Swiginac - Extending Python with Symbolic Mathematics

Ola Skavhaug[1,2] Ondrej Certic[3,4]

Simula Research Laboratory[1]

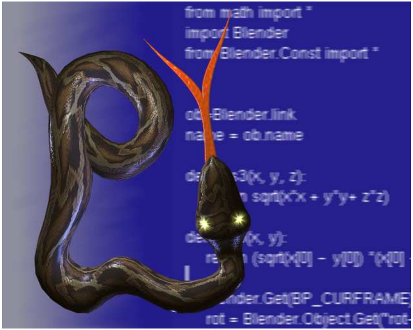
Dept. of Informatics, University of Oslo[2]

Faculty of Mathematics and Physics[3]

Charles University in Prague[4]

October 19–20 2005, TTI, Chicago

Outline



List of Topics

List of Topics

GiNaC, http://www.ginac.de

GiNaC is not a CAS

GiNaC is a C++ library for applications in need of symbolic manipulation. Python is such an application

Features

- Symbols and expressions with arithmetic operations
- Multivariate polynomials and rational functions
- Matrices and vectors
- Linear systems solver
- Tayler series expansions
- Differentiation and integration
- Output C, Python and LaTeX code
- ...

GiNaC can be used from Python

Existing Python bindings to GiNaC:

- PyGiNaC by Pearu Peterson (http://cens.ioc.ee/projects/pyginac/)
- PyGiNaC by by Jonathan Brandmeyer (http://pyginac.sourceforge.net/)

Both interfaces are generated with Boost.Python. This procedure requires quite a lot manual work, but produces efficient wrapper code. Pearu Peterson's project looks dead

Swiginac anno 2003

Why another Python interface to GiNaC?

In 2003, Pearu's PyGiNaC was the only alternative, and it failed the 5 min time limit for installation.

- We used SWIG; a simplified wrapper interface generator developed by David Beazly at Chicago
- We had successfully used SWIG to interface Diffpack
- For code verification using the method of manufactured solutions, only a limited interface to GiNaC was needed
- Strategy: Automatically generate the interface files by running the preprocessor on the GiNaC header files
- The resulting interface was rather crude, and a higher–level Python module, Symbolic, was implemented on top

Swiginac anno 2005

A new interface

Ondrej Certic at Charles University in Prague contacted me and wanted to improve the bindings

- New strategy: Manually convert the GiNaC header files to SWIG interface files, and implement a set of typemaps to make a higher–level interface
- A lot, but not all, of the GiNaC classes are now exposed to Python
- Certain GiNaC structures are converted to Python types in the interface and vice versa

SWIG typemapping example

We convert various types to GiNaC's proxy class *ex*

```
%typemap(in) ex & {
    $1 = type2ex($input);
    if (!$1) return NULL;
}
```

```
ex * type2ex(PyObject * input) {
  basic *btmp; GETDESC(basic);
  if (not((SWIG_ConvertPtr(input,(void **)&btmp,basicdescr,0))==-1))
    return new ex((*btmp));
  if (PyInt_Check(input))
    return new ex(numeric(PyInt_AsLong(input)));
  if (PyFloat_Check(input))
    return new ex(numeric(PyFloat_AsDouble(input)));
  if (PyList_Check(input)) {
    lst *l=list2lst(input);
    if (l==NULL) return NULL;
    return new ex(l->eval());
  }
  return NULL;
}
```

Symbols

Symbols are basic units in Swiginac

```
from swiginac import *
a = symbol('a', r'\alpha')
b = symbol('b', r'\beta')
print b
u = b + a
u.set_print_context('tex')
print u
```

Prints *b* and $\beta + \alpha$ (in LaTeX)

All expressions in GiNaC are built with symbols. The drawback of this approach is that the level of abstraction is limited

Functions

Lots of functions are available

```
u = sin(exp(b))
print u.printlatex()

v = tgamma(a+sqrt(b))
print v.printlatex()
```

Prints sin(exp(β)) and $\Gamma(\alpha + \sqrt{\beta})$ (in LaTeX)

