Quick introduction to 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, from topology or numerical analysis to physics. The three main advantages of the system are its speed, the possibility of using directly data types which are familiar to mathematicians, and its extensive algebraic number theory module.

PARI is used in three different ways:

1) as a library libpari, which can be called from an upper-level language application, for instance written in ANSI C or C++;
2) as a sophisticated programmable calculator, named gp, whose language GP contains most of the control instructions of a standard language like C;
3) the compiler gp2c translates GP code to C, and loads it into the gp interpreter. A typical script compiled by gp2c runs 3 to 10 times faster. The generated C code can be edited and optimized by hand. It may also be used as a tutorial to libpari programming.

In this short lecture we will focus to the programmable calculator, named gp. The PARI/GP is istalled on crocs.fi.muni.cz (accessible from directly from FI net, or through aisa from outside networks). PARI/GP can be also easily downloaded from http://pari.math.u-bordeaux.fr/download.html and installed in both Windows and Linux environments.

Start of GP
To start the calculator, the general command line syntax is: gp [options][-D key=val] [files]. To exit gp, type quit, \q, or <Ctrl-D> at prompt. E.g., option "-s stacksize" can be used to increase size of stack in memory.

How to use help:

1) describe function ?function (e.g. ?isprime),
2) extended description ??keyword (e.g., ??sqrt)
3) list of relevant help topics ???pattern (e.g., ???"elliptic curve"),

Other basic commands/functions are described in PARI/GP reference card at http://pari.math.u-bordeaux.fr/pub/pari/manuals/2.7.0/refcard.pdf. All PARI/GP manuals are available at: http://pari.math.u-bordeaux.fr/doc.html.

1. Introduction

User interaction with a gp session proceeds as follows. First, one types a sequence of characters at the gp prompt. When you hit the <Return> key, gp gets your input, evaluates it, then prints the result and assigns it to an "\history" array. The input is case-sensitive and, outside of character strings, blanks are completely ignored.

An expression is formed by combining constants, variables, operator symbols, functions and control statements. It is evaluated using the conventions about operator priorities and left to right associativity. An expression always has a value, which can be any PARI object:

```
? 1 + 1
%1 = 2             \\ an ordinary integer
? x
%2 = x             \\ a polynomial of degree 1 in the unknown x
? print("Hello")
Hello              \\ void return value
? f(x) = x^2
%3 = (x)->x^2      \\ a user function
```

In the third example, Hello is printed as a side effect, but is not the return value. The print command is a procedure, which conceptually returns nothing. But in fact procedures return a special void object, meant to be ignored (but which evaluates to 0 in a numeric context, and stored as 0 in the history or results). The final example assigns to the variable f the function x -> x^2, the alternative form f = x->x^2 achieving the same effect; the return value of a function definition is, unsurprisingly, a function object.

The gp history of results
This is not to be confused with the history of your commands, maintained by readline. The gp history contains the results they produced, in sequence. The successive elements of the history array are called %1, %2, ... As a shortcut, the latest computed expression can also be called %, the previous one %`, the one before that %`` and so on. When you suppress the printing of the result with a semicolon (;), it is still stored in the history, but its history number will not appear either. It is a better idea to assign it to a variable for later use than to mentally re-compute what its number is. Of course, on the next line, you may just use %.

The time used to compute that history entry is also stored as part of the entry and can be recovered using the %# operator: %#1, %#2, %#`; %# by itself returns the time needed to compute the last result (the one returned by %).

Remark. The history "\array" is in fact better thought of as a queue: its size is limited to 5000 entries by default, after which gp starts forgetting the initial entries. So %1 becomes unavailable as gp prints %5001. You can modify the history size using *histsize*.

Special editing characters
A GP program can of course have more than one line. Since your commands are executed as soon as you have finished typing them, there must be a way to tell gp to wait for the next line or lines of input before doing anything. There are three ways of doing this.

