A Tutorial

for

PARI / GP

(version 2.19.1)

The PARI Group

Institut de Mathématiques de Bordeaux, UMR 5251 du CNRS.
Université de Bordeaux, 351 Cours de la Libération
F-33405 TALENCE Cedex, FRANCE
e-mail: pari@math.u-bordeaux.fr

Home Page:
https://pari.math.u-bordeaux.fr/

Copyright © 2000–2025 The PARI Group

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions, or translations, of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

PARI/GP is Copyright © 2000–2025 The PARI Group

PARI/GP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY WHATSOEVER.

Table of Contents

1. Greetings! . . . 4
2. Warming up . . . 7
3. The Remaining PARI Types . . . 9
4. Elementary Arithmetic Functions . . . 14
5. Performing Linear Algebra . . . 15
6. Using Transcendental Functions . . . 17
7. Using Numerical Tools . . . 20
8. Polynomials . . . 22
9. Power Series . . . 25
10. Working with Elliptic Curves . . . 25
11. Working in Quadratic Number Fields . . . 30
12. Working in General Number Fields . . . 35
    - 12.1. Defining equations and elements in the field . . . 35
    - 12.2. Ideals . . . 39
    - 12.3. Class Groups and Units, `bnf` . . . 41
    - 12.4. Class Field Theory, `bnr` . . . 43
    - 12.5. Galois Theory over $\mathbf{Q}$ . . . 44
    - 12.6. Creating and Listing Number Fields . . . 45
13. Working with Associative Algebras . . . 46
    - 13.1. Arbitrary Associative Algebras . . . 46
    - 13.2. Central Simple Algebras over Number Fields . . . 49
14. Plotting . . . 52
    - 14.1. Function graphs . . . 53
    - 14.2. Low-level plotting . . . 54
15. GP Programming . . . 59

This booklet is a guided tour and a tutorial to the gp calculator. Many examples will be given, but each time a new function is used, the reader should look at the appropriate section in the *User's Manual to PARI/GP* for detailed explanations. This chapter can be read independently, for example to get acquainted with the possibilities of gp without having to read the whole manual. At this point.

1. Greetings!.

So you are sitting in front of your workstation (or terminal, or PC...), and you type gp to get the program started (or click on the relevant icon, or select some menu item). It says hello in its particular manner, and then waits for you after its prompt, initially ? (or something like **gp >**). Type

```
2 + 2
```

What happens? Maybe not what you expect. First of all, of course, you should tell gp that your input is finished, and this is done by hitting the Return (or Newline, or Enter) key. If you do exactly this, you will get the expected answer. However some of you may be used to other systems like Gap, Macsyma, Magma or Maple. In this case, you will have subconsciously ended the line with a semicolon ";" before hitting Return, since this is how it is done on those systems. In that case, you will simply see gp answering you with a smug expression, i.e. a new prompt and no answer! This is because a semicolon at the end of a line tells gp not to print the result (it is still stored in the result history). You will certainly want to use this feature if the output is several pages long. Try

```
27 * 37
```

Wow! even multiplication works. Actually, maybe those spaces are not necessary after all. Let's try 27*37. Seems to be ok. We will still insert them in this document since it makes things easier to read, but as gp does not care about them, you don't have to type them all.

Now this session is getting lengthy, so the second thing one needs to learn is to quit. Each system has its quit signal. In gp, you can use quit or \q (backslash q), the q being of course for quit. Try it.

Now you've done it! You're out of gp, so how do you want to continue studying this tutorial? Get back in please.

Let's get to more serious stuff. I seem to remember that the decimal expansion of 1/7 has some interesting properties. Let's see what gp has to say about this. Type

```
1 / 7
```

What? This computer is making fun of me, it just spits back to me my own input, that's not what I want!

Now stop complaining, and think a little. Mathematically, 1/7 is an element of the field **Q** of rational numbers, so how else but 1/7 can the computer give the answer to you? Well maybe 2/14 or 7^{-1}, but why complicate matters? Seriously, the basic point here is that PARI, hence gp, will almost always try to give you a result which is as precise as possible (we will see why "almost" later). Hence since here the result can be represented exactly, that's what it gives you.

But I still want the decimal expansion of 1/7. No problem. Type one of the following:

```
1./ 7
1 / 7.
1./ 7.
1 / 7 + 0.
```

Immediately a number of decimals of this fraction appear (exactly 38), and the repeating pattern is $142,857$. The reason is that you have included in the operations numbers like `0.`, `1.` or `7.` which are *imprecise* real numbers, hence `gp` cannot give you an exact result.

Why 38 decimals by the way? Well, it is the default initial precision. This has been chosen so that the computations are very fast, and gives already more accuracy than conventional double precision floating point operations. The precise value is 128 bits of accuracy by default, which translates to $128 \log_{10} 2 \approx 38.53$ decimal digits, rounded to 38. Of course, you can extend the precision (almost) as much as you like as we will see in a moment.

I'm getting bored, why don't we get on with some more exciting stuff? Well, try `exp(1)`. Presto, comes out the value of e to 38 digits. Try `log(exp(1))`. Well, we get a floating point number and not an exact 1, but pretty close! That's what you lose by working numerically.

What could we try now? Hum, `pi`? The answer is not that enlightening. `Pi`? Ah. This works better. But let's remember that `gp` distinguishes between uppercase and lowercase letters. `pi` was as meaningless to it as `stupid garbage` would have been: in both cases `gp` will just create a variable with that funny unknown name you just used. Try it! Note that it is actually equivalent to type `stupidgarbage`: all spaces are suppressed from the input. In the `27 * 37` example it was not so conspicuous as we had an operator to separate the two operands. This has important consequences for the writing of `gp` scripts. More about this later.

By the way, you can ask `gp` about any identifier you think it might know about: just type it, prepending a question mark "`?`". Try `?Pi` and `?pi` for instance. On most systems, an extended online help should be available: try doubling the question mark to check whether it's the case on yours: `??Pi`. In fact the `gp` header already gave you that information if it was the case, just before the copyright message. As well, if it says something like "`readline enabled`" then you should have a look at the `readline` introduction in the User's Manual before you go on (try `??readline`): it will be easier to type in examples and correct typos after you've done that.

Please explore by typing `?*` (list of all help topics) then trying out `?`n and `??`n for various integers n covering the various topics.

Now try `exp(Pi * sqrt(163))`. Hmmm, we suspect that the last digit may be wrong, can this really be an integer? This is the time to change precision. Type `\p 50`, then try `exp(Pi * sqrt(163))` again. We were right to suspect that the last decimal was incorrect, since we get quite a few nines in its place, but it is now convincingly clear that this is not an integer. Maybe it's a bug in PARI, and the result is really an integer? Type

```
(log(%) / Pi)^2
```

immediately after the preceding computation; `%` means the result of the last computed expression. More generally, the results are numbered `%1, %2, ...` *including* the results that you do not want to see printed by putting a semicolon at the end of the line, and you can evidently use all these quantities in any further computations. The result seems to be indistinguishable from 163, hence it does not seem to be a bug.

In fact, it is known that $\exp(\pi * \sqrt{n})$ not only is not an integer or a rational number, but is even a transcendental number when n is a nonzero rational number.

So gp is just a fancy calculator, able to give me more decimals than I will ever need? Not so, gp is incredibly more powerful than an ordinary calculator, independently of its arbitrary precision possibilities.

Additional comments. (you are supposed to skip this at first, and come back later)

1) If you are a PARI old timer, say the last version of PARI you used was released around 1996, you have certainly noticed already that many many things changed between the older 1.39.xx versions and this one. Conspicuously, most function names have been changed. To know how a specific function was changed, type `whatnow(function)`.

2) It seems that the text implicitly says that as soon as an imprecise number is entered, the result will be imprecise. Is this always true? There is a unique exception: when you multiply an imprecise number by the exact number 0, you will get the exact 0. Compare `0 * 1.4` and `0. * 1.4`.

3) Not only can the number of decimal places of real numbers be large, but the number of digits of integers also. Try `1000!`. It is never necessary to tell gp in advance the size of the integers that it will encounter. The same is true for real numbers, although most computations with floating point assume a default precision and truncate their results to this accuracy; initially 38 decimal digits, but we may change that with `\p` of course.

4) Come back to 38 digits of precision (`\p 38`), and type `exp(100)`. As you can see the result is printed in exponential format. This is because gp never wants you to believe that a result is correct when it is not. We are working with 38 digits of precision, but the integer part of exp(100) has 44 decimal digits. Hence if gp had dutifully printed out 44 digits, the last few digits would have been wrong. Hence gp wants to print only 38 significant digits, but to do so it has to print in exponential format.

5) There are two ways to avoid this. One is of course to increase the precision. Let's try it. To give it a wide margin, we set the precision to 50 decimals. Then we recall our last result (`%` or `%n` where `n` is the number of the result). What? We still have an exponential format! Do you understand why?

Again let's try to see what's happening. The number you recalled had been computed only to 38 decimals, and even if you set the precision to 1000 decimals, gp knows that your number has only 38 digits of accuracy but an integral part with 44 digits. So you haven't improved things by increasing the precision. Or have you? What if we retype `exp(100)` now that we have 50 digits? Try it. Now we no longer have an exponential format.

6) What if I forget what the current precision is and I don't feel like counting all the decimals? Well, you can type `\p` by itself. You may also learn about gp internal variables (and change them!) using `default`. Type `default(realprecision)`, then `default(realprecision, 38)`. Huh? In fact this last command is strictly equivalent to `\p 38`! (Admittedly more cumbersome to type.) There are more "defaults" than just `format` and `realprecision`: type `default` by itself now, they are all there.

7) Note that the `default` command reacts differently according to the number of input arguments. This is not an uncommon behavior for gp functions. You can see this from the online help, or the complete description in Chapter 3: any argument surrounded by braces `{}` in the function prototype is optional, which really means that a *default* argument will be supplied by gp. You can then check out from the text what effect a given value will have, and in particular the default one.

8) Try the following: starting in precision 38, type first `default(format, "e0.100")`, then `exp(1)`. Where are my 100 significant digits? Well, `default(format,)` only changes the output format, but *not* the default precision. On the other hand, the `\p` command changes both the precision and the output format.

2. Warming up.

Another thing you better get used to pretty fast is error messages. Try typing `1/0`. Could not be clearer. But why has the prompt become funny, turning from `?` to `break>` ? When an error occurs, we enter a so-called *break loop*, where you get a chance, e.g to inspect (and save!) values of variables before the prompt returns and all computations so far are lost. In fact you can run an arbitrary command at this point, and this mechanism is a tremendous help in debugging. To get out of the break loop, type `break`, as instructed in the error message last line.

Comment. You can enter the break loop at any time using `Control-C`: this freezes the current computation and gets you a new prompt so that you may e.g., increase debugging level, inspect or modify variables (again, run arbitrary commands), before letting the program go on.

Now, back to our favorite example, in precision 38, type

```
floor(exp(100))
```

`floor` is the mathematician's integer part, not to be confused with `truncate`, which is the computer scientist's: `floor(-3.4)` is equal to -4 whereas `truncate(-3.4)` is equal to -3. You get a more cryptic error message, which you would immediately understand if you had read the additional comments of the preceding section. Since you were told not to read them, here's the explanation: `gp` is unable to compute the integer part of `exp(100)` given only 38 decimals of accuracy, since it has 44 digits.

Some error messages are more cryptic and sometimes not so easy to understand. For instance, try `log(x)`. It simply tells you that `gp` does not understand what `log(x)` is, although it does know the `log` function, as `?log` will readily tell us.

Now let's try `sqrt(-1)` to see what error message we get now. Haha! `gp` even knows about complex numbers, so impossible to trick it that way. Similarly, try typing `log(-2)`, `exp(I*Pi)`, `I^I`... So we have a lot of real and complex analysis at our disposal. There always is a specific branch of multivalued complex transcendental functions which is taken, specified in the manual. Again, beware that `I` and `i` are not the same thing. Compare `I^2` with `i^2` for instance.

Just for fun, let's try `6*zeta(2) / Pi^2`. Pretty close, no?

Now `gp` didn't seem to know what `log(x)` was, although it did know how to compute numerical values of `log`. This is annoying. Maybe it knows the exponential function? Let's give it a try. Type `exp(x)`. What's this? If you had any experience with other computer algebra systems, the answer should have simply been `exp(x)` again. But here the answer is the Taylor expansion of the function around `x` $= 0$, to 17 terms. Note the `O(x^17)` which ends the series, and which is trademark of power series in `gp`. It is the familiar "big–oh" notation of analysis. Why 17 terms? This is governed by the `seriesprecision`, which can be changed by typing `\ps` n or `default(seriesprecision,` n`)` where n is the number of terms that you want in your power series and is 16 by default. Converting a polynomial or rational function to a power series will yield 16 significant terms: so x gets converted to $x+O(x^{17})$; this is completely analogous to `realprecision` when an exact integer

or rational number is converted to a floating point real number. Then we take the exponential of this new object and, since it has positive valuation, we can actually deduce 17 significant terms from the given 16. This is in keeping with PARI's philosophy of always returning a result which is as precise as possible from a given input.

You thus automatically get the Taylor expansion of any function that can be expanded around 0, and incidentally this explains why we weren't able to do anything with `log(x)` which is not defined at 0. (In fact `gp` knows about Laurent series, but `log(x)` is not meromorphic either at 0.) If we try `log(1+x)`, then it works, but this time we *lose* one significant term: the result is $x - 1/2x^2 + \ldots + 1/15x^{15} + O(x^{16})$. (Do you understand why ?)

But what if we wanted the expansion around a point different from 0? Well, you're able to change x into $x + a$, aren't you? So for instance you can type `log(x+2)` to have the expansion of `log` around `x` $= 2$. As exercises you can try

```
cos(x)
cos(x)^2 + sin(x)^2
exp(cos(x))
gamma(1 + x)
exp(exp(x) - 1)
1 / tan(x)
```

for different values of `serieslength` (change it using `\ps` *newvalue*).

Let's try something else: type `(1 + x)^3`. No `O(x)` here, since the result is a polynomial. Haha, but I have learnt that if you do not take exponents which are integers greater or equal to 0, you obtain a power series with an infinite number of nonzero terms. Let's try. Type `(1 + x)^(-3)` (the parentheses around `-3` are not necessary but make things easier to read). Surprise! Contrary to what we expected, we don't get a power series but a rational function. Again this is for the same reason that `1 / 7` just gave you 1/7: the result being exact, PARI doesn't see any reason to make it inexact.

But I still want that power series. To obtain it, you can do as in the 1/7 example and force a conversion using the $O(x^n)$ notation:

```
(1 + x)^(-3) + O(x^16)
(1 + x)^(-3) * (1 + O(x^16))
(1 + x + O(x^16))^(-3)
```

(Not on this example, but there is a difference between the first 2 methods. Do you spot it?) Better yet, use the series constructor which transforms any object into a power series, using the current `seriesprecision`, and simply type

```
Ser( (1 + x)^(-3) )
```

Now try `(1 + x)^(1/2)`: we obtain a power series, since the result is an object which PARI does not know how to represent exactly. (We could teach PARI about algebraic functions, but then take `(1 + x)^Pi` as another example.) This gives us still another solution to our preceding exercise: we can type `(1 + x)^(-3.)`. Since `-3.` is not an exact quantity, PARI has no means to know that we are dealing with a rational function, and will instead give you the power series, this time with real instead of integer coefficients.

To summarize, in this section we have seen that in addition to integers, real numbers and rational numbers, PARI can handle complex numbers, polynomials, rational functions and power

series. A large number of functions exist which handle these types, but in this tutorial we will only look at a few.

Additional comments. (as before, you are supposed to skip this at first reading)

1) In almost all cases, there is no loss of information in PARI output: what you see is all that PARI knows about the object, and you can happily copy-paste it into another session. There are exceptions, though. Type `n = 3 + 0*x`, then `n` is not the integer 3 but a constant polynomial equal to $3x^0$. Check it with `type(n)`.

However, it *looks* like an integer without being one, and this may cause some confusion in programs which actually expect integers. Hence if you try to `factor(n)`, you obtain an empty factorization ! (Because, once considered as a polynomial, `n` is a unit in $\mathbf{Q}[x]$.)

If you try to apply more general arithmetic functions, say the Euler totient function (known as `eulerphi` to `gp`), you get an error message worrying about integer arguments. You would have guessed yourself, but the message is difficult to understand since 3 looks like a genuine integer! Please make sure you understand the above, it is a common source of incomprehension.

2) If you want the final expression to be in the simplest form possible (for example before applying an arithmetic function, or simply because things will go faster afterwards), apply the function `simplify` to the result. This is done automatically at the end of a `gp` command, but *not* in intermediate expressions. Hence `n` above is not an integer, but the final result stored in the output history is! So if you type `type(%)` instead of `type(n)` the answer is `t_INT`, adding to the confusion.

3) As already stated, power series expansions are always implicitly around `x` = 0. When we want them around `x = a`, we replace `x` by `z + a` in the function we want to expand. For complicated functions, it may be simpler to use the substitution function `subst`. For example, if `p = 1 / (x^ 4 + 3*x^3 + 5*x^2 - 6*x + 7)`, you may not want to retype this, replacing `x` by `z + a`, so you can write `subst(p, x, z+a)` (look up the exact description of the `subst` function).

4) The valuation at `x` = 0 for a power series `p` is obtained as `valuation(p, x)`.

3. The Remaining PARI Types.

Let's talk some more about the basic PARI types.

Type `p = x * exp(-x)`. As expected, you get the power series expansion to 17 terms (if you have not changed the default). Now type `pr = serreverse(p)`. You are asking here for the *reversion* of the power series `p`, in other words the expansion of the inverse function. This is possible only for power series whose first nonzero coefficient is that of x^1. To check the correctness of the result, you can type `subst(p, x, pr)` or `subst(pr, x, p)` and you should get back `x + O(x^ 18)`.

Now the coefficients of `pr` obey a very simple formula. First, we would like to multiply the coefficient of `x^n` by `n!` (in the case of the exponential function, this would simplify things considerably!). The PARI function `serlaplace` does just that. So type `ps = serlaplace(pr)`. The coefficients now become integers, which can be immediately recognized by inspection. The coefficient of x^n is now equal to n^{n-1}. In other words, we have

$$\mathtt{pr} = \sum_{n \geq 1} \frac{n^{n-1}}{n!} X^n.$$

Do you know how to prove this? (The proof is difficult.)

Of course PARI knows about vectors (rows and columns are distinguished, even though mathematically there is no difference) and matrices. Type for example `[1,2,3,4]`. This gives the row vector whose coordinates are 1, 2, 3 and 4. If you want a column vector, type `[1,2,3,4]~`, the tilde meaning of course transpose. You don't see much difference in the output, except for the tilde at the end. However, now type `\B`: lo and behold, the column vector appears as a proper vertical thingy now. The `\B` command is used mainly for this purpose. The length of a vector is given by, well `length` of course. The shorthand "cardinality" notation `#v` for `length(v)` is also available, for instance `v[#v]` is the last element of `v`.

Type `m = [a,b,c; d,e,f]`. You have just entered a matrix with 2 rows and 3 columns. Note that the matrix is entered by *rows* and the rows are separated by semicolons "`;`". The matrix is printed naturally in a rectangle shape. If you want it printed horizontally just as you typed it, type `\a`, or if you want this type of printing to be the permanent default type `default(output, 0)`. Type `default(output, 1)` if you want to come back to the original output mode.

Now type `m[1,2]`, `m[1,]`, `m[,2]`. Are explanations necessary? (In an expression such as `m[j,k]`, the `j` always refers to the row number, and the `k` to the column number, and the first index is always 1, never 0. This default cannot be changed.)

Even better, type `m[1,2] = 5; m`. The semicolon also allows us to put several instructions on the same line; the final result is the output of the last statement on the line. Now type `m[1,] = [15,-17,8]`. No problem. Finally type `m[,2] = [j,k]`. You have an error message since you have typed a row vector, while `m[,2]` is a column vector. If you type instead `m[,2] = [j,k]~` it works.

