1 Installation

Insure that you have installed GMP, FLTK, optionnaly GSL, NTL and PARI and you that have a recent version of `gcc` (e.g. 2.95, 2.96, note that GCC 3.0 will compile GMP but not FLTK AFAIK). Then :

```
tar xvfz giac-0.2.2.tar.gz
cd giac-0.2.2
./configure --enable-fltk-support --enable-debug-support
make
```

Now become `root`:

```
su
```

and eventually :

```
make install
```

2 Using the `cas` command.

You can invoke `cas` directly from the command line (in an `xterm` window for example). But (except for very simple input) you must quote the arguments so that the shell does not interpret badly parentheses or `*`, ... For example :

```
cas 'factor(x^3-1)'
```

Or you can call `factor` as a command, like :

```
factor x^3-1
```

Note that you don't require quote here since argument can not be interpreted by the shell.

You can also put a filename instead of an argument. Then all commands in this filename will be executed. For example, using your favorite editor (e.g. emacs) create a file named `test` containing:

```
factor(x^100-1);
rref([[1,2,3],[4,5,6]])
```

(note that you don't have to quote inside a file) and run :

```
cas test
```

If you want to assign variables a value, just make a file having the variable name with the value of the variable in the file. For example, edit a file named `mat`, write `[[1,2,3],[4,5,6]]`, save the file and try one of the following command :

```
cas 'ker(mat)'
ker mat
```

You can store the result of a `cas` command using the usual redirection symbol, for example :

`ker mat > kermat` will create a file named `kermat` that you can use as a variable name later. You can also pipe the result of a command as argument of another command, e.g. :

```
gcd x^4-1 x^6-1 | factor
```

The syntax is as similar as possible to usual CAS (especially HP49/40G CAS). Vectors are delimited by `[ ]` and coordinates are separated by `,`. Matrices are vectors of vectors.

If you want to know how much time was needed to evaluate your cas query, just define the environment variable `SHOW_TIME`, if your shell is `tcsh` :
`setenv SHOW_TIME 1`
will define the environment variable and
`unsetenv SHOW_TIME`
will undefine it. With `bash`,
`export SHOW_TIME=1`
defines the variable and
`unset SHOW_TIME`
undefines the variable.

Currently implemented :

1. usual arithmetic on integers, reals, complex, vectors and matrices: `abs`, `arg`, `conj`, `evalf`, `im`, `inv`, `max`, `min`, `re`, `sign`, `sqrt`. For the usual operations, since `*` is interpreted by the shell, you must quote `'*'` or escape `\*` the multiplication symbol. For division, I use currently the quoted or escaped symbol `%`, this is likely to change.

2. more advanced arithmetic: `cyclotomic`, `egcd`, `gcd`, `ichinrem`, `iquo`, `irem`, `is_prime` (returns 2 if certainly prime, 0 if not prime, 1 if probably prime), `nextprime`, `prevprime`, `jacobi`, `legendre`, `smod`.

3. transcendental functions: `acos`, `acosh`, `alog`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `exp`, `Log`, `log10`, `sin`, `sinh`, `tan`, `tanh`.

4. polynomial functions: `normal`: rational simplification
   `factor`: factorization over the integers or Gauß integers
   `partfrac`: partial fraction expansion
   `resultant`: resultant of 2 polynomials
   `solve`: solving polynomial-like equations

5. rewriting functions: `fdistrib` (full distribute $\times$ over $+$)
   `simplify`: currently rational simplification only
   `texpand`, `tlin`: trigonometric expansion and linearization

6. calculus: `derive`: derivation
   `lim`, `series` : limits and series expansion
   `integrate`: integration of rational fractions

7. linear algebra: `rref`, `ker`, `image`, `det`, `pcar`, `trace`, `tran`, `egv`, `egvl`, `jordan`.

8. conversion functions: `e2r` (entier to rational) and `r2e` (rational to entier). They expect a list of variables with respect to which the expression should be a rational fraction, you can use `lname` or `lvar` to get this list. The

internal format for rational fractions is parsed directly from the command line:

