Options for Commutative Algebra in SAGE

Martin Albrecht (malb@informatik.uni-bremen.de)

February 12, 2007

Outline

V(I) Datastructures

Consider for example the ring $\mathbb{Q}[x, y, z]$ and $f = 5 * x^2y^3 + z^4 - 2$. This boils down to

$\{ \{ 0:2, 1:3 \}:5, \{ 2:4 \}:1, \{ \}:-2 \}$

in the current implementation. This data structure is called a **PolyDict** in SAGE. Every **MPolynomial** has such a thing (it isn't one). The exponent dictionaries are called **ETuple**. PolyDict and ETuple are implemented in *not-optimized* Pyrex/SageX using Python dictionaries. MPolynomial is implemented in Python.

Operations

Simple operations (Addition, Multiplication, etc.) are implemented **natively** (and naively btw.) while **more complicated operations** (factorization, gcd, division) are performed **using Singular**. This adds additional overhead as data has to be passed back and forth between Singular and SAGE.

Btw.: There is also a tiny wrapper around **libCF** in SAGE which I use sometimes for polynomial evaluation and such. It is faster than the native SAGE implementation but slower than Singular as it wraps a library meant for factorization.

Outline

V(I) Improving the Current Implementation

- need to do it anyway: no library allows polynomial rings over Python objects
  - need to have fallback implementation
  - might be okay if it is not optimized well
  - might even stay in Python for a while
- There is lots of room for improvements
  - don't use Python dictionaries for the ETuples
  - improve overall implementation, use Pyrex tricks, use better algorithms
  - push multivariate polynomials down to SageX
  - ... but is it worth it?
- Feature-wise: Many monomial orderings are not implemented, e.g. block orderings, which are important for crypto.

Outline

What is Singular I

- computer algebra system focused on commutative algebra
- developed since 1980s in Kaiserslautern (Greuel) and Berlin (Pfister).
- current version is 3-0-2.
- written in C-ish C++. (good, since C is better understood by me and Pyrex than C++)

V(I) What is Singular II

- claims to have the fastest multivariate polynomial arithmetic overall. Claim backed by William's and my experience. Actually,
  - polynomial arithmetic faster than MAGMA
  - coefficient arithmetic supposed to be slower than MAGMA

```
#MAGMA 2.13-5 (32-bit, not optimized for my machine)
> e := Random(1000^400,1000^410)/Random(1000^400,1000^410);
> t:= Cputime();
> for i in [1..10^5] do; f := e*e; end for;
> Cputime(t);
3.070

#SAGE (64-bit, local build)
sage: e = ZZ.random_element(1000^400,1000^410)/\
          ZZ.random_element(1000^400,1000^410)
sage: time for i in range(10^5): f = e*e
CPU times: user 1.16 s, sys: 0.00 s, total: 1.16 s

#Singular's RR in SAGE (64-bit, local build)
sage: P.<x,y,z> = MPolynomialRing_si(QQ,3)
sage: ep = P(e)
sage: time for i in range(100000): f = ep*ep
CPU times: user 1.11 s, sys: 0.00 s, total: 1.71 s
```

What is Singular III

- Also, very rich set of features: set related ideal operations, radicals, closures
- term orderings: Matrix ordering, block orderings
- higher level algorithms: solving, Gröbner basis algorithms, Gröbner walks
- We use a lot of its functionality via pexpect already

Problem: Singular is a stand-alone application

- cannot link against Singular (symbols not exported, main)
- not designed to play nice with other components (e.g. memory management)
- no bird's eye view API documentation
- pexpect too slow for low level arithmetic

... fixed.

