&v: List all current variables. Their values as well as their names will be displayed unless &N (see above) has been set. Alternatively, follow with a colon to suppress the values. Only about the first 150 lines of a large polynomial are shown, unless &_s has been set.

&V: Turn on "verbose display." Will display the progress of the more involved and time-consuming procedures, such as Chpoly, Smith, or matrix inverse. A good thing to turn on. Post 2023: There is a function Verbose to report if it is on.

&W: Turn on a second level of "verbose display."

&z: Toggle switch to turn on/off the Zippel-like GCD algorithm. On by default. Should be on if there are more than, say, 7 poly vars. See Appendix 6.

&_G: sort the heap garbage. This can be added to the user's functions periodically during memory intensive polynomial calculations. A noticeable speedup occurs when used between repetitions of an intensive calculation.

&x: Stop reading from the input file. This command provides a break, allowing the input file to contain data in blocks, whose reading can be interrupted by computation. The next read command causes reading to begin right after the &x. There should be an &x at the end of the file.

&_d: Change the width of the display on the window (Linux, Unix, OSX).

&_r: Change the default divisor for the random number generator. More on this below under built-in functions, *Rand* = random number.

&_m: Toggle switch to turn modular arithmetic off/on in modular mode. Do not use it in the situations when modular mode is automatically turned off (and then on again): during the computation of exponents (i.e., following ^), in array coordinates, in character strings, after *Sigma* or *Prod*, and in format specifications, such as $!!x:8$. More on this later under "Arithmetic Modes". The variant *local form* &_m(<*expression*>) turns modular mode off, computes the expression, then restores modular mode.

Because typing &_m(... is sometimes awkward, the symbol _ or _(...)may also be used to suppress modular.

&_l: list-of-monomials display. See Appendix Four.

&_P: Push new command level. Begin new computation. More on this later in the chapter "Popping, Pushing, Debugging, and Panic Stops".

&_p: Pop command level – return to lower level. Return to computation on that level.

&_s: Suppress/don't suppress display of long polynomials.

&_t: Toggle switch to turn on/off a certain fast probabalistic algorithm to test if one multivariate polynomial divides another over ground ring Z. Rarely, this technique can fail, in which case you will see a "Fermat error" about "number in trial_poly_divide". Then turn it off.

&@: Panic stop – return to lowest command level after an interrupt.

&J: Adjoin a new polynomial variable: allow manipulation of polynomials in an unevaluated variable. The user is prompted to enter the variable name. If entered within a function, use the imperative form &(J = t) to directly supply the name. There can be up to 121 such variables. Names cannot be repeated.

You can drop the polynomial variable t by entering $-t$ after the prompt.

&l: toggle switch to change Laurent. When on, Laurent polynomials are allowed – those with negative exponents. When this switch is changed, all the current variables are scanned and their format changed if appropriate. For example, $1/t$ will become t^{-1} if Laurent is now true. See the later chapter on Laurent polynomials.

&|: toggle switch to affect the value returned by the built-in function *mod*. When on (the default) *mod* returns non-negative values, so that $(-2)|3 = 1$. When off, $n|m$ simply returns the remainder produced by dividing n by m, so $(-2)|3 = -2$. The second choice executes very slightly faster; it takes an extra step to ensure that a non-negative is returned.

&B: toggle switch to change the block on the Chpoly polynomials. This is rather technical; see the later chapter on "The Dangerous Commands."

Imperative Form of Switches

The switches &l, &a, &e, &p, &n, &N, &|, &t and a few others can be used in an *imperative* sense rather than the *toggle* sense described above. &_r, &m, &M, &d, &D, &P, &R, and &S can also be used imperatively, i.e., in a non-interactive way. To turn timing on, whatever the current status, enter &(t=1). Similarly, to turn it off, enter &(t=0). 1 means on, 0 means off. To forcefully enter modular mode with modulus n, use &(p = n). To get to rational mode in this way, use &(p=_i). Similarly for the others.

&(S = <*filename*>) is equivalent to entering &S, and then responding with <*filename*> when prompted. This form does it all at once without the prompt. It can therefore be placed in a function. Similarly for &(R = <*filename*>). For the rules of *filename* formation, see the later chapter on "Names." You may also use a character string stored in an array to name the file, as in &(S = [x]). This allows file names to be computed within functions.

&_r = <*expression*>) sets the random divisor.

When you save to a file using &s or &S, the appropriate commands will be inserted to record the status of polynomials, modular mode, etc. It is this imperative form of these commands that is placed in the saved file.

Terminal Input/Output, List Output, and Dump Save

!(variable, array): write to terminal; display on the screen. If $F(x) = x * x + !x$, F returns x^2 and displays x on the screen. This could also be written $F(x) = x * x;\ !x$. The first version will return the value x^2 (since $!x$ returns value 0), the second will return 0

(the last value computed – displaying returns value 0). Another possibility is $F(x) = !x;$ $Return(x * x)$.

You can display written messages, such as:

$$!'value\ of\ x\ :'; !x; !; !'value\ of\ y'; !y; \ldots$$

You can have any character except the quote $'$ itself in a quoted message. Quoted messages can extend over line boundaries.

$!x$ does not move up to a new line, so the next use of this command will continue displaying on the same line. Using ! by itself, !; will start a new line of output. Alternatively, to display and move up all at once, use the double exclamation: $!!x$.

On some systems $!x$ will not necessarily display x *immediately*; rather, the data will be displayed when the buffer is full. To force the immediate display, use $!!x$.

List Output: You can use the syntax

$$!(< expression >, < expression >, \ldots, < expression >)$$

to display a list of expressions all on one line. Here <*expression*> means an algebraic expression or a quoted string. A certain default amount of spacing is inserted between the items. For example, to show variables x, y, and z with a message between, you could use the syntax $!(x, y,'here\ is\ z:\ ', z)$. The user can add spaces after any item in such a list with the syntax $!(x, y : 8,'here\ is\ z:\ ', z : 8)$. y and z will occupy a total of eight spaces; however if y or z itself requires more than 8, it will be printed in its entirety. Any expression may be substituted for the 8 (it is truncated to a real integer if necessary). The spacing conventions differ slightly in modular arithmetic mode (which is explained in "Arithmetic Modes" below.)

In any mode, there is a certain minimum amount of spacing that can't be overridden.

Note that the spacing feature applies only to list output – $!x : 8$ is illegal.

$!!(x, y, \ldots)$ will display a list and then move up to a new line.

As a time- and space-saving aid, one can add the ^ when saving, as in !!(&o, 'q := ', ^q, ';'). Without the ^, q is duplicated in the course of expression evaluation. That might be a big waste of time or space.

Dumb Save: You can append raw data (as opposed to the "prepared" data the &s command creates) to a save file by using &o as the first argument in a list output command, such as $!!(\&o, x, y, x + y)$. &o means the previously specified save file.

Note that $!!(\&o, x, y, \ldots)$ is fine but $!!(\&T, x, y, \ldots)$ is an error. If the first argument in a list display starts with & then it must be &o.

Displaying Arrays: Two formats are available for both ordinary and *Sparse* arrays, "short form" and "long form." The short form is the same for both, but the long form is not. [See the chapter "Variables and Arrays" below for more about *Sparse* arrays.]

First, for ordinary arrays: Since numbers in Fermat are often many digits long or complicated fractions, the long format display $![a]$ conveniently shows one value of the array per line, **in column-major order.** If you expect a matrix to be composed of small integers, however, the short form $![a$ is nice: it displays the matrix in the familiar square pattern.

Analogously with list output, the command ![x : 8 will add spaces after every displayed number. However, ![x] : 8 is illegal – indeed, it makes no sense.

The long form displays the name and coordinates as well as the value of each entry. For example, if you have $x[2,2]$ with $x[i,j] = i + j$, ![x] produces:

$$> x[1,1] := 2$$
$$> x[2,1] := 3$$
$$> x[1,2] := 3$$
$$> x[2,2] := 4$$

The short form: Fermat displays the name of the matrix, as in:

```
>[x] := [[  1,        3,        12, ‘
            0,       24,        -5   ]]
```

You may change any of the components, change the name if you wish, and enter the entire matrix (by highlighting it with the mouse), such as:

```
>[y] := [[  2,        3,        12, ‘
            0,       24,        -5   ]]
```

where $x \to y$ and $1 \to 2$.

Note: In the interests of saving both time and space, Fermat will not display all of a huge number (integer constant) within a matrix when writing to the console terminal. It only displays the first couple hundred digits. This is controlled by &_s.

Sparse arrays:

The long form displays the array as a list of rows, only showing those entries that are not zero. For example, if [x] is *Sparse*, ![x] may produce:

```
>[x] := [ [ 1,    [2, -1],    [3, 19 ]] ‘
          [ 2,    [ 1, 8],    [2, -2 ]] ‘
          [ 3,    [ 2, 1],    [3, 11 ]]  ]
```

This [x] has in row 1 a -1 in column 2 and a 19 in column 3, in row 2 an 8 in column 1 and a -2 in column 2, etc. The dimensions of [x] are not apparent from the long form; it may have more than 3 rows, so long as all entries are zero there.

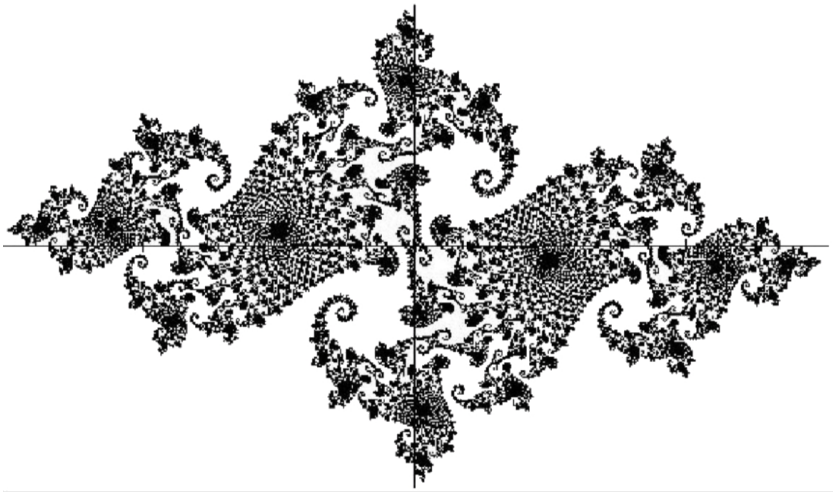
The short form is the same as for ordinary matrices. If the above [x] is 3×3, ![x produces:

```
>[x] := [[  0,    -1,    19, ‘
            8,    -2,     0, ‘
            0,     1,    11   ]]
```

?(variable, array): get input from the terminal; interrogate user. If $F(x) =?y; x - y$. F asks the user to enter a value for y, then computes $x - y$. Fermat halts when it encounters $?y$ and prompts the user. Nothing will happen until the user enters a value for y, which may be any legal expression.

When used with an array, such as ?[x] , Fermat displays the name and coordinates of each entry in the matrix **in row-major order** and waits for the user to enter a value for each successive coordinate. If you've made a mistake or for any reason want to break out of this array input mode, just enter '@', the cancel symbol.

Interrogation cannot be used with *Sparse* arrays.



A well known Julia Set. 1 minute on a Macmini

***** As of January 2021 all 32 bit versions are obsolete. Most of**
***** This manual still applies to them, but there are no guarantees.**

Built-in functions

The built-in functions operate on numbers, arrays, or polynomials in arithmetic or algebraic fashion. Initially I intended each function to have a special one-character symbol as its name, but that gets unwieldy. Many have such symbols, and many also have "ordinary" mnemonic names, which always start with a capital letter. These names are reserved and cannot be used by the programmer for any other purpose. With very few exceptions, the name of the function precedes its argument. Also with very few exceptions, the parentheses around the argument are optional if the symbolic name is used but must be there if the ordinary name is used, unless the argument is an array. Putting the parentheses in when not necessary costs a very tiny additional amount of time to parse them.

Conventions:

In the following, if I make a statement like "x can be a polynomial but not a quolynomial," I mean that x cannot be a quolynomial with denominator $\neq 1$. Similarly, if I say "x can be a number but not a polynomial," I of course mean that it cannot be a polynomial of nonzero degree.

By *number* I mean a rational or modular number. By *scalar* I allow in addition a polynomial or quolynomial.

Return

This built-in allows a user's function (see the chapter on "Functions") to terminate and pass back a specified value. Syntax of use is *Return*(x), where x is any expression. Often not necessary, except when *Integer* is used.

Toot: sound the system beep.

Basic Arithmetic

$+, -, *, /$: obvious, except they can be applied to arrays as well as scalars, in which case they often act component-wise. More on this later under "array expressions".

The Increment Command:

A surprisingly large proportion of all commands written in programs are of the form $x := x + y$, and a great many of these are of the form $x := x + 1$. To obviate the need to look up x twice or read the 1, Fermat allows the programmer to write $x :+$ to increment x by 1, and $x :+(\ldots)$ to increment x by whatever is between the parentheses. Similarly, $x :-$ to decrement x by 1, and $x :-(\ldots)$ to decrement by $(\ldots)$. However, x must be an integer and less than 2^{28} in absolute value.

Implicit Multiplication:

If two names are juxtaposed Fermat assumes that they are to be multiplied, as in ordinary mathematical notation. Thus, $5x$ means 5 times x. As long as you leave a blank between them, the same works for two adjacent variables, $x\ y$.
Note: this assumed multiplication does not apply within array expressions or within functions with the *Integer* option, where the multiplication sign $*$ must always be used. See the later chapters on array expressions and on the "dangerous commands".

More Basic Arithmetic:

\ = integer division, i.e., divide and truncate. In rational mode, in $x\backslash y$, if x or y is not an integer, the denominators are ignored and the result computed from the numerators alone. This yields an awkward way to extract the numerator of a rational number – the built-in function *Numer* is more straight-forward. It can also be very confusing, especially when used with divide in the same expression. For example, $4/3\backslash 2 = 2$.

\ can also be used with arrays, as in $[x]\backslash 3$.

If x and y are polynomials, $x\backslash y$ acts as one would expect in the one variable case. For several variables, sometimes only a "pseudo-quotient" can be computed. See the later chapter on polynomials.

| = modulo. Naively, $n|m$ is the remainder produced by the division of n by m, but there are several complications. Suppose first that n and m are integers. Then the problem is what to do about negatives: should $(-2)|3$ be -2 or 1? The default in Fermat is to force a non-negative value, but the user may disable this extra step with the interpreter command &| (discussed in the earlier chapter).

In rational mode, if x or y is nonintegral, the denominators are ignored.

| can also be used with arrays, as in $[x]|3$.

If x and y are polynomials, $x|y$ acts as one would expect in the one variable case or when y is monic, otherwise only a "pseudo-remainder" can be computed. If x is a rational function ("quolynomial") then $x|y$ is not an error in Fermat.

Beware of sneaky syntax mistakes! $(2 + y)/3x$ does not mean $(2 + y)/(3x)$. It means divide by 3 and multiply by x.

$|\ldots|$ = absolute value. Note: The argument ... must be a *factor*, not an *expression*, i.e., $|x + y|$ is an error; $|(x + y)|$ is okay. Here we are using the following standard grammatical terms: An algebraic *expression* is a sum of *terms*. A *term* is a product or quotient of *factors*. A *factor* is either a simple variable, a constant, a function call, or an *expression* bounded by parentheses.

Numerical Functions

$ = greatest integer function. Can be applied to polynomials, in which case it is done to the coefficients.

$x\hat{}n$ means x^n. In rational mode, n has to be a *small integer* ($< 2^{28}$ in absolute value).

Sqrt(x) = square root. Computes the (non-negative) square root. In rational arithmetic, this function returns the largest integer less than or equal to the square root. In modular mode, this function is disabled.

$n!$ = n factorial. If n is not an integer, it is truncated first.

Bin = binomial coefficient. If necessary, n and r are rounded to integers.

Isprime(n) = is n prime? 1 means n is prime, else it returns the smallest prime factor. n can be up to $2^{63} - 1$. The algorithm is elementary.

Numvars = number of poly vars attached.

Prime(n) $= n^{th}$ prime. $0 < n < 1230$.

Rand = random number. This function (which has no argument) returns the next in a sequence of pseudo-random numbers. The numbers are evenly distributed integers between 0 and $2^{28} - 1$. In modular mode, the integers are reduced by the modulus. The precise sequence is always the same. This sameness can be useful in debugging, but is also inconvenient at times. Fermat therefore has a variation, indicated by *RRand*, which factors in the time of day to produce a "truly" random sequence.

The array form $[x] :=$ *Rand* or $[x] :=$ *RRand* fills a *previously created* array with random numbers.

Additionally, one can set a global variable *divisor for random*, which is 1 by default. The integers returned by random are divided by this. For example, if you set this constant equal to $2^{28} - 1$, random will return fractions evenly distributed between 0 and 1. To set this variable, use either the command &_r, which will prompt you for the value, or the imperative form &(_r = <*expression*>).

$<$ = last computed value. Fermat has a hidden "system variable" where all computed scalars are put automatically. This function accesses that variable. Typical use: $x := <$ to move the value to the variable x. The array form $[<]$ is discussed below under "Array Functions."

Sigma = add up. As in the mathematical notation, Σ. Here are some examples:

Sigma$< i = 1, n >$ [$1/i$] $= 1 + 1/2 + 1/3 + \ldots + 1/n$

Sigma$< i = 1, n >< j = 1, m >$ ($a[i, j]$) = the sum of the elements in the matrix $[a]$ (assuming that the global initial index variable has been set to 1). Note the set brackets and the square brackets surrounding the expression to be added up. In the second example, round parentheses were used instead of square brackets for clarity; Fermat allows either. Any number of indexing assignments (inside the set brackets) listed one after the other, is allowed. The index variables i and j are actual variables in the Fermat session. Modular mode is automatically turned off during the loop control evaluations, i.e., between the angle brackets $<$ and $>$. The syntax here resembles that of loops, which are described later.

Prod = multiply out, as in standard mathematical notation, Π. Analagous to *Sigma*; see above.

Modulus. Returns the numerical value of the modulus now in effect, or last in effect. If not in modular mode (i.e., ground ring Z/p) , may return a nonsensical value.

Modmode. Boolean function; returns 1 (true) when in modular mode, 0 (false) when not.

Powermod. See Appendix 4.

Time. It displays the time and date. Visible on startup.

Timecpu displays total CPU time since startup, returns 0. Compare &T and &^ .

True. Always returns 1.

False. Always returns 0.

Post 2023: There is a function Verbose to report if the verbose flag is on.

Polynomial and Quolynomial Functions

Deg. Two distinct uses: degree of a polynomial (or quolynomial), or number of elements in an array. Here we discuss the former. There are two variants: (1) $Deg(x)$ computes the highest exponent in x (any expression) of the highest precedence polynomial variable. (2) $Deg(x, i)$ computes the highest exponent in x of the i^{th} polynomial variable, where the highest level variable has the ordinal 1. $Deg(x, t)$ computes the highest exponent in x of the polynomial variable t. In modular mode, *Deg* returns an actual integer, not reduced modulo the modulus. For a quolynomial, it returns the degree of the numerator.

Codeg = "codegree," just like *Deg* except computes the *lowest* exponent.

= polynomial evaluation. $x \# y$ replaces the highest precedence variable everywhere in x with y. $x \# (u = y)$ allows for replacing other variables (here u) than the highest. x could be a quolynomial but y must be a polynomial only. $x \# (u = y_1, y_2, y_3)$ replaces u with y_1, the variable below u with y_2, etc. Similarly if $[t]$ is an array, $x \# (u = [t])$ replaces the variables from u on down with the entries of $[t]$ in column major order until $[t]$ is used up. It is an error if $[t]$ has too many entries. See the later chapter on "Polynomials".

Fermat also allows evaluation of polynomials at a square matrix. The syntax is $x \# [y]$. The highest precedence polynomial variable in x is replaced with the matrix $[y]$ and the resulting expression simplified. $[y]$ can contain entries that are quolynomials. This command was used in the example in the opening chapter of this manual.

Numb = is the argument a number? If so (as opposed to a polynomial or quolynomial), the result is 1, else it's 0.

Numer = numerator of a quolynomial. Also gives the numerator of a rational number.

Denom = denominator of a quolynomial. Also gives the denominator of a rational number.

Log2 = number of binary digits in the largest numerical coefficient of a quoly.

Equal and *Equalneg.* If-statements can involve a wasteful evaluation of arguments, as in: $if\ x = y[i]$ then .... when x and $y[i]$ are large polynomials or rational functions. First x is evaluated, which means duplicating its storage, then $y[i]$ is duplicated, etc. *Equal* and *Equalneg* do not duplicate storage. The syntax is rather obvious: $if\ Equal(x, y)$ then ... where x and y can be any existing scalar variables, including array references. $Equalneg(x, y)$ is logically equivalent to $x = -y$. The speedup in time can be profound.

Move and *Swap.* In the same vein as *Equal* and *Equalneg*, $Move(x, y)$ will transfer the data of x into y. y's previous value is discarded and at the end x is 0. x and y must be existing scalar variables. Similarly $Swap(x, y)$ interchanges two such variables' data.

Remquot = remainder and quotient. $Remquot(x, d, q)$ returns the (pseudo) remainder (r, say) of dividing d into x and assigns the (pseudo)quotient to q. $c^k x = qd + r$, where c is the leading coefficient of d and $k = deg(x) - deg(d) + 1$ (unless d is a number or c is invertible; then $k = 0$).

Coef = coefficient in a polynomial (or quolynomial). Suppose first that only one polynomial variable t has been adjoined. Then the syntax of use is either $Coef(x)$ or $Coef(x, n)$. x can be any expression. n must be a number. In modular mode, modular arithmetic is ignored while n is evaluated. In the first form, without n, the leading coefficient is computed.

If x is a quolynomial, the denominator is ignored.

If there are several polynomial variables, the exact coefficient desired is specified by listing the exponents of the variables in precedence order, highest first. For example, if t, u, and v have been adjoined in that order, and
$x = (3u - 3t - 3)v^2 + (7u^2 + (6t - 8)u + 3t^2 + 3)v + u^3 + (-5t - 3)u^2 + (3t^2 + 3)u - t^3 - 3t - 1$
then $Coef(x, 1, 2) = 7$, $Coef(x, 1, 1, 0) = -8$, $Coef(x, 0, 2, 1) = -5$, $Coef(x, 2) = 3u - 3t - 3$, $Coef(x, 0, 0) = -t^3 - 3t - 1$, $Coef(x, 0, 0, 0) = -1$. $Coef(x, t, n)$ is the coefficient in x of t^n.

Lterm = leading term of a polynomial. Follow it with a polynomial expression, whose leading term will be computed.

Lcoef = the leading coefficient of a polynomial.

Nlcoef = the leading numerical coefficient of a polynomial. Unlike *Lcoef*, always returns a number.

Ntcoef = the trailing numerical coefficient of a polynomial. That number is always an actual coefficient, so can never be 0.

Zncoef = the "last" numerical coefficient of a polynomial. It will be 0 if there is no constant term.

" = derivative of a polynomial (or quolynomial) with respect to the highest precedence variable. This symbol is shift-'. Precede it with an expression to be differentiated.

GCD = greatest common divisor, as in $GCD(x, y)$, x and y can be numbers or polynomials, but not quolynomials. If numbers, the result is always positive, except that $GCD(0, 0) = 0$. $GCD(0, x) = x$ if x is not 0. If they are both polynomials, the result always has positive leading coefficient. In cases where the ground ring is a field, the result has leading coefficient 1.

EGCD = *extended GCD*, as in $EGCD(x, y, u, v)$, x and y are one-variable polynomials over a finite field. Compute u and v s. t. $u * x + v * y = GCD(x, y)$.

Content = the content of a polynomial relative to the highest polynomial variable; i.e., the GCD of all its coefficients. $Content(x, i)$ is relative to the i^{th} variable. $Content(x, t)$ is relative to variable t.

Numcon = numerical content, the GCD of all its numerical coefficients.

Var. Followed by an expression that evaluates to a positive integer, as in $Var(i)$, returns the i^{th} polynomial variable, counting the highest (last created) as 1.

Height = the difference between the levels (ordinals) of the polynomial variables in an expression. For example, if three variables have been attached, say t, u, and v in that order, and $x = v + t$, then $Height(x) = 3$. If $y = u + t$, then $Height(y) = 2$. $Height(t) = 1 = Height(u)$.

Level = the ordinal position of the highest precedence polynomial variable in an expression. For example, if three variables have been attached, say t, u, and v in that order, and $x = v + t$, then $Level(x) = 1$. If $y = u + t$, then $Level(y) = 2$. $Level(t + 2) = 3$. $Level(u) = 2$.

Raise = Two forms: $Raise(x)$ and $Raise(x, i)$. In the first, replace each polynomial variable with the variable one level higher. In the situation of *Level*, above, $Raise(y) = v + u$. $Raise(t + 2) = u + 2$. $Raise(x)$ is an error. The second form allows the user to provide an expression i that evaluates to a positive integer, and raises x that many levels, if possible.

Lower = The inverse of *Raise*. See above.

Rcoef = change the coefficient of a term within an existing variable. For example, if only one polynomial variable t exists, *Rcoef*$(x, n) := y$ will change the coefficient of t^n in x to the expression y. y must be a number.

With several polynomial variables, the idea is similar to *Coef*. In *Rcoef*$(x, \ldots) := y$, y must be suitable to be such a coefficient, else an error occurs. For example, if t, u, and v have been adjoined in that order, and

$$x = (3u - 3t - 3)v^2 + (7u^2 + (6t - 8)u + 3t^2 + 3)v + u^3 + (-5t - 3)u^2 + (3t^2 + 3)u - t^3 - 3t - 1$$

Rcoef$(x, 2) := 3v - 3u - t$ is an error. So is *Rcoef*$(x, 0, 0) := -u^3 - 3t - 1$.

In *Rcoef*$(x, \ldots) := \ldots$, x must be a polynomial, not a quolynomial. But in the latter case, the commands *Rcoef*$(Numer(x), \ldots) := \ldots$ and *Rcoef*$(Denom(x), \ldots) := \ldots$ are useful.