Type now `h = mathilbert(20)`. You get the so-called "Hilbert matrix" whose coefficient of row i and column j is equal to $(i+j-1)^{-1}$. Incidentally, the matrix `h` takes too much room. If you don't want to see it, simply type a semi-colon "`;`" at the end of the line (`h = mathilbert(20);`). This is an example of a "precomputed" matrix, built into PARI. We will see a more general construction later.

What is interesting about Hilbert matrices is that first their inverses and determinants can be computed explicitly (and the inverse has integer coefficients), and second they are numerically very unstable, which make them a severe test for linear algebra packages in numerical analysis. Of course with PARI, no such problem can occur: since the coefficients are given as rational numbers, the computation will be done exactly, so there cannot be any numerical error. Try it. Type `d = matdet(h)`. The result is a rational number (of course) of numerator equal to 1 and denominator having 226 digits. How do I know, by the way? Well, type `sizedigit(1/d)`. Or `#Str(1/d)`. (The length of the character string representing the result.)

Now type `hr = 1.* h;` (do not forget the semicolon, we don't want to see the result!), then `dr = matdet(hr)`. You notice two things. First the computation, is much faster than in the rational case. (If your computer is too fast for you to notice, try again with `h = mathilbert(40)`, or even some larger value.) The reason for this is that PARI is handling real numbers with 38 digits of accuracy, while in the rational case it is handling integers having up to 226 decimal digits.

The second, more important, fact is that the result is terribly wrong. If you compare with `1.*d` computed earlier, which is the correct answer, you will see that few decimals agree! (None agree if you replaced 20 by 40 as suggested above.) This catastrophic instability is as already mentioned one of the characteristics of Hilbert matrices. In fact, the situation is worse than that. Type `norml2(1/h - 1/hr)` (the function `norml2` gives the square of the L^2 norm, i.e. the sum of

the squares of the coefficients). The result is larger than 10^{32}, showing that some coefficients of `1/hr` are wrong by as much as 10^{16}. To obtain the correct result after rounding for the inverse, we have to use a default precision of 57 digits (try it).

Although vectors and matrices can be entered manually, by typing explicitly their elements, very often the elements satisfy a simple law and one uses a different syntax. For example, assume that you want a vector whose i-th coordinate is equal to i^2. No problem, type for example `vector(10,i, i^2)` if you want a vector of length 10. Similarly, if you type

```
matrix(5,5, i,j, 1 / (i+j-1))
```

you will get the Hilbert matrix of order 5, hence the `mathilbert` function is in fact redundant. The `i` and `j` represent dummy variables which are used to number the rows and columns respectively (in the case of a vector only one is present of course). You must not forget, in addition to the dimensions of the vector or matrix, to indicate explicitly the names of these variables. You may omit the variables and the final expression to get zero entries, as in `matrix(10,20)`.

Warning. The letter `I` is reserved for the complex number equal to the square root of -1. Hence it is forbidden to use it as a variable. Try typing `I = 2`, the error message that you get clearly indicates that `gp` does not consider `I` as a variable. There are other reserved variable names: `Pi`, `Euler`, `Catalan` and `oo`. All function names are forbidden as well. On the other hand there is nothing special about `i`, `pi`, `euler` or `catalan`.

When creating vectors or matrices, it is often useful to use Boolean operators and the `if()` statement. Indeed, an `if` expression has a value, which is of course equal to the evaluated part of the `if`. So for example you can type

```
matrix(8,8, i,j, if ((i-j)%2, 1, 0))
```

to get a checkerboard matrix of `1` and `0`. Note however that a vector or matrix must be *created* first before being used. For example, it is possible to write

```
v = vector(5);
for (i = 1, 5, v[i] = 1/i)
```

but this would fail if the vector `v` had not been created beforehand. Of course, the above example is better written as

```
v = vector(5, i, 1/i);
```

Another useful way to create vectors and matrices is to extract them from larger ones. For instance, if `h` is the 20×20 Hilbert matrix as above,

```
h = mathilbert(20);
h[11..20, 11..20]
```

is its lower right quadrant.

The last PARI types which we have not yet played with are closely linked to number theory. People not interested in number theory can skip ahead.

The first is the type "integer–modulo". Let us see an example. Type

```
n = 10^15 + 3
```

We want to know whether this number is prime or not. Of course we could make use of the built-in facilities of PARI, but let us do otherwise. We first trial divide by the built-in table of

primes. We slightly cheat here and use a variant of the function `factor` which does exactly this. So type `factor(n, 200000)`. The last argument tells `factor` to trial divide up to the given bound and stop at this point. Set it to 0 to trial divide by the full set of built-in primes, which goes up to $2^{20} = 1,048,576$ by default.

As for all factoring functions, the result is a 2 column matrix: the first column gives the primes and the second their exponents. Here we get a single row, telling us that if primes stopped at $200,000$ as we made `factor` believe, then `n` would be prime. (Or is that a contradiction?) More seriously, `n` is not divisible by any prime up to $200,000$.

We could now trial divide further, or cheat and call the PARI function `factor` without the optional second argument, but before we do this let us see how to get an answer ourselves.

By Fermat's little theorem, if n is prime we must have $a^{n-1} \equiv 1 \pmod{n}$ for all a not divisible by n. Hence we could try this with $a = 2$ for example. But 2^{n-1} is a number with approximately $3 \cdot 10^{14}$ digits, hence impossible to write down, let alone to compute. But instead type `a = Mod(2,n)`. This creates the number 2 considered now as an element of the ring $R = \mathbf{Z}/n\mathbf{Z}$. The elements of R, called intmods, can always be represented by numbers smaller than `n`, hence small. Fermat's theorem can be rewritten $a^{n-1} = \texttt{Mod}(1, n)$ in the ring R, and this can be computed very efficiently. Elements of R may be lifted back to $\mathbf{Z}$ with either `lift` or `centerlift`. Type `a^(n-1)`. The result is definitely *not* equal to `Mod(1,n)`, thus *proving* that `n` is not a prime. If we had obtained `Mod(1,n)` on the other hand, it would have given us a hint that `n` is maybe prime, but not a proof.

To find the factors is another story. In this case, the integer n is small enough to let trial division run to completion. Type `#` to turn on the `gp` timer, then

```
for (i = 2, ceil(sqrt(n)), if (n%i==0, print(i); break))
```

This should take less than 2 seconds. In general, one must use less naive techniques than trial division, or be very patient. Type `fa = factor(n)` to let the factoring engine find all prime factors. You may stop the timer by typing `#` again.

Note that, as is the case with most "prime"-producing functions, the "prime" factors given by `factor` are only strong pseudoprimes, and not *proven* primes. Use `isprime( fa[,1] )` to rigorously prove primality of the factors. The latter command applies `isprime` to all entries in the first column of `fa`, i.e to all pseudoprimes, and returns the column vector of results: all equal to 1, so our pseudoprimes were true primes. In fact, it has been checked that the strong pseudoprimes output by `factor` (Baillie-Pomerance-Selfridge-Wagstaff pseudoprimes, without small divisors) are true primes at least up to 2^{64}, and no explicit counter-example is known.

The second specifically number-theoretic type is the p-adic numbers. I have no room for definitions, so please skip ahead if you have no use for such beasts. A p-adic number is entered as a rational or integer valued expression to which is added `O(p^n)`, or simply `O(p)` if `n` $= 1$, where `p` is the prime and `n` the p-adic precision. Note that you have to explicitly type in `3^2` for instance, `9` will not do. Unless you want to cheat `gp` into believing that `9` is prime, but you had better know what you are doing in this case: most computations will yield a wrong result.

Apart from the usual arithmetic operations, you can apply a number of transcendental functions. For example, type `n = 569 + O(7^8)`, then `s = sqrt(n)`, you obtain one of the square roots of `n`; to check this, type `s^2 - n`). Type now `s = log(n)`, then `e = exp(s)`. If you know about p-adic logarithms, you will not be surprised that `e` is not equal to `n`. Type `(n/e)^6`: `e` is in fact equal to `n` times the $(p-1)$-st root of unity `teichmuller(n)`.

Incidentally, if you want to get back the integer 569 from the p-adic number `n`, type `lift(n)` or `truncate(n)`.

The third number-theoretic type is the type "quadratic number". This type is specially tailored so that we can easily work in a quadratic extension of a base field, usually $\mathbf{Q}$. It is a generalization of the type "complex". To start, we must specify which quadratic field we want to work in. For this, we use the function `quadgen` applied to the *discriminant* `d` (as opposed to the radicand) of the quadratic field. This returns a number equal to $(\mathtt{d}+a)/2$ where a is equal to 0 or 1 according to whether `d` is even or odd. The function `quadgen` takes an extra parameter which is how the number will be printed. To avoid confusion, this number should be set to a variable of the same name, i.e. do `w = quadgen(d, 'w)`.

So type `w = quadgen(-163,'w)`, then `charpoly(w)` which asks for the characteristic polynomial of `w`. The result shows what `w` will represent. You may ask for `1.*w` to see which root of the quadratic has been taken, but this is rarely necessary. We can now play in the field $\mathbf{Q}(\sqrt{-163})$. Type for example `w^10`, `norm(3 + 4*w)`, `1 / (4+w)`. More interesting, type `a = Mod(1,23) * w` then `b = a^264`. This is a generalization of Fermat's theorem to quadratic fields. If you do not want to see the modulus 23 all the time, type `lift(b)`.

Another example: type `p = x^2 + w*x + 5*w + 7`, then `norm(p)`. We thus obtain the quartic equation over $\mathbf{Q}$ corresponding to the relative quadratic extension over $\mathbf{Q}(\mathtt{w})$ defined by `p`.

On the other hand, if you type `wr = sqrt(w^2)`, do not expect to get back `w`. Instead, you get the numerical value, the function `sqrt` being considered as a "transcendental" function, even though it is algebraic. Type `algdep(wr,2)`: this looks for algebraic relations involving the powers of `w` up to degree 2. This is one way to get `w` back. Similarly, type `algdep(sqrt(3*w + 5), 4)`. See the user's manual for the function `algdep`.

The fourth number-theoretic type is the type "polynomial–modulo", i.e. polynomial modulo another polynomial. This type is used to work in general algebraic extensions, for example elements of number fields (if the base field is $\mathbf{Q}$), or elements of finite fields (if the base field is $\mathbf{Z}/p\mathbf{Z}$ for a prime p). In a sense it is a generalization of the type quadratic number. The syntax used is the same as for intmods. For example, instead of typing `w = quadgen(-163,'w)`, you can type

```
w = Mod(x, quadpoly(-163))
```

Then, exactly as in the quadratic case, you can type `w^10`, `norm(3 + 4*w)`, `1 / (4+w)`, `a = Mod(1,23)*w`, `b = a^264`, obtaining of course the same results. (Type `lift(...)` if you don't want to see the polynomial `x^2 - x + 41` repeated all the time.) Of course, you can work in any degree, not only quadratic. For the latter, the corresponding elementary operations will be a little slower than with quadratic numbers. Start the timer, then compare

```
w = quadgen(-163,'w); W = Mod(x, quadpoly(-163));
a = 2 + w;            A = 2 + W;
b = 3 + w;            B = 3 + W;
for (i=1,10^6, a+b)
for (i=1,10^6, A+B)
for (i=1,10^6, a*b)
for (i=1,10^6, A*B)
for (i=1,10^6, a/b)
for (i=1,10^6, A/B)
```

Don't retype everything, use the arrow keys!

There is however a slight difference in behavior. Keeping our polmod `w`, type `1.*w`. As you can see, the result is not the same. Type `sqrt(w)`. Here, we obtain a vector with 2 components, the two components being the principal branch of the square root of all the possible embeddings of `w` in $\mathbf{C}$. More generally, if `w` was of degree n, we would get an n-component vector, and similarly for all transcendental functions.

We have at our disposal the usual arithmetic functions, plus a few others. Type `a = Mod(x, x^3 - x - 1)` defining a cubic extension. We can for example ask for `b = a^5`. Now assume we want to express `a` as a polynomial in `b`. This is possible since `b` is also a generator of the same field. No problem, type `modreverse(b)`. This gives a new defining polynomial for the same field, i.e. the characteristic polynomial of `b`, and expresses `a` in terms of this new polmod, i.e. in terms of `a`. We will see this in more detail in the number field section.

An important special case of the above construction allows to work in finite fields, by choosing an irreducible polynomial T of degree f over $\mathbf{F}_p$ and considering $\mathbf{F}_p[t]/(T)$. As in

```
T = ffinit(5, 6, 't); \\  degree 6, irreducible over F_5
g = Mod(t, T)
```

Try a few elementary operations involving g, such as g^{100}. This special case of `t_POLMOD`s is in fact so important that we now introduce a final dedicated number theoretical type `t_FFELT`, for "finite field element", to simplify work with finite fields: `g = ffgen(5^6, 't)` computes a suitable polynomial T as above and returns the generator $t \bmod T(t)$. This has major advantages over the generic `t_POLMOD` solution: elements are printed in a simplified way (in lifted form), and functions can assume that T is indeed irreducible. A few dedicated functions `ffprimroot` (analog of `znprimroot`), `fforder` (analog of `znorder`), `fflog` (analog of `znlog`) are available. Rational expressions in the variable t can be mapped to such a finite field by substituting t by g, for instance

```
? g = ffgen(5^6, 't);
? g.mod  \\  irreducible over F_5, defines F_5^6
%2 = t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
? Q = x^2 + t*x + 1
? factor(subst(Q,t,g))
%3 =
[    x + (t^5 + 3*t^4 + t^3 + 4*t + 1) 1]

[x + (4*t^5 + 2*t^4 + 4*t^3 + 2*t + 4) 1]
```

factors the polynomial $Q \in \mathbf{F}_{5^6}[x]$, where $\mathbf{F}_{5^6} = \mathbf{F}_5[t]/(\texttt{g.mod})$.

4. Elementary Arithmetic Functions.

Since PARI is aimed at number theorists, it is not surprising that there exists a large number of arithmetic functions. We have already seen several, such as `factor`. Note that `factor` handles not only integers, but also univariate polynomials. Type for example `factor(x^200 - 1)`. You can also factor a polynomial over a finite field or a number field.

Evidently, you have functions for computing GCD's (`gcd`), extended GCD's (`gcdext`), solving the Chinese remainder theorem (`chinese`) and so on.

In addition to the factoring facilities, you have a few functions related to primality testing such as `isprime`, `ispseudoprime`, `precprime`, and `nextprime`. As previously mentioned, only

strong pseudoprimes are produced by the latter two (they pass the `ispseudoprime` test); the more sophisticated primality tests in `isprime`, being so much slower, are not applied by default.

We also have the usual multiplicative arithmetic functions: the Möbius μ function (`moebius`), the Euler ϕ function (`eulerphi`), the ω and Ω functions (`omega` and `bigomega`), the σ_k functions (`sigma`), which compute sums of k-th powers of the positive divisors of a given integer, etc...

You can compute continued fractions. For example, type `\p 1000`, then `contfrac(exp(1))`: you obtain the continued fraction of the base of natural logarithms, which as you can see obeys a very simple pattern. Can you prove it?

In many cases, one wants to perform some task only when an arithmetic condition is satisfied. `gp` gives you the following functions: `isprime` as mentioned above, `issquare`, `isfundamental` to test whether an integer is a fundamental discriminant (i.e. 1 or the discriminant of a quadratic field), and the `forprime`, `fordiv` and `sumdiv` loops. Assume for example that we want to compute the product of all the divisors of a positive integer `n`. The easiest way is to write

```
p = 1; fordiv(n,d, p *= d); p
```

(There is a simple formula for this product in terms of n and the number of its divisors: find and prove it!) The notation `p *= d` is just a shorthand for `p = p * d`.

If we want to know the list of primes p less than 1000 such that 2 is a primitive root modulo p, one way would be to write:

```
forprime(p=3,1000, if (znprimroot(p) == 2, print(p)))
```

Note that this assumes that `znprimroot` returns the smallest primitive root, and this is indeed the case. Had we not known about this, we could have written

```
forprime(p=3,1000, if (znorder(Mod(2,p)) == p-1, print(p)))
```

(which is actually faster since we only compute the order of 2 in $\mathbf{Z}/p\mathbf{Z}$, instead of looking for a generator by trying successive elements whose orders have to be computed as well.) Once we know a primitive root g, we can write any nonzero element of $\mathbf{Z}/p\mathbf{Z}$ as g^x for some unique x in $\mathbf{Z}/(p-1)\mathbf{Z}$. Computing such a discrete logarithm is a hard problem in general, performed by the function `znlog`.

Arithmetic functions related to quadratic fields, binary quadratic forms and general number fields will be seen in the next sections.

5. Performing Linear Algebra.

The standard linear algebra routines are available: `matdet`, `mateigen` (eigenvectors), `matker`, `matimage`, `matrank`, `matsolve` (to solve a linear system), `charpoly` (characteristic polynomial), to name a few. Bilinear algebra over $\mathbf{R}$ is also there: `qfgaussred` (Gauss reduction), `qfsign` (signature). Type `?*` then print the list of linear algebra functions (`?10`). Can you guess what each of these do?

Let us see how this works. First, a vector space (or module) is given by a generating set of vectors (often a basis) which are represented as *column* vectors. This set of vectors is in turn represented by the columns of a matrix. Quadratic forms are represented by their Gram matrix. The base field (or ring) can be any ring type PARI supports. However, certain operations are specifically written for a real or complex base field, while others are written for $\mathbf{Z}$ as the base ring.

We had some fun with Hilbert matrices and numerical instability a while back, but most of the linear algebra routines are generic. If as before `h = mathilbert(20)`, we may compute

```
matdet(h * Mod(1,101))
matdet(h * (1 + O(101^100)))
```

in $\mathbf{Z}/101\mathbf{Z}$ and the p-adic ring $\mathbf{Z}_{101}$ (to 100 words of accuracy) respectively. Let `H = 1/h` the inverse of `h`:

```
H = 1/h;  \\ integral
L = primes([10^5, 10^5 + 1000]);  \\ pick a few primes
v = vector(#L, i, matdet(H * Mod(1,L[i])));
centerlift( chinese(v) )
```

returns the determinant of `H`. (Assuming it is an integer less than half the product of elements of `L` in absolute value, which it is.) In fact, we computed an homomorphic image of the determinant in a few small finite fields, which admits a single integer representative given the size constraints. We could also have made a single determinant computation modulo a big prime (or pseudoprime) number, e.g `nextprime(2 * B)` if we know that the determinant is less than `B` in absolute value. (Why is that 2 necessary?)

By the way, this is how you insert comments in a script: everything following a double backslash, up to the first newline character, is ignored. If you want comments which span many lines, you can brace them between `/* ... */` pairs. Everything in between will be ignored as well. For instance as a header for the script above you could insert the following:

```
/* Homomorphic imaging scheme to compute the determinant of a classical
 * integral matrix.
 * TODO: Look up the explicit formula
 */
```

(I hope you did not waste your time copying this nonsense, did you?)

In addition, linear algebra over $\mathbf{Z}$, i.e. work on lattices, can also be performed. Let us now consider the lattice Λ generated by the columns of `H` in $\mathbf{Z}^{20} \subset \mathbf{R}^{20}$. Since the determinant is nonzero, we have in fact a basis. What is the structure of the finite abelian group $\mathbf{Z}^{20}/\Lambda$? Type `matsnf(H)`. Wow, 20 cyclic factors.

There is a triangular basis for Λ (triangular when expressed in the canonical basis), perhaps it looks better than our initial one? Type `mathnf(H)`. Hum, what if I also want the unimodular transformation matrix? Simple : `z = mathnf(H, 1); z[1]` is the triangular HNF basis, and `z[2]` is the base change matrix from the canonical basis to the new one, with determinant ± 1. Try `matdet(z[2])`, then `H * z[2] == z[1]`. Fine, it works. And `z[1]` indeed looks better than `H`.

