Tutorial for the Pari Library

Henri Cohen

June 22, 2025

1 A First Example: Somos Sequences

This tutorial is meant as a gentle introduction to programming the Pari Library, and does not replace fuller courses that may be given. We assume reasonable knowledge of the use of the `GP` interpreter, and also a reasonable familiarity with the C language: note that I am not at all an expert in the C language, but I have been using it to program the Library for decades without knowing all the subtleties.

We assume installed the complete source code, for instance from a `GIT` repository.

I believe that the best way to start is simply to write a new function for `Pari/GP`, and we will comment each step as we go along.

Example 1. As a reasonably complete simple first example, let us write a program which computes Somos sequences. Let us start with the simplest non-trivial one, we will generalize later.

The Somos-4 sequence can be defined by $a_1 = a_2 = a_3 = 1$, $a_4 = 2$, and for $n \geq 5$ by the recursion

$$a_n = \frac{a_{n-1}a_{n-3} + a_{n-2}^2}{a_{n-4}}$$

What makes this sequence interesting is that all the a_n are integers, although we keep dividing by a_{n-4}.

Given an integer N, we want to output the N-component vector $(a_1, \ldots, a_N)$, and a possible `GP` program (with no checks whatsoever) would be the following:

```
somos4(N)=
{ my(V);
  V=vector(N);for(i=1,3,V[i]=1); V[4]=2;
  for(n=5,N,
    my(R=V[n-1]*V[n-3]+V[n-2]^2);
```

```
    if(R%V[n-4],error("Somos4 sequence is not integral ???"));
    V[n]=R/V[n-4]);
  return(V); }
```

We would now like to add this nice function to the GP interpreter. First, we have to put the C code which we are going to write *somewhere*. Of course we could do everything from scratch, but we are lazy and will use the existing Library tree. Since this is an arithmetic function, we will add it to one of the arithmetic programs of the Library, which not surprisingly are in `src/basemath/arithx.c` with x=1 or 2. Let us add it to `arith2.c`. So open that file, and at the end (after the `issquarefree` program) let us write our code. We begin as follows, omitting for now some checks and declarations:

```
GEN
somos4(ulong N)
{
  GEN V;
  V = cgetg(N + 1, t_VEC);
  for (i = 1; i <= 3; i++) gel(V, i) = gen_1;
  gel(V, 4) = gen_2;
```

We first meet the most important word in the Library: `GEN`. All Pari-specific objects are `GEN`'s, we will of course see how they are handled. `GEN` simply stands for "GENeric", and is an alias for pointer to `long`.

So the above declares a function `somos4` which will return a `GEN` and takes as argument an ordinary C `ulong` (unsigned `long`). The first instruction says that we will need a vector V (we do not know that it is a vector yet).

The next command `cgetg`, here `V = cgetg(N + 1, t_VEC);`, is by far the most important way to construct new Pari objects, in other words new `GEN's`. This command *creates* on the Pari stack an object which will occupy $N + 1$ words. In addition, it fills the code word (or words) with the necessary code which will tell the Library that it will be a vector, represented by `t_VEC`. Two crucial things to know about this construction: first, although the Library now knows that we will have a vector, the components are completely undefined, and trying to access them (we will see how) will probably result in a segfault. Note that this is completely different from the `GP` function `vector(N)` which not only creates the vector, but also fills it with zeros (and which is coded in the Library as `zerovec(N)`). Second, it is essential to study the Library manual to know how many codewords are necessary. Many Pari types need only ONE codeword, with the essential exception of integers (type `t_INT`) and reals (type `t_REAL`), which have their specific `cgetg` functions called (of course) `cgeti` and `cgetr`, as well as polynomials (type `t_POL`) and power series (type `t_SER`) for instance. So let us

forget this for a moment: since a vector needs a single codeword, to create an N-element vector we thus need $N+1$ words, explaining the command `V = cgetg(N + 1, t_VEC)`.

The next instructions first involve universal constants: `gen_1`, and `gen_2`: there also exist `gen_m2`, `gen_m1`, `gen_0`, `ghalf`, as well as `genI()`, which is in fact a function with 0 arguments. They also introduce another essential function, in fact a macro, `gel(V, i)` (`gel` is for Gen ELement), which means the i-th *component* (not codeword) of the vector `V`. The reason for the existence of this macro is as follows: we would have liked to write simply `V[i]`, and in fact this is correct C programming. But since a `GEN` is a pointer to `long`, `V[i]` is a `long`, not a `GEN`, so we would have to typecast it as `(GEN)V[i]`. This macro can be used as here on the left-hand side of an assignment (an *lvalue*), or on the right-hand side, so we will almost never have to do any typecasting.

(Note: we would have liked to have the type `GEN` to be a pointer also to a `GEN`, but this is not possible in the C-language, although it is possible in other languages.)

Let us now continue our program:

```
GEN
somos4(ulong N)
{
  GEN V;
  V = cgetg(N + 1, t_VEC);
  for (i = 1; i <= 3; i++) gel(V, i) = gen_1;
  gel(V, 4) = gen_2;
  for (n = 5; n <= N; n++)
  {
    GEN R = gadd(gmul(gel(V, n - 1), gel(V, n - 3)), gsqr(gel(V, n - 2)));
    if (!gequal0(gmod(R, gel(V, n - 4))))
      pari_err(e_MISC, "Somos4 sequence is not integral ???");
    gel(V, n) = gdiv(R, gel(V, n - 4));
  }
  return V;
}
```

We now meet the standard arithmetic operations such as: addition (`gadd`), multiplication (`gmul`), squaring (`gsqr`), division (`gdiv`), inversion (`ginv`), remaindering, i.e., modulo (not to be confused with INTMODs) (`gmod`), comparison (`gcmp`), as well as the comparison to 0, (`gequal0`), or to 1 (`gequal1`). The names are usually self-explanatory, the initial letter `g` meaning usually that the *result* is a `GEN` (except for the comparison operators which return 0 or 1), and the arguments are by default also `GEN`'s (we will see below what to do when the result or some of the arguments

are longs). We will see later the powering function gpow, which needs some extra information.

Note that a function such as gadd, gmul, and so on require two GEN arguments and *creates* a new GEN on the Pari stack. It is possible to avoid creating a new GEN using other types of functions, but we will not see this here.

We also meet the function pari_err, which has numerous incarnations that we will see below, but for now simply with the first argument e_MISC (for MISCellaneous) we just write an error message, just as GP's error, and then aborts the program.

Our program is now almost ready, we need of course to add the forgotten declaration ulong i, n;. Remember that in the GP interpreter, loop variables such as i and n above *should not* be declared (using my() or local()), but of course in C it is compulsory.

Now, how do we use this program from GP ? Nothing simpler. Since we cleverly have added it to the file arith2.c which is in the usual tree for compiling the Pari source, we simply go the head of the tree, usually a directory simply called pari, and type make. If you did not make any typos, it should compile perfectly, and you can now launch a new version of GP, probably by typing ./gp, and to have access to your program, you simply use the install command of the GP interpreter, and since we have put our program in the standard Library which is the default, there is no need to specify a library, so we simply write for instance inside GP:

```
? install(somos4, L);
? somos4(10)
% = [1, 1, 1, 2, 3, 7, 23, 59, 314, 1529]
```

The install prototype L means that the argument of the function is a long (as opposed to a GEN, which would be coded G) which also codes for ulong and int, and since the output is not specified, by default it is a GEN. Note for future reference that if the output had been a long or an int, the prototype must then *begin* with a lowercase l, for instance install(myprog, lL). There are of course more complicated prototypes, we will see some of them in later examples.

Remark. The above program is given as a simple example of Library programming. Very often, programming directly in the Library gives faster programs than the corresponding GP script, but in the present example this is not the case.

The above program, given for simplicity since it is our first use of the Library, lacks at least three additional parts to make it completely robust, plus optimizations to make it more efficient. Skip this on first reading.

1. First, *type checking* of the function arguments. Here the argument is an `ulong`, so it is the C compiler which does the type checking, but when an argument is a `GEN`, we usually want it to be of a certain `Pari` *type*, for instance an integer, or a polynomial, or a vector, etc... In `GP` you would write for instance

   ```
   if (type(z) != "t_VEC", error("incorrect type in myprogram"));
   ```

   This translates in the Library as:

   ```
   if (typ(z) != t_VEC) pari_err_TYPE("myprogram", z);
   ```

   Thus you must use `typ` and not `type`, there are no quotes around the type names, and there is a specific `pari_err` program called `pari_err_TYPE` which also allows you to print the offending argument.

2. Second, *domain checking*. For instance, to be rigorous the program that we wrote above does not work if $N \leq 3$. In `GP` we would write `if (N <= 3, error("N <= 3 in somos4"));` . In Library mode, we use `pari_err_DOMAIN` which has a slightly more complicated but almost self-explanatory syntax such as:

   ```
   if (N <= 3)
     pari_err_DOMAIN("somos4", "argument", "<=", utoi(3), utoi(N));
   ```

   Note that `utoi` is the function transforming a C `ulong` into a `Pari GEN` of type `t_INT`. We will see a much more detailed discussion of this type of functions in Section 5.

3. Third, *garbage collecting*, either inside the function, or at the end before the `return` statement. We will devote Section 3 to this very important aspect.

4. Finally, note that if we want to make such a program as a permanent Library feature, it is essential to optimize, and also to do what we can call microoptimizations (as opposed to algorithmic optimizations). Indeed, the general operation programs such as `gadd`, `gmul`, etc... are completely general programs working on `GEN`'s. But here, we know that we only work with specific `GEN`'s which are `t_INT`s. Thus, instead of `gadd`, it is better to use `addii`, instead of `gmul` we use `mulii`, etc... The names are self-explanatory (we will give more explanations below), the only specific functions whose names are more complicated are `dvmdii` and `truedvmdii`, both for integer division with remainder, for `dvmdii` the remainder being of the same sign as the dividend, and for `truedvmdii` the remainder being nonnegative. A possible more optimized program (still without checks or garbage collecting) is as follows:

```
GEN
somos42(long N)
{
  GEN V;
  long i, n;
  V = cgetg(N + 1, t_VEC);
  for (i = 1; i <= 3; i++) gel(V, i) = gen_1;
  gel(V, 4) = gen_2;
  for (n = 5; n <= N; n++)
  {
    GEN R = addii(mulii(gel(V, n - 1), gel(V, n - 3)), sqri(gel(V, n - 2)));
    GEN r;
    gel(V, n) = dvmdii(R, gel(V, n - 4), &r);
    if (!gequal0(r))
      pari_err(e_MISC, "Somos4 sequence is not integral ???");
  }
  return V;
}
```

As mentioned above, whenever possible it is better to use specific functions such as `addii`, `mulii`, and `sqri`, but the gain in efficiency is completely negligible. On the contrary, the use of the `dvmdii` function which gives both the quotient and the remainder gives a considerable speed gain since it avoids one division per loop.