Totdeg = total degree. See appendix four.

WDeg = Withdraw subpolynomial relative to a variable list. See appendix four.

Divides = *Divides*(n, m): does n divide evenly into m? See Appendix 4.

Deriv = *Deriv*(x, t, n) returns the n^{th} derivative of x with respect to t.

Vars(x) = number of variables that actually occur in x.

Array Functions

Most of the ordinary arithmetic built-in functions can be applied to arrays, as in $[y] :=$ \$$[x]$. See Appendix 2, last column.

Sparse arrays are implemented in Fermat. This is an alternate mode of storing the data that constitute the array. In an "ordinary" $n \times m$ array, nm adjacent spots in memory are allocated to hold the entries in the array. If an array consists of mostly 0's, this is wasteful of space. In a *Sparse* implementation, the non-zero entries only are stored via list structures. The fact that an array has the *Sparse* or the "ordinary" storage structure is often transparent to the user; however, some of the functions listed below do not work on *Sparse* arrays. More on *Sparse* arrays later, under "Variables and Arrays," "Expressions and Assignment," and "Array Expressions."

Det, is used in several ways to compute a scalar from an array argument. If used by itself on a square matrix, *Det* is determinant. *Det*#$([x] = a)$ returns the number of entries in $[x]$ that equal a.

Similarly *Det*#$([x] > a)$ and *Det*#$([x] < a)$ compute the number of entries of $[x]$ larger or smaller than a. In modular mode (ground ring $\boldsymbol{Z}_n$), the order is the obvious one on the set of elements of $\boldsymbol{Z}_n = \{0, 1, 2, 3, \ldots, n-1\}$. If any entry is a polynomial, an error results. *Det*^$[x]$ returns the index of the largest element of $[x]$ (in column major order if $[x]$ is a matrix). *Det*_ $[x]$ returns the index of the smallest element of $[x]$. *Det*_+$[x]$ returns the index of the smallest nonzero element of $[x]$, or -1 if there is no such element.

Fermat allows a wide range of data types for the entries of a matrix – integers, rationals, modular numbers, polynomials, quolys, polymods. No single method is best for all cases. Fermat uses expansion by minors, Gaussian elimination, and reducing modulo n for some n's. (Also Gauss-Bareiss and LaGrange interpolation; see appendix four.) The last of these

is used for matrices of integers or polynomials with integer coefficients. The actual determinant can be reconstructed from its values modulo n (for a "good" set of n's) by the Chinese Remainder Theorem. Alternatively, it is often possible to work modulo an easily computed "pseudo determinant" known to bound the actual determinant. Gaussian elimination is applicable whenever one can invert any nonzero element in a matrix. If the matrix is small enough, expansion by minors is faster (see the command &D.) Gaussian elimination can be problematical in modular arithmetic over a nonprime modulus, in polynomial rings, and in polynomial rings modulo a polynomial. Fermat has heuristics to guide its choice of method.

For many matrices of polynomials, it is faster to figure out the degree of the determinant, evaluate the determinant at a set of integers, and then interpolate to compute the determinant. A short Fermat program to do this is given in Appendix 3, example 2. This method is built into Fermat, in which the interpolation algorithm is probabalistic, with very high probability of success. It is stunningly fast, especially for two or more variable polynomials.

Nonetheless, if there are many polynomial variables and the matrix is sparse, expansion by minors can be by far the fastest method. Setting the determinant cutoff (with &D) at least as large as the number of rows will force Fermat to do this method.

As of October 2009, the LaGrange modular determinant coefficients can be dumped to a file, rather than stored in RAM. This can be a big space saving when doing a very large computation. The command is &(_L=1). In other words, if the highest precedence variable is x and lower ones are $y, z, \ldots$, and if a determinant $c_n x^n + c_{n-1} x^{n-1} + \ldots$ is being computed with LaGrange interpolation, the coefficients $c_0, c_1, ..., c_n$ (which are polynomials in $y, z, \ldots$) will be dumped to the output file to save RAM.

SDet = "Space-saving determinant." Space is saved when computing over ground ring Z using LaGrange interpolation and the Chinese Remainder Theorem. Fermat has aways used the the CRT algorithm from Knuth volume 2 on page 277 (exercise 7). The advantage is one can do everything "on the fly." The disadvantage is that if one will need, say, 50 primes, then every determinant modulo the 50 primes is stored until the end, when the answer is computed over Z by combining them. That might need too much space. Instead, $SDet$ implements formulas 7-9 on page 270 of Knuth.

The disadvantage is it can't be done on the fly: one needs to know in advance how many primes will be needed. The user must (over)estimate this number.

Input: integer and a square matrix of polynomials. Output: determinant. Call: $SDet(n, [m])$. n = how many primes will be needed. $SDet$ is not guaranteed to work with the more sophisticated options of Det, i.e. $Det([m4], r, d1, d2)$.

Adjoint = adjoint of a square matrix.

Chpoly = characteristic polynomial of a square matrix. Parentheses mandatory. See appendix four.

Sumup = add up the elements of an array.

Trace = trace of a matrix.

Minpoly A sophisticated probabalistic algorithm for computing the minimal polynomial of matrices. See Appendix Four.

Altmult. Multiply two matrices using the algorithm of Knuth volume II, p. 481. A big time saver when multiplication in the ring is much slower than addition. Especially good for Polymods (see that chapter). Syntax is *Altmult*$([x],[y])$.

Altpower. Use *Altmult* to take a matrix $[x]$ to the power n. Syntax is *Altpower* $([x],n)$.

MPowermod$([x],n,m)$ computes $[x]^n \ mod \ m$, analagously to *Powermod*. See Appendix Four.

Reverse$[a]$ will reverse the elements in Array $a[n]$. If $[a]$ has one column, this is obvious. Otherwise, the exact behavior depends on whether $[a]$ is a standard array or a sparse array. For standard, each $a[i]$ is swapped with $a[n+1-i]$, where you think of $[a]$ as in column-major order. For sparse $[a]$, the rows are swapped.

Trans = transpose matrix, as in $[y] :=$ *Trans*$[x]$.

STrans = transpose a matrix in place, as in *STrans*$[x]$. Much fqster than *Trans*.

Diag refers to the diagonal of a matrix, as in *Diag*$[y] := [x]$. $[x]$ is considered a linear array. The diagonal of $[y]$ becomes the entries of $[x]$. If the name $[y]$ does not yet exist, a new square matrix will be created with off-diagonal entries 0. If square matrix $[y]$ of the right size (i.e., rows equal to the number of entries of [x]) does exist then the off-diagonal elements are not changed.

Dually, *Diag* can be used on the right side of an assignment, as in $[y] :=$ *Diag*$[x]$, which sets $[y]$ equal to a linear array consisting of the diagonal elements of $[x]$. $[x]$ does not have to be square.

There is a small subtlety in nonsquare arrays. *Diag* expects the number of diagonal entries to equal the rows of the matrix. For example, if $[e]$ is a 4×3 matrix, *Diag*$[e] :=$ $[(9,9,9,8)]$ is ok – but only the three 9's will go into $[e]$.

To create a diagonal matrix with all entries equal to a constant, say 1, you can use the easier form $[x] := [1]$, if $[x]$ already exists as a square matrix.

Cols$[x]$ = number of columns of array $[x]$.

Deg = degree of a polynomial (or quolynomial), or number of elements in an array. In the first case, follow it with any expression. If that expression is a number, then of course the result is 0. In modular mode, it returns an actual integer, not reduced modulo the modulus. For a quolynomial, it returns the difference in degrees of the numerator and denominator.

When used with arrays the next character must be the square bracket, [. *Deg*$[x]$ = total size of array $[x]$ (rows $\times$ columns).

$[<]$ = last computed array. Analogous with the preceding use of $<$, Fermat has a hidden system array. If you type the command $[x]+[y]$, arrays $[x]$ and $[y]$ will be added and, since you didn't provide an assignment of the result, the result will go into the system array. You can later access it by typing, for example, $[z] := [z] + [<]$. Subarrays cannot be used with $[<]$.

Note that the command $[z]+2$ will display 2 plus every element in $[z]$, but typing $2+[z]$ will produce an error. In reading from left to right, Fermat encounters the 2 first and cannot later switch to array parsing.

_ = concatenate arrays; glue two arrays together to form a larger one, as in $[z] := [x]$ _ $[y]$. Neither array can be *Sparse*. See the chapter on array expressions.

Iszero = is the argument (an array) entirely 0? If so, return 1, else return 0. Syntax: *Iszero*$[x]$.

Switchrow = Interchange two rows in an array. Syntax: Switchrow$([x], n, m)$. The matrix itself will be changed.

Switchcol = Interchange two columns in an array. Syntax: Switchcol$([x], n, m)$. The matrix itself will be changed.

Normalize = Normalize a matrix. The matrix must not be *Sparse*. Convert it to a diagonal matrix, i.e., all off-diagonal entries will be 0. The matrix itself will be changed. If requested, Fermat will also return the change of basis matrices that it used in normalizing. Possible invocations include *Normalize*$([x])$ and *Normalize*$([x], [a], [b], [c], [d])$. In the second case, matrices $[a]$, $[b]$, $[c]$, and $[d]$ will be returned that satisfy $[a] * [x'] * [b] = [x]$, where $[x']$ is the original $[x]$, and where $[c] = [a]^{-1}$ and $[d] = [b]^{-1}$. The value returned by *Normalize* is the rank of $[x]$.

The inverse change of basis matrices $[c]$ and $[d]$ are provided because it is far faster to compute the inverses of $[a]$ and $[b]$ "along the way" then it is to use matrix inversion after *Normalize* finishes. However, the computation of each of these matrices adds to the total execution time, and your application may not need them all. Therefore, Fermat allows you to omit any one(s) you wish. For example, *Normalize*$([x], , [b], , [d])$ and *Normalize*$([x], [a], , [c])$ are possible. Each comma promises that an argument will eventually follow, so *Normalize*$([x], [a], [b], [c],)$ is illegal.

See also the commands FFLU and FFLUC.

Colreduce = Column reduce a matrix. The matrix may NOT be *Sparse* (see *Rowreduce*). By column manipulations, the argument is converted to a lower triangular matrix. The matrix itself will be changed. If requested, Fermat will also return the change of basis (or conversion) matrices that it used in normalizing. Possible invocations include *Colreduce*$([x])$ and *Colreduce*$([x], [a], [b], [c], [d])$. In the second case, matrices $[a]$, $[b]$, $[c]$, and $[d]$ will be returned that satisfy $[a] * [x'] * [b] = [x]$, where $[x']$ is the original $[x]$, and where $[c] = [a]^{-1}$ and $[d] = [b]^{-1}$. The value returned by *Colreduce* is the rank of $[x]$.

As in *Normalize*, one may omit computing some of the conversion matrices, for example, *Colreduce*$([x], , [b], , [d])$ or *Colreduce*$([x], [a], , [c])$.

Rowreduce = Row reduce a matrix. The matrix must be *Sparse*. Exactly like *Colreduce* but for sparse arrays and row reduction.

Smith = Normalize a matrix of integers into *Smith normal form*. The matrix may be *Sparse*. This function can only be used in rational mode, and will work only if every entry is an integer. (Any denominator encountered will be ignored, with unpredictable results.) By row and column manipulations, the argument is converted to a diagonal matrix of non-negative integers. Furthermore, each integer on the diagonal divides all the following integers. The set of such integers is an invariant of the matrix.

The matrix itself will be changed. If requested, Fermat will also return the integer change of basis (or conversion) matrices that it used in normalizing. Possible invocations

include *Smith*($[x]$) and *Smith*($[x], [a], [b], [c], [d]$). In the second case, matrices $[a]$, $[b]$, $[c]$, and $[d]$ will be returned that satisfy $[a] * [x'] * [b] = [x]$, where $[x']$ is the original $[x]$, and where $[c] = [a]^{-1}$ and $[d] = [b]^{-1}$. The value returned by *Smith* is the rank of $[x]$.

The change of basis matrices $[c]$ and $[d]$ are handled just as in *Colreduce* or *Normalize*.

If you do not require any conversion matrices then it is possible to greatly speed up *Smith* in most cases by working modulo a "pseudo-determinant", a multiple of the gcd of the determinants of all the maximal rank minors (see Kannan and Backem, SIAM Journal of Computing vol 8, no. 4, Nov 1979). Do this in Fermat with the command *MSmith*. Not to work modulo some number invites a horrendous explosion in the intermediate entries. On the other hand, for relatively small matrices or sparse matrices, it's faster to forgo the modding out. Fermat will compute the pseudo-determinant if the matrix is *Sparse*. If you already have a pseudo-determinant *pd* to use, use the syntax $MSmith([x], pd)$. (If the matrix is not *Sparse*, you *must* use the latter method. *Pseudet* may be helpful.)

Hermite = Column reduce a matrix of integers. The matrix may be *Sparse*. This function can only be used in rational mode, and will work only if every entry is an integer. (Any denominator encountered will be ignored, with unpredictable results.) By column manipulations and row permutations, the argument is converted to a lower triangular matrix of integers. All diagonal entries are non-negative.

The matrix itself will be changed. If requested, Fermat will also return the integer change of basis (or conversion) matrices that it used in normalizing, exactly as in *Smith*, above.

If the matrix is "large" and "dense" a horrendous explosion is possible in the off diagonal intermediate entries, and in the entries of the conversion matrices.

Redrowech = the reduced row echelon form, good for elementary matrix equations of the form $AX = B$. (Other Fermat commands do column manipulations as well, which of course could be used to solve $AX = B$ but take an extra step.) Invoke with *Redrowech*($[a]$), where all columns but the last in $[a]$ represent the matrix A and the last column represents B (i.e., *Redrowech* never pivots on the last column.) Alternately, *Redrowech*($[a], [u], [v]$) will return in $[u]$ the transition matrix representing the row manipulations it used in normalizing $[a]$. $[v]$ is $[u]^{-1}$. As in other similar Fermat commands, you can also do *Redrowech*($[a], , [v]$).

FFLU and *FFLUC* are for fraction-free LU factorization of matrices. See the two articles in the September 1997 SIGSAM Bulletin: "Fraction-free Algorithms for Linear and Polynomial Equations," by Nakos, Turner, and Williams; and "The Turing Factorization of a Rectangular Matrix," by Corless and Jeffrey. $FFLU$ is invoked as: $FFLU([x], [p], [l], [a], [b])$. $[x]$ is the $n \times m$ matrix to be factored. $[p]$ is an $n \times n$ diagonal matrix consisting of the pivots used, $[p] = diag(p_1, p_2, ..., p_{n-1}, 1)$. $[l]$ is the unit lower triangular matrix, the first factor. $[a]$ (optional) is the $n \times n$ permutation matrix of row swaps. $[b]$ (optional) is $[a]^{-1}$. At the end, $[x]$ is in upper triangular form. Let $[z]$ be a copy of the original $[x]$. If $[f]$ and $[g]$ are the matrices called f_1 and f_2 in the Corless and Jeffrey article, then at the end one has $[f] * [a] * [z] = [l] * [g] * [x]$. Note that $[f]$ and $[g]$ are not computed by $FFLU$; however it is obvious how to get them from $[p]$.

$FFLUC$ allows column swaps as well as row swaps. In this way, the size of the pivots can be reduced. $FFLUC$ is invoked as: $FFLUC([x], [p], [l], [a], [b], [c], [d])$. As above, $[x]$ is the $n \times m$ matrix to be factored. $[p], [l], [a]$, and $[b]$ are the same as above. At the end, $[x]$

is in upper triangular form. $[c]$ and $[d]$ (optional) are permutation matrices coming from column swaps ($[d]$ is $[c]^{-1}$). More information is in Appendix Four.

Several other functions are described in Chapter 15 and Appendix Four.

Pivot Strategies: As of January 2009 there are options for the heuristics that direct the pivot choice in the normalization of matrices. This can have a large effect on time and space, though often it does not. The heuristic is set with the command &u or &(u = val). val is an integer from 0 - 5. The size or mass of a potential pivot can be measured by just its number of terms (called term# below), or by one of two mass heuristics (called mass0 and mass1 below).

Setting val =

0 is the default previous heuristic, which selects the least massive entry by the original mass0 heuristic.

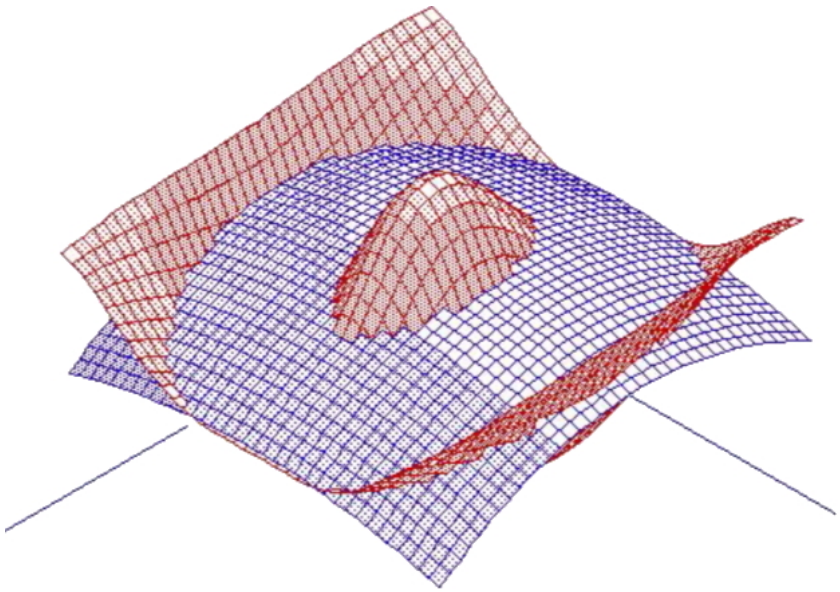
1 selects the 'lightest' entry where weight = term# + sum of term# for the entire row entry is in.

2 selects the 'lightest' entry where weight = mass0 + sum of term# for the entire row entry is in.

3 selects the 'lightest' entry where weight = mass1 + sum of term# for the entire row entry is in.

4 selects the 'lightest' entry where weight = term#.

5 is like 3 but also counts the weight of the column an entry is in.



Intersection of surfaces, resembling a manta.

Names

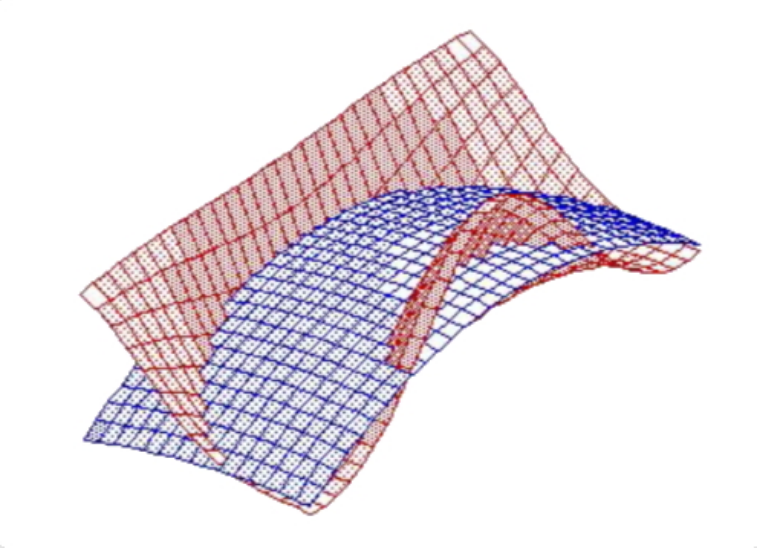
The user creates names for scalar and array variables, for files, for polynomial variables, and for functions. All but file names must be less than or equal to 10 characters long and consist entirely of lower case letters, upper case letters, the underline "_", and the 10 digits. Spaces and tab marks are ignored. Fermat distinguishes between upper and lower case letters.

The names of variables (scalar, polynomial, or array) must begin with a lower case letter. When referring to an entire array, array names are surrounded by the brackets "[" and "]".

The names of functions must begin with an upper case letter.

Names of files should be just like variable names.

If you create a file with the &s or &S command, say *abc*, the name of the file will be just that – *abc* with no *.txt* or other suffix. If you create a text file with some word processor or editor under Windows, the suffix *.txt* will probably be appended, whether you see it or not. To read such a file in a Fermat session, you will have to put the name in quotes and add the *.txt*, as in *'abc.txt'*.



Fermat's Cut command applied to the "manta" object of previous page.

Variables and Arrays

The name of a variable (see preceding chapter) must begin with a lower case letter. A variable is created by assigning a value to the name.

Fermat allows one or two dimensional arrays. The name must begin with a lower case letter. Before using an array, it must be created. To explicitly create an array called b of 10 elements, use *Array* $b[10]$; to create a 3×3 matrix c use *Array* $c[3,3]$. Several arrays can be declared on one line, such as *Array* $x[3,3]$, $y[4,4]$, $z[60,60]$ *Sparse*, $a[7,7]$. In modular mode, the dimensions are evaluated ignoring modular arithmetic. There can be both a scalar variable and an array of the same name.

The maximum array size is 1,000,000. The maximum total size of all arrays is 5,120,000. (This does not apply to Sparse arrays, for which there are no limits imposed by Fermat. However, the number of rows must be $< 2^{28}$ and the number of cols must be $< 2^{28}$, otherwise integer overflow will occur within the Fermat interpreter. This limit will probably be removed in future versions.)

When an array is created, its values are undefined.

Sparse arrays are implemented in Fermat. This is an alternate mode of storing the data that constitute the array. In an "ordinary" $n \times m$ array, nm adjacent spots in memory are allocated to hold the entries in the array. If an array consists of mostly 0's, this is wasteful of space. Furthermore, multiplying two such matrices $[x] * [y]$ is wasteful of time, since almost all multiplications $x[i,k] * y[k,j]$ result in 0. In a *Sparse* implementation, only the non-zero entries are stored in a linked list structure, in which each node contains a pointer to the actual data item and the row and column of the item.

A *Sparse* array is created by following the creation command with the keyword "*Sparse*," as in *Array* $x[5,5]$ *Sparse*. There is no limitation on the size of the matrix in Fermat. *Sparse* arrays do not contribute to the total array size limit. An array $[x]$ already created can be converted to *Sparse* format with the command *Sparse* $[x]$.

The fact that an array has the *Sparse* or the "ordinary" storage structure is often transparent to the user. More on *Sparse* arrays below, under "Expressions and Assignment" and "Array Expressions." See also "sparse access loops" in the index.

Arrays are accessed, or indexed, with the syntax $x[i]$, where i can be any expression evaluating to an integer. In modular ground ring, i is evaluated ignoring modular arithmetic. One has a choice of how to label the first element of an array – usually one thinks of the first element as $x[1]$, but sometimes $x[0]$ is more convenient. The default in Fermat is $x[1]$. This can be changed by entering the command &a, which switches the initial array index to 0. Entering &a again switches back to 1. Note that this is not a property of any particular array, but *of how all arrays are indexed.* Creation of arrays is not affected – it's still $x[8]$ for an array with 8 elements. For two dimensional arrays, the first element is then $x[0,0]$.

Dynamic Allocation of Arrays

Arrays that are no longer needed can be freed to provide space for new arrays. This is done with the cancel command, whose syntax is @[x], or, to free several, @([x], [y], [z]). As arrays are created and destroyed, space is allocated and freed within a linear list of available space in the order that the commands are received. If you have several arrays and free the first one created, the others are moved up to occupy the empty slots. Actually, this moving is not especially time consuming since only pointers need to be changed. Nonetheless, if you are going to do a lot of creating and cancelling of arrays, it is best to follow the *last in, first out* policy, thereby treating the linear list as a stack.

Expressions and Assignment

An *expression* is any algebraic formula following the usual rules of precedence and involving scalar variables, constants, function calls, or interpreter commands. Example:

$$x * (z + 2) + ?y + F(y, 2) + a[2, 1]$$

The syntax for assignment is

$$< varname > := < expression >$$

When given from the terminal, an assignment statement returns the value of *expression*. But when entered from within a function, the value of the statement is 0. See the later chapter on "Functions."

There is a shortcut *increment command*. Fermat allows the programmer to write $x :+$ to increment x by 1, and $x :+(\ldots)$ to increment x by whatever is between the parentheses. Similarly, $x :-$ to decrement x by 1, and $x :-(\ldots)$ to decrement by $(\ldots)$. However, x must be an integer and less than 2^{28} in absolute value.

Efficient Use of Storage

An innocuous statement like $x := x + y$ can cause problems if x and y are large polynomials with thousands, or even millions, of terms. The problem is that in evaluating the right side, x's storage is duplicated. That may blow out the RAM. This is silly since x is going to be changed anyway. Therefore there is an alternate syntax using the symbol $*$ to indicate that the value and storage of a variable are disposable, namely $x := *x + y$. Variants are $x := *x + *m[2, 3]$ and $x := *x * *y$. Should the user interrupt Fermat during the evaluation and inquire what x or y is, he will find it has been set to 1. See also *Move* and *Swap*.

When a variable is assigned a value, it is put on an "environment stack" of all active names. Any previous value is lost. The cancel or rubout command @ removes the name from the stack and destroys the value. The command &v displays the active variables (follow with a colon to supress the values).

Similarly, arrays are put on a stack of active array names. With arrays, however, every act of creation, such as *Array* $a[3, 3]$ puts a new array called a of dimension 3×3 on the stack. Previous arrays called a are not lost - they are just pushed down. Any reference to array $[a]$ is to the top most one. To reaccess the lower one(s), cancel the top one or rename it using the "rename command", $Rname[c] := \ '[d]'$.

Mixing arrays and scalars in an expression is sometimes legal. This is clearly not:

$$a := 2 + [c]$$

You can mix arrays and scalars in an array assignment statement,

$$[a] \ := < array \ expression >$$

For example, $[z] := [a] + 3$. Array expressions must involve arithmetic only, no function calls and no interpreter calls. See the next chapter, "Array Expressions."