Can we do better? Perhaps, but then we'd better drop the requirement that the basis be triangular, since the latter is essentially canonical. Type

```
M = H * qflll(H)
```

Its columns give an LLL-reduced basis for Λ (`qflll(H)` itself gives the base change matrix). The LLL algorithm outputs a nice basis for a lattice given by an arbitrary basis, where nice means the basis vectors are almost orthogonal and short, with precise guarantees on their relations to the shortest vectors. Not really spectacular on this example, though.

Let us try something else, there should be an integer relation between $\log 3$, $\log 5$ and $\log 15$. How to detect it?

```
u = [log(15), log(5), log(3)];
m = matid(3); m[3,] = round(u * 10^25);
v = qflll(m)[,1] \\ first vector of the LLL-reduced basis
u * v
```

Pretty close. In fact, `lindep` automates this kind of search for integer relations; try `lindep(u)`.

Let us come back to Λ above, and our LLL basis in `M`. Type

```
G = M~*M  \\ Gram matrix
m = qfminim(G, norml2(M[,1]), 100, 2);
```

This enumerates the vectors in Λ which are shorter than the first LLL basis vector, at most 100 of them. The final argument 2 instructs the function to use a safe (slower) algorithm, since the matrix entries are rather large; trying to remove it should produce an error, in this case. There are `m`[1] = 6 such vectors, and `m[3]` gives half of them (`-m[3]` would complete the lot): they are the first 3 basis vectors! So these are optimally short, at least with respect to the Euclidean length. Let us try

```
m = qfminim(G, norml2(M[,4]), 100, 2);
```

(The flag 2 instructs `qfminim` to use a different enumeration strategy, which is much faster when we expect more short vectors than we want to store. Without the flag, this example requires several hours. This is an exponential time algorithm, after all!) This time, we find a slew of short vectors; `matrank(m[3])` says the 100 we have are all included in a 2-dimensional space. Let us try

```
m = qfminim(G, norml2(M[,4]) - 1, 100000, 2);
```

This time we find 50,886 vectors of the requested length, spanning a 4-dimensional space, which is actually generated by `M[,1]`, `M[,2]` `M[,3]` and `M[,5]`.

6. Using Transcendental Functions.

All the elementary transcendental functions and several higher transcendental functions are provided: Γ function, incomplete Γ function, error function, exponential integral, Bessel functions (H^1, H^2, I, J, K, N), confluent hypergeometric functions, Riemann ζ function, polylogarithms, Weber functions, theta functions. More will be written if the need arises.

In this type of functions, the default precision plays an essential role. In almost all cases transcendental functions work in the following way. If the argument is exact, the result is computed using the current default precision. If the argument is not exact, the precision of the argument is used for the computation. A note of warning however: even in this case the *printed* value is the current real format, usually the same as the default precision. In the present chapter we assume that your machine works with 64-bit long integers. If it is not the case, we leave it to you as a good exercise to make the necessary modifications.

Let's assume that we have 38 decimals of default precision (this is what we get automatically at the start of a `gp` session on 64-bit machines). Type `e = exp(1)`. We get the number $e = 2.718\ldots$ to 38 decimals. Let us check how many correct decimals we really have. Change the precision to a substantially higher value, for example by typing `\p 100`. Then type `e`, then `exp(1)` once again. This last value is the correct value of the mathematical constant e to 100 decimals, while

the variable `e` shows the value that was computed to 38 decimals. Clearly they coincide to exactly 38 significant digits.

So 38 digits are printed, but how many significant digits are actually contained in the variable `e`? Type `#e` which indicates we have exactly 2 mantissa words. Since $2\ln(2^{64})/\ln(10) \approx 38.5$ we see that we have 38 or 39 significant digits (on 64-bit machines).

Come back to 38 decimals (`\p 38`). If we type `exp(1.)` you can check that we also obtain 38 decimals. However, type `f = exp(1 + 1E-40)`. Although the default precision is still 38, you can check using the method above that we have in fact 96 significant digits! The reason is that `1 + 1E-40` is computed according to the PARI philosophy, i.e. to the best possible precision. Since `1E-40` has 39 significant digits and 1 has "infinite" precision, the number `1 + 1E-40` will have at least $79 = 39 + 40$ significant digits, hence `f` also.

Now type `cos(1E-19)`. The result is printed as $1.0000\ldots$, but is of course not exactly equal to 1. Using `#%`, we see that the result has 4 mantissa words, giving us the possibility of having 77 correct significant digits. PARI gives you as much as it can, and since 3 mantissa words would have given you only 57 digits, it uses 4. But why does it give so precise a result? Well, it is the same reason as before. When x is close to 1, $\cos(x)$ is close to $1 - x^2/2$, hence the precision is going to be approximately the same as when computing this quantity, here $1 - 0.5 * 10^{-38}$ where $0.5 * 10^{-38}$ is considered with 38 significant digit accuracy. Hence the result will have approximately $38+38 = 76$ significant digits.

This philosophy cannot go too far. For example, when you type `cos(0)`, `gp` should give you exactly 1. Since it is reasonable for a program to assume that a transcendental function never gives you an exact result, `gp` gives you $1.000\ldots$ with as many mantissa word as the current precision.

Let's see some more transcendental functions at work. Type `gamma(10)`. No problem (type `9!` to check). Type `gamma(100)`. The number is now written in exponential format because the default accuracy is too small to give the correct result. To get all digits, the most natural solution is to increase the precision; since `gamma(100)` has 156 decimal digits, type `\p 170` to be on the safe side, then `gamma(100)` once again. Another one is to compute `99!` directly.

Try `gamma(1/2 + 10*I)`. No problem, we have the complex Γ function. Now type

```
t = 1000;
z = gamma(1 + I*t) * t^(-1/2) * exp(Pi/2*t) / sqrt(2*Pi)
norm(z)
```

The latter is very close to 1, in accordance with the complex Stirling formula.

Let's play now with the Riemann zeta function. First turn on the timer (type `#`). Type `zeta(2)`, then `Pi^2/6`. This seems correct. Type `zeta(3)`. All this takes essentially no time at all. However, type `zeta(3.1)`. You will notice that the time is substantially larger; if your machine is too fast to see the difference, increase the precision to `\p1000`. This is because PARI uses special formulas to compute `zeta(n)` when `n` is an integer.

Type `zeta(1 + I)`. This also works. Now for fun, let us compute in a naive way the first complex zero of `zeta`. We know that it is of the form $1/2 + i * t$ with t between 14 and 15. Thus, we can use the following series of instructions. But instead of typing them directly, write them into a file, say `zeta.gp`, then type `\r zeta.gp` under `gp` to read it in:

```
{
  t1 = 1/2 + 14*I;
```

```
  t2 = 1/2 + 15*I; eps = 1E-50;
  z1 = zeta(t1);
  until (norm(z2) < eps,
    z2 = zeta(t2);
    if (norm(z2) < norm(z1),
      t3 = t1; t1 = t2; t2 = t3; z1 = z2
    );
    t2 = (t1+t2) / 2.;
    print(t1 ": " z1)
  )
}
```

Don't forget the braces: they tell `gp` that a sequence of instructions is going to span many lines. We thus obtain the first zero to 25 significant digits.

By the way, you don't need to type in the suffix `.gp` in the `\r` command: it is supplied by `gp` if you forget it. The suffix is not mandatory either, but it is convenient to have all GP scripts labeled in the same distinctive way. Also, some text editors, e.g. Emacs or Vim, will recognize GP scripts as such by their suffix and load special colourful modes.

As mentioned at the beginning of this tutorial, some transcendental functions can also be applied to p-adic numbers. This is as good a time as any to familiarize yourself with them. Type

```
a = exp(7 + O(7^10))
log(a)
```

All seems in order.

```
b = log(5 + O(7^10))
exp(b)
```

Is something wrong? We don't recover the number we started with? This is normal: type

```
exp(b) * teichmuller(5 + O(7^10))
```

and we indeed recover our initial number. The Teichmüller character `teichmuller(x)` on $\mathbf{Z}_p^*$ is the unique $(p-1)$-st root of unity which is congruent to `x` modulo p, assuming that `x` is a p-adic unit.

Let us come back to real numbers for the moment. Type `agm(1,sqrt(2))`. This gives the arithmetic-geometric mean of 1 and $\sqrt{2}$, and is the basic method for computing complete elliptic integrals. In fact, type

```
Pi/2 / intnum(t=0,Pi/2, 1 / sqrt(1 + sin(t)^2)),
```

and the result is the same. The elementary transformation `x = sin(t)` gives the mathematical equality

$$\int_0^1 \frac{dx}{\sqrt{1-x^4}} = \frac{\pi}{2\mathrm{AGM}(1,\sqrt{2})} \ ,$$

which was one of Gauss's remarkable discoveries in his youth.

Now type `2 * agm(1,I) / (1+I)`. As you see, the complex AGM also works, although one must be careful with its definition. The result found is almost identical to the previous one. Do you see why?

Finally, type `agm(1, 1 + 7 + O(7^10))`. So we also have p-adic AGM. Note however that since the square root of a p-adic number is not in general an element of the same p-adic field, only certain p-adic AGMs can be computed. In addition, when $p = 2$, the congruence restriction is that `agm(a,b)` can be computed only when `a/b` is congruent to 1 modulo 16, and not 8 as could be expected.

Now type `?*` then `?11`. This gives you the list of all transcendental functions. Instead of continuing with more examples, we suggest that you experiment yourself with this list. Try integer, real, complex and p-adic arguments. You will notice that some have not been implemented (or do not have a reasonable definition).

7. Using Numerical Tools.

Although not written to be a numerical analysis package, PARI can nonetheless perform some numerical computations. Since linear algebra and polynomial computations are treated somewhere else, this section focuses on solving equations and various methods of summation.

You of course know the formula $\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots)$ which is deduced from the power series expansion of `atan(x)`. You also know that π cannot be computed from this formula, since the convergence is so slow. Right? Wrong! Type

```
\p 100
4 * sumalt(k=0, (-1)^k/(2*k + 1))
```

In a split second, we get π to 100 significant digits (type `Pi` to check).

Similarly, try

```
sumpos(k=1, k^-2)
```

Although once again the convergence is slow, the summation is rather fast; compare with the exact result `Pi^2/6`. This is less impressive because a bit slower than for alternating sums, but still useful.

Even better, `sumalt` can be used to sum divergent series! Type

```
zet(s) = sumalt(k=1, (-1)^(k-1) / k^s) / (1 - 2^(1-s))
```

Then for positive values of `s` different from 1, `zet(s)` is equal to `zeta(s)` and the series converges, albeit slowly; `sumalt` doesn't care however. For negative `s`, the series diverges, but `zet(s)` still gives the correct result! (Namely, the value of a suitable analytic continuation.) Try `zet(-1)`, `zet(-2)`, `zet(-1.5)`, and compare with the corresponding values of `zeta`. You should not push the game too far: `zet(-100)`, for example, gives a completely wrong answer.

Try `zet(I)`, and compare with `zeta(I)`. Even (some) complex values work, although the sum is not alternating any more! Similarly, try

```
sumalt(n=1, (-1)^n / (n+I))
```

More traditional functions are the numerical integration functions. Try `intnum(t=1,2, 1/t)` and presto! you get 100 decimals of log(2). Look at Chapter 3 to see the available integration functions.

With PARI, however, you can go further since complex types are allowed. For example, assume that we want to know the location of the zeros of the function $h(z) = e^z - z$. We use Cauchy's

theorem, which tells us that the number of zeros in a disk of radius r centered around the origin is equal to

$$\frac{1}{2i\pi}\int_{C_r}\frac{h'(z)}{h(z)}\,dz\ ,$$

where C_r is the circle of radius r centered at the origin. The function we want to integrate is

```
fun(z) = my(u = exp(z)); (u-1) / (u-z)
```

(Here `u` is a local variable to the function `f`: whenever a function is called, `gp` fills its argument list with the actual arguments given, and initializes the other declared parameters and local variables to 0. It will then restore their former values upon exit. If we had not declared `u` in the function prototype, it would be considered as a global variable, whose value would be permanently changed. It is not mandatory to declare in this way all parameters, but beware of side effects!)

Type now:

```
zero(r) = r/(2*Pi) * intnum(t=0, 2*Pi, real( fun(r*exp(I*t)) * exp(I*t) ))
```

The function `zero(r)` will count the number of zeros of `fun` whose modulus is less than `r`: we simply made the change of variable $z = r*\exp(i*t)$, and took the real part to avoid integrating the imaginary part. Actually, there is a built-in function `intcirc` to integrate over a circle, yielding the much simpler:

```
zero2(r) = intcirc(z=0, r, fun(z))
```

(This is a little faster than the previous implementation, and no less accurate.)

We may type `\p 9` since we know that the result is a small integer (but the computations should be instantaneous even at `\p` 100 or so), then `zero(1)`, `zero(1.5)`. The result tells us that there are no zeros inside the unit disk, but that there are two (necessarily complex conjugate) in the annulus $1 < |z| < 1.5$. For the sake of completeness, let us compute them. Let $z = x + iy$ be such a zero, with x and y real. Then the equation $e^z - z = 0$ implies, after elementary transformations, that $e^{2x} = x^2+y^2$ and that $e^x\cos(y) = x$. Hence $y = \pm\sqrt{e^{2x} - x^2}$ and hence $e^x\cos(\sqrt{e^{2x} - x^2}) = x$. Therefore, type

```
fun(x) = my(u = exp(x)); u * cos(sqrt(u^2 - x^2)) - x
```

Then `fun(0)` is positive while `fun(1)` is negative. Come back to precision 38 and type

```
x0 = solve(x=0,1, fun(x))
z = x0 + I*sqrt(exp(2*x0) - x0^2)
```

which is the required zero. As a check, type `exp(z) - z`.

Of course you can integrate over contours which are more complicated than circles, but you must perform yourself the variable changes, as we have done above to reduce the integral to a number of integrals on line segments.

The example above also shows the use of the `solve` function. To use `solve` on functions of a complex variable, it is necessary to reduce the problem to a real one. For example, to find the first complex zero of the Riemann zeta function as above, we could try typing

```
solve(t=14,15, real( zeta(1/2 + I*t) )),
```

but this does not work because the real part is positive for `t` = 14 and 15. As it happens, the imaginary part works. Type

```
solve(t=14,15, imag( zeta(1/2 + I*t) )),
```

and this now works. We could also narrow the search interval and type for instance

```
solve(t=14,14.2, real( zeta(1/2 + I*t) ))
```

which would also work.

8. Polynomials.

First a word of warning: it is essential to understand the difference between exact and inexact objects. Try

```
gcd(x - Pi, x^2 - 6*zeta(2))
```

We return a trivial GCD because the notion of GCD for inexact polynomials doesn't make much sense. A better quantitative approach is to use

```
polresultant(x - Pi, x^2 - 6*zeta(2))
```

A result close to zero shows that the GCD is nontrivial for small deformations of the inputs. Without telling us what it is, of course. This being said, we will mostly use polynomials (and power series) with exact coefficients in our examples.

The simplest way to input a polynomial, is to simply write it down, or use an explicit formula for the coefficients and the function `sum`:

```
T = 1 + x^2 + 27*x^10;
T = sum(i = 1, 100, (i+1) * x^i);
```

but it is in much more efficient to create a vector of coefficients then convert it to a polynomial using `Pol` or `Polrev` (`Pol([1,2])` is $x+2$, `Polrev([1,2])` is $2x+1$) :

```
T = Polrev( vector(100, i, i) );

for (i=1, 10^4, Polrev( vector(100, i, i) ) )    \\ time: 60ms
for (i=1, 10^4, sum(i = 1, 100, (i+1) * x^i) ) \\ time: 1,74ms
```

The reason for the discrepancy is that the explicit summation (of densely encoded polynomials) is quadratic in the degree, whereas creating a vector of coefficients then converting it to a polynomial type is linear.

We also have a few built-in classical polynomial families. Consider the 15-th cyclotomic polynomial,

```
pol = polcyclo(15)
```

which is of degree $\varphi(15) = 8$. Now, type

```
r = polroots(pol)
```

We obtain the 8 complex roots of `pol`, given to 38 significant digits. To see them better, type `\B`: they are given as pairs of complex conjugate roots, in some order. In fact, the function `polroots` returns the real roots first, in increasing order, then the other roots by increasing absolute value of their imaginary parts (so that pairs of complex conjugate roots appear together).

The roots of `pol` are by definition the primitive 15-th roots of unity. To check this, simply type `rc = r^15`. Why, we get an error message! Fair enough, vectors cannot be multiplied, even less raised to a power. However, type

```
rc = r^15.0
```

without forgetting the '.' at the end. Now it works, because powering to a nonintegral exponent is a transcendental function and hence is applied termwise. Note that the fact that 15.0 is a real number which is representable exactly as an integer has nothing to do with the problem.

We see that the components of the result are very close to 1. It is however tedious to look at all these real and imaginary parts. It would be impossible if we had many more. Let's do it automatically. Type

```
rr = round(rc)
exponent(rc - rr)
```

We see that `rr` is indeed all 1's, and that the sup-norm of `rc - rr` is around 2^{-125}, reasonable enough when we work with 128 bits of accuracy! In fact, `round` itself already provides a built-in rough approximation of the error:

```
rr = round(rc, &e)
```

Again, `e` contains the number of error bits when rounding `rc` to `rr`; in other words the sup norm of `rc` $-$ `rr` is bounded by $2^{-\texttt{e}}$.

Now type

```
pol = x^5 + x^4 + 2*x^3 - 2*x^2 - 4*x - 3
factor(pol)
factor( poldisc(pol) )
fun(p) = factor(pol, O(p^10));
```

The polynomial `pol` factors over $\mathbf{Q}$ (or $\mathbf{Z}$) as a product of two factors, and the primes dividing its discriminant are 11, 23 and 37. We also created a function `fun(p)` which factors `pol` over $\mathbf{Q}_p$ to p-adic precision 10. Type

```
fun(5)
fun(11)
fun(23)
fun(37)
```

to see different splittings. You can use `lift` to convert p-adic numbers to neighbouring rational numbers.

Similarly, type

```
lf(p) = lift( factor(pol, p) );
lf(2)
lf(11)
lf(23)
lf(37)
```

which show the different factorizations, this time over $\mathbf{F}_p$. In fact, even better: type successively

```
T = ffgen(3^3, 't) \\ we want t to be a free variable
factor(pol, t)
```

The element T is printed as `t` but is actually defined modulo the irreducible polynomial `t^3 + 2t + 2` over $\mathbf{F}_3$ (see `t.mod`): it defines the finite field $\mathbf{F}_{27}$. Note that we introduced a new variable t to express elements in this nonprime field. More generally, the generic `factor` function allows a

second argument which describes the domain over which we want to factor, here $\mathbf{F}_{27}$ (and, above, $\mathbf{Q}_p$ to 10 digits of accuracy and $\mathbf{F}_p$). Typing `factor(pol)` directly would factor it over $\mathbf{Q}$, not what we wanted.

Similarly, type

```
pol2 = x^4 - 4*x^2 + 16
fn = lift( factor(pol2, t^2 + 1) )
```

and we get the factorization of the polynomial `pol2` over the number field $\mathbf{Q}[t]/(t^2+1)$, i.e. over $\mathbf{Q}(i)$. Without the `lift`, the result would involve number field elements as `t_POLMOD`s of the form `Mod(1+t, t^2+1)`, which are more explicit but less readable.

To summarize, in addition to being able to factor integers, you can factor polynomials over $\mathbf{C}$ using `polroots`, over $\mathbf{R}$ using `factor(,1.0)`, over finite fields using `factor(, p)`, over $\mathbf{Q}_p$ using `factor(, O(p^n))`, over $\mathbf{Q}$ using `factor`, and over the number field $\mathbf{Q}[t]/(T)$ using `factor(, T)`. Note however that `factor` itself will guess intelligently over which ring you want to factor: try

```
pol = x^2 + 1;
factor(pol)
factor(pol *1.)
factor(pol * (1 + 0*I))
factor(pol * (1 + 0.*I))
factor(pol * Mod(1,2))
factor(pol * Mod(1, Mod(1,3)*(t^2+1)))
pol2 = x^2 + y^2;
factor(pol2)
factor(pol2 * Mod(1,5))
```

In the present version 2.19.1, it is not possible to factor over other base rings than the ones mentioned above, but multivariate polynomials over those rings are allowed as shown in the last examples. Other functions related to factoring are `padicappr`, `polrootsmod`, `polrootspadic`, `polsturm`. Play with them a little.