Remark. The above example where we first give functions acting on general `GEN`s such as `gadd`, `gmul`, etc..., and then more specific and slightly faster functions such as `addii`, `mulii`, etc... is quite general in the Library. For many general functions there exist more specialized functions specific to given types, and in the present tutorial we will only mention a few.

Exercise

1. In fact, the `somos4` sequence starts at $n = 0$ with $a_0 = a_1 = a_2 = a_3 = 1$. We started at $n = 1$ to simplify, and you can easily modify it so that the program returns the $N+1$-component vector $(a_0, a_1, \ldots, a_N)$ (you have two methods: either shift all the indices by 1 in the program, or preconcatenate the result with 1 using `GEN gconcat(GEN x, GEN y)`).

   Write a similar program for the Somos5 sequence defined by $a_i = 1$ for $0 \leq i \leq 4$ and

$$a_n = \frac{a_{n-1}a_{n-4} + a_{n-2}a_{n-3}}{a_{n-5}}$$

2. The k-th Somos sequence is defined by $a_i = 1$ for $0 \le i \le k-1$, and

$$a_n = \frac{\sum_{1 \le j \le [k/2]} a_{n-j} a_{n-(k-j)}}{a_{n-k}} \ .$$

Write a general program `somos(N, k)` for computing N terms of the k-th Somos sequence. Check numerically that the sequence seems integral for $4 \le k \le 7$ (this is true but not completely trivial to prove), and not always integral for $k \ge 8$.

2 Example: Computation of an Infinite Series

You probably know that `Pari/GP` has very powerful numerical summation and integration programs such as `sumnum` and `intnum`. Our goal in the present example is not to replace or improve these programs (if it was reasonably possible, we would have done it), but to show how to program in C the sum of a sufficiently rapidly convergent series, so that `sumnum` is not necessary, but the more simple minded `suminf` (which sums the terms until they are sufficiently small) suffices.

Consider the function of a complex variable τ with $\Im(\tau) > 0$ defined by

$$E_2(\tau) = 1 - 24 \sum_{n \ge 1} \frac{n q^n}{1-q^n} \ , \quad \text{where} \quad q = e^{2\pi i \tau} \ .$$

Even if you do not know the theory of modular forms or elliptic functions, you should know that this function is of great importance.

We want to write a program to compute its values for $\Im(\tau) > 0$ without using the modular properties of E_2, but simply by summing the defining series. A straightforward `GP` program would be as follows:

```
E2(tau)=
{ my(q=exp(2*Pi*I*tau));
  return(1-24*suminf(n=1,n*q^n/(1-q^n))); }
```

Translating this into a C program is not completely trivial because the `GP suminf` function has a more complicated interface than we want for the beginning of a tutorial (we will see in Section 13 below how to do this). So we will write a longer but more explicit `GP` script, as follows:

```
E2(tau)=
{ my(q=exp(2*Pi*I*tau),B=getlocalbitprec(),S=0.,n=0,qn=1.);
  while(exponent(qn)>-B,
    n++;qn*=q;S+=n*qn/(1-qn));
  return(1-24*S); }
```

A corresponding C program could start as follows:

```
GEN
E2(GEN tau, long prec)
{
  GEN q = gexp(gmul(gmul(gmul(gen_2, mppi(prec)), gen_I()), tau), prec);
  GEN S = real_0(prec), qn = real_1(prec);
  long n = 0, B = prec2nbits(prec);
```

The first important thing to notice is that, although in `GP` the function `E2` has a single variable `tau`, in the C Library it is essential to include the default precision `prec` with which the computation will be done. Two crucial things to note concerning this: first, `prec` is *not* a reserved word, you can call it whatever you like, and when we will declare the prototype of `E2` we will write `install(E2,Gp)`, the lowercase `p` indicating that the second variable will be the precision of the computation.

Second, if the computation involves only inexact real or complex numbers, the computation will be done with the accuracy of the *input*, so `prec` will be ignored, but nonetheless must be included in the function definition. Consider for instance the exponential function `gexp`, with prototype `GEN gexp(GEN z, long prec)`. If you call `gexp(gen_1, prec)`, it absolutely needs to know `prec` to perform the computation. On the contrary, if you call `gexp(mppi(precold), prec)`, where `mppi(precold)` computes π to accuracy `precold`, then `prec` will be ignored, and the computation of the exponential will be done with the accuracy `precold`.

The computation of `q` in the next line looks complicated, but we have written it in this way so that everything is explicit. We will see that the Library has shortcuts (macros or inline functions) which will make programming much simpler.

The quantity $2\pi i$ involves three constants 2, π, and i. We have seen that 2 is represented by `gen_2`. However, multiplying by 2 or a power of 2 is so common that the function `gmul2n` with prototype `GEN gmul2n(GEN z, long k)` exists for this purpose: writing `gmul2n(z, k)` multiplies the `GEN z` by 2^k, where `k` can be a (positive or negative) `long`.

The "constant" π depends of course on the desired accuracy, so is not really a constant but a function `mppi(prec)` of the unique variable `prec`. Finally, the constant `i` is represented as the 0-variable function `gen_I()`. Now since $2\pi i$ is so often encountered in mathematics, the Library has a short-hand for it: the function `PiI2n(long n, long prec)` computes $2^n\pi i$ (in passing, note that `Pi2n(long n, prec)` computes $2^n\pi$). Thus instead of `gmul(gmul(gen_2, mppi(prec)), gen_I())`, we will simply write `PiI2n(1, prec)`.

On the next line, we initialize `S` to 0. and `qn` to 1.. In fact, in the present example it would not have mattered if we initialized them to 0 (`S=gen_0`)

and 1 (`qn=gen_1`), but in other circumstances it is essential to initialize to real numbers (think of the difference between `sum(n=1,1000000,1/n)` and `sum(n=1,1000000,1/n,0.)`). This is done using the functions `real_0` and `real_1`. Note that `real_1` is really useful, but it is in general bad practice to use `real_0`, but let us forget this for now.

The last line contains the macro `prec2nbits`. Now I must admit that I don't really understand if `prec` stands for the number of words occupied by a real number (`prec = 4` at 38D on a 64 bit machine) or the number of bits of accuracy (`prec = 128` at 38D). It doesn't matter, `prec2nbits` does know, so B will always be the number of bits. In this example I believe that `B = prec` would also work, but let us be safe. We now continue our program, after simplifying the first instruction.

```
GEN
E2(GEN tau, long prec)
{
  GEN q = gexp(gmul(PiI2n(1, prec), tau), prec);
  GEN S = real_0(prec), qn = real_1(prec);
  long n = 0, B = prec2nbits(prec);
  while (gexpo(qn) > -B)
  {
    n++; qn = gmul(qn, q);
    S = gadd(S, gmulsg(n, gdiv(qn, gsubsg(1, qn))));
  }
  return gsubsg(1, gmulsg(24, S));
}
```

The function `gexpo` is for all practical purposes the same as the `GP` function `exponent`. The new functions that we meet are `gsubsg` and `gmulsg`, where the letter `s` stands for "single", meaning a C `long`, and of course `g` for `GEN`. Thus for all the standard arithmetic operations `gadd`, `gsub`, etc..., adding the suffix `sg` means that the first operand will be a `long` (i.e., not a `GEN`), adding the suffix `gs` would mean that the second operand is a `long`, and even adding the suffix `ss` would mean that both operands are `long`s, the result still being a `GEN`.

As before, include this in the Library tree, for instance at the end of `src/basemath/trans3.c`, compile it, and install it in the modified `GP` by `install(E2, Gp)`, where the letter `p` is essential to indicate the variable `prec`. For instance, check that $E_2(i) = 3/\pi$.

Exercise

1. For $k \geq 2$ an even integer, define

$$E_k(\tau) = 1 - \frac{2k}{B_k} \sum_{n \geq 1} n^{k-1} \frac{q^n}{1-q^n} , \quad \text{where} \quad q = e^{2\pi i \tau} .$$

Write a more general program `E(k,tau,prec)` which computes this quantity. The Bernoulli numbers are obtained thanks to the function called `bernfrac` both in `GP` and in the Library.

2. Improve your program by replacing $q^n/(1-q^n)$ by $1/(q^{-n}-1)$.

3. For a few complex values of τ with $\Im(\tau) > 0$, check for instance that $E_4(\tau)^2 = E_8(\tau)$ and $E_4(\tau)E_6(\tau) = E_{10}(\tau)$.

For this last question, you will of course write simple `GP` commands. It is however essential to know how to create complex numbers in the Library. Assume you want to create the complex number $a+bi$, where a and b are real numbers represented as `GEN`'s (usually if not always of type `t_INT`, `t_FRAC`, or `t_REAL`). There are two equivalent methods: the lazy method is simply to write `z=mkcomplex(a, b)`. The completely explicit method would be to use explicitly the internal structure of complex numbers as follows:

```
z = cgetg(3, t_COMPLEX);
gel(z, 1) = a; gel(z, 2) = b;
```

Although semantically correct, it would be *very bad* practice to write

```
z = gadd(a, gmul(b, gen_I()));
```

3 Same Example, with Garbage Collecting

We now come to one of the most subtle but crucial part of Library programming, garbage collecting.

We keep the `E2` example given above, installed in `GP`. Under `GP`, type the following:

```
? install(E2,Gp)
? \p10000
? E2(I/100);
```

Since we were not at all clever, choosing $\tau = i/100$ will give a very slowly convergent series, so this should require a ridiculous 30 or so seconds, while if we had been clever and used the modular properties of E_2, it would have required less than 0.5 seconds. But that is not the point: before the program even finishes, you will (probably) get warnings from `GP` that it needs to increase the stack size, and in fact if your defaults sizes are small, the program may even abort before finishing.

To avoid this, we need some way to do garbage collecting during the computation. Indeed, the variables `qn` and `S` change at each iteration, so we can delete all previous values as we go along.

Before doing this, let us consider the following modification to our program:

```
GEN
E2new(GEN tau, long prec)
{
  pari_sp av = avma;
  return gc_GEN(av, E2(tau, prec));
}
```

(of course in practice we do not create a new function but simply include the first instruction at the beginning and the second at the end of the initial program `E2`). We meet here the two basic constructs of garbage collection: there is a reserved identifier `avma` (for AVailable Memory Address) which at all times, as its name indicates, contains the first available address where `Pari` can create new objects. Although this is of course a pointer to `long`, for good programming practice we MUST use the specific type `pari_sp` meaning Pari stack pointer. Thus, to do garbage collecting the first thing to do is to keep the initial value of `avma` in some variable, here `av`.

The second thing to do is to clear the garbage generated by the computations in the function. In most cases this is done with the simple function `gc_GEN`: the instruction `gc_GEN(av, z)` clears everything that you have computed before `z` itself, and then `copies` the value of `z` at the initial available address `av` and returns this copy. Note in passing that the Library function which creates a copy of a `GEN` together with all its components is `GEN gcopy(GEN z)`.

Exercise.

