A Tutorial

for

PARI-GP

C. Batut, K. Belabas, D. Bernardi, H. Cohen, M. Olivier

Laboratoire A2X, U.M.R. 9936 du C.N.R.S.
Université Bordeaux I, 351 Cours de la Libération
33405 TALENCE Cedex, FRANCE
e-mail: `pari@math.u-bordeaux.fr`

Home Page:
`http://pari.home.ml.org/`

Primary `ftp` site:
`ftp://megrez.math.u-bordeaux.fr/pub/pari/`

last updated March 10, 1998
(this document distributed with version 2.0.13)

This booklet is intended to be 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 for detailed explanations. Hence although this chapter can be read independently (for example to get rapidly acquainted with the possibilities of GP without having to read the whole manual), the reader will profit most from it by reading it in conjunction with the reference manual.

1. Greetings!

So you are sitting in front of your workstation (or terminal, or PC,...), and you type `gp` to get the program started (remember to always hit the `Enter` key and not the `Return` key on a Macintosh computer).

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`) key, or the `Enter` key on the Mac. If you do exactly this, you will get the expected answer. However some of you may be used to other systems like Macsyma 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 in GP tells it to keep the result, but not to print it (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`. Yup, seems to be ok. We'll 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 (to name a few: `quit`, `exit`, `system`,...). 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 (see above).

Let's get to more serious stuff. Let's see, 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. This system has been written mainly for pure mathematicians, and not for physicists (although they are welcome to use it). And mathematically, 1/7 is an element of the field $\mathbf{Q}$ of rational numbers, so how else but 1/7 can the computer give the answer to you? (well maybe 2/14, 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 you gave an operation whose result can be represented exactly, that's what it gives you.

OK, 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 (28 on most systems, 38 on the others), and the repeating pattern is 142857. 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 28 / 38 decimals by the way? Well, it is the default initial precision, as indicated when you start GP. This has been chosen so that the computations are very fast, and gives already 12 decimals more accuracy than conventional double precision floating point operations. The precise value depends on a technical reason: if your machine supports 64-bit integers (the standard library can handle integers up to 2^{64}), the default precision will be 38 decimals, and 28 otherwise. As the latter is most probably the case (if your machine is not a `DEC alpha`, that is), we'll assume it henceforth.

Only large mainframes or supercomputers have 28 digits of precision in their standard libraries, and that is their absolute limit. Not here 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 28 digits. Try `log(exp(1))`. Well, it's not exactly equal to 1, but pretty close! That's what you lose by working numerically.

Let's see, 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 some systems, an extended online help might 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 Section 2.10.2 in the User's Manual before you go on: it will be much easier to type in examples and correct typos after you've done that.

Now try `exp(Pi * sqrt(163))`. Hmmm, since from our last example 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 even more nines than before, 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 `sqr(log(%) / Pi)` 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. `sqr` is the square function (`sqr(x) = x * x`), not to be confused with `sqrt` which is the square root function). 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 non-zero 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, even 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, you will notice pretty soon (or maybe you have already?) that many many things changed between the older 1.39.xx versions and this one. And conspicuously, most function names have been changed. We sincerely think it's for the best since they are much more logical now and the systematic prefixing is very convenient coupled with the automatic completion mechanism: it's now very easy to know what functions are available to deal with, say, elliptic curves since they all share the prefix `ell`.

Of course, this is going to break all your nice old scripts. Well, you can either change the compatibility level (typing `default(compatible, 3)` will send you back to the stone-age behaviour of good ol' version 1.39.15), or rewrite the scripts. We really advise you to do the latter if they are not too long, since they can now be written much more cleanly than before (especially with the new control statements: `break`, `next`, `return`), and besides it'll be as good a way as any to get used to the new names. We *might* provide an automatic transcriptor with future versions.

To know how a specific function was changed, just 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 `100!`. It is never necessary to tell GP in advance the size of the integers that it will encounter, since this is adjusted automatically. On the other hand, for many computations with real numbers, it is necessary to specify a default precision (initially 28 digits).

4) Come back to 28 digits of precision (`\p 28`), and type `exp(24 * Pi)`. 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 28 digits of precision, but the integer part of $\exp(24 * \pi)$ has 33 decimal digits. Hence if GP had dutifully printed out 33 digits, the last few digits would have been wrong. Hence GP wants to print only 28 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 to more than 33 decimals. Let's try it. To give it a wide margin, we set the precision to 40 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 28 decimals, and even if you set the precision to 1000 decimals, GP knows that your number has only 28 digits of accuracy but an integral part with 33 digits. So you haven't improved things by increasing the precision. Or have you? What if we retype `exp(24 * Pi)` now that we have 40 digits? Try it. Now we do not have an exponential format, and we see that at 28 decimals the last 6 digits would have been wrong if they had been printed in fixed-point format.

6) *Warning*. Try the following: starting in precision 28, type first `default(format, "e0.50")`, then `exp(24 * Pi)`. Do you understand why the result is so bad, and why there are lots of zeros at the end? Convince yourself by typing `log(exp(1))`. The moral is that the `default(format,)` command changes only the output format, but *not* the default precision. On the other hand, the `\p` command changes both the precision and the output format.

7) What if I forget what the current precision is and I don't feel like counting all the decimals? Well, you can 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.

8) Note that the `default` command reacts differently according to the number of input arguments. This is not an uncommon behaviour 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.

2. Warming up

Another thing you better get used to pretty fast is error messages. Try typing `1/0`. Couldn't be clearer. Taking again our universal example in precision 28, type `floor(exp(24 * Pi))` (`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 I told you not to read them, the explanation is simply that GP is unable to compute the integer part of `exp(24 * Pi)` given only 28 decimals of accuracy, since it has 33 digits.

Some error messages are even much more cryptic than that and are sometimes not so easy to understand (well, it's nothing compared to TEX's error messages!).

For instance, try `log(x)`. Not really clear, is it? But no matter, it simply tells you that GP simply 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 (note that there is always 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 had any experience with other 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 16 terms. 16 is the default `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 (note the `O(x^16)` which ends the series, and which is trademark of this type of object in GP. It is the familiar "big–oh" notation of analysis).

You will 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. But if we try `log(1+x)`, then it works. 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, try `cos(x)`, `cos(x)^2 + sin(x)^2`, `exp(cos(x))`, `exp(exp(x) - 1)`, `gamma(1 + x)`, with different values of `serieslength` if you like.

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 non-zero 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 non-exact.

But I still want that power series. To obtain it, just do as in the 1/7 example: type

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

, etc...

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.

Finally, if you want to be really fancy, you can type `taylor((1 + x)^(-3), x)` (look at the entry for `taylor` for the description of the syntax), but this is in fact almost never used.

To summarize, in this section we have seen that in addition to integers, real numbers and rational numbers, PARI can handle complex numbers, polynomials, power series, rational functions. 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) To be able to duplicate the following example, first type `\y` to suppress automatic simplification.

A complex number has a real part and an imaginary part (who would have guessed?). However, beware that when the imaginary part is the exact integer zero, it is not printed, but the complex number is not converted to a real number. Hence it may *look* like a real number without being one, and this may cause some confusion in programs which expect real numbers. For example, type `n = 3 + I - I`. The answer reads `3`, as expected. But it is still a complex number. Check it with `type(n)`. Hence if you then type `(1+x)^n`, instead of getting the polynomial `1 + 3*x + 3*x^2 + x^3` as expected, you obtain a power series. Worse, when you try to apply an arithmetic function, say the Euler totient function (known as `eulerphi` to GP), you get a sententious error message recalling you that "arithmetic functions want integer arguments". You would have guessed yourself, but the message is difficult to understand since 3 looks like a genuine integer!!! (Please read again if this is not clear. It is a common source of incomprehension).

Similarly, `3 + x - x` is not the integer 3 but a constant polynomial (in the variable `x`), equal to $3 = 3x^0$.

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. In fact, by default GP does this automatically at the end of a GP command. Note that it does *not* simplify in intermediate expressions. This default can be switched off and on by the command `\y`. This is why I asked you to type this before starting.

2) As already stated, power series expansions are always implicitly around $\texttt{x} = 0$. When we wanted them around $\texttt{x} = \texttt{a}$, we replaced `x` by `z + a` in the function we wanted 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).