Wrapping Singular

- wrote **libsingular.so.3-0-2** prototype + some SAGE bindings
- changes to Singular library are minimal so far (just don't create main())
- Singular team is supportive for my effort, i.e. changes might hit upstream
- API is surprisingly easy to understand for internal code, but some quirks necessary (global variables)
- aim to support polynomials over $\mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{F}_p \mathbb{F}_{p^n}$ + quotient rings over them.
- considering to link Singular against Givaro for faster arithmetic over $\mathbb{F}_{p^n}$.
- need to sort out memory management issues at some point
- will take a lot of time till a production ready version is released (time limit: end of summer)

Preliminary Timing I

object creation isn't killing us for small examples.

```
#SAGE using Singular (64-bit, custom build)
sage: time for i in range(1000000): f = (x*y + z)^5
CPU times: user 3.33 s, sys: 0.01 s, total: 3.34 s

#SAGE using Singular in SageX loop (64-bit, custom build)
sage: time sometest(P,1000000)
CPU times: user 1.40 s, sys: 0.00 s, total: 1.40 s

#MAGMA 2.11-2 (32-bit, not optimized for my machine)
> t:= Cputime();
> for i in [1..1000000] do; f := (x*y + z)^5; end for;
> Cputime(t);
3.709
```

bigger examples look even better

```
#SAGE Singular
sage: time for i in range(1000000): f = (x*y^3 + z^2)^20
CPU times: user 8.21 s, sys: 0.02 s, total: 8.23 s

#MAGMA 2.11-2 (32-bit)
> t:= Cputime();
> for i in [1..1000000] do; f := (x*y^3 + z^2)^20; end for;
> Cputime(t);
25.709
```

Preliminary Timing II

Incidently:

```
#Singular via SAGE/Python
sage: time for i in range(1000000): f = (x*y + z)
CPU times: user 0.71 s, sys: 0.01 s, total: 0.72 s
Wall time: 0.72

#Singular's own interpreter, what's going wrong?
sage: time singular.eval("poly f; for(int i=0;i<1000000;i++) { f = x*y + z; }")
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 12.85
```

Outline

What is CoCoALib

"CoCoALib is now GPL'd! However we still ask you not to disclose this address to others, but to invite them to contact us!"

. . . a secret GPL'd C++ library for multivariate polynomial arithmetic. Rumored to be released to the general public end of February. It is a complete rewrite of the CoCoA computer algebra system.

Sorry

I haven't really looked into CoCoALib yet. However, this is how it looks like:

```
// Q
ring Q = NewFractionField(Z);

//-------------------------------------------------------------------
// Q[y]

// Indet name is y[0]
PolyRing P = NewPolyRing(Q, 1, symbol("y"));

// set the C++ variable y to the value of the indeterminate y in ring P
RingElem y = indet(P, 0);
RingElem f = 15 * y + power(y,3);
RingElem g = 4  * y - 3 * power(y,7);
GlobalOutput() << "--------------------------------------------" << endl;
GlobalOutput() << "   ring is Q[y[0]]" << endl;
TestRing(P, f, g);
```

Outline

Quotient Ring over GF(2) I

We can represent monomials as bitstrings in $\mathbb{F}_2[x_1, \dots, x_n]/ < x_1^2 - x_1, x_n^2 - x_n >$. Examples:

```
# Multiply
 x *  y = xy ||  x *  x =  x
-------------------------------
10 | 01 = 11 || 10 | 10 = 10
# Division (if divisible)
xy /  y =  x || xy /  x =  y
-------------------------------
11 ^ 01 = 10 || 11 ^ 10 = 01

# Divisibility testing
x divides xy       = True
-------------------------------
(10 ^ 11) & (~10) = 1
      01  &   01  = 0

xy divides x       = False
-------------------------------
(11 ^ 10) & (~11) = 0
      01  &   00  = 0
```

Quotient Ring over GF(2) II

- using SSE2 (x86) or AltiVec (PPC) instruction set: monomial multiply of up to 128 variables in one instruction.
- have (okay) implementation in my thesis.
- still many stupid things in there: use Python dictionary in multiplication to ensure uniqueness of terms in resulting polynomial.
- For very large rings with very sparse polynomials: slow.

Questions?

Thank You!