To set $[c]$ equal to $[b]$, the command is $[c] := [b]$. In this case, $[c]$ does not have to have been explicitly created before; it will get the same dimensions as $[b]$. If an array $[c]$ of the same dimensions as $[b]$ already exists and is on the top of the stack of $[c]$ names, its old entries will be destroyed and new ones created equal to those of $[b]$. Otherwise, a new version of $[c]$ will be created.

Constant and Diagonal Matrices:

$[b] := 2$ will set every component of b to 2. $[b] := [1]$ will set $[b]$ equal to the identity matrix, if $[b]$ has been declared to be a square matrix. (If there is no $[b]$ yet, it is an error. If there is a non-square $[b]$, a new square $[b]$ will be created having the number of rows of the other $[b]$.) Any constant may be substituted for the 1, including polynomials. However, a polynomial inside the brackets must be written using Fermat's conventions of parenthesation and precedence, such as $[(5t^2+2t+1)u^2+(3t-1)u+7t^2-3t+1]$, assuming that polynomial variables t and u have been created, in that order. For more information, see the later chapter on polynomials, about "polynomial read-in."

Diagonal matrices are *Sparse* arrays.

Here is a further example. Suppose $[a]$ has been declared to be 4×4, but $[b]$ has not been created. Then this will work: $[b] := [a] + [1]$. The interpreter will figure out that the identity matrix [1] should be 4×4, because $[a]$ is. But just saying $[b] := [1]$ won't work – the interpreter has no way of knowing how big [1] is supposed to be.

Array Expressions

In Fermat one can write expressions involving matrices and scalars that follow familiar mathematical syntax, such as

$$[c] := [a] * [x] + 3 * [b]$$

If $[a]$ is an existing square matrix, $[b] := 1/[a]$ sets $[b]$ to the inverse of $[a]$, unless $[a]$ is singular.

Fermat returns the value 0 after an array assignment. It does not automatically display the new array.

In array expressions the multiplication sign $*$ must be used to effect multiplication. In ordinary expressions, if two variables or factors are juxtaposed, multiplication is assumed. That won't work here. The reason is ambiguities like $s[x]$. Is this s times $[x]$, or is it entry x in array $[s]$?

Ordinary and *Sparse* matrices can be mixed in expressions. If any term in a matrix expression is an ordinary matrix or is a scalar, the result will be ordinary (unless it would too big – then it's an error); otherwise it will be *Sparse*. This provides an inelegant mechanism for converting a *Sparse* matrix, say $[a]$, to an ordinary one: $[x] := [a] + 0$.

Fermat allows subarray expressions. That is, part of an array $[c]$ can be assigned part of an array $[a]$. For example,

$$[c[1 \sim 4, 2 \sim 6]] := [a]$$

sets rows 1 to 4 and columns 2 to 6 of $[c]$ equal to $[a]$. This assumes that $[a]$ is declared to be 4×5 and $[c]$ is at least 4×6. In defining the subarray, if one of the coordinate expressions (1, 4, 2, and 6 in the above example) is left out, the obvious default values are used. For example, if $[c]$ has four rows then $[c[, 2 \sim 6]] := [a]$ is equivalent to the above. Similarly, one can use expressions like $[c[3 \sim, 2 \sim 6]] := [a]$ or $[c[\sim 4, 2 \sim 6]] := [a]$, in which case the default lower row coordinate is the array initial index, 0 or 1. The extreme example of these shortcuts is $[x[,]]$, which is legal, if pointless. $[x[\ ]]$ is not legal.

Subarrays can be used in either the source or target of an assignment statement (right side or left side).

In subarray assignments, a vector declared to be one-dimensional (like $a[5]$) is treated as a column vector, i.e., $a[5, 1]$.

Both $![x[\ldots]]$ and $?[x[\ldots]]$ work with subarrays. *Diag* and *Det* work with subarrays.

As of April 2016, in 64 bit Fermat, subarray can be used with Sparse matrices in assignments and with %.

However, do not write expressions that mix subarrays of sparse and ordinary matrices! No guarantees there.

There is a similar function *Minors* that can; see Appendix Four.

Getting Data Into Arrays

Besides subarray expressions and the interrogation command ?[x], another way to easily get data into an array is with list input, similar to list output, described above. For example,

$$[x] := [(2, 3, 4, 6, 8)]$$

creates an array of the five indicated numbers. Note the bracket and parenthesis. If [x] had already been declared to be two-dimensional, the numbers will be inserted into the array in column-major order. If [x] previously existed but was not of this total size, a new [x] is created.

Yet another way to get data into a matrix is by setting up the data on the screen in rectangular array in the same format as Fermat's short form display of matrices (which has been discussed previously under the 'display' built-in function). For example,

```
>[y] := [[  0,    -1,    19,  '
            0,     1,    11   ]]
```

Any expression may be used to specify an entry, not just constants. Just put the cursor at the end and hit return.

For a *Sparse* array, one may use as well the long form, as in:

```
>[x] := [ [ 1,    [2, -1],    [3, 19 ]]  '
          [ 2,    [ 1, 8],    [2, -2 ]]  '
          [ 3,    [ 2, 1],    [3, 11 ]]  ]
```

The entries do not have to be in increasing column order within the rows. Furthermore, repeated columns are allowed, in which case the sum of all the given terms ends up in that column.

Yet another way to easily get data into an array is with "pseudo-loops". For example,

$$[x] := [\ < i = 1, n > < j = 1, n >\ \ 1/(i + j - 1)\ ]$$

will create the $n \times n$ Hilbert matrix. The index variables are actual variables in the Fermat session. (Of course, they don't have to be i and j.) Modular mode is automatically turned off during the loop control evaluations, i.e., inside the set brackets. The complete syntax is

[< i = <exp_1>, <exp_2>, <exp_3> > < j= <exp_4>, <exp_5>, <exp_6> > <exp_7>]

<exp_7> must be an algebraic expression – no loops, ifs, etc. (Of course if you want to create a matrix for which the expression <exp_7> would have to have ifs or loops, use a function to do so.) Each of the indexing expressions is truncated to an integer, if necessary, and must be a small integer ($< 2^{28}$). All six of them must be at least the array initial index. <exp_3> and <exp_6> are optional, as in for-loops. This command creates an <exp_2> $\times$ <exp_5> matrix if the array initial index is 1, or an ($<exp_2> + 1$) $\times$ ($<exp_5> + 1$) matrix otherwise. If <exp_3> or <exp_6> is not 1, then a matrix is created which has some undefined elements.

Arithmetic of Arrays

Multiplication of arrays $[a]*[b]$ follows this hierarchy: If $[a]$ is $n \times m$ and $[b]$ is $m \times p$, an $n \times p$ is created, as is usual in matrix multiplication. If $[a]$ and $[b]$ do not match this way but they are of the same total size, then component-wise multiplication is done. Otherwise it's an error.

Similarly for division. If $[b]$ is square and $[a]$ has the same number of columns, $[a]$ / $[b]$ means $[a]$ times the inverse matrix of $[b]$. Otherwise it means componentwise division, if the sizes are the same.

$[a]$ | 3 takes each entry of $[a]$ modulo 3. If $[a]$ and $[b]$ have the same total size, $[a]|[b]$ returns each entry of $[a]$ modulo the corresponding entry of $[b]$ (otherwise it's an error). Similarly for $[a] \setminus [b]$ and $[a] \setminus 3$.

$2*[a]$, or $[a]*2$, multiplies every component of $[a]$ by 2. $[a]+3$ adds 3 to every component of $[a]$, and so forth. Suppose you want a 4×4 square matrix $[y]$ with 7's on the diagonal and 1's everywhere else. Use $[y] := Diag[(6,6,6,6)] + 1$.

You can add or subtract arrays of the same total size. If they aren't of the same declared dimensions, such as adding a 2×3 to a 3×2, the result may not be what you thought (try it).

Array exponentiation is implemented. The syntax is just like scalars, $[x]\text{^}n$. n must be a real integer. $[x]$ must be a square matrix. If n is negative, $[x]$ must be invertible.

Other Miscellaneous Array Operations

All arrays can be accessed via the syntax $x[number]$, even if they were declared to be 2 dimensional. This is occasionally useful. For example, if you have an array $x[4,4]$, $x[7]$ means $x[2,3]$. (Assuming that the array initial index is 1. See the built-in function &a.)

Matrices can be normalized with the built-in functions *Normalize* and *Smith*. Their kernels can be computed with *Colreduce* and *Hermite*. See the chapter on built-in functions for other array built-ins.

Arrays can be "adjoined" or "concatenated" with the operator _. The syntax is $[z] := [x]$ _ $[y]$. If two arrays have the same number of columns, the result is as if one array were put "under" the other. If not but they have the same number of rows, it is as if they were put next to each other. Otherwise an error results. It is also legal to write $[w] := [x]$ _ 12, in which case 12 is appended to the end of $[x]$ and *a one-dimensional* $[w]$ is the result. Similarly, the scalar could be the first argument. This command is mostly of use for manipulating strings (see the next chapter on character strings).

There are syntactic restrictions on array expressions that do not apply to scalar expressions. For example, array expressions may not contain unlimited nesting of parentheses. An expression like $x*(x+x*(x+x*(x+x*(x+y))))$ may very well produce an error if x and y are replaced by arrays. Such complex syntax is not necessary. Instead, first set $[d] := [x] + [y]$, then multiply $[x]*[d]$, etc.

With either syntax $[y] := 1/[x]$ or $[y] := [x]\text{^}-1$, the inverse of $[x]$ is computed via Gaussian elimination, modular methods, or the Leverrier-Faddeev algorithm (See example 4, Appendix 3). The various versions of Fermat have heuristics for choosing a method.

The Array of Arrays

Fermat does not allow general three-dimensional arrays, but there are times when they are very convenient. For example, a group may be represented as a set of matrices, and one would naturally wish to access the ith element of the group as $G[i]$, or something like that. To provide for this, Fermat has a system wide *array of arrays*. 400,000 pointers are provided, each of which can be set to "point to" or "be an alias for" an array that has already been created in the usual manner. These pointers are accessed by the name %.

Suppose that $x[3,3]$ and $y[2,2]$ have been created. To associate the first pointer with $[x]$ use the command %[1] := $[x]$. Similarly %[2] := $[y]$ associates the second with $[y]$. Then %[1][2,2] is the same as $x[2,2]$, %[2][1,2] is the same as $y[1,2]$, etc. **Note Carefully:** $x[2,2]$ is not *duplicated* to become %[1][2,2], rather %[1][2,2] *is* $x[2,2]$. The assignment %[1][2, 2] := 3 changes $x[2,2]$.

Note that the array of arrays allows a more general data structure than typical three-dimensional arrays, because the arrays being indexed are arbitrary.

Until 2011, the pointer %$[n]$ could not appear by itself on the right in an assignment statement. Of course, an element of an array, such as %$[n][k,m]$ can appear there. In other words, you can't do $[x]$:= %[1]. When assigning to a pointer, the right side must be a simple array name, no expressions and no subarrays.

It is now (2011) possible to assign pointers, as %[1] := %[2]; no storage is moved. This is good for swapping data. One can also do $[c]$:= [%[1]].

Display and subarray work with array of arrays. Changing the array initial index with &a affects %. Interrogation from the terminal (the ? command) does not work with %.

Beware of dangling pointers! If %[1] is associated with $[x]$, and then $[x]$ is cancelled, further use of %[1] will probably crash the system. Perhaps % belongs in the chapter on "Dangerous Commands"!

Example one in Appendix 3 would have been cleaner using %.

Functions

All programming languages have the idea of "procedures" or "subroutines" that perform specific, often repeated tasks. In Fermat, these are called "functions."

Every function returns a value, the last expression it computes. (Recall that assignment statements within functions yield the result 0.) One can also use the *Return* built-in function to exit a function with a certain value, as in $Return(x)$.

Unlimited recursion is allowed.

The syntax of a function definition is

$$Function \ < left\ hand\ side > \ = \ < right\ hand\ side > \ .$$

"Func" may be substituted for "Function." Note the period at the end.

The left hand side must look like <*func name*> $(x, y, ...)$ or just <*func name*>. A <*func name*> is a name that starts with a capital letter. The right hand side is explained below under "The Main Body of the Function."

The name of a function must begin with a capital letter. The user can redefine a function F simply by typing a new definition. The function definitions are put on a stack, so any reference to F is to the topmost. $@F$ destroys the topmost F. $@<F>$ destroys, or *purges*, all previous definitions of F. The commands &f, or &F display the current function definitions. To also see the body of the functions, enter &f$*$.

It is recommended that most functions be created in a file with a text editor before Fermat is invoked (save the file as "text only"). However, it is possible to edit your function definition from the terminal. You must copy the old function, paste it after the cursor, make changes there, and then hit <*return*>. To facilitate this editing, when Fermat displays a function it adds the continuation character ‘ (upper left key on keyboard) to each line.

The order that the functions are defined (in the input file or the Fermat session) is completely irrelevant. Of course, if you *invoke* F and F invokes G, G must have been defined.

In $F(x, y, z, ...)$, $x, y, z, ...$ are the *parameters*. When the function is called, the interpreter evaluates the arguments and puts the parameters on the environment stack with those values. This mechanism of parameter - argument assignment is called *call by value*. When the function ends, those names (and their values) are popped off the stack. This is the same basic strategy used by APL, Lisp, and other languages, but is unlike Pascal, C, or Fortran. If you have a variable called x whose value is 3, and then you call a function F with parameters x, y, while F is executing x means F's parameter. When F is finished, its parameter x is popped off, allowing your original x with value 3 to become visible again.

Fermat allows another kind of parameter - argument correspondence called *call by value-result*. Syntactically, a value-result parameter is indicated by putting the exponentiation sign ^ in front of the name in the parameter list of the function, as in $F(x, \hat{}y)$. The argument passed to y must be a single variable name, not an expression. This provides a mechanism for allowing the changes made to y to survive the termination of F. Suppose you invoke F with $F(3, z)$. Then as the parameters are matched with their arguments, y gets z's value. As the function executes, any change to y, such as $y := y + 1$, is done only to y. When the function ends, z gets y's value. (*Therefore, you must not have cancelled* z,

although Fermat should be able to catch this mistake.) For more information, consult any text on the theory of programming languages.

In Fermat, the same syntax in some built-in functions implements *call-by-reference.* For example, time and compare $Terms(x)$ and $Terms(\hat{}x)$ for a large x.

Local Variables

A parameter not matched by an argument when the function is called becomes a *local variable* with initial value 0. For example, if you define $F(x, y, z)$ and invoke F with $F(a, b+c)$, z is initialized to 0 and cancelled when F ends. In the meantime, z is on the environment stack, suppressing or hiding any earlier z's. Any function that F calls has automatic access to z, as long as it (or some other function) doesn't create another z.

If z is a value-result parameter, it must be matched with an argument at function invocation time, and so cannot become a local variable.

The Main Body of the Function

The right hand side of a function (abbreviated $< r.h.s. >$) is either an expression, an assignment, an array expression, a for-loop, a while-loop, an if-statement, or a sequence of these separated with semicolons. The function definition ends with a period. The only other places were periods are allowed in a function definition are inside literals and inside comments.

If-Statements

The syntax of an if-statement is

if <*condition*> *then* <*r.h.s.*> [*else* <*r.h.s.*>] *fi*

where the first alternative is chosen if the condition is true, and the second (optional) alternative if it is false. Note the reserved word "fi". This if-fi syntax originated in Algol68. If the second is absent and the condition is false, nothing happens – it's as if the statement were not there (except for possible side effects introduced by the evaluation of the condition).

There are two ways to form a $<$ *condition* $>$. The first is simply to provide any expression. If the expression evaluates to 1, that's interpreted as true; anything else is false. You can set a variable, say, *condition* to 1, and write loops like

while condition do ...

Secondly, (and more conventionally) define a *simple condition* as:

<*expression*> $<$, $=$, $>$, $>=$, $<=$, or $<>$, <*expression*>

Simple conditions can be combined to form a *condition* with the Boolean connectives *and*, *or*, and *not*, such as:

if $i > 3$ *or* $x < 4$ *then* ... ...

While-Loops

The syntax of a while-loop is

while <*condition*> *do* <*r.h.s.*> *od*

Note the "od." The condition is evaluated, and, if true, the loop is entered. At the end of the statement(s) in the loop, the condition is evaluated again, and, if true, the loop is entered again. This pattern repeats until the condition is false.

For-Loops

The syntax of a for-loop is

for <*var name*> = <*expression*>, <*expression*> *do* <*r.h.s.*> *od*

or

for <*var name*> = <*expression*>, <*expression*>, <*expression*> *do* <*r.h.s.*> *od*

or

for <*var name*> *from* <*expression*> *to* <*expression*> *do* <*r.h.s.*> *od*

or

for <*varname*> *from* <*expression*> *to* <*expression*> *by* <*expression*> *do* <*r.h.s.*> *od*

The first two expressions are the initial and final values for the index variable. The optional third expression is the size of the increment. It can be negative. At the beginning of the loop, these three expressions are evaluated and truncated to integers, if necessary. The resulting integer in all three cases must be less than 2^{28} in absolute value. After each pass, the index variable is incremented. If it exceeds (or is less than, for a negative increment) the terminating value, the loop is over. After the conclusion of the for-loop, the index variable has value one more than the terminating value (or one less if the increment was negative). It is possible for the "< *r.h.s.* >" to contain a statement that (apparently) changes the index variable. This confusing practise should be avoided. It may cause a crash.

The words *if, then, else, for, while, do, from, by, fi,* and *od* are key words and should not be used for any other purpose inside functions.

Early Termination of Loops

Fermat has no "goto" statement; instead there is a "leave loop" command &>, a "cycle" command &], and an "exit function" command &}. "Leave loop" and "exit function" are self-explanatory. The cycle command causes Fermat to skip the remaining statements in the loop body and return to the condition or incrementation heading the loop.

Note that the leave loop command &> is independent of function structure. That is, if F calls G and &> appears in G after any loop in G, then upon return to F, the first loop that Fermat enters (or had entered) will be immediately exited. In general, &> *forces immediate termination of the next (or present) loop that Fermat enters (or has entered).*

Examples

$$F(x) = if\ x < 0\ then\ -x\ else\ x\ fi.$$

is absolute value.

$$W(n) = if\ n < 2\ then\ 1\ else\ n * W(n-1)\ fi.$$

is the factorial function.

$$G(n; i, s) = \ s := 1;\ for\ i = 1, n\ do\ s := s * i\ od;\ s.$$

is also n factorial. A typical call to G would be $G(6)$, in which i and s are unmatched parameters. That's fine - they are initialized to 0. The semicolon in the parameter list is just for the convenience of the user, to separate the real parameters from the local variables. A comma would have sufficed. (Of course, the built-in absolute value and factorial functions would execute far more quickly.)

Sparse Access Loops

There is a need for a way to work efficiently with sparse arrays. For example, suppose you have a sparse array of 60000 rows and 50000 columns with only 10 or so entries in each row (this is quite realistic). Suppose you wanted to add up all the entries. Naively, one could write something like:

$for\ i\ = 1, 60000\ do\ for\ j = 1, 50000\ do\ sum\ :=\ sum + x[i, j]\ od\ od$

But this will do 3,000,000,000 additions, almost all of which are adding 0! This is a preposterous waste of time. The solution is "sparse column access loops" for sparse arrays. The syntax is, continuing the example above,

$for\ i\ = 1, 60000\ do\ for\ j = [x]i\ do\ sum\ :=\ sum + x[i, j]\ od\ od.$

"*for j = [x]i do*" means find the i^{th} row of $[x]$ and let j run down it – of course encountering only the entries actually there! So j takes on whatever the column indices are in which $x[i, j] \neq 0$. $[x]$ must be an existing sparse array, and i must have a value suitable for $[x]$ at the start of the loop. More generally, one may use the syntax: *for j = [x]i,k do ...* . Here i and k both refer to rows of the sparse matrix $[x]$. At the start of the loop, all nonzero column coords in both rows are found. Then as the loop proceeds, j runs through those values in order. Any number of row indices is allowed. There is no analogous procedure for "sparse row loops" due to the way Fermat stores sparse matrices. If necessary, transpose the matrix.

Arrays as parameters and arguments

$$F(x, n; i, s) = for\ i = 1, n\ do\ s := s + x[i]\ od;\ s.$$

adds up the first n elements of array $[x]$. A typical call would be $F([a], 20)$. Notice how to pass an array argument. (The same function could be accomplished much faster with the built-in *Sumup* or *Sigma* functions.)

Comments

Comments within functions are implemented with the curly brackets { and }. Any character string is allowed between them, but note that **the first } terminates the comment.** A comment may extend over more than one line. When editing a function definition on the screen, a multiline comment does not need to be terminated with the continuation character. Indeed, if it is, something different will result – try it. It is possible to have literals (quoted strings) inside comments and visa-versa, but this very confusing practice should be avoided.

The best practice is to devote an entire line to a comment. However, it is possible to use less – for example, a comment may be placed between the end of an expression and the semicolon at the end of the line. But – **do not place a comment immediately before the period terminating a function.** In particular, one cannot have a function consist only of comments. A comment is not a statement, so do not place a semicolon after a comment.

One can also have comments of a different sort placed anywhere in an input file (*between* function definitions, for example) for documentation. Start such a line with a semicolon, such as:

; This is the factorial function.
$W(n) = if\ n < 2\ then\ \ldots$

Having seen a semicolon, the interpreter simply skips over such a line when reading the input file. Therefore there is no way *within the Fermat session* to read these comments.

More on Arrays as Parameters

Scalar parameters are evaluated at the time of call and their values put on the environment stack. This is "call by value". But array parameters are not passed that way – it would be too wasteful of both time and space. Instead arrays are passed "by reference". This implies, first of all, that the argument must be simply an array name, not an array expression, and secondly, that any change made by the function to its parameter will survive the termination of the function; i.e., it will really change the argument.

A subtle problem sometimes arises in passing arrays as parameters to functions. If the function is changing an array, sometimes the *size* of the output array is not known before the function call. For example, a function *Add* could be written to add two polynomials represented as arrays of coefficients. The size of the answer could be much smaller than that of either of the input arguments because of terms adding to zero. The caller wants to write $Add([x],[y],[z])$, meaning $[z]$ is to be set to the sum of $[x]$ and $[y]$. The $[z]$ existing at that moment may be of inappropriate size, yet the calling function must be written assuming the name of the sum is $[z]$. A solution is to have the function *Add* create the array $[z]$, and have the caller know this. But then the problem is what to do about repeated calls to *Add* (in a loop, say), especially when this $[z]$ is later passed as one of the inputs to a different invocation of *Add*. To provide an efficient way out of this, Fermat has a *change of array name* feature. The command $Rname[c] :=' [z]'$ will change the name of array $[c]$ to $[z]$. The solution to the above problem is to have the function *Add* create an array $[c]$ to store the sum. Then the caller of *Add* changes the name to $[z]$.

You cannot cancel an array parameter. (There is one arcane exception to this – if you somehow find yourself in the lowest command level with array parameters still on the list of currently active names, you can and should delete them. This situation is barely possible.) If you cancel within a function the argument that an array parameter is matched with, the argument will indeed be cancelled, but in the course of bookkeeping its symbol table, Fermat will notice that the array parameter is now referring to nothing. An error will be generated. If you had not set the command &e, everything will be fine after the error. But if you had set &e, then in the new level of the command language you must do a panic stop &@ immediately, else Fermat may crash.

You cannot change the name of an array parameter. You *can* change the name of some other array to that of an array parameter. This is a bizarre thing to do, especially since at the end of the function's execution, Fermat pops off all the array parameter names, so access to the genuine arrays's name is lost.

As one can see from the above paragraphs, array name changing and array cancelling must be done carefully, especially inside functions. For these reasons, I do not recommend using the purge-array command inside a function.

The System Function

Just as there is a system variable (<) and a system array([<]), there is a system function. One frequently wishes to write a loop to compute something, for example,

$$for\ i = -4n, 4n\ do\ Myproc(i/n, n/i)\ od$$

One could certainly give this a name and make it the body of an ordinary function, but often one would like to not have to bother with that, hence the system function concept, whereby one can simply enter the loop on the terminal. As soon as the line is entered, the loop is executed. Furthermore, the function has been stored and can be executed again by entering the name of the system function, <F.

The system function can consist of only one loop or if-statement.

No function can call <F.

Arithmetic Modes (Ground Rings)

Two arithmetic modes are possible in Fermat, *rational arithmetic* and *modular*. (In other words, the ground ring is either the integers $\boldsymbol{Z}$, $\boldsymbol{Z}/n$, or a finite field GF(2^n) for $n = 8, 16$.)

A session with Fermat starts in rational mode. All arithmetic is that of rational numbers: to add you get a lowest common denominator, etc.

Changing Modes – Ground Rings

To leave rational mode and enter modular mode, the command is &p. The interpreter will display the message "Changing arithmetic mode" and will wait for the user to enter an integer n which will become the modulus. n must be at least 2 and no more than $2^{31} - 1 = 2147483629$ (Linux, Unix, OSX) or 92000099 (Windows). Arithmetic will be slightly faster if n is no more than $2^{16} - 1 = 65535$. The current values on the environment stack will be converted as follows: for the rational number $x = r/s$, compute r mod n, then s mod n, then divide the two by inverting s. If s is not invertible mod n, x will be set to 0 and a warning printed.

If you enter 256 or 65536, the ground ring will become $GF(2^8)$ or $GF(2^{16})$. See Appendix Five.

To change back to rational mode, just enter &p again. You can change back and forth as often as you wish.

If you enter a constant from the keyboard in modular mode, it will be reduced modulo the modulus.

It is possible to temporarily turn off modular arithmetic in modular mode. This is useful for the following reason. Suppose you want a loop in a function to execute m times and you write

$$for\ i = 1, m\ do\ \ldots$$

Suppose m should be 2 more than the size of some array $[y]$, so you write $m := Deg[y] + 2$. But the addition will be done modulo n and may well give a result that is too small. To overcome this difficulty, enter the command &_m , then assign m, then switch back by again entering &_m. While the loop is running, if you refer to i in a Fermat expression, it will return its value mod n. Meanwhile, m really contains a technically illegal value – use it in a computation only with care.