Finally, type

```
polsym(pol2, 20)
```

where `pol2` was defined above. This gives the sum of the k-th powers of the roots of `pol2` up to $k = 20$, of course computed using Newton's formula and not using `polroots`. You notice that every odd sum is zero (expected, since the polynomial is even), but also that the signs follow a regular pattern and that the (nonzero) absolute values are powers of 2. This is true: prove it, and more precisely find an explicit formula for the k-th symmetric power not involving (nonrational) algebraic numbers.

9. Power Series.

Now let's play with power series as we have done at the beginning. Type

```
N = 39;
8*x + prod(n=1,N, if(n%4, 1 - x^n, 1), 1 + O(x^(N+1)))^8
```

Apparently, only even powers of `x` appear. This is surprising, but can be proved using the theory of modular forms. Note that we initialize the product to `1 + O(x^(N+1))`, otherwise the whole computation would be done with polynomials; this would first have been slightly slower and also totally useless since the coefficients of `x^(N+1)` and above are irrelevant anyhow if we stop the product at `n` = `N`.

While we are on the subject of modular forms (which, together with Taylor series expansions are another great source of power series), type

```
\ps 122      \\ shortcut for default(seriesprecision, 122)
d = x * eta(x)^24
```

This gives the first 122 terms of the Fourier series expansion of the modular discriminant function Δ of Ramanujan. Its coefficients give by definition the Ramanujan τ function, which has a number of marvelous properties (look at any book on modular forms for explanations). We would like to see its properties modulo 2. Type `d%2`. Hmm, apparently PARI doesn't like to reduce coefficients of power series, or polynomials for that matter, directly. Can we do it without writing a little program? No problem. Type instead

```
lift(Mod(1,2) * d)
centerlift(Mod(1,3) * d)
```

and now this works like a charm. The pattern in the first result is clear; the pattern is less clear in the second result, but nonetheless there is one. Of course, it now remains to prove it (see Antwerp III or your resident modular forms guru).

10. Working with Elliptic Curves.

Now we are getting to more complicated objects. Just as with number fields which we will meet later on, the first thing to do is to initialize them. That's because a lot of data will be needed repeatedly, and it's much more convenient to have it ready once and for all. Here, this is done with the function `ellinit`.

So type

```
e0 = ellinit([6,-3,9,-16,-14])
```

This computes a number of quantities related to the elliptic curve defined by the affine equation

$$y^2 + 6xy + 9y = x^3 - 3x^2 - 16x - 14 \ .$$

It is not that clear what all these funny numbers mean, except that we recognize the first few of them as the coefficients we just input. To retrieve meaningful information from such complicated objects (and number fields will be worse), one uses so-called *member functions*. Type `?.` to get a complete list. Whenever `ell` appears in the right hand side, we can apply the corresponding function to an object output by `ellinit`. (I'm sure you know how the other `init` functions will be called now, don't you? Oh, by the way, neither `clgpinit` nor `pridinit` exist.)

Let's try it. The discriminant `e0.disc` is equal to 37, hence the conductor of the curve is 37. Of course in general it is not so trivial. In fact, although the equation of the curve is clearly minimal (since the discriminant is 12th-power-free), it is not in standard reduced form, so type

```
e = ellminimalmodel(e0)
```

which gives the `ell` structure attached to the standard model, exactly as if we had used `ellinit` on a reduced equation. For some related data, type

```
gr = ellglobalred(e0)
```

The first component `gr[1]` tells us that the conductor is 37 as we already knew. The second component is a 4-component vector which allows us to get the minimal equation: in fact `e` is `ellchangecurve(e0, gr[2])`. Type

```
q0 = [-2,2]
ellisoncurve(e0, q0)
q = ellchangepoint(e0, q0, gr[2])
ellisoncurve(e, q)
```

The point `q0` is on the curve, as checked by `ellisoncurve`, and we transferred it onto the minimal model `e`, using `ellchangepoint` and the change of variable computed above. The point at infinity is represented as `[0]` on all elliptic curves; this is the identity for the group law.

Here, `q=[0,0]` obviously lies on `e`, which has equation $y^2+y=x^3-x$. Let us now play a little with points on `e`. The group law on an elliptic curve is implemented with the functions `elladd` for addition, `ellsub` for subtraction and `ellmul` for multiplication by an integer. For example, the negative of `q` is `ellsub(e,[0],q)`, and the double is obtained either as `ellmul(e,q,2)` or as `elladd(e,q,q)`.

Now `q` may be a torsion point. Type `ellheight(e, q)`, which computes the canonical Neron-Tate height of `q`. Note that `ellheight` does not assume that `e` is *minimal*! (Although it is, making things a little faster.) This is nonzero, hence `q` is not torsion. To see this even better, type

```
for(k = 1, 20, print(ellmul(e, q, k)))
```

and we see the characteristic parabolic explosion of the size of the points. (And another proof that `q` is not torsion, assuming Mazur's bound on the size of the rational torsion.) We could can also type `ellorder(e, q)` which returns 0, telling us yet again that `q` is nontorsion. As a consistency check, type

```
ellheight(e, ellmul(e, q,20)) / ellheight(e, q)
```

We indeed find $400 = 20^2$ as it should be.

Notice how (almost) all those `ell`–prefixed functions take our elliptic curve as a first argument? This will be true with number fields as well: whatever object was initialized by an *ob*–`init` function will have to be used as a first argument of all the *ob*–prefixed functions. Conversely, you won't be able to use any such high-level function before you correctly initialize the relevant object.

Ok, let's try another curve. Type

```
E = ellinit([0,-1,1,0,0])
q = [0,0]; ellheight(E, q)
```

This corresponds to the equation $y^2+y=x^3-x^2$ and an obvious rational point on it. Again from the discriminant we see that the conductor is equal to 11, and if you type `ellminimalmodel(E)`

you will see that the equation for `E` is minimal. This time the height is exactly zero, hence `q` must be a torsion point. Indeed, typing

```
for(k=1, 5, print(ellmul(E, q,k)))
ellorder(E, q)   \\ simpler
```

we see in two different ways that `q` is a point of order 5. Moreover, typing

```
elltors(E)
```

shows that `q` generates all the torsion of `E`, which is cyclic of order 5.

Let's try still another curve, $y^2 + y = x^3 - 7x + 6$:

```
e = ellinit([0,0,1,-7,6])
ellglobalred(e)
```

As before, this is a minimal equation; now the conductor is 5077. There are some trivial integral points on this curve, but let's try to be more systematic. Typing

```
elltors(e)
```

shows that the torsion subgroup is trivial, so we don't have to worry about torsion points. Next, the function `ellratpoints` allows us to find rational points of small height

```
v = ellratpoints(e,1000)
```

The vector v contains all 130 rational points (x, y) on the curve whose x-coordinate is n/d with $|n|$ and $|d|$ both less than 1000. Note that `ellratpoints(e,10^6)` takes less than 1 second, and produces 344 points. Of course, these are grouped by pairs: if (x, y) is on the curve, its opposite is $(x, -y - 1)$ as

```
ellneg(e, ['x,'y])
```

shows. Note that there is no problem with manipulating points with formal coordinates. This is large for a curve having such a small conductor. So we suspect (if we do not know already, since this curve is quite famous!) that the rank of this curve must be large. Let's try and put some order into this. First, we eliminate one element in each pair of opposite points:

```
v = vecsort(v, 1, 8)
```

The argument 1 specifies a comparison function: we sort the points by first coordinate only, in particular two points with the same x-coordinate compare as equal; the 8 flag eliminates "duplicates". The same effect could be obtained in a more verbose way using an inline anonymous function

```
v = vecsort(v, (P,Q) -> sign(P[1]-Q[1]), 8)
```

We now order the points according to their canonical height:

```
hv = [ ellheight(e,P) | P <- v ];
v = vecextract(v, vecsort(hv,,1)) \\ indirect sort wrt h, then permute
```

It seems reasonable to take the numbers with smallest height as possible generators of the Mordell-Weil group. Let's try the first four: type

```
m = ellheightmatrix(e, v[1..4]); matdet(m)
```

Since the curve has no torsion, the determinant being close to zero implies that the first four points are dependent. To find the dependency, it is enough to find the kernel of the matrix `m`. So type

`matker(m)`: we indeed get a nontrivial kernel, and the coefficients are close to integers. Typing `elladd(e, v[1],v[3])` does indeed show that it is equal to `v[4]`.

Taking any other four points, we seem to always find a dependency. Let's find all dependencies. Type

```
vp = v[1..3];
m = ellheightmatrix(e,vp);
matdet(m)
```

This is now clearly nonzero so the first 3 points are linearly independent, showing that the rank of the curve is at least equal to 3. (In fact, `e` is the curve of smallest conductor having rank 3.) We would like to see whether the other points are dependent: if `Q` is some point which is dependent on `v[1],v[2]` and `v[3]` and

```
c = [ellheight(e, P, Q) | P <- vp]~
```

then `m^(-1) * c` will give the coefficients of the dependence relation. If these coefficients are not close to integers, then there is no dependency, otherwise we can round and check rigourously using a function such as the following:

```
ellcomb(e, P, L) =
{ my (Q = [0]);
  for (i = 1, #P, Q = elladd(e, Q, ellmul(e, P[i], L[i])));
  return (Q);
}
```

This is safer than using the `matker` function. Thus, type

```
mi = m^(-1);
w = vector(#v, k, mi * [ellheight(e, P, v[k]) | P <- vp]~)
wr = round(w, &e)
```

We "see" that the coefficients are all very close to integers, and we quantified it with the last instruction: `wr` is the vector expressing all the components of `v` on its first 3, and 2^{-e} gives an upper bound on the maximum distance to an integer. The rigorous check is positive:

```
for (i=1, #w, if (v[i] != ellcomb(e,vp,wr[i]), error(i)));
```

No error! We must make two technical remarks at this point:

(1) Using the height pairing to find dependence relations as we have done only finds relations modulo torsion; but in this example, the curve has trivial torsion, as we checked earlier.

(2) In the above calculation we were lucky that all the `v[j]` were $\mathbf{Z}$-linear combinations of `v[1]`, `v[2]` and `v[3]` and not just $\mathbf{Q}$-linear combinations; in general the results in w might have given a vector of rationals: if $k \geq 2$ is minimal such that kQ is in the subgroup generated by `v[1]`, `v[2]` and `v[3]`, then the entries of `matsolve(m, ellbil(e, vp,Q))` will be rationals with common denominator k. This can be detected by using `bestappr` instead of `round` in the above.

We are thus led to strongly believe that the curve has rank exactly 3, with generators `v[1]`, `v[2]` and `v[3]`, and this can be proved to be the case. Indeed, all this (and much more) is automated in the built-in `ellrank` function:

```
[r, R, s, P] = ellrank(e);
Q = ellsaturation(e, P, 100)
```

We find $r = R = 3$ and a vector P containing 3 rational points. This performs 2-descent on the curve: the rank belongs to $[r, R]$ hence is equal to 3. The points in P are independent modulo torsion : since $\#\mathtt{P} = r = R$, they generate a subgroup of finite index. The final `ellsaturation` provides a new vector of points which is p-saturated for all primes $p \leq 100$: the index of the subgroup they generate is not divisible by any such prime.

In other situations, we could be missing some points and have $\#\mathtt{P} < r$: for instance

```
e = ellinit([-127^2, 0]);
ellrank(e)
```

does not find any point although the rank is 1; in this particular case (rank 1 and "small" conductor), `ellheegner` finds the missing point, which happens to be a generator.

Let us explore a few more elliptic curve related functions. Keep our curve `e` of rank 3, and type

```
v1 = [1,0]; z1 = ellpointtoz(e, v1)
v2 = [2,0]; z2 = ellpointtoz(e, v2)
```

We thus get the complex parametrization of the curve. To add the points `v1` and `v2`, we should of course type `elladd(e, v1,v2)`, but we can also type `ellztopoint(e, z1 + z2)` which has the disadvantage of giving complex numbers, but illustrates how the group law on `e` is obtained from the addition law on $\mathbf{C}$.

Type

```
f = x * Ser(ellan(e, 30))
```

This gives a power series which is the Fourier expansion of a modular form of weight 2 for $\Gamma_0(5077)$. (This has been proved directly, before Wiles's general result.) In fact, to find the modular parametrization of the curve, type

```
modul = elltaniyama(e)
u = modul[1];
v = modul[2];
```

We can check in various ways that this indeed parametrizes the curve:

```
(v^2 + v) - (u^3 - 7*u + 6)
```

is close to 0 for instance, or simply that `ellisoncurve(e,modul)` returns 1. Now type

```
x * u' / (2*v + 1)
```

and we see that this is equal to the modular form `f` found above; the quote `'` tells `gp` to take the derivative of the expression with respect to its main variable. The functions `u` and `v`, considered on the upper half plane with $x = e^{2i\pi\tau}$, are in fact modular *functions* for $\Gamma_0(5077)$.

The function `ellan(e, 30)` gives the first 30 coefficients a_n of the L-series of `e`. One can also ask for a single coefficient: the millionth is `ellak(e, 10^6)`. Note however that calling `ellan(e,100000)` is much faster than the equivalent `vector(100000,k,ellak(e,k))`. For a prime `p`, `ellap(e,p)` is equivalent to `ellak(e,p)`; this is the integer a_p such that the number of points on `e` over $\mathbf{F}_p$ is $1 + p - a_p$. (With the standard PARI distribution, `ellap` is the only way to obtain the order of an elliptic curve over $\mathbf{F}_p$ in `gp`. The external package `ellsea` provides much more efficient routines.)

Finally, let us come back to the curve `E` defined above by `E = ellinit([0,-1,1,0,0])`, which had an obvious rational 5-torsion point. The sign of the functional equation, given by `ellrootno(E)`, is +1. Assuming the rank parity conjecture, it follows that the Mordell-Weil group of E has even rank. The value of the L-function of `E` at 1

```
ls = lfun(E, 1)
```

is definitely nonzero, so `E` has rank 0. According to the Birch and Swinnerton-Dyer conjecture, which is proved for this curve, `ls` is given by the following formula (in this case):

$$L(E,1) = \frac{\Omega \cdot c \cdot |\text{Ш}|}{|E_{\text{tors}}|^2} \ ,$$

where Ω is the real period of E, c is the global Tamagawa number, $|\text{Ш}|$ is the order of the Tate-Shafarevich group, and E_{tors} is the torsion group of E.

Now we know many of these quantities: Ω is equal to `E.omega[1]` The global Tamagawa number c is given by `elltamagawa(E)` and is 1 We already know that the torsion subgroup of E contains a point of order 5, and typing `elltors(E)` shows that it is of order exactly 5. So type

```
ls * 25/E.omega[1]
```

This shows that Ш must be the trivial group. A short hand for $\frac{\Omega \cdot c}{|E_{\text{tors}}|^2}$ is `ellbsd(E)`, so the previous line can be written as

```
ls / ellbsd(E)
```

For more detailed information on the local reduction of an elliptic curve at a specific prime `p`, use the function `elllocalred(E,p)`; the second component gives the Kodaira symbol in an encoded form. See the manual or online help for details.

11. Working in Quadratic Number Fields.

The simplest of all number fields outside $\mathbf{Q}$ are quadratic fields and are the subject of the present section. We shall deal in the next one with general number fields (including $\mathbf{Q}$ and quadratic fields!), and one should be aware that all we will see now has a more powerful, in general easier to use, equivalent in the general context. But possibly much slower.

Such fields are characterized by their discriminant. Even better, any nonsquare integer D congruent to 0 or 1 modulo 4 is the discriminant of a specific order in a quadratic field. We can check whether this order is maximal with `isfundamental(D)`. Elements of a quadratic field are of the form $a + b\omega$, where ω is chosen as $\sqrt{D}/2$ if D is even and $(1 + \sqrt{D})/2$ if D is odd, and are represented in PARI by quadratic numbers. To initialize working in a quadratic order, one starts by the command `w = quadgen(`D`,'w)`.

This sets `w` equal to ω as above, and is printed `w`. If you need several quadratic orders at the same time, you can use different variable names:

```
w1 = quadgen(-23,'w1)
w2 = quadgen(-15,'w1)
```

In addition to elements of a quadratic order, we also want to be able to handle ideals of such orders. In the quadratic case, it is equivalent to handling binary quadratic forms. Quadratic forms

are triples (a, b, c) representing the form $ax^2 + bxy + cy^2$. Such a form will be printed as, and can be created by, `Qfb(`a`,`b`,`c`)`.

Such forms can be multiplied, divided, powered as many PARI objects using the usual operations, and they can also be reduced using the function `qfbred` or `qfbredsl2` (which also returns the unimodular transformation matrix). One can also use explicitly `qfbcomp` / `qfbpow` (same as multiplication) and `qfbcompraw` / `qfbpowraw` (composition without reduction) (It is not the purpose of this tutorial to explain what all these things mean.) In addition, Shanks's NUCOMP algorithm has been implemented (functions `qfbnucomp` and `qfbnupow`), and this is usually a little faster.

Finally, you have at your disposal the functions `qfbclassno` which (*usually*) gives the class number, the function `qfbhclassno` which gives the Hurwitz class number, and the much more sophisticated `quadclassunit` function which gives the class number and class group structure.

Let us see examples of all this at work.

Type `qfbclassno(-10007)`. `gp` tells us that the result is 77. However, you may have noticed in the explanation above that the result is only *usually* correct. This is because the implementers of the algorithm have been lazy and have not put the complete Shanks algorithm into PARI, causing it to fail in certain very rare cases. In practice, it is almost always correct, and the much more powerful `quadclassunit` program, which *is* complete (at least for fundamental discriminants) can give confirmation; but now, under the Generalized Riemann Hypothesis!

So we may be a little suspicious of this class number. Let us check it. First, we need to find a quadratic form of discriminant -10007. Since this discriminant is congruent to 1 modulo 8, we know that there is an ideal of norm equal to 2, i.e. a binary quadratic form (a, b, c) with $a = 2$. To compute it we type `f = qfbprimeform(-10007, 2)`. OK, now we have a form. If the class number is correct, the very least is that this form raised to the power 77 should equal the identity. Type `f^77`. We get a form starting with 1, i.e. the identity. Raising `f` to the powers 11 and 7 does not give the identity, thus we now know that the order of `f` is exactly 77, hence the class number is a multiple of 77. But how can we be sure that it is exactly 77 and not a proper multiple? Well, type

```
sqrt(10007)/Pi * prodeuler(p=2,500, 1./(1 - kronecker(-10007,p)/p))
```

This is nothing else than an approximation to the Dirichlet class number formula. The function `kronecker` is the Kronecker symbol, in this case simply the Legendre symbol. Note also that we have written `1./(1 - ...)` with a dot after the first 1. Otherwise, PARI may want to compute the whole thing as a rational number, which would be terribly long and useless. In fact PARI does no such thing in this particular case (`prodeuler` is always computed as a real number), but you never know. Better safe than sorry!

We find 77.77, pretty close to 77, so things seem in order. Explicit bounds on the prime limit to be used in the Euler product can be given which make the above reasoning rigorous.

Let us try the same thing with $D = -3299$. `qfbclassno` and the Euler product convince us that the class number must be 27. However, we get stuck when we try to prove this in the simple-minded way above. Indeed, we type `f = qfbprimeform(-3299, 3)` (2 is not the norm of a prime ideal but 3 is), and we see that `f` raised to the power 9 is equal to the identity. This is the case for any other quadratic form we choose. So we suspect that the class group is not cyclic. Indeed, if we list all 9 distinct powers of `f`, we see that `qfbprimeform(-3299, 5)` is not on the list, although its cube is as it must. This implies that the class group is probably equal to a product of a cyclic group of order 9 by a cyclic group of order 3. The Euler product plus explicit bounds prove this.

