Large Scale Symbolic Programming with GiNaC

Alexander Frink, Christian Bauer, Richard Kreckel*

Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

Abstract. GiNaC is a free framework that embeds symbolic manipulation consistently into the C++ programming language. It deliberately neglects the split-up into a low level language and a high level language, traditional in the design of computer algebra systems. The user usually interacts with GiNaC directly in C++. GiNaC was designed to provide efficient handling of multivariate polynomials, algebras and some special functions that are needed for loop calculations in HEP. But it also bears some potential to become a more general purpose symbolic system.

INTRODUCTION

When we start a software project that relies to some extent on manipulating symbolic expressions (as opposed to quickly checking some result with our favoured Computer Algebra System (CAS)), we are usually faced with a multi-lingual situation. We start by implementing some formulae using a symbolic package and the language it provides. Then we want to get numerical results out of it which is usually done by the CAS' code-generator which produces C or FORTRAN code which we compile and let run. Sometimes, we also wish to share our work and provide an intuitive user interface for our program so others may trigger combined symbolical/numerical/graphical computations with a few keystrokes.

Even if we have mastered all those individual steps, it is not uncommon to see how the interaction of the different software packages we have been using so far makes the whole endeavor fail. It may be that our CAS' language is too restriced to formulate larger programs. It may also be that our programs or the scripts we wrote to glue everything together break at each software upgrade. It may even turn out that we cannot convince our colleagues to help us with coding in three different languages on one single project. In any case, large scale projects tend to become unmaintainable. This is a situation not uncommon in physics projects. GiNaC[1] was designed to overcome such problems by providing fundamental symbolic facilities in the C++ programming language.

* Authors' email addresses: Alexander.Frink@Uni-Mainz.DE, Christian.Bauer@Uni-Mainz.DE, Richard.Kreckel@Uni-Mainz.DE. This work was supported by 'Graduiertenkolleg Eichtheorien – Experimentelle Tests und theoretische Grundlagen' at University of Mainz.

[1] GiNaC stands for 'GiNaC is Not a CAS'.

HOW PROGRAMS ARE WRITTEN DOWN WITH GINAC

GiNaC deliberately denies the need for a distinction of implementation language at different steps of a project. It is entirely written in C++ and adheres to the international ISO standard (1). The user can interact with it directly in that language, freely build upon it and extend it. (Compare this with closed systems where expanding the kernel is impossible.) Here is a complete program that uses a Rodrigues repesentation $H_n(x) == (-1)^n e^{x^2} (d/dx)^n e^{-x^2}$ to compute Hermite polynomials:

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

ex HermitePoly(const symbol & x, int n)
{
    const ex HGen = exp(-pow(x,2));
    return normal(pow(-1,n)*HGen.diff(x, n)/HGen);
}

int main(int argc, char **argv)
{
    int degree = atoi(argv[1]);
    numeric value = numeric(argv[2]);
    symbol z("z");
    ex H = HermitePoly(z,degree);
    cout << "H_" << degree << "(z) == "
         << H << endl;
    cout << "H_" << degree << "(" << value << ") == "
         << H.subs(z==value) << endl;
    return 0;
}
```

When this program is compiled and called with `6` and `0.8-0.5*I` as command line arguments it will readily print out the sixth Hermite polynomial together with that polynomial evaluated numerically at $z = 0.8 - 0.5i$:

```
$ c++ hermite.cc -o hermite -lginac
$ ./hermite 6 0.8-0.5*I
H_6(z) == -120-480*z^4+720*z^2+64*z^6
H_6(0.8-0.5*I) == 350.865216-267.07455999999999996*I
```

Alternatively, arbitrary length exact rational arguments are also honored, thus avoiding rounding errors:

