Interaction between Libpari and GP

A. Page

IMB
CNRS/Université de Bordeaux

25/06/2025

Plan

1. Finding C functions
2. Writing documentation
3. C functions and GP
4. Writing tests

Finding C functions

Finding the C function you need

Here are some methods to find the C function you are looking for:

- guessing the name (cf. the API tutorial);
- from the libpari documentation (libpari.dvi);
- from the GP documentation (users.dvi);
- reading the code of functions that you guess should use it.

Navigating the C code

Using tags you can:

- jump to the definition of a function by typing its name (`vi -t <name>` or `:ta <name>` with vi);
- idem from its appearance in the code (`Ctrl+]` to jump, `Ctrl+T` to go back with vi).

This allows you to efficiently navigate the code without knowing which function is defined in which file.

Writing documentation

GP functions documentation

GP functions should be documented in its installation file `src/functions/<section>/<function>`. The fields are:

- `Function:` name of the GP function.
- `Section:` section of the documentation.
- `C-Name:` name of the C function.
- `Prototype:` cf. next slide.
- `Help:` short help, in plain text.
- `Doc:` long help, in tex.
- (optional) `Variant:` related C functions.
- (optional) `Description/Wrapper:` GP2C related.

Lines not starting by one of these keywords should start with a space.

GP functions documentation

Each section of the documentation of GP functions (Chapter 3) starts by the paragraph `src/functions/<section>/HEADER` (written in tex).

The file `doc/usersch3.tex` is generated from the content of `src/functions/` and from `doc/usersFUNCS.tex`, and should never be modified directly.

Public C functions (i.e. declared in `paridecl.h`) should be documented in `usersch*.tex`. If you want you function to be private, declare it in `paripriv.h` (to be used in other by PARI C files) or make it `static`.

Basic prototypes

Input types:

- `G`: GEN
- `&`: GEN*
- `L`: long
- `n`: variable (becomes a long in C)
- `D?`: optional `?`$\in$ {`G`,`&`}, NULL if absent

Return type:

- nothing: GEN
- `l`: long
- `i`: int
- `v`: void
- `m`: unsafe GEN

(more in Section 5.8.3 of the libpari doc)

Examples

```
long issquareall(GEN x, GEN *pt) -> lGD&

GEN minpoly(GEN x, long v) -> GDn
```

Tex macros

In the tex parts of the documentations, the following macros are available:

- `\Z, \Q, \R, \C`
- `\kbd{}, \tet{}`
  for texttt, but tet creates an entry in the index.
- `\typ{}`
  for GEN types.
- `\bprog ... @eprog`
  for GP code examples (include examples in your doc!).
- `\fl`
  for flags.
- `\fun{return type}{name}{arguments}`
  to describe C function prototypes.

Refcards

Do not forget to add your function to `doc/refcard-*.tex`!

- Not necessarily with all the optional arguments.
- Same function can appear twice for argument variants.

Testing documentation examples

GP has a special mode to test documentation examples, the `doctest` default:

```
\z
```

This makes GP delete the initial `?` and ignore the `%n =` lines.

C functions and GP

Naming

Try to use the same name for the GP function and the corresponding C function.

Typical counterexample: we want to add an optional argument to an existing function `fun`. This is backwards compatible in GP, but not in C. Usually, we preserve the C function `fun` and create a new C function `fun0` that corresponds to the GP function `fun`.

Output sanity

Make sure that the output of your function is suitable for `gc_upto`. This is easy to ensure by a call to `gc_GEN`. Beware of unclean constructors such as `mkvec` and friends!

Input checking

Add simple and cheap argument checks to your functions:

- type (`typ`) and length (`lg`), usually not recursively (maybe one level).
- validity of the input, if cheap compared to the cost of the function.

You may document the behaviour as undefined when preconditions are not met.

Precision

Function returning real or complex numbers have a precision argument. This argument is not provided by the user, but the default `realbitprecision` is used.

- In C, this is `long prec`, in bits.
- Prototype: `p` (rounded up to a multiple of `BITS_IN_LONG`) or `b` (bitprecision).

The semantic is:

- if the input is exact, return the output at the given precision;
- if the input is inexact, return the output at the highest possible precision given the input.

Variables

Usually, when you need variables in C code, the variable is provided by the user. If you need to create a temporary variable, use `fetch_var_higher` to create a variable with higher priority that all existing ones, and do not forget to `delete_var` so as not to leak variables!

Install

While developping, it is often useful to install C functions to test them under GP.

- Basic use: `install(name,prototype);`
- Remove the `static` keyword and recompile gp if necessary.
- The `m` prototype is useful for unclean functions.
- Functions that can return NULL can be handled with Bill's isNULL trick:

  `isNULL(z=NULL)=z;`

  Then `isNULL(fun(x))` will return `NULL` (a GP variable) if `fun(x)=NULL` (the C NULL pointer) and `fun(x)` otherwise.

Writing tests

Test suite

Running a test with `make test-foo` consists in feeding gp with the content of `src/test/in/foo` and computing a diff with the expected output `src/test/32/foo` (excluding the timing from the diff), resulting in a diff file `Oxxx/foo-sta.dif` or `Oxxx/foo-dyn.dif`. The test is considered passed if the diff is empty.

Running tests faster

To save time when running tests, it is convenient to only run the `sta` suite (statically linked, faster than the dynamically linked one), and only a subset of the tests with:

```
make TESTS="foo1 foo2 ..." statest-all
```

or

```
make dotestSUF=sta test-foo1 test-foo2 ...
```

Patching

Procedure for adding tests:

1. Add a test to the test file `foo` (or create it).
2. If you created the file, do `./Configure -l`.
3. Run `make test-foo`.
4. Check whether `Oxxx/foo-sta.dif` is what you expected.
5. Update the output file with `patch -p1 Oxxx/foo-sta.dif` (only if the output is correct!).
6. Don't forget to add `src/test/32/foo` to your commit!

Tests guidelines

- No `install()` in tests (not portable enough).
- Also test bad inputs and inputs that trigger an error.
- Try to write stable tests, i.e. tests that do not depend on the least significant bits of approximate computations, on a choice of basis, on 32 vs 64 bits architecture.
- Use `setrand` before tests that use a probabilistic algorithm (especially if the output is not unique), so that adding tests to the file does not break your test.
- Try to write test that do not use a lot of stack. `parisizemax` is disallowed in tests; you may change `parisize` but keep it reasonable.
- Write tests that take only a few seconds (in 64 bits), at most one minute. Split your test file if necessary.

Testing in 32 bits

To run your test in 32 bits (much slower), you need the following packages:

```
  sudo apt install gcc-multilib lib32readline-dev
```

Then compile and run with:

```
CFLAGS=-m32 linux32 ./Configure
linux32 make gp
linux32 make test-foo
```

Your diff files will be in something like `Olinux-i686/`.

Thank you!

Have fun with Pari!