Another way to check it is to use the `quadclassunit` function by typing for example

```
quadclassunit(-3299)
```

Now, if we believe in GRH, the class group is as we thought (see Chapter 3 for a complete description of this function). Note that using the even more general function `bnfinit` (which handles general number fields and gives more complicated results), we could *certify* this result, i.e remove the GRH assumption. Let's do it, type

```
bnf = bnfinit(x^2 + 3299); bnfcertify(bnf)
```

A nonzero result (here 1) means that everything is ok. Good, but what did we certify after all? Let's have a look at this `bnf` (just type it!). Enlightening, isn't it? Recall that the `init` functions (we've already seen `ellinit`) store all kind of technical information which you certainly don't care about, but which will be put to good use by higher level functions. That's why `bnfcertify` could not be used on the output of `quadclassunit`: it needs much more data.

To extract sensible information from such complicated objects, you must use one of the many *member functions* (remember: `?.` to get a complete list). In this case `bnf.clgp` which extracts the class group structure. This is much better. Type `%.no` to check that this leading 27 is indeed what we think it is and not some stupid technical parameter. Note that `bnf.clgp.no` would work just as well, or even `bnf.no`!

As a last check, we can request a relative equation for the Hilbert class field of $\mathbf{Q}(\sqrt{-3299})$: type `quadhilbert(-3299)`. It is indeed of degree 27 so everything fits together.

Working in real quadratic fields instead of complex ones, i.e. with $D > 0$, is not very different.

The same `quadgen` function is used to create elements. Ideals are again represented by binary quadratic forms (a, b, c), this time indefinite. However, the Archimedean valuations of the number field start to come into play, hence in fact quadratic forms with positive discriminant can also be represented as a pair $[q, d]$ where $q =$ `Qfb(a,b,c)` is the indefinite quadratic form $ax^2 + bxy + cy^2$, and $d \in \mathbf{R}$ is a "distance" component, as defined by Shanks and Lenstra.

The form q can be multiplied, divided, powered, and they can be reduced using `qfbred`. To compose extended forms $[q, d]$ and $[q, d']$, one may use `qfbcomp` and `qfbpow` but ordinary multiplication and powerings also accept pairs.

The `qfbred` function applies a succession of elementary reduction steps corresponding essentially to a continued fraction expansion, and a single one of these steps can be achieved by adding an (optional) flag to the arguments of `qfbred`. Again, the function `qfbprimeform` gives a prime form, but the form which is given corresponds to an ideal of prime norm which is usually not reduced. If desired, it can be reduced using `qfbred`.

Finally, you still have at your disposal the function `qfbclassno` which gives the class number (this time *guaranteed* correct), `quadregulator` which gives the regulator, and the much more sophisticated `quadclassunit` giving the class group's structure and its generators, as well as the regulator. The `qfbclassno` and `quadregulator` functions use an algorithm which is $O(\sqrt{D})$, hence become very slow for discriminants of more than 10 digits. `quadclassunit` can be used on a much larger range.

Let us see examples of all this at work and learn some little known number theory at the same time. First of all, type

```
D = 3 * 3299; qfbclassno(D)
```

We see that the class number is 3. (We know in advance that it must be divisible by 3 from the `D = -3299` case above and Scholz's mirror theorem.) Let us create a form by typing

```
f = qfbred(qfbprimeform(D,2))
```

This gives us a prime ideal of norm equal to 2. Is this ideal principal? Well, one way to check this, which is not the most efficient but will suffice for now, is to look at the complete cycle of reduced forms equivalent to `f`. Type

```
g = f; until(g == f, g = qfbred(g, 1); print(g))
```

(the flag 1 in `qfbred` means to perform a single reduction step). We see that we come back to the form `f` without meeting the principal form (starting with ± 1) in the cycle, so the ideal corresponding to `f` is not principal.

Since the class number is equal to 3, we know however that `f^3` will be a principal ideal $\alpha \mathbf{Z}_K$. How do we find α? For this, type

```
f3 = qfbpowraw(f, 3)
```

This computes the cube of `f`, without reducing it. Hence it corresponds to an ideal of norm equal to $8 = 2^3$, so we already know that the norm of α is equal to ± 8. We need more information, and this is given by supplementing the form by a *distance* component d, a real number that tracks archimedean information that will eventually allows to recover α: we call such a pair $[f, d]$ an *extended form*.

```
F3 = [f3, 0.0]; /* initialize */
while(abs(component(F3[1], 1)) != 1, F3 = qfbred(F3, 1); print(F3))
d = F3[2];
```

We reduce the form `F3` until we reach the unit form (exactly 6 times), then extract the Archimedean component, say $d = 8.59\ldots$ By definition of this distance, we know that

$$\frac{\alpha}{\sigma(\alpha)} = \pm e^{2d},$$

where σ denotes real conjugation in our quadratic field. Thus, if we type

```
Na = 8; /* |Norm(alpha)| */
a = sqrt(Na * exp(2*d))
sa = Na / a
```

we know that up to sign, `a` and `sa` are numerical approximations of α and $\sigma(\alpha)$. Of course, α can always be chosen to be positive, and a quick numerical check shows that the difference of `a` and `sa` is close to an integer, and not the sum, so that in fact the norm of α is equal to -8 and the numerical approximation to $\sigma(\alpha)$ is $-$`sa`. Thus we type

```
p = x^2 - round(a-sa)*x - 8
```

and this $(x^2 - 15221x - 8)$ is the characteristic polynomial of α. (A more user-friendly, but much less efficient, way to obtain the same result is `algdep(a,2)` or `algdep(sa,2)`.) We can check that the discriminant of this polynomial is a square multiple of `D` using `issquare(poldisc(p) / D)`, so α is indeed in our field. More precisely, solving for α and using the numerical approximation that we have to resolve the sign ambiguity in the square root, we get explicitly $\alpha = (15221 + 153\sqrt{D})/2$. Note that this can also be done automatically using the functions `polred` and `modreverse`, as we will see later in the general number field case. Or by solving a system of 2 linear equations

in 2 variables. (Exercise: now that we have α, check that it is indeed a generator of the ideal corresponding to the form `f3`.)

Let us now play a little with cycles. Type

```
D = 10^7 + 1
quadclassunit(D)
```

We get as a result a 5-component vector, which tells us that (under GRH) the class number is equal to 1, and the regulator R is approximately equal to 2641.5. You may certify this with

```
bnf = bnfinit(x^2 - D, 1);  \\  insist on finding fundamental unit
bnfcertify(bnf);
```

which is fast for small D but costly in general. Indeed `bnfcertify` needs the fundamental unit, which is large: we expect coefficients of about $R/\log(10) \approx 1147$ digits, see `bnf.fu`! (We inserted the flag 1 so that `bnfinit` would at least compute this fundamental unit in factored form, which allows to reconstruct `bnf.fu`; `bnfcertify` will often fail without it.) On the other hand, you can also try `quadunit(D,'w)`. Impressive, isn't it? (You can check that its logarithm is indeed equal to the regulator.)

Let's now assume that we want the regulator R to 500 decimals, say. (Without cheating and computing the fundamental unit exactly first!) This can be computed with no effort, simply from the crude approximation 2641.5 above. This time, we start with the unit form. Type:

```
u = [qfbprimeform(D, 1), 0.0];
R = 2641.5; /* rough approximation */
```

This initializes the distance to 0. Now type `f = qfbred(u, 1)`. This is the first form encountered along the principal cycle. For the moment, keep the precision low, for example the initial default precision. The distance from the identity to `f` is around 4.253, given by the second component. The distance function is approximately additive. Since we want a total distance of R, this should be encountered approximately at $R/f[2] \approx 621$ times the distance of `f`. Hence, as a first try, we type `f ^621`. Oops, we overshoot: the distance is now $3173.018\ldots > R$. Now we can refine our initial estimate: we should be close to the correct distance if we raise `f` to the power $621 * R/3173$ which is a little less than 517. Now if we compute `f^517` we hit the principal form right on the dot. This is not a lucky accident: we must land extremely close to the correct target using this method, and usually at most one reduction correction step is necessary. And the distance component can tell us where we are along the cycle.

Up to now, we have only worked to low precision. The goal was to obtain this unknown integer 517. Note that this number has no mathematical significance: indeed the notion of reduction of a form with positive discriminant is not well defined since there are usually many reduced forms equivalent to a given form. (In fact, `f^516` also yields the principal form, so you can replace 517 by 516 if you wish!)

However, when PARI computes with extended forms, the specific order and reductions that it performs are dictated entirely by the coefficients of the quadratic form itself, and not by the distance component, hence the precision used has no effect.

We now start again by setting the precision to (for example) 500 decimal digits, we set `u = precision(u, 500)` (why is this necessary?), and then `qfbred(u, 1)^517` (or to the power 516). We already know that we land on the unit form, and the distance component gives us the regulator to 500 decimal places with no effort at all.

In a similar way, we could obtain the so-called *compact representation* of the fundamental unit itself, or p-adic regulators. I leave this as exercises for the interested reader.

You can try the `quadhilbert` function on that field but, since the class number is 1, the result won't be that exciting. If you try it on our preceding example ($3 * 3299$) it should take about 2 seconds.

Time for a coffee break?

12. Working in General Number Fields.

12.1. Defining equations and elements in the field.

The situation here is of course more difficult. First of all, remembering what we did with elliptic curves, we need to initialize data linked to our base number field, with something more serious than `quadgen`. For example assume that we want to work in the number field K defined by one of the roots of the equation $x^4 + 24x^2 + 585x + 1791 = 0$. This is done by typing

```
T = x^4 + 24*x^2 + 585*x + 1791
nf = nfinit(T)
```

We get an `nf` structure but, thanks to member functions, we do not need to know anything about it. If you type `nf.pol`, you will get the polynomial `T` which you just input. `nf.sign` yields the signature (r_1, r_2) of the field, `nf.disc` the field discriminant, `nf.zk` an integral basis, etc....

The integral basis is expressed in terms of a generic root `x` of `T` and we notice it's very far from being a power integral basis, which is a little strange for such a small field. Hum, let's check that: `poldisc(T)`? Ooops, small wonder we had such denominators, the index of the power order $\mathbf{Z}[x]/(T)$ in the maximal order $\mathbf{Z}_K$ is, well, `nf.index`. Note that this is also given by

```
sqrtint(poldisc(nf.pol) / nf.disc)
```

Anyway, that's 3087, we don't want to work with such a skewed polynomial! So, we type

```
A = polredbest(T)
```

The `polredbest` function usually gives a simpler polynomial, where *red* stands for "reduction" and *best* for "best effort", meaning that the function strives to find a good defining polynomial while guaranteeing that the result is found quickly (in polynomial time). The polynomial $x^4 - x^3 - 21x^2 + 17x + 133$ defines the same field as T, with somewhat smaller coefficients. Type `poldisc(A)`, this is better, but maybe not optimal yet (the index is still 7).

In fact, had you typed `nfinit(T, 3)`, `nfinit` would first have tried to find a good polynomial defining the same field (i.e. one with small index) before proceeding. It's not too late, let's redefine our number field:

```
NF = nfinit(nf, 3)
```

The output is a two-component vector. The first component is the new `nf`: type

```
nf = NF[1];
```

If you type `nf.pol`, you notice that `gp` indeed replaced your bad polynomial T by the better one A. (Small wonder, `nfinit` internally called `polredbest`.) The second component enables you to switch conveniently to our new polynomial.

Namely, call θ a root of our initial polynomial T, and α a root of A. These are algebraic numbers, and as already mentioned are represented as polmods. For example, in our special case θ and α are equal to the polmods

```
THETA = Mod(x, x^4 + 24*x^2 + 585*x + 1791)
ALPHA = Mod(x, x^4 - x^3 - 21*x^2 + 17*x + 133)
```

respectively. Here we are considering only the algebraic aspect, and hence ignore completely *which* root θ or α is chosen.

Now you may have a number of elements of your number field which are expressed as polmods with respect to your old polynomial, i.e. as polynomials in θ. Since we are now going to work with α instead, it is necessary to convert these numbers to a representation using α. This is what the second component of `NF` is for: type `C = NF[2]`, you get

```
Mod(-10/7*x^3 + 43/7*x^2 + 73/7*x - 64, x^4 - x^3 - 21*x^2 + 17*x + 133)
```

meaning that $\theta = -\frac{10}{7}\alpha^3 + \frac{43}{7}\alpha^2 + \frac{73}{7}\alpha - 64$, and hence the conversion from a polynomial in θ to one in α is easy, using `subst`. (We could get this polynomial from `polred` as well, try `polred(T, 2)`.) If we want the reverse, i.e. to go back from a representation in α to a representation in θ, we use the function `modreverse` on this polmod `C`. Try it. The result has a big denominator (1029) essentially because our initial polynomial `T` was so bad. By the way, to get that 1029, you should type `denominator(content(C))`. Trying `denominator` by itself would not work since the denominator of a polynomial is defined to be 1 (and its numerator is itself). The reason for this is that we think of a polynomial as a special case of a rational function.

From now on, we forget about `T`, and use only the polynomial `A` defining α, and the components of the vector `nf` which gives information on our number field K. Type

```
u = Mod(x^3 - 5*x^2 - 8*x + 56, A) / 7
```

This is an element in K. There are three equivalent representations for number field elements: polmod, polynomial, and column vector giving a decomposition in the integral basis `nf.zk` (*not* on the power basis $(1, x, x^2, \ldots)$). All three are equally valid when the number field is understood (is given as first argument to the function). You will be able to use any one of them as long as the function you call requires an `nf` argument as well. However, most PARI functions will return elements as column vectors. It's an important feature of number theoretic functions that, although they may have a preferred format for input, they will accept a wealth of other different formats. We already saw this for `nfinit` which accepts either a polynomial or an `nf`. It will be true for ideals, congruence subgroups, etc.

Let's stick with elements for the time being. How does one go from one representation to the other? Between polynomials and polmods, it's easy: `lift` and `Mod` will do the job. Next, from polmods/polynomials to column vectors: type `v = nfalgtobasis(nf, u)`. So `u` $= \alpha^3 - \alpha^2 - \alpha + 8$, right? Wrong! The coordinates of `u` are given with respect to the *integral basis*, not the power basis $(1, \alpha, \alpha^2, \alpha^3)$ (and they don't coincide, type `nf.zk` if you forgot what the integral basis looked like). As a polynomial in α, we simply have `u` $= \frac{1}{7}(\alpha^3 - 5\alpha^2 - 8\alpha + 56)$, which is trivially deduced from the original polmod representation!

Of course `v = nfalgtobasis(nf, lift(u))` would work equally well. Indeed we don't need the polmod information since `nf` already provides the defining polynomial. To go back to polmod representation, use `nfbasistoalg(nf, v)`. Notice that `u` is an algebraic integer since `v` has integer coordinates (try `denominator(v) == 1`, which is of course overkill here, but not so in a program).

Let's try this out. We may for instance compute `u^3`. Try it. Or, we can type `1/u`. Better yet, if we want to know the norm from K to $\mathbf{Q}$ of `u`, we type `norm(u)` (what else?); `trace(u)` works as well. Notice that none of this would work on polynomials or column vectors since you don't have the opportunity to supply `nf`! But we could use `nfeltpow(nf,u,3)`, `nfeltdiv(nf,1,u)` (or `nfeltpow(nf,u,-1)`) which would work whatever representation was chosen. Of course, there is also an `nfeltnorm` function (and `nfelttrace` as well). You can also consider (u) as a principal ideal, and just type

```
idealnorm(nf,u)
```

Of course, in this way, we lose the *sign* information. We will talk about ideals later on.

If we want all the symmetric functions of `u` and not only the norm, we type `charpoly(u)`. Note that this gives the characteristic polynomial of `u`, and not in general the minimal polynomial. We have `minpoly(u)` for this.

Exercise. Find a simpler expression for `u`.

Now let's work on the field itself. The `nfinit` command already gave us some information. The field is totally complex (its signature `nf.sign` is $[0, 2]$), its discriminant `nf.disc` is 18981 and `nf.zk` is an integral basis. The Galois group of its Galois closure can be obtained by typing `polgalois(A)`. The answer `[8,-1,1,"D(4)"]` shows that it is equal to D_4, the dihedral group with 8 elements, i.e. the group of symmetries of a square.

This implies that the field is "partially Galois", i.e. that there exists at least one nontrivial field isomorphism which fixes K, exactly one in this case. Type `nfgaloisconj(nf)`. The result tells us that, apart from the trivial automorphism, the map

$$\alpha \mapsto \frac{1}{7}(-\alpha^3 + 5\alpha^2 + \alpha - 49)$$

is the only field automorphism.

```
nfgaloisconj(nf);
s = Mod(%[2], A)
charpoly(s)
```

and we obtain `A` once again.

Let us check that `s` is of order 2: `subst(lift(s), x, s)`. It is. We may express it as a matrix:

```
w = Vec( matid(4) ) \\ canonical basis
v = vector(#w, i, nfgaloisapply(nf, s, w[i]))
M = Mat(v)
```

The vector `v` contains the images of the integral basis elements (as column vectors). The last statement concatenates them into a square matrix. So, `M` gives the action of `s` on the integral basis. Let's check `M^2`. That's the identity all right.

The fixed field of this automorphism is going to be the only nontrivial subfield of K. I seem to recall that `polred` told us this was the third cyclotomic field. Let's check this: type `nfsubfields(nf)`. Indeed, there's a quadratic subfield, but it's given by `T = x^2 + 22*x + 133` and I don't recognize it. But `nfisisom(T, polcyclo(3))` indeed tells us that the fields $\mathbf{Q}[x]/(T)$

and $\mathbf{Q}[x]/(x^2+x+1)$ are isomorphic. (In fact, `polred(T)` would tell us the same, but does not correspond to a foolproof test: `polred` could have returned some other polynomials.)

We may also check that `k = matker(M-1)` is two-dimensional, then `z = nfbasistoalg(nf, k[,2])` generates the quadratic subfield. Notice that 1, `z` and `u` are $\mathbf{Q}$-linearly dependent, and in fact $\mathbf{Z}$-linearly as well. Exercise: how would you check these two assertions in general? (Answer: `concat`, then respectively `matrank` or `matkerint` (or `qflll`)). `z = charpoly(z)`, `z = gcd(z,z')` and `polred(z)` tell us that we found back the same subfield again (as we ought to!).

Final check: type `nfrootsof1(nf)`. Again we find that K contains a cube root of unity, since the torsion subgroup of its unit group has order 6. The given generator happens to be equal to `u`.

Additional comment. (you are no longer supposed to skip this, but do as you wish):

Before working with ideals, let us note one more thing. The main part of the work of `polred` or `nfinit`(T) is to compute an integral basis, i.e. a $\mathbf{Z}$-basis of the maximal order $\mathbf{Z}_K$ of K. This implies factoring the discriminant of the polynomial T, which is often not feasible. The situation may be improved in many ways:

1) First, it is often the case that our number field is of quite a special type. For example, one may know in advance some prime divisors of the discriminant. Hence we can "help" PARI by giving it that information. More precisely, we can use the function `addprimes` to inform PARI to keep on eye for these prime numbers. Do it only for big primes ! (Say, larger than `primelimit`.)

2) The second way in which the situation may be improved is that often we do not need the complete information on the maximal order, but only require that the order be p-maximal for a certain number of primes p — but then, we may not be able to use functions which require a genuine `nf`. The function `nfbasis` specifically computes the integral basis and is not much quicker than `nfinit` so is not very useful in its standard use. But we can optionally provide a list of primes: this returns a basis of an order which is p-maximal at the given primes. For example coming back to our initial polynomial T, the discriminant of the polynomial is $3^7 \cdot 7^6 \cdot 19 \cdot 37$. If we only want a 7-maximal order, we simply type

```
nfbasis([T, [7]])
```

Of course, `nfbasis([T, [2,3,5,7]])` would return an order which is maximal at $2, 3, 5, 7$. A variant offers a nice generalization:

```
nfbasis([T, 10^5])
```

will return an order which is maximal at all primes less than 10^5.