```
$ ./hermite 6 4/5-1/2*I
```

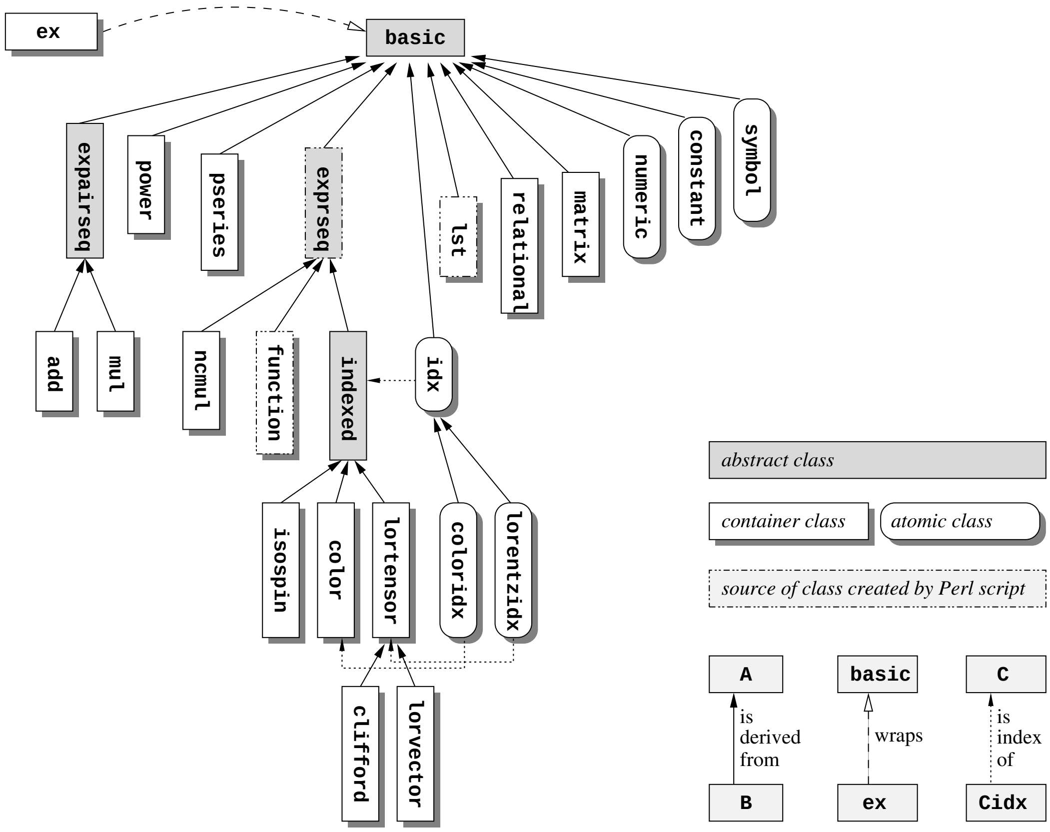

FIGURE 1. GiNaC's class hierarchy and some of the relations between the classes.

```
H_6(z) == -120-480*z^4+720*z^2+64*z^6
H_6(4/5-1/2*I) == 5482269/15625-834608/3125*I
```

Syntactically, the program shows how symbolic expressions are written down in GiNaC pretty much like common numeric terms thanks to operator overloading between some of the classes in Fig. 1.

GiNaC can be configured to work with Masaharu Goto's Cint C/C++ interpreter[2]. It is then possible to work with GiNaC interactively, defining symbolic expressions, control structures like loops and conditionals and even functions. A little example shows how one may symbolically compute the non-relativistic approximation of $\gamma = (1-(v/c)^2)^{-1/2}$ to ten orders in v and then inverse it and get the original result back up to an $O(v^{10})$ term:

```
$ ginaccint
Welcome to GiNaC-cint (GiNaC V0.7.0, Cint V5014062)
  __,  _______  GiNaC: (C) 1999-2000 Johannes Gutenber
 (__) *       | Germany.  Cint C/C++ interpreter: (C)
  ._) i N a C | Goto and Agilent Technologies, Japan.
<-------------' with ABSOLUTELY NO WARRANTY.  For deta
Type `.help' for help.

