Pari-GP

Presentation
Mohamadou Sall

Ecole Mathématique Africaine, Franceville 2018

5 avril 2018

Summary

Presentation of Pari/GP

PARI/GP is a specialized computer algebra system

- primarily aimed at number theorists, but
- has been put to good use in many other different fields,

The main advantages of the system are

1. its speed,
2. its extensive algebraic number theory module

PARI/GP is Free Software, covered by the GNU General Public License

PARI is used in three different ways :

1. As a library **libpari** : **PARI** the C library
2. As a sophisticated programmable calculator, named **gp**, whose language is **GP**
3. The **compiler gp2c** translates GP code to C

Installation

Windows users

Download

- Stable 32-bit version : Pari32-2-9-4.exe
- Stable 64-bit version : Pari64-2-9-4.exe

from

`https://pari.math.u-bordeaux.fr/download.html`

Run the setup

Debian/Ubuntu users

1. Build pari from source :

   sudo apt-get build-dep pari

   sudo apt-get install pari-gp

2. Download pari-2.9.4.tar.gz from

   `https://pari.math.u-bordeaux.fr/download.html`

Debian/Ubuntu users

1. unpack the download file

   tar -xvf nameofyourfile.tar.gz

2. Type

   ./Configure

   in the toplevel directory.

3. To compile the gp binary and build the documentation type

   sudo make all

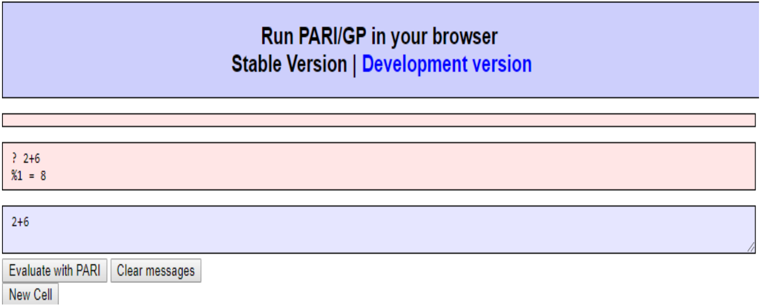
4. When everything looks fine, type (with superuser privileges)

   sudo make install

Browser version

Use the following address

`https://pari.math.u-bordeaux.fr/gp.html`



Getting Started

1. For window's users : run the executable file
2. For UNIX versions : open a terminal and type *gp*
3. Connect to `https://pari.math.u-bordeaux.fr/gp.html`

Basic Operations

To **enable logging** of your input for future reference :

```
> \l .../mylogfile.log
```

Standard arithmetic operations

- Addition/Subtraction ; Multiplication/Division : $+, -, *, /$

  ```
  > 2018+1; \\ basic operation, no printout

  > 2018*1
  %4 = 2018
  ```

- Euclidean division ; Euclidean remainder ; Exponentiation $\backslash$; % ; $\wedge$

  ```
  > 20\3
  %8 = 6
  > 20%3
  %9 = 2
  > 13^7
  %11 = 62748517
  ```

Basic Operations

- Logical operators giving as results 1 **(true)** or 0 **(false)**.

```
> 2==2
%12 = 1
> 2!=2
%13 = 0
```

- To make a comment use $/ * \cdots * /$ or $\backslash\backslash$

```
> /*
comment> ceci est un commentaire sous GP
comment> */
```

Some implemented functions

- Extended Euclidean Algorithm

```
> gcdext(4, 2018)
%75 = [-504, 1, 2]
```

- Factorization

```
> factor(1000000000000000000000)
%76 =
[2 21]

[5 21]
```

- Primality test

```
> isprime(1009)
%77 = 1
```

Some implemented functions

- Generate a **random prime number**

```
> randomprime(2^100)
%78 = 470942683887785808879503146627
> isprime(randomprime(2^100))
%79 = 1
```

Exercise :

1. Generate a prime number between 2 and $N = 2^{100}$
   - Multiply the two generated numbers
   - Try to factor the result
2. Do the same thing with $N = 2^{500}$

To get **help**

- use the user's guide, or the refcard or
- in the interpreter type *?NameOfFunction*, eg *?gcdext*

Programming in GP

The GP language is **not typed**. It has about 23 types of which

- *t_FFELT* : Finite field element including
    - the field characteristic p,
    - an irreducible polynomial $T \in \mathbf{F}_p[X]$ and
    - the element expressed as a polynomial in (the class of) X
- *t_PADIC* : $p-$adic numbers with three components :
    - the prime p,
    - the "modulus" p^k, and
    - an approximation to the $p - adic$ number.
- *t_POL*, *t_POLMOD* : Polynomials and polynomials modulo $P(X)$