3) Building on the previous points, *if* the field discriminant is y-smooth (never mind the polynomial discriminant), up to a few big primes known to `addprimes`, then `bas = nfbasis(T, y)` returns a basis for the maximal order! We can then input the resulting basis to `nfinit`, as `nfinit([T, bas])`. Better: the [T, *listP*] format can be directly used with nfinit, where *listP* specifies a finite list of primes in one of the above ways (explicit list or primes up to some bound), and the result can be unconditionally certified, independently of the *listP* parameter:

```
T = polcompositum(x^7-2, polcyclo(5))[1];
K = nfinit( [T, [2,5,7]] );
nfcertify(K)
```

The output is a list of composite integers whose complete factorization must be computed in order to certify the result (which may be very hard, hence is not done on the spot). When the list is empty, as here, the result is unconditional

```
nfcertify(nf)
```

12.2. Ideals.

We now want to work with ideals and not only with elements. An ideal can be represented in many different ways. First, an element of the field (in any of the various guises seen above) will be considered as a principal ideal. Then the standard representation is a square matrix giving the Hermite Normal Form (HNF) of a $\mathbf{Z}$-basis of the ideal expressed on the integral basis `nf.zk`. Standard means that most ideal related functions will use this representation for their output.

Prime ideals can be represented in a special form as well (see `idealprimedec`) and all ideal-related functions will accept them. On the other hand, the function `idealtwoelt` can be used to find a two-element $\mathbf{Z}_K$-basis of a given ideal (as $a\mathbf{Z}_K + b\mathbf{Z}_K$, where a and b belong to K), but this is *not* a valid representation for an ideal under `gp`, and most functions will choke on it (or worse, take it for something else and output a meaningless result). To be able to use such an ideal, you will first have to convert it to HNF form.

Whereas it's very easy to go to HNF form (use `idealhnf(nf,id)` for valid ideals, or `idealhnf(nf,a,b)` for a two-element representation as above), it's a much more complicated problem to check whether an ideal is principal and find a generator. In fact an `nf` does not contain enough data for this particular task. We'll need a Buchmann Number Field, or `bnf`, for that. In particular, we need the class group and fundamental units, at least in some approximate form. More on this later (which will trivialize the end of the present section).

Let us keep our number field K as above and its `nf` structure. Type

```
P = idealprimedec(nf,7)
```

This gives the decomposition of the prime number 7 into prime ideals. We have chosen 7 because it divides `nf.index` (in fact, is equal to it), hence is the most difficult case to treat.

The result is a vector with 4 components, showing that 7 is totally split in the field K into prime ideals of norm 7 (you can check: `idealnorm(nf,P[1])`). Let us take one of these ideals, say the first, so type

```
pr = P[1]
```

We obtain its inertia and residue degree as `pr.e` and `pr.f`, and its two generators as `pr.gen`. One of them is `pr.p` $= 7$, and the other is guaranteed to have valuation 1 at `pr`. What is the Hermite Normal Form of this ideal? No problem:

```
idealhnf(nf,pr)
```

and we have the desired HNF. Let's now perform ideal operations. For example type

```
idealmul(nf, pr, idealmul(nf, pr,pr))
```

or more simply

```
pr3 = idealpow(nf, pr,3)
```

to get the cube of the ideal `pr`. Since the norm of this ideal is equal to $343 = 7^3$, to check that it is really the cube of `pr` and not of other ideals above 7, we can type

```
for(i=1, #P, print( idealval(nf, pr3, P[i]) ))
```

and we see that the valuation at `pr` is equal to 3, while the others are equal to zero. We could see this as well from `idealfactor(nf, pr3)`.

Let us now work in the class group "by hand" (we shall see simpler ways later). We shall work with *extended ideals*: an extended ideal is a pair `[A, t]`, where A is an ordinary ideal as above, and t a number field element; this pair represents the ideal $(t)A$.

```
id3 = [pr3, 1]
r0 = idealred(nf, id3)
```

The input `id3` is an extended ideal: pr3 together with 1 (trivial factorization). The new extended ideal `r0` is equal to the old one, in the sense that the products $(t)A$ are the same. It contains a "reduced" ideal equivalent to `pr3` (modulo the principal ideals), and a generator of the principal ideal that was factored out.

Now, just for fun type

```
r = r0; for(i=1,3, r = idealred(nf,r, [1,5]); print(r))
```

The ideals in the third `r` and initial `r0` are equal, say $(t)A = (t_0)A$: this means we have found a unit (t_0/t) in our field, and it is easy to extract this unit given the extended component:

```
t0 = r0[2]; t = r[2];
u = nfeltdiv(nf, t0, t)
u = nfbasistoalg(nf, u)
```

The last line recovers the unit as an algebraic number. Type

```
ch = charpoly(u)
```

and we obtain the characteristic polynomial `ch` of u again. (Whose constant coefficient is 1, hence u is indeed a unit.)

There is of course no reason for u to be a fundamental unit. Let us see if it is a square. Type

```
F = factor(subst(ch, x, x^2))
```

We see that `ch(x^2)` is a product of 2 polynomials of degree 4, hence u is a square. (Why?) We now want to find its square root. A simple method is as follows:

```
NF = subst(nf,x,y);
r = F[1,1] % (x^2 - nfbasistoalg(NF, u))
```

to find the remainder of the characteristic polynomial of `u2` divided by `x^2 -` u. This is a polynomial of degree 1 in `x`, with polmod coefficients, and we know that `u2`, being a root of both polynomials, is the root of `r`, hence can be obtained by typing

```
u2 = -polcoef(r,0) / polcoef(r,1)
```

There is an important technicality in the above: why did we need to substitute `NF` to `nf`? The reason is that data related to `nf` is given in terms of the variable `x`, seen modulo `nf.pol`; but we need `x` as a free variable for our polynomial divisions. Hence the substitution of `x` by `y` in our `nf` data.

The most natural method is to try directly

```
nffactor(nf, y^2 - u)
```

Except that this won't work for the same technical reason as above: the main variable of the polynomial to be factored must have *higher* priority than the number field variable. This won't be possible here since `nf` was defined using the variable `x` which has the highest possible priority. So we need to substitute variables around:

```
nffactor(NF, x^2 - nfbasistoalg(NF, subst(lift(u),x,y)))
```

(Of course, with better planning, we would just have defined `nf` in terms of the `'y` variable, to avoid all these substitutions.)

A much simpler approach is to consider the above as instances of a *discrete logarithm* problem, where we want to express some elements an abelian group (of finite type) in terms of explicitly given generators, and transfer all computations from abstract groups like $\mathrm{Cl}(K)$ and $\mathbf{Z}_K^*$ to products of simpler groups like $\mathbf{Z}^n$ or $\mathbf{Z}/d\mathbf{Z}$. We shall do exactly that in the next section.

Before that, let us mention another famous (but in fact, simpler) *discrete logarithm* problem, namely the one attached to the invertible elements modulo an ideal: $(\mathbf{Z}_K/I)^*$. Just use `idealstar` (this is an `init` function) and `ideallog`.

Many more functions on ideals are available. We mention here the complete list, referring to Chapter 3 for detailed explanations:

`idealadd`, `idealaddtoone`, `idealappr`, `idealchinese`, `idealcoprime`, `idealdiv`, `idealfactor`, `idealhnf`, `idealintersect`, `idealinv`, `ideallist`, `ideallog`, `idealmin`, `idealmul`, `idealnorm`, `idealpow`, `idealprimedec`, `idealred`, `idealstar`, `idealtwoelt`, `idealval`, `nfisideal`.

We suggest you play with these to get a feel for the algebraic number theory package. Remember that when a matrix (usually in HNF) is output, it is always a $\mathbf{Z}$-basis of the result expressed on the *integral basis* `nf.zk` of the number field, which is usually *not* a power basis.

12.3. Class Groups and Units, `bnf`.

Apart from the above functions you have at your disposal the powerful function `bnfinit`, which initializes a `bnf` structure, i.e. a number field with all its invariants (including class group and units), and enough technical data to solve discrete logarithm problems in the class and unit groups.

First type `setrand(1)`: this resets the random seed (to make sure we and you get the exact same results). Now type

```
bnf = bnfinit(NF);
```

where `NF` is the same number field as before. You do not want to see the output clutter a number of screens so don't forget the semi-colon. (Well if you insist, it is about three screenful in this case, but may require several Megabytes for larger degrees.) Note that `NF` is now expressed in terms of the variable `y`, to avoid later problems with variable priorities.

A word of warning: both the `bnf` and all results obtained from it are *conditional* on a Riemann Hypothesis at this point; the `bnf` must be certified before the following statements become actual theorems.

Member functions are still available for `bnf` structures. So, let's try them: `bnf.pol` gives `A`, `bnf.sign`, `bnf.disc`, `bnf.zk`, ok nothing really exciting. In fact, an `nf` is included in the `bnf` structure: `bnf.nf` should be identical to `NF`. Thus, all functions which took an `nf` as first argument, will equally accept a `bnf` (and a `bnr` as well which contains even more data).

Anyway, new members are available now: `bnf.no` tells us the class number is 4, `bnf.cyc` that it is cyclic (of order 4 but that we already knew), `bnf.gen` that it is generated by the ideal `g = bnf.gen[1]`. If you `idealfactor(bnf, g)`, you recognize `P[2]`. (You may also play in the other direction with `idealhnf`.) The regulator `bnf.reg` is equal to 3.794... . `bnf.tu` tells us that the roots

of unity in K are exactly the sixth roots of 1 and gives a primitive root $\zeta = \frac{1}{7}(\alpha^3 - 5\alpha^2 - 8\alpha + 56)$, which we have seen already. Finally `bnf.fu` gives us a fundamental unit $\epsilon = \frac{1}{7}(\alpha^3 - 5\alpha^2 - \alpha + 28)$, which must be linked to the units `u` and `u2` found above since the unit rank is 1. To find these relations, type

```
bnfisunit(bnf, u)
bnfisunit(bnf, u2)
```

Lo and behold, `u` = $\zeta^2\epsilon^2$ and `u2` = $\zeta^4\epsilon^1$.

Note. Since the fundamental unit obtained depends on the random seed, you could have obtained another unit than ϵ, had you not reset the random seed before the computation. This was the purpose of the initial `setrand` instruction, which was otherwise unnecessary.

We are now ready to perform operations in the class group. First and foremost, let us certify the result: type `bnfcertify(bnf)`. The output is `1` if all went well; in fact no other output is possible, whether the input is correct or not, but you can get an error message (or in exceedingly rare cases an infinite loop) if it is incorrect.

It means that we now know the class group and fundamental units unconditionally (no more GRH then!). In this case, the certification process takes a very short time, and you might wonder why it is not built in as a final check in the `bnfinit` function. The answer is that as the regulator gets bigger this process gets increasingly difficult, and becomes soon impractical, while `bnfinit` still happily spits out results. So it makes sense to dissociate the two: you can always check afterwards, if the result is interesting enough. Looking at the tentative regulator, you know in advance whether the certification can possibly succeed: if `bnf.reg` is large, don't waste your time.

Now that we feel safe about the `bnf` output, let's do some real work. For example, let us take again our prime ideal `pr` above 7. Since we know that the class group is of order 4, we deduce that `pr` raised to the fourth power must be principal. Type

```
pr4 = idealpow(nf, pr, 4)
v = bnfisprincipal(bnf, pr4)
```

The first component gives the factorization of the ideal in the class group. Here, `[0]` means that it is up to equivalence equal to the 0-th power of the generator `g` given in `bnf.gen`, in other words that it is a principal ideal. The second component gives us the algebraic number α such that `pr4` = $\alpha\mathbf{Z}_K$, α being as usual expressed on the integral basis. Type `alpha = v[2]`. Let us check that the result is correct: first, type `idealnorm(bnf, alpha)`. (Note that we can use a `bnf` with all the `nf` functions; but not the other way round, of course.) It is indeed equal to $7^4 = 2401$, which is the norm of `pr4`. This is only a first check. The complete check is obtained by computing the HNF of the principal ideal generated by `alpha`. To do this, type `idealhnf(bnf, alpha) == pr4`.

Since the equality is true, `alpha` is correct (not that there was any doubt!). But `bnfisprincipal` also gives us information for nonprincipal ideals. For example, type

```
v = bnfisprincipal(bnf, pr)
```

The component `v[1]` is now equal to `[3]`, and tells us that `pr` is ideal-equivalent to the cube of the generator `g`. Of course we already knew this since the product of `P[3]` and `P[4]` was principal (generated by `al`), as well as the product of all the `P[`i`]` (generated by 7), and we noticed that `P[2]` was equal to `g`, which has order 4. The second component `v[2]` gives us α on the integral basis such that `pr` = α`g`^{3}. Note that if you *don't* want this α, which may be large and whose computation may take some time, you can just add the flag 1 (see the online help) to the arguments of `bnfisprincipal`, so that it only returns the position of `pr` in the class group.

12.4. Class Field Theory, `bnr`.

We now survey quickly some class field theoretic routines. We must first initialize a Buchmann Number Ray, or `bnr`, structure, attached to a `bnf` base field and a modulus. Let's keep K, and try a finite modulus $\mathfrak{f} = 7\mathbf{Z}_K$. (See the manual for how to include infinite places in the modulus.) Since K will now become a base field over which we want to build relative extensions, the attached `bnf` needs to have variables of lower priority than the polynomials defining the extensions. Fortunately, we already took care that, but it would have been easy to deal with the problem now (as easy as `bnf = subst(bnf, x, y)`). Then type

```
bnr = bnrinit(bnf, 7, 1);
bnr.cyc
```

tells us the ray class group modulo $\mathfrak{f}$ is isomorphic to $\mathbf{Z}/24\mathbf{Z}\times\mathbf{Z}/6\mathbf{Z}\times\mathbf{Z}/2\mathbf{Z}$. The attached generators are `bnr.gen`. Just as a `bnf` contained an `nf`, a `bnr` contains a `bnf` (hence an `nf`), namely `bnr.bnf`. Here `bnr.clgp` refers to the ray class group, while `bnr.bnf.clgp` refers to the class group.

```
rnfkummer(bnr,, 2)
rnfkummer(bnr,, 3)
```

outputs defining polynomials for the 2 abelian extensions of K of degree 2 (resp. 3), whose conductor is exactly equal to $\mathfrak{f}$ (the modulus used to define `bnr`). (In the current implementation of `rnfkummer`, these degrees must be *prime*.) What about other extensions of degree 2 for instance?

```
L0= subgrouplist(bnr, [2])
L = subgrouplist(bnr, [2], 1)
```

`L0`, resp. `L` is the list of those subgroups of the full ray class group mod 7, whose index is 2, and whose conductor is 7, resp. arbitrary. (Subgroups are given by a matrix of generators, in terms of `bnr.gen`.) `L0` has 2 elements, attached to the 2 extensions we already know. `L` has 7 elements, the 2 from `L0`, and 5 new ones:

```
L1 = setminus(Set(L), Set(L0))
```

The conductors are

```
vector(#L1, i, bnrconductor(bnr, L1[i]))
```

among which one sees the identity matrix, i.e. the trivial ideal. (It is `L1[3]` in my session, maybe not in yours. Take the right one!) Indeed, the class group was cyclic of order 4 and there exists a unique unramified quadratic extension. We could find it directly by recomputing a `bnr` with trivial conductor, but we can also use

```
rnfkummer(bnr, L1[3]) \\  pick the subgroup with trivial conductor!
```

directly which outputs the (unique by Takagi's theorem) class field attached to the subgroup `L1[3]`. In fact, it is of the form $K(\sqrt{-\epsilon})$. We can check this directly:

```
rnfconductor(bnf, x^2 + bnf.fu[1])
```

12.5. Galois Theory over Q.

PARI includes a nearly complete set of routines to compute with Galois extensions of $\mathbf{Q}$. We start with a very simple example. Let ζ a 8th-root of unity and $K = \mathbf{Q}(\zeta)$. The minimal polynomial of ζ is the $8th$ cyclotomic polynomial, namely `polcyclo(8)` $(=x^4+1)$.

We issue the command

```
G = galoisinit(x^4 + 1);
```

to compute $G = \mathrm{Gal}(K/\mathbf{Q})$. The command `galoisisabelian(G)` returns `[2,0;0,2]` so G is an abelian group, isomorphic to $(\mathbf{Z}/2\mathbf{Z})^2$, generated by σ=`G.gen[1]` and τ=`G.gen[2]`. These automorphisms are given by their actions on the roots of x^4+1 in a suitable p-adic extension. To get the explicit action on ζ, we use `galoispermtopol(G,G.gen[i])` for $i = 1, 2$ and get $\sigma(\zeta) = -\zeta$ and $\tau(\zeta) = \zeta^3$. The last nontrivial automorphism is $\sigma\tau$=`G.gen[1]*G.gen[2]` and we have $\sigma\tau(\zeta) = -\zeta^3$. (At least in my version, yours may return a different set of generators, rename accordingly.)

We compute the fixed field of K by the subgroup generated by τ with

```
galoisfixedfield(G, G.gen[2], 1)
```

and get x^2+2. Now we want the factorization of x^4+1 over that subfield. Of course, we could use `nffactor`, but here we have a much simpler option: `galoisfixedfield(G, G.gen[1], 2)` outputs

```
[x^2 + 2, Mod(x^3 + x, x^4 + 1), [x^2 - y*x - 1, x^2 + y*x - 1]]
```

which means that $x^4 + 1 = (x^2 - \alpha x - 1)(x^2 + \alpha x - 1)$ where α is a root of $x^2 + 2$, and more precisely, $\alpha = \zeta^3 + \zeta$. So we recover the well-known factorization:

$$x^4 + 1 = (x^2 - \sqrt{-2}x - 1)(x^2 + \sqrt{-2}x - 1)$$

For our second example, let us take the field K defined by the polynomial

```
P = x^18 - 3*x^15 + 115*x^12 + 104*x^9 + 511*x^6 + 196*x^3 + 343;
G = galoisinit(P);
```

Since `galoisinit` succeeds, the extension $K/\mathbf{Q}$ is Galois. This time `galoisisabelian(G)` returns 0, so the extension is not abelian; however we can still put a name on the underlying abstract group. Use `galoisidentify(G)`, which returns $[18, 3]$. By looking at the GAP4 classification we find that $[18, 3]$ is $S_3 \times \mathbf{Z}/3\mathbf{Z}$. This time, the subgroups of G are not obvious, fortunately we can ask PARI : `galoissubgroups(G)`.

Let us look for a polynomial Q with the property that K is the splitting field of $Q(x^2)$. For that purpose, let us take σ=`G.gen[3]`. We check that `G.gen[3]^2` is the identity, so σ is of order 2. We now compute the fixed field K^σ and the relative factorization of P over K^σ:

```
F = galoisfixedfield(G, G.gen[3], 2);
```

So K is a quadratic extension of K^σ defined by the polynomial `R=F[3][1]`. It is well-known that K is also defined by $x^2 - D$, where D is the discriminant of R (over K^σ). To compute D, we issue:

```
D = poldisc(F[3][1]) * Mod(1,subst(F[1],x,y));
```

Note that since `y` in `F[3][1]` denotes a root of `F[1]`, we have to use `subst(,x,y)`. Now we hope that D generates K^σ and compute `Q=charpoly(D)`. We check that $Q = x^9 + 270x^6 + 12393 \times x^3 + 19683$ is irreducible with `polisirreducible(Q)`. (Were it not the case, we would multiply D by a random square.) So D is a generator of K^σ and $\sqrt{D}$ is a generator of K. The result is that K is the splitting field of $Q(x^2)$. We can check that with `nfisisom(P,subst(Q,x,x^2))`.

12.6. Creating and Listing Number Fields.

Suppose you want to work with cyclic quartic fields (i.e., fields with Galois group $C_4 = \mathbf{Z}/4\mathbf{Z}$ over $\mathbf{Q}$): you could look in a database or in a book for examples, but PARI provides the simple but powerful `nflist` command: in this case you could simply write `V = nflist("C4")`, which would immediately return a list of polynomials (in the present case 18, in some random order) defining cyclic quartic fields. Note that these polynomials are usually not the simplest, you can for instance do `V = apply(polredbest, V)` to reduce them.