GiNaC> symbol v("v"), c("c");}
```

[2] This amazing tool is also the basis of the well-known ROOT object-oriented data analysis framework (3)

```
GiNaC> ex gamma = 1/sqrt(1 - pow(v/c,2));
GiNaC> ex gamma_nr = gamma.series(v==0,10);
GiNaC> cout <<pow(gamma_nr,-2) <<endl;
1+(1/2*c^(-2))*v^2+(3/8*c^(-4))*v^4+(5/16*c^(-6))*v^6+
(35/128*c^(-8))*v^8+Order(v^10)^(-2)
GiNaC> cout <<pow(gamma_nr,-2).series(v==0,10) <<endl;
1+(-c^(-2))*v^2+Order(v^10)
```

Loops are of course written down in `ginaccint` just as they would be written for a compiled program:

```
GiNaC> for (int i=0; i<10000; i+=2) {
     >     cout << bernoulli(numeric(i)) << ", ";
     > }
1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -361
7/510, 43867/798, -174611/330, 854513/138, -236364091/
2730, 8553103/6, -23749461029/870, 8615841276005/14322
, -7709321041217/510, 2577687858367/6, -26315271553053
477373/1919190, 2929993913841559/6, -26108271849644912
2051/13530, 1520097643918070802691/1806, -278332695793
01024235023/690, 596451111593912163277961/282, -560940
3368997817686249127547/46410, 495057205241079648212477
```

... and so on. Due to a limitation of Cint, however, function definitions need some special help:

```
GiNaC> //GiNaC-cint.function
next expression can be a function definition
GiNaC> const ex EulerNumber(const unsigned n)
     > {
     >     const symbol xi;
     >     const ex generator = pow(cosh(xi),-1);
     >     return generator.diff(xi,n).subs(xi==0);
     > }
creating file /tmp/ginac26197caa
```

```
GiNaC> cout << EulerNumber(42) << endl;
Out3 = -10364622733519612119397957304745185976310201
GiNaC> quit;
```

It is thus possible to write large scripts and later compile them and link them in to the user's application.

HOW GINAC WORKS

GiNaC implements a number of symbolic classes as shown in Fig. 1. All classes are referenced by the class of all expressions `ex`. There is reference counting at work here, providing GiNaC with a transparent non-interruptive memory management. It is implemented in the interplay between class `ex` and the abstract base class `basic`, so the user who wishes to extend the system does not have to worry about memory management. For all kinds of numbers (integer, rational, float, complex) GiNaC uses Bruno Haible's super-efficient C++-library CLN (2) as a foundation class. Our class `numeric` is basically a wrapper class around CLN's class `cl_N`.

COMPARISON AND BENCHMARKS

Is GiNaC competitive? Certainly it does not feature such a load of features as some big commercial systems. But even with the tasks it can do, is it efficient? We try to answer this question with some tests. The first two try to measure the scaling behaviour when problems become very big.

The first one is a rather common three step substitute-expand consistency benchmark:

- let e be the expanded sum of n symbols squared: $e \leftarrow (\sum_{i=0}^{n-1} a_i)^2$,
- in e substitute $a_0 \leftarrow -\sum_{i=2}^{n-1} a_i$,
- expand e again, it collapses to ${a_1}^2$.

Using `ginaccint` this test may be formulated interactively in an elegant way as follows:

```
$ ginaccint
Welcome to GiNaC-cint (GiNaC V0.7.0, Cint V5014062)
  __,  _______  GiNaC: (C) 1999-2000 Johannes Gutenber
 (__) *       | Germany.  Cint C/C++ interpreter: (C)
  ._) i N a C | Goto and Agilent Technologies, Japan.
<-------------' with ABSOLUTELY NO WARRANTY.  For deta
Type `.help' for help.