In a `GP` session where the above functions have been `install`ed, type `\x`. You will see a description in hexadecimal of the last computed expression in the `Pari` stack of your session, and in particular the first hex number is the address of that expression. Now call `E2(I)`, and again `\x`. The difference between the two addresses (minus the space occupied by your last expression) will be the size occupied by your computation. This should be around 19600 bytes. Now do the same, but call `E2new(I)` instead. Here the difference should be around 1744 bytes. The difference is not negligible, and of course if it accumulates, the stack will soon overflow.

Let us come back to the initial problem of stack overflow *during* the computation of `E2`. Here we need to save both the values of `qn` and `S`. A very clumsy (but perfectly correct) way of doing this is to use a temporary Pari vector variable containing both and doing the garbage collecting on this vector. Also, note that we want to keep the value of `q`, so we cannot do the inner garbage collecting using `av`, but we need a new stack pointer which is set *after* the creation of `q`. We could thus write something like this:

```
GEN
E2(GEN tau, long prec)
{
```

```
  pari_sp av = avma, av2;
  GEN q = gexp(gmul(PiI2n(1, prec), tau), prec);
  GEN S = real_0(prec), qn = real_1(prec);
  long n = 0, B = prec2nbits(prec);
  av2 = avma;
  while (gexpo(qn) > -B)
  {
    GEN qnS;
    n++; qn = gmul(qn, q);
    S = gadd(S, gmulsg(n, gdiv(qn, gsubsg(1, qn))));
    qnS = gc_GEN(av2, mkvec2(qn, S));
    qn = gel(qnS, 1); S = gel(qnS, 2);
  }
  return gc_GEN(av, gsubsg(1, gmulsg(24, S)));
}
```

In this program, we use the function `mkvec2` which creates a vector with 2 `GEN` components (there also exist `mkvec`, `mkvec3`, `mkvec4`, `mkvec5`, and `mkvecn`). An instruction of the form `z = mkvec2(qn, S)` is essentially equivalent to `z = cgetg(3, t_VEC); gel(z, 1) = qn; gel(z, 2) = S;`. Note that these are *not* copies of `qn` and `S`, but simply pointers to their location, so they would be destroyed by garbage collecting. Luckily, `gc_GEN` does do this copying, so is safe. But (see below) other garbage collecting functions such as `gc_upto` do *not* copy, so cannot be used with `mkvecx`.

You can now try the same experiment as before, calling this at 10000D with $\tau = i/100$, it will indeed still require around 30s, but there will be no more warnings concerning the stack.

Now since this is an extremely frequent occurrence, the Library has of course considerably simplified this clumsy process. Instead of introducing a temporary variable `qnS` and writing the complicated last line of the while loop, we use the command `gc_all` as follows:

```
gc_all(av2, 2, &qn, &Sn);
```

This is equivalent to the last two lines of the while loop above. The number 2 indicates the number of variables to be kept, and of course do not forget the `&` sign, since `qn` and `Sn` will be moved, so we not only need to know their values but also their addresses. (To be absolutely correct, we should write `(void)gc_all(...)` since `gc_all` returns its first `GEN` argument, here `qn`.)

There is still something a little stupid in the above program. Sure, the garbage accumulates, but not that much, so it is a waste of time to do the garbage collecting at *each* loop. We could be clever, and only do it every 100 loops, say (try it!). If you do try it, you will notice that the program

indeed runs faster since we have suppressed almost all garbage collection, but only *slightly* faster (maybe 1 or 2%). This is because the garbage handling of `Pari/GP` (all the functions starting with `gc_`) is extremely efficient compared to other methods, used by almost all other systems.

We can do something more elegant than that by using the function `gc_needed`: we write:

```
if(gc_needed(av2, 1)) gc_all(av2, 2, &qn, &S);
```

This instruction says that if 50% of the stack is full, one should do a garbage collect (note: the integer 1 in `gc_needed` can be changed to 2 or more (which changes what proportion of the stack you allow to fill), but apart for very special needs, 1 is sufficient).

The final program is thus:

```
GEN
E2(GEN tau, long prec)
{
  pari_sp av = avma, av2;
  GEN q = gexp(gmul(PiI2n(1, prec), tau), prec);
  GEN S = real_0(prec), qn = real_1(prec);
  long n = 0, B = prec2nbits(prec);
  av2 = avma;
  while (gexpo(qn) > -B)
  {
    n++; qn = gmul(qn, q);
    S = gadd(S, gmulsg(n, gdiv(qn, gsubsg(1, qn))));
    if (gc_needed(av2, 1)) (void)gc_all(av2, 2, &qn, &S);
  }
  return gc_GEN(av, gsubsg(1, gmulsg(24, S)));
}
```

Of course since this is such an important function, a much faster implementation of this program `E2` and the more general program `E` written in the above exercise, already exist in the `Pari` Library under the name `cxEk`, which must be installed under `GP` or accessed using the `GP` function `elleisnum`, which however multiplies the result by suitable constants.

4 Example: Gegenbauer Polynomials

There exist many preprogrammed polynomial families in `Pari/GP` such as `pollegendre`, `polchebyshev`, `polhermite`, etc... Let us add one more: the Gegenbauer polynomials, defined by

$$C_n^\alpha(x) = \sum_{0 \le k \le \lfloor n/2 \rfloor} (-1)^k \frac{(\alpha)_{n-k}}{k!(n-2k)!}(2x)^{n-2k} ,$$

where $(\alpha)_m = \alpha(\alpha+1)\cdots(\alpha+m-1)$ is the rising Pochhammer symbol. A naive GP implementation is as follows:

```
poch(a,m)=prod(j=0,m-1,a+j);

polgegen(a,n)=sum(k=0,n\2,(-1)^k*poch(a,n-k)/(k!*(n-2*k)!)*(2*'x)^(n-2*k));
```

To see how to convert this into a C program for the Library, the only new thing to learn is how to represent the variable 'x (or any other variable). This is simply done using the function pol_x(long v), where v is the variable number. For now, you only need to know that there exist two predefined variables: 'x and 'y, with respective variable numbers 0 and 1. There also exist pol_0(long v) and pol_1(long v) which create the constant polynomials 0 and 1 in the variable number v.

We give directly the full program, including garbage collection, as follows (evidently we should use Horner's rule, and more clever computation of the coefficients using a trivial recursion, but we are giving a simple example so no optimization), and will comment after:

```
GEN
poch(GEN a, long m)
{
  GEN P = gen_1;
  long j;
  for (j = 0; j < m; j++) P = gmul(gaddgs(a, j), P);
  return P;
}

GEN
polgegen(GEN a, long n)
{
  pari_sp av = avma;
  GEN P = gen_0, X2 = gmul2n(pol_x(0), 1);
  long k;
  for (k = 0; k <= n/2; k++)
  {
    GEN C;
    C = gdiv(poch(a, n - k), gmul(mpfact(k), mpfact(n - 2*k)));
    C = gmul(C, gpowgs(X2, n - 2*k));
    if (odd(k)) C = gneg(C);
    P = gadd(P, C);
  }
  return gc_GEN(av, P);
}
```

Comments on these programs. The `poch` program is straightforward. For `polgegen`, we first set `X2` equal to `2x` (I usually reserve the variable `X` for `x` itself). The `for` loop used to perform the summation has the variable k going up to `n/2`, but remember that in C, `n/2` is computed as the integer part of `n/2`, while in `GP`, `n/2` would be a fraction if `n` is odd, so we needed to write `n\2` in the `GP` script (in fact `n/2` would also have worked, but would have been less elegant).

In the program we meet several new functions: first `mpfact`, which is the ordinary factorial function giving an *integer*, i.e., a `t_INT`. Very often (not here), the factorials are large, and it is sufficient to have approximations as real numbers (as opposed to integers): in that case, you should use the function `mpfactr`, which returns a `t_REAL`.

Second, the function `gpowgs`. As its last two letters implies, it takes two arguments, the first a `GEN`, the second a C long. Concerning this: the most general powering function is (of course) called `gpow`. But in general, we compute x^y as $\exp(y\log(x))$, so we need to specify the working accuracy, so `gpow` has three variables `gpow(GEN x, GEN y, long prec)`. But when y is a C `long`, we compute x^y by the binary powering algorithm, so there is no need for `prec`, and the prototype for `gpowgs` is thus simply `GEN gpowgs(GEN x, long y)` (if y is a `t_INT`, one also uses the binary powering algorithm, so `prec` is ignored, and you may write `gpow(x, y, 0)`, or better, use the specific function `powgi`).

Concerning the necessity of a `prec` variable, a useful hint: in some cases, even though a function may *require* a precision variable such as `prec`, we may want either to ignore it completely, or to perform the computation at low accuracy (typical example: using a nontrivial formula you want to compute the number of terms necessary to obtain a certain accuracy in a computation. You could do the computation using C `double`, but you can also do it in the Library with low precision). You have available for this purpose some predefined `prec`'s, the most useful being `DEFAULTPREC`, which is 64 bits, so slightly more than a C `double` which is 53 bits.

The last two new functions that we meet in the program are `gneg` (negation) and `odd`. Instead of `odd(k)` we could have written `k%2`, or better `k&1L`, but `odd(k)` is clearer. Note that this applies to C longs: for `GEN` integers you have the function `mpodd`.

Another possible `GP` implementation of the Gegenbauer polynomials is to first create a vector of coefficients, and then apply `Pol` (`GEN gtopoly(GEN V, long v = -1)` in the Library, as follows:

```
polgegen2(a,n)=
{ my(P=Pol(vector(n\2+1,j,my(k=j-1);(-1)^k*poch(a,n-k)/(k!*(n-2*k)!))));
  P=subst(P,x,4*x^2);if(n%2,P*=2*x);P; }
```

This can be translated as follows:

```
GEN
polgegen2(GEN a, long n)
{
  pari_sp av = avma;
  long n2 = n/2, k;
  GEN V = cgetg(n2 + 2, t_VEC), P, X2 = gmul2n(pol_x(0), 1);
  for (k = 0; k <= n2; k++)
  {
    GEN C = gdiv(poch(a, n - k), gmul(mpfact(k), mpfact(n - 2*k)));
    gel(V, k + 1) = odd(k) ? gneg(C) : C;
  }
  P = gsubst(gtopoly(V, 0), 0, gsqr(X2));
  if (odd(n)) P = gmul(P, X2);
  return gc_upto(av, P);
}
```

In addition to the `gtopoly` function, we encounter `GEN gsubst(GEN x, long v, GEN z)`, where contrary to the `GP` syntax, the second argument `v` is not a variable such as `x`, but a variable *number*.

We could of course just as easily have written a slightly more general program which returns the Gegenbauer polynomials in any desired variable, not just `x`.

Since this example is about polynomials, here is some more detailed information and more useful functions for dealing with them. Let us assume that the `GEN` variable `P` is of type `t_POL`, a polynomial. The following functions, given with their prototypes, are essential (apart from `varn`, their names are identical to the `GP` names):

`long varn(P)`: the variable number of the main variable of `P`, so that the variable itself of `P` is `pol_x(varn(P))`.

