1 Instalación

Asegúrese que tienes instalado GMP, FLTK, y opcionalmente GSL, NTL y PARI y una versión reciente de `gcc` (p.e. 2.95, 2.96, notése que GCC 3.0 compilará GMP pero no FLTK). Entonces :

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

Ahora entra como `root`:

```
su
```

y eventualmente :

```
make install
```

2 Usando el comando `cas`

Puedes invocar `cas` directamente desde la línea de comandos (en una ventana `xterm` por ejemplo). Pero (excepto para entradas muy simples) debes entrecomillar `' '` los argumentos, así el shell no malinterpretará los paréntesis o `*`, ... Por ejemplo : `cas 'factor(x^3-1)'`

O puedes llamar `factor` como un comando, como :

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

Nótese que no necesitas entrecomillar aquí puesto que el argumento no puede ser interpretado por el shell.

También puedes poner un nombre de fichero como argumento. Entonces todos los comandos en este fichero serán ejecutados. Por ejemplo, usando tu editor favorito (p.e. emacs) crea un fichero llamado `test` conteniendo:

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

(nótese que no tienes que entrecomillar dentro de un fichero) y ejecuta :

```
cas test
```

Si quieres asignar valores a variables, crea un fichero con el nombre de la variable que contenga el valor de la misma. Por ejemplo, edita un fichero llamado `mat`, escribe `[[1,2,3],[4,5,6]]`, guarda el fichero e intenta el comando siguiente :

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

Puedes guardar el resultado de un comando `cas` usando el símbolo de redirección habitual, por ejemplo :

`ker mat > kermat` creará un fichero llamado `kermat` que podrás usar como nombre de variable más adelante. También puedes pasar el resultado de un comando como argumento de otro comando, p.e. :

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

La sintásis es similar a habituales CAS (especialmente HP49/40G CAS). Los vectores están delimitados por `[ ]` y las coordenadas están separadas por `,`. Las matrices son vectores de vectores.

Si quieres saber cuanto tiempo ha sido necesario para evaluar tu orden, define la variable de entorno `SHOW_TIME`, si tu shell es `tcsh` :
`setenv SHOW_TIME 1`
definirá la variable de entorno y
`unsetenv SHOW_TIME`
la indefinirá. Con `bash`,
`export SHOW_TIME=1`
defines la variable y
`unset SHOW_TIME`
indefines la variable.

Actualmente implementado :

1. aritmética habitual en enteros, reales, complejos, vectores y matrices: `abs`, `arg`, `conj`, `evalf`, `im`, `inv`, `max`, `min`, `re`, `sign`, `sqrt`. Para las operaciones habituales, puesto que `*` es interpretado por el shell, debes entrecomillar `'*'` o escape `\*` el símbolo de la multiplicación. Para la división, uso actualmente el símbolo entrecomillado o escape `%`, esto cambiará.

2. aritmética más avanzada: `cyclotomic`, `egcd`, `gcd`, `ichinrem`, `iquo`, `irem`, `is_prime` (devuelve 2 si es seguro primo, 0 si no es primo, 1 si es probablemente primo), `nextprime`, `prevprime`, `jacobi`, `legendre`, `smod`.

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

4. funciones polinómicas: `normal`: simplificación racional
`factor`: factorización de enteros o enteros de Gauß
`partfrac`: expansión parcial de fracciones
`resultant`: resultante de 2 polinómios
`solve`: resolver ecuaciones polinómicas

5. re-escribiendo funciones: `fdistrib` (completa distribución de $\times$ sobre la $+$)
`simplify`: actualmente sólo simplificación racional
`texpand`, `tlin`: expansión trigonométrica y linealización

6. cálculo: `derive`: derivación
`lim`, `series` : límites y desarrollo en serie
`integrate`: integración de fracciones racionales

7. álgebra lineal: `rref`, `ker`, `image`, `det`, `pcar`, `trace`, `tran`, `egv`, `egvl`, `jordan`.

8. funciones de conversión: `e2r` (entero a racional) y `r2e` (racional a entero). Esperan una lista de variables con respecto a las cuales la expresión debería ser una fracción racional, puedes usar `lname` o `lvar` para obtener esta lista.

El formato interno para fracciones racionales es interpretado directamente de la línea de comandos:

- enteros, enteros de Gauß : notación habitual (`2`, `3-5*i`)
- polinómios densos univariables : como un vector con coeficientes en orden descendiente de potencias (`[1,2,3]` para $x^2 + 2x + 3$)
- polinómios univariables en potencias racionales y desarrolos en serie: una suma de monomios, cada monomio es un par con coeficiente y exponente separados por `,`, p.e. `{1,1/2}+{2,3}` para $x^{1/2}+2x^3$. Para un desarrollo en seire, usa `undef` como coeficiente para el término restante.
- polinómios multivariables en potencias racionales: misma notación, pero el segundo término del par es un vector de índices, las potencias de las variables en el monomio, p.e. `{ 2,[1,3]}` , para el polinómio $2xy^3$ respecto a la lista de variables $[x, y]$.
- (interna) extensión de objectos algebraicos . Notación similar, pero usa `:` en vez de `,`. El primer término de la pareja es un polinómio respecto a un entero algebraico θ, definido por el segundo término del par. Este segundo término puede ser el polinómio mínimo de θ u otra extensión con como el primer término un valor aproximado o un índice y segundo término el polinómio mínimo en orden para diferenciar las diferentes raíces del polinómio mínimo. El polinómio mínimo es una polinómio denso univariable. Para extensiones de segundo orden, sólo hay usados dos modelos de mínimo mómio: x^2-d if $d \neq 1 \pmod 4$ y $x^2 - x = \frac{d-1}{4}$ Si no, y por convenio $\theta = \sqrt{d}$ en el primer caso o $\theta = \frac{1+\sqrt{d}}{2}$. Ésto es para asegurar que cada monómio de segundo orden puede ser factorizado sobre una extensión sin introducir fracciones.
- Fracciones: usando el signo de división `/` .

Un ejemplo algo más complicado :

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

Primero calculamos $\sin(x)$ y $\tan(x)$, después pasamos ambas respuestas a la funciíon división, y pasamos el resultado a la función límite. Ésto es equivalente a :

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

pero demuestra cómo es posible construir expresiones usando la sintásis del shell: el shell es usado como una calculadora con notación polaca inversa con sintásis mixta (infija para sin y tan, notación polaca inversa cuando pasamos resultados). Ésto da más flexibilidad para hacer pequeños programas usando el shell.

Un ejemplo final: abre un fichero `testjordan` y escribe :

```
[[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]]
```

después escribe el comando :
`( jordan testjordan ; cas p j ) | sto`
o `cas 'sto(jordan(testjordan),[p,j])'` que calculará la descomposición de Jordan de la matriz y guardará la matriz de paso en `p` y la forma normal de Jordan en `j`. Si quieres comprobar que pjp^{-1} es la matriz original, puedes escribir el comando siguiente :
`cas p j 'inv(p)' | \* | normal`
o `cas 'normal(p*j*inv(p))'`

3 Traducción a TEX

La traducción al LATEX de los comandos es guardada en el fichero `session.tex` en el directorio actual, salvo cuando se activan las variables de entorno `SHOW_TIME`. Requieren un preámbulo que puede ser copiado del archivo `doc/preamble.tex`.

Nótese que cada comando nuevo es añadido a `session.tex`, es una buena idea borrar este fichero de vez en cuando.

Puedes convertir fórmulas a LATEX usando el comando `cas2tex` y obtener directamente a fichero fuente compilable directamente en LATEX . Por ejemplo :
`cas2tex '[[1,2],[3,4]]' > essai.tex`
seguido de :
`latex essai.tex`
O en un sólo paso :
`cas2tex '[[1,2],[3,4]]' | latex --`
(produce el archivo `texput.dvi`)

4 Programando en C++

Primer ejemplo :

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

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

Escribe esto como `essai.cc` y compílalo :
`g++ -g essai.cc -lgiac -lgmp`
y ejecútalo :
`./a.out`

4.1 Organización del código fuente

Nótese que puedes usar `#include <giac/giac.h>` para incluir todas las cabeceras de la librería.

- `gen.cc/.h`: operaciones artiméticas en la clase base entera
- `identificateur.cc/.h`: nombre global
- `unary.cc/.h`: clase operadores unarios incluyendo operadores no unarios vistos como operación unaria sobre un vector de sus argumentos
- `symbolic.cc/.h`: clase objetos simbólicos
- `usual.cc/.h` Operaciones unarias habituales
- `vecteur.cc/.h`: algebra lineal
- `derive.cc intg.cc lin.cc series.cc subst.cc/.h`: Cálculo. Derivación está bien, así como integración de fraciones racionales, el resto tiene que ser implementado
- `moyal.cc/.h`: operadores pseudo-diferenciales
- `tex.cc/.h`: conversión a LaTeX
- `sym2poly.cc/.h`: conversión de polinómios a simbólicos
- `index.cc/index.h`: clase para indexación de tensores multivariables
- `poly.h, monomial.h`: clase plantilla de tensores multivariables
- `gausspol.cc/.h`: especialización de la plantilla de tensores para coeficientes enteros
- `input_parser.yy input_lexer.ll input_lexer.h`: intérprete
- `modpoly.cc/.h`: polinómios densos univariables de enteros y enteros modulares (Atención: funciones tipo gcd no funcionan si no es aritmética modular)
- `modfactor.cc/.h`: factorización de polinómios univariables densos sin cuadrados (Requiere NTL o PARI para lll y knapsack para ser rápido y funcional)
- `series.cc/.h`: desarrollo en series y límites usando el algorítmo mrv .

Lée `giac.texinfo` para obtener una corta descripción de las clases disponibles. Algunos ejemplos de programas están incluidos: `src/factor.cc`, `src/normalize.cc`, `src/cas.cc`, `src/partfrac.cc` and `src/integrate.cc`. Para compilar estos programas, puedes, o bien usar :
`g++ -g` *name.cc* `-lgmp -lgiac`
o lanzar `emacs` y ejecutar su comando de compilación (menú Herramientas).