Development and structure of the PARI library

Karim Belabas

http://pari.math.u-bordeaux.fr/

Warning

This talk focuses on the current development version of the PARI library (2-2-8, to be released), freely available from anonymous CVS (Concurrent Version System), see

`http://pari.math.u-bordeaux.fr/CVS.html`

Development goals

- Uniformize, clean up, improve with respect to speed, memory use, and reliability.
- User-driven and application-driven development. Priority : clearing significant bottlenecks for applications to algebraic number theory (factoring polynomials over finite fields, linear algebra in large dimensions, subresultants and gcds).
- Devise algorithms and experiment with them, usually after quick prototyping in GP.
- Document what is there, including algorithms used and references.
- Having fun !

An example : `factorint`

- ECM (1.x.x, Bernardi)
- MPQS (2.0.9, Papanikolaou, Roblot, based on LiDIA code)
- Pollard-Brent ρ, general factorization driver (2.0.10, Niklasch)
- ECM-rewrite (2.0.12) (Niklasch)
- SQUFOF (2.0.21, Niklasch)
- countless MPQS/ECM/ρ-tunings (Niklasch)

To be done : double large prime variation in MPQS, Montgomery arithmetic in ECM, ρ, primality tests...

Components

Apart from high-level modules (number fields, elliptic curves, primality and factorization) and specialized ones (gp-specific, graphisms), the PARI library has five basic components

- Input/Output and memory management.
- Kernel : the 4 basic operations `+`, `-`, `*`, `/` (the last one having many variants)
- Polynomial arithmetic
- Linear algebra
- Transcendental functions

Memory management (1/2)

Results are allocated sequentially on a huge (e.g. 10MB) connected chunk of preallocated memory, the PARI `stack`. It is the user's responsibility to

- ensure allocated stack is large enough (otherwise `The PARI stack overflows !`).
- collect garbage, once in a while or systematically.

Memory management (2/2)

- save/restore :

```
pari_sp ltop = avma ; /* save stack pointer */
...
avma = ltop ; /* restore it. Erase accumulated data */
```

- save/copy/restore/copy :

```
pari_sp ltop = avma ; /* save stack pointer */
GEN a, b, c, d ; /* will hold objects to be preserved */
...
gerepileall(ltop, 4, &a,&b,&c,&d) ; /* clean up */
```

- save/copy/restore : when data and garbage both connected. Saves one copy but not always applicable. And more prone to user error.

Kernel

Implements mostly the 4 operations : `+`, `-`, `*`, `/`

- Level 0 : operations on longs, e.g. `addll` (add two unsigned longs and possibly set a carry bit). Mostly assembly and inlined routines.
- Level 1 : operations on `t_INT`s, e.g. `addii`, `t_REAL`s, e.g. `addrr`, and combinations of these, e.g. `mpadd`. Currently two versions : native and GMP (`mpn` level).
- Level 2 : operations on polynomials and vector/matrices with coefficients in a specified ring, e.g.`Flx_add`, `FpX_add`, `ZX_add`, `FpM_mul`, `FpM_FpV_mul`, etc.
- Level 3 : generic operations, e.g. `gadd`, `gmul`.

New Level 2 Kernel

Function names are built by concatenating the “types” of the arguments, then an operation. A “type” name is a base ring followed by a letter indicating the structure : e.g. `X` for univariate polynomials, `V` for vectors, `M` for matrices, `Q` for classes in a polynomial quotient ring. Base rings are

`Fl` : $\mathbb{Z}/l\mathbb{Z}$ where l is a small integer, not necessarily prime. Implemented using ulongs

`Fp` : $\mathbb{Z}/p\mathbb{Z}$ where p is a `t_INT`, not necessarily prime. Implemented as `t_INT`s z, $0 \leqslant z < p$.

`Fq` : $\mathbb{Z}[X]/(p, T(X))$, p a `t_INT`, T a `t_POL` with `Fp` coefficients or NULL. Implemented as `t_POL`s z with `Fp` coefficients, $\deg(z) < \deg$ `T`.

`Z` : the integers $\mathbb{Z}$, implemented as `t_INT`s.

`z` : the integers $\mathbb{Z}$, implemented using longs

`Q` : the rational numbers $\mathbb{Q}$, implemented as `t_INT`s and `t_FRAC`s.

`Rg` : a commutative ring, whose elements can be `gadd`-ed, `gmul`-ed, etc.

Applications

Many specialized routines are built on top of these basic ones, e.g. `FpM_ker`, `FpY_FpXY_resultant`. And then of course, all high-level routines eventually call such functions.

While programming with the PARI library, everything may be emulated by generic routines and higher-level types such as `t_POLMOD`s, `t_INTMOD`s. At a significant cost, in time and space.

To be done : implement asymptotically fast(er) algorithms, where that would make a difference to intended applications. Tighter interfaces with GMP.

Plans

- Release a new stable version (2.1 was released in 2000)
- Replace the GP parser by GP2C (Problem : cannot maintain backward compatibility)
- Tighter integration with GMP (real arithmetic), fix transcendental functions (too slow).
- Screen crucial individual routines and algorithms to detect problems and inefficiencies.
- Document all PARI routines, add examples to all GP functions, write specialized test suites.
- Add selected useful algorithms. Either as new C code modules, or as GP scripts.