Now try typing `p = 1 + x + x^2 + O(x^10)`, then `subst(p, x, z+1)`. Do you understand why you get an error message?

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 16 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 inverse function. This is possible only for power series whose first non-zero 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^17)`.

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

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

Do you know how to prove this? (If you have never seen 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 vector does become a column. The `\b` command is used mainly for this purpose.

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, 1)`. Type `default(output, 0)` if you want to come back to the original default.

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 will be 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. There are only a few. We will see later an example of a much more general construction.

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)` (you have to be a little patient, this is quite a complicated computation). The result is a rational number (of course) of numerator equal to 1 and denominator having 226 decimal digits. How do I know, by the way? I did not count! Instead, simply type `1.* d`. The result is now in exponential format, and the exponent gives us the answer. Another, more natural, way would be to simply type `sizedigit(1/d)`.

Now type `hr = 1.* h;` (do not forget the semicolon, we don't want to see all the junk!), then `dr = matdet(hr)`. You notice two things. First the computation, although not instantaneous, is much faster than in the rational case. The reason for this is that PARI is handling real numbers with 28 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 correct, you will see that only 2 decimals agree! This catastrophic instability is as already mentioned one of the characteristics of Hilbert matrices. In fact, the situation is much 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). Again be patient since computing `1/h` will take even more time (not much) than computing `matdet(h)`. The result is larger than 10^{50}, showing that some coefficients of `1/hr` are wrong by as much as 10^{24} (the largest error is in fact equal to $7.644 \cdot 10^{24}$ for the coefficient of row 15 and column 14, which is a 28 digit integer).

To obtain the correct result after rounding for the inverse, we have to use a default precision of 56 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 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.

Warning. The letter `I` is reserved for the complex number equal to the square root of -1. Hence it is absolutely forbidden to use it as a variable. Try typing `vector(10,I, I^2)`, the error message that you get clearly indicates that GP does not consider `I` as a variable. There are two other reserved variable names: `Pi` and `Euler`. All function names are forbidden as well. On the other hand there is nothing special about `i`, `pi` or `euler`.

When creating vectors or matrices, it is often useful to use boolean operators and the `if()` statement (see the section on programming for details on using this 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, x, 0))
```

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

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

if the vector `v` has not been created beforehand (for example by a `v = vector(5,j,0)` command).

The last PARI types which we have not yet played with are types which are closely linked to number theory (hence 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. (Note that PARI does not actually prove that a number is prime: any strong pseudoprime for a number of bases is declared to be prime.) 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. You can set it to 0 to trial divide by the full set of built-in primes, which goes up to 500000 by default).

The result is a 2 column matrix (as for all factoring functions), the first column giving the primes and the second their exponents. Here we get a single row, telling us that `n` is not divisible by any prime up to 200000. We could now trial divide further, or even cheat completely 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 $R = \mathbf{Z}/\mathtt{n}\mathbf{Z}$. The elements of R, called integermods, can always be represented by numbers smaller than `n`, hence very small. Fermat's theorem can be rewritten $\mathtt{a}^{n-1} = \mathtt{Mod(1,n)}$ in the ring R, and this can be computed very efficiently. 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 never a proof).

To find the factors is another story. One must use less naive techniques than trial division (or be very patient). To finish this example, type `factor(n)` to see the factors. Since the smallest factor is 14902357, you would have had to be very patient with trial division! Note that none of the factors in the decomposition are *proven* primes.

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 $\mathtt{n} = 1$) where `p` is the prime and `n` the p-adic precision. 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` (if you want to check, type `s^2 - n`). Type now `l = log(n)`, then `e = exp(l)`. 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 a $(p-1)$-st root of unity.

Incidentally, if you want to get back the integer 569 from the p-adic number `n`, type `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 (always printed as `w`) equal to $(\mathtt{d}+a)/2$ where a is equal to 0 or 1 according to whether `d` is even or odd. The behavior of `quadgen` is a little special: although its result is always printed as `w`, the variable `w` itself is not set to that value. Hence it is necessary to write systematically `w = quadgen(d)` using the variable name `w` (or `w1` etc. if you have several quadratic fields), otherwise things will get confusing.

So type `w = quadgen(-163)`, then `charpoly(w)` which asks for the characteristic polynomial of `w` (in the variable `x`; you can type `charpoly(w, y)` to directly express it in terms of some other variable without resorting to the `subst` function). The result shows what `w` will represent. You can also 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, defined by an integermod). In a sense it is a generalization of the type quadratic number.

The syntax used is the same as for integermods. For example, instead of typing `w = quadgen(-163)`, you can type `w = Mod(x, x^2 - x + 41)`. 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, the basic interest is that you can work in any degree, not only quadratic (of course, even for quadratic moduli, the corresponding elementary operations will be slower than with quadratic numbers).

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 $\mathbf{C}$ (*not* the two square roots). More generally, if `w` was of degree n, we would get an n-component vector, and similarly for 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 much more detail in the number field section.

4. Elementary Arithmetic Functions

Since PARI is aimed at number theorists, it is not surprising that there exists a large number of arithmetic functions (see the list in the corresponding section of the users manual). We have already seen several, such as `factor`. Note that `factor` handles not only integers, but also (univariate) polynomials. Type for example `factor(x^15 - 1)`. You can also ask to factor a polynomial modulo a prime p (`factormod`) and even in a finite field which is not a prime field (`factorff`).

Evidently, you have functions for computing GCD's (`gcd`), extended GCD's (`bezout`), 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; there is no sophisticated primality test.

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 the ring of integers 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 very simple formula for this product: find and prove it). The notation `p *= d` (inherited from the C programming language) 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.)

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

5. Performing Linear Algebra

All the standard linear algebra programs are available of course, and many more. In addition, linear algebra over $\mathbf{Z}$, i.e. work on lattices, can also be performed. Let us see how this works. First of all, a vector space (more generally a module) will be given by a generating set of vectors (often a basis) which will be representend as *column* vectors. This set of vectors will in turn be represented as a matrix: in PARI, we have chosen to consider matrices as row vectors of column vectors. The base field (or ring) can be any ring type PARI supports (except p-adics which are currently not correctly handled by the linear algebra package). However, certain operations are specifically written for a real or complex base field, while others are written for $\mathbf{Z}$ as the base ring.

— TO BE COMPLETED —

6. Using Transcendental Functions

All the elementary transcendental functions and several higher transcendental functions (gamma function, incomplete gamma function, error function, exponential integral, K-bessel functions, confluent hypergeometric functions, Riemann ζ function, polylogarithms, Weber functions, theta functions) are provided. More may 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 will be 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 will be the current real format (usually the same as the default precision). In the present chapter we assume that your machine works with 32-bit long integers. If it is not the case, we leave it to you as a very good exercise to make the necessary modifications.

Let's assume that we have 28 decimals of default precision (this is what we get automatically at the start of a GP session on 32-bit machines). Type `e = exp(1)`. We get the number $e = 2.718\ldots$ to 28 decimals. Let us check how many correct decimals we really have. The hard (but reasonable) way is to proceed as follows. Change the precision to a substantially higher value, for example by typing `\p 50`. Then type `e`, then `exp(1)` once again. This last value is the correct value of the mathematical constant e to 50 decimals, while the variable `e` shows the value that was computed to 28 decimals. Clearly they coincide to exactly 29 significant digits.

A simpler way to see how many decimals we have, is to ask for `length(e)` which indicates we have exactly 3 mantissa words. Since $3\ln(2^{32})/\ln(10) \approx 28.9$ we see that we have 28 or 29 significant digits (on 32-bit machines).

Come back to 28 decimals (`\p 28`). If we type `exp(1.)` you can check that we also obtain 28 decimals. However, type `f = exp(1 + 10.^(-30))`. Although the default precision is still 28, you can check using one of the two methods above that we have in fact 59 significant digits! The reason is that `1 + 10.^(-30)` is computed according to the PARI philosophy, i.e. to the best possible precision. Since `10.` has 29 significant digits and 1 has "infinite" precision, the number `1 + 10.^(-30)` will have $59 = 29 + 30$ significant digits, hence `f` also.

Now type `cos(10.^(-15))`. The result is printed as $1.0000\ldots$, but is of course not exactly equal to 1. Using `length(%)`, we see that the result has 7 mantissa words, giving us the possibility of having 67 correct significant digits. In fact (look in precision 100), only 60 are correct. PARI gives you as much as it can, and since 6 mantissa words would have given you only 57 digits, it uses 7. 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 this quantity, which will be $1 - 0.5 * 10^{-30}$ where $0.5 * 10^{-30}$ is considered with 28 significant digit accuracy, hence the result will have approximately $28 + 30 = 58$ significant digits.

Unfortunately, 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 $1.000\ldots$ to one more mantissa word than the current precision.

OK, now 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 (type `99!` to get the complete answer). To get the complete value, there are two solutions. The first and most natural one 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. After some work, the printed result is this time perfectly correct.

However, this is probably not the best way to proceed. Come back first to the default precision (type `\p 28`). As the gamma function increases very rapidly, one usually uses its logarithm. Type `lngamma(100)`. This time the result has a reasonable size, and is exactly equal to `log(99!)`.

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

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

then `norm(z)`. We see that `norm(z)` 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.)`. Although the result is the same, you will notice that the time is substantially larger (if your machine is too fast to see the difference, increase the precision!). This is because PARI uses special formulas to compute `zeta(n)` when `n` is a (positive or negative) integer.

Type `zeta(1 + I)`. This also works. Now for fun, let us compute in a very naïve 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 = 10^-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 (another, less convenient, way would be to type `\` at the end of each line to tell the parser that the input was not yet finished). By the way, you don't need to type in the suffix `.gp` it will be supplied by GP, if you forget it (the suffix is not mandatory either, it is just more convenient to have all your GP scripts labeled in the same distinctive way).

We thus obtain the first zero to 25 significant digits.

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))`, then `log(a)`. All seems in order. Now type `b = log(5 + O(7^10))`, then `exp(b)`. Is something wrong, since we don't recover the number we started with? Absolutely not. Type `exp(b) * teichmuller(5 + O(7^10))`, and we indeed recover our initial number. The function `teichmuller(x)` is the Teichmüller character, and is characterized by the fact that it is the unique $(p-1)$-st root of unity (here with $p = 7$) 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. Exercise: 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 (not 8 as could be expected).

Now type `?3`. This gives you the list of all transcendental functions. Instead of continuing with more examples, we suggest that you experiment yourself with the list of functions. In each case, 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. We leave for a subsequent section linear algebra and polynomial computations.

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` (just to make it more interesting), then `4 * sumalt(k=0, (-1)^k/(2*k + 1))`. In a split second (admittedly more than simply typing `Pi`), we get π to 100 significant digits (type `Pi` to check). In version 1.38, the method used was a combination of a method due to Euler for accelerating alternating sums, and a programming trick due to the Dutch mathematician van Wijngaarden (see one of the Numerical Recipes books for an explanation). The method which we presently use is considerably better, and is based on a combination of ideas of F. Villegas, D. Zagier and H. Cohen.

Similarly, type `\p 28` (otherwise the time will be too long) and `sumpos(k=1, 1 / k^2)`. Although once again the convergence is slow, a similar trick allows to compute the sum when the terms are positive (compare with the exact result `Pi^2/6`). This is much less impressive because it is much slower, but is 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 cvonverges, albeit slowly (sumalt doesn't care however). For negative `s`, the series diverges, but `zet(s)` still gives the correct result! 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(-14.5)`, 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. For example, type `intnum(t=1,2, 1/t)` and presto! you get 26 decimals of $\log(2)$. Look at Chapter 3 to see the different integration functions which are available.

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. Hence type