Control Statements

A number of control statements are available in GP

- **for(X = a, b, seq)** : evaluates *seq*, where the formal variable X goes from a to b.

  ```
  > for(i=1, 10, i=i++; print(i))
  2
  4
  6
  8
  10
  ```

  specific loop

  ```
  > forprime(i=4, 10, print(i))
  5
  7
  ```

Control Statements

- **if(a, {seq1 } ; {seq2})** : Evaluates the expression sequence **seq1** if a is non-zero, otherwise the expression **seq2**.
```
> if(2==2, print(OK), print(KO))
OK
if(!2==2, print(OK), print(KO))
KO
```
- **until(a, seq)** : evaluates **seq** until a is not equal to 0.
```
> a=2; until(a==5, print(continuer); a==a++)
continuer
continuer
continuer
> a
%9 = 5
```
- **while(a, seq)** : while a is non-zero, evaluates **seq**.

Rewrite the above command using while

Exercise Rewrite the above command using while

Using Scripts

Create a file *prog.gp* with the following content

```
somme(a,b) = a+b;
```

and save. In GP type

```
> \r .../prog.gp
```

Try the function

```
> somme(a, b)
```

```
> somme(4, 6)
%7 = 10
```

Structure of a function in GP

```
FunctionName(Parameters)=
{
    Instructions
}
```

- Put {, } on the line after the sign =
- End the function with ;
- Declare local variables with *my()*
- The function returns the last computed result

Indent the code for legibility

Example of a Function

Add in the file *prog.gp*

```
fibo(n)=
{
my(u0=0,u1=1);
for(i=2,n,
[u0,u1]=[u1,u0+u1]);
u1;
}
```

To view *user-defined functions* type

```
> ?0
somme
```

Exercise Compute the fibonacci number of index 100 using ???

1. the created function *fibo*
2. the corresponding Pari function

Elliptic Curve Discrete Logarithm Problem

To **specify** an elliptic curve (**for cryptography**) one specifies

1. a base field : a prime number p and a dimension n
2. an equation over the finite field
3. a base point P : the subgroup generated by P

Theorem (Existence and uniqueness of finite fields)

For every prime p and every integer $n > 0$ there exist a finite field with p^n elements, that is isomorphic to $\mathbf{F}_{p^n}$.

There are two types of finite fields.

1. Prime finite fields, $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$ where p is a prime integer.
2. Finite fields $\mathbf{F}_q$ where $q = p^r$, is such that $r > 1$ and p a prime integer.

Arithmetic Properties : Let x and y in $\mathbf{F}_{q^n}$, then

1. $x^{q^n-1} = 1, \; x \neq 0$
2. $x^{q^n} = x$

Prime finite fields

1. To create a random prime number :
   - *randomprime({N})* : returns a strong pseudo prime in [2, N-1].
   - ```
     > p=randomprime(2^100)
     %1 = 792438309994299602682608069491
     ```
2. To create an element of the finite field $\mathbf{F}_p$
   - ```
     > a=Mod(2,p)
     ```
   - ```
     > a^(p-1) \\ powering : Test if a is in the created field
     %2 = Mod(1, 792438309994299602682608069491)
     > lift(a^(p-1))
     %3 = 1
     ```
     ```

General finite fields

To build an irreducible polynomial of degree n of $\mathbf{F}_p$ use **ffinit(p,n)**.

```
> P=ffinit(13, 2)
%21 = Mod(1, 13)*x^2 + Mod(1, 13)*x + Mod(12, 13)
> polisirreducible(P)
%22 = 1
```

To build an element of $\mathbf{F}_{p^n}$ use **ffgen**

```
> a=ffgen(P, 'a);
> a^(13^2-1) /* Test if the created element is in the field */
%25 = 1
```

Operations over finite field

- random element and order of an element

```
> fforder(a) \\ order of an element
%29 = 28
> random(a) \\ random element of F_p^n
%31 = 7*a + 2
```

- To get a **generator** of $\mathbf{F}_{p^n}^*$:

```
> b = ffprimroot(a)
%32 = 7*a + 8
> fforder(b)
%33 = 168
```

Discrete logarithm of a finite field element

Definition

Let $\mathbf{F}_q$ be the finite field of order q and g the generator of $\mathbf{F}_q^*$. The **discrete logarithm** $\log_g(h)$ of an element $h = g^n$ is equal to n.

Discrete logarithm of a finite field element

Use the function **fflog**

- Very easy case