GiNaC> #include <sstream>
GiNaC> vector<symbol> a;
GiNaC> ex bigsum = 0;
GiNaC> for (int i=0; i<50; ++i) {
     >         ostringstream buf;
     >         buf << "a" << i << ends;
     >         a.push_back(symbol(buf.str()));
     >         bigsum += a[i];
     > }
GiNaC> ex sbtrct = -bigsum + a[0] + a[1];
GiNaC> cout << pow(bigsum,2).expand()
     >                         .subs(a[0]==sbtrct)
     >                         .expand() << endl;
a1^2
GiNaC> quit;
```

The results are shown in Fig. 2. The system used was an Alphaserver 8400 running at 300MHz with roughly 1GB of main memory. We have chosen this system in order to give Maple a chance since MapleV has a built in limitation to $2^{16}-1$ terms in the representation of sums on all 32bit platforms which cause this test to break down quite early at $n = 181$. The timings show the expected n^2 scaling behavior for the GiNaC, Mathematica and MuPAD with GiNaC being the fastest of the three while the curves for Reduce and Maple are odd, since they take off very fast for small problems and become very slow at the upper end. The fundamental difference here lies in the memory management: GiNaC, Mathematica and MuPAD use reference counts, while Maple and Reduce (being Lisp-based) use a garbage collector.

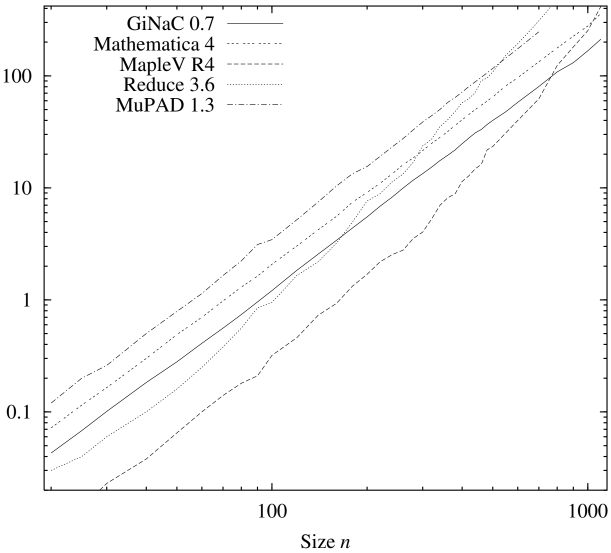

FIGURE 2. Absolute runtimes for the substitute-expand test in seconds.

The second test was done on an Intel P-III with 384MB RAM running at 450MHz. The systems were asked to expand the Gamma function around its first pole at $x = 0$ to high orders symbolically. The series expansion is: $\Gamma(x) = \frac{1}{x} - \gamma + \left(\frac{\pi^2}{12} + \frac{\gamma^2}{2}\right)x - \left(\frac{\pi^2\gamma}{12} + \frac{\gamma^3}{6} + \frac{\zeta(3)}{3}\right)x^2 + \ldots$ It can then easily be checked for consistency numerically: $\Gamma(x) = \frac{1}{x} - 0.5772157 + 0.9890560x - 0.9074791x^2 + \ldots$ Fig. 3 shows the expected breakdown of MapleV at order $n = 35$ when intermediate results exceed $2^{16}-1$ terms. There are two curves shown for Mathematica here. This is because by default Mathematica expands the result only partially to a form comparable with the other systems. Only after applying `FunctionExpand` it gets the same result, but extremely slowly so. But since it turns

Table 1. Runtimes in seconds for the tests proposed by R. Lewis and M. Wester (only as far as applicable to GiNaC) on an Intel P-III 450MHz, 384MB RAM running under Linux. Abbreviations used: GU (gave up, like Maple's error `object too large`), CR (crashed, out of memory), NA (not available), UN (unable, a prerequisite test failed).

	Benchmark	**GiNaC 0.7**	**MapleV R4**	**MuPAD 1.4.1**	**Pari-GP 2.0.19β**	**Singular 1-3-7**
A:	divide factorials $\left.\frac{(1000+i)!}{(900+i)!}\right\vert_{i=1}^{100}$	0.20	4.11	1.13	0.37	19.0