- integers, Gauß integers: usual notation (`2`, `3-5*i`)
- dense univariate polynomial: like a vector with coefficients by descending power (`[1,2,3]` for x^2+2x+3)
- sparse univariate polynomial and series expansion: a sum of monomials, each monomial is a couple of coefficient and exponent separated by `,`, e.g. `{1,1/2}+{2,3}` for $x^{1/2}+2x^3$. For a series expansion, use `undef` as coefficient for the remainder term.
- sparse multivariate polynomials: same notation, but the second term of the couple is a vector of indices, the powers of the variables in the monomial, e.g. `{ 2,[1,3]}` , for the 2-d polynomial $2xy^3$ with respect to the list of variables $[x,y]$.
- (internal) algebraic extension objects. Similar notation, but use `:` instead of `,`. The first term of the couple is a polynomial with respect to an algebraic integer θ, defined by the second term of the couple. This second term might be the minimal polynomial of θ or another extension with as first term an approximate value or an index and second term the minimal polynomial in order to differentiate the different roots of the minimal polynomial. The minimal polynomial is a dense univariate polynomial. For second order extension, there are only two models of monic minimal polynomials used: x^2-d if $d \neq 1 \pmod 4$ and $x^2-x=\frac{d-1}{4}$ otherwise and by convention $\theta=\sqrt{d}$ in the first case or $\theta=\frac{1+\sqrt{d}}{2}$. This to insure that every monic polynomial of second order can be factored over such an extension without introducing fractions.
- Fractions: using the `/` division sign.

A somewhat more complex example :

```
( sin x ; tan x ) | \% | lim
```

We first compute $\sin(x)$ and $\tan(x)$, then we pipe both answer to the division function and pipe the result to the limit function. This is equivalent to :

```
lim 'sin(x)/tan(x)'
```

but it demonstrates how it is possible to build expressions using the shell syntax: the shell is used as a polish notation calculator with mixed syntax (infix for sin and tan, reverse polish notation when we pipe). This gives some flexibility to make small programs using the shell.

A final example: open a file `testjordan` and write :

```
[[1,1,-1,2,-1],\
 [2,0,1,-4,-1],\
 [0,1,1,1,1],\
 [0,1,2,0,1],\
 [0,0,-3,3,-1]]
```

then write the command :
`( jordan testjordan ; cas p j ) | sto`
or `cas 'sto(jordan(testjordan),[p,j])'` that will compute the Jordan decomposition of the matrix and store the passage matrix in `p` and the Jordan normal form in `j`. If you want to check that pjp^{-1} is the original matrix, you can write the following command :
`cas p j 'inv(p)' | \* | normal`
or `cas 'normal(p*j*inv(p))'`

3 TEX translation

The LaTeX translation of the commands are logged in the file `session.tex` in the current directory except when you set the `SHOW_TIME` environment variables. They require a preamble that you can copy from the file `doc/preamble.tex`.

Note that every new command is appended to `session.tex`, it is a good idea to remove this file from time to time.

You can translate formulas in LaTeX using the `cas2tex` command and get directly a compilable LaTeX source file. For example :
`cas2tex '[[1,2],[3,4]]' > essai.tex`
followed by :
`latex essai.tex`
Or in one step :
`cas2tex '[[1,2],[3,4]]' | latex --`
(this produces the file `texput.dvi`)

4 Programming in C++

First example :

```
#include <giac/giac.h>
using namespace std;
using namespace giac;

int main(){
  gen e(string("x^2-1"));
  cout << factor(e) << endl;
}
```

Write this as `essai.cc` and compile it :
`g++ -g essai.cc -lgiac -lgmp`
and run it :
`./a.out`

4.1 Organization of the source code

Note that you can use `#include <giac/giac.h>` to include all headers of the library.

- `gen.cc/.h`: arithmetic operations on the base class entier
- `identificateur.cc/.h`: global name
- `unary.cc/.h`: unary operators class including non unary operations viewed as unary operation on the vector of it's arguments
- `symbolic.cc/.h`: symbolic object class
- `usual.cc/.h` Usual unary operations
- `vecteur.cc/.h`: linear algebra
- `derive.cc intg.cc lin.cc series.cc subst.cc/.h`: Calculus. Derive is OK as well as rational fraction integration, the rest has to be implemented
- `moyal.cc/.h`: pseudo-diff operators
- `tex.cc/.h`: LaTeX conversion
- `sym2poly.cc/.h`: conversion polynomials to symbolic
- `index.cc/index.h`: class for indexation of multivariate tensors
- `poly.h, monomial.h`: multivariate template class tensors
- `gausspol.cc/.h`: specialization of template tensors to entier coeffs
- `input_parser.yy input_lexer.ll input_lexer.h`: parser
- `modpoly.cc/.h`: univariate dense polynomials over integers and modular integers (Warning: gcd-like functions do not work if non modular arithmetic)
- `modfactor.cc/.h`: factorization of univariate dense square-free polynomials (Requires NTL or PARI for lll and knapsack to be full speed functional)
- `series.cc/.h`: series expansion and limits using the mrv algorithm.

See `giac.texinfo` for a short description of the classes available. Some examples of programs are provided: `src/factor.cc`, `src/normalize.cc`, `src/cas.cc`, `src/partfrac.cc` and `src/integrate.cc`. To compile these programs, you can either use :
`g++ -g` *name.cc* `-lgmp -lgiac`
or launch `emacs` and run it's compile command (menu Tools).