Using `vecmax(apply(z->abs(nfdisc(z)),V))`, you can check that the largest discriminant is 35152. Thus, if you only want discriminants up to 10000 you would write `V = nflist("C4", [1, 10000])`, which would now output only 10 number fields. Or if you want only fields with absolute discriminant exactly equal to 8000, you would write `V = nflist("C4", 8000)`, which would output the two polynomials $x^4 \pm 10x^2 + 20$, which define non-isomorphic number fields (using `polsturm`, you can check that one is totally real, the other totally complex). Finally, if you only want C_4-fields with a given number `s` of complex places, you would write `nflist("C4",,s)` (don't forget the two consecutive commas, since the second argument is the discriminant range), for instance:

```
for(s = 0, 2, print1([s, #nflist("C4", , s)]," "))
```

would output `[0, 10] [1, 0], [2, 10]`, so `nflist` gives you 10 totally real and 10 totally complex cyclic quartic fields, but none with mixed signature $(2, 1)$, which is a good thing since they do not exist!

Of course you can ask for many more: for instance in a few seconds

```
apply(length, #nflist("C4", [1, 10^14], -2))
```

will return `[607589, 0, 607609]`, which tells you that there are $607,589$ totally real and $607,609$ totally complex cyclic quartic fields with absolute discriminant up to 10^{14}: this illustrates the special use of the parameter $s = -2$, which separates the number fields by increasing number of complex places (the default is $s = -1$ which ignores the signature altogether).

Of course, `nflist` applies to many other Galois groups than C_4 (where "Galois group" is understood as Galois group of the Galois closure, see below), but before leaving cyclic quartic fields, let us see two more related instructions. Starting again with `V = nflist("C4")`, write `R = apply(nfresolvent, V)`: this outputs a list of 18 quadratic polynomials, which define the unique quadratic subfield of each quartic field. In the present example, the `nfsubfields` command (more precisely `R = apply(z -> polredbest(nfsubfields(z, 2, 1)[1]), V)`) would have given an identical result, but the purpose of `nfresolvent` is different. First, for many Galois groups, it really outputs what one usually calls the resolvent polynomial, for instance a quadratic polynomial for an S_3 field, a cubic for a non-Galois quartic, etc... Second, as here, the fields are often constructed as extensions of subfields, and `nfresolvent` gives the subfield on which the field has been constructed. Better, typing `nfresolvent(V[1], 1)` outputs `[x^2 - x - 1, [[5, 2; 0, 1], [1, 1]]]`, which not only gives the subfield, but the relative conductor as a `module` (a pair ideal, list of real places).

The second instruction is `polsubcyclo`, which has existed for a long time in PARI. The manual states that `polsubcyclo(n,d)` gives the subfields of degree d of $\mathbf{Q}(\zeta_n)$. However, if n is very large, previous versions would take an inordinate amount of time. The present version allows us to take n very large, but the price to pay is that the degree d of the desired subfields must be very small, for instance $d \leq 30$, and in addition must be a prime number if $d \geq 7$. For instance:

```
? p = 10^16 + 649759; /* A prime chosen so that p = 1 mod 11*19*29*41
```

```
? polsubcyclo(p, 11);
time = 42 ms. /* Would have been impossible in previous versions */
? polsubcyclo(p, 19);
time = 348 ms.
? polsubcyclo(p, 29);
time = 10,304 ms. /* Still reasonable */
? polsubcyclo(p, 41);
time = 6min, 43,181 ms. /* Slow but feasible. */
```

The most common Galois groups corresponding to number fields of small degree have been implemented: all solvable groups up to degree 7 (there are 13 in degree 6 for instance), the groups C_ℓ and D_ℓ for ℓ prime. In addition, the simple group A_5 has also been included, but by cheating since it is obtained by table lookup (and available only of you have downloaded and installed the `nflistdata` package), in particular because of its relation with modular forms of weight 1.

13. Working with Associative Algebras.

Beyond the realm of number fields, we can perform operations with more general associative algebras, that need not even be commutative! Of course things become more complicated. We have two different structures: the first one allows us to manipulate any associative algebra that is finite-dimensional over a prime field ($\mathbf{Q}$ or $\mathbf{F}_p$ for some prime p), and the second one is dedicated to central simple algebras over number fields, which are some nice algebras that behave a lot like number fields. Like in other parts of `gp`, every function that has to do with associative algebras begins with the same prefix: `alg`.

13.1. Arbitrary Associative Algebras.

In order to create an associative algebra, you need to tell `gp` how to multiply elements. We do this by providing a *multiplication table* for the algebra, in the form of the matrix of left multiplication by each basis element, and use the function `algtableinit`.

For instance, let us work in $\mathbf{F}_3[x]/(x^2)$. Of course, we could use polmods of intmods to represent elements in this algebra, but let's introduce the general mechanism with this simple example! This algebra has a basis with two elements: 1 and ϵ, the image of x in the quotient. By the way, `gp` will only accept your multiplication table if the first basis vector is 1. The multiplication matrix of 1 is the 2×2 identity matrix, and since $\epsilon \cdot 1 = \epsilon$ and $\epsilon \cdot \epsilon = 0$, the left multiplication table of ϵ is $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$. So we use the following multiplication table:

```
mt1 = [matid(2), [0,0;1,0]]
```

Since we want our algebra to be over $\mathbf{F}_3$, we have to specify the characteristic and create the algebra with

```
al1 = algtableinit(mt1, 3);
```

Let's create another one: the algebra of upper-triangular 2×2 matrices over $\mathbf{Q}$. This algebra has dimension 3, with basis 1, $a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ and $b = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, and these elements satisfy $a^2 = 0$, $ab = a$, $ba = 0$ and $b^2 = b$. Watch out, even though a and b are 2×2 matrices, their left multiplication tables are 3×3 matrices! The left multiplication tables of a and b are respectively

```
ma = [0,0,0; 1,0,1; 0,0,0];
mb = [0,0,0; 0,0,0; 1,0,1];
```

The multiplication table of our second algebra is therefore

```
mt2 = [matid(3), ma, mb];
```

and we can create the algebra with

```
al2 = algtableinit(mt2,0);
```

In fact, we can omit the second argument and type

```
al2 = algtableinit(mt2);
```

and the characteristic of the algebra will implicitly be 0. Warning: in characteristic 0, `algtableinit` expects an integral multiplication table.

In fact, gp does not check that the multiplication table you provided really defines an associative algebra. You can check it a posteriori with

```
algisassociative(al2)
```

or before creating the algebra with

```
algisassociative(mt1,3)
```

After creating the algebra, you can get back the multiplication table that you provided with `algmultable(al2)`, the characteristic with `algchar(al1)` and the dimension with `algdim(al2)`.

13.1.1. Elements.

In an associative algebra, we represent elements as column vectors expressing them on the basis of the algebra. For instance, in `al1` $= \mathbf{F}_3[\epsilon]$, the element $1 - \epsilon$ is represented as `[1,-1]~`. Similarly, in `al2` we can define a, b and $c = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$ by

```
a = [0,1,0]~
b = [0,0,1]~
c = [2,1,-1]~
```

We can also draw random elements in a box using

```
algrandom(al2, 10)
```

You can compute any elementary operation: try various combinations of `algadd`, `algsub`, `algneg`, `algmul`, `alginv`, `algsqr`, `algpow`, using the syntax

```
algmul(al2, b, a)
algpow(al2, c, 10)
```

and the natural variants. In every algebra we have the left regular representation, which sends every element x to the matrix of left multiplication by x. In gp we access it by calling

```
algtomatrix(al2, c)
```

For every element x in an associative algebra, the trace of that matrix is called the *trace* of x, the determinant is called the *norm* of x, and the characteristic polynomial is called the *characteristic polynomial* of x. We can compute them:

```
algtrace(al2, a)
algnorm(al2, c)
algcharpoly(al2, b)
```

13.1.2. Properties.

Now let's try to compute some interesting properties of our algebras. Maybe the simplest one we want to test is whether an algebra is commutative; `algiscommutative` does that for us: you can use it to check that `al1` is of course commutative, but `al2` is not since for instance $ab = a \neq 0 = ba$. More precisely, we can compute a basis of the center of an algebra with `algcenter`. Since `al1` is commutative, we obtain the identity matrix, and

```
algcenter(al2)
```

tells us that the center of `al2` is one-dimensional and generated by the identity.

An important object in the structure of associative algebras is the *Jacobson radical*: the set of elements that annihilate every simple left module. It is a nilpotent two-sided ideal. An algebra is *semisimple* if its Jacobson radical is zero, and for every algebra A with radical J, the quotient A/J is semisimple. You can compute a basis for the Jacobson radical using `algradical`. For instance, the radical of `al1` is generated by the element ϵ. Indeed, in a commutative algebra the Jacobson radical is equal to the set of nilpotent elements. The radical of `al2` has basis a: it is the subspace of strictly upper-triangular 2×2 matrices. You can also directly test whether an algebra is semisimple using `algissemisimple`. Let's compute the semisimplification of our second algebra by quotienting out its radical:

```
al3 = algquotient(al2, algradical(al2));
```

Check that `al3` is indeed semisimple now that you know how to do it!

Group algebras provide interesting examples: when G is a finite group and F is a field, the group algebra $F[G]$ is semisimple if and only if the characteristic of F does not divide the order of G. Let's try it!

```
K = nfsplitting(x^3-x+1); \\ Galois closure -> S_3
G = galoisinit(K)
al4 = alggroup(G, 5) \\ F_5[S_3]
algissemisimple(al4)
```

Check what happens when you change the characteristic of the algebra.

The building blocks of semisimple algebras are *simple* algebras: algebras with no nontrivial two-sided ideals. Since the Jacobson radical is a two-sided ideal, every simple algebra is semisimple. You can check whether an algebra is simple using `algissimple`. For instance, you can check that

```
algissimple(al1)
```

returns `0`, but this is not very interesting since `al1` is not even semisimple. Instead we can test whether `al3` is simple, and since we already know that it is semisimple we can prevent `gp` from checking it again by using the optional second argument:

```
algissimple(al3, 1)
```

We see that `al3` is not simple, and this implies that we can decompose it further. Indeed, every semisimple algebra is isomorphic to a product of simple algebras. We can obtain this decomposition with

```
dec3 = algsimpledec(al3)[2];
apply(algdim, dec3)
```

We see that `al3` is isomorphic to $\mathbf{Q} \times \mathbf{Q}$. Similarly, we can decompose `al4`.

```
dec4 = algsimpledec(al4)[2];
apply(algdim, dec4)
```

We see that `al4` is isomorphic to $\mathbf{F}_5 \times \mathbf{F}_5 \times A$ where A is a mysterious 4-dimensional simple algebra. However every simple algebra over a finite field is isomorphic to a matrix algebra over a possibly larger finite field. By computing the center

```
algcenter(dec4[3])
```

we see that this algebra has center $\mathbf{F}_5$, so that `al4` is isomorphic to $\mathbf{F}_5 \times \mathbf{F}_5 \times M_2(\mathbf{F}_5)$.

13.2. Central Simple Algebras over Number Fields.

As we saw, simple algebras are building blocks for more complicated algebras. The center of a simple algebra is always a field, and we say that an algebra over a field F is *central* if its center is F. The most natural noncommutative generalization of a number field is a *division algebra* over $\mathbf{Q}$: an algebra in which every nonzero element is invertible. Since the center of a division algebra over $\mathbf{Q}$ is a number field, we do not lose any generality by considering central division algebras over number fields. However, the tensor product of two central division algebras is not always a division algebra, but division algebras are always simple and central simple algebras are closed under tensor products, giving a much nicer global picture. This is why we choose central simple algebras over number fields as our noncommutative generalization of number fields.

13.2.1. Creation.

Let's create our first central simple algebras! A well-known construction is that of *quaternion algebras*, which proceeds as follows. Let F be a number field, and let $a, b \in F^\times$; the quaternion algebra $(a, b)_F$ is the F-algebra generated by two elements i and j satisfying $i^2 = a$, $j^2 = b$ and $ij = -ji$. Hamilton's quaternions correspond to the choice $a = b = -1$, but it is not the only possible one! Here is our first quaternion algebra:

```
Q = nfinit(y);
al1 = alginit(Q,[-2,-1]); \\  (-2,-1)_Q
```

Note that we represented the rationals $\mathbf{Q}$ with an `nf` structure and used the variable y in that structure. The reason for this variable choice will be clearer after looking at the next construction. You can see from the definition of a quaternion algebra that it has dimension 4 over its center, a fact that you can check in our example:

```
algdim(al1)
```

Cyclic algebras generalize the quaternion algebra construction. Let F be a number field and K/F a cyclic extension of degree d with $\sigma \in \mathrm{Gal}(K/F)$ an automorphism of order d, and let $b \in F^\times$; the *cyclic algebra* $(K/F, \sigma, b)$ is the algebra $\bigoplus_{i=0}^{d-1} u^i K$ with $u^d = b$ and $ku = u\sigma(k)$ for all $k \in K$. It is a central simple algebra of dimension d^2 over F. Let's construct one. First, start with a cyclic extension of number fields. A simple way of obtaining such an extension is to take a cyclotomic field over $\mathbf{Q}$.

```
T = polcyclo(5)
K = rnfinit(Q,T);
aut = Mod(x^2,T)
```

Here the variable of `T` must have higher priority than that of `Q` to build the `rnf` structure. Now choose an element $b \in \mathbf{Q}^\times$, say 3.

```
b = 3
```

Now we can create the algebra and check that it has the right dimension: 16.

```
al2 = alginit(K, [aut,b]);
algdim(al2)
```

We can recover the field `K`, the automorphism `aut` and the element `b` respectively as follows.

```
algsplittingfield(al2)
algaut(al2)
algb(al2)
```

In order to see how we recover quaternion algebras with this construction, let's look at

```
algsplittingfield(al1).pol
algaut(al1)
algb(al1)
```

We see that the quaternion algebra $(-2,-1)_{\mathbf{Q}}$ constructed by `gp` is represented as a cyclic algebra $(\mathbf{Q}(\sqrt{-2})/\mathbf{Q},\sigma,-1)$ by writing $\mathbf{Q}+\mathbf{Q}i+\mathbf{Q}j+\mathbf{Q}ij=\mathbf{Q}(i)+j\mathbf{Q}(i)$.

A nice feature of central simple algebras over a given number field is that they are completely classified up to isomorphism, in terms of certain invariants. We therefore provide functions to create an algebra directly from its invariants. The first basic invariant is the dimension of the algebra over its center. In fact, the dimension of a central simple algebra is always a square, and we call the square root of the dimension the *degree* of the algebra. For instance, a quaternion algebra has degree 2. We can access the degree with

```
algdegree(al1)
algdegree(al2)
```

Let F be a number field and A a central simple algebra of degree d over F. To every place v of F we can attach a *Hasse invariant* $h_v(A)\in(\frac{1}{d}\mathbf{Z})/\mathbf{Z}\subset\mathbf{Q}/\mathbf{Z}$, with the additional restrictions that the invariant is 0 at every complex place and in $(\frac{1}{2}\mathbf{Z})/\mathbf{Z}$ at every real place. These invariants are 0 at all but finitely many places. For instance we can compute the invariants for our algebras:

```
alghassei(al1)
alghassef(al1)
alghassei(al2)
alghassef(al2)
```

The output of `alghassei` (infinite places) is a vector of integers of length the number of real places of F, and the corresponding Hasse invariants are these integers divided by d. Here we learn that the invariant of `al1` at ∞ is $1/2$ and that of `al2` is 0. Similarly, the output of `alghassef` (finite places) is a pair of vectors, one containing primes of the base field, and a vector of integers of the same length representing the Hasse invariants at finite places. We learn that `al1` has invariant $1/2$ at 2 and 0 at every other finite place, and that `al2` has invariant $-1/3$ at 3, invariant $1/3$ at 5, and 0 at every other finite place. These invariants give a complete classification of central simple algebras over F:

• two central simple algebras over F of the same degree are isomorphic if and only if they have the same invariants;

• the sum of the Hasse invariants is 0 in $\mathbf{Q}/\mathbf{Z}$;

• for every degree d and finite collection of Hasse invariants satisfying the conditions above, there exists a central simple algebra over F having those invariants.

Let's use gp to construct an algebra from its invariants. First let's construct a number field and a few places.

```
nf = nfinit(y^2-2);
p3 = idealprimedec(nf,3)[1]
p17 = idealprimedec(nf,17)[1]
```

Now let's construct an algebra of degree 6. The following Hasse invariants satisfy the correct conditions since $1/3 + 1/6 + 1/2 = 0$ in $\mathbf{Q}/\mathbf{Z}$:

```
hf = [[p3,p17],[1/3,1/6]]; hi = [1/2,0]; \\ nf has 2 real places
```

Finally we can create the algebra:

```
al3 = alginit(nf,[6,hf,hi]);
```

This will require less than half a minute but a lot of memory: this is an algebra of dimension $2\times 6^2 = 72$ over $\mathbf{Q}$ and we are doing a nontrivial computation in the initialization! You can check that the dimension over $\mathbf{Q}$ is what we expect:

```
algdim(al3,1)
```

During the initialization, gp computes an integral multiplication table for the algebra. This allows us to recreate a table version of the algebra:

```
mt3 = algmultable(al3);
al4 = algtableinit(mt3);
```

We can then check that the algebra is simple as expected:

```
algissimple(al4)
```

Finally, an important test for a central simple algebra is whether it is a division algebra.

```
algisdivision(al3)
```

13.2.2. Elements.

In a cyclic algebra $(K/F, \sigma, b)$ of degree d, we represent elements as column vectors of length d, expressed on the basis of the K-vector space $\bigoplus_{i=0}^{d-1} u^i K$:

```
a = [x^3-1, -x^2, 3, -1]~*Mod(1,T) \\represents an element in al2
```

To represent elements in the quaternion algebra al1, we must view it as a cyclic algebra, and therefore use the representation al1 $= \mathbf{Q}(i) + j\mathbf{Q}(i)$. The standard basis elements $1, i, j, k = ij = -ji$ become

```
one = [1,0]~
i = [x,0]~
j = [0,1]~
k = [0,-x]~
```

The expected equalities hold:

```
algsqr(al1,i) == -2*one
algsqr(al1,j) == -1*one
algsqr(al1,k) == -2*one
algmul(al1,i,j) == k
algmul(al1,j,i) == -k
```

Like `nfinit` for number fields, the `alginit` function computes an integral basis of the algebra being initialized. More precisely, an *order* in a $\mathbf{Q}$-algebra A is a subring $\mathcal{O} \subset A$ that is finitely generated as a $\mathbf{Z}$-module and such that $\mathbf{Q}\mathcal{O} = A$. In an algebra structure computed with `alginit`, we store a basis of an order $\mathcal{O}_0$, which we will call the integral basis of the algebra. There is no canonical choice for such a basis, and not every integral element has integral coordinates with respect to that basis (the set of integral elements in A does not form a ring in general). By default, $\mathcal{O}_0$ is a maximal order and hence behaves in a way similar to the ring of integers in a number field. You can disable the (costly) computation of a maximal order with an optional argument:

```
al5 = alginit(nf,[6,hf,hi],,0);
```

This command should be faster than the initialization of `al3`. As in number fields, you can represent elements of central simple algebras in *algebraic form*, which means the cyclic algebra representation we described above, or in *basis form*, which means as a $\mathbf{Q}$-linear combination of the integral basis. You can switch between the two representations:

```
algalgtobasis(al2, a)
algalgtobasis(al1, j)
algbasistoalg(al3, algrandom(al3,1))
```

As usual you can compute any elementary operation: try various combinations of `algadd`, `algsub`, `algneg`, `algmul`, `alginv`, `algsqr`, `algpow`.