```
{
  fun(z, u)=
  u = exp(z);
  (u-1) / (u-z)
}
zero(r) = r/(2*Pi) * intnum(t=0, 2*Pi, fun(r*exp(I*t)) * exp(I*t))
```

(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 to 0. It will then restore their former values upon exit. If, however, we had not declared `u` in the function prototype, it would be considered as a global variable, whose value would be permanently changed.)

The function `zero(r)` will then count the number of zeros (we have simply made the change of variable $z = r * \exp(i * t)$).

Now type `\p 9` (otherwise the computation would take too long, and anyway we know that the result is an integer), 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) whose modulus is between 1 and 1.5. For the sake of completeness, let us compute them. Let z be such a zero, and write $z = x + iy$ 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, u)=
  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 28 and type

```
x0 = solve(x=0,1, fun(x))
```

This quickly gives us the value of `x`, and we can then type

```
z = x0 + I*sqrt(exp(2*x0) - x0^2)
```

which (together with its complex conjugate) is the required zero. As a check, type `exp(z) - z`, and also `abs(z)`.

Of course you can integrate over contours which are more complicated than circles, but you must perform yourself the changes of variable 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 would not work because the real part is positive for `t=14` and `t=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. Functions Related to Polynomials and Power Series

First a word of warning to the unwary: it is essential to understand the crucial difference between exact and inexact objects (see Section 1). This is especially important in the case of polynomials. Let's immediately take a plunge into these problems. Type

```
gcd(x^2 - 1, x^2 - 3*x + 2)
```

No problem, the result is `x - 1` as expected. But now type

```
gcd(x^2 - 1., x^2 - 3.*x + 2.)
```

You are lucky, the result is almost correct except for a bizarre factor of 3 which comes from the way PARI does the computation. In any case, it is still essentially a reasonable result. But now type `gcd(x - Pi, x^2 - 6*zeta(2))`. Although this should be equal to `x - Pi`, PARI finds a constant as a result. This is because the notion of GCD of non-exact polynomials doesn't make much sense. However, type `polresultant(x - Pi, x^2 - 6*zeta(2))`. The result is extremely close to zero, showing that indeed the GCD is non-trivial, without telling us what it is. This being said, we will usually use polynomials (and power series) with exact coefficients in our examples.

Set `pol = polcyclo(15)`. This gives the 15-th cyclotomic polynomial, which is of degree $\varphi(15) = 8$. Now, type `r = polroots(pol)`. You have the 8 complex roots of pol given to 28 significant digits. To see them better, type `\b`. As you see, they are given as pairs of complex conjugate roots, in a random order. The only ordering done by the function `polroots` concerns the real roots, which are given first, and in increasing order.

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! Well, fair enough, vectors cannot be multiplied (even less raised to a power) that easily. However, type `rc = r^15.` with a . at the end. Now it works, because powering to a non-integer (here real) exponent is a transcendental function and hence is applied termwise. Note that the fact that 15. 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(real(rc))`, then `sqrt(norml2(rc-rr))`. We see that `rr` is indeed all 1's, and that the L2-norm of `rc - rr` is around 10^{-27}, reasonable enough when we work with 28 significant digits. Note that the function `norml2`, contrary to what it may seem, does not give the L2 norm but its square, hence we must take the square root (well, this is not absolutely necessary in this case!).

Now type `pol = x^5 + x^4 + 2*x^3 - 2*x^2 - 4*x - 3`, then `factor(pol)`. The polynomial `pol` factors over $\mathbf{Q}$ (or $\mathbf{Z}$) as a product of two factors. Type `fun(p)= factorpadic(pol,p,10)`. This creates a function `fun(p)` which factors `pol` over $\mathbf{Q}_p$ to p-adic precision 10. If now we type `factor(poldisc(pol))`, we learn that the primes dividing the discriminant are 11, 23 and 37. Type `fun(5)`, `fun(11)`, `fun(23)`, and `fun(37)` to see different splittings.

Similarly, we can type `lf(p)= lift(factormod(pol,p))`, and `lf(2)`, `lf(11)`, `lf(23)` and `lf(37)` which show the different factorizations, this time over $\mathbf{F}_p$. In fact, even better: type successively