The variant form &_m(< *expression* >), or just _(...), turns modular mode off, computes the expression, then restores modular mode.

Never do modular arithmetic, especially multiplication, with outsized values that have been created with &_m. (They *may* be used in arithmetic that is enclosed by &_m.) At the very least garbage will be introduced. The machine may crash. When modular mode is in effect, use these values only for special purposes, such as to control loops.

However, it is permissible and sometimes convenient to set a variable equal to the modulus, such as $mod := \&\_m(\ldots)$. Constructions of the form $x|mod$ are legal in modular mode.

Don't make the mistake of inadvertently turning modular mode on when you thought you were turning it off. For example, if you are in modular mode with modulus 7 and wish to access entry 7 in array x, the well intentioned command $x[\&\_m(7)]$ will fail, because Fermat *automatically* turns modular mode off when it computes array coordinates. You inadvertently turned it back on.

A message of acknowledgement will be printed if you execute &_m from the keyboard, but not if it is executed inside a function.

Disabling modular mode in this way creates a small problem for saving and loading. If a variable is saved with a value of, say, -5, and the file later read during modular mode, the -5 will not be loaded, rather it will be reduced modulo the modulus. However, modular arithmetic may be disabled during polynomial read-in, so a polynomial with negative or outsized coefficients will be read as is. See below "polynomial read-in" and &n.

Mode Conversion Automatically Stored

If you are in modular mode when you save to output, the command

&(p = <*modulus*>)

with the correct number will be inserted in the file. Thus, in a later session, when you read that file, conversion to modular mode will be done automatically. Of course, if you wish, you may insert this command yourself in an input file. The syntax is &(p = <*modulus*>) . When the file is read, the expression <*modulus*> will be evaluated.

Fast, Selective Mode Conversion

This is one of the "Dangerous Commands." See that later chapter.

Polynomials

Having chosen an arithmetic mode, the user may change the ground field (or ring) by converting it into a *polynomial ring*. This is done by adjoining variables – as many as desired – that remain unevaluated. The names of these variables follow the usual rules of variable names in Fermat. To adjoin the variable "t", enter the *adjoin polynomial* command, &J. You will be prompted for the name of the variable. Variables added later have higher precedence than those earlier. Among other things, this means that if you adjoin t and then u, the polynomial $u * t$ is displayed as $(t)u$, not $(u)t$.

To place this command in an input file, use the imperative form &(J = t). If you have attached this polynomial variable and later enter the save command, the command &(J = t) will be automatically included in the saved file. Repeat for other variables. A variable name can be read from an array, just as the name of a file can be read from an array during read and save commands. This allows the easy creation of many names. For example, to create variables *x1, x2, x3, ..., x9*, set an array, say $[x]$ equal to 'x ' (note the blank). Then write a loop *for i = 1,9 do* $x[2] := i + 48$; &(J = $[x]$) *od*. (The rather puzzling "48" converts integers to ASCII.)

The polynomial variable t can be dropped or cancelled by entering &J , followed by $-t$. A name once chosen for a polynomial variable cannot be used for any other purpose. If a variable previously existed called "t", it will be inaccessible.

Using &J adjoins new variables "above" the previous ones. However, as of January 2009, it is possible to adjoin a polynomial variable "at the bottom." So if, say, x and y exist, y later or "above" x, one can do &(J>z), which will insert z as the lowest variable (below x) rather than the highest. You cannot cancel from the botttom.

Exponents must be at least 0, unless the Laurent option has been chosen – see that chapter for more information. In any case, they must have absolute value less than 2^{28}. (Technically, an exponent can be up to $2^{31} - 1$, but Fermat will not let you directly assign an exponent larger than $2^{28} - 1$.)

Producing an exponent larger than 2^{31} by multiplication or exponentiation will probably not result in an error message. But later results will be garbage!

Letting F denote the ground ring, suppose variables have been adjoined to create the polynomial ring $F[t, u, v, \ldots]$. Arithmetic is done in the obvious way, with the ordinary signs $+, -, *,$ ^ . The div operator \ and the mod operator | work about as one would expect, at least if there is only one variable. Division, using / as in x/y, reduces the quotient by dividing out the greatest common divisor of the two polynomials. The result may be a polynomial, if y divides evenly into x, or, more likely, will be an element of the quotient field $F(t, u, v, \ldots)$. Such elements are called *quolynomials*, and are written as fractions or quotients of polynomials. This topic is discussed in greater detail in the following chapter. There is one mathematical complication. In modular mode, F might not be an integral domain. Then neither is $F[t, u, v, \ldots]$. Therefore, there is no quotient field, and writing quotients of polynomials is a bit of a sham. Fermat basically ignores this. When division is indicated, the same g.c.d. algorithms are invoked as in the field F case. Fermat leaves it to the user to make sure that the results computed have any meaning. Even more trouble occurs when polymods are used with division. See the later chapter on "Polymods."

The operator "div", \ is essentially obvious when applied to single variable polynomials, but not to multivariable ones. For in that case, there is no division, there is only pseudo-division. (See any text on abstract algebra, or Knuth volume 2.) Fermat will yield the result of pseudo-division in this case. Similarly, "mod", |, returns the pseudo-remainder. Ideally, $x\backslash y = q$ and $x|y = r$ mean $c^k x = qy + r$, where $c =$ the leading coefficient of y and $k \geq 0$ is an integer. (Fermat adopts the convention that $k = \deg(x) - \deg(y) + 1$.) However, if y is at a lower level than x, the equation $c^k x = qy + r$ will probably fail because y is divided into each coefficient of x and a different k will probably arise in each case.

Polynomial Built-in Functions

Many of the built-in functions cannot be given polynomials of positive degree. Those that can, like greatest integer, work upon all the coefficients.

There is a special operator for *polynomial evaluation*, denoted #. Suppose first that only one polynomial variable, say t, exists. Then $x \# y$ replaces every occurrence of t in x with y. For example, if $x = t^3 + 3t^2 + 3t + 1$, then $x \# 2 = 27$. y can be any expression evaluating to a polynomial.

If several polynomial variables exist, then $x \# y$ replaces the top-most polynomial variable in x with y, i.e., the latest created. To replace one of the other variables, say u, use the alternate form $x \# (u = y)$. The name of any polynomial variable may be used instead of u, and any expression may be used for y.

A fast shortcut form of evaluation called *total evaluation* exists. To evaluate x at every variable, use the syntax $x \# (v_1, v_2, \ldots)$, where all the v_i are numbers. There must be a number corresponding to each polynomial variable, in the precedence order – highest precedence (last attached) listed first. The following more general syntax is also allowed. Suppose there are five poly vars, e, d, c, b, a in that order (e last and highest). Then $q \# (d = w, x, y)$ will replace each d in q with w, each c with x, each b with y. e and a are untouched. Further, w, x, and y can be arbitrary quolynomials. Similarly if $[t]$ is an array, $q \# (d = [t])$ replaces the variables from d on down with the entries of $[t]$ in column major order until $[t]$ is used up. It is an error if $[t]$ has too many entries.

Fermat has a built-in function for polynomial coefficients. Syntax of use is *Coef*(x, n), or *Coef*$(x, n_1, n_2, \ldots)$ if several polynomial variables have been adjoined. x can be any expression. The n_i must be numbers. In modular mode, modular arithmetic is ignored while the n_i are evaluated.

There is a built-in function to compute the degree of a polynomial, using the symbol *Deg*. Syntax: *Deg* x (note space) or *Deg*(x, i). In the second form, the i names the i^{th} polynomial variable. *Deg*(x) returns the largest exponent within x of the specified polynomial variable. If no specification is given, the highest precedence variable is assumed. In modular mode, it returns an actual integer, not reduced modulo the modulus.

Similarly, there is a built-in function to compute the "codegree" of a polynomial, using *Codeg*. It returns the smallest exponent of the specified polynomial variable. Syntax as with *Deg*.

There is a built-in function to compute the leading term of a polynomial, using the *Lterm*. Follow it with the expression whose leading term is to be computed. By definition, the leading term of a polynomial is $coef * u^n$ where u is the variable of highest precedence, n is the highest degree, and $coef$ is the coefficient of u^n.

Fermat has a built-in function to compute the derivative of a polynomial, using the symbol ". Precede it with the expression whose derivative is to be computed, such as x". Differentiation is with respect to the variable of highest precedence.

There is a function *Remquot* to return the (pseudo)remainder (r, say) of dividing d into x and the (pseudo)quotient, q. $c^k x = qd + r$, where c is the leading coefficient of d and $k = deg(x) - deg(d) + 1$ (unless d is a number or c is invertible; then $k = 0$).

There are functions *Factor*, *Sqfree*, and *Irred* to factor one variable polynomials and test for irreducibility. These are described in detail below.

See the earlier chapter on built-in functions for descriptions of *Level, Height, Raise, Lower, Lcoef, Content*, and *Numcon*.

Polynomial Read-in

Fermat has a feature that facilitates the reading of large polynomial constants, the *polynomial read-in*, &_n. It is used by simply putting &_n in front of a polynomial constant that is written out fully following Fermat's conventions.

Example:

$$y := \&\_n\ (5t^2 + 2t + 1)u^2 + (3t - 1)u + 7t^2 - 3t + 1$$

where the two polynomial variables t and u have been attached in that order (u has higher precedence, but there could be polynomial variables between them).

The point of this feature is speed. If the &_n is omitted in the above example, Fermat will square t, multiply by 5, multiply 2 times t, add to the previous result $5t^2$, then add 1, then square u, then multiply by the previous result, etc. – in other words, it will evaluate the expression "normally." But since the expression is written in the canonical order, y can be pieced together without any calls to addition or multiplication at all. The &_n signals Fermat to expect the canonical order and simply "attach" the terms as they come to y. This provides a very large saving of time, which is very noticeable even for polynomials of moderate size.

The canonical order is that of decreasing exponents, no zero exponents, coefficients written first in each term, and nesting according to the precedence of the polynomial variables. There must be no spurious parentheses, no multiplication signs $*$, no division /, no extra plus signs, like $+7t - 3$, and no 0's (as stand-alone constants). There must be no variables or any other built-in functions. Here is another example:

$s := \&\_n\ v^3 + (3u - 2t + 3)v^2 + (3u^2 + (-4t + 6)u + 4t^2 - 4t + 6)v + u^3 + (-2t + 3)u^2 + (4t^2 + 4t + 6)u - 7t^3 + 4t^2 - 4t - 2$

If you are unsure of some of the fine points of canonical form, create some polynomials and watch how Fermat displays them.

During polynomial read-in in modular mode, one sometimes wants modular arithmetic disabled. One may want the read command to load correctly (i.e, as is) data that was created when modular mode was off. Note, however, that non-polynomial data of that type will not be read "correctly." On the other hand one may indeed want the coefficients to be modded out. The toggle switch &n allows the user to choose.

You can pretty much ignore polynomial read-in, especially if you use only one polynomial variable and have no polynomials of more than 15 or 20 terms. However, Fermat will often insert it in saved files and on the screen when it dumps a list of variables, so you must at least be aware of its existence.

Factoring Polynomials

Fermat allows the factoring of one or two variable polynomials over any finite field. The finite field is created by simply being in modular mode over a prime modulus, being in $GF(2^8)$ or $GF(2^{16})$, or by additionally modding out by irreducible polynomials to form a more complex finite field, as described in the later section "Polymods." Factoring into irreducibles or square-free polynomials is possible, or polynomials can be checked for irreducibility. Square-free factoring applies to more general ground rings, not just finite fields.

Pre 2023: For factoring, the syntax is $Factor(< poly >, [x])$ or $Factor(< poly >, [x], < level >)$. The factors of $< poly >$ will be deposited into an array $[x]$ having two columns and as many rows as necessary. (The number of factors (rows) is returned as the value of $Factor$.) In each row, the first entry is an irreducible polynomial $p(t)$ and the second is the largest exponent e such that $p(t)^e$ divides $< poly >$.

Post 2023: The above paragraph applies to one-variable polynomials. For two-variable, often only one factor is returned. Factor doesn't work for more than two.

In the second form, $< level >$ specifies the subfield to factor over. For example, suppose over the prime $p = 30203$ two polynomial variables t and u have been attached, and the irreducible polynomial $t^4+t^3+t^2+t+1$ has been modded out to form the field of order p^4. Then $Factor(u^{12}+1, [x], 0)$ returns six quadratic irreducible factors – irreducible over F_p. So setting *level = 0* suppresses the existence of the field of order p^4. $Factor(u^{12}+1, [x], 1)$ produces a factorization over the ground ring F_p plus the first modded-out polynomial, i.e. the field of order p^4. The result is twelve irreducible factors involving t. Note that the same twelve factors could be attained *much more rapidly* by factoring each of the six quadratics obtained at level 0. Similarly, if now a further field extension is created by modding out on u, the most efficient way to factor is level-by-level, to the extent possible. *It is therefore best for factoring to use as many variables t, u, ... as possible in creating the field.*

Sqfree is like *Factor* except it produces factors that are square-free only, and can be done over more general ground rings.

Pre 2023: *Irred* tells if its argument is irreducible, and, if not, describes the factorization. Syntax is $Irred(< poly >)$ or $Irred(< poly >, < level >)$ ("level" is explained above). The value returned is as follows:

-1 means can't decide (too many variables, for instance).

0 means the argument is a number or is a field element.

1 means irreducible.

$n > 1$ means the argument is the product of n distinct irreducibles of the same degree (and is therefore not irreducible).

x, a poly, means x is a factor of the argument (which is therefore not irreducible).

Fermat uses the algorithms of Cohen, "A Course in Computational Number Theory," Springer Verlag, 1993, pages 123-130. There is some randomness in these algorithms, so

the time it takes to factor a given example can vary.

Post 2023: *Irred* applies to any polynomial over any ground ring. It uses the idea of von zur Gathen to use linear substitutions to reduce to a two-variable poly over a finite field.

0 means the argument is a number or is a field element.

1 means irreducible.

x, a poly, means not irreducible. x is probably not a factor (it will be for a one variable poly), rather it is a factor of the reduced two-variable poly. Its form may be of use to indicate the form of an actual factor.

Irred sometimes has trouble with difficult cases over Z, such as $Irred(x^4 + y^4)$. It may report reducible when it is not!

There is now a file of functions called factr available on the Fermat website that implements Wans method of factoring multivariable polys. See D. Wan, Factoring multivariate polynomials over large finite fields. Mathematics of Computation volume 54. number 190, April 1990, pages 755-770.

The function FCTZ in that file returns -1 when given $x^4 + y^4$. Technically -1 means can't decide, but it usually, in fact, means irreducible.

Quolynomials

Letting F denote the ground ring, suppose variables have been adjoined to create the polynomial ring $F[t, u, v, \ldots]$. Division, using / as in x/y, reduces the quotient by dividing out the greatest common divisor of the two polynomials. The result may be a polynomial, if y divides evenly into x, or, more likely, will be an element of the quotient field $F(t, u, v, \ldots)$. Such elements are called *quolynomials*, and are written as fractions or quotients of polynomials. Fermat automatically creates quolynomials whenever it is necessary.

Basic arithmetic of quolynomials is essentially obvious to anyone who remembers high school algebra.

There are three mathematical complications. Over a modular ring Z/n ("modular mode"), F might not be an integral domain. Then neither is $F[t, u, v, \ldots]$. Therefore, there is no quotient field, and writing quotients of polynomials is a sham. Fermat basically ignores this. Certainly these polynomials can be added, subtracted, and multiplied with no problem. When division is indicated, there may or may not be an immediate problem. For example, $1/(2t + 2)$ will cause an error message if $F = \boldsymbol{Z}_4$, because 2 is not invertible in F. If some computation f/g does not cause this problem, the same g.c.d. algorithms are invoked as in the field F case. Fermat leaves it to the user to make sure that the results computed have any meaning.

Secondly, a similar problem arises when polynomials $p, q, r, \ldots$ are used to mod out and form the quotient ring $F[t, u, v, \ldots] / < p, q, r, \ldots >$ of *polymods*. If the polynomials $p, q, r, \ldots$ are not all irreducible, then this ring is not a field, and dividing and forming quolynomials is a sham. Fermat will not stop you from trying to do this, and there is no guarantee of the results. In particular, there will no longer be *canonical forms*, i.e., different looking expressions may in fact be equal. If they are subtracted, the result may or may not be an honest zero. The following chapter describes polymods.

Div and mod applied to quolynomials act only on the numerators.

There are built-in functions *Numer* and *Denom* that return the numerator and denominator of a quoly. The built-in function degree *Deg* returns the difference between the degrees of the numerator and denominator. Leading coefficient *Coef* ignores any denominator. Leading term *Lterm* produces an error if called on a non-polynomial.

Polymods

Choosing an arithmetic mode establishes the ground ring F. On top of this may be attached any number of unevaluated variables $t, u, \ldots$, thereby creating the polynomial ring $F[t, u, \ldots]$ and its quotient field, the field of rational functions or quolynomials. Further, certain polynomials $p, q, \ldots$ can be chosen to mod out with, creating the quotient ring $F[t, u, \ldots] / < p, q, \ldots >$, whose elements are called *polymods*, and, implicitly, *quolymods*, which are expressed as quotients of polynomials, superficially the same as ordinary quolynomials. If the polynomial p involves the variable t, we say that t has a *relation* imposed on it.

For example, in rational mode, $F[t] / < t^2 + 1 >$ is the mathematical ring $\boldsymbol{Z}(i)$, with t playing the role of $\sqrt{-1}$.

The basic command to mod out by a polynomial is &P. You will be prompted for the quotient polynomial, which must be monic, say $t^n + c_1 t^{n-1} + \ldots$. The imperative form of this command is &(P $= t^n + c_1 t^{n-1} + \ldots$).

Ideally this command should be given in rational mode (i.e., over Z). It is OK to give it in modular mode, but then one cannot change to rational.

In principle, any monic polynomial may be modded out, say $t^n + c_1 t^{n-1} + \ldots$. However, Fermat is best at the case where the quotient ring becomes a field (note well, $\boldsymbol{Q}[t, u, \ldots] / < p, q, \ldots >$ is a field, $\boldsymbol{Z}[t, u, \ldots] / < p, q, \ldots >$ is not). Specifically, suppose the polynomial variables have been attached in the temporal sequence $t, u, v, \ldots$. Begin by modding out a monic irreducible polynomial $p(t)$ such that $F_1 = \boldsymbol{Z}[t]/ < p >$ is an integral domain, and its field of fractions is $\boldsymbol{Q}[t]/ < p >$. Then, if desired, mod out by a monic polynomial $q(u, t)$ such that $F_2 = F_1[u]/ < q >$ is an integral domain, and continue in this manner always creating an integral domain, and, by the same stroke, its field of fractions.

You must tell Fermat that a field will result, and it is your responsibility to check this. Do this at each step by adding a comma and 1, as &(P $= t^n + c_1 t^{n-1} + \ldots$, 1). In Fermat, you may append a list of primes q such that the modder $p(t)$ remains irreducible mod q. For example, &(P $= t^3 + t^2 + t + 2, 1 : 151, 167, 191, 839, 859, 863, 907, 911, 991$). Add as many as you like, at least 30 is desirable. Or, probably better, omit the list and Fermat will compute it for you. (If you save the session to a file, Fermat will include the list in the saved file and just reload it next time.) Fermat then computes a second list of auxiliary primes of a different sort: modulo these primes the modding polynomial has a root. Both types of primes are used to speed up g.c.d. computations. If Fermat cannot find enough of either type, it will instruct you how to get more, using the commands &A and &B.

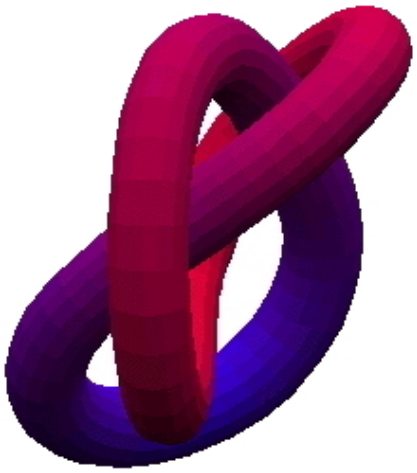
When you have implemented a field in the above manner, division is done correctly. For example, with the above $p(t)$, $1/t^2$ is computed as $(t^2 - t - 1)/4$, which is correct. If you have not told Fermat that a field results, $1/t^2$ is left as $1/t^2$. Whether this has any meaning is, of course, your problem.

To later rescind modding out on t, enter &P again and then $-t$. To change the modding polynomial to another, you must first rescind the current one.

Note that polynomial evaluation (substitution) of a variable involved in a modding relation is not a homomorphism, and may not make sense. This is the user's responsibility.

If a field has been formed in this way, one can drop into modularmode mod any prime such that the modding polynomials remain irreducible. It is the user's responsibility to check this.

To create a poly variable and mod by it all at once do, for example, $\&(J = t, P = t^2 + 1, 1)$. This is more efficient than two separate commands – indeed it can be much more efficient if a lot of data exists.



Trefoil knot viewed from an interesting angle.

Laurent Polynomials

In Fermat a Laurent polynomial is one with negative exponents. If you wish to allow this, activate the toggle switch &l. All of the variables you have created up to that point will be converted to the new format. For example, $1/(t^2 + 2t)$ will become $t^{-1}/(t + 2)$. No negative exponents are left in the denominator of a quolynomial and all positives are factored out and moved to the numerator.

Unlike polymods, Laurent polynomials present no inherent mathematical difficulty. Be aware that in Laurent mode, the function GCD treats variables t as invertibles, and always presents answers with no negative exponents.

Laurent mode should be used if you expect a fair amount of results to have denominators that a variable could be factored out of. If a matrix has several entries such as $1/t$, use this mode to compute its determinant or inverse.

Laurent polynomials implement computations in the group ring $\boldsymbol{Z}(\boldsymbol{Z})$.

Character Strings

Fermat allows the assignment of characters to scalar variables and *character strings* or *literals* to arrays. In the first case the syntax is $y := \ 'a'$ to assign the character $'a'$ to the variable y. In the second case the syntax is $[x] := \ 'sample'$. (Use the single quote, or apostrophe.)

What actually happens is that y gets the ASCII code of $'a'$. It is still therefore "really" a number. If one simply types y or $'a'$, the ASCII code is displayed. (This works in all arithmetic modes. In modular mode the result will not be reduced modulo the modulus.) To see the character 'a' instead, use the display command $!(y : char)$. Note the reserved word *char*.

Strings are more useful than characters, for which one needs arrays. If the user first creates an array $[x]$ of size, say, six, and then enters the command $[x] := \ 'sample'$, each character is stored in the six slots of $[x]$. Entering $![x$ will display the ASCII codes of the six letters. To see the letters, one needs the *char* attribute, the syntax of which is $![x : char$. (Note that this is array short form.) In the above example, this will produce the original string *sample*. A spacing attribute can be added, as in $![x : char : n$, which places each character in the left of a field n units wide. Compare the display command form, $!([x :$ *char*). One can also use array long form $![x]:$ *char*, which produces one character per line, or $!([x]:$ *char*). Try them both.

Array features discussed in earlier chapters enable one to work with substrings, concatenate, find the length of a string, etc. For example, to concatenate a string stored in $[x]$ with a string stored in $[y]$, enter $[z] := [x] \_ [y]$. (This could also be executed more cumbersomely with subarrays.)

Character strings can easily be read into an array. The command is $?[a] : char$, which will display on the next line $> [a] \ := \ '$. Note the prompt $>$ and single quote. The user then enters any character string followed by the single quote and hits return. Then array $[a]$ gets the ASCII codes for the characters.

There are various fine points, which are here explained:

(1) If the array $[x]$ has some undefined elements, they will appear as $'* \ '$ upon execution of $![x : char$. Note the space after the $*$.

(2) If $[x]$ has not yet been created, $[x] := \ 'sample'$ creates a 1×6 array $x[1, 6]$, not an array $x[6]$. Predefining $[x]$ as one-dimensional $x[6]$ solves the problem (see "array assignments" above).

(3) The command to display as a character string can be given to any array. Values which do not correspond to legal ASCII codes will result in the printing of a blank or of a little square box.

(4) These two are equivalent: $[x] := \ 'abc'$ and $[x] := [(\ 'a',\ 'b',\ 'c')]$.

(5) When typing from the terminal, the continuation character ‘ should not be in a literal. For comparison, enter these two statements:

$[x] := \ 'abc$

def'

and

$[y] := \ 'abc$
$def'.$

(Both will have to be selected with the mouse and then entered.) Then display each as a character string.

(6) Displaying a two-dimensional array $x[n, m]$ as a character string produces n rows of length m character strings.

(7) Constants like $'e'$ have a somewhat ambiguous existence in Fermat – are they scalars or arrays? If you simply enter $'e'$, the ASCII code for the letter e will be displayed, so it seems to be a scalar. If you enter $[x] := \ 'e'$ you will get a 1×1 array. If you enter $[w] := \ 'e' \_ \ [x]$. where $[x]$ is any array, the $'e'$ will be treated as a scalar, so $[w]$ will be a one dimensional array with first character e.

More Built-in Functions

Divides is implemented in Fermat. Syntax is *Divides*(n, m) or *Divides*(n, m, q). Does n divide evenly into m? n and m may not have denominators. As always, 0 means false, 1 means true. If q is present, the quotient will be returned if *Divides* is true. Various time-saving ideas based on homomorphisms are used.

PDivides(n, m) also asks if n divides evenly into m. The strategy is to substitute each polynomial variable except the highest with a constant. *PDivides* says true iff these reduced polynomials divide evenly. The constants are chosen with some care. Nonetheless, this is a probabalistic algorithm. An answer of False is always correct, but an answer of True has an infinitesimal probability of being wrong. *SDivide*(n, m) is similar but faster; see Appendix 4.

Irred, Factor, and *Sqfree* are available and are described in the Polynomial chapter and in Appendix 4.