`long poldegree(GEN P, long v)`: degree of `P` with respect to the variable `v`. Here and everywhere else, `v = -1` codes for the main variable of `P`. If the polynomial is in the variable `'x`, you can also put `v = 0`.

`GEN pollead(GEN P, long v)`: leading coefficient with respect to `v`.

`GEN polcoef(GEN P, long n, long v)`: coefficient of degree `n` in the variable `v` of the polynomial `P`. Note that this will usually create a *copy* of the desired coefficient, which wastes a little time. If (and only if) you are *certain* that `P` is of type `t_POL`, that `v` is its main variable (or of course `v = -1`), *and* that `n` satisfies $0 \leq n \leq \deg_v(P)$, you *may* use instead `gel(P, n + 2)`, since a `t_POL` has *two* codewords (the second codeword contains two items of information: first and foremost the variable number `v`, and second, a bit indicating whether the polynomial is identically 0 or not). The advantage of using `gel` is that it avoids unnecessary copies, contrary to `polcoef`, but you must be careful that *all* the conditions stated above are satisfied, otherwise

you will certainly get segfaults. So do *not* use it unless you know what you are doing.

Note that the function `GEN polcoef_i(GEN P, long n, long v)` does this for you.

For instance, the `pol_x` function is programmed as follows:

```
GEN
pol_x(long v)
{
  GEN p = cgetg(4, t_POL); /* 4 = 2 codewords + 2 coefficients */
  p[1] = evalsigne(1) | evalvarn(v);
         /* set sign to nonzero and variable to v */
  gel(p, 2) = gen_0; gel(p, 3) = gen_1;
         /* coefficients 0*x^0 + 1*x^1 */
  return p;
}
```

This is as good a time as any to give the list of accessors to the components of types of `GEN`'s, including many that we have not met:

For `V` of type `t_VEC` (vector) or `t_COL` (column vector), `V[i]` is accessed by `gel(V, i)`. For `V` of type `t_VECSMALL`, one accesses `V[i]` directly by `V[i]`. Also useful for `t_VEC`s and `t_COL`s is `GEN vecslice(GEN V, long y1, long y2)` which is essentially the Library equivalent of `V[y1..y2]` in `GP`.

If the vector or column vector `V` has components which are themselves vectors or column vectors, to access `V[i][j]` you use `gmael(V, i, j)` (there also exists `gmael3`, etc...).

For `M` of type `t_MAT`, you can use the macro `gel(M, j)` to access the jth *column* of `M`, since a matrix is implemented as a vector of columns. There is no macro to access the ith row, but there is a *function* `GEN row(GEN A, long i)` which does this for you (as an exercise, you may want to write it yourself). To access an individual entry (line i, column j) of the matrix, you could use `gmael`, since we know that a matrix is a vector of columns, but as `gmael(j, i)` with i and j exchanged, which is not very elegant. Instead, use the macro `gcoeff(i, j)` (which is of course aliased to `gmael(j, i)`).

For `P` of type `t_POL` or `t_SER`, use the macro `polcoef`, unless all the conditions mentioned above (and similar ones for `t_SER`) in which case you *may* use `gel`, but with a shift of 2 (`gel(P, m + 2)` instead of `polcoef(P, m, -1)`). Of course, an essential function is `gsubst`, seen above.

For `Q` of type `t_FRAC` or `t_RFRAC`, use the functions `numer` and `denom` for the numerator and denominator. You could of course use `gel(Q, 1)` and `gel(Q, 2)`, but this is very bad programming practice, and in fact would *crash* if at some point the fraction has simplified to an integer or polynomial.

For `z` of type `t_COMPLEX`, use the functions `real_i` and `imag_i`. Same remark as above: do *not* use the `gel` macro. Note that the functions `greal` and `gimag` create a *copy* of the component you want to access, which is less efficient.

Exercises.

1. We have mentioned that our two `polgegen` programs are particularly stupidly programmed since both the factorials and the Pochhammer symbols should be computed recursively. Try to write a much more optimized program where first, these expressions are indeed computed recursively, and second, where no use is made of the slightly expensive functions `gsubst` and `gtopoly`, but instead the polynomial is created from scratch using a similar method than the `pol_x` example given above, and then the coefficients are filled recursively.

2. As all orthogonal polynomials, the Gegenbauer polynomials satisfy a linear recursion, more precisely $C_0^\alpha(x) = 1$, $C_1^{(\alpha)}(x) = 2\alpha x$, and for $n \geq 2$:

$$C_n^\alpha(x) = \frac{1}{n}\left(2x(n+\alpha-1)C_{n-1}^\alpha - (n+2\alpha-2)C_{n-2}^\alpha\right)$$

   Write a C program to compute a vector of the first N Gegenbauer polynomials using this recursion.

5 Interlude: Using C `longs` and `ulongs` in the Library

It is of course essential for efficiency to use C `long`s or `ulong`s (and `double`s for floating point computations) as much as possible. Before seeing the specific tools that `Pari` has for this purpose, some important warnings.

If the operations that you perform can hold in the C `long` or `ulong` (unsigned long) types, you can (and should) of course use directly the C operations. Remember, however that they differ in several respects. First, in C division always gives the quotient: `1/2` in C gives 0, while in `GP` it is equal to 1/2. Second, and this is in my opinion a flaw in the initial design of C, as in most other programming languages: division involving negative integers *truncate* instead of taking the *floor*. Thus all of `(-7)/3`, `7/(-3)`, and `-7/3` give -2, and not -3. Same remarks of course for the remaindering operator %. Also, do not abuse of `ulong` as opposed to `long`: it may happen that some functions which you think returns only nonnegative integers may also return negative ones, in which case combining this result with an `ulong` will create nonsense. In preparing this tutorial I got stung by this bug because of the `cbezout` function, see the `iswolstenfast` program in Section 7 below.

Second, the powering symbol ^ used in GP means something completely different in C. If you want to compute x^y where x and y are unsigned C longs, you can use `ulong upowuu(ulong x, ulong y)` if you are sure that the result will fit in an ulong, and otherwise `GEN powuu` (note: powss does not exist).

Third, do not confuse C arrays with Pari vectors of type t_VEC or even t_VECSMALL. If you do not use at all any constructs using GENs you may of course use C arrays as much as you like, but it is essentially impossible (or at least strongly ill advised) to mix C arrays and Pari vectors.

The Library gives you a few functions which compensate for the lack of the corresponding operations in C, and which have two long arguments and returns a long (similar functions exist for ulong): smodss (the true remainder), sdivss_rem (the true Euclidean division with remainder), maxss and minss (which should have been called smaxss and sminss since the corresponding functions on GENs are gmax and gmin).

The Library of course has functions to convert from some C types to GEN and conversely. The general function to convert a C long (resp., an ulong) to t_INT is stoi (Single TO Int) (resp., utoi), the reverse being itos (resp., itou).

Beware that of course stoi and utoi always work, but itos or itou may overflow if the t_INT is too large.

For floating point operations, there exists the function gtodouble which tries if possible to convert a GEN to a C double, and the function dbltor which converts a C double to a t_REAL with accuracy 64 bits (even though double only has 53). These names are historical (I am responsible), and dbltor should have probably been called doubletor to be consistent.

Finally, as you know from GP itself, there is the t_VECSMALL type, which is a vector which contains only C longs (as mentioned above, do *not* confuse this with C arrays). Note that since the contents of a t_VECSMALL are not GENs, to access or to fill such a vector you must *not* use the gel (Gen ELement) macro, otherwise you will get a type mismatch, but simply ordinary C vector notation: for instance

```
GEN V = cgetg(11, t_VECSMALL);
for(i = 1; i <= 10; i++) V[i] = i*i;
```

Note that, as in GP, operations on t_VECSMALLs are rather limited.

6 Examples: Divisors, Factoring

After all, initially Pari was designed to help number theorists, so we are going to give a few easy examples coming from number theory, and the C-functions that we can use.

Recall that in our second example we defined $E_2(\tau) = 1-24\sum_{n\geq 1} nq^n/(1-q^n)$ with $q = e^{2\pi i\tau}$. If we expand the power series, we have in fact

$$E_2(\tau) = 1 - 24\sum_{n\geq 1}\sigma_1(n)q^n\ ,$$

where more generally $\sigma_k(n)$ is the sum of the kth power of the (positive) divisors of n. This last function is of course preprogrammed in the Library.

The main functions dealing with factoring and divisors are (not surprisingly) `factor` and `divisors`. But let us see this in more detail: the general factoring program (at least over $\mathbb{Q}$ or $\mathbb{Q}(X)$) is called in C simply `factor`, as in `GP`. This is a very sophisticated program since the underlying domain can be almost anything. If we are dealing with integers, it is more elegant (although not compulsory) to use the specific programs available in that case: `Z_factor` if the input is a `t_INT`, in which case the result is as usual a 2-column matrix of primes and exponents, or `factoru` if the input is a C `ulong`, in which case the result is a 2-component vector of two `t_VECSMALL`s, the primes and the exponents.

Of course, once known the factorization (of an integer, say), it is easy (but not completely trivial) to obtain its divisors. The Library provides the function `divisors` if the input is a `t_INT` (more general `GEN`s such as `t_POL`s are of course also supported), and `divisorsu` if the input is a C `ulong`. In the first case the result is a `t_VEC`, and in the second case the result is a `t_VECSMALL`.

Thus, the σ_k function could be reprogrammed as follows, assuming that the arguments are `ulong`s:

```
GEN
mysigma(ulong n, ulong k)
{
  pari_sp av = avma;
  GEN D = divisorsu(n), S = gen_0;
  long i, ld = lg(D);
  for (i = 1; i < ld; i++) S = gadd(S, powuu(D[i], k));
    /* D[i] and not gel(D, i) since D is a vecsmall */
  return gc_INT(av, S);
}
```

Here we meet the crucial function `lg` which we have not met up to now: for any `GEN D` whatsoever, `lg(D)` is the length (in words) of the object, including codewords, so you better know the number of codewords for each type. By far the most frequent use is for vectors/columns, in which case `lg(D)` is simply one more than the number of components, since there is a single codeword, explaining the termination instruction `i < ld = lg(D)`.

Also some more garbage collecting information: we have said that `gc_GEN` not only does garbage collecting, but mainly for safety reasons, also *recopies*

the result (essential for instance when the result is an `mkvecn`, as we have seen above). But if we are really sure that the final result has already been copied (this is *guaranteed* by all official `GP` functions, here `gadd`), then it is wasteful to do an additional copy (admittedly, the loss is only a few nanoseconds). Thus, instead of the safe `gc_GEN`, you can use instead the function `gc_upto`. Even better, since we know that the result is a `t_INT`, we can use the more specialized `gc_INT` as we have done in the above example.