```
> fflog(a, b)
%50 = 6
> b^6==a \\ Test for the DL result
%51 = 1
```

- another case

```
> F = ffinit(2, 127);
> a = ffgen(F, 'a);
> b = ffprimroot(a);
> # \\ activates the timer
   timer = 1 (on)
> fflog(a, b)
time = 4,734 ms.
%70 = 162485313910162364684918816260658444080
```

Discrete logarithm of a finite field element :

- Beyond the default stack of Pari

```
> F=ffinit(5, 127);
> at = ffgen(F, 'at);
> fforder(at)
%46 = 29387358770557187699218413430556141945466638919302188
> bt = ffprimroot(at)
> fflog(at, bt)
  ***   at top-level: fflog(at,bt)
  ***                 ^-------------
  *** fflog: the PARI stack overflows !
  current stack size: 8000000 (7.629 Mbytes)
  [hint] set 'parisizemax' to a non-zero value in your GPRC
```

- It is possible to modify the default parisize

```
> default(parisize, "32M")
  ***   Warning: new stack size = 32000000 (30.518 Mbytes).
```

Elliptic curves over finite fields

Initialization of a base field

```
> P=ffinit(13,2);
> a=ffgen(P,'a);
```

From a **short Weierstrass model**

$$y^2 = x^3 + a_4 x + a_6$$

```
> Es=ellinit([a^4,a^6],a)
%6 = [0, 0, 0, 10*a + 2, 5*a + 5, 0, 7*a + 4, 7*a + 7, 8*a, a +
```

From a **long Weierstrass model**

$$y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$$

```
> E=ellinit([a,a^2,a^3,a^4,a^6],a);
```

Basic Functions

- **Coefficients** of E

```
> [a1, a2, a3, a4, a6]=E[1..5]
%11 = [a, 12*a + 1, 2*a + 12, 10*a + 2, 5*a + 5]
```

- $j-$**invariant**

```
> E.j
%12 = 11
```

- **Discriminant**

```
> E.disc
%13 = 4*a + 8
```

Basic Functions

The discriminant

```
Disc := -a1^6*a6 + a1^5*a3*a4 - a1^4*a2*a3^2 - 12*a1^4*a2*a6 +
8*a1^3*a2*a3*a4 + a1^3*a3^3 + 36*a1^3*a3*a6 - 8*a1^2*a2^2*a3^2
- 48*a1^2*a2^2*a6 + 8*a1^2*a2*a4^2 - 30*a1^2*a3^2*a4 +
72*a1^2*a4*a6 + 16*a1*a2^2*a3*a4 + 36*a1*a2*a3^3 +
144*a1*a2*a3*a6 - 96*a1*a3*a4^2 - 16*a2^3*a3^2 - 64*a2^3*a6 +
16*a2^2*a4^2 + 72*a2*a3^2*a4 + 288*a2*a4*a6 - 27*a3^4
- 216*a3^2*a6 - 64*a4^3 - 432*a6^2;
```

of $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ is a non zero element

Minimal generating set

- Random points on the curve

```
> P = random(E)
%15 = [8*a + 4, a + 6]
> Q=random(E)
%16 = [8*a + 11, 2*a + 11]
```

- Check that a point is on the curve

```
> ellisoncurve(E, P)
%17 = 1
```

Group Operations

- Addition

```
> elladd(E, P, Q)
%18 = [3*a + 12, 12*a]
```

- Scalar multiplication

```
> ellmul(E, P, 100)
%19 = [6*a, 5*a + 11]
```

- Subtraction

```
> ellsub(E, P, P)
%20 = [0]
```

Discrete Logarithm

Let E be an elliptic curve over a finite field

Definition

The discrete logarithm on E is the problem, given a point $Q \in E$, find an integer $n \in \mathbf{Z}$ such that $Q = nP$ if such an integer n exists.

Discrete Logarithm

- **Cardinality** of E and **order** of a point on E

```
> ellcard(E)
%24 = 192
> oP = ellorder(E, P)
%25 = 48
```

- Discrete Logarithm