Powermod(x, n, m) computes x^n *mod* m. x must be a polynomial or integer, n must be a positive integer, and m must be a monic polynomial or positive integer. You may omit the third argument if you are in modular mode or polymodding. Note that n often needs to be very large. In modular mode, this is a problem. The solution is that n must be either a constant or must involve only variables that have been created in rational mode while under "Selective Mode Conversion" – see that topic under "The Dangerous Commands." *Otherwise Fermat will crash.*

Adjoint$[x]$ is the adjoint of matrix $[x]$.

PRoot(x) returns the p^{th} root of x, when $x \in$ the ground ring, a field of characteristic p.

Totdeg$(x, [a], n)$ returns the largest and smallest total degrees of the monomials in x . See Appendix Four.

WDeg$(x, [a], n)$ "withdraws" the subpolynomial of x of total degree n in the variables listed by $[a]$. See Appendix Four.

In Fermat $GCD(x, y)$ for polynomials is sophisticated. Hensel's Lemma and the Chinese Remainder Theorem are used for $GCD(x, y)$. These techniques provide orders of magnitude improvements in speed when x and y are multivariate.

If x is a rational function ("quolynomial") and y a polynomial, then $x|y$ will mod out the numerator and denominator of x by y, then form the quotient. This may not be reasonable mathematically – that's the user's problem.

Rat(x) returns 1 if the expression x is rational but not integral, otherwise it returns 0. $LCM([x])$ computes the least common multiple of all the denominators in matrix $[x]$. "Denominators" means those of rational numbers or of expressions like $(t^2+3t+1)/17$ or $3/(2t)$. The denominator of $2/(3+2t)$ is ignored, since you can't clear it by multiplying $[x]$ by any number (except 0!).

Fermat has an interpreter command &H, with imperative form &(H = ...) to affect the display of constants in modular mode. When turned on, modular numbers will be displayed in the form $-m, -m+1, \ldots, 0, 1, \ldots, n$ where $m =$ (modulus-1) div 2 and $n =$ modulus div 2.

Deriv(x, t, n) returns the n^{th} derivative of x with respect to t. x is any (scalar) expression, t is a polynomial variable, and $n \geq 0$.

In Fermat, with the degree-of-polynomial function *Deg* there is a difference between *Deg* x and *Deg*(x). The second form (where x can be any expression) is slower because it will spend time duplicating x before it computes the degree. In the first form, x must be a variable name.

The Characteristic Polynomial

Chpoly computes the characteristic polynomial of a square matrix, in terms of the polynomial variable of highest precedence. The syntax of the command and the method used depends on whether the matrix is sparse or "ordinary."

With the ordinary matrix storage scheme, LaGrange interpolation is used when the matrix consists of all numbers. It is to your advantage to clear the matrix of all numerical denominators before invoking *Chpoly*. Use *LCM* to do this. When using the LaGrange interpolation method on integer matrices, Fermat computes the many necessary determinants using the Chinese Remainder Theorem. To do so, it must make an initial estimate of the absolute value of the determinant. The estimate is often rather liberal, resulting in longer than necessary computation time. Now, the determinants in question are simply $f(c_i)$, where f is the characteristic polynomial and $\{c_i\}$ is a set of certain "sample points." You, the user, may be able to supply a far better bound on $|f(c_i)|$. For example, you may have some estimate of the location of the roots of f. For this reason, there is a second optional argument to *Chpoly* in Fermat, a polynomial g such that $|f(t)| \leq |g(t)|$ for all t. The syntax is *Chpoly*$([x], g)$.

With sparse matrices, a clever way to compute characteristic polynomial is the Leverrier-Faddeev method. (See example 4 in Appendix 3 for a short Fermat program that implements the algorithm.) This method often loses to the standard one, $det([x] - \lambda I)$, but it can be faster for matrices that contain quolynomials. The user may choose the method with the second argument. If $n = 0$, *Chpoly*$([x], n)$ will simply subtract the poly var from the diagonal and invoke determinant. The determinant method will depend on choices made for &K, &L, etc. as described elsewhere. If $n \neq 0$ the Leverrier-Faddeev method is used.

The "modifed Mills method" is a stunningly fast probabalistic "black box" or Wiedeman algorithm that computes the minimal polynomial $M(t)$ of a sparse matrix of integers, or, more precisely, a factor of the minimal polynomial. (See Continued Fractions and Linear Recurrences; W. H. Mills. Mathematics of Computation, Vol. 29, No. 129 (1975), pp. 173-180.) If one of the roots of $M(t)$ is 0, the associated factor t of $M(t)$ will not show up, but other factors may not show up either. This algorithm is built into Fermat via the command *Minpoly*. Syntax is *Minpoly*$([a]$, *level*, *bound*$)$. $[a]$ is the matrix, which must be *Sparse*. *level* $= 0, 1, 2, 3, 4$ is a switch to tell *Minpoly* how much effort to expend in its basic strategy. Larger levels take longer, but have a better chance of giving the entire minimal polynomial. *bound* is an integer at least as big as any coefficient in the minimal polynomial. This argument can be omitted, in which case Fermat will estimate it with the well-known Gershgorn's Theorem.

Repeated calls to *Minpoly* may return slightly different answers. It may be worthwhile to run it several times and compute the l. c. m. of the answers.

Minpoly is available in rational or modular mode.

The Dangerous Commands

This chapter describes several commands that should not be used by novices nor the timid. Each one can produce impressive speedups in execution time. Each one, if misapplied, can crash the system, or worse, produce reasonable but erroneous output.

Nothing is free. The cost of speed is loss of flexibility, loss of error checking, and a certain amount of fragility.

Integer

Integer is a directive to Fermat that promises that every scalar quantity encountered during the execution of a function will be a number, *not a polynomial*, and more, not merely a number but an integer of absolute value less than 2^{28}. Beware therefore of overflow, especially when multiplying! Beware of computing something like $Bin(50, 20)$, which is larger than 2^{28}!

Fermat does not check to see that quantities encountered really are (small) numbers; if one is not a number, the system may crash. Nor is overflow checked for.

Setting *Integer* in function F has no effect on a function G that F may invoke. If F calls itself recursively, the later invocations will not automatically inherit *Integer* from the earlier. For it to hold in any invocation, the command *Integer* must be executed in that invocation.

With *Integer*, functions MUST use *Return* to pass back a value.

To further enhance speed, while *Integer* is in effect some range checking is disabled. If you access an array via $x[20]$ and $[x]$ has less than 20 elements, the ordinary error message will not be generated. What actually will happen is completely unpredictable.

Integer will not catch the error of treating an array parameter as if it were a scalar parameter.

Implicit multiplication doesn't work under *Integer*. You must write $2 * x$, not $2x$.

As of 2010, a new arithmetic command has been added for when a function is in Integer mode: $x :* y$ to set $x := x * y$; NB: this only works in Integer.

Ideally, one should modularize a computation so that segments that can take advantage of this feature are moved into their own functions. Very impressive speedups are possible.

Compile

Like *Integer*, *Compile* is a directive to Fermat, and applies only to the function in which it is located. *Compile* implements one of the many functionalities of a genuine compiler. It is a promise to Fermat that the variables given to *Compile* will not change their identity throughout *all* the executions of the function. Note carefully: the *identity* must not change; it is perfectly fine to change the *value* of the variable. Essentially, "identity" here means "address" or "access path."

To use this feature, the command *Compile* must be the very first command in the function. (If *Integer* is to be used also, *Integer* should be the second command.)

For example, the command $Compile(x, [y], p[2], z)$ promises that x, $[y]$, $p[2]$, and z will always refer to the *same* variables, everywhere within the function and in every call of the function. The first time the function is executed, when *Compile* is seen, Fermat looks up

the names x, $[y]$, $p[2]$, and z. It then scans the entire function definition and augments every occurrence of these names with a pointer to the symbol table record of the name *that exists at that instant*. When a reference to one of the variables is encountered during execution, the need to look up the name in the symbol table is therefore obviated. This can provide a very impressive speedup. (Note: there is a speed advantage to compiling $p[2]$, not just $[p]$. But make sure $p[2]$ has been initialized before the *Compile* statement is executed. Unfortunately, two dimensional references like $p[2,3]$ cannot be compiled. Such a reference can be converted to one dimensional form using column-major order.)

Only the programmer can tell for sure that it is safe to compile a certain name, say z. If z is a global variable one time the function is executed, but is some other function's local variable another time, it must not be listed. Furthermore, if z is another function's parameter or local variable, it is not the *same* z during different invocations of that function – recall that the symbol table entry for local variables is removed after invocation ends – and cannot be compiled. **This is the most common mistake in using Compile. It is an insidious error.**

The list of names following *Compile* must under no circumstances contain the parameters and local variables of the function. These are automatically compiled, when *Compile* is specified (but see next paragraph). Rather, the list should be of names created elsewhere, outside the function. If the list (and the parentheses) are omitted, only the parameters and local variables will be compiled.

It is possible not to compile the parameters, by using the syntax *Compile*$*$, i.e., placing a $*$ after the word "Compile" and before the parenthesis. Then only the variables in the following list will be compiled. **Make sure the list does not contain the name of a parameter! The system may crash if it does.** A compiled function cannot be called recursively, unless you use *Compile*$*$. (Indeed, that is why *Compile*$*$ exists as an option.) There will be no error message if you try it. Rather, garbage will be generated or the system will crash. **Beware of this mistake when using Compile!**

A subtlety arises involving compiled array references and the array initial indexing value, 0 or 1, which can be changed by the command &a. If you place $x[5]$ in the list of variables to be compiled, the address of $x[5]$ will be determined according to the array initial indexing value that exists *at the instant of compilation*. If you execute &a within the function before a reference to $x[5]$, you have a problem – at the very least, the function will not execute the way it would without compilation.

This problem does not arise from compiled references of the type $[x]$, as in *Compile*$([x])$. Later executions of &a are irrelevant. Therefore, a cure to the problem of the previous paragraph is to use *Compile*$([x])$ instead of *Compile*$(x[5])$ when &a occurs within the function being compiled. (*Compile*$(x[5])$ is, of course, slightly more efficient.)

Range checking of compiled array references is disabled. If you access an array via $x[20]$ and $[x]$ has less than 20 elements, the ordinary error message will not be generated. What actually will happen is completely unpredictable.

Compile and *Sparse* do not work together: *Sparse* arrays cannot be passed as arguments to a compiled array parameter. Nor can *Sparse* arrays appear in the list of variables after the *Compile* command.

Changes to a variable's identity *before* the compiled function is executed the first time

are irrelevant. *Compile* is an executable statement, and is not noted at function definition time.

There is no point incurring the cost of compilation unless the function will be executed often, or contains a loop that will be executed often. If a function is dominated by polynomial or matrix operations, little relative improvement will result from compilation.

I have seen the use of *Integer*, *Compile*, and the increment command eliminate 90% of the execution time of some functions.

Selective Change of Arithmetic Mode

Many algorithms in number theory compute results by performing subcomputations modulo n, for various n's, and then constructing the answer in the field of rationals. To do this well in Fermat it is necessary to drop in and out of modular mode quickly, converting only certain variables. This selective change of arithmetic mode is implemented, for example, with the syntax &$(p = 19 : (x, [y], z))$ and &(p=_i:$(x, [y], z)$). The first command enters modular mode with modulus 19, converting only $x, [y]$, and z. The second returns to rational mode. Other variables are never touched. Attempting to reference other pre-existing variables while in modular mode could lead to a crash. An assignment such as $w := 2$ performed while in modular mode will be dangerous if w pre-existed. If not, and if w is added to the list when converting back to rational mode, all will be o.k.

In the example above the variables $x, [y]$, and z will not have their initial values when rational mode is reinstated. For example, 21 would be converted to 2, which will be "converted" back to 2; the 21 will be lost unless the user has gone to the trouble of storing it somewhere else. To preserve the initial value of, say, x and z, use the syntax &$(p = 19 : (*x, [y], *z))$ and &(p=_i:$(*x, [y], *z)$). No variable preserved in this fashion should be changed while in modular mode; think of such variables as read-only input to modular mode. To preserve all, use &$(p = 19 : (**))$. Then in returning to rational mode, probably a new variable has been created (otherwise you could preserve results only by writing to a file), say x, so use &(p=_i: (x)). (Note that there need be no $*$ in the command.)

If you enter modular mode selectively, you must leave it selectively – i.e., with a command of the form &(p=_i: (...)) with *something* in the parenthses.

If you enter modular mode selectively, you must be cognizant of the fragility of the situation. While in modular mode, do not change Laurent. Do not adjoin or cancel a polynomial variable. Do not change the polymod situation.

Here is a subtlety with selective change of arithmetic mode. The built-in function *Chpoly* works quickly because certain polynomials are precomputed once and for all, and then used to compute the characteristic polynomial of any matrix given later. The computation of these hidden polynomials takes a fair amount of time, which is why the command to adjoin a polynomial variable, say t, takes longer when it is the first variable adjoined – the polynomials are being computed. When changing arithmetic mode, these polynomials have to be converted to the new format. That takes some time, and would defeat the purpose of quickly dropping in and out of modular mode.

The solution is an interpreter command &B (for *Block*) that sets a switch to block the conversion of these hidden polynomials. When switched on (the default), the polynomials

will not be converted. So if *Chpoly* is called in modular mode, it will be done the straightforward but relatively slow way of subtracting t from the diagonal and computing the determinant of the resulting matrix. &B cannot be executed while in modular mode.

Full Interrupt

Many of Fermat's more sophisticated procedures for g.c.d., factoring, etc., affect system-wide global status variables. For example, the g.c.d. routines in rational mode drop in and out of modular mode. Ordinarily, Fermat does not allow user-interrupt at these times, for the very good reason that the user would be in modular mode, not know it, and may attempt to access some now undefined variable, with bizarre or devastating results. However, some of these procedures can be very time consuming, and the user often wants to interrupt them. The solution is to allow the user to set a special flag that allows interrupts even at these dangerous times. The command is &Z, a simple toggle switch. When switched on, and when &m has been set on, the user can get pretty fast interrupts just about all the time. It is a good idea to immediately do an &g to see if the system globals have changed. And be careful!

Errors and Warnings

When an error occurs (dividing by 0, array index out of bounds, etc.) execution of the function containing the offending statement stops and the interpreter returns to the lowest interactive level and displays a message describing the error. If the error occurred during or because of a command entered from the terminal, the portion of the command successfully parsed is displayed. Then a function trace dump is displayed: the stack of function calls from most recent to earliest. However this feature is not perfect: if the stack of calls is very deep, only the first 500 (i.e, the 500 most recent) are to be believed.

If the error occurred during a function evaluation, the interpreter then displays two statements, the last one successfully executed followed by the expression it was working on when the interrupt occurred, up to the offending character. Usually the offending character is not displayed, but sometimes it is the last thing displayed. Sometimes the first of these statements (the last one successfully executed) is not displayed, usually because the interrupt occurred in the first line of a function. Keep in mind that the two statements may not be physically adjacent in the function definition.

If a great many function calls have been stacked up when an error occurs, then if the interpreter simply returned to the lowest level, all of those functions' parameters would be on the environment stack. The user would find this extremely inconvenient: his "real" variables would be buried. Thus, the interpreter also pops off all the now useless parameters, up to around 500 calls deep.

The command &e can be used to change the way errors are handled in some cases – see the next chapter.

Here is an example of an error that occurred during a function evaluation, taken from a Fermat session. Line numbers [n] added:

```
   >G2(10)
    *** Inappropriate symbol:  ’
[1]  error in function definition.  ‘:’  or ‘;’ expected.
    *** Error occurred at this point:
[2]  G2(10);
    *** The most recent function evaluations were:
[3]  Undo
    Next
    G2
    *** Function was interrupted at:
[4]  n<0;
    varcount[n] ’
```

The user invoked $G2(10)$. In line [1], Fermat reports the basic problem, apparently a quote in place of a colon in some assignment statement. Line [2] reports the offending place in the input line (read from the terminal) that caused the error. Line [3] begins the

trace dump of function evaluation calls. $G2$ called $Next$ which called $Undo$. The error is therefore inside $Undo$. Line [4] and the following are the last two commands that Fermat executed. $n < 0$ was executed, and execution terminated at the end of the next line, where we can see the offending quote.

Warnings

There are a few situations that produce *warning messages*, such as entering the command &> from the console. No error occurs, zero is returned, and normal execution proceeds. The more serious of these are called "Fermat Warnings" and "Fermat Errors", which occur when some internal condition that "must" be true is in fact false. For example, in the middle of certain G.C.D. computations, some polynomials "must" divide into others. If that fails, a Fermat Warning is issued. Usually in a Fermat Error the interpreter pops up to a new level.

Please report all Fermat Warnings and Errors to the author!

A commonly encountered but usually harmless warning message is "end of line found inside a comment." It occurs whenever Fermat reads a function definition and sees a multi-line comment. It is certainly allowable to have multi-line comments. Yet the warning message must be produced in order to catch the error of entirely omitting the end-of-comment symbol } . If you do that, Fermat will be stuck in an infinite loop, and your only notification of that fact will be the above warning message.

Popping, Pushing, Debugging, and Panic Stops

Let's say that a computation seems to be going on too long, or for some other reason you want to stop it. Fermat will break if you hit cntl-c. The interpreter will push on a new command level. You can now examine any of the variables, or do anything else – it's just as good as the original command level (almost – more later). This is a useful debugging feature. To resume the computation (and fall back to the original command level), enter the command &_p ("pop"). To stop the computation completely and fall back to the lowest level (in effect, a panic stop), enter &@.

See "Full Interrupt" under "The Dangerous Commands").

An interrupt subtlety: if in the new command level you delete the function you were executing, when you fall back to that command level, the function will still be able to resume, i.e., it still exists internally until its execution terminates. However, since its name has been erased from the list of functions, no function can call it. Therefore, an attempt at recursion will result in an error.

If you interrupt a Fermat built-in function, do not invoke the SAME built-in at the higher command level.

The commands &_p and &@ can be inserted in a program, as can &_P ("push"), which has the same effect as pressing cntl-c.

Command levels can be stacked up to level 9.

In command levels beyond the first errors are handled differently. When an error occurs, the interpreter tries to recover from it, prints a warning, and continues the computation, with nonguaranteed results. For example, division by 0 causes Fermat to print a warning and use 0 for the result. Obviously, this may spawn further errors. If no reasonable recovery is apparent, the Fermat interpreter is forced to bomb back to the lowest command level. There is then no way to restart the computation at the point that originally created the interrupt.

To somewhat overcome this annoyance, the command &e has been added to Fermat. This sets a flag that changes the way errors are handled during function executions and at higher command levels than the first. *If you have previously executed* &e, when an error occurs, Fermat will push on a new command level and halt. You can then examine the variables and try to rectify the error yourself, and then pop down to the previous level to allow the computation to continue. (You may find that new errors are being spawned. It may take several &_p commands to get you down to the previous level, so keep trying.) Note however that this is still less then perfect. For instance, if the problem was a syntax error in a function, it will do no good to cancel the function and enter a patched up version – upon popping down to continue the previous level, Fermat will still be reading the previous version of the function. Furthermore, if your attempted patch is not good enough, Fermat may crash. It is safer, therefore, to do a panic stop instead of trying to resume the lower level.

See "Full Interrupt" under "The Dangerous Commands."

Initializing with Ferstartup

Fermat has a "ferstartup" file to provide for user-defined initializations. The last thing Fermat does while booting up is to read this file as if it were an input file and execute the statements it finds there. The file can contain any Fermat statements, including read commands, each followed by a semicolon. It should not have &x. Lines can be commented out by starting them with a semicolon.

Besides mere convenience, the initialization file provides the capability of invoking Fermat from other Fermat execution files, since it is possible to direct Fermat without ever typing a command on the keyboard. For example, some program (another Fermat?) could create data for Fermat and store it in a file in Fermat-readable form, then insert the right read command in ferstartup, then invoke Fermat.

For Linux, Unix, and OSX the first line of ferstartup must be the number of megabytes of RAM on your machine. The default is 1000 meaning one gigabtye.

Ferstartup must be at the top level inside the Fermat folder (Windows; inside folder BACKWARD for Linux and OSX, FORWARD for Unix). [The Unix version is obsolete.]

Hints and Observations

These comments resulted from questions by users of Fermat.

• When functions are defined, they are not parsed completely. Only certain key features are looked for, such as comments, loops, and if-statements. Therefore, many syntax errors are not detected until the function is executed.

• The order that the functions are defined in the input file is completely irrelevant, unless you put a statement in the input file that actually invokes one of the functions. Then, of course, that function must have been previously defined, and any function it calls must have been previously defined.

• The order in which most mode-changing commands are given is irrelevant. You may add a polynomial variable, then set the Laurent flag, then change array indexing – in this order or another.

• Fermat is an interactive language, reading and writing to the console, i.e., to the same file. This is somewhat difficult to arrange. Other interactive languages require the user to type a special character, usually a semicolon, at the end of each line. I have used some of those languages and been continually annoyed by the necessity of adding semicolons. Therefore, I arranged it that Fermat does not require them – although they *are* required at the end of statements in an input file. But nothing is free, and hence the necessity for the continuation character ‘ when entering more than one line from the terminal. The only place you may find this annoying is when you are editing a function and you forget to put a ‘ at the end of every line. When you attempt to enter the new amended function, Fermat will detect this omission and print out the offending line.

• Functions can contain anything that can be entered from the terminal [not &F or &v]. In particular, they can contain the definitions of other functions, or even redefine “themselves.”

• Very large numbers (those occupying 300 or more lines) or polynomials take time to display on the screen. For example, if you enter 8000!, more time will be spent computing the digits of this number in base 10 then in actually multiplying it out in the first place. [Note: when I first wrote this manual around 1992, I used the example 790!, which now computes and displays in time 0.000. Progress.]

• The first command that you execute after starting Fermat may take about a tenth of a second longer that it otherwise would. This seems to be due to memory management.

• Suppose you want to invert matrix $[a]$. Typing [a]^-1 will work, but typing $1/[a]$ will not. In the first case, the initial “[” alerts Fermat to expect an array expression.

• Over ground ring Z with several polynomial variables, there is a fast probabalistic test for a polynomial dividing into another. It sometimes fails. The usual ramification of this a “Fermat Error” of some kind. Turn the test off with $\&\_t$.

• Several matrix algorithms, such as determinant, inverse, row reduction, depend on efficiently choosing a pivot element. For sparse matrices there are several ways to do this, controlled by the command $\&u$. I use $\&(u = 5)$.

• Except for small matrices containing constants (fewer than 15 or so rows) all matrices should be declared sparse. The procedures for sparse matrices are more sophisticated.

• If you execute a save command &s, and later the machine crashes for some reason, the material that was written to the file may not survive, because the file was never closed. To overcome this, enter &S and name a new file. The original file will be closed.

However, sometimes material is still lost after a bad crash. You can absolutely save something by interrupting Fermat, displaying the variables you want, grabbing the data with Copy (command-c), and then transferring to some word processor. Open a document there and save the material.

• In reading and saving, the difference between &r and &R, and &s and &S, may be confusing. Use the small letter form either the first time you give a read or save command, or to continue reading or saving to the same file as you previously specified. Use the capital letter form to change to a different file, thereby closing the former one.

In either case, unless you give the imperative form of the command, Fermat will prompt you for the name of the file. This means that if you put &S in the middle of a function, you cannot leave the terminal until the function has gotten to that point in the program; someone must be there to respond to the prompt. The problem is solved by using the imperative form: &(S = <*filename*>). This command can be used whether or not it is the first save. You can simply start up the function and leave it to run overnight.

You may also use a character string stored in an array to name the file, as in &(S = $[x]$). This allows file names to be computed within functions.

If you create a file with the &s or &S command, say *abc*, the name of the file will be just that – *abc* with no *.txt* or other suffix. If you create a text file with some word processor or editor under Windows, the suffix *.txt* will probably be appended, whether you see it or not. To read such a file in a Fermat session, you will have to put the name in quotes and add the *.txt*, as in *'abc.txt'*.

• Deleting Array side effect: For at least fifty years, authors of programming texts have been decrying side effects. A sneaky one possible here comes from deleting an array. Consider the assignment statement $m[i] :=<$ *expression* $>$. Fermat executes this by first checking that array m exists, then computing i, then setting a pointer to the place in memory where $m[i]$ is. Suppose the expression deletes an array. Ordinary arrays are kept in a linear structure. When an ordinary array is deleted, all the arrays farther down in the structure slide up to fill the "empty" spots. That might change where m is! The pointer computed for $m[i]$ might now be incorrect! Watch out! [This is not possible with sparse arrays.]

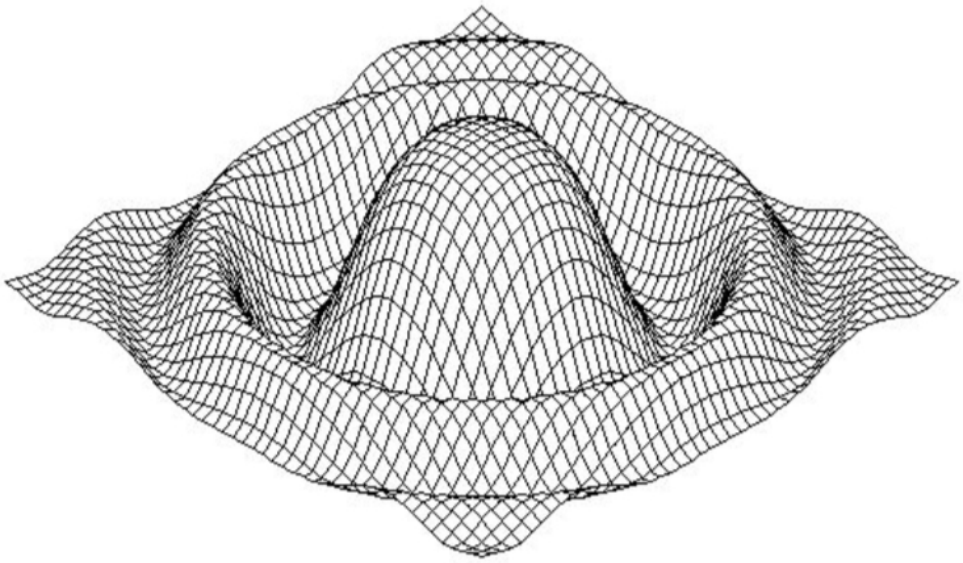
• Fermat has been designed to catch all sorts of errors, but no program can allow for all eventualities. Besides the features described in the chapter "The Dangerous Commands," one thing that will cause Fermat to crash is too much recursion. If a function calls itself several thousand times without terminating, all stack memory will be exhausted.