Still another aspect of garbage collecting: if your function, not only has only C `long` arguments, but has a `long` (or even `void`) result, then the garbage collecting is much simpler: we simply must put back `avma` to its initial values. After an initial `av = avma;`, in principle it suffices at the end instead of doing a `gc_GEN` or `gc_upto`, to write `avma = av;` This usually works, but for technical reasons which I will not explain here it is better to use the command `set_avma`. For instance, in the case $k = 1$, if the C `ulong` n is reasonably small, we can be sure that $\sigma_1(n)$ fits in a `ulong`. One can then write (with no overflow check):

```
ulong
usigma1u(ulong n)
{
  pari_sp av = avma;
  GEN D = divisorsu(n);
  ulong S = 0, i, ld = lg(D);
  for (i = 1; i < ld; i++) S += D[i];
  set_avma(av);
  return S;
}
```

Note that it is essential to use `set_avma(av)` *after* the final `ulong` result has been computed. Consider for instance the following hypothetical program:

```
long
myprog(ulong n)
{
  pari_sp av = avma;
  GEN a, b, gR;
  ulong R;
  gR = Result of a complicated computation;
  set_avma(av);
  return (itou(gR));
}
```

(`itou` converts, if possible, a `t_INT` into a C `ulong`).

While the above program will almost always work, there may be some outside interference such as a user interrupt or other just between the last

two instructions, so that `gR` may have been corrupted. Thus, the proper way to program the last two instructions is

```
R = itou(gR);
set_avma(av);
return R;
```

Instead of worrying about this, the library provides trivial (because exactly equivalent to the above) garbage collecting functions for C `int`, `long`, and `ulong`, of course called `gc_int`, etc... Thus the program snippet above should simply be written `return gc_ulong(av, itou(gR))`, and this is what we will do from now on. Incidentally, note that *usually*, but not always (see the example of `iswolstenfast` below) `long` and `ulong` are treated in the same way, both in C and in the Library, but it is good programming practice to specify `ulong` instead of `long` if it is important in a program.

7 Example: Wolstenholme Primes

In the Library there are of course C equivalents of the `GP` functions involving primes, such as `nextprime`, `isprime`, etc... More subtle is the C-equivalent of `forprime`, when we want to loop over primes. We choose as example the search for so-called *Wolstenholme primes*. Let p be a prime such that $p \geq 11$. Wolstenholme's theorem says that the numerator of $H_{p-1} = \sum_{1 \leq a \leq p-1} 1/a$ is divisible by p^2. An interesting exercise in number theory shows that for $p \geq 11$ prime the following five properties are equivalent:

1. p^3 divides the numerator of $H_{p-1} = \sum_{1 \leq a \leq p-1} 1/a$
2. p^2 divides the numerator of $H^{(2)}_{p-1} = \sum_{1 \leq a \leq p-1} 1/a^2$
3. p divides the numerator of $\sum_{p/6 < a < p/4} 1/a^3$.
4. p divides the numerator of the Bernoulli number B_{p-3}.
5. $\binom{2p-1}{p-1} \equiv 1 \pmod{p^4}$.

A prime satisfying any one (or all) of these conditions is called a *Wolstenholme prime*.

We want to write a C program using the Library to search for such primes (without giving the answer, which of course you can find instantly by googling, only two are known, both less than 3000000). Each of the above criteria would give a new program, but we want to focus on the prime search. So assume that we have written a yes/no program `int iswolsten(ulong p)`, which knowing that `p` is prime, outputs 1 if p is a Wolstenholme prime and 0 otherwise. We want to write a program which

ranges from $p = 11$ to some limit `lim` and outputs the Wolstenholme primes. If `iswolsten` is callable from `GP` (which is easily done via the install command `install(iswolsten,lL)`, where the *initial* letter `l` means that the result is a `long`, which by abuse is identified with `int`), we would simply write:

```
install(iswolsten,lL);
do(lim)=forprime(p=11,lim,if(iswolsten(p),print(p)));
```

We are now going to see how to do this `forprime` loop in the Library. We will assume that we only work with `ulong`'s, although corresponding loops exist for `t_INT`s.

The three types and functions that one must learn are:

`forprime_t`: This is a type which will contain the iterator which will produce the primes.

The function `u_forprime_init` with prototype

```
int u_forprime_init(forprime_t *T, ulong a, ulong b)
```

where the primes will run from `a` to `b` (of course `a` and `b` need not be primes). For `t_INT` the function is of course `forprime_init` with two `GEN` arguments replacing the two `ulong` arguments.

The function `u_forprime_next` with prototype

```
ulong u_forprime_next(forprime_t *T)
```

(or simply `forprime_next` for `t_INT`), which as its name indicates, gives you the next prime as an `ulong`, and returns 0 when the loop is finished, no more primes to use.

Thus, to program the above `forprime` loop, one writes the following

```
void
dowol(long lim)
{
  pari_sp av = avma;
  forprime_t S;
  ulong p;
  u_forprime_init(&S, 11, lim);
  while((p = u_forprime_next(&S)))
    if(iswolsten(p)) pari_printf("p = %ld\n", p);
  set_avma(av);
}
```

We have not yet seen how to print a result using the Library. One of the most general commands is `pari_printf`, which has more or less the same syntax as C `printf`, the essential difference being that it also supports the type `GEN`, which is coded with the format `%Ps`.

Note that we have written compactly `while((p = u_forprime_next(&S)))`, which does first the assignment of the next prime to the variable `p`, and second a test to see it `p` is zero. We could have separated those two actions, but it would be more clumsy.

Aside on iterators. The only reason that we need the above type of construction is that for obvious reasons `forprime` does not exist in the C language, and is not easily simulated (for instance `forstep` does not exist in C but is trivially simulated by an ordinary `for` loop). Thus the Library has a large number of such iterators, with essentially identical use, corresponding to the `GP` iterators such as `forsquarefree`, `forvec`, `forpart`, etc... For instance for `forvec` you have the type `forvec_t`, the initialization `forvec_init`, and the iterator itself `forvec_next`.

Now that the main program is done, it remains to write one of the functions `iswolsten`. We will write the simplest one (but probably the least efficient one), and suggest to the reader to write all the others and compare their speed.

```
int
iswolsten(ulong p)
{
  pari_sp av = avma;
  GEN N = numer(bernfrac(p - 3));
  long R = smodis(N, p);
  return gc_int(av, R == 0);
}
```

As the last two letters indicate, the function `smodis` returns the remainder (MOD) of the division of a `t_INT` by a C long, which is itself a long, whence the initial `s`.

Note also the use of `gc_int`, which is the C `int` version of `gc_INT` seen above, and which is exactly equivalent to `set_avma(av); return R == 0` as we have already mentioned above (we could even use `gc_bool`!).

Exercise. As suggested above, write four other `iswolsten` programs in the Library using the four other equivalent conditions for being a Wolstenholme prime, and compare their speed. Note that if you program them directly using existing Library functions like `harmonic`, `harmonic0`, or `binomialuu`, you probably will not get very far in your search.

So let us try to be efficient. We will use the criterion that p divides the numerator of $\sum_{p/6<a<p/4} 1/a^3$, and assume that $p < 2^{31}$ on a 64-bit machine (we will be happy to change the program if we reach that limit) so that the product of two integers reduced modulo p still fits in a C `ulong`.

The idea is simply that if we denote by abuse of notation by a^{-1} any *integer* which is the inverse of a modulo p, the criterion is equivalent to

saying that the sum of the C-*integers* $(a^{-1})^3$ (suitably reduced modulo p) with a ranging between $p/6$ and $p/4$ is divisible by p. Finding the inverse of an integer modulo another is done using the extended Euclidean algorithm, called `bezout` in `GP` and in C, but there is of course a corresponding Library function for C `long` called `cbezout` (and the GCD is called `cgcd` instead of `ggcd` for `GEN`). Thus a possible program is as follows:

```
int
iswolstenfast(long p)
{
  long lim1 = (p+5)/6, lim2 = p/4;
  long a, S = 0, ct = 0;
  for (a = lim1; a <= lim2; ct++, a++)
  {
    long u, v, u3;
    (void)cbezout(a, p, &u, &v);
    /* u will be an inverse of a mod p with |u|<p */
    u3 = (u*u)%p; u3 = (u*u3)%p; /* works if p < 2^31 */
    S += u3; ct++; if (ct%100 == 0) S %= p;
  }
  return S%p == 0;
}
```

Note that we do not need the result of the GCDs since we know they will all be equal to 1, so we typecast to `(void)` the result.

A crucial feature of this example is as follows: if `p` had been declared `ulong` as it should, then the operations using `long u, v, u3` afterwards would be completely wrong since `u` may be negative. It is thus essential to declare `p` as a `long` and not as an `ulong` in this program (we could also keep `p` as an ulong and add after the `cbezout` call: `while (u < 0) u += p`).

There is, however, a more elegant (although sometimes very slightly slower) way of doing the above computation, by using the Library `Fl_xxx` functions, or if we had `t_INT`s instead of C `long`s, the `Fp_xxx` functions.

These functions, such as `Fl_add`, `Fl_mul`, `Fl_sqr`, `Fl_inv`, etc..., assume that their argument(s) z are all C `long` or `ulong` such that $0 \leq z < p$, and guarantee that their result also satisfies this. All these functions are essentially trivial, with the exception of `Fl_inv` which calls an analogue of `cbezout` to compute the inverse modulo p (not necessarily prime). We can thus rewrite our program as follows:

```
int
iswolstenfast2(ulong p)
{
  long lim1 = (p+5)/6, lim2 = p/4, a;
```

```
  ulong S = 0;
  for (a = lim1; a <= lim2; a++)
    S = Fl_add(S, Fl_powu(Fl_inv(a, p), 3, p), p);
  return S == 0;
}
```

It is not any faster, but is more elegant, and incidentally has the (very slight) advantage of not being limited to 2^{31}. However, a big advantage is that one *can* make it considerably faster by using the `Flv_inv` program which does a batch inversion modulo p using a trick due to P. Montgomery. I leave this as an exercise for the reader.

As mentioned above, there also exist analogues of these instructions for `Pari's t_INT`s, and are called perhaps improperly `Fp_xxx`, although most of them do not only apply to primes.

Exercise. We can completely avoid inversions and hence obtain an even faster program using the following idea: let $(a_i)_{i\geq 1}$ be any sequence, and define

$$q_n = \prod_{1\leq i\leq n} a_i \quad \text{and} \quad p_n = q_n \sum_{1\leq i\leq n} \frac{1}{a_i}$$

Then of course $\sum_{1\leq i\leq n} 1/a_i = p_n/q_n$, but the main point is that we have the simple recursions with no inversions: $q_n = a_n q_{n-1}$ and $p_n = a_n p_{n-1} + q_{n-1}$. Write a Library program implementing this idea for $a_i = 1/i^3$ for $p/6 < i < p/4$, using `Fl_xxx` instructions.

8 Example: Bernoulli Numbers

Considering their importance, Bernoulli numbers have always been implemented in the Library, for instance by `bernfrac` and `bernvec`, which have the same C-name. The implementation is very efficient, but to familiarize ourselves with other types and functions in the Library, we are going to play with much less (in fact usually ridiculously less) efficient methods for computing them.

Example 1: Use of power series.