```
nP = ellmul(E, P, random(oP))
%47 = [9*a + 4, 9*a + 2]
> n = elllog(E, nP, P)
%49 = 40
> ##
  ***   last result computed in 0 ms.
> nP==ellmul(E, P, n)\\Test if n is the DL of nP in base P
%50 = 1
```

Very easy to find n.

Precomputations of Elliptic Curve Cryptography

Exercise :

Suppose that we are given

$$(190; 271) \equiv k(1; 237) \pmod{1009}$$

with

$$E : y^2 = x^3 + 71x + 602 \pmod{1009}$$

then it is easy to find

$$k = ???$$

Implement it in GP

1. Print the number of point of E : use *ellcard*
2. Print the order of $[1, 237]$: use *ellorder*
3. Find the value of k and print it : use *elllog*

For help type *?NameOfFunction*, eg *?ellcard*

Some certicom ECDLP challenge problems

Curves	Bits	Operations	Prizes (US dollars)	Status
ECCp-97	97	$3.0 \cdot 10^{14}$	\$5,000	1998
ECCp-109	109	$2.1 \cdot 10^{16}$	\$10,000	2002
ECCp-131	131	3.5×10^{19}	\$20,000	?
ECCp-163	163	2.4×10^{24}	\$30,000	?
ECCp-191	191	4.9×10^{28}	\$40,000	?
ECCp-239	239	8.2×10^{35}	\$50,000	?
ECCp-359	359	9.6×10^{53}	\$100,000	?

ECCp-97, the prime number p (with $\mathbf{F}_p$ the base field) has 97 bits. Use the the function **binary(x)** to get the number of bits of x.

Some issues that must be addressed

- Choose a suitable finite field $\mathbf{F}_q$
  - exercise level with bits less than 109,
  - rather easy level with bits in $109 - 131$, and
  - very hard level with bits in 163-359.

- Choose a suitable elliptic curve $E/\mathbf{F}_q$,
  - Wenger and Wolger solved the 113−bit ECDLP over the Koblitz Curve

$$E_{/\mathbf{F}_{2^{113}}} : y^2 + xy = x^3 + x^2 + 1$$

- Choose a point $P \in E(\mathbf{F}_q)$
  - The solution time of the ECDLP depends only on the largest prime dividing the order of P.

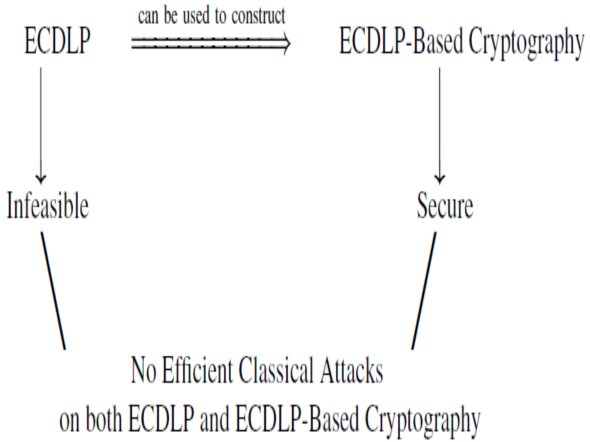
Generating Cryptographic Curves

Exercise(cryptanalysis)
Develop a function that compute the discrete logarithm (if exist). Take as parameters.

- an elliptic curve
- two points of the elliptic curve

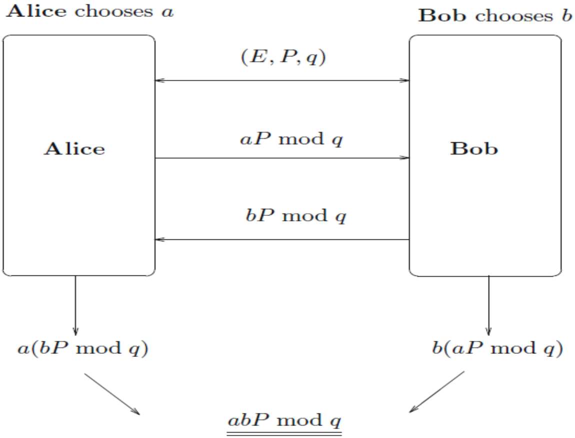
ECDLP-Based Cryptography

Basic Ideas in ECDLP-Based Cryptography



ECDLP-Based Cryptography

Elliptic Curve Diffie Hellman



Exercise :

1. Implement in GP a function with parameters.
   - an elliptic curve E over a finite field $\mathbf{F}_q$
   - a point $P \in E$ of order l.

   and output a DH shared Key.
2. Generate a shared key for Alice and Bob using
   - $E_{/\mathbf{F}_{199}} : y^2 = x^3 + x - 3$
   - $P = [1, 76] \in E$

For more details about safe elliptic curve visit

`http://safecurves.cr.yp.to/index.html`

A join work by Daniel J. Bernstein, and Tanja Lange

For implementation of Jacobian of hyperelliptic curve visit

http ://magma.maths.usyd.edu.au/magma/

the official website of magma (computer algebra)

The End

Thank you for your attention ! ! !