Another thing that will cause a crash is to execute a change of precision &p after you have pushed to a new command level by pressing cntl-c. Upon resuming the previous computation, Fermat will probably crash. There are also a few strange things you can do with array parameters to cause a crash, and with *Powermod.*

There's an old Henny Youngman joke: A man goes to see his doctor. He lifts his arm up and down in an odd fashion, saying "Doc, it hurts when I do this." The doctor says, "Don't do that."

Please send constructive criticisms to Robert H. Lewis, Department of Mathematics, Fordham University, Bronx NY 10458, rlewis@fordham.edu.

September 24, 2023



z = 1.6 exp(-r^2/5) cos(r^2); r^2 = x^2 + y^2

-3 ≤ x ≤ 3, -3 ≤ y ≤ 3

Appendix 1. This is now obsolete.

Appendix 2. Table of Fermat Special Symbols

Symbol	*Keystroke*	*Meaning*	*Use*	*Apply to arrays?*
\	\	divide and truncate	$x\backslash y$, $[x]\backslash y$	yes
|	shift-\	modulo	$x\|y$, $[x]\|y$	yes
|...|	shift-\	absolute value	$\|(x+y)\|$	no
<>	<>	not equal to	$x <> y$	no
>=	>=	greater than or equal	$x >= y$	no
<=	<=	less than or equal	$x <= y$	no
$	shift-4	greatest integer	$\$x$	yes
^	shift-6	raise to power	$x\hat{}n$, $[x]\hat{}n$	yes
_	shift- -	suppress modular	_9087	no
:	shift- ;	suppress display; terminal only	10^9999 :	no
’	’	start, end quote	’ ... ’	no
”	shift-’	derivative of poly, quoly	x”	no
!	shift-1	factorial	x!	no
!	shift-1	display	$!(x,y,z)$, $!!x$, $![z$	yes
?	shift-/	get from terminal	?s, $?[x]$	yes
<	shift-,	previous value	$x := <$	yes
#	shift-3	polynomial evaluation	$x\#y$, $x\#(u = y)$, $x\#[y]$	yes
@	shift-2	panic stop	&@	no
@	shift-2	cancel	$@[x]$, $@F$, @<[x]>	yes
_	shift- -	concatenate arrays	$[x]$ _ $[y]$	yes
J	shift-j	add (drop) poly. var.	&J	no
{	shift-[	start comment	{...}	no
}	shift-]	end comment	{...}	no
}	shift-]	exit function	&}	no
>	shift-.	exit loop	&>	no
]	]	cycle (in loop)	&]	no
~	shift-‘	ellipsis	x[2~ ,4 ~]	yes
‘	‘	line continuation	only from terminal	no
<F	shift-, shift-f	system function	only from terminal	no
%	shift-5	array of arrays	%[1] := $[x]$	yes

Appendix 3. Some Significant Examples

These examples are not necessarily "best possible" solutions.

Example 1.

Note: To appease the typesetting program that created this document, a few small changes were made in non-ASCII symbols.

```
;  This set of functions was used in November 1990 by Robert H. Lewis to
; answer an unsolved question in group theory. A certain property of some
; groups called the Gkj groups was checked by counting the number of
; homomorphisms from Gkj -> SL(2, Z/4). It turns out that for certain k and
; j, this number can vary from the expected number. Gilbert Baumslag
; presented the question in this form.  Gkj is a one relator group with two
; generators; the relation involves the two generators raised to powers k
; and j. To find a homomorphism from Gkj into a group it is necessary and
; sufficient to find 2 elements of that group that satisfy the same relation.
; SL(2, Z/4) is picked because it is finite, 2x2 matrices are easy to
; compute with, and it has an appropriate lower central series.

; Fermat note: Fermat reads the first six statements and does them. Then
; it reads the function definitions. Then it executes the last 11 statements.
; There is no reason to have this particular order of statements in this file.
; In particular, the function definitions can be in any order.

&(t = 1);
&(m = 0);      ;; Turn off the user interrupt capability.
```

&(J = x); &(J = y); &(J = z); &(J = w); ;; Adjoin 4 polynomial variables.

```
;; Find [p] among the list of 2x2 matrices representing SL(2,Z/4).
 Function Find(i,size,j,ans) =
    ans := 0;
    for j = i, size do
       if p[1] <> e[j] then &]
       fi;
       if p[2] <> f[j] then &]
       fi;
       if p[3] <> g[j] then &]
       fi;
       if p[4] = h[j] then
          ans := &_m(j);
          & >
       fi
    od;
    ans.;

;; Find where each element [p]'s inverse transpose is in SL(2,Z/4).
```

```
 Function SetInvTr(i) =
    for i = 1, invs do
       [p] := [(e[i],f[i],g[i],h[i])];
       { ith element of SL(2,Z/4)  }
       STrans[p];
       [p] := [(p[4],-p[2],-p[3],p[1])];
       { fast inverse  }
       it[i] := Find(1, &_m(invs))
    od.;

;; Insert is used by DelConj.
 Function Insert(x1,i,k) =
    i := 1;
    while i <= j do
       if x1 > conjs[i] then
          for k = j + 1, i + 1, - 1 do
             conjs[k] := conjs[k-1]
          od;
          conjs[i] := x1;
          &_m(j :+ );
          &}
       else
          if x1 = conjs[i] then
             &}
          else
             &_m(i :+ )
          fi
       fi
    od;
    conjs[i] := x1;
    &_m(j :+ ).;

; Compute the conjugacy classes in SL(2,Z/4). Look at each element of
; SL(2,Z/4). Delete from the list of SL(2,Z/4) all of its conjugates.
; When finished, one representative is left in each class.
 Function DelConj(i,j,k,n,spot) =
    Array conjs[invs\2];
    Array a1[2,2];
    Array count[2];
    [count] := 0;
    &_m(size := invs);
    i := 1;
    while i <= size do
       [a1] := [(e[i],f[i],g[i],h[i])];
       j := 0;
       { Fill the array conjs to tell where the conjugates of [a1] are. }
       for k = 1, invs do
          [p] := [(e1[k],f1[k],g1[k],h1[k])];
          [q] := [(p[4],-p[2],-p[3],p[1])];
          [p] := [q]*[a1]*[p];
          spot := Find(i, size);
```

```
         if spot = 0 then
            !!'spot is 0'
         fi;
         if spot <> i then
            Insert(spot)
         fi
      od;
      !!('number of conjugates is ', &_m(j + 1));
      [count] := [count] \_ &_m(j + 1);
      { Delete the conjugates of [a1].  Change [it] accordingly.   }
      for k = 1, j do
         &_m(size:-);
         for n = 1, conjs[k] - 1 do
            if it[n] = conjs[k] then
               it[n] := i
            fi;
            if it[n] > conjs[k] then
               &_m(it[n] := it[n] - 1)
            fi
         od;
         for n = conjs[k], size do
            e[n] := e[n+1];
            f[n] := f[n+1];
            g[n] := g[n+1];
            h[n] := h[n+1];
            it[n] := it[n+1];
            if it[n] = conjs[k] then
               it[n] := i
            fi;
            if it[n] > conjs[k] then
               &_m(it[n]:-)
            fi
         od
      od;
      &_m(i :+ )
   od;
   [count] := [count[3~]];
   @([conjs], [a1]).;

;; OneDet tests every possible 2x2 matrix over Z/4 and saves the
;; invertible ones.  j1 is owned by CreateData.
 Function OneDet(i,j,k,l) =
   for i = 0, dim - 1 do
      for j = 0, dim - 1 do
         for k = 0, dim - 1 do
            for l = 0, dim - 1 do
               if i*l - k*j = 1 then
                  &_m(j1 := j1 + 1);
                  e[j1] := i;
                  f[j1] := j;
                  g[j1] := k;
```

```
                  h[j1] := l
               fi
            od
         od
      od
   od.;

;; Count the number of all 2x2 matrices over Z/4 that have determinant 1.
 Function CountInv(i,j,k,l,j1) =
   for i = 0, dim - 1 do
      for j = 0, dim - 1 do
         for k = 0, dim - 1 do
            for l = 0, dim - 1 do
               if i*l - k*j = 1 then
                  &_m(j1 :+ )
               fi
            od
         od
      od
   od;
   j1.;

 Function CreateData(j1) =
   Array e[invs];
   Array f[invs];
   Array g[invs];
   Array h[invs];
   OneDet;
   [e1] := [e];
   [f1] := [f];
   [g1] := [g];
   [h1] := [h].;

;    Main tries one pair of exponents, exp1 and exp2.  It counts the number
; of homomorphisms from the group they define into the 2x2 matrix group.
; The basic word that defines the group has been massaged to minimize
; the amount of work within the innermost loop.  That is also the reason
; for using the polynomials -- part of the word can be evaluated in advance
; "once and for all".
 Function Main(exp1,exp2,i,j,k,l,yes) =
   time := &T;
   hits := 0;
   for i = 1, size do
      [c] := [(e[i],f[i],g[i],h[i])];
      [ci] := [c]^exp1;
      [cj] := [c]^exp2;
      [prod1] := [cj]*[b];
      [prod2] := [b]*[cj];
      { If [c] is in the center, so that count[i] = 1,
        the answer is easy. Increment and cycle.   }
      if count[i] = 1 then
```

```
         &_m(hits := hits + invs);
         &]
      fi;
      for j = 1, invs do
         &(m = 1);
         { Allow mouse interrupts.   }
         [a] := [(e1[j],f1[j],g1[j],h1[j])];
         [p] := [ci]*[a];
         &(m = 0);
         { Turn off mouse interrupts.   }
         [a1] := [(p[4],-p[2],-p[3],p[1])];
         [test] := [a1]*[a]*[p];
         [test] := [prod1] - [prod2]*[test];
         for k = 1, invs do
            yes := 1;
            for l = 1, 4 do
               if test[l]#(h1[k], g1[k], f1[k], e1[k]) <> 0 then
                  yes := 0;
                  & >
               fi
            od;
            if yes = 1 then
               &_m(hits := hits + count[i])
            fi
         od
      od
   od;
   !!('time is ', &_m((&T - time)/60), 'seconds.');
   !!('i,  j,  hits are ', exp1:3, exp2:3, hits).;

; Driver is the 'main program'. It first counts the number of invertible
; 2x2 matrices over Z/dim, using CountInv. (dim = 4 is the right choice,
; for which invs is 48). CreateData then actually creates the 48 matrices.
; Matrices are stored in 4 arrays [e], [f], [g], [h],one for the upper
; left coordinates, one for the lower left, etc. (This is inelegant and
; could be done better with Fermat's array of arrays.)  These are
; duplicated into [e1], [f1], etc. Then SetInvTr finds where each matrix's
; inverse transpose is. This data is stored in array [it]. DelConj breaks
; the group into conjugacy classes. (This is done because the basic defining
; word is invariant under conjugation, so we save some time this way.)
; When it's finished, [e], [f], [g], [h] store 1 representative from each
; class, and [it] says where the inverse transpose of each of these
; representatives is.  But it turns out that it[j] = j for all j.  Had
; this not been true, a further speed up would have been possible. Then a
; loop indexed on i tests the exponents from the arrays [try1] and [try2]
; thereby looking for maps from the group Gkj,where k = try1[i] and j =
; try2[i]. Main counts the number of homomorphisms from the Gkj group into
; SL(2,Z/4).
 Function Driver(size,time,invs) =
   time := &T;
   Array a[2,2];
```

```
    Array b[2,2];
    Array c[2,2];
    Array ci[2,2];
    Array cj[2,2];
    Array p[2,2];
    Array a1[2,2];
    Array b1[2,2];
    Array q[2,2];
    Array test[2,2];
    [b] := [(x,z,y,w)];
    invs := CountInv;
    !!('order of group is ', invs);
    Array it[invs];
    CreateData;
    SetInvTr;
    !!('initial [it]');
    ![it;   !;
    DelConj;
    !!('time is ', &_m((&T - time)/60), ' seconds.');
    !!('size is ', size);
    { size = number of conjugacy classes.   }
    !!('Now [it] is ');
    ![it[1~size];  !;
    !!'[count] is ';
    ![count;  !;
    for i = 1, 100 do
       Main(try1[i], try2[i])
    od.;

?dim;                       ; Ask user for dim.
&(p = dim);                 ; Go to modular mode mod dim. In converting, dim
                            ;   becomes 0.  (of course)
dim := -1;
&_m(dim := dim + 1);        ;  dim now holds dim, not 0.
Array try1[100];            ; Create exponents to try for k and j.
Array try2[100];
[try1[1~7]] := 1;
[try1[8~14]] := 2;
&(a = 0);
[try2] := [<i=0,99> ( &_m (i|7 + 2) )];
&(a = 1);

&x;
```

Example 2.

These functions compute the determinant *det* of a matrix $[x]$ with one-variable polynomial entries. This method is now part of the built-in determinant function. The strategy is to figure out a bound on the degree of the determinant *det*, then evaluate *det* at a set of points, then use interpolation to construct *det*.

The hard thing is to estimate in advance the degree of the determinant. Several methods

are possible. This one has the advantages of being fairly fast and not often overestimating the degree. A bad overestimate will cost dearly in the final interpolation, here accomplished with the Sigma command. This method also spotlights many of Fermat's features.

We assume that the matrix $[x]$ has already been defined. 'Degree' estimates the degree of *det* by simulating what would happen to the terms of highest degree in the matrix if the determinant were actually computed with row and column manipulations.

Rational mode is assumed.

```
Function Degree(n,i,j,k,diff,answer) =
   { Create matrix of degrees of entries in [x]. n = dim. of [x]. }
   Array y[n,n];
   for i = 1, n*n do
      if x[i] = 0 then  y[i] := -10^10   { " -infinity" }
         else y[i] := Deg(x[i]) fi
   od;
   answer := 0;
   { This loop simulates row and column manipulations in [x] by working
      with the degrees in [y].  }
   for i = 1, n do
      { Try to factor out as many powers as you can.  }
      Rowscan(n, i);
      Colscan(n, i);
      { Find spot with lowest degree >= 0.  }
      j := Det_ + ([y[i~,i~]] + 1);
      if j = -1 then  & > fi;
      { Convert linear to (row, column) coordinates in [y].  }
      k := (j - 1)\(n - i + 1) + i;
      j := (j - 1)|(n - i + 1) + i;
      { If the remaining matrix is all constants, stop now.  }
      if Maxdegree(n, i) = 0 then & >  fi;
      { Simulate row and column manipulations.  }
      Switchrow([y[,i~]], i, j);
      Switchcol([y[i~,]], i, k);
      for j = i + 1, n do
         diff := y[j,i] - y[i,i];
         for k = i + 1, n do
            [y[j,k]] := Max(y[j,k], diff + y[i,k])
         od
      od;
      answer := answer + y[i,i]
   od;
   answer.;

Function Rowscan(n,i,r,j,deg) =
   for r = i, n do
      { Find position and value of smallest nonnegative entry in rth row
             in columns >= i.  }
      j := Det_ + ([y[r,i~]] + 1);
      deg := y[r,j+i-1];
      { If deg is positive, factor it out, changing answer accordingly.  }
      if deg > 0 then
```

```
         [y[r,i~]] := [y[r,i~]] - deg;
         answer := answer + deg
      fi
   od.;

Function Colscan(n,i,r,j,deg) =
   for r = i, n do
      { Find position and value of smallest nonnegative entry in rth column
            in rows >= i.  }
      j := Det_ + ([y[i~,r]] + 1);
      deg := y[j+i-1,r];
      { If deg is positive, factor it out, changing answer accordingly.  }
      if deg > 0 then
         [y[i~,r]] := [y[i~,r]] - deg;
         answer := answer + deg
      fi
   od.;

Function Maxdegree(n,i,r,s,deg) =
   deg := 0;
   for r = i, n do
      for s = i, n do
         if y[r,s] > deg then
            deg := y[r,s];
            &>
         fi
      od;
      if deg > 0 then &>  fi
   od;
   deg.;

Function Max(a,b) =  if a > b then a else b fi.;

{ Assume that t is the polynomial variable. }

n := Degree(Cols[x]) \2;
{ Standard LaGrange interpolation at points -n, -n+1, ..., n-1, n:}

f := Prod < i=-n,n > [ t-i ];

det := Sigma<j=-n,n>[ Det([x]#j)*f/(t-j)/ Prod<i=-n,j-1> (j-i)/
       Prod<i=j+1,n> (j-i) ];
```

Example 3.

The Leverrier-Faddeev method for computing matrix inverses and characteristic polynomials (see "The College Mathematics Journal," May 1992, p. 196.) I present two slightly different programs for each case.

```
Function CharPoly(a,n,i,pm) =
   { Assume t is a polynomial variable.  }
   n := Cols[a];
```

```
   Array c[n];
   [q] := [a];
   c[1] := -Trace[a];
   for i = 2, n do
      [q] := [q] + c[i-1]*[1];
      [q] := [a]*[q];
      c[i] := -Trace[q]/i
   od;
   { Correct for odd n.  }
   if n|2 then
      pm :=  - 1
   else
      pm := 1
   fi.;

Function Invert(a,q,n,i) =
   { Set [q] = the inverse of [a]  }
   n := Cols[a];
   Array c[n];
   [p] := [a];
   c[1] := -Trace[a];
   for i = 2, n do
      [q] := [p] + c[i-1]*[1];
      [p] := [a]*[q];
      c[i] := -Trace[p]/i
   od;
   [q] := [q]/(-c[n]);
   @([p], [c]).;
```

If appropriate, the inclusion of *Integer* in either procedure will provide a very dramatic increase in speed. However, *Integer* cannot be placed in the first procedure as it stands, because of the presence of the polynomial variable t. Rather, all but the the last line could be placed in a seperate procedure that may contain *Integer*.

Appendix 4. Summary of Matrix and Polynomial Features

Matrix and polynomial computations are the heart of Fermat. Here is a concise annotated list of relevant features and commands. Most are explained further earlier in the manual.

Polynomials

Recall that most built-in functions allow *call-by-reference* for parameters. For example, time and compare $Terms(x)$ and $Terms(\hat{}x)$ for a large x.

Adjoin polynomial variable: use the command &J (Fermat interrogates user for name), or &(J=t) (imperative to adjoin t).

Cancel (delete) polynomial variable: similar to above, &(J=$-t$).

Quotient rings and fields: One may choose to mod out by some of the polynomial variables to create quotient rings (or fields). The chapter on "Polymods" describes how. In principle, any monic polynomial may be modded out, say $t^n + c_1 t^{n-1} + \ldots$.

However, Fermat is best at the case where the quotient ring becomes a field (note well, $\boldsymbol{Q}[t, u, \ldots] \; / < p, q, \ldots >$ is a field, $\boldsymbol{Z}[t, u, \ldots] \; / < p, q, \ldots >$ is not). Specifically, suppose the polynomial variables have been attached in the temporal sequence $t, u, v, \ldots$. Begin by modding out a monic irreducible polynomial $p(t)$ such that $F_1 = \boldsymbol{Z}[t]/ < p >$ is an integral domain, and its field of fractions is $\boldsymbol{Q}[t]/ < p >$. Then, if desired, mod out by a monic polynomial $q(u, t)$ such that $F_2 = F_1[u]/ < q >$ is an integral domain, and continue in this manner always creating an integral domain, and, by the same stroke, its field of fractions.

You must tell Fermat that a field will result, and it is your responsibility to check this. Do this at each step by adding a comma and 1, as &(P $= t^n + c_1 t^{n-1} + \ldots$, 1). You may append a list of primes q such that the modder $p(t)$ remains irreducible mod q. For example, &(P $= t^3 + t^2 + t + 2, 1 : 151, 167, 191, 839, 859, 863, 907, 911, 991$). If you omit the list Fermat will compute it for you. (This takes a while, so if you save the session to a file, Fermat will include the list in the saved file and just reload it next time.) Fermat then computes a second list of auxiliary primes: modulo these primes the modding polynomial has a root. Both types of primes are used to speed up g.c.d. computations. If Fermat cannot find enough of either type, it will tell you and instruct you how to get more, using the commands &A and &B.

Laurent Polynomials: a polynomial with negative exponents. To allow this, activate the toggle switch &l. All of the variables you have created up to that point will be converted so that no negative exponents are in the denominator of a quolynomial and all positives are factored out and moved to the numerator. For example, $1/(t^2+2t)$ will become $t^{-1}/(t+2)$.

Basic arithmetic: $+$, $-$, $*$, $/$, $|$, $\backslash$, $\hat{}$. p/q creates a rational function ("quolynomial") unless q divides evenly into p. In any event, $GCD(p, q)$ is computed and divided into p and q. $p|q$ means p mod q and $p\backslash q$ means p div q, i.e. divide and truncate.

Mod and div may be "pseudo" results: if c is the leading coefficient of q and is not invertible, there exist polynomials r and y such that $c^k p = yq + r$, where $0 \le k \le deg(p) + deg(q)$. Fermat chooses the smallest possible k. $p|q$ is r and $p\backslash q$ is y.

Remquot = remainder and quotient. Syntax is *Remquot*(x, y, q). q gets the quotient of dividing x by y and the function returns the remainder. Almost twice as fast as calling *mod* and *div* separately. Pseudo-remainder and quotient will be returned if necessary.

Deg. degree of a polynomial (or quolynomial). There are three variants: (1) *Deg*(x) computes the highest exponent in x (any expression) of the highest precedence polynomial variable. (2) *Deg*(x, i) computes the highest exponent in x of the i^{th} polynomial variable, where the highest level variable has the ordinal 1. (3) *Deg*(x, t) computes the highest exponent in x of the polynomial variable t. In modular mode, *Deg* returns an actual integer, not reduced modulo the modulus. For a quolynomial, it returns the degree of the numerator.

Codeg, "codegree," just like *Deg* except it computes the *lowest* exponent.

$\#$ = polynomial evaluation. $x\#y$ replaces the highest precedence variable everywhere in x with y. $x\#(u = y)$ replaces the variable u with y. x and y could be quolynomials.

There is a fast shortcut form of evaluation called *total evaluation*. To evaluate x at every variable, use the syntax $x\#(v_1, v_2, \ldots)$, where all the v_i are numbers. There must be a number corresponding to each polynomial variable, in the precedence order – highest precedence (last attached) first.

Fermat also allows evaluation of polynomials at a square matrix. The syntax is $x\#[y]$. The highest precedence polynomial variable in x is replaced with the matrix $[y]$ and the resulting expression simplified. $[y]$ can contain entries that are quolynomials.

The following more general syntax is allowed. Suppose there are five poly vars, e, d, c, b, a in that order (e last and highest). Then $q\#(d = w, x, y)$ will replace each d in q with w, each c with x, each b with y. e and a are untouched. Further, w, x, and y can be arbitrary quolynomials.

Similarly if $[t]$ is an array, q # $(d = [t])$ replaces the variables from d on down with the entries of $[t]$ in column major order until $[t]$ is used up. It is an error if $[t]$ has too many entries.

Numb = is the argument a number (as opposed to a polynomial or quolynomial)? If so the result is 1, else it's 0.

Numer = numerator of a quolynomial. In rational mode, also gives the numerator of a rational number.

Denom = denominator of a quolynomial. In rational mode, also gives the denominator of a rational number.

Coef in a polynomial (or quolynomial).
(1) Suppose first that only one polynomial variable t has been adjoined. Then the syntax of use is either *Coef*(x) or *Coef*(x, n). x can be any expression. n, the desired exponent, must be a number. In the first form, without n, the leading coefficient is computed. *Coef*(x, n) returns the coefficient of t^n in the polynomial x. If x is a quolynomial, the denominator is ignored.

To replace a coefficient, use *Rcoef*$(x, n) := y$; the coefficient of t^n in x will be set to the expression y. y must be a number.

If there are several polynomial variables, the coefficient desired is specified by listing the exponents of the variables in precedence order, such as *Coef*$(x, 1, 2)$.

In *Rcoef(*$x, \ldots$*)* $= y$, x must be a polynomial and ... must be suitable for y.
(2) If t is any polynomial variable, *Coef(*x, t, n*)* computes the coefficient in x of t^n, as if t were the highest level variable. In other words, if x were written out as a sum of monomial terms, find all the terms containing exactly t^n and factor out the t^n. x must be a variable name, either a scalar name or an array reference $x[i]$. This form cannot be used on the left of an assignment. Especially useful for $n = 0$.

Killden(x) sets the denominator of x to 1 – it actually changes x.

Lterm(x) = leading term of polynomial x.

Lcoef(x) = leading numerical coefficient of polynomial x.

Flcoef(x) = leading field-element coefficient of polynomial x, when a quotient field has been created.

Lmon(z) = leading monomial of z. *Lmon* has an optional second argument. First, *Lmon(z)* returns the leading monomial of z. This is always an authenic monomial. If you think of a multivariate polynomial in nested (recursive) form, *Lmon* recursively finds the first term in each level and throws away all the other terms. *Lmon(z, x)*, where x is a poly var, stops the recursion at the level of x. For example, suppose u is the higher variable and t the lower variable. Let $z = (t^2 + 3t + 5)u^2 + 5t * u + 7t - 2$. Then *Lmon(z)* is $t^2 * u^2$ but *Lmon(z, t)* is $(t^2 + 3t + 5)u^2$.

Mcoef(x, m1, m2) = monomially-oriented coefficient. $m1$ (and $m2$ if present) must evaluate to a monomial; their numerical coefficient is irrelevant. Factor out $m1$ from all the terms in x that contain it *exactly,* and return the factor. $m2$ (optional) specifies variables that must occur with exponent 0 (as they cannot be included in $m1$!).

Mfact(x, m) = monomially-oriented factor. m must evaluate to a monomial; its numerical coefficient is irrelevant. Factor out m from all the terms in x that it *divides into* and return the factor. m may not contain any negative exponents.