After all, by definition the Bernoulli numbers are defined by

$$\frac{x}{e^x - 1} = \sum_{n\geq 0} \frac{B_n}{n!} x^n$$

We are going to reimplement `bernvec`, such that `bernvec(N)` gives the $N+1$-component vector $(B_0, B_2, \ldots B_{2n})$. A possible `GP` program could be

```
mybernvec(N)=
{
```

```
  S = x/(exp(x+O(x^(2*N+2)))-1);
  return (vector(N+1,k,(2*k-2)!*polcoef(S, 2*k-2))); }
```

This assumes that the power series of $\exp(x)$ has been programmed, which is of course the case, but if we make no such assumption we should write the following:

```
mybernvec(N)=
{
  my(E,S);
  E = sum(k = 0, 2*N+1, x^k/k!, O(x^(2*N+2)));
  S = x/(E-1);
  return (vector(N+1, k, (2*k-2)!*polcoef(S, 2*k-2))); }
```

Let us see how to do this in the Library. For the first method, we first need to know what is the C name of the exponential function, not surprisingly `gexp`. But a new problem arises: how do we implement $O(x^{2N+2})$? There are a few ways to do this, but the simplest, if not the most elegant, is to use the function `zeroser` with prototype `GEN zeroser(long v, long e)`, which returns $O(X^e)$, where X is the variable with variable number v. We can now easily write our program:

```
GEN
mybernvec(long N)
{
  pari_sp av = avma;
  GEN X = pol_x(0), E, S, V;
  long k;
  E = gexp(gadd(X, zeroser(0, 2*N + 2)), 0);
  S = gdiv(X, gsubgs(E, 1));
  V = cgetg(N + 2, t_VEC);
  for (k = 1; k <= N + 1; k++)
    gel(V, k) = gmul(mpfact(2*k - 2), polcoef(S, 2*k - 2, 0));
  return gc_GEN(av, V);
}
```

Two things to note about this program, which otherwise is totally straightforward. First, note that the prototype of the exponential function `gexp` is `GEN gexp(GEN z, long prec)`, so even, as here, where `prec` is unnecessary since all the computations will be done in integers, it *must* be included, so we simply set it to 0. However, if instead we had needed something like $\exp(x+1+O(x^{2N+2}))$, writing

```
gexp(gadd(gaddgs(X, 1), zeroser(0, 2*N + 2)), 0);
```

would crash or give an error, since the program needs to compute exp(1), where 1 is the exact number 1, and it cannot do this without knowing the accuracy to which it must be computed.

A much more minor point is that we have written `polcoef(S, 2*k - 2, 0)` because we *know* that we have constructed the series S with the variable $'x$ with number 0. It would perhaps be safer in other circumstances to write `polcoef(S, 2*k-2, -1)`, since the variable -1 always denotes the main variable.

Let us now implement the second program. The only thing to change is that we must reimplement $\exp(x)$, i.e., the instruction

```
E = sum(k=0,2*N+1,x^k/k!,O(x^(2*N+2)));
```

Again not caring about efficiency, we simply write the following program snippet, where we recall that `X` has already been set to `pol_x(0)`:

```
E = zeroser(0, 2*N + 2);
for (k = 0; k <= 2*N + 1; k++)
  E = gadd(E, gdiv(gpowgs(X, k), mpfact(k)));
```

Let us now be a little more efficient. Even in `GP`, our original `mybernvec` program can be considerably improved (notwithstanding the fact that we should not recompute $(2k-2)!$ each time but by a recursion). There exists the function `serlaplace` (simply `laplace` in the Library) which does exactly what we want and gives a one-liner:

```
mybernvec(N)=Vec(serlaplace(x/(exp(x+O(x^(2*N+2)))-1)));
```

Exercises.

1. Programming this in the Library is of course trivial. Improve the efficiency of your program by noting that $x/2 + x/(\exp(x) - 1)$ is an even function, so that you can avoid doing `serlaplace` on half of the coefficients since they vanish. For this, reprogram the Library's `serlaplace` program so that it skips the odd-degree coefficients.

2. It is of course much better to write

$$x = (e^x - 1) \sum_{n \geq 0} \frac{B_n}{n!} x^n \ ,$$

   and by expanding the series product, to deduce the well-known recursion for Bernoulli numbers:

$$B_{2n} = -\frac{1}{2n+1} \left(-n - \frac{1}{2} + \sum_{0 \leq j \leq n-1} \binom{2n+1}{2j} B_{2j} \right) \ .$$

Write a new C program for computing `bernvec(N)` using this recursion. For 1/2, use `ghalf`, and For the binomial coefficients, use the function `ulong binomialuu(long n, long k)`, or if you feel like it, reprogram it yourself!

3. A much less trivial formula is as follows:

$$B_{2n} = -\frac{1}{(n+1)(2n+1)} \sum_{1 \le j \le \lfloor n/2 \rfloor} (2n-2j+1)\binom{n+1}{2j+1} B_{2n-2j}$$

for $n \geq 2$, which has the advantage of needing only $n/2$ terms instead of n in the previous recursion. Write another C program using this recursion together with $B_2 = 1/6$, and compare its speed with the previous one. To represent 1/6, you may either use the basic operations, necessarily using `stoi` or something similar, or a useful function `GEN sstoQ(long n, long d)` which creates n/d.

Example 2: Use of Truncated Pascal Matrices

Consider Pascal's triangle with the main diagonal of 1's suppressed, in other words the matrix $M_N = (m_{i,j})_{1 \le i \le N}$ with $m_{i,j} = \binom{i}{j-1}$ if $i \geq j$ and 0 otherwise. For instance

$$M_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 1 & 3 & 3 & 0 \\ 1 & 4 & 6 & 4 \end{pmatrix}$$

It is an amusing theorem that the first column of the inverse of this matrix is exactly the vector of all Bernoulli numbers B_i for $0 \leq i \leq N$ (thus to simulate `bernvec` one must then extract only the B_{2i}). This is of course an incredibly inefficient way of computing them, but let us program this in C, the `GP` script for the computing the matrix being trivially

```
M(N)=matrix(N,N,i,j,if(i<j,0,binomial(i,j-1)));
```

Remember that a matrix is a (row) vector of column vectors. We thus write the following:

```
GEN
mybernvec2(long N)
{
  pari_sp av = avma;
  GEN M, R, V;
  long j;
  M = cgetg(2*N + 2, t_MAT); /* matrix with 2N+1 columns */
  for (j = 1; j <= 2*N + 1; j++)
```

```
  {
    GEN C = cgetg(2*N + 2, t_COL); /* Prepare space for a column */
    long i;                        /*         with 2N+1 entries */
    for (i = 1; i <= 2*N + 1; i++)
    {
      if (i < j) gel(C, i) = gen_0;
      else gel(C, i) = binomialuu(i, j - 1);
    }
    gel(M, j) = C;
  }
  R = gel(ginv(M), 1); /* get first column */
  V = cgetg(N + 2, t_VEC);
  for (j = 0; j <= N; j++) gel(V, j + 1) = gel(R, 2*j + 1);
  return gc_GEN(av, V);
}
```

It is good programming practice to include the instruction `gel(M,j)=C` *after* having filled the column `C`, although it would also work if we included it before.

The new instructions that we meet here are first `binomialuu(long n, long k)`, which computes as a `GEN` the binomial coefficient $\binom{n}{k}$, and the function `ginv` for inverse.

Exercise. There exists a function `matpascal` which returns the complete Pascal matrix of binomial coefficients. To do the above in `GP` you can simply type the one-liner

```
mybernvec3(N)=((matpascal(N)-1)[^1,^-1]^(-1))[,1];
```

Program this in the Library.

9 Interlude: Fun with `Pari` Types

To get a feel of how to use most `Pari` types, we will just for fun reimplement the standard function $\exp(x)$, with no attempt at efficiency or even correctness in certain cases, but just so as to introduce new functions. The driver function is uninteresting, and would look something like:

```
GEN
mygexp(GEN x, long prec)
{
  pari_sp av = avma;
  GEN RES = gen_0; /* to keep the compiler happy */
  long tx = typ(x);
  switch(tx)
  {
```

```
    case t_INT: RES = mygexpint(x, prec); break;
    case t_REAL: RES = mygexpreal(x, prec); break;
    case t_COMPLEX: RES = mygexpcomplex(x, prec); break;
    ... other types
    default: pari_err(e_MISC, "mygexp not implemented for this type");
  }
  return gc_GEN(av, RES);
}
```

Thus, the function `typ` gives the type of an object as a long (recall that in `GP` the function `type` gives a `string` such as `"t_INT"`).

We thus need to implement $\exp(x)$ for the types that we want. It is however not necessary to have one function per type. A very useful function when the input is scalar is the Library function `GEN gtofp(GEN z, long prec)` which converts `t_INT`, `t_FRAC`, and `t_QUAD` to real or complex numbers with precision `prec`. Note that on `t_REAL` and `t_COMPLEX` inputs it changes the accuracy to be `prec`, which can increase or decrease the accuracy of the input. Thus, we can considerably simplify the `switch` in the above program and write for instance

```
case t_INT: case t_FRAC: case t_QUAD: case t_REAL: case t_COMPLEX:
  RES = mygexpcomplex(gtofp(x, prec), prec); break;
```

(or include the `gtofp` instruction in the function `mygexpcomplex` itself). If we do not want to change the accuracy of a `t_REAL` or of a `t_COMPLEX` with already inexact components, we should of course be a little more careful.

Exercise. When you are a little more familiar with Library programming, look at the implementation of `gtofp`, and write a new one which does not change the accuracy of inexact components.

A completely naive way of implementing the `mygexpcomplex` function is similar to what we did for `E2`:

```
static GEN
mygexpcomplex(GEN x, long prec)
{
  GEN S = real_1(prec), xn = real_1(prec), factn = xn;
  long B = prec2nbits(prec), n = 0;
  while (gexpo(xn) - gexpo(factn) > -B)
  {
    n++; xn = gmul(x, xn); factn = mulsr(n, factn);
    /* xn will contain x^n and factn will contain n! */
    S = gadd(S, gdiv(xn, factn));
  }
  return S;
}
```

A few remarks:

1. Since `mygexpcomplex` is a subprogram of the driver program `mygexp`, it is good programming practice to declare it `static`. For the same reason, since garbage collecting will be done in the the driver program, it would be a waste of time to do it here.

2. Note that to avoid any worry with the size of $n!$, we compute it as a `t_REAL` instead of a `t_INT`, although in practice this will never be a problem.

3. Note the use of the specific `mulsr` function (multiply C `long` by `t_REAL`) instead of the generic `gmulsg`. Although only infinitesimally faster, it is a good programming habit to use such specific functons whenever possible.

Note that as written, this program could be incorrect since the quantity $x^n/n!$ may first increase in size before decreasing and tending rapidly to 0. As an exercise, you may want to find if the program is really incorrect, and for which complex values of x.