Every central simple algebra A over a number field F admits a *splitting field*, i.e. an extension K/F such that $A \otimes_F K \cong M_d(K)$. We always store such a splitting in an `alginit` structure, and you can access it using

```
algsplittingfield(al1) \\ K as an rnf structure
algtomatrix(al1, k) \\ image of k by the splitting isomorphism
```

For every $x \in A$, the trace (resp. determinant, characteristic polynomial) of that matrix is in K (resp. K, $K[X]$) and is called the *reduced trace* (resp. *reduced norm*, *reduced characteristic polynomial*) of x. You can compute them using

```
algtrace(al3, vector(72,i,i==3)~)
algnorm(al2, a)
algcharpoly(al1, -1+i+j+2*k)
```

14. Plotting.

PARI supports high and low-level graphing functions, on a variety of output devices: a special purpose window under standard graphical environments (the `X Windows` system, Microsoft Windows, macOS), or a `PostScript` file ready for the printer, or a Scalable Vector Graphics (SVG) image ready for the web.

14.1. Function graphs.

We first decrease drastically the default precision.

```
\p 9
```

This is very important, since plotting involves calculation of functions at a huge number of points, and a relative precision of 38 significant digits is an obvious overkill: the output device resolution certainly won't reach $10^{38} \times 10^{38}$ pixels!

Start with a simple plot:

```
ploth(x = -2*Pi, 2*Pi, sin(x))
```

Straightforward. Let's complicate things:

```
ploth(x = -2, 2, sin(x^7))
```

You can see the limitations of the "straightforward" mode of plotting: while the first several cycles of `sin` around $x = 0$ reach -1 and 1, the cycles which are closer to the left and right border do not. This is understandable: PARI is calculating $\sin(X^7)$ at many evenly spaced points, but these points have no direct relationship to the "interesting" points on the graph of this function. No value close enough to the extrema are calculated, which leads to wrong turning points in the graph. To fix this, one may use adaptive steps which are made smaller where the function varies rapidly:

```
ploth(x = -2, 2, sin(x^7), "Recursive")
```

However, the recursive plotting (named so since PARI subdivides intervals until the graph becomes almost straight) has its own pitfalls. Try

```
ploth(x = -2, 2, sin(7*x), "Recursive")
```

The graph looks correct far away, but it has a straight interval near the origin, and some sharp corners as well. This happens because the graph is symmetric with respect to the origin, thus the middle 3 points calculated during the initial subdivision of $[-2, 2]$ are exactly on the same line. To PARI this indicates that no further subdivision is needed, and it plots the graph on this subinterval as a straight line.

There are many ways to circumvent this. Say, one can make the right limit 2.1. Or one can ask PARI for an initial subdivision into 16 points instead of the default 15:

```
ploth(x = -2, 2, sin(x*7), "Recursive", 16)
```

These arrangements break the symmetry of the initial subdivision and make the problem go away. Eventually PARI will be able to better detect such pathological cases, but currently some manual intervention may be required.

The function `ploth` has some additional enhancements which allow graphing in situations when evaluation the function is expensive. As a moderatly complicated example, let us plot the real part or $\zeta(\frac{1}{2} + it)$:

```
ploth(t = 100, 110, real(zeta(0.5+I*t)), /*empty*/, 1000)
```

(1000 is close to the default for many plotting devices, we specify it explicitly so that the result does not depend on the output device.) Try the recursive plot:

```
ploth(t = 100, 110, real(zeta(0.5+I*t)), "Recursive")
```

It takes approximately twice the time. Now try specifying fewer points, but make PARI approximate the data by a smooth curve:

```
ploth(t = 100, 110, real(zeta(0.5+I*t)), "Splines", 100)
```

This is more than 10 times faster, and the output is practically the same. How to compare these two outputs? We will see it shortly. Right now let us plot both real and complex parts of ζ on the same graph:

```
f(t) = my(z = zeta(0.5+I*t)); [real(z),imag(z)];
ploth(t = 100, 110, f(t), /*empty*/, 1000)
```

(Note the use of the temporary variable `z`; `my` declares it local to the function's body.)

Note how one half of the roots of the real and imaginary parts coincide. Why did we define a function `f(t)`? To avoid calculation of $\zeta(\frac{1}{2} + it)$ twice for the same value of t. Similarly, we can plot parametric graphs:

```
ploth(t = 100, 110, f(t), "Parametric", 1000)
```

In that case (parametric plot of the real and imaginary parts of a complex function), we can also use directly

```
ploth(t = 100, 110, zeta(0.5+I*t), "Complex", 1000)
```

or even combine:

```
ploth(t = 100, 110, zeta(0.5+I*t), "Complex|Splines", 100)
```

If your plotting device supports it, you may ask PARI to show the points in which it calculated your function:

```
ploth(t = 100, 110, f(t), "Parametric|Splines|Points_too", 100)
```

As you can see, the points are very dense on the graph. To see some crude graph, one can even decrease the number of points to 30. However, if you decrease the number of points to 20, you can see that the approximation to the graph now misses zero. Using splines, one can create reasonable graphs for larger values of t, say with

```
ploth(t = 100000, 100004, f(t), "Parametric|Splines|Points_too", 50)
```

14.2. Low-level plotting.

How can we compare two graphs of the same function plotted by different methods? Documentation shows that `ploth` does not provide any direct method to do so. However, it is possible, and even not very complicated.

PARI has a power mix of high level plotting function with low level plotting functions, and these functions can be combined together to obtain many different effects. We will use a multitude of numeric flags. To simplify understanding we first give these flags symbolic names:

```
\\ Relative positioning of graphic objects:
nw = 0;  se = 4;  relative = 1;
sw = 2;  ne = 6;

\\ String positioning:
/* V */ bottom  =  0;  /* H */  left   = 0;  /* Fine tuning */ hgap = 16;
        vcenter =  4;            center = 1;                    vgap = 32;
```

```
top     =  8;           right  = 2;
```

Return back to the graph of $\sin(x^7)$. Suppose we want to create an additional rectangular frame around our graph. No problem!

First, all low-level graphing work takes place in some virtual drawing boards (numbered from 0 to 15), called "rectangles" (or "rectwindows"). So we create an empty "rectangle" and name it rectangle 2 (any number between 0 and 15 would do):

```
plotinit(2)
plotscale(2, 0,1, 0,1)
```

This creates a rectwindow whose size exactly fits the size of the output device, and makes the coordinate system inside it go from 0 to 1 (for both x and y). Create a rectangular frame along the boundary of this rectangle:

```
plotmove(2, 0,0)
plotbox(2, 1,1)
```

(nothing is output yet, these commands only fills the virtual drawing boards with PARI graphic objects).

We now draw the graph inside a subrectangle of this with upper and left margins of 0.10 (so 10% of the full rectwindow width), and lower and top margins of 0.02, just to make it more interesting. That makes it an 0.88×0.88 subrectangle; so we create another rectangle (call it 3) of linear size 0.88 of the size of the initial rectangle and graph the function in there:

```
plotinit(3, 0.88, 0.88, relative)
plotrecth(3, x = -2, 2, sin(x^7), "Recursive")
```

Finally, output rectangles 2 and 3 on the same plot, with the required offsets (counted from upper-left corner):

```
plotdraw([2, 0,0,  3, 0.1,0.02], relative)
```

The output misses something comparing to the output of `ploth`: there are no coordinates of the corners of the internal rectangle.

However, it is easy to put x- and y-limits on the graph. In the coordinate system of the rectangle 2 the corners are $(0.1, 0.1)$, $(0.1, 0.98)$, $(0.98, 0.1)$, $(0.98, 0.98)$. We can mark lower x-limit by doing

```
plotmove(2, 0.1,0.1)
plotstring(2, "-2.000", left+top+vgap)
```

Computing the minimal and maximal y-coordinates might be trickier, since in principle we do not know the range in advance (though for $\sin(x^7)$ it is easy to guess!). Fortunately, `plotrecth` returns the x- and y-limits.

Here is the complete program:

```
plotinit(3, 0.88, 0.88, relative)
lims = plotrecth(3, x = -2, 2, sin(x^7), "Recursive")
\p 3          \\  3 significant digits for the bounding box are enough
plotinit(2);      plotscale(2, 0,1, 0,1)
plotmove(2, 0,0); plotbox(2, 1,1)
plotmove(2, 0.1,0.1);
```

```
plotstring(2, lims[1], left+top+vgap)
plotstring(2, lims[3], bottom+vgap+right+hgap)
plotmove(2, 0.98,0.1); plotstring(2, lims[2], right+top+vgap)
plotmove(2, 0.1,0.98); plotstring(2, lims[4], right+hgap+top)
plotdraw([2, 0,0,  3, 0.1,0.02], relative)
```

We started with a trivial requirement: have an additional frame around the graph, and it took some effort to do so. But at least it was possible, and PARI did the hardest part: creating the actual graph. Now do a different thing: plot together the "exact" graph of $\zeta(1/2+it)$ together with one obtained from splines approximation. We can emit these graphs into two rectangles, say 0 and 1, then put these two rectangles together on one plot. Or we can emit these graphs into one rectangle 0.

However, a problem arises: note how we introduced a coordinate system in rectangle 2 of the above example, but we did not introduce a coordinate system in rectangle 3. Plotting a graph into rectangle 3 automatically created a coordinate system inside this rectangle. If we use two different methods of graphing, the bounding boxes of the graphs will not be exactly the same, thus outputting the rectangles may be tricky. Thus during the second plotting we ask `plotrecth` to use the coordinate system of the first plotting. Let us add another plotting with fewer points too:

```
plotinit(0, 0.9,0.9, relative)
plotrecth(0, t=100,110, f(t), "Parametric",300)
plotrecth(0, t=100,110, f(t), "Parametric|Splines|Points_too|no_Rescale",30);
plotrecth(0, t=100,110, f(t), "Parametric|Splines|Points_too|no_Rescale",20);
plotdraw([0, 0.05,0.05], relative)
```

This achieves what we wanted: we may compare different ways to plot a graph, but the picture is confusing: which graph is what, and why are there multiple boxes around the graph? At least with some output devices one can control how the output curves look like, so we can use this to distinguish different graphs. And the mystery of multiple boxes is also not hard to solve: they are bounding boxes for calculated points on each graph. We can disable output of bounding boxes with appropriate options for `plotrect`. With these frills the script becomes:

```
plotinit(0, 0.9,0.9, relative)
plotrecth(0, t=100,110, f(t), "Parametric|no_Lines", 300)
opts="Parametric|Splines|Points_too|no_Rescale|no_Frame|no_X_axis|no_Y_axis";
plotrecth(0, t=100,110,f(t), opts, 30);
plotdraw([0, 0.05,0.05], relative)
```

Plotting axes on the second graph would not hurt, but is not needed either, so we omit them. One can see that the discrepancies between the exact graph and one based on 30 points exist, but are pretty small. On the other hand, decreasing the number of points to 20 makes a noticeable difference.

Additionally, one can ask PARI to convert a plot to PS (PostScript) or SVG (Scalable Vector Graphics) format: just use the command `plotexport` instead of `plotdraw` in the above examples (or `plothexport` instead of `ploth`). This returns a character string which you can then write to a file using `write`.

Now suppose we want to join many different small graphs into one picture. We cannot use one rectangle for all the output as we did in the example with $\zeta(1/2+it)$, since the graphs should go into different places. Rectangles are a scarce commodity in PARI, since only 16 of them are user-accessible. Does it mean that we cannot have more than 16 graphs on one picture? Thanks

to an additional operation of PARI plotting engine, there is no such restrictions. This operation is `plotcopy`.

The following script puts 4 different graphs on one plot using 2 rectangles only, `A` and `T`:

```
A = 2;    \\ accumulator
T = 3;    \\ temporary target
plotinit(A);          plotscale(A, 0, 1, 0, 1)

plotinit(T, 0.42, 0.42, relative);
plotrecth(T, x= -5, 5, sin(x), "Recursive")
plotcopy(T, 2, 0.05, 0.05, relative + nw)

plotmove(A, 0.05 + 0.42/2, 1 - 0.05/2)
plotstring(A,"Graph", center + vcenter)

plotinit(T, 0.42, 0.42, relative);
plotrecth(T, x= -5, 5, [sin(x),cos(2*x)], 0)
plotcopy(T, 2, 0.05, 0.05, relative + ne)

plotmove(A, 1 - 0.05 - 0.42/2, 1 - 0.05/2)
plotstring(A,"Multigraph", center + vcenter)

plotinit(T, 0.42, 0.42, relative);
plotrecth(T, x= -5, 5, [sin(3*x), cos(2*x)], "Parametric")
plotcopy(T, 2, 0.05, 0.05, relative + sw)

plotmove(A, 0.05 + 0.42/2, 0.5)
plotstring(A,"Parametric", center + vcenter)

plotinit(T, 0.42, 0.42, relative);
plotrecth(T, x= -5, 5, [sin(x), cos(x), sin(3*x),cos(2*x)], "Parametric")
plotcopy(T, 2, 0.05, 0.05, relative + se)

plotmove(A, 1 - 0.05 - 0.42/2, 0.5)
plotstring(A,"Multiparametric", center + vcenter)

plotmove(A, 0, 0)
plotbox(A, 1, 1)

plotdraw(A)
\\ s = plotexport(A, relative); write("foo.ps", s) \\  if hard copy needed
```

The rectangle `A` plays the role of accumulator, rectangle `T` is used as a target to `plotrecth` only. Immediately after plotting into rectangle `T` the contents is copied to accumulator. Let us explain numbers which appear in this example: we want to create 4 internal rectangles with a gap 0.06 between them, and the outside gap 0.05 (of the size of the plot). This leads to the size 0.42 for each rectangle. We also put captions above each graph, centered in the middle of each gap. There is no need to have a special rectangle for captions: they go into the accumulator too.

To simplify positioning of the rectangles, the above example uses relative "geographic" notation: the last argument of `plotcopy` specifies the corner of the graph (say, northwest) one counts offset from. (Positive offsets go into the rectangle.)

To demonstrate yet another useful plotting function, design a program to plot Taylor polynomials for a $\sin x$ about 0. For simplicity, make the program good for any function, but assume that a function is odd, so only odd-numbered Taylor series about 0 should be plotted. Start with defining some useful shortcuts

```
xlim = 13;  ordlim = 25;  f(x) = sin(x);
default(seriesprecision,ordlim)
farray(t) = vector((ordlim+1)/2, k, truncate( f(1.*t + O(t^(2*k+1)) )));
FARRAY = farray('t);  \\'t to make sure t is a free variable
```

`farray(x)` returns a vector of Taylor polynomials for $f(x)$, which we store in `FARRAY`. We want to plot $f(x)$ into a rectangle, then make the rectangle which is 1.2 times higher, and plot Taylor polynomials into the larger rectangle. Assume that the larger rectangle takes 0.9 of the final plot.

First of all, we need to measure the height of the smaller rectangle:

```
plotinit(3, 0.9, 0.9/1.2, 1);
opts = "Recursive | no_X_axis|no_Y_axis|no_Frame";
lims = plotrecth(3, x= -xlim, xlim, f(x), opts,16);
h = lims[4] - lims[3];
```

Next step is to create a larger rectangle, and plot the Taylor polynomials into the larger rectangle:

```
plotinit(4, 0.9,0.9, relative);
plotscale(4, lims[1], lims[2], lims[3] - h/10, lims[4] + h/10)
plotrecth(4, x = -xlim, xlim, subst(FARRAY,t,x), "no_Rescale");
```

Here comes the central command of this example:

```
plotclip(4)
```

What does it do? The command `plotrecth(..., "no_Rescale")` scales the graphs according to coordinate system in the rectangle, but it does not pay any other attention to the size of the rectangle. Since `xlim` is 13, the Taylor polynomials take very large values in the interval `-xlim...xlim`. In particular, significant part of the graphs is going to be *outside* of the rectangle. `plotclip` removes the parts of the plottings which fall off the rectangle boundary

```
plotinit(2)
plotscale(2, 0.0, 1.0, 0.0, 1.0)
plotmove(2,0.5,0.975)
plotstring(2,"Multiple Taylor Approximations",center+vcenter)
plotdraw([2, 0, 0,  3, 0.05, 0.05 + 0.9/12,  4, 0.05, 0.05], relative)
```

These commands draw a caption, and combine 3 rectangles (one with the caption, one with the graph of the function, and one with graph of Taylor polynomials) together. The plots are not very beautiful with the default colors. See `examples/taylor.gp` for a user function encapsulating the above example, and a colormap generator.

This finishes our survey of PARI plotting functions, but let us add some remarks. First of all, for a typical output device the picture is composed of small colored squares (pixels), as a very large checkerboard. Each output rectangle is a disjoint union of such squares. Each drop of paint in the rectangle will color a whole square in it. Since the rectangle has a coordinate system, it is important to know how this coordinate system is positioned with respect to the boundaries of these squares.

The command `plotscale` describes a range of x and y in the rectangle. The limit values of x and y in the coordinate system are coordinates *of the centers* of corner squares. In particular, if ranges of x and y are $[0, 1]$, then the segment which connects (0,0) with (0,1) goes along the *middle* of the left column of the rectangle. In particular, if we made tiny errors in calculation of endpoints

of this segment, this will not change which squares the segment intersects, thus the resulting picture will be the same. (It is important to know such details since many calculations are approximate.)

Another consideration is that all examples we did in this section were using relative sizes and positions for the rectangles. This is nice, since different output devices will have very similar pictures, while we did not need to care about particular resolution of the output device. On the other hand, using relative positions does not guarantee that the pictures will be similar. Why? Even if two output devices have the same resolution, the picture may be different. The devices may use fonts of different size, or may have a different "unit of length".

The information about the device in PARI is encoded in 6 numbers: resolution, size of a character cell of the font, and unit of length, all separately for horizontal and vertical direction. These sizes are expressed as numbers of pixels. To inspect these numbers one may use the function `plothsizes`. The "units of length" are currently used to calculate right and top gaps near graph rectangle of `ploth`, and gaps for `plotstring`. Left and bottom gaps near graph rectangle are calculate using both units of length, and sizes of character boxes (so that there is enough place to print limits of the graphs).

What does it show? Using relative sizes during plotting produces *approximately* the same plotting on different devices, but does not ensure that the plottings "look the same". Moreover, "looking the same" is not a desirable target, "looking tuned for the environment" will be much better. If you want to produce such fine-tuned plottings, you need to abandon a relative-size model, and do your plottings in pixel units. To do this one removes flag `relative` from the above examples, which will make size and offset arguments interpreted this way. After querying sizes with `plothsizes` one can fine-tune sizes and locations of subrectangles to the details of an arbitrary plotting device.

The last two elements of the array returned by `plothsizes` are the dimensions of the display, if applicable. If there is no real display, like in svg or postscript plots, the width and height of display are set to 0.

To check how good your fine-tuning is, you may test your graphs with a medium-resolution plotting (as many display output devices are), and with a low-resolution plotting (say, with `plot-term("dumb")` of gnuplot).

15. GP Programming.

Do we really need such a section after all we have learnt so far? We now know how to write scripts and feed them to `gp`, in particular how to define functions. It's possible to define *member* function as well, but we trust you to find them in the manual. We have seen most control statements: the missing ones (`while`, `break`, `next`, `return` and various `for` loops) should be straightforward. (You won't forget to look them up in the manual, will you?)

Output is done via variants of the familiar `print` (to screen), `write` (to a file). Input via `read` (from file), `input` (querying user), or `extern` (from an external auxiliary program).

To customize `gp`, e.g. increase the default stack space or load your private script libraries on startup, look up `The preferences file` section in the User's manual. We strongly advise to set `parisizemax` to a large nonzero value, about what you believe your machine can stand: this both limits the amount of memory PARI will use for its computation (thereby keeping your machine usable), and let PARI increases its stack size (up to this limit) to accommodate large computations. If you regularly see `PARI stack overflows` messages, think about this one!

For clarity, it is advisable to declare local variables in user functions (and in fact, with the smallest possible scope), as we have done so far with the keyword `my`. As usual, one is usually better off avoiding global variables altogether.

Break loops are more powerful than we saw: look up `dbg_down` / `dbg_up` (to get a chance to inspect local variables in various scopes) and `dbg_err` (to access all components of an error context).

To reach grandwizard status, you may need to understand the all powerful `install` function, which imports into `gp` an (almost) arbitrary function from the PARI library (and elsewhere too!), or how to use the `gp2c` compiler and its extended types. But both are beyond the scope of the present document.

Have fun!