A brutal introduction to libpari programming

B. Allombert

IMB
CNRS/Université de Bordeaux

23/06/2025

libpari C headers

PARI code can be compiled in three ways:

1. as a standalone program
2. as a loadable module
3. directly inside libpari

In the first two cases the headers are included as follow

```
#include <pari/pari.h>
```

in the third case

```
#include "pari.h"
```

after all extra system headers have been included.
In the first case, PARI needs to be initialized with `pari_init` before being used.

libpari C types

The PARI library API mostly relies on three C types: `long`, `ulong` (short for `unsigned long`) and `GEN`.
PARI denotes the number of bits in a `ulong` by `BITS_IN_LONG`.
A `GEN x` is a pointer to a data structure representing a PARI object.
$x[0]$ contains the type and the length of the object, which are accessed using `typ` and `lg`. The other components can be either codeword or pointers to other `GEN` (which can contains pointers to other `GEN` etc.) `GEN` can have several components that point to the same sub-`GEN`, but cycles are not allowed.

The GEN types

`typ` returns one of the following `enum` values.

Leaf types (all components are codeword)

`t_INT`	arbitrary precision integers
`t_REAL`	arbitrary precision real numbers
`t_VECSMALL`	vectors of long
`t_STR`	character string
`t_INFINITY`	$\pm\infty$

Recursive types (some components are pointers to other `GEN`s)

`t_INTMOD`	$\mathbb{Z}/n\mathbb{Z}$
`t_FRAC`	rational numbers
`t_FFELT`	finite field elt.
`t_COMPLEX`	complex numbers
`t_PADIC`	p-adic numbers
`t_QUAD`	quadratic numbers (deprecated)
`t_POLMOD`	$K[X]/T$

The GEN types

`t_POL`	polynomials
`t_SER`	power series
`t_RFRAC`	rational function
`t_QFB`	binary quadratic form
`t_VEC`	row vector
`t_COL`	column vector
`t_MAT`	matrix
`t_LIST`	list
`t_CLOSURE`	GP functions
`t_ERROR`	error context

It is customary to call a `GEN` of type `t_INT` a `t_INT`, etc.

Warning about use of `long` and `ulong`

- According to the C standard, `ulong` are wrapping, that is all operations are done modulo $2^{\texttt{BITS\_IN\_LONG}}$, but this is not the case for `long`, where overflows are undefined.
- % and / in C follow FORTRAN semantic and not PARI semantic when the operands are negative: $-1\%3 = -1$. PARI provides `smodss` and `umodsu` to avoid such problem.
- Immediate constants sometime need to be suffixed with L or UL to avoid confusion with `int` (especially in variadic functions like `mkvecsmalln`).
- C `int` must generally be avoided. In PARI they normally only takes the values 0, 1 and -1.

GEN

- typ(x): return the type of x.
- lg(x): return the length of x.
- settyp(x,t): set the type of x to t.
- setlg(x,l): set the length of x to l.
- cgetg(l,t); allocate a `GEN` of length l and type t on the PARI stack.

t_INT object

t_INT are arbitrary precision relative integers.

- signe(x) : sign of x, 0 is $x == 0$
- lgefint(x) : actual size in words (can be smaller than lg(x)).
- expi(x) : exponent (i.e. logint(x,2)).

Access to the mantissa words of a t_INT is done using the macro int_W, see the documentation. The sign can be changed with setsigne.

Small integers are available as universal objects.

-2	gen_m2
-1	gen_m1
0	gen_0
1	gen_1
2	gen_2

t_INT object

In the API, the operand types are encoded by the letter

- s : long (for "small integer")
- u : ulong
- i : t_INT

For example, for conversion:

- stoi: convert a long to a t_INT
- utoi: convert a ulong to a t_INT
- itos: convert a t_INT to a long
- itou: convert a t_INT to a ulong

Comparing:

- equality: equalii, equaliu, equalis
- equality to 1 or -1: equali1, equalim1
- comparison: cmpii, cmpis, cmpiu cmpsi, cmpui, cmpss, cmpuu : return the sign of $x - y$ as a int.