Mono(a,b) to compare monomials; returns true iff $a \leq b$. Numerical coefficients are irrelevant. Individual variables are ranked by order of their creation, latest is largest. Monomials are ranked first by length = #variables, i.e. all univariates < all bivariates < all trivariates, ... etc. Within equal #vars, ranking is by subset of vars. Within subset, ranking is exponent arrays. See "Sort" below. This is the order that is used internally by Fermat for certain algorithms.

Mons(x, [a]) dumps the monomials of x into a linear array $[a]$. If you want each monomial stripped of its numerical coefficient, use *Mons(x, [a], 1)*.

Nextvar: Post 2023: *Nextvar* returns the position (level) of the second variable in the poly argument. If there is only one var, returns 0. If input is a number or field element, returns -1.

PRoot(x) returns the p^{th} root of x, when $x \in$ the ground ring, a field of characteristic p.

Splice: Splice(s, c, m) multiplies c by x^m and "tacks it in front" of s; it adds them, if there is overlap. c becomes 1, s gets the answer. see *Split*.

Split is basically the dual of *Splice*. Invoke with *Split*(w, n, v) or *Split*(w, n). w and v are

existing scalar variables (including entries in an array). n is an integer. w becomes w mod x^n, where x is the highest variable. v (if present, optional) becomes $w \backslash x^n$.

Terms(x) = if x were written out as a sum of monomial terms, the number of such terms. x must be a variable name, either a scalar name or an array reference $x[i]$. does not count field variables (see next function).

Vars(x) = number of variables that actually occur in x.

Termsf(x) = counts the total number of terms when polymodding to create a field, counting the field variables. If $s = (t+1)x + t - 1$ where t is a field variable, *Termsf(s)* = 4 while *Terms(s)* = 2.

Post 2023: *Terms* and *Termsf* can have a second argument to stop counting if a specified limit is reached.

&_l: Toggle switch to display each polynomial as a list of monomials.

&c: Enable full Hensel checking. This one is quite technical. If this flag is on (the default) Fermat will double-check the results of certain Hensel Lemma GCD computations. Leaving it off will slightly speed up GCD but introduce an extremely minute probability of GCD giving the wrong answer. See &O next below.

&O: Toggle switch to disable the Hensel and Chinese Remainder Theorem (CRT) methods for polynomial gcd. This is a good idea only when you are working over Z_p for small primes, say $p < 30$, and the degrees in each variable are fairly small. For such small primes, the Hensel and CRT methods often fail, for "lack of room."

" = *derivative* of a polynomial (or quolynomial) with respect to the highest precedence variable (the last attached), as in x" (see also *Deriv* below).

GCD = *greatest common divisor*, as in $GCD(x, y)$, x and y can be numbers or polynomials, but not quolynomials. If numbers, the result is always positive, except that $GCD(0, 0) = 0$. $GCD(0, x) = |x|$ if x is a number not 0, and is 1 if x is a polynomial. If they are both polynomials, the result always has positive leading coefficient. If in rational mode, the result has all coefficients integral and content 1. In cases where the ground ring is a field, the result has leading coefficient 1.

EGCD = *extended GCD*, as in $EGCD(x, y, u, v)$, x and y are one-variable polynomials over a finite field. Compute u and v s. t. $u * x + v * y = GCD(x, y)$.

Content = *content* of a polynomial; i.e., the GCD of all its coefficients. *Content(x,i)* is content w.r.t. i^{th} variable.

Numcon = *numerical content*, the GCD of all its numerical coefficients.

Var. Followed by an expression that evaluates to a positive integer, as in $Var(i)$ returns the i^{th} polynomial variable, counting the highest (last created) as 1.

Height = the difference between the levels (ordinals) of the polynomial variables in an expression.

Level = the ordinal position of the highest precedence polynomial variable in an expression.

Raise = Two forms: *Raise*(x) and *Raise*(x, i). In the first, replace each polynomial variable with the variable one level higher. The second form allows the user to provide an expression i that evaluates to a positive integer, and raises x that many levels, if possible.

Lower = The inverse of *Raise*. See above.

Divides(n, m) = does n divide evenly into m?

PDivides(n, m) = does n divide evenly into m? When n and m are multivariable polynomials, this procedure attempts to answer quickly by substituting each polynomial variable except the highest with a constant. *PDivides* says true iff these reduced polynomials divide evenly. The constants are chosen with care. Nonetheless, this is a probabalistic algorithm. An answer of False is always correct, but an answer of True has an infinitesimal probability of being wrong.

SDivide(n, m) = does n divide evenly into m? "S" stands for "space-saving". To save space m is cannibalized. If n does divide m, m becomes the quotient; if not, m becomes 0. m must be a variable name, not an expression. Not probabalistic. Use when you are virtually certain that n divides m and you want the quotient in the fastest way.

Powermod(x, n, m) computes $x^n \ mod \ m$. x must be a polynomial or integer, n must be a positive integer, and m must be a monic polynomial or positive integer. You may omit the third argument if you are in modular mode or polymodding. Note that n often needs to be very large. In modular mode, this is a problem. The solution is that n must be either a constant or must involve only variables that have been created in rational mode while under "Selective Mode Conversion."

Deriv(x, t, n) returns the n^{th} derivative of x with respect to t, where t is one of the polynomial variables.

Poly$[a]$: With x = the highest polynomial variable, *Poly*$[a]$ takes an array OF NUMBERS and yields $\Sigma < i = 1, n > [\, a[i]\, x^{n+1-i}\,]$, for a standard array. For a sparse array, only the first entry in each row is used.

Splice: Splice(s, c, m) multiplies c by x^m and "tacks it in front" of s; it adds them, if there is overlap. c becomes 1, s gets the answer. see *Split*.

Split is basically the dual of *Splice*. Invoke with *Split*(w, n, v) or *Split*(w, n). w and v are existing scalar variables (including entries in an array). n is an integer. w becomes w mod x^n, where x is the highest variable. v (if present, optional) becomes $w \backslash x^n$.

Totdeg$(x, [a])$ returns the array $[a]$ of largest and smallest monomial degrees. $[a]$ need not exist before. x is a polynomial; laurent is ok. x is homogeneous iff $a[1] = a[2]$.

WDeg$(x, [a], n)$ "withdraw" the subpolynomial of x of total degree n in the variables listed in $[a]$. $[a]$ is an existing array. Each entry should be a single polynomial variable, in no particular order. n is an integer. x is a polynomial; laurent is ok. "Variables" at field depth or below should not be included in $[a]$.

Vars(x) = number of variables that actually occur in x.

Factoring Polynomials: Pre 2023: Fermat allows the factoring of monic one-variable polynomials over any finite field. The finite field is created by simply being in modular mode over a prime modulus, or by additionally modding out by irreducible polynomials to form a more complex finite field, as described in the section "Polymods." Factoring into irreducibles or square-free polynomials is possible, or polynomials can just be checked for irreducibility.

$Factor(poly, [x])$ or $Factor(poly, [x], level)$. The factors of *poly* will be deposited into an array $[x]$ having two columns and as many rows as necessary. (The number of factors (rows) is returned as the value of $Factor$.) In each row, the first entry is an irreducible

polynomial $p(t)$ and the second is the largest exponent e such that $p(t)^e$ divides *poly*. In the second form, *level* specifies the subfield to factor over. Examples are given earlier in this manual. It is best for factoring to use as many variables t, u, ... as possible in creating the field.

Sqfree is similar to *Factor* except it produces factors that are square-free only.

Sqfree works for any number of variables and over $\boldsymbol{Q}$. Also, it works recursively by first extracting the content of its argument and factoring it. Over quotient fields, the product of all the factors in the answer may differ from the argument by an invertible factor.

Irred tells if its argument is irreducible, and, if not, describes the factorization. The syntax is $Irred(< poly >)$ or $Irred(< poly >, < level >)$ ("level" is explained above). The value returned is as follows:

-1 means can't decide (too many variables, for instance).
0 means the argument is a number or a field element.
1 means irreducible.
$n > 1$ means the argument is the product of n distinct irreducibles of the same degree.
x, a poly, means x is a factor of the argument (which is therefore not irreducible).

Fermat uses the algorithms of H. Cohen, "A Course in Computational Number Theory," Springer Verlag, 1993, p. 123-130.

Post 2023: *Factor* and *Irred* have been significantly generalized. See p. 47 - 48.

Post 2023: *Revpoly* returns the reverse of its argument.

Matrices

Creation: An $n \times m$ matrix is created with the command *Array* $x[n, m]$. Access elements in such an array via $x[i, j]$ or via $x[k]$, which returns the k^{th} element in column-major order. To refer to an entire matrix, use the syntax $[x]$.

Sparse Matrices: Sparse matrices are implemented in Fermat. This is an alternative mode of storing the data of the array. In an "ordinary" $n \times m$ matrix, nm adjacent spots in memory are allocated. If an array consists of mostly 0's, this is wasteful of space. In a *Sparse* implementation, only the non-zero entries are stored in a list structure.

A *Sparse* matrix is created by following the creation command with the keyword "*Sparse*," as in *Array* $x[5, 5]$ *Sparse*. There is no size limitation in Fermat. An array $[x]$ already created can be converted to *Sparse* format with the command *Sparse* $[x]$. There is no requirement that $[x]$ actually have a certain number of zeros.

Indexing: One has a choice of how to index the first element of an array. The default in Fermat is $x[1]$. This can be changed by entering the command &a, which switches the initial array index to 0. Entering &a again switches back to 1. Note that this is not a property of any particular array, but of how all arrays are indexed.

Dynamic Allocation of Arrays: Arrays that are no longer needed can be freed to provide space for new arrays. This is done with the cancel command, whose syntax is @$[x]$, or, to free several, @$([x], [y], [z])$.

Arithmetic: Most of the ordinary arithmetic built-in functions can be applied to arrays. See Appendix 2, last column. For example, $[x]+[y]$ is the sum. $2*[a]$, or $[a]*2$, multiplies every component of $[a]$ by 2. $[a]+3$ adds 3 to every component of $[a]$, and so forth. $[a] := [1]$ sets an already existing square matrix $[a]$ equal to the identity. $[a] := 1$ sets every entry to 1. $[z] := [x] * [y]$ is the product. $[z] := 1/[y]$ is the inverse. Matrix exponentiation (including inverse) is just like scalars (but see *Altpower* below), such as $[z] := [x]\hat{}n$.

Arithmetical Expressions: Like numerical expressions, such as $[z] := [a] * ([x] + [y] - [1])$.

Parameters: Matrices may be parameters in functions.

Subarrays: Fermat allows subarray expressions. That is, part of an array $[c]$ can be assigned part of an array $[a]$. For example, $[c[1 \sim\ 4,2 \sim\ 6]] := [a]$ sets rows 1 to 4 and columns 2 to 6 of $[c]$ equal to $[a]$. This assumes that $[a]$ is declared to be 4×5 and $[c]$ is at least 4×6. (Here $\sim$ is shift-‘). In defining the subarray, if one of the coordinate expressions is left out, the obvious default values are used. For example, if $[c]$ has four rows then $[c[, 2\sim\ 6]] := [a]$ is equivalent to the above. Similarly, one can use expressions like $[c[3\sim\ ,2\sim\ 6]] := [a]$ or $[c[\sim$ 4,2$\sim\ 6]] := [a]$, in which case the default lower row coordinate is the array initial index, 0 or 1.

In subarray assignments, a vector declared to be one-dimensional (like $a[5]$) is treated as a column vector, i.e., $a[5, 1]$.

As of April 2016 in 64 bit, subarray can be used with Sparse matrices. Minors is similar; see below.

Matrix Built-in Functions:

Det, is used in several ways to compute a scalar from an array argument. If used by itself on a square matrix, *Det* is determinant. *Det#*$([x] = a)$ returns the number of entries in $[x]$ that equal a. Similarly *Det#*$([x] > a)$ and *Det#*$([x] < a)$ compute the number of entries of $[x]$ larger or smaller than a. If any entry is a polynomial, an error results. *Det^*$[x]$ returns the index of the largest element of $[x]$ (in column major order if $[x]$ is a matrix). *Det_*$[x]$ returns the index of the smallest element of $[x]$. *Det_*$+[x]$ returns the index of the smallest nonzero element of $[x]$, or -1 if there is no such element.

Determinant: Fermat uses four basic methods to compute determinant: expansion by minors, Gaussian elimination, Lagrangian interpolation, and reducing modulo n for some n's. The last of these is used for matrices of integers or polynomials with integer coefficients. The actual determinant can be reconstructed from its values modulo n (for a “good” set of n's) by the Chinese Remainder Theorem (see Knuth volume 2). Alternatively, it is often possible to work modulo an easily computed “pseudo determinant” known to bound the actual determinant. Gaussian elimination is applicable in all situations – all one needs is the ability to invert any nonzero element in a matrix. If the matrix is small enough, expansion by minors is faster (see &D.) Gaussian elimination can be nontrivial and even problematical in modular arithmetic over a nonprime modulus, in polynomial rings, and in polynomial rings modulo a polynomial. Fermat has heuristics to guide its choice of method.

When the matrix has all polynomial entries, Fermat has two other methods. It may also compute the determinant with Lagrangian interpolation: constants are substituted for the

highest polynomial variable everywhere in the matrix, and so on recursively. The algorithm is probabalistic, with very high probability of success. It is very fast, especially for two or more variable polynomials.

Nonetheless, if there are many polynomial variables and the matrix is sparse or has a regular pattern of zeros, expansion by minors can be by far the fastest method. Setting the determinant cutoff (with &D) at least as large as the number of rows will force Fermat to do this method. This is true of Sparse matrices as well as "ordinary."

Secondly, Fermat uses the well-known Gauss-Bareiss method (for a matrix of all polynomial entries).

Fermat has heuristics to choose among the methods, but the user may override them and force a particular method. Assuming an $m \times m$ matrix of all polynomial entries, if m is more than three and the user has left &D $= -1$, the default method is Lagrangian interpolation, unless the "mass" of the matrix is very small. The "mass" is estimated by a heuristic and compared to a cutoff. The user can change the cutoff with the command &L. The default is 5000. Therefore, to turn off Lagrangian interpolation, give a very large value (up to $2^{31} - 1$). To then choose Gauss-Bareiss, set the command &K $= 1$. This is a bit confusing, so summary:

To force Gauss-Bareiss, set &K $= 1$ and &D $= 2$.

To force Gaussian elimination, set &K $= 0$ and &D $=$ any $d > 0$. At the $d \times d$ stage, it will switch to expansion by minors.

If $m \geq 4$, to force Lagrangian interpolation, set &D $= -1$ and &L $= 1$.

LCM = the least common multiple of all the denominators in a matrix. "Denominators" means those of rational numbers or of expressions like $(t^2+3t+1)/17$ or $3/(2t)$. Use this to clear a matrix of its integer denominators. The denominator of $2/(3+2t)$ is ignored, since you can't clear it by multiplying $[x]$ by any number.

Adjoint = adjoint of a square matrix.

Chpoly = the characteristic polynomial of a square matrix. The syntax of the command and the method used depend on whether the matrix is sparse or "ordinary."

With the ordinary matrix storage scheme, LaGrange interpolation is used when the matrix consists of all numbers. It is to your advantage to clear the matrix of all numerical denominators before invoking *Chpoly*. To do LaGrange interpolation, Fermat computes the necessary determinants using the Chinese Remainder Theorem. To do so, it must make an initial estimate of the absolute value of the determinant. The estimate is often rather liberal. The determinants in question are simply $f(c_i)$, where f is the characteristic polynomial and $\{c_i\}$ is a set of "sample points." The user may be able to supply a better bound on $|f(c_i)|$, so there is a second optional argument to *Chpoly*, a polynomial g such that $|f(t)| \leq |g(t)|$ for all t. The syntax is *Chpoly*$([x], g)$.

With sparse matrices, a clever way to compute characteristic polynomial is the Leverrier-Faddeev method. This method often loses to the standard one, $det([x] - \lambda I)$, but it can be faster for matrices that contain quolynomials. The user may choose the method with the second argument: *Chpoly*$([x], n)$ selects the standard method when $n = 0$ and the Leverrier-Faddeev when n is any other integer.

As of October 2009, the LaGrange modular determinant coefficients can be dumped to a file, rather than stored in RAM. This can be a big space saving when doing a very

large computation. The command is &(_L=1). In other words, if the highest precedence variable is x and lower ones are $y, z, \ldots$, and if a determinant $c_n x^n + c_{n-1} x^{n-1} + \ldots$ is being computed with LaGrange interpolation, the coefficients $c_0, c_1, ..., c_n$ (which are polynomials in $y, z, \ldots$) will be dumped to the output file to save RAM.

Minpoly: The "modifed Mills method," a fast probabalistic algorithm that computes the minimal polynomial $M(t)$ of a sparse matrix of integers, or, more precisely, a factor of the minimal polynomial. If one of the roots of $M(t)$ is 0, the associated factor t of $M(t)$ will not show up, but other factors may not show up either. This algorithm is built into Fermat via the command *Minpoly*. Syntax of use is *Minpoly*([*a*], *level*, *bound*). [*a*] is the matrix, which must be *Sparse*. *level* $= 0, 1, 2, 3, 4$ is a switch to tell *Minpoly* how much effort to expend in its basic strategy. Larger levels will take longer, but have a better chance of giving the entire minimal polynomial. *bound* is an integer at least as big as any coefficient in the minimal polynomial. This argument can be omitted, in which case Fermat will supply an estimate based on the well-known Gershgorn's Theorem.

Repeated calls to *Minpoly* may return different answers. It may be worthwhile to run it several times and compute the l. c. m. of the answers.

Sumup = add up the elements of an array.

Trace = trace of a matrix.

Altmult. Multiply two matrices using the algorithm of Knuth volume II, p. 481. A big time saver when multiplication in the ring is much slower than addition. Especially good for Polymods (see that chapter). Syntax is *Altmult*([*x*], [*y*]).

Altpower. Uses *Altmult* to take a matrix [*x*] to the power n. Syntax is *Altpower*([*x*], n).

MPowermod([*x*], n, m) computes $[x]^n \ mod \ m$. [*x*] contains only polynomials or integers, n must be a positive integer, and m must be a monic polynomial or positive integer. You may omit the third argument if you are in modular mode or polymodding. Note that n often needs to be very large. In modular mode, this is a problem. The solution is that n must be either a constant or must involve only variables that have been created in rational mode while under "Selective Mode Conversion."

To see the effect of this command, try creating a 20×20 matrix of random integers, let $n = 100$ and $m = 10^8$. Compute $[x]^n \ mod \ m$ both ways.

Trans = transpose matrix, as in [*y*] := *Trans*[*x*].

STrans = transpose a matrix in place, as in *STrans*[*x*].

Diag refers to the diagonal of a matrix, as in *Diag*[*y*] := [*x*]. [*x*] is considered a linear array. The diagonal of [*y*] becomes the entries of [*x*]. If the name [*y*] does not yet exist, a new square matrix will be created with off-diagonal entries 0. If square matrix [*y*] of the right size (i.e., rows equal to the number of entries of [x]) does exist then the off-diagonal elements are not changed.

Dually, *Diag* can be used on the right side of an assignment, as in [*y*] := *Diag*[*x*], which sets [*y*] equal to a linear array consisting of the diagonal elements of [*x*]. [*x*] does not have to be square.

To create a diagonal matrix with all entries equal to a constant, say 1, you can use the easier form [*x*] := [1], if [*x*] already exists as a square matrix.

Cols$[x]$ = number of columns of array $[x]$.

Deg = number of elements in an array. *Deg*$[x]$ = total size of array $[x]$ (rows $\times$ columns).

$[<]$ = last computed array. Fermat has a hidden system array. If you type the command $[x] + [y]$, arrays $[x]$ and $[y]$ will be added and, since you didn't provide an assignment of the result, the result will go into the system array. You can later access it by typing, for example, $[z] := [z] + [<]$. Subarrays cannot be used with $<$.

_ = concatenate arrays; glue two arrays together to form a larger one, as in $[z] := [x]\_[y]$. Neither array can be *Sparse*.

Iszero = is the argument (an array) entirely 0? If so, return 1, else return 0. Syntax: *Iszero*$[x]$.

Switchrow = Interchange two rows in an array. Syntax: Switchrow$([x], n, m)$.

Switchcol = Interchange two columns in an array. Syntax: Switchcol$([x], n, m)$.

Normalize = convert to a diagonal matrix. The matrix must not be *Sparse*. If requested, Fermat will return the change of basis matrices used in normalizing. Possible invocations include *Normalize*$([x])$ and *Normalize*$([x], [a], [b], [c], [d])$. In the second case, matrices $[a]$, $[b]$, $[c]$, and $[d]$ will be returned that satisfy $[a] * [x'] * [b] = [x]$, where $[x']$ is the original $[x]$, and where $[c] = [a]^{-1}$ and $[d] = [b]^{-1}$. The value returned by *Normalize* is the rank of $[x]$.

You can omit any of the change of basis matrices. For example, *Normalize*$([x], , [b],,$ $[d])$ and *Normalize*$([x], [a], , [c])$. Every comma promises that an argument will eventually follow.

Colreduce = Column reduce a matrix. The matrix may NOT be *Sparse*. By column manipulations, the argument is converted to a lower triangular matrix. If requested, Fermat will also return the change of basis (or conversion) matrices that it used in normalizing. Possible invocations include *Colreduce*$([x])$ and *Colreduce*$([x], [a], [b], [c], [d])$. In the second case, matrices $[a]$, $[b]$, $[c]$, and $[d]$ will be returned that satisfy $[a] * [x'] * [b] = [x]$, where $[x']$ is the original $[x]$, and where $[c] = [a]^{-1}$ and $[d] = [b]^{-1}$. The value returned by *Colreduce* is the rank of $[x]$. As with *Normalize*, you can omit any of the change of basis matrices. *Colreduce* cannot be used on sparse arrays. In addition a function *Pseudet* is implemented. *Pseudet*$([x])$ computes a "pseudo-determinant," a nonzero determinant of a maximal rank submatrix. It returns the rank of the matrix and leaves the matrix in diagonal form (so $[x]$ is changed). The product of the diagonal entries is (up to sign) the "pseudo-determinant." The optional form *Pseudet*$([x], [rc])$ returns a $2 \times rank[x]$ matrix $[rc]$ specifying the rows (first row of $[rc]$) and columns (second row of $[rc]$) that constitute the maximal rank submatrix. A second optional form is *Pseudet*$([x], [rc], k)$ or *Pseudet*$([x], , k)$. Integer k specifies the last row/col to pivot on. The entries beyond spot $[k, k]$ are left.

Rowreduce = Row reduce a matrix. The matrix must be *Sparse*. Exactly like *Colreduce* but for sparse arrays and row reduction.

Smith = Put a matrix of integers into *Smith normal form*. The matrix may be *Sparse*. This function can only be used in rational mode, and assumes that every entry is an integer. (Any denominator encountered will be ignored, with unpredictable results.) By row and column manipulations, the argument is converted to a diagonal matrix of non-negative integers.

Furthermore, each integer on the diagonal divides all the following integers. The set of such integers is an invariant of the matrix.

As with *Normalize*, you can omit any of the change of basis matrices.

If you do not require any conversion matrices then it is possible to greatly speed up *Smith* in most cases by working modulo a "pseudo-determinant", a multiple of the gcd of the determinants of all the maximal rank minors (see Kannan and Backem, SIAM Journal of Computing vol 8, no. 4, Nov 1979). Do this in Fermat with the command *MSmith*. For relatively small matrices or sparse matrices, it's faster to forgo the modding out. Fermat will compute the pseudo-determinant if the matrix is *Sparse*. If you already have a pseudo-determinant *pd*, use the syntax $MSmith([x], pd)$. (If the matrix is not *Sparse*, you *must* use the latter method. *Pseudet* may be helpful.)

Hermite = Column reduce a matrix of integers. The matrix may be *Sparse*. This function can only be used in rational mode, and assumes that every entry is an integer. By column manipulations and row permutations, the argument is converted to a lower triangular matrix of integers. All diagonal entries are non-negative. This is often referred to as *Hermite normal form.*

If requested, Fermat will also return the integer change of basis (or conversion) matrices used in normalizing, exactly as in *Smith*.

Be aware that if the matrix is "large" and "dense" a horrendous explosion is possible in the intermediate entries, and in the entries of the conversion matrices.

Hampath = a pretty fast algorithm for finding a Hamiltonian path in a graph. The algorithm is from an exercise in a text book by Papadimitriou. Given the $n \times n$ adjacency matrix of a simple graph, the program constructs an $n^2+1 \times n^2+1$ sparse matrix. Each entry is either 0, 1, or a polynomial variable $x_i, i = 1, \ldots, n$. The graph has a Hamiltonian path iff the term $x_1 x_2 \cdots x_n$ appears in its determinant. We do not actually compute the determinant. If there is a Hamiltonian path, this method often proves it very quickly. *Hampath* returns 1 or 0 for path or no path.

The basic invocation is $Hampath[w]$ where $[w]$ is the adjacency matrix, symmetrical with each entry either 0 or 1. However, on any graph of more than 15 or so nodes, it is better to use the alternate form $Hampath([w], k)$, where k is a positive integer. k controls the probabalistic search by terminating a search tree once more than k leaves in the tree are visited with no resolution, and starting over. A reasonable heuristic for k is around $2n^2$.

Disclaimers: $Hampath$ does *not* first check elementary properties of a graph that might easily decide the issue, like connectivity. That's up to the user. I make no claims about efficiency relative to other methods.

Redrowech = the reduced row echelon form, for elementary matrix equations of the form $AX = B$. (Other Fermat commands do column manipulations as well, which could be used to solve $AX = B$ but take an extra step.) Invoke with $Redrowech([a])$, where all columns but the last in $[a]$ represent the matrix A and the last represents B (i.e., *Redrowech* never pivots on the last column.) Alternately, $Redrowech([a], [u], [v])$ will return in $[u]$ the transition matrix used in normalizing $[a]$. $[v]$ is $[u]^{-1}$. As in other similar Fermat commands, you can also do $Redrowech([a], , [v])$.