The first one is to use the backslash character \ at the end of the line that you are typing, just before hitting <Return>. This tells gp that what you will write on the next line is the physical continuation of what you have just written. In other words, it makes gp forget your newline character. You can type a \ anywhere. It is interpreted as above only if (apart from ignored whitespace characters) it is immediately followed by a newline. For example, you can type

```
? 3 + \
4
```

instead of typing 3 + 4.

The second one is a variation on the first, and is mostly useful when defining a user function. since an equal sign can never end a valid expression, gp disregards a newline immediately following an =.

```
? a =
123
%1 = 123
```

The third one is in general much more useful, and uses braces { and }. An opening brace { signals that you are typing a multi-line command, and newlines are ignored until you type a closing brace }. There are two important, but easily obeyed, restrictions: first, braces do not nest; second, inside an open brace-close brace pair, all input lines are concatenated, suppressing any newlines.

PARI types and operators
All PARI types are described in **section 2.3** of PARI user manual. Those that are especially interesting with respect to cryptography are: Integers, Intmods, Finite field elements, Quadratic numbers, Polmods, and sometimes Matrices. E.g., type Intmods is defined as:
Intmods (t_INTMOD). To create the image of the integer a in Z=bZ (for some non-zero integer b), type Mod(a,b); ***not a%b****. Internally, all operations are done on integer representatives belonging to [0; b - 1].*

Note that this type is available for convenience, not for speed: each elementary operation involves a reduction modulo b. If x is a t_INTMOD Mod(a,b), the following member function is defined: x.mod: return the modulus b.

GP contains many different operators, either unary (having only one argument) or binary, plus a few special selection operators. Unary operators are defined as either prefix or postfix , meaning that they respectively precede (op x) and follow (x op) their single argument. Some symbols are syntactically correct in both positions, like !, but then represent different operators: the ! symbol represents the negation and factorial operators when in prefix and postfix position respectively. Binary operators all use the (in_x) syntax x op y.

Most operators are standard (+, %, =), some are borrowed from the C language (++, <<), and a few are specific to GP (\, #). Beware that some GP operators differ slightly from their C counterparts. For instance, GP's postfix ++ returns the new value, like the prefix ++ of C, and the binary shifts <<, >> have a priority which is different from (higher than) that of their C counterparts. When in doubt, just surround everything by parentheses; besides, your code will be more legible.

The operators and their priorities are defined in **section 2.4** of user manual.

2. Let's do some experiments with Arithmetic functions

These functions are by definition functions whose natural domain of definition is either Z (or $Z_{>0}$). The way these functions are used is completely different from transcendental functions in that there are no automatic type conversions: in general only integers are accepted as arguments.

Mod(a; b): In its basic form, creates an intmod or a polmod (amod b); b must be an integer or a polynomial.

```
? t = Mod(19,17)
%1  = Mod(2, 17)
? t = Mod(2,17);t^8
%2  = Mod(1, 17)
```

lift(x; fvg): If v is omitted, lifts intmods from Z=nZ in Z, p-adics from Q_p to Q (as truncate), and polmods to polynomials. Otherwise, lifts only polmods whose modulus has main variable v.

```
? lift(Mod(5,3))
%1 = 2
? lift(Mod(x,x^2+1))
%2 = x
```

factor(x, {lim}): General factorization function. The result is a two-column matrix: the first contains the irreducibles dividing x, and the second the exponents.

```
? factor(36)
%1 =
[2 2]
[3 2]
```

eulerphi(x): Euler's (totient) function of the integer |x|.

```
? eulerphi(36)
%1 = 12
```

factorial(x): Factorial of x. The expression x! gives a result which is an integer, while factorial(x) gives a real number.

factorint(x; {flag = 0}): Factors the integer n into a product of pseudoprimes. The output is a two-column matrix as for factor: the first column contains the "\prime" divisors of n, the second one contains the (positive) exponents.

```
? factorint(561)
%1 =
[ 3 1]
[11 1]
[17 1]
```

factorint is using Shanks SQUFOF and Pollard Rho method (with modifications due to Brent), Lenstra's ECM (with modifications by Montgomery), and MPQS (the latter adapted from the LiDIA code with the kind permission of the LiDIA maintainers), as well as a search for pure powers.