Operations on `t_INT`

- `addii`, `addis`, `addiu`, `addss`, `adduu`: return the sum (return a `t_INT`).
- idem with `add` replaced by `sub`, `mul`, `mod`.
- `negi(x)` returns $-x$, `absi(x)` return $|x|$.
- `sqri`, `sqrs`, `sqru` return the square.
- `shifti(x,n)` shift x of n bits (n can be positive or negative).
- `truedvmdii`, `truedivii`, `modii` euclidean division.
- `smodis`, `smodss`: return the remainder as a `long`.
- `umodiu`, `umodsu`: return the remainder as a `ulong`.
- `gc_INT` faster version of `gc_GEN` for `t_INT`.
- `gc_stoi` faster version of `gc_GEN(av,stoi(...))`
- `gc_utoi` faster version of `gc_GEN(av,utoi(...))`

In-place operations

To operate on `t_INT` in place:

- `affii(x,y)` set the value of y to the value of x, assuming $\texttt{lg}(y) \geq \texttt{lgefint}(x)$.
- `affsi(x,y)`, `affui(x,y)` set the value of y to the value of x, assuming $\texttt{lg}(y) \geq 3$
- `z=cgeti(l)` allocates a `t_INT` with $\texttt{lg}(z) = \texttt{l}$.
- `nbits2lg(n)` returns the length needed for a `t_INT` of n bit.
- `bit_accuracy(x)` return the number of bits of the `t_INT` x.

t_REAL

t_REAL are arbitrary precision floating points real numbers

- signe(x) : sign of x, 0 is $x == 0$
- realprec(x) : precision in bits, always a multiple of BITS_IN_LONG.
- expo(x) : exponent of x
- mantissa_real(x,&e) return the mantissa as a t_INT.

The sign can be changed with setsigne, the exponent with setexpo.

- GEN real_1(long prec): return 1. to precision prec.
- GEN real_0(long prec): return 0. to precision prec.
- GEN real_m1(long prec): return -1. to precision prec.

The code letter for `t_REAL` is `r`. Functions that need to convert integers to `t_REAL`s need an extra argument called `long prec` which is the precision (in bit) wanted.

- `GEN stor(long x, long prec)`: convert a `long` to a `t_REAL`.
- `GEN utor(ulong x, long prec)`: convert a `ulong` to a `t_REAL`.
- `GEN itor(GEN x, prec)`: convert a `t_INT` to a `t_REAL`.
- `GEN dbltor(double x)`: convert a C double to a `t_REAL`.
- `GEN rtor(GEN x, prec)`: convert a `t_REAL` to a `t_REAL` with a different precision.
- `double rtodbl(GEN x)`: convert a `t_REAL` to a C double.

Operations on `t_REAL`

- equality: `equalrr`, `equalri`, `equalrs`
- comparison: `cmprr`, `cmpri`, `cmprs`, `cmpir`, `cmpsr`.
- `addrr`, `addri`, `addrs`, `addir`, `addsr`: return the sum (return a `t_REAL`).
- idem with add replaced by `sub`, `mul`, `div`
- `negr(x)` returns $-x$, `absr(x)` return $|x|$, `sqrr(x)` returns x^2. `shiftr(x,n)` multiply x by 2^n (n can be positive or negative).
- `divrr`, `divri`.
- `truncr`, `floorr`, `ceilr` `roundr`.

In-place operations

To operate on `t_REAL` in place:

- `affrr(x,y)` set the value of y to the value of x converted to the precision of y.
- `affsr(x,y)`, `affur(x,y)` set the value of y to the value of x converted to a `t_REAL` with the same precision as y.
- `z=cgetr(l)` allocates a `t_REAL` with $\mathtt{lg}(z) = \mathtt{l}$.
- `prec2lg(n)` returns the length needed for a `t_REAL` of precision n.

Vectors

Vectors are available in two variants `t_VEC` and `t_COL`. Since PARI uses French linear algebra convention, `t_COL` is often more natural. To test if a type `t` is either `t_VEC` or `t_COL`, use `is_vec_t(t)`. if v is a vector, and `l=lg(v)`, then v has `l` -1 components, `gel(v,1),...,gel(v,l-1)`.
To allocate a vector with n undefined components , do `v = cgetg(n+1, t_VEC);` or `v = cgetg(n+1, t_COL);`.
Note than this is not a valid object until all components have been set (by using `gel(v,i) = ...`).