```
pol2 = x^3 + x^2 + x - 1
fq = factorff(pol2, 3, t^3 + t^2 + t - 1)
centerlift(lift(fq))
```

This factors the polynomial `pol2` over the finite field $\mathbf{F}_3(\theta)$ where θ is a root of the polynomial `t^3 + t^2 + t - 1`. This is of course a form of the field $\mathbf{F}_{27}$. We know that $\mathrm{Gal}(\mathbf{F}_{27}/\mathbf{F}_3)$ is cyclic of order 3 generated by the Frobenius homomorphism $u \mapsto u^3$, and the roots that we have found give the action of the powers of the Frobenius on `t` (if you do not know what I am talking about, please try some more examples, it's not so hard to figure out).

Similarly, type `pol3 = x^4 - 4*x^2 + 16` and `fn = factornf(pol3,t^2 + 1)`, and we get the factorization of the polynomial `pol3` over the number field defined by `t^2 + 1`, i.e. over $\mathbf{Q}(i)$. To see the result even better, type `lift(fn)`, remembering that `t` stands for the generator of the number field (here equal to $i = \sqrt{-1}$).

Note that it is possible, although ill advised, to use the same variable for the polynomial and the number field. You may for example type `fn2 = factornf(pol3, x^2 + 1)`, and the result is correct. However, the PARI object thus created may give unreasonable results. For example, if you type `lift(fn2)` in the example above, you will get a strange object, with a symbol such as `x*x` typed. This is because PARI knows that the dummy variable `x` is not the same as the explicit variable `x`, but since it must print it when you lift, it has to do something.

To summarize, in addition to being able to factor integers, you can factor polynomials over $\mathbf{C}$ and $\mathbf{R}$ (this is the function `polroots`), over $\mathbf{F}_p$ (the function `factormod`, over $\mathbf{F}_{p^k}$ (the function `factorff`), over $\mathbf{Q}_p$ (the function `factorpadic`), over $\mathbf{Q}$ or $\mathbf{Z}$ (the function `factor`), and over number fields (the functions `factornf` and `nffactor`). Note however that `factor` itself will try to guess intelligently over which ring you want to factor: set `pol = x^2+1` and try to `factor` successively `pol`, `pol * 1.`, `pol * (1+0.*I)`, `pol * Mod(1,2)`, `pol * Mod(1,Mod(1,3)*(t^2+1))`.

In the present version 2.0.13, it is *not* possible to factor over other rings than the ones mentioned above, for example GP cannot factor multivariate polynomials.

Other functions related to factoring are `padicappr`, `polrootsmod`, `polrootspadic`, `polsturm`. Play with them a little.

Now let's type `polsym(pol3,20)`, where `pol3` is the same polynomial as above. This gives the sum of the k-th powers of the roots of `pol3` up to $k = 20$, of course computed using Newton's formula and not using `polroots`. You notice that every odd sum is zero (this is trivial since the polynomial is even), but also that the signs follow a regular pattern and that the (non-zero) 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 (non-rational) algebraic numbers.

You can also try the function `polkaramul` (for Karatsuba multiplication). Put the timer on (`#`; well, if you've been very dedicated and followed this tutorial all the way from the beginning, the timer should already be on), and then set `pol4 = (x + 1)^200;` (do not forget the semicolon, you don't want to see the result, just the timing). Similarly for the next few commands. Then type `pol5 = pol4 * pol4;`, and look at the timing. Now type instead `pol6 = polkaramul(pol4,pol4,5);`. This should be about twice as fast (it is platform dependent so the gain may vary). Try with other values than 5. Also type `pol5 - pol6` just to be sure that the result is correct. So if you have polynomials which are that large, it may be worthwhile to use this function.

By the way, try `pol4^2;`. Surprised? Well, if you write down the multiplication rules for polynomials, you'll notice that it's easier to compute a square than a general product (you can save about half of the computations). This is of course taken into account in the powering operation, but not when doing a multiplication, since GP does not check whether the two operands are equal.

Now let's play a little with power series. We have already done so a little at the beginning. Type

```
8*x + prod(n=1,39, if(n%4, 1 - x^n, 1), 1 + O(x^40))^8
```

You obtain a power series which has apparently only even powers of `x` appearing. This is surprising, but can be proved using the theory of modular forms. Note that we have initialized the product to `1 + O(x^40)` and not simply to 1 otherwise the whole computation would have been done with polynomials, and this would first have been slightly slower and also totally useless since the coefficients of `x^40` and above are irrelevant anyhow if we stop the product at `n=39`.

While we are on the subject of modular forms (which, together with taylor series expansions of common functions, are another great source of power series), type `\ps 122` (which is a shortcut for `default(seriesprecision, 122)`), then `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 giving 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)` and now this works like a charm.

The pattern in the result is clear. Of course, it now remains to prove it (again modular forms, see Antwerp III or your resident modular forms guru). Similarly, type `centerlift(Mod(1,3) * d)`. This time the pattern is less clear, but nonetheless there is one. Refer to Anwerp III again.

9. 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` (try to guess what we'll use for number fields...).

So type `e = ellinit([6,-3,9,-16,-14])`. This computes a number of things about the elliptic curve defined by the affine equation

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

It's 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 much worse), you are advised to use the 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, there is no `clgpinit` function).

Let's try it. We see that the discriminant `e.disc` is equal to 37, hence the conductor of the curve is 37. Of course in general it is not so trivial. In fact, the equation of the curve is clearly not minimal, so type `r = ellglobalred(e)`. The first component `r[1]` tells us that the conductor is 37 as we already knew. The second component is a 4-component vector which will allow us to get the minimal equation: simply type `e = ellchangecurve(e, r[2])` and the new `e` is now our minimal equation with corresponding data. You can for the moment ignore the third component `r[3]` (for the impatient reader, this is the product of the local Tamagawa numbers, c_p).

The new `e` tells us that the minimal equation is $y^2 + y = x^3 - x$. Let us now play a little with points on `e`. Type `q = [0,0]`, which is clearly on the curve (type `ellisoncurve(e, q)` to check). Well, `q` may be a torsion point. Type `ellheight(e, q)`, which computes the canonical Neron-Tate height of `q`. This is non-zero, hence `q` is not torsion. To see this even better, type

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

and we see the characteristic parabolic explosion of the size of the points. As a further check, type `ellheight(e, ellpow(e, q,20)) / ellheight(e, q)`. We indeed find $400 = 20^2$ as it should be. You can also type `ellorder(e, q)` which returns 0, telling you that `q` is non-torsion.

Notice how 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])`. This corresponds to the equation $y^2 + y = x^3 - x^2$. Again from the discriminant we see that the conductor is equal to 11, and if you type `ellglobalred(e)` you will see that the equation for `e` is minimal. Type `q = [0,0]` which is clearly a point on `e`, and `ellheight(e, q)`. This time we obtain a value which is very close to zero, hence `q` must be a torsion point. Indeed, typing `for(k=1,5, print(ellpow(e, q,k)))` we see that `q` is a point of order 5 (note that the point at infinity is represented as `[0]`). More simply, you can type `ellorder(e, q)`.

Let's try still another curve. Type `e = ellinit([0,0,1,-7,6])` to get the curve $y^2 + y = x^3 - 7x + 6$. Typing `ellglobalred(e)` shows that this is a minimal equation and that the conductor, equal to the discriminant, is 5077. There are some trivial integral points on this curve, but let's try to be more systematic.

First, let's study the torsion points. Typing `elltors(e)` shows that the torsion subgroup is trivial, so we won't have to worry about torsion points. Next, the member `e.roots` gives us the 3 roots of the minimal equation over $\mathbf{C}$, i.e. $Y^2 = X^3 - 7X + 25/4$ (set $(X, Y) = (x, y + 1/2)$) so if (x, y) is a real point on the curve, x must be at least equal to the smallest root, i.e. $x \geq -3$. Finally, if (x, y) is on the curve, its opposite is clearly $(x, -y - 1)$. So we are going to use the `ellordinate` function and type (for instance in `points.gp` which you can read in with `\r points` as we saw before)

```
{
  v=[];
  for (x = -3, 1000,
    s = ellordinate(e,x);
    if (length(s),      \\ we could use if (!s, instead
```

```
      v = concat(v, [[x,s[1]]])
    )
  ); v
}
```

By the way, this is how you insert a comment 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 file `points.gp` you could insert the following:

```
/*  Finds rational points on the elliptic curve e, using the naivest
    algorithm I could think of right now (TO BE IMPROVED).
    e should have rational coefficients.
    TODO: Make that into a usable function.
 */
```

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

We thus get a large number (18) of integral points. Together with their opposites and the point at infinity, this makes a total of 37 integral points, which is large for a curve having such a small conductor. So we suspect (if we don't know already, since this curve is quite famous!) that the rank of this curve must be high. Let's try and put some order into this (note that we work only with the integral points, but in general rational points should also be considered).

Type `hv = ellheight(e, v)`. This gives the vector of canonical heights. Let us order the points according to their height. For this, type

`iv = vecindexsort(hv)`, then `hv = vecextract(hv,iv)` and `v = vecextract(v,iv)`.

It seems reasonable to take the numbers with smallest height as generators of the Mordell-Weil group. Let's try the first 4: type

`m = ellheightmatrix(e, vecextract(v,[1,2,3,4])); 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 non-trivial kernel, and the coefficients are (close to) integers as they should. Typing `elladd(e, v[1],v[3])` does indeed show that it is equal to `v[4]`.

Taking any other four points, we would in fact always find a dependency. Let's find them all. Type `vp = [v[1],v[2],v[3]]~; m = ellheightmatrix(e,vp); matdet(m)`. This is now clearly non-zero so the first 3 points are linearly independent, showing that the rank of the curve is at least equal to 3 (it is in fact equal to 3, and `e` is the curve of smallest conductor having rank 3). We would like to see whether the other points are dependent. For this, we use the function `ellbil`. Indeed, if `Q` is some point which is dependent on `v[1]`,`v[2]` and `v[3]`, then `matsolve(m, ellbil(e, vp,Q))` will by definition give the coefficients of the dependence relation. If these coefficients are close to integers, then there is a dependency, otherwise not. This is much safer than using the `matker` function. Thus, type

```
w = vector(18,k, matsolve(m, ellbil(e, vp,v[k])))
```

We "see" that the coefficients are all very close to integers, and we can prove it by typing

```
wr = round(w); sqrt(norml2(w - wr))
```

which gives an upper bound on the maximum distance to an integer. Thus `wr` is the vector expressing all the components of `v` on its first 3. We are thus led to strongly believe that the curve has rank exactly 3, and this can be proved to be the case.

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

```
v1 = [1,0]; v2 = [2,0];
z1 = ellpointtoz(e, v1)
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 of course 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, but also follows from Wiles' result since `e` is semi-stable). In fact, to find the modular parametrization of the curve, type `modul = elltaniyama(e)`, then `u=modul[1]; v=modul[2];`. Type

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

to see that this indeed parametrizes the curve.

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 $\Gamma_0(5077)$.

Finally, let us come back to the curve defined by typing `e = ellinit([0,-1,1,0,0])` where we had seen that the point `q = [0,0]` was of order 5. We know that the conductor of this curve is equal to 11 (type `ellglobalred(e)`). We want the sign of the functional equation. Type

`elllseries(e, 1,-11,1.1)`, then `elllseries(e, 1,-11,1)`.

Since the values are clearly different, the sign cannot be $-$. In fact there is an algebraic algorithm which would allow to compute this sign, but it has not yet been completely written, although in case of conductors prime to 6 it is very simple.

Now type `ls = elllseries(e, 1,11,1)`, and just as a check `elllseries(e, 1,11,1.1)`. The values agree (approximately) as they should, and give the value of the L-function of `e` at 1. Now according to the Birch and Swinnerton-Dyer conjecture (which is proved for this curve), `ls` is given by the following formula (in this case):

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

where Ω is the real period of E, c is the global Tamagawa number, product of the local c_p for primes p dividing the conductor, $|\text{III}|$ 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]` (if there had been 3 real roots instead of 1 in `e.roots`, Ω would be equal to `2 * e.omega[1]`). The Tamagawa number c is given as the last component of `ellglobalred(e)`, and here is equal to 1. We already know that the torsion subgroup of E contains a point of order 5, and typing `torsell(e)` shows that it is of order exactly 5. Hence type `ls * 25/e[15]`. This shows that $|\text{III}|$ must be equal to 1.

10. Working in Quadratic Number Fields

The simplest of all number fields outside $\mathbf{Q}$ are quadratic fields. Such fields are characterized by their discriminant, and even better, any non-square 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 by using the command `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 should start by the command `w = quadgen(`D`)`.

This sets `w` equal to ω as above, and is printed `w`. Note however that if several different quadratic orders are used, a printed `w` may have several different meanings. For example if you type

```
w1 = quadgen(-23); w2 = quadgen(-15);
```

Then ask for the value of `w1` and `w2`, both will be printed as `w`, but of course they are not equal. Hence beware when dealing with several quadratic orders at once.

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, and this has been chosen in PARI. For negative discriminants, 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` (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. Let's check this. Type `f^77`. We get a form starting with 1, i.e. the identity, so this test is OK. 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)` (here 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)
```

Note that this function cheats a little and could still give a wrong answer, even assuming GRH (we could get a subgroup and not the whole class group). If we want to use proven bounds under GRH, we have to type

```
quadclassunit(-3299,,[1,6])
```

The double comma `,,` is not a typo, it means we omit an optional second argument (we would use it to compute the narrow class group, which would be the same here of course). As we want to use the optional *third* argument, we have to indicate to GP we skipped this one.

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 much more complicated results), we could *certify* this result (remove the GRH assumption). Let's do it, type

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

A non-zero 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 some 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 (as is clear if one considers ideles instead of ideals), hence in fact quadratic forms with positive discriminant will be represented as a quadruplet (a, b, c, d) where the quadratic form itself is $ax^2 + bxy + cy^2$ with a, b and c integral, and d is the Archimedean component, a real number. For people familiar with the notion, d represents a "distance" as defined by Shanks and Lenstra.

To create such forms, one uses the same function as for definite ones, but you can add a fourth (optional) argument to initialize the distance:

`Qfb(`a`, `b`, `c`, `d`)`

If the discriminant of (a, b, c) is negative, d is silently discarded. If you omit it, this component is set to `0.` (i.e. a real zero to the current precision).

Again these forms can be multiplied, divided, powered, and they can be reduced using the function `qfbred`. This function is in fact 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 using this function. Since handling the fourth component d usually involves computing logarithms, the same flag may be used to ignore the fourth component. Finally, it is sometimes useful to operate on forms of positive discriminant without performing any reduction (this is useless in the negative case), the functions `qfbcompraw` and `qfbpowraw` do exactly that.

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 theorem). Let us create a form by typing `f = qfbred(qfbprimeform(d,2), 2)` (the last 2 tells `qfbred` to ignore the archimedean component). 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; for(i=1,20, g = qfbred(g, 3); print(g))
```

(this time the 3 means to do a single reduction step, still not using Shanks's distance). We see that we come back to the form `f` without having 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 will be given by the fourth component of the form. Reduce your form until you reach the unit form (you will have to type `qfbred(%, 1)` exactly 6 times).

Extract the Archimedean component by typing `c = component(%, 4)`. By definition of this distance, we know that

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

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

```
a = sqrt(8 * exp(2*c))
```

and then `sa = 8 / 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 is the characteristic polynomial of α. We can check that the discriminant of this polynomial is a square multiple of `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 explicitly, check that it is indeed a generator of the ideal corresponding to the form `f3`.

Let us now play a little with cycles. Set `D = 10^7 + 1`, then type

```
quadclassunit(D,,[1,6])
```

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 is approximately equal to 2641.5. If you want to certify this, use `qfbclassno` and `quadregulator`, *not* `bnfinit` and `bnfcertify`, which will take an absurdly long time (well, about 5 minutes if you are careful and set the initial precision correctly). Indeed `bnfcertify` needs the fundamental unit which is so large that `bnfinit` will have a (relatively) hard time computing it: you will need about $R/\log(10) \approx 1147$ digits of precision! On the other hand, you can try `quadunit(D)`. Impressive, isn't it? (you can check that its logarithm is indeed equal to the regulator).

Now just as an example, let's assume that we want the regulator to 500 decimals, say (without cheating and computing the fundamental unit exactly first!). I claim that by simply knowing the crude approximation above, this can be computed with no effort.

This time, we want to start with the unit form. Since `D` is odd, we can type:

```
u = qfbred(Qfb(1,1,(1 - D)/4), 2)
```

We use the function `qfbred` with no distance since we want the initial distance to be equal to 0.