gcd(x; {y}): Creates the greatest common divisor of x and y. If you also need the u and v such that x*u + y*v = gcd(x; y), use the **bezout** function. If y is omitted and x is a vector, returns the gcd of all components of x, i.e. this is equivalent to **content(x)**.

```
? gcd(9,12)
%1 = 3
```

isprime(x; {flag = 0}): True (1) if x is a prime number, false (0) otherwise. A prime number is a positive integer having exactly two distinct divisors among the natural numbers, namely 1 and itself.

This routine proves or disproves rigorously that a number is prime, which can be very slow when x is indeed prime and has more than 1000 digits, say. Use ispseudoprime to quickly check for compositeness. See also factor.

It accepts vector/matrices arguments, and is then applied componentwise.

```
? isprime(561)
%1 = 0
? nextprime(561)
%2 = 563
? isprime(563)
%3 = 1
```

prime(n): Prints Nth prime number.

```
? prime(5)
%1 = 11
? prime(10^9)
%2 = 22801763489
```

primes(n): Creates a row vector whose components are the first n prime numbers. (Returns the empty vector for n <= 0.) A t_VEC n = [a; b] is also allowed, in which case the primes in [a; b] are returned.

```
? primes(10)          \\ the first 10 primes
%1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes([0,29])      \\ the primes up to 29
%2 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes([15,30])
%3 = [17, 19, 23, 29]
```

random(x): Generate a (pseudo)random number.

getrand(s)/setrand(s): Get/set pseudorandom number generator seed.

znprimroot(n): Returns a primitive root (generator) of $(Z/nZ)^*$, whenever this latter group is cyclic ($n = 4$ or $n = 2p_k$ or $n = p_k$, where p is an odd prime and $k >= 0$). If the group is not cyclic, the result is undefined. If n is a prime power, then the smallest positive primitive root is returned. This may not be true for $n = 2p^k$, p odd.

```
      ? znprimroot(10007)
      %1 = Mod(5, 10007)
```

znlog(x; g; {o}): Discrete logarithm of x in $(Z/nZ)^*$ in base g. The result is [] when x is not a power of g. If present, o represents the multiplicative order of g; the preferred format for this parameter is [ord, factor(ord)], where ord is the order of g. This provides a definite speedup when the discrete log problem is simple.

```
      ? #
      ? p = nextprime(10^4);g = znprimroot(p);o = [p-1, factor(p-1)];
      ? for(i=1,10^4, znlog(i, g, o))
      time = 205 ms.
      ? for(i=1,10^4, znlog(i, g))
      time = 244 ms. \\ a little slower
```

ellinit(x; {D = 1}): Initialize an ell structure, associated to the elliptic curve E. E is either a 5-component vector [a_1; a_2; a_3; a_4; a_6] (Weierstrass equation: $Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$;) or a 2-component vector [a_4; a_6] (short Weierstrass equation: $Y^2 = X^3 + a_4X + a_6$) or a character string in Cremona's notation, e.g. "11a1", in which case the curve is retrieved from the elldata database if available. If the curve (seen over the domain D) is singular, fail and return an empty vector [].

```
      ? E = ellinit([0,0,0,0,1]);      \\ y^2 = x^3 + 1, over Q
      ? E = ellinit([0,1]);            \\ the same curve, short form
      ? E = ellinit("36a1");           \\ sill the same curve,
                                          Cremona's notations
```

All functions related to elliptic curves are described in section 3.5 of user manual.