Vector example

```
GEN fun(long n)
{
  long i;
  GEN v = cgetg(n+1, t_COL);
  for (i = 1; i <= n; i++)
    gel(v,i) = sqru(i);
  return v;
}
```

Vectors

`zerovec(n)` and `zerocol(n)` create a vector of `gen_0` that can be filled later. `const_vec(n,x)` and `const_col(n,x)` create vectors of x.
Fixed-length short vectors can be created with `mkvec(x1)`, `mkvec2(x1,x2)`, `mkvec3(x1,x2,x3)`, `mkvec4(x1,x2,x3,x4)`, `mkvec5(x1,x2,x3,x4,x5)`, `mkvecn(n,x1,...,xn)`, `mkcol(x1)`, `mkcol2(x1,x2)`, `mkcol3(x1,x2,x3)`, `mkcol4(x1,x2,x3,x4)`, `mkcol5(x1,x2,x3,x4,x5)`. `mkcoln(n,x1,...,xn)`.
For example `[0,1,2]` can be created with `mkvec3(gen_0,gen_1,gen_2)`.

t_MAT

t_MAT are represented as vector of t_COL of identical length. if m is a t_MAT, and l=lg(m), then m has $\mathtt{l}-1$ columns, gel(m,1),...,gel(m,l-1), which have all the same length. Thus the number of row of a matrix with zero columns is not defined. The coefficients of m can be accessed with gcoeff(m,i,j) which is a short-hand for gel(gel(m,j),i).
To allocate a t_MAT with n undefined columns, do m = cgetg(n+1, t_MAT) then set the columns with gel(v,i) = ...
zeromatcopy(n,m) creates a matrix of gen_0 that can be filled later.

Matrix example

```
GEN fun(long n, long m)
{
  long i, j;
  GEN v = cgetg(m+1, t_MAT);
  for (i = 1; i <= m; i++)
  {
    GEN c = cgetg(n+1, t_COL);
    for (j = 1; j <= n; j++)
      gel(c,j) = mulss(i,j);
    gel(v, i) = c;
  }
  return m;
}
```

t_VECSMALL

t_VECSMALL is a low-level type used for vector of long or ulong depending on the context. If v is a t_VECSMALL and l=lg(v), the components are v[1],...,v[l-1] in the long case and uel(v,1),...,uel(v,l-1).

To allocate a t_VECSMALL with n undefined components , do

```
v = cgetg(n+1, t_VECSMALL);
```

and then set v[1],...,v[n] or uel(v,1),...,uel(v,n).

t_VECSMALL example

```
GEN fun(long n)
{
  long i;
  GEN v = cgetg(n+1, t_VECSMALL);
  for (i = 1; i <= n; i++)
    uel(v,i) = i;
  return v;
}
```

t_VECSMALL

`zero_zv(n)` creates a vector of 0 that can be filled later.
`const_vecsmall(n,x)` create vectors of x.
Fixed-length short vectors can be created with `mkvecsmall(x1)`, `mkvecsmall2(x1,x2)`, `mkvecsmall3(x1,x2,x3)`, `mkvecsmall4(x1,x2,x3,x4)`, `mkvecsmall5(x1,x2,x3,x4,x5)`, `mkvecsmalln(n,x1,...,xn)`.

t_POL

t_POL are polynomials.

- signe(x): 0 if $x = 0$, 1 otherwise.
- varn(x): variable number of x.
- degpol(x): degree of x (-1 if $x = 0$), degpol(x)=lg(x)-3.
- lgpol(x): 1+degpol(x), lg(x)-2.
- leading_coeff(x): leading coefficient.
- constant_coeff(x): constant coefficient.
- pol_0(v), pol_1(v), pol_x(v): polynomials 0, 1, x in variable v.

The leading coefficient of a non-constant polynomial cannot be an exact zero. However a polynomial can have signe 0 even if its degree is not -1, if all its coefficients are inexact zero. A constant polynomial can be 0 if its leading coefficient is 0 but not `gen_0`.
If P is a `t_POL` of degree d, the coefficients of degree $0 \leq i \leq d$ can be accessed with `gel(P,i+2)`.
The variable number can be set with `setvarn(x,v)`. All variables that appears in components of polynomial must have strictly lower priorities than `varn(x)`
Priority are compared using `varncmp(v,w)`.

t_POL

Creating a t_POL of degree d and variable number v requires four steps:

allocation `P = cgetg(d+3, t_POL);`

setting the variable `P[1] = evalvarn(v);`

filling the coefficients set `gel(P,i+2)` for all `i`.

renormalize `P = RgX_renormalize_lg(P, d+3);`

The last step will take care of setting the sign correctly.

t_POL example

```
GEN fun(long d, long v)
{
  long i;
  GEN P = cgetg(d+3, t_POL);
  P[1] = evalvarn(v);
  for (i = 0; i <= n; i++)
    gel(P, 2+i) = sqrs(i);
  return RgX_renormalize_lg(P, d+3);
}
```

t_STR

A t_STR is a (NUL-terminated) character string.

