Summary of Fermat

Robert H. Lewis

http://www.bway.net/~lewis/

Fermat is an interactive system for mathematical experimentation. It is a super calculator -- computer algebra system, in which the basic items being computed can be rational numbers, real numbers, complex numbers, elements of finite fields, multivariable polynomials, multivariable rational functions, or multivariable polynomials modulo other polynomials. Fermat is available for Mac OS, Windows, Unix, and Linux. It is shareware.

In Fermat, the basic ground ring is *Z* (and its field of fractions *Q*). One may choose to work modulo a specified integer *n*, thereby in effect changing the *ground field* (or *ground ring*) *F* from *Q* to *Z*/*n*. The ground rings GF(2^8) and GF(2^16) are also efficiently implemented. On top of this may be attached any number of unevaluated variables t_1, t_2, ..., t_n, thereby creating the polynomial ring $F[t_1, t_2, ..., t_n]$ and its quotient field, the field of rational functions, whose elements are called *quolynomials*. Further, polynomials *p*, *q*,... can be chosen to mod out with, creating the quotient ring $F(t_1, t_2, ...) \;/ < p, q, ... >$, 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, and matrix inverse. It is consistently faster than some well known computer algebra systems - orders of magnitude faster in many cases.

Fermat has a complete programming language. Programs and data can be saved to an ordinary text file, read during a later session, or read by some other software system.

Fermat has several features that enhance debugging. It is possible to interrupt Fermat. 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.

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* > . (< *return* > means press the return key.)

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, until a later result supplants it.
Computed values and function definitions can be saved from one session to the next. They are stored in an ordinary text file that can be edited with an ordinary editor.

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* ` on the end of each line (except the last, of course).

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)
```

to open the file for reading and bring the data into this session. Suppose there is a 3 X 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]           [   and 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 X 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 quickly compute characteristic polynomials in Fermat.)

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 $Z[t]$ to $Z[t]/ < t^2+1 >$. 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
```

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 in the full manual, on line at
http://www.bway.net/~lewis
We do not discuss here programming in Fermat.

Polynomials

Recall that most built-in functions allow *call-by-reference* for parameters. For example, time and compare *Terms*(x) and *Terms(^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 $t^n + c_1 t^{n-1}$ + ...

However, Fermat is best at the case where the quotient ring becomes a field (note well, $Q[t, u, ...] \ / < p, q,... >$ is a field, $Z[t, u, ...] \ / < p, q, ... >$ is not). Specifically, suppose the polynomial variables have been attached in the temporal sequence $t, u, v, ...$ Begin by modding out a monic irreducible polynomial $p(t)$ such that $F_1 = Z[t]/ < p >$ is an integral domain, and its field of fractions is $Q[t]/ < p >$. 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 = $t^n + c_1 t^{n-1}$ + ..., 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.

In all of the above the base field is Q. To make a finite field, first do the above then go into modular mode with &(p = some prime n), where the prime n and the polynomials $p, q, ...$ have been chosen so that $Z/n[t, u, ...] \ / < p, q, ... >$ is a field.

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: +, -, *, /, |, \, ^. 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 $c^k p = y q + r$, where 0 £ k £ $deg(p) + deg(q)$. 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#*$(v_1, v_2, ...)$, 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 much 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. One can also evaluate at an array of values, via $q\#(d = [x])$

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, ...) = 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. The result is clearly a monomial. *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 $(t^2 + 3t + 5)u^2$. This allows one to have lower variables of a different " status," as if the lower variables are "parameters" and the upper ones are the "real" variables.

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 (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.

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)*.

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

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 $x[i]$.

&_l: Display each polynomial as a list of monomials.

&c: Enable full Hensel checking; rather 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, not complex, the result is normalized to have all coefficients integral and be of content 1. In cases where the ground ring is a field, the result has leading coefficient 1.

Content = *content* of a polynomial; i.e., the GCD of all its coefficients.

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.

SDivides(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.

Totdeg(x, [a], n) returns 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. &oe is useful.

Factoring Polynomials: 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 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. There is some randomness in the algorithms, so the time it takes to factor can vary.

Matrices

Creation: An n X 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 X 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*. The only size limitation in Fermat is that the number of rows and the number of columns must each be less than 2^{28}. 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 $[z] := [x]$^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 ~ 4, 2 ~ 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 X 5 and $[c]$ is at least 4X 6. (Here ~ 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~ 6]] : = [a] is equivalent to the above. Similarly, one can use expressions like [c[3~ , 2~ 6]] : = [a] or [c[~ 4, 2~ 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., $a[5,1]$.
Subarray cannot be used with *Sparse* matrices. But see *Minors* 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 five basic methods to compute determinant: expansion by minors, Gaussian elimination, Gauss-Bareiss, LaGrange initerpolation, 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 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 choose among the methods, but the user may override them and force a particular method. Assuming an *m* X *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* X *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*. 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. The determinants in question are simply $f(c_i)$, where f is the characteristic polynomial and $\{ c_i \}$ is a set of " 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 $|f(t)| <= |g(t)|$ for all t. The syntax is *Chpoly*([x], g).

Minpoly: The "modifed Mills method," a 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. 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, analagously to *Powermod*. [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."

Trans = transpose matrix, as in [y] : = *Trans*[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 X columns).

_ = 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 (but see Rowreduce below). In addition a function *Pseudet* is implemented. *Pseudet*([*x*]) computes (indirectly) 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 X *rank*[*x*] matrix [*rc*] specifying the rows (first row of [*rc*]) and columns (second row of [*rc*])that constitute the maximal rank submatrix.

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*. 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 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*.

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.) Altnately, *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.

FFLU is invoked as: *FFLU*([*x*], [*p*], [*l*], [*a*], [*b*]). [*x*] is the n X m matrix to be factored. [*p*] is an n X 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 X 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 reduced. *FFLUC* is invoked as: *FFLUC*([*x*], [*p*], [*l*], [*a*], [*b*], [*c*], [*d*]). As above, [*x*] is the n X 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, for example *FFLUC*([*x*], [*p*], [*l*], [*a*], , [*c*]). Note the space to indicate no [*b*].

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 ith 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] <> 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.