The next interesting types for which the exponential function can be applied are polynomials, power series, and rational functions, of types `t_POL`, `t_SER`, and `t_RFRAC` respectively. In the same way that `gtofp(z, prec)` converts integers, reals, fractions, and complex numbers to a real/complex with accuracy `prec`, the function `GEN toser_i(GEN x)` converts polynomials and rational functions series to the `t_SER` type, essentially by adding $O(X^d)$ (i.e., `zeroser(v, d)`), where `X` is the main variable of the polynomial or rational function, for a suitable integer `d`, the default *series precision*. But contrary to *real precision*, which is specified in all functions involving non-exact complex numbers as a last argument often (but not necessarily) named `prec`, here there exists a *default* series accuracy called `precdl` which is a reserved identifier name. Thus `toser_i` does not have a second argument giving the series precision, but uses `precdl` by default (for fun, ask us about the meaning of this ending "`dl`"). Of course, nothing prevents you from adding `zeroser(v, d)` yourself to your polynomial or rational function, with a series precision `d` of your own choosing.

To summarize, one could use the code snippet

```
case t_POL: case t_RFRAC: case t_SER:
  RES = mygexpser(toser_i(x), prec); break;
```

It remains to program `mygexpser`, hence first a small briefing on the `t_SER` type. Let `z` be a `GEN` of this type, and for brevity let us assume that it is not the zero series. As for polynomials you use `varn(z)` to obtain the variable number of the main variable of `z`, and use `valser` to obtain the valuation and `lg(z) - 2` to obtain the number of terms in the series

(since there are two codewords, you must subtract 2). For instance, the series $z = 2/x + 3 + 4x^2 + O(x^3)$ has `varn(z)=0` since the variable is `x`, `valser(z)=-1`, and `lg(z) = 2 + 4 = 6` since the coefficients are $(2, 3, 0, 4)$.

Finally, note that, as in `GP`, you access coefficients of `z` using `polcoef`, and as usual do not forget to include the variable number at the end (or -1 if it is the main variable).

The beginning of the program could be as follows (assuming that we know that the input `z` is of type `t_SER`, or has been converted to that type using `toser_i`):

```
GEN mygexp(GEN z, long prec);

static GEN
mygexpser(GEN z, long prec)
{
  GEN S, E = gen_1;
  long v = valser(z), vkeep = v, n, N;
  if (v < 0)
    pari_err(e_MISC, "exponential of a power series with negative valuation");
  if (v == 0)
  {
    GEN a0 = polcoef(z, 0, -1);
    z = gsub(z, a0); v = valser(z); E = mygexp(a0, prec);
  }
```

Self-explanatory: one cannot compute $\exp(z)$ with $v < 0$, but if $v = 0$, z is of the form $a_0 + a_1 X + \cdots$, so we write $\exp(z) = \exp(a_0)\exp(z - a_0)$, where $z - a_0$ will now have strictly positive valuation. However, to compute $\exp(a_0)$ we need an accuracy with which to make the computation, so we realize that we need to add a `prec` argument to the function, which initially was not necessary (and is not necessary if $v > 0$, in other words $a_0 = 0$). We keep `E = mygexp(a0, prec)` (initialized to `gen_1` by default) so as to multiply by it at the end. Note the necessity in C of declaring the forward reference to `mygexp`.

We now deal with a series with strictly positive valuation v. If $v = 1$, the number of terms that we must take in the exponential series is exactly the number of significant terms of the series, which is `lg(z) - 2`, except when `z=0`, in which case one must take one term. If $v > 1$, the number of necessary terms is the ceiling of this quantity divided by v. Thus, we finish our program as follows (and we lazily compute $z^n/n!$):

```
  N = (maxss(1, lg(z) - 2) + v - 1)/v; /* ceiling of a quotient in C */
  S = gen_1;
  for (n = 1; n <= N; n++) S = gadd(S, gdiv(gpowgs(z, n), mpfact(n)));
  return gmul(E, S);
```

```
}
```

Of course, in practice we compute $z^n/n!$ more intelligently exactly as we did for `mygexpcomplex`, and if we want to be clever, we return `vkeep > 0 ? S : gmul(E, S)`.

There is in fact a very subtle but nasty bug in the above program. Try applying it to the series $z = \pi + x + O(x^3)$ for instance. If you have been lazy and not written the C program, you can also do it directly in `GP`, the same bug appears, as shown by the following `GP` example:

```
? z = Pi + x + O(x^3)
% = 3.1415926... + x + O(x^3)
? a0 = polcoef(z, 0); z -= a0
% = 0.E-37 + x + O(x^3)
? valuation(z, x)
% = 0
```

Unfortunately, or perhaps fortunately, `0.E-37` is not considered to be zero, so the valuation of `z-a0` is still 0, unfortunate if we want to compute its exponential. The way out of this is trivial *once we understand the problem*: under `GP` one uses the `serchop(z,1)` function, which applied to our `z` will return the desired `x + O(x^3)`, and the same is valid in Library mode where you use `GEN serchop(GEN z, long n)`. Thus, we must include the command `z = serchop(z, 1);` just after `z = gsub(z, a0)`, or more simply *replace* `z = gsub(z, a0)` by `z = serchop(z, 1)` to be sure that no constant term remains in `z`.

A word about p-adic numbers, type `t_PADIC`. As mentioned in the `Pari` manual, their implementation makes them slower than `t_INTMOD`s, and even for those it is usually much faster to do operations directly on `t_INT`s, and do the modulo operations at the end. On the other hand, it is a fact that essentially all complex functions have p-adic analogues, and in that case it is much simpler to use the `t_PADIC` type.

A p-adic number z is represented as $p^v u$, where $u \in \mathbb{Z}_p$ is either 0 or a p-adic unit, and u is defined modulo p^d, i.e., is of the form $u = a_0 + a_1 p + a_2 p^2 + \cdots + a_{d-1} p^{d-1} + O(p^d)$. The Library functions `padic_p`, `padic_pd`, `padic_u`, `precp`, and `valp` return respectively p, p^d, u, d, and v. Analogously to the Library version of $O(X^e)$ being `zeroser(v, e)`, the Library version of $O(p^e)$ is `zeropadic(p, e)`.

We will not give explicitly examples with p-adic numbers, but simply mention that the `mygexppadic` program that we must write is very analogous, to `mygexpser`: it suffices to replace the series valuation `valser` by the p-adic valuation `valp`, to note that contrary to series one (probably) cannot compute the exponential of a p-adic with zero valuation, and that the number of significant terms which was `lg(z)-2` is now simply `precp(z)`. However, beware that now $n!$ has a considerable influence on the number of

terms to take in the iteration. I leave the explicit writing of the program as an exercise.

The final types to which one can think of applying `mygexp` are linear algebra types, in particular `t_VEC`, `t_COL`, and `t_MAT`. For `t_VEC` and `t_COL`, the only reasonable thing to do is to compute the exponential componentwise. The program is thus trivial:

```
GEN mygexp(GEN z, long prec);

static GEN
mygexpveccol(GEN z, long prec)
{
  GEN V = gcopy(z); /* We should be more clever, see below */
  long i, lv = lg(V);
  for (i = 1; i < lv; i++) gel(V, i) = mygexp(gel(z, i), prec);
  return V;
}
```

Even though we lose time, the advantage of copying `z` instead of creating a new vector from scratch, is that it copies both the type (`t_VEC` or `t_COL`, but even `t_MAT` would work since the program is recursive) and the length.

We could do slightly better by using `shallowcopy` instead of `gcopy` since this only copies the pointers, not the tree. But of course, the proper way is simply to replace `V = gcopy(z)` by `V = cgetg(lg(z), typ(z))`. This exact instruction exists as `cgetg_copy` in the Library with prototype `GEN cgetg_copy(GEN z, long *plz)`, where `plz` is a pointer which will contain the length of `z`. So the above program should more properly be written:

```
static GEN
mygexpveccol(GEN z, prec)
{
  long lz, i;
  GEN V = cgetg_copy(z, &lz);
  for (i = 1; i < lz; i++) gel(V, i) = mygexp(gel(z, i), prec);
  return V;
}
```

Two remarks concerning applying functions to vectors. In the initial `Pari` design, the philosophy was (and still is in large part) that any operation which could make some kind of sense should be allowed. In particular, applying functions such as transcendental functions to vectors makes sense if we apply them componentwise as we did above. This was probably a bad decision, in particular it may hide bugs. Indeed, since there now exists the `GP` function `apply`, it is not necessary to have this behavior since you can simply write for instance `apply(exp, V)`.

In fact, the Library gives you macros which allow you to do exactly this in C. For instance, the above be written much more simply as

```
static GEN
mygexpveccol(GEN x, prec)
{ pari_APPLY_same(mygexp(gel(x, i), prec)); }
```

which will produce exactly the same effect as above, keeping the same type (usually `t_VEC` or `t_COL`), or

```
static GEN
mygexpveccol(GEN x, prec)
{ pari_APPLY_type(t_VEC, mygexp(gel(x, i), prec)); }
```

if we want to impose a specific type, here `t_VEC` (there also exist `pari_APPLY_long` for `t_VECSMALL`).

Note that these instructions are very fragile, and have the following constraints:

– the `GEN` variable *must* be called `x`, and the loop variable *must* be called `i` and should *not* be declared.

– The `return` statement is included, so do not add one.

We will see a further use of these `pari_APPLY` commands in Section 10.

Concerning the final type `t_MAT`, one could try to write a program for the exponential of a matrix, and not simply an element-wise exponentiation, but we will not do this here.

10 Adding a Function to the Library

We are now going to implement a small useful function which would deserve to be in the library, and explain how to add it so that it becomes part of your personal version of `Pari/GP` and directly callable by `GP` without any install.

This new function is `fracdep`, a generalization of `lindep`. Given two real or complex numbers x and y, we would like to know if there exist reasonably small integers a, b, c, and d such that $ad-bc \neq 0$ and $y = (ax+b)/(cx+d)$, i.e., if x and y are linked by a linear fractional transformation, and which returns that rational function if yes, and 0 otherwise. We even want a vectorized version, where we can test simultaneously if some element of a first vector is linked to some element of a second vector. This of course is easily done as a `GP` script, but we want to include it in the library.