- `GSTR(x)`: return the string pointer.
- `long nchar2nlong(long n)`: number of long to allocate for n characters.
- `GEN strtoGENstr(const char *s)`: convert a C string to a t_STR.

The PARI stack

Since `GEN` can be quite complex, PARI uses a dedicated memory management system: the PARI stack. The PARI stack is a contiguous chunk of memory used as a scratchpad for computation. It is made of two consecutive chunks (allocated with `mmap`). The first chunk is of length `parisize` starts from `top` down to `bot` and is allocated as real memory. The second chunk starts from `bot` down to `vbot` and is allocated as virtual memory. The total length from `top` to `vbot` is `parisizemax`.
The stack pointer is called `avma`.

$$\left.\begin{array}{l}\left.\begin{array}{c}\mathit{top}\\ |\\ \mathit{avma}\\ |\\ \mathit{bot}\end{array}\right\}\mathit{parisize}\text{ real memory}\\ \left.\begin{array}{c}|\\ \mathit{vbot}\end{array}\right\}\text{virtual memory}\end{array}\right\}\mathit{parisizemax}$$

When `avma` reaches `bot`, the `bot` is lowered (and a `Warning: increasing stack size` occurs), When `bot` reaches `vbot`, a `PARI stack overflow` error occurs. The virtual memory between the old and new `bot` is then converted to real memory.

The low-level function for allocating memory is very simple:

```
INLINE GEN
new_chunk(size_t x) /* x is a number of longs */
{
  GEN z;
  if (x > (avma-bot) / sizeof(long))
    new_chunk_resize(x);
  z = ((GEN) avma) - x;
  avma = (pari_sp)z;
  return z;
}
```

The PARI stack has several advantage.

- memory allocation are very fast.
- it is fully reentrant.
- it prevents memory leak.
- it is always obvious who owns a particular address.
- it allows object to be serialized.

In principle, `GEN` can exist anywhere in memory, however all libpari functions that return new `GEN`s allocate them on the PARI stack.

A function should normally start by recording the stack pointer avma of type pari_sp and restore the stack at the end. For that purpose, gc_GEN, gc_long, gc_ulong are available.

```
<TYPE> fun(...)
{
  pari_sp av = avma;
  <TYPE> z;
  ...
  z = ...;
  return gc_<TYPE>(av, z);
}
```

where <TYPE> can be any of long, ulong, GEN. If the GEN is known to be a leaf type, gc_leaf should be used. For void function, use set_avma(av).

gc_GEN and gc_upto

`gc_GEN(av, z)` works by copying recursively the `GEN z` outsize the stack, resetting `avma` to `av` and copying back `z` at `avma`. The cost only depends on the size of `z`

`gc_upto(av, z)` is a faster version that just moves `z` to `avma`, shifting the pointers as needed. However it has two requirements.

1. the pointer `z` must be created before its components.
2. The part of the stack used by `z` and its components need to be connected.

`GEN` produced by `gc_GEN` always have this property.
If furthermore, there were no temporaries created, `return z` is sufficient.

Examples

```
pari_sp av = avma;
GEN a = utoi(3), b = utoi(4);
GEN V = cgetg(3,t_VEC);
gel(V,1) = a;
gel(V,2) = b;
return gc_GEN(av, V);
```

In this example, the first condition is not respected, `gel(V,1)` and `gel(V,2)` are created before V.

```
GEN V = cgetg(3,t_VEC);
gel(V,1) = utoi(3);
gel(V,2) = utoi(4);
return V;
```

In this example, there is no temporaries created, no need for gc.

```
pari_sp av = avma;
GEN V = cgetg(3,t_VEC);
gel(V,1) = addiu(shifti(gen_1,128),1);
gel(V,2) = utoi(4);
return gc_GEN(av, V);
```

In this example, the second condition is not respected, the object `shifti(gen_1,128)` is a temporary is the middle of V.

```
pari_sp av = avma;
GEN z = shifti(gen_1,128);
GEN V = cgetg(3,t_VEC);
gel(V,1) = addiu(z,1);
gel(V,2) = utoi(4);
return gc_upto(av, V);
```

In this example, the temporary is created before V, so now both condition hold.

```
pari_sp av = avma;
GEN a = addiu(shifti(gen_1,128), 1);
GEN V = cgetg(3,t_VEC);
gel(V,1) = a;
gel(V,2) = utoi(4);
return gc_GEN(av, V);
```

In this example, `gel(V,1)` is created before `V`.

mkvec2 and retmkvec2

```
{
  pari_sp av = avma;
  V = mkvec2(utoi(3), utoi(4));
  return gc_GEN(av, V);
}
```