B:	$\sum_{i=1}^{1000} 1/i$	0.019	0.08	0.10	0.041	0.54
C:	gcd(big integers)	0.25	9.92	3.01	1.65	0.11
D:	$\sum_{i=1}^{10} iyt^i/(y+it)^i$	0.78	0.13	1.21	0.20	NA
E:	$\sum_{i=1}^{10} iyt^i/(y+\vert 5-i\vert t)^i$	0.63	0.05	2.33	0.11	NA
F:	gcd(2-var polys)	0.080	0.10	0.21	0.057	0.13
G:	gcd(3-var polys)	2.50	1.85	3.31	99.5	0.38
H:	det(rank 80 Hilbert)	10.0	42.9	42.5	3.97	CR
I:	invert rank 40 Hilbert	3.38	7.48	12.0	0.62	CR
J:	check rank 40 Hilbert	1.61	2.61	2.95	0.22	UN
K:	invert rank 70 Hilbert	22.1	113.8	74.0	5.90	CR
L:	check rank 70 Hilbert	9.19	24.1	14.2	1.57	UN
M_1:	rank 26 symbolic sparse, det	0.36	0.40	0.75	0.016	0.003
M_2:	rank 101 symbolic sparse, det	1903.3	GU	CR	CR	251.2
N:	eval poly at rational functions	CR	GU	CR	CR	NA
O_1:	three rank 15 dets (average)	43.2	GU	CR	CR	CR
O_2:	two GCDs	CR	UN	UN	UN	UN
P:	det(rank 101 numeric)	1.10	12.6	44.3	0.09	0.85
P':	det(less sparse rank 101)	6.07	13.6	46.2	0.38	1.25
Q:	charpoly(P)	103.9	1453.2	741.7	0.15	4.4
Q':	charpoly(P')	212.8	1435.6	243.1	CR	5.0

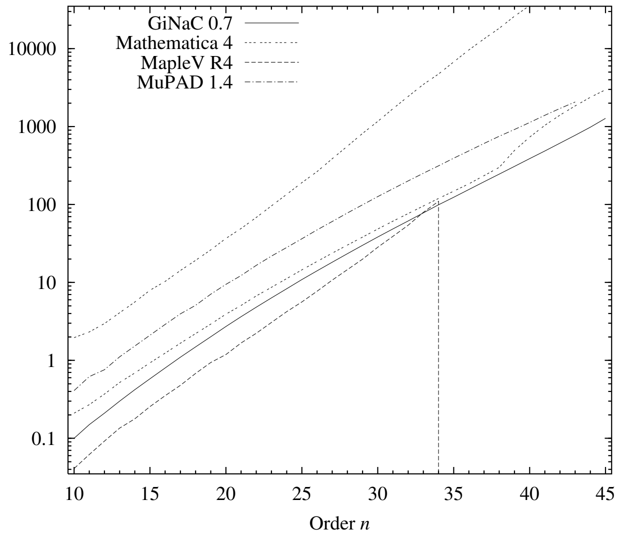

FIGURE 3. Absolute runtimes to expand the Γ-function around $x = 0$ up to order n in seconds.

out that the intermediate result is already useful to some extent (as can be seen by evaluating it numerically) we also give it credit by showing the timings for computing the series without applying `FunctionExpand`.

As yet another test, we apply GiNaC to a number of tests invented by Robert Lewis and Michael Wester (described in (4)) as far as they are applicable to GiNaC. The results are shown in Table 1.

STATUS AND AVAILABILITY

Being a special-purpose system, GiNaC aims at being a fast and reliable foundation for combined symbolical/numerical/graphical projects in C++. It is currently used as a symbolic engine for loop calculations in the HEP project (5). It may be downloaded and distributed under the terms of the GNU general public license from `http://www.ginac.de/`. The supporting CLN library can also be obtained from there and is also available packaged for some Linux distributions. A tutorial introduction and complete cross references of the source code can also be found there.

REFERENCES

1. American National Standards Institute, *ISO/IEC 14882-1998(E) Programming languages — C++* (1998)
2. Bruno Haible, Richard Kreckel, *CLN Class Library for Numbers*, `ftp://ftp.ilog.fr/pub/Users/haible/gnu/cln-1.1.tar.gz`, (2000)
3. Rene Brun, Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework Proceedings of AIHENP 97*, Laussanne, (1996)
4. Robert H. Lewis, Michael Wester, *Comparison of Polynomial-Oriented Computer Algebra Systems*. (Presented at the 1999 ISSAC Conference, Vancouver), available from `http://www.fordham.edu/lewis/cacomp.html`, (1999)
5. Lars Brücher, Johannes Franzkowski, Dirk Kreimer, *Comput. Phys. Commun.* **115**, (1998) 140–160.