Now we 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 of `f` is around 4.253. Very crudely, since we want a distance of 2641.5, this should be encountered approximately at $2641.5/4.253 = 621$ times the distance of `f`. Hence, as a first try, we type `f^621`. Oops, we overshot, since the distance is now 3173.02. Now we can refine our initial estimate and believe that we should be close to the correct distance if we raise `f` to the power $621 * 2641.5/3173$ which is close to 517. Now if we compute `f^517` we hit the principal form right on the dot. Note that this is not a lucky accident: we will always land extremely close to the correct target using this method, and usually at most one reduction correction step is necessary. Of course, only 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 absolutely 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. However, when PARI makes its computations, 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.

Hence we now start again by setting the precision to (for example) 500, we retype the definition of `u` (why is this necessary?), and then `f = qfbred(u, 1)` and finally `f^517`. Of course we know in advance that we land on the unit form, and the fourth 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 five minutes (time for a coffee break?).

11. Working in General Number Fields

Note for the present release: this section is a little obsolete since many new powerful functions are available now. This needs to be rewritten entirely.

The situation here is of course more difficult. First of all, remembering what we did with elliptic curves, we need to initialize it (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 quite a complicated object but, thanks to member functions, we don't need to know anything about its internal structure (which is dutifully described in Chapter 3). 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 is, well, type `sqrt(% / nf.disc)`. That's 3087, we don't want to work with such a badly skewed polynomial!

So, type `P = polred(T)`. We see from the third component that the polynomial $x^4 - x^3 - 21x^2 + 17x + 133$ defines the same field with much smaller coefficients, so type `A = P[3]`. The `polred` function gives a (usually) simpler polynomial, and also sometimes some information on the existence of subfields. For example in this case, the second component of `polred` tells us that the field defined by $x^2 - x + 1 = 0$, i.e. the field generated by the cube roots of unity, is a subfield of our number field K. Note this is given as incidental information and that the list of subfields found in this way is usually far from complete. To get the complete list, you will have to use the function `nfsubfields` (we'll do that later on).

Type `poldisc(A)`, this is much better, but maybe not optimal yet (the index is still 7). Type `polredabs(A)` (the `abs` stands for absolute). Since it seems that we won't get anything better, we'll stick with `A` (note however that `polredabs` finds a smallest generating polynomial with respect to a bizarre norm which ensures that the index will be small, but not necessarily minimal). 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 components 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 a much better one, which happens to be `A` (small wonder, `nfinit` internally called `polredabs`!). 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 the one that `polred` has found, namely `A`. These are algebraic numbers, and as already mentioned are represented as polmods. For example, in our special case θ is equal to the polmod

```
Mod(x, x^4 + 24*x^2 + 585*x + 1791)
```

while α is equal to the polmod

```
Mod(x, x^4 - x^3 - 21*x^2 + 17*x + 133)
```

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

Now probably 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 `NF[2]`, you get

```
Mod(x^2 + x - 11, x^4 - x^3 - 21*x^2 + 17*x + 133)
```

meaning that $\theta = \alpha^2 + \alpha - 11$, 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 to do the reverse, i.e. go back from a representation in α to a representation in θ, we use the function `modreverse` on this polynomial `NF[2]`. Try it. The result has a big denominator (147) essentially because our initial polynomial `T` was so bad. By the way to get the 147, you should type `denominator(content(NF[2]))`. 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 "surprising" behaviour is that we think of a polynomial as a special case of a rational function.

From now on, we completely 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 essentially 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 a very 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, ideles, congruence subgroups, etc.

Ok, 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` $= \frac{1}{7}\alpha^3 - \frac{5}{7}\alpha^2 - \frac{8}{7}\alpha + 8$, 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 in fact an 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. There is no `nfeltnorm` function (`nfelttrace` does not exist either), but we can easily write one:

```
{
  nfeltnorm(nf,u, t)=
  t = type(u);
  if (t != "t_POLMOD",
    if (t == "t_COL",
      u = nfbasistoalg(nf, u)
    ,
      u = Mod(u, nf.pol)
    )
  );
  norm(u)
}
```

Notice that this is certainly not foolproof (try it with complex or quadratic arguments!), but we could refine it if the need arose. In fact there was no need for this function, since you can consider (`u`) as a principal ideal, and just type `idealnorm(nf,u)` whatever the chosen representation for `u`. We'll talk about ideals later on.

If we want all the symmetric functions of `u` and not only the norm, we type `charpoly(u)` (we could write `charpoly(u, y)` to tell GP to use the variable `y` for the characteristic polynomial). Note that this gives the characteristic polynomial of `u`, and not in general the minimal polynomial. Exercises: how does one (easily) find the minimal polynomial from this? Find a simpler expression for `u`.

OK, 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 $D = 18981$ and $(1, \alpha, \alpha^2, \frac{1}{7}\alpha^3 + \frac{2}{7}\alpha^2 + \frac{6}{7}\alpha)$ is an integral basis (`nf.zk`). The Galois group of its Galois closure can be obtained by typing `polgalois(A)`. The answer ($[8, -1, 1]$) 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 non-trivial field isomorphism which fixes K (exactly one in this case). To find out which it is, we use the function `nfgaloisconj`. This uses the LLL algorithm to find linear relations. So type `nfgaloisconj(nf)`. The result tells us that, apart from the trivial automorphism, the map $\alpha \mapsto (-\alpha^3 + 5\alpha^2 + \alpha - 49)/7$ (in the third component) is a field automorphism. Indeed, if we type `s = Mod(%[3], A); charpoly(s)`, we obtain the polynomial `A` once again.

The fixed field of this automorphism is going to be the only non-trivial 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 `Q = x^2 + 22*x + 133` and I don't recognize it. Now `polred(Q)` proves that this subfield is indeed the field generated by a cube root of unity. Let's check that `s` is of order 2: `subst(lift(s), x, s)`. Yup, it is. Let's express it as a matrix:

```
{
  v = [;]; b = nf.zk;
  for (i=1, 4,
    v = concat(v, nfalgtobasis(nf, nfgaloisapply(nf, s, b[i])))
  )
}
```

`v` gives the action of `s` on the integral basis. Let's check `v^2`. That's the identity all right. `k = matker(v-1)` is indeed two-dimensional, and `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!).

As a final check, type `nfrootsof1(nf)`. Again we find that K contains a cube root of unity, since the torsion subgroup of its unit group is of order 6. And the given generator happens to be equal to `u` (so if you did not do the above exercise as you should have, you now know the answer anyway).

Additional comment (you're not supposed to skip this anymore, 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` is to compute an integral basis, i.e. a $\mathbf{Z}$-basis of the maximal order $\mathbf{Z}_K$ of K. For a large polynomial, this implies factoring the discriminant of the polynomial, which is very often out of the question. There are two ways in which the situation may be improved:

1) First, it is often the case that the polynomial that one considers is of quite a special type, giving some information on the discriminant. For example, one may know in advance that the discriminant is a square. Hence we can "help" PARI by giving it that information. More precisely, using the extra information that we have we may be able to factor the discriminant of the polynomial. We can then use the function `addprimes` to inform PARI of this factorization. Namely, add the primes which are known to divide the discriminant; this will save PARI some work. Do it

only for big primes (bigger than `primelimit`, whose value you can get using `default`) — it will be useless otherwise.