In this example, `utoi(3)` and `utoi(4)` are created before V.

```
{ retmkvec2(utoi(3), utoi(4)); }
```

Here, `retmkvec2` is a macro that ensures that `cgetg(3,t_VEC)` is called before `utoi(3)` and `utoi(4)` are evaluated.

```
#define retmkvec2(x,y)\
  do { GEN _v = cgetg(3, t_VEC);\
       gel(_v,1) = (x);\
       gel(_v,2) = (y); return _v; } while(0)
```

Functions returning several `GEN`

If a function returns several `GEN` (using pointers), one can use `gc_all(av, n, &x_1,...,&x_n)` to restore the stack while preserving `x_1,...,x_n`. `gc_all` returns `x_1` so that

```
return gc_all(av, n, &x_1,...,&x_n)
```

is valid.

Example

```
GEN extgcd(GEN A, GEN B, GEN *U, GEN *V)
{
  pari_sp av = avma;
  GEN ux = gen_1, vx = gen_0, a = A, b = B;
  while (!gequal0(b))
  {
    GEN r, q = dvmdii(a, b, &r), v = vx;
    vx = subii(ux, mulii(q, vx));
    ux = v; a = b; b = r;
  }
  *U = ux;
  *V = diviiexact( subii(a, mulii(A,ux)), B );
  return gc_all(av, 3, &a, U, V);
}
```

Cleaning up the stack inside loops

In this example we clean up the temporaries so `v` stays connected.

```
GEN fun(long n)
{
  long i;
  GEN v = cgetg(n+1, t_COL);
  for (i = 1; i <= n; i++)
  {
    pari_sp av2 = avma;
    GEN w = sqri(addiu(sqru(i),1));
    gel(v,i) = gc_upto(av2, w);
  }
  return v;
}
```

Cleaning up the stack inside loops

In this example we clean up the temporaries at each turn of the loop. We need to make sure to list all variables that need to be preserved.

```
GEN fibo(long n) {
  pari_sp av = avma;
  GEN a = gen_0, b = gen_1;
  long i;
  for (i = 1; i < n; i++)
  {
    GEN c = b;
    b = addii(a,b); a = c;
    gc_all(av, 2, &a, &b);
  }
  return gc_INT(av, b);
}
```

Cleaning up the stack inside loops

To avoid cleaning the stack too often, the macro gc_needed(,1) is used to detect when the stack is half full.

```
GEN fibo(long n) {
  pari_sp av = avma;
  GEN a = gen_0, b = gen_1;
  long i;
  for (i = 1; i <= n; i++)
  {
    GEN c = b;
    b = addii(a,b); a = c;
    if (gc_needed(av, 1))
      gc_all(av, 2, &a, &b);
  }
  return gc_INT(av, b);
}
```

Cleaning the stack inside loops

When cleaning the stack inside loop, one should add a warning:

```
for (i = 1; i <= n; i++)
{
  GEN c = b;
  b = addii(a,b); a = c;
  if (gc_needed(av, 1))
  {
    if (DEBUGMEM > 1)
      pari_warn(warnmem,"fibo, step %ld", i);
    gc_all(av, 2, &a, &b);
  }
}
```

Using affii

```
GEN fibo(long n) {
  long l = nbits2lg(n);
  GEN b = cgeti(l);
  pari_sp av = avma;
  GEN a = cgeti(l);
  pari_sp av2 = avma;
  for (i = 1; i <= n; i++)
  {
    GEN c = addii(a,b);
    affii(b,a);
    affii(c,b);
    set_avma(av2);
  }
  set_avma(av); return b;
}
```

Clones

Sometime it is inconvenient to keep some `GEN` in the PARI stack.

- `gclone`: return a copy of a `GEN` outside the PARI stack. This copy must be freed at some point using `gunclone`.
- `gunclone`: free a clone
- `gcopy`: copy a `GEN` to the PARI stack, in a way suitable for `gc_upto`.
- `icopy`: as `gcopy` for `t_INT`.

GP variables and history entries (%) are clones.

Example: gc_GEN

A slower version of gc_GEN

```
GEN my_gc_GEN(pari_sp av, GEN s)
{
  GEN c = gclone(s);
  set_avma(av);
  s = gcopy(c);
  gunclone(c);
  return s;
}
```

Example: gc_GEN

A slower version of gc_GEN

```
GEN my_gc_GEN(pari_sp av, GEN s)
{
  return gc_upto(av, gcopy(s));
}
```

However this version has the major drawback of risking a stack overflow if s is large.