All trigonometric and hyperbolic functions are implemented in GiNaC, most of them interfaced in swiginac

Symbolic differentiation

Objects have the method `diff` for differentiation:

```
x = symbol('x')
y = symbol('y')
P =  x**5 + x**2 + y
P.diff(x, 1) # 5*x**4+2*x
P.diff(x, 2) # 2+20*x**3

u = sin(exp(x))
u.diff(x,2) # -sin(exp(x))*exp(x)**2+exp(x)*cos(exp(x))
```

Matrices

```
mat1 = matrix(2,2) #Two by two matrix
mat1[0,0] = v
mat1[1,1] = u
print mat1.printlatex()
# Equivalent: mat1 = diag_matrix([u,v])
```

Output:

$$\begin{pmatrix} \Gamma(\alpha + \sqrt{\beta}) & 0 \\ 0 & \sin(\exp(\beta)) \end{pmatrix}$$

```
mat2 = matrix([[sqrt(a),0],[1.0, cosh(b)]])
print mat2.printc()
```

Output:

[[pow(a,(1.0/2.0)),0.0],[1.000000000000000e+00,cosh(b)]]

Matrices

```
mat1 = matrix(2,2) #Two by two matrix
mat1[0,0] = v
mat1[1,1] = u
print mat1.printlatex()
# Equivalent: mat1 = diag_matrix([u,v])
```

Output:

$$\begin{pmatrix} \Gamma(\alpha+\sqrt{\beta}) & 0 \\ 0 & \sin(\exp(\beta)) \end{pmatrix}$$

```
mat2 = matrix([[sqrt(a),0],[1.0, cosh(b)]])
print mat2.printc()
```

Output:

```
[[pow(a,(1.0/2.0)),0.0],[1.0000000000000000e+00,cosh(b)]]
```

Simple integral support

We can construct integral objects and integrate either symbolically or numerically:

```
x = symbol('x')
integ = integral(x, 0, 1, x*x)
print integ.printlatex()
print integ.eval_integ()
print integ.evalf()
```

Output:

$\int_0^1 dx\, x^2$

$1/3$

0.33333333333333333332

Substitution

Algebraic objects in expressions can be substituted

```
u = sin(exp(b))
v = u.subs(exp(b)==sqrt(a)) # v = sin(a**(1/2))
w = v.subs(a==2).evalf()    # Convert sin(2**(1/2)) to numeric
float(w)                    # Convert to Python double
```

Sub–expressions do not match:

```
x = symbol('x'); y = symbol('y'); z = symbol('z')
u = sin(x+y+z)
v = u.subs(x+y==4) # v = sin(x+y+z)
w = u.subs([x==1, y==2, z==3]) # Same as u.subs(x+y+z==6)
```

Substitution

Algebraic objects in expressions can be substituted

```
u = sin(exp(b))
v = u.subs(exp(b)==sqrt(a)) # v = sin(a**(1/2))
w = v.subs(a==2).evalf()    # Convert sin(2**(1/2)) to numeric
float(w)                    # Convert to Python double
```

Sub–expressions do not match:

```
x = symbol('x'); y = symbol('y'); z = symbol('z')
u = sin(x+y+z)
v = u.subs(x+y==4) # v = sin(x+y+z)
w = u.subs([x==1, y==2, z==3]) # Same as u.subs(x+y+z==6)
```

Solving linear systems

`lsolve` solves linear systems:

```
>>> x = symbol('x')
>>> y = symbol('y')
>>> lsolve([3*x + 5*y == 2, 5*x+y == -3], [x,y])
[x==-17/22, y==19/22]
```

And finally, we have Taylor series expansion

Expressions can expand themselves as a Taylor series:

```
x =symbol("x")
>>> sin(x).series(x==0, 8)
1*x+(-1/6)*x**3+1/120*x**5+(-1/5040)*x**7+Order(x**8)
```

Summary

- Swiginac is a free CAS for Python
- It is GPL, since GiNaC is
- It can do basic symbolic manipution
- Useful for MMS, generating talks, and probably a whole lot more
- Much remaining work and many unresolved questions

Try it out:

Visit `http://swiginac.berlios.de`