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 the 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 you can provide a factorization of the discriminant as an optional third argument. And here we can cheat, and give on purpose an incomplete factorization involving only the primes we want. 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, ,[7,6; 1537461,1])
```

and the factors of 1537461 will not be looked at! (of course in this example it would be stupid to cheat, but if the discriminant has 2000 digits, this can be a handy trick).

We now would like to work with ideals (and even with ideles) 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 of a $\mathbf{Z}$-basis of the ideal expressed on the integral basis. Standard means that most ideal related functions will use this representation for their output. Note that, as mentioned before, we always represent elements on the integral basis and not on a power basis.

Prime ideals can be represented in a special form as well (see the description of `idealprimedec` in Chapter 3) 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 completely 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 `ideal-hnf(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 Big Number Field (or `bnf`) for that (in particular, we need the class group and fundamental units). More on this later.

An "idele" will be represented as a 2-element vector, the first element being the corresponding ideal (in any valid form), which summarizes the non-archimedean information, and the second element a vector of real and complex numbers with $r_1 + r_2$ components, the first r_1 being real, the remaining ones complex. In fact, to avoid certain ambiguities, the first r_1 components are allowed to have an imaginary part which is a multiple of π. These $r_1 + r_2$ components correspond to the Archimedean places of the number field K.

Let us keep our number field K as above, and hence the vector `nf`. Type

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

This gives the decomposition of the prime number 7 into prime ideals. We have chosen 7 because it is the index of $\mathbf{Z}[\theta]$ in $\mathbf{Z}_K$, hence is the most difficult case to treat. The index is given in `nf[4]` and cannot be accessed through member functions since it's rarely needed. It is in any case trivial to compute as `sqrtint(poldisc(nf.pol) / nf.disc)`.

The result is a vector with 4 components, showing that 7 is totally split in the field K into four 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 would like to have the Hermite Normal Form of this ideal. No problem: since ideal multiplication always gives the result in HNF, we will simply multiply our prime ideal by $\mathbf{Z}_K$, which is represented by the identity matrix. So type

```
idealmul(nf, matid(4), pr)
```

or in fact simply

```
idealmul(nf, 1, pr)
```

or even simpler yet

```
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=1,4, 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 "idelize" `pr3` by typing `id3 = [pr3, [0,0]]`. (We need $r_1+r_2 = 2$ components for the second vector.) Then type `r1 = idealred(nf, id3)`. We get a new ideal which is equivalent to the first (modulo the principal ideals). The Archimedean component is non-trivial and gives the distance from the reduced ideal to the original one (see H. Cohen *A Course in Computational Algebraic Number Theory*, GTM **138** for details, especially Sections 5.8.4 and 6.5). Now, just for fun type

```
r = r1; for(i=1,3, r = idealred(nf,r,[1,5]); print(r))
```

We see that the third `r` is equal to the initial `r1`. This means that we have found a unit in our field, and it is easy to extract this unit given the Archimedean information (clearly it would be impossible without). We first form the difference of the Archimedean contributions of `r` and `r1` by typing

```
arch = r[2] - r1[2]; l1 = arch[1]; l2 = arch[2];
```

From this, we obtain the logarithmic embedding of the unit by typing

```
l = real(l1 + l2) / 4; v1 = [l1,l2,conj(l1),conj(l2)]~/ 2 - [l,l,l,l]~;
```

This 4-component vector contains by definition the logarithms of the four complex embeddings of the unit. Since the matrix `nf[5][1]` contains the values of the r_1+r_2 embeddings of the elements of the integral basis, we can obtain the representation of the unit on the integral basis by typing

```
{
  m1 = nf[5][1];
  m = matrix(4,4,j,k,
    if (j<=2,
      m1[j,k]
    ,
      conj(m1[j-2, k])
```

```
    )
  );
  v = exp(v1);
  au = matsolve(m,v);
  vu = round(real(au))
}
```

Then `vu` is the representation of the unit on the integral basis. The closeness of the approximation of `au` to `vu` gives us confidence that we have made no numerical mistake. To be sure that `vu` represents a unit, type `u = nfbasistoalg(nf,vu)`, then typing `norm(u)` we see that it is equal to 1 so `u` is a unit.

There is of course no reason for `u` to be a fundamental unit. Let us see if it is a square. Type `f1 = factor(subst(charpoly(u,x), x, x^2))`. We see that the characteristic polynomial of `u` where `x` is replaced by `x^2` is a product of 2 polynomials of degree 4, hence `u` is a square (Exercise: why?).

We now want to find the square root of `u`. We can again use the `matsolve` function as above. For this we need to take the square root of each element of the vector `v`, and hence there are sign ambiguities. Let's do it anyway. Type `v = sqrt(v)`. We see that `v[1]` and `v[3]` are conjugates, as well as `v[2]` and `v[4]`, so for the moment the signs seem OK. Now try `au = matsolve(m,v)`. The numbers obtained are clearly not integers, hence the last remaining sign change must be performed. Type `v[1] = -v[1]; v[3] = -v[3]` (they must stay conjugate) and then again `au = matsolve(m,v)`. This time the components are close to integers, so we are done (after typing `vu = round(real(au))` as before).

Anyway, we find that a square root `u2` of `-u` is represented by the vector `vu=[-4,1,1,-1]~` on the integral basis, and this is in fact a fundamental unit.

The function `polred` gives us another method to find `u2` as follows: type `q = polred(f1[1,1], 2)`. We recognize the polynomial `A` as the component `q[3,2]`. To obtain the square root of our unit we then simply type `up2 = modreverse(Mod(q[3,1], f1[1,1]))` (Exercise: why?). We find that `up2` is represented by the vector `[-3,-1,0,0]~` on the integral basis, which is not the result that we have found before nor its opposite. Where is the error? (Please think about this before reading on. There is a mathematical subtlety hidden here.)

Have you solved the problem? Good! The problem occurs because as mentioned before (but you may not have noticed since it is not stressed in standard textbooks) although the number field K is not Galois over $\mathbf{Q}$, there does exist a non-trivial automorphism, and we have found it above by using the function `nfgaloisconj`. Indeed, if we apply this automorphism to `up2` (by typing `nfgaloisapply(nf,s,up2)` where `s` is the non-trivial component of `nfgaloisconj(nf)` computed above), we find the opposite of `u2`, which is OK.

Still another method which avoids all sign ambiguities and automorphism problems is as follows. Type `r = f1[1,1] % (x^2 - u)` to find the remainder of the characteristic polynomial of `u2` divided by `x^2 - u`. This will be a polynomial of degree 1 in `x` (with polmod coefficients) and we know that `u2`, being a root of both polynomials, will be the root of `r`, hence can be obtained by typing `u2 = -coeff(r,0) / coeff(r,1)`. Indeed, we immediately find the correct result with no trial and error.

Still another method to find the square root of `u` is to use `nffactor(nf,y^2 + u)`. Except that this won't work as is since 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 (using `subst`). I leave you to work out the details.

Now ideals can be used in a wide variety of formats. We have already seen HNF-representations, ideles and prime ideals. We can also use algebraic numbers. For example type `al = Mod(x^2 - 9, A)`, then `ideleprincipal(nf,al)`. We obtain the idele corresponding to `al` (see the manual for the exact description). However it is usually not necessary to compute this explicitly since this is done automatically inside PARI. For example, you can type

```
for(i=1,4, print(i ":  " idealval(nf,al,P[i])))
```

We see that the valuation is non-zero (equal to 1) at the prime ideals `P[2]` and `P[3]`. In addition, typing `norm(al)` shows that `al` is of norm $49 = 7^2$ (`idealnorm(nf,al)` gives the same result of course, and would be more generic). Let's check this differently. Type `P23 = idealmul(nf,P[2],P[3])` and then `idealhnf(nf,al)`. We see that the results are the same, hence the product of the two prime ideals `P[2]` and `P[3]` is equal to the principal ideal generated by `al`. There is still something to be done with this example as we shall see below after we introduce Big Number Fields (which will trivialize the examples we have just seen).

Essentially all functions that you would want 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`, `idealprincipal`, `idealred`, `idealstar`, `idealtwoelt`, `idealval`, `ideleprincipal`, `nfisideal`.

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

Apart from the above functions you have at your disposal the very powerful functions `bnfclassunit` which is of the same type as `quadclassunit` seen above, but for general number fields, and hence much slower. See Chapter 3 for a detailed explanation of its use.

First type `setrand(1)`: this resets the random seed (to make sure we get the exact same results). Now type `m = bnfclassunit(A)`, where `A` is the same polynomial as before. After some work, we get a matrix with one column, which does not look too horrible (this is because much of the really useful information has been discarded, this is not an `init` function). Let's extract the column with `v = m[,1]`.