3. Strings

GP variables can hold values of type character string (internal type t_STR). The general way to input a string is to enclose characters between quotes ". This is the only input construct where whitespace characters are significant: the string will contain the exact number of spaces you typed in. Besides, you can \escape" characters by putting a \ just before them; the translation is as follows

\e: <Escape>
\n: <Newline>
\t: <Tab>

For any other character x, \x is expanded to x. In particular, the only way to put a " into a string is to escape it. Thus, for instance, "\"a\"" would produce the string whose content is \a". This is definitely not the same thing as typing "a", whose content is merely the one-letter string a. You can concatenate two strings using the concat function. If either argument is a string, the other is automatically converted to a string if necessary (it will be evaluated first).

```
      ? concat("ex", 1+1)
      %1  = "ex2"
      ? a = 2; b = "ex"; concat(b, a)
      %2  = "ex2"
      ? concat(a, b)
      %3  = "2ex"
```

Some functions expect strings for some of their arguments: **print** would be an obvious example, **Str** is a less obvious but useful one.

Vecsmall(x, {n}): Transforms the object x into a row vector of type t_VECSMALL. The

dimension of the resulting vector can be optionally specified via the extra parameter n. I.e., the function converts a character string into a corresponding vector of ASCII encodings.
Strchr(x): Converts x to a string, translating each integer into a character.
```
? Strchr(97)
%1 = "a"
? Vecsmall("hello world")
%2 = Vecsmall([104, 101, 108, 108, 111, 32, 119, 111, 114, 108,
100])
? Strchr(%)
%3 = "hello world"
```
Str({x}*): Converts its argument list into a single character string (type t_STR, the empty string if x is omitted). To recover an ordinary GEN from a string, apply eval to it. The arguments of Str are evaluated in string context.
```
? x2 = 0; i = 2; Str(x, i)
%1 = "x2"
? eval(%)
%2 = 0
```

4. Programming in GP: control statements
A number of control statements are available in GP. They are simpler and have a syntax slightly different from their C counterparts, but are quite powerful enough to write any kind of program. Some of them are specific to GP, since they are made for number theorists. As usual, X will denote any simple variable name, and seq will always denote a sequence of expressions, including the empty sequence.

for(X = a; b; seq): Evaluates seq, where the formal variable X goes from a to b. Nothing is done if a > b. a and b must be in R.
```
for(x=1,11,print(isprime(x)))
0
1
1
0
1
```

forprime(p = a; {b}; seq): Evaluates seq, where the formal variable p ranges over the prime numbers between the real numbers a to b, including a and b if they are prime. More precisely, the value of p is incremented to nextprime(p + 1), the smallest prime strictly larger than p, at the end of each iteration. Nothing is done if a > b.
```
? forprime(p = 4, 10, print(p))
5
7
```

forsubgroup(H = G; {bound}; seq): Evaluates seq for each subgroup H of the abelian group G (given in SNF form or as a vector of elementary divisors).
```
? G = [2,2]; forsubgroup(H=G, 2, print(H))
[1; 1]
[1; 2]
[2; 1]
[1, 0; 1, 1]
```

while(a; seq): While a is non-zero, evaluates the expression sequence seq. The test is made before evaluating the seq, hence in particular if a is initially equal to zero the seq will not be evaluated at all.

if(a; {seq1}; {seq2}): Evaluates the expression sequence seq1 if a is non-zero, otherwise the expression seq2 . Of course, seq1 or seq2 may be empty.

```
      x = if(n % 4 == 1, y, z);
      //sets x to y if n is 1 modulo 4, and to z otherwise.
```

read("file.gp"): Reads .gp script(e.g., single function) from working directory. Example of simple file.gp content:

```
test(p) =
 { u = 2*p;
 }
```

Function definitions are described in section 2.7.1. of user manual.

5. Assignment [10 points]

Write a set of commands that will do in PARI/GP a following task:

1) Randomly generate RSA key pair (private/public keys). Modulus n should be long enough for encryption of our two plaintexts (see below).
2) Implement RSA encryption and use public key to encrypt a message: (decimal number) 181.
3) Implement RSA decryption and use private key to decrypt a message encrypted in previous step.
4) Check that the message before encryption and message after decryption is identical.
5) Modify input/output of your RSA to work with text strings (i.e., add transformation of text string to/from a number). Repeat steps 1-4 and encrypt/decrypt a text string "PV181".