Minors: extract minors from sparse arrays. The syntax is e.g. $[y] := Minors([x], [r], [c])$. $[x]$ is an existing sparse array. $[r]$ and $[c]$ are existing ordinary arrays specifying the rows

and columns to be extracted. The result is stored in $[y]$, which will be a new sparse array of the right dimensions. $[x]$ is untouched.

FFLU and *FFLUC* are for fraction-free LU factorization of matrices. See the two articles in the September 1997 SIGSAM Bulletin: "Fraction-free Algorithms for Linear and Polynomial Equations," by Nakos, Turner, and Williams; and "The Turing Factorization of a Rectangular Matrix," by Corless and Jeffrey. See especially Theorem 4, p. 26 of the latter. $FFLU$ is invoked as: $FFLU([x],[p],[l],[a],[b])$. $[x]$ is the $n \times m$ matrix to be factored. Presumably $[x]$ contains integers or polynomials, but this is not enforced. $[p]$ is an $n \times n$ diagonal matrix consisting of the pivots used, $[p] = diag(p_1, p_2, ..., p_{n-1}, 1)$. $[l]$ is the unit lower triangular matrix, the first factor. $[a]$ (optional) is the $n \times n$ permutation matrix of row swaps. $[b]$ (optional) is $[a]^{-1}$. At the end, $[x]$ is in upper triangular form. Let $[z]$ be a copy of the original $[x]$. If $[f]$ and $[g]$ are the matrices called f_1 and f_2 in the Corless and Jeffrey article, then at the end one has $[f] * [a] * [z] = [l] * [g] * [x]$. Note that $[f]$ and $[g]$ are not computed by $FFLU$; however it is obvious how to get them from $[p]$. Note also that $[p]$ is not necessarily the diagonal of $[x]$: if columns of 0s are encountered along the way, $[x]$ will be in row-echelon form and may have 0s on its main diagonal.

$FFLUC$ allows column swaps as well as row swaps. In this way, the size of the pivots can be further reduced. $FFLUC$ is invoked as: $FFLUC([x],[p],[l],[a],[b],[c],[d])$. As above, $[x]$ is the $n \times m$ matrix to be factored. $[p], [l], [a]$, and $[b]$ are the same as above. At the end, $[x]$ is in upper triangular form. $[c]$ and $[d]$ (optional) are permutation matrices coming from column swaps ($[d]$ is $[c]^{-1}$). Let $[z]$ be a copy of the original $[x]$. If $[f]$ and $[g]$ are the matrices called f_1 and f_2 in the Corless and Jeffrey article, then at the end one has $[f] * [a] * [z] * [c] = [l] * [g] * [x]$. $[p], [l], [a]$, etc. need not be existing matrices when the function is invoked. Matrices of those names with the right size will be created at the end. Saying that $[a], [c]$, etc. are optional above means that they may be ommitted, as for example $FFLUC([x],[p],[l],[a],,[c])$. Note the space to indicate no $[b]$.

Reverse$[a]$ will reverse the elements in Array $a[n]$. If $[a]$ has one column, this is obvious. Otherwise, the exact behavior depends on whether $[a]$ is a standard array or a sparse array. For standard, each $a[i]$ is swapped with $a[n+1-i]$, where you think of $[a]$ as in column-major order. For sparse $[a]$, the rows are swapped.

Sort: sort an (already existing) array $[a]$ of polynomials, actually monomials. $[a]$ can be either sparse or ordinary. The first column holds the keys. For ordinary arrays, every entry in column 1 must have been assigned a value. 0 is a legal value. For sparse arrays, it is more subtle: every row must have an entry, and the first entry in each row is taken as the key. (Recall that sparse arrays never contain 0). To avoid confusion, the best policy is to have the first column contain the key. As swaps are made, the entire row is swapped. The algorithm is quicksort. Quolynomials are allowed; denominators are ignored.

The order is a monomial order, the one built into Fermat via the *Mono* command. In comparing a and b, only the leading monomials are compared - the ones you see first when they are displayed. Numerical coefficients are irrelevant. All numbers are considered equal. To illustrate, create a multivariate polynomial w, do $Mons(w, [a], 1)$, then $Sort[a]$. Syntax: *Sort*$[a]$.

Sparse Access Loops

There is a need for a way to work efficiently with sparse arrays. For example, suppose you have a sparse array of 60000 rows and 50000 columns with only 10 or so entries in each row (this is quite realistic). Suppose you wanted to add up all the entries. Naively, one could write something like:

$for\ i\ = 1, 60000\ do\ for\ j = 1, 50000\ do\ sum\ :=\ sum + x[i, j]\ od\ od$

But this will do 3,000,000,000 additions, almost all of which are adding 0! This is a preposterous waste of time. The solution is "sparse column access loops" for sparse arrays. The syntax is, continuing the example above,

$for\ i\ = 1, 60000\ do\ for\ j = [x]i\ do\ sum\ :=\ sum + x[i, j]\ od\ od.$

"*for j = [x]i do*" means find the i^{th} row of $[x]$ and let j run down it – of course encountering only the entries actually there! So j takes on whatever the column indices are in which $x[i, j] \neq 0$. $[x]$ must be an existing sparse array, and i must have a value suitable for $[x]$ at the start of the loop. More generally, one may use the syntax: *for j = [x]i,k do ...* . Here i and k both refer to rows of the sparse matrix $[x]$. At the start of the loop, all nonzero column coords in both rows are found. Then as the loop proceeds, j runs through those values in order. Any number of row indices is allowed. There is no analogous procedure for "sparse row loops" due to the way Fermat stores sparse matrices. If necessary, transpose the matrix.

Dixon Resultant Technique

To help compute the Dixon resultant, a feature has been added to *Det*. It may be used when a system of equations exhibiting symmetry leads to a determinant that is a polynomial in one variable, say t, over Z. Set the flags to select LaGrange interpolation. The syntax is $Det([m], dr, power1, power2, exp)$. $[m]$ is the matrix in question (must be *Sparse*).

Note: the second degree, power2, was added in February 2008.

Type 1: the determinant will be of the form $dr * f(t)^{exp}$, and of degree $power1$. dr is a known divisor of the determinant, a "spurious factor." Knowing dr, exp, and $power1$ enables Fermat to interpolate for f very efficiently. Type 2: the determinant will be of the form $dr * f(t^{exp})$, and of degree $power1$. Signal this type by setting exp to be negative; i.e. enter -5 instead of 5. exp must be either odd or a power of 2. $power2$ is the degree of the determinant in the second highest variable. Leave this field blank if there is no second variable, or you don't know the degree. For an introduction to the Dixon method, see: Lewis, R. H. and P. F. Stiller, "Solving the recognition problem for six lines using the Dixon resultant," Mathematics and Computers in Simulation 49 (1999) p. 205-219.

As of March 2007, $Det([m], dr, power1, power2, exp)$ will work recursively if more than one variable is present in array [m]. Note, however, that the exp field is ignored if $[m]$ is not Sparse.

Appendix 5. Finite Fields $GF(2^8)$ and $GF(2^{16})$

As of version 3.4.7 Fermat implements the finite field $GF(2^8)$ in an efficient way. The 256 elements are represented as bit strings, simply the integers 0 - 255 in one byte in the obvious way. These integers are mapped in 1-1 fashion to the AES-Rijndael implementation of this field, $Z/2\ [t]\ /\ <\ t^8 + t^4 + t^3 + t + 1\ >$ (see The Design of Rijndael: AES - The Advanced Encryption Standard, Information Security and Cryptography, by Joan Daemen and Vincent Rijmen.) The mapping is: f in $Z/2\ [t]\ /\ <\ t^8 + t^4 + t^3 + t + 1\ >$ becomes $f\#(t = 2)$; i.e. f evaluated at $t = 2$.

Elements are added by exclusive-or. They are multiplied and inverted by table lookup.

The field $GF(2^{16})$ is built on top of $GF(2^8)$ and is likewise just the first 65536 bit strings 0 - 65535. The first (lower) byte is $GF(2^8)$. Elements are added by exclusive-or. They are inverted by table lookup and multiplied by a table of logarithms. This is slightly slower than the table lookup of $GF(2^8)$.

To make these fields the ground ring in Fermat, simply do &p followed by 256 or 65536.

The problem for the user (speaking from my own experience) will be forgetting that there is no useful homomorphism from Z to these fields. One must be scrupulous in using &_m and _(...) to suppress modular!

Example:

```
BAD:  for i = 1, m do
        if n = i+1 then ....   ;; i+1 is NOT 1 more than i (!)
GOOD: for i = 1, m do
        if n = _(i+1) then ...

BAD:  rc := 1 + Sigma<j=1,n> [ Deg(za,j)*exp[j] ]
GOOD: rc := 1 + _Sigma<j=1,n> [ Deg(za,j)*exp[j] ]
```

Technically, over any modular ground ring, one should use the &_m or _(...) as above, but I at least usually use large primes p in Z/p, maybe 44449, so no harm results – usually. But over $GF(2^8)$ and $GF(2^{16})$, one needs constant vigilance. For example, 2 is not 2 (!)

Remember that modular is automatically suppressed in for-loops, exponents, and array indexes. So these should be OK:

```
for i = 1, 2n do...
x := y^(i+2);
s := c[2*n+1];
```

It is fine to attach poly vars on top of these ground fields. However, do not do &P on top of that. I have not thoroughly checked that.

Appendix 6. New Features Added Between June 2005 and April 2021

In roughly chronological order. Most of these features are described earlier in this manual too. The new parts are emphasized here.

Especially note monomial multiply and Zippel GCD.

&_s: Suppress/don't suppress display of long polynomials.

&_t: Toggle switch to turn on/off a certain fast probabalistic algorithm to test if one multivariate polynomial divides another over ground ring Z. Rarely, this technique can fail, in which case you will see a "Fermat error" about "number in trial_poly_divide". Then turn it off.

&_G: sort the heap garbage. This can be added to the user's functions periodically during memory intensive polynomial calculations. A noticeable speedup occurs when used between repetitions of an intensive calculation.

&v: List all current variables. ... Only about the first 150 lines of a large polynomial are shown, unless &_s has been set.

To help compute the Dixon resultant, a feature has been added to *Det*. It may be used when a system of equations exhibiting symmetry leads to a determinant that is a polynomial in one variable, say t, over Z. Set the flags to select LaGrange interpolation. The syntax is $Det([m], dr, power1, power2, exp)$. $[m]$ is the matrix in question (must be *Sparse*). ...

The new part is the second degree, power2.

As of March 2007, $Det([m], dr, power1, power2, exp)$ will work recursively if more than one variable is present in array [m]. Note, however, that the *exp* field is ignored if [m] is not Sparse.

STrans = transpose a matrix in place, as in $STrans[x]$. Much faster than *Trans*.

Isprime(n) = is n prime? 1 means n is prime, else it returns the smallest prime factor. n can be up to $2^{63} - 1$. The algorithm is elementary, so it's slow beyond 2^{50}.

Using &J adjoins new variables "above" the previous ones. However, as of January 2009, it is possible to adjoin a polynomial variable "at the bottom." So if, say, x and y exist, y later or "above" x, one can do &(J>z), which will insert z as the lowest variable (below x) rather than the highest. You cannot cancel from the botttom.

Lcoef = the leading coefficient of a polynomial.

Nlcoef = the leading numerical coefficient of a polynomial. Unlike *Lcoef*, always returns a number.

Ntcoef = the trailing numerical coefficient of a polynomial. That number is always an actual coefficient, so can never be 0.

Zncoef = the "last" numerical coefficient of a polynomial. It will be 0 if there is no constant term.

Pivot Strategies: As of January 2009 there are options for the heuristics that direct the pivot choice in the normalization of matrices. This can have a large effect on time and

space, though often it does not. The heuristic is set with the command &u or &(u = val). val is an integer from 0 - 5. The size or mass of a potential pivot can be measured by just its number of terms (called term# below), or by one of two mass heuristics (called mass0 and mass1 below).

Setting val =

0 is the default previous heuristic, which selects the least massive entry by the original mass0 heuristic.

1 selects the 'lightest' entry where weight = term# + sum of term# for the entire row entry is in.

2 selects the 'lightest' entry where weight = mass0 + sum of term# for the entire row entry is in.

3 selects the 'lightest' entry where weight = mass1 + sum of term# for the entire row entry is in.

4 selects the 'lightest' entry where weight = term#.

5 is like 3 but also counts the weight of the column an entry is in.

Termsf(x) = counts the total number of terms when polymodding to create a field, counting the field variables. If $s = (t+1)x + t - 1$ where t is a field variable, *Termsf(s)* = 4 while *Terms(s)* = 2.

As of October 2009, the LaGrange modular determinant coefficients can be dumped to a file, rather than stored in RAM. This can be a big space saving when doing a very large computation. The command is &(_L=1). In other words, if the highest precedence variable is x and lower ones are $y, z, \ldots$, and if a determinant $c_n x^n + c_{n-1} x^{n-1} + \ldots$ is being computed with LaGrange interpolation, the coefficients $c_0, c_1, ..., c_n$ (which are polynomials in $y, z, \ldots$) will be dumped to the output file to save RAM.

As a time- and space-saving aid, one can add the ^ when saving, as in

!!(&o, 'q := ', ^q, ';'). Without the ^, q is duplicated in the course of expression evaluation. That might be a big waste of time or space.

The display of elapsed time changed in 2010. When timing is enabled, two numbers are displayed, called "Elapsed CPU time" and "Elapsed real time". CPU time is just the CPU time used by Fermat. This is what has been displayed by Fermat in most previous versions. However, the number is meaningful only up to about an hour. For much longer times, the value shown is meaningless. Elapsed real time is wall clock time, just as it sounds. If the elapsed real time is more than 5 seconds, then it is also displayed.

Sort: sort an (already existing) array [a] of polynomials, actually monomials. [a] can be either sparse or ordinary. The first column holds the keys. For ordinary arrays, every entry in column 1 must have been assigned a value. 0 is a legal value. For sparse arrays, it is more subtle: every row must have an entry, and the first entry in each row is taken as the key. (Recall that sparse arrays never contain 0). To avoid confusion, the best policy is to have the first column contain the key. As swaps are made, the entire row is swapped. The algorithm is quicksort. Quolynomials are allowed; denominators are ignored.

The order is a monomial order, the one built into Fermat via the *Mono* command. In comparing a and b, only the leading monomials are compared - the ones you see first when they are displayed. Numerical coefficients are irrelevant. All numbers are considered equal.

To illustrate, create a multivariate polynomial w, do $Mons(w,[a],1)$, then $Sort[a]$. Syntax: $Sort[a]$.

Mono(a,b) to compare monomials; returns true iff $a \leq b$. Numerical coefficients are irrelevant. Individual variables are ranked by order of their creation, latest is largest. Monomials are ranked first by length = #variables, i.e. all univariates < all bivariates < all trivariates, ... etc. Within equal #vars, ranking is by subset of vars. Within subset, ranking is exponent arrays. See "Sort" below. This is the order that is used internally by Fermat for certain algorithms.

Splice: $Splice(s,c,m)$ multiplies c by x^m and "tacks it in front" of s; it adds them, if there is overlap. c becomes 1, s gets the answer. see *Split*.

Split is basically the dual of *Splice*. Invoke with $Split(w,n,v)$ or $Split(w,n)$. w and v are existing scalar variables (including entries in an array). n is an integer. w becomes w mod x^n, where x is the highest variable. v (if present, optional) becomes $w\backslash x^n$.

Time displays the time and date. Visible on startup.

Vars(x) = number of variables that actually occur in x.

SDet = "Space-saving determinant." Space is saved when computing over ground ring Z using LaGrange interpolation and the Chinese Remainder Theorem. Fermat has aways used the the CRT algorithm from Knuth volume 2 on page 277 (exercise 7). The advantage is one can do everything "on the fly." The disadvantage is that if one will need, say, 50 primes, then every determinant modulo the 50 primes is stored until the end, when the answer is computed over Z by combining them. That might need too much space. Instead, $SDet$ implements formulas 7-9 on page 270 of Knuth.

The disadvantage is it can't be done on the fly: one needs to know in advance how many primes will be needed. The user must (over)estimate this number.

Input: integer and a square matrix of polynomials. Output: determinant. Call: $SDet(n,[m])$. n = how many primes will be needed. $SDet$ is not guaranteed to work with the more sophisticated options of Det, i.e. $Det([m4],r,d1,d2)$.

Toot: sound the system beep.

As of 2010, a new arithmetic command has been added for when a function is in Integer mode: $x :* y$ to set $x := x * y$; NB: this only works in Integer.

Reverse$[a]$ will reverse the elements in Array $a[n]$. If $[a]$ has one column, this is obvious. Otherwise, the exact behavior depends on whether $[a]$ is a standard array or a sparse array. For standard, each $a[i]$ is swapped with $a[n+1-i]$, where you think of $[a]$ as in column-major order. For sparse $[a]$, the rows are swapped.

Poly$[a]$: With x = the highest polynomial variable, Poly[a] takes an array **of numbers** and yields $\Sigma < i = 1,n > [\, a[i]\, x^{n+1-i}\,]$. At least that is what happens on a standard array. For a sparse array, only the first entry in each row is used. So again, if $[a]$ has one column *Poly* produces what you'd think.

400,000 pointers have been set aside for the array of arrays, up from the previous 2000. It is now possible to assign array of arrays pointers, as %[1] := %[2]. This is good for swapping data. One can also do $[c]$:= [%[1]].

The action of *Totdeg* has changed. New *WDeg* performs the old *Totdeg*. See Appendix Four.

Monomial-oriented multiplication: October 2013. Version 5.0 of 64 bit Fermat implements a new method for multiplication of multivariate polynomials. This provides very impressive speedups in real problems, often 40 - 70%, even more in some test cases. The method can be modified by the user command &_o.

Basically, the idea is to store each term, or monomial, of a multivariate polynomial in a single node instead of storing the polynomial as a recursive structure of nodes at one level pointing to nodes at a lower level.

For example, consider $(x^2+y^2+z^3+1)^3$. This polynomial has 20 terms. As a recursive list structure it is $z^9 + (3y^2 + 3x^2 + 3)z^6 + (3y^4 + (6x^2 + 6)y^2 + 3x^4 + 6x^2 + 3)z^3 + y^6 + (3x^2 + 3)y^4 + (3x^4 + 6x^2 + 3)y^2 + x^6 + 3x^4 + 3x^2 + 1$. This means that at the highest level, z, there are four nodes, one each for z^9, z^6, z^3, and z^0. Each node has a pointer to its coefficient, a polynomial at the next level down, for y. The coefficient of z^6 is the polynomial $3y^2 + 3x^2 + 3$. That has two nodes, one for y^2 and for one for y^0. The coefficient of y^0 is the polynomial $3x^2 + 3$, which is one more level down, and so forth. (Eventually there is a pointer for the numerical coefficient, stored in a different kind of node.) This has always been the storage structure of polynomials in Fermat. It has many advantages, one of which is that any exponent may be easily accommodated. Another is that multiplication is easily written as a recursive program; each level is handled in the same way as all others.

However, the disadvantage of the recursive structure is that there are many links to follow, and much space is devoted to storing these links.

In contrast, the monomial structure stores the above example as a list of twenty nodes, each of which contains all the exponents for that term. The polynomial is thought of as $z^9 + 3y^2z^6 + 3x^2z^6 + 3z^6 + 3y^4z^3 + 6x^2y^2z^3 + 6y^2z^3 + 3x^4z^3 + 6x^2z^3 + 3z^3 + y^6 + 3x^2y^4 + 3y^4 + 3x^4y^2 + 6x^2y^2 + 3y^2 + x^6 + 3x^4 + 3x^2 + 1$. For example, $6x^2y^2z^3$ is a node containing a pointer to the numerical coefficient, then the three exponents 2 2 3. These are stored as fields within a single long integer. For example, if one knew that only three-variable polynomials would ever be considered, one could use a 32 bit integer to store all three, with the exponent for x (lowest precedence) going in spots 0-9, for y in spots 10-19, and for z in spots 20-29. The upper two spots 30 and 31 would be unused.

The advantage is that multiplication of terms is now very easy – just add the nodes containing the three exponents, a single 32 bit integer addition! (Of course, one multiplies the coefficients too.) This is fine as long as one does not encounter an exponent as large as $2^{10} = 1024$. Then disastrous overflow would occur, perhaps crashing Fermat, or worse, introducing undetected nonsense.

In Fermat we expect to use far more than three polynomial variables. I have therefore implemented this idea with 128 bit long long integers instead of 32 bit integers. The user may choose the "monolength", the number of bits allocated to each variable's exponent (it was 10 in the example above) with the command &_o. Fermat will then compute and display monovars = 128 div monolength, mononum = $2^{monolength}$, and monogap = 128 - monolength * monovars. The default monolength is 9.

If after the &_o prompt you enter a value > 30 or < 5, the monomial multiplication method is disabled; multiplication will proceed recursively. Since legal values are between 5 and 30, possible monovars are between 25 and 4. It is probably unwise to use a monolength < 7, so the most polynomial variables that can reasonably be multiplied by this method is

18. Therefore, one may well have a situation where one has more polynomial variables than mononvars. This is fine: Fermat begins the standard, recursive method, and switches when the depth of recursion allows it (depth remaining $\leq$ monovars) for multiplying coefficients.

There is no guarantee against overflow, other than the wisdom of the user. If overflow is suspected, there will be warning messages displayed. If enough messages occur, an error occurs.

There is no change in addition or subtraction of polynomials. The user's variables are still stored in the recursive structure. When the need to multiply arises, they are converted to the new format, multiplication is done, and the answer converted to the recursive form for storage.

What is all this for? Speed! Unless one has only two variables, or one of the multiplicands has few terms, say < 60, a noticeable improvement in speed results. On real problems taking many minutes to complete involving thousands of multiplications of polynomials with six or more variables, I routinely see 30 - 80% improvements.

Zippel-like GCD algorithm: As of early 2017, Fermat 6.* implements a Zippel-like interpolation algorithm for multivariate polynomial GCD of six or more variables. (See R. Zippel. Interpolating Polynomials from their Values. J. Symbolic Comp. 9(3), 375-403, 1990.) If you do not have at least six variables, you will not see any change.

Years ago, Fermat was the fastest CAS for polynomial arithmetic. One reason for this is the polynomial GCD algorithms. However, despite continual improvements in those methods, it became clear by 2006 that a better method for problems involving more than four variables was often required. Fermat 6.0 provides that method. Some problems that took several days on previous versions of Fermat now complete in less than a minute.

Magma, a well-respected CAS, has long been considered to have very good polynomial arithmetic. <a href="http://home.bway.net/lewis/fermagcomp.html"> is a suite of problems arising from actual applications that compares the new Fermat 6.0 with Magma. On the whole, it would seem from these tests that Fermat is a bit better, especially with large problems that take more than two minutes.

The new algorithm in Fermat applies to ground ring Z or Z/p for prime p. However, p should be "fairly large". In my own work I never use primes less than 20000 and often use $p = 2^{31} - 19$. I have tested the new method on primes as small as 2003 and it worked fine. On $p = 181$ it worked but had to restart itself at least once. (It still took only one fifth the time of Fermat 5.25.) It will certainly fail on much smaller primes, as it needs a certain amount of "room" to succeed.

The method can be turned off with the new toggle command &z. It is on by default.

Subarrays: As of April 2016, in 64 bit Fermat, subarray can be used with Sparse matrices in assignments and with %.

However, do not write expressions that mix subarrays of sparse and ordinary matrices! No guarantees there.

Times and Timing: March 2021. There are four different functions related to time.

Timecpu displays the total amount of time Fermat has used since startup. It returns 0. It's just a display. *Time* displays the date and time. It returns 0. It's just a display. &T returns the elapsed time in milliseconds since startup. Eventually it will hit integer over-

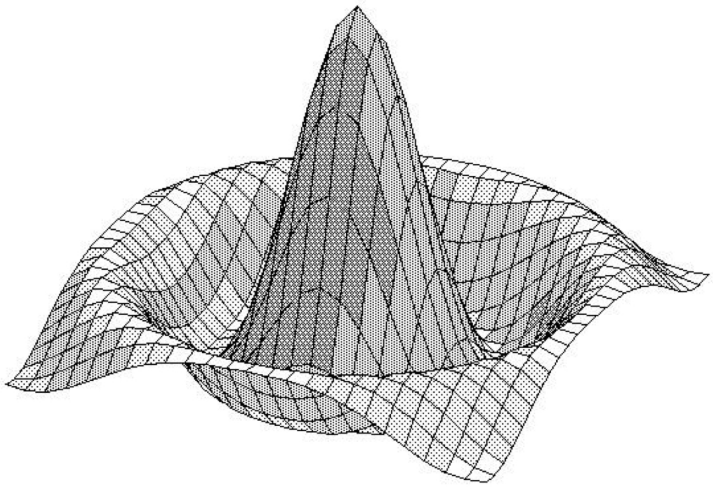
flow, at around 4470 minutes. &^ returns the elapsed time in microseconds since startup. Eventually it will hit integer overflow, at around 268 seconds.

The latter two can be used to time how fast your functions execute.

Determinant Cutoff, Sparse: April 2021. The command &D has been described before. With ordinary arrays, &(D=n) sets Fermat to run Gaussian elimination until there are n rows left, then finish with expansion by minors. With a sparse array, until April 2021 the command had no effect; Gaussian elimination would go all the way. Now, Gaussian elimination will proceed through the n^{th} row, then stop, returning the determinant so far computed (product of diagonal terms). The final $n-1$ rows and columns will be left in whatever state Gaussian elimination has produced. To finish the computation, extract the $n-1$ rows and columns in the lower right and compute their determinant. Multiply that by the earlier result.

Monomial-oriented multiply: Revised slightly June 2021. It is now turned off for three variable polynomials, and for four variable polynomials with rational ground field.

If you use cntl-C to interrupt a computation with mono-multiply, then do a panic stop back to the top level, the internal data structures for mono-multiply will probably be unusable. Reset them with the command *Xmono*.



The exponential-cosine sombrero.

Index