Then `v` is a vector with 10 components. We immediately recognize the first component as being the polynomial `A` once again. In fact member functions are still available for `m`, even though it was not created by an `init` function. So, let's try them: `m.pol` gives `A`, `m.sign`, `m.disc`, `m.zk`, ok nothing really exciting. But new ones are available now: `m.no` tells us the class number is 4, `m.cyc` that it's cyclic (of order 4 but that we already knew), `m.gen` that it's generated by a prime ideal above seven (in HNF form). If you play a little bit with the `idealhnf` function you see that this ideal is `P[4]`.

The regulator `m.reg` is equal to $3.794\ldots$. `m.tu` tells us that the roots of unity in K are exactly the sixth roots of 1 and gives a primitive root. Finally `m.fu` gives us a fundamental unit (which must be linked to the unit `u2` found above since the unit rank is 1).

To find the relation (without trial and error, because in this case it is quite easy to see it directly!), let us use the logarithmic embeddings. Type `uf = m.fu[1]`, `uu= m.tu[2]` to get the generators of the unit group, then

```
cu2 = log(conjvec(u2));
cuf = log(conjvec(Mod(uf,A)));
cuu = log(conjvec(Mod(uu,A)));
```

to get the (complex) logarithmic embeddings. Then type

```
lindep(real([cu2[1], cuf[1], cuu[1]]))
```

to find a linear dependence. Unfortunately, the result $[0, 0, 1]$ is to be expected since `uu` is a root of unity, hence the components of `cuu` are pure imaginary. Hence we must not take the real part. However, in that case the logs are defined only up to a multiple of $2i\pi$. Hence we type

```
lindep([cu2[1], cuf[1], cuu[1], 2*I*Pi])
```

The result $[1, -1, -3, 0]$ shows that `u2` is probably equal to `uf * uu^3`, which is clear since `uu` is a sixth root of unity, and which can be checked directly. This works quite nicely, but we'll see later on a much better method.

Note: since the fundamental unit obtained depends on the random seed, you could have obtained a different unit from the one given here, had you not reset the random seed before the computation. This was the purpose of the initial `setrand` instruction (which was otherwise completely unnecessary of course).

If you followed closely what component of `m` the various member functions were giving (after possibly some cosmetic change), you must have noticed that the seventh and tenth components were not given. You may ignore the last component of `v` (it gives the accuracy to which the fundamental unit was computed). The seventh is a technical check number, which must be close to 1 (between $1/\sqrt{2}$ and $\sqrt{2}$ at the very least). This tells us, under LOTS of assumptions (notably the Generalized Riemann Hypothesis), that our result is correct (i.e. neither the class group nor the regulator are smaller). We'll see shortly how to remove all those assumptions: we need much more data than was output here. This is a little wasteful since the missing data *were* computed, then discarded to provide a human-readable output. Thus you will usually not use this function, but its `init` variant: `bnfinit`.

So type: `setrand(1); bnf = bnfinit(A);`.

This performs exactly the same computations as `bnfclassunit`, but keeps much more information. In particular, you don't *want* to see the result, whence the semicolon (if you really want to, you can have a look, it's only about three screenful). All the member functions that we used previously still apply to this `bnf` object. In fact, `bnfinit` even recomputed the information provided by `nfinit` (we could have typed `bnfinit(NF)` to avoid the waste), and it's included in the `bnf` structure: `bnf.nf` should be exactly 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).

We are now ready to perform more sophisticated operations in the class group. First and foremost, we can now certify the result: type `bnfcertify(bnf)`. The output (`1` if all went well) is not that impressive, but it means that we now know the class group and fundamental units unconditionally (in particular, the GRH assumption could be removed)! In this case, the certification process takes a very short time, and you might wonder why it was 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 make the check afterwards, if the result is interesting enough (and looking at the tentative regulator, you know in advance whether the certification can possibly succeed: if `bnf.reg` is about 2000, don't waste your time).

Ok, 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)` then `vis = bnfisprincipal(bnf,pr4)`. The function `bnfisprincipal` now uses all the information contained in `bnf` and tells us that indeed `pr4` is principal. More precisely, the first component, `[0]`, gives us 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 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`$= \alpha\mathbf{Z}_K$, where $\mathbf{Z}_K$ is the ring of integers of our number field, α being as usual expressed on the integral basis. To get α as an algebraic number, we type `alpha = nfbasistoalg(bnf,vis[2])` (note that we can use a `bnf` with all the `nf` functions; but not the other way round, of course).

Let us check that the result is correct: first, type `norm(alpha)` (`idealnorm(nf, vis[2])` would have worked directly). It is indeed equal to $7^4 = 2401$, which is the norm of `pr4` (it could also have been equal to -2401). 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(nf,alpha)`.

The result that we obtain is identical to `pr4`, thus showing that `alpha` is correct (not that there was any doubt!). You may ignore the third component of `vis` which just tells us that the accuracy to which `alpha` was computed was amply sufficient.

But `bnfisprincipal` also gives us information for non-principal ideals. For example, type

`vit = bnfisprincipal(bnf, pr)`.

The component `vit[1]` is now equal to [3], and tells us that `pr` is ideal-equivalent to the cube of the generator `g` given by `bnfinit`. Of course we already knew this since the product of `P[2]` and `P[3]` was principal, as well as the product of all the `P[`*i*`]` (generated by 7), and we noticed that `P[4]` was of order 4 when we looked at `bnf.gen`.

The second component `vit[2]` gives us α on the integral basis such that `pr`$= \alpha$`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 a flag (1 in this case, see the online help) to the arguments of the `bnfisprincipal` function, so that it only returns the position of `pr` in the class group.

Let us now take the example of the principal ideal `P23` that we have seen above. We know that `P23` is principal, but of course we have forgotten the generator that we had found, so we need another one. For this, we type `pp = bnfisprincipal(bnf,P23)`. The first component of the result is `[0]`, telling us that the ideal is indeed principal, as we knew already. But the second component gives us the components of a generator of `P23` on the integral basis. Now we remember suddenly that we already had a generator `al` for the same ideal. Hence type `u3=nfeltdiv(nf,al2,al)`. This must be a unit. It's already obviously an integer and `idealnorm(nf,%)` tells us that `u3` is indeed a unit. You can again find out what unit it is as we did above. However, as we mentioned, this is not really the best method.

To find the unit, we use explicitly the third component of the vector `bnf` given by `bnfinit`. This contains an $(r+1)\times r$ complex matrix whose columns represent the complex logarithmic embedding of the fundamental units. Here $r = r_1 + r_2 - 1$ is the unit rank. We first compute the component of `u3` on the torsion-free part of the group of units by proceeding as follows. Type

```
me = concat(bnf[3],[2,2]~).
```

Indeed, this is a variant of the regulator matrix and is more practical to use since it is more symmetric and avoids suppressing one row arbitrarily. Now type

```
cu3 = ideleprincipal(nf,u3)[2]~
```

to get the complex logarithmic embedding of `u3` (as a column vector). We could of course also have computed this logarithmic embeddings directly using `conjvec` as we did above, but then we must take care of the factors of 1 and 2 occurring.

Then type `xc = matsolve(real(me), real(cu3))`

Whatever field we are in, if `u3` is a unit this *must* end with a 0 (approximate of course) because of the "spurious" vector $[2,2]$, and the other components (here only one) give the exponents on the fundamental units. Here the only other component is the first, with a coefficient of 1 (we could type `round(xc)` to tidy up the result). So we know that `u` is equal to `uf` multiplied by a root of unity.

To find this root of unity, we type `xd = cu3 - me*xc` then `xu = ideleprincipal(nf,uu)[2]` and finally `xd[1] / xu[1]`. We find 3 as a result, so finally our unit `u3` must be equal to `uu^3 * uf` itself, which is the case.

Of course, you don't need to do all that: just type `bnfisunit(bnf,u3)`. Like the `bnfisprincipal` function, this gives us the decomposition of some object (here a unit) on the precomputed generators (here `bnf.tufu`) of some abelian group of finite type (here the units of K). The result `[1,Mod(3,6)]` tells us that `u3` is equal to `uu^3 * uf` as before.

Another famous so-called *discrete logarithm* problem can be easily solved with PARI, namely the one associated to the invertible elements modulo an ideal: $(\mathbf{Z}_K/I)^*$. Just use `idealstar` (this is an `init` function) and `ideallog`.

—– TO BE COMPLETED —–

12. GP Programming

—– TO BE COMPLETED —–