Testing if $y = (ax+b)/(cx+d)$ is equivalent to testing whether xy, y, x, and 1 are $\mathbb{Q}$-linearly dependent, which is done with the function `lindep` in `GP`, or `lindep0` in the Library. Recall that the `GP lindep` has a second `flag` argument, by default 0, if we want to specify some accuracy to check

linear dependence, so we will also use this flag in `lindep0`. We will also use a trick suggested by Bill to add an irrational number to the list to check for correctness of the `lindep` output. This number must of course not be related to the inputs. We are going to use Euler's constant γ (`Euler()` in `GP`, and `mpeuler(prec)` in the Library) even though it is not known to be irrational, since it probably will never occur in the use of `fracdep` (if it did, simply use another constant).

```
static int
is_complex_type(GEN z)
{
  long t = typ(z);
  return is_real_t(t) || (t = t_COMPLEX &&
         is_real_t(typ(gel(z, 1))) && is_real_t(typ(gel(z, 2))));
}

static GEN
fracdep_i(GEN x, GEN y, long flag, long prec)
{
   GEN V, a, b, c, d, X, N, D;
   if (!is_complex_type(x)) pari_err_TYPE("fracdep", x);
   if (!is_complex_type(y)) pari_err_TYPE("fracdep", y);
   V = lindep0(mkvec5(mpeuler(prec), gmul(x, y), y, x, gen_1), flag);
   if (!gequal0(gel(V, 1))) return gen_0;
/* linear dependence with gamma, highly implausible */
   c = gel(V, 2); d = gel(V, 3); a = gel(V, 4); b = gel(V, 5);
   if ((gequal0(a) && gequal0(c)) || gequal(gmul(a, d), gmul(b, c)))
     return gen_0;
/* If a=c=0, fixed rational b/d, if ad-bc=0 also a constant, wrong */
   if (signe(c) < 0) { c = gneg(c); d = gneg(d); }
   else { a = gneg(a); b = gneg(b); }
/* So as to always have c >= 0 */
   X = pol_x(0); D = gadd(gmul(c, X), d); N = gadd(gmul(a, X), b);
   return gdiv(N, D);
}
```

Remarks.

1. Since `iscomplex` already exists in the Library but meaning something completely different, viz., not real, we name `is_complex_type` our auxiliary function, where we use the function `is_real_t(t)` which checks if the type `t` is one of `t_INT`, `t_REAL`, or `t_FRAC`.

2. Since type errors in functions are so common, there exists a specific error handling function `pari_err_TYPE` as above, whose first argument

is a string, typically the function name, and the second is a `GEN`, the offending argument of the function. If we had used `pari_err(e_MISC,...)`, it would have been awkward to also print the offending arguments.

3. We use the function `signe` on the variable `a` because we are *guaranteed* by `lindep` that `a` will be a `t_INT` (in fact, because of this guarantee, we could also use `negi` instead of `gneg` in the definition of `a` and `b`). Indeed, the macro `signe` may crash on other types, so in that case use the more general function `gsigne` which tries to give a sign to an object of any sign (note that this French name is used in part to avoid using `sign`, which could be used on C `long`s in certain cases).

4. Instead of writing `gadd(gmul(c, X), d)` etc..., it is preferable (faster and simpler) to use a short-hand called `GEN deg1pol(GEN c, GEN d, long v)`, or even better the shallow version with no copying `deg1pol_shallow`, so we do not need `X` and simply write:

```
D = deg1pol_shallow(c, d, 0); N = deg1pol_shallow(a, b, 0);
```

We can now easily write the vector version in a recursive manner, using the `pari_APPLY_same` function seen above:

```
GEN fracdep(GEN Vx, GEN Vy, long flag, long prec);

static GEN
fracdep_left(GEN x, GEN Vy, long flag, long prec)
{ pari_APPLY_same(fracdep(gel(x, i), Vy, flag, prec)); }

static GEN
fracdep_right(GEN Vx, GEN x, long flag, long prec)
{ pari_APPLY_same(fracdep(Vx, gel(x, i), flag, prec)); }

GEN
fracdep(GEN Vx, GEN Vy, long flag, long prec)
{
  pari_sp av = avma;
  GEN R;
  if (typ(Vx) == t_VEC) R = fracdep_left(Vx, Vy, flag, prec);
  else if (typ(Vy) == t_VEC) R = fracdep_right(Vx, Vy, flag, prec);
  else R = fracdep_i(Vx,  Vy, flag, prec);
  return gc_GEN(av, R);
}
```

As usual, if you have included it in the Library tree, for instance in the file `src/basemath/bibli1.c` which already contains `lindep`, you can use this function from `GP` after typing `install(fracdep, "GGD0,L,p");`,

where we recall that `D0,L,` means that the third argument is a long which defaults to 0 if it is not given. But we want to do more, and install it in our version of the library without needing to do any `install`.

For this, you need to do all of the following:

1. Add the name of the function with all declarations (here `GEN fracdep(GEN Vx, GEN Vy, long flag, long prec);`) in the file `src/headers/paridecl.h`, if possible near similar functions, in the present case near `lindep0`.

2. Create a description file having the `GP` name of your function, and add it to the directory `src/functions/xxx` containing similar functions. In our case, create a description file (we will see how below) called `fracdep`, and put that file in the directory `src/functions/linear_algebra`.

3. Recompile your `GP` tree after either doing a completely new `./Configure`, or more simply after doing `./Configure -l` which simply looks for new files. Your new function should now be available.

The way to write a description file is explained in detail in the Library manual, but we simply give the present example, with a few comments, and it can easily be taken as a template.

```
Function: fracdep
Section: linear_algebra
C-Name: fracdep
Prototype: GGD0,L,p
Help: fracdep(x,y,{flag=0}): look whether y is a reasonably simple
 expression of the form (ax+b)/(cx+d), return 0 otherwise. x and y
 can be either scalars or vectors.
Doc: look whether $y=(ax+b)/(cx+d)$ for reasonable $(a,b,c,d)$, and
 return $0$ otherwise. $x$ and $y$ can be either scalars or vectors.
 \kbd{flag} is as in \kbd{lindep}.

 \bprog
 ? fracdep(zeta(2), Pi^2)
 % = 6*x
 ? fracdep(Pi, (3*Pi + 4)/(Pi + 1))
 % = (3*x + 4)/(x + 1)
 ? fracdep(Pi, exp(1))
 % = 0
 ? V1 = [Pi^2, Pi, exp(1)]; V2 = [tanh(1/2), asin(1/2), zeta(2)];
 ? fracdep(V1, V2)
 % = [[0, 0, 1/6*x], [0, 1/6*x, 0], [(x - 1)/(x + 1), 0, 0]]
 @eprog
```

The template is self-explanatory. The C-name does not have to be the same as the `GP` name (although it is preferable); the prototype is of course what you type in the `install` command. `Help:` (the colon is compulsory) is the message that you obtain when you type in `GP ?fracdep`, with a single question mark, so simply write text, you cannot use TEX . `Doc:` (also compulsory colon) is the message that you obtain when you type a double question mark `??fracdep`. Here the message can contain some very primitive TEX (certainly not Latex nor amstex). In particular you can have mathematical formulas between dollar signs, and code snippets or other in typewriter mode `tt` using the macro `\kbd`, for KeyBoarD.

Crucial point: for technical reasons, all the lines in `Help:` and `Doc:` (apart of course from the first) MUST begin by a whitespace, as above.

After these descriptions, you may optionally add some programming examples (still keeping the rule that every line must begin with a whitespace), or simply some additional comments. The comments are in the same primitive TEX language as in the `Doc:` description, and the programs are in a special `verbatim` mode beginning by `\bprog` and ending with `@eprog` as above.

Finally, you may want to add tests, either by adding tests to an existing test file, or by creating a new one. In the present case, for such a simple function it is not necessary to create a new one, so we will add tests to the test function for `lindep`, which is a very close relative. For this, you must edit the file `lindep` in the directory `src/test/in/`, and add the tests somewhere in the file (if the file says keep errors at the end, do that of course, but not the `lindep` file). For instance add the following (or if you prefer the examples given in the above documentation):

```
fracdep(zeta(2),Pi^2)
fracdep(Pi,(3*Pi+4)/(Pi+1))
fracdep(Pi,exp(1))
fracdep(Pi,[1/(Pi+1),exp(1),sumalt(n=1,(-1)^n/(2*n-1))])
```

Now go to the head of the `Pari` source tree and type `make test-lindep`. You will of course get several error messages, the most important saying that files explain problems in `diff` format in a directory which should be `pari/Olinux-x86_64` if you are on a standard Linux system. In that directory, look at the file `lindep-sta.dif` and check that the new results (normally in green) are correct. Here you should see

```
! 6*x
! (3*x + 4)/(x + 1)
! 0
! [1/(x + 1), 0, -1/4*x]
```

(ignore the error concerning `Total time spent`).

We must now patch the output file (which is in the directory `src/test/32`, but you should *never* modify these files yourself, only look at them if you want), using the `patch` command as follows:

```
patch -p1 < Olinux-x86_64/lindep-sta.dif
make test-lindep
```

After the patch do another `make test-lindep` and look at the output file to see that all is OK.

11 A Word about Input/Output

Up to now, for simplicity we have not considered input/output in detail. As already mentioned, the Library provides the very versatile function `pari_printf`, which behaves in large part like `printf`, as well as `pari_flush()`, roughly equivalent to `fflush(stdout)`. As for `printf`, you first specify the format, using for instance `%d`, `%ld`, etc..., but in addition you have the specific `%Ps` to print an arbitrary `GEN`. I reproduce the example from the `Libpari` manual:

```
pari_printf("x[%d] = %Ps is not invertible!\n", i, gel(x, i))
```

Concerning input, you can use `GEN gp_read_str(const char *s)`, or `GEN gp_read_stream(FILE *file)` to read from a file. I refer to the manual for a full explanation.

There are also numerous functions for handling strings, in particular to convert strings to `GEN` and conversely. In the Library, a `GEN` containing a string is of type `t_STR`. To convert a string to this type, the basic function (there are many others) is `GEN strtoGENstr(const char *s)`. To concatenate strings you have `gconcat` and `gconcat1` which are the library equivalent of `GP`'s `concat(x,y)` and `concat(x)`, but it is in general better to use *shallow* versions which only copy the head of the `GEN` tree and not its components, here `shallowconcat` and `shallowconcat1` (there are a few other shallow functions such as `shallowcopy` instead of `gcopy`).

However in general, it is better to create a `t_VEC`, fill it with strings as you go along, and do a `gconcat1` or a `shallowconcat1` at the end to obtain your final string.

Another string function is `strsplit` if you want to split a string into its component characters.

Just for fun, here is an example: recall that the output of `contfrac(z)` for a real number `z` is a vector containing the simple continued fraction of `z` with the number of terms depending on the accuracy of `z`, i.e., the vector `[a_0,a_1,...,a_n]` such that $z = a_0 + 1/(a_1 + 1/(a_2 + \cdots + 1/a_n))$, with the `a_i` strictly positive for $i \geq 1$. We are going to write a simple-minded program which converts such a vector to a string. We note that there are $n + 1$ integers, n times the string "+1/", and $n - 1$ pairs of opening and closing parentheses, for a total of $4n - 1$ strings to concatenate.

```
GEN
cftochar(GEN V)
{
  GEN R, pus = strtoGENstr("+1/");
  GEN op = strtoGENstr("("), cp = strtoGENstr(")");
  long n = lg(V) - 2, i, ct;
  if (n == 0) return strtoGENstr(""); /* empty string */
  R = cgetg(4*n, t_VEC);
  gel(R, 1) = gel(V, 1); ct = 1;
  for (i = 1; i <= n; i++)
  {
    ct++; gel(R, ct) = pus;
    if (i < n) { ct++; gel(R, ct) = op; }
    ct++; gel(R, ct) = gel(V, i + 1);
  }
  for (i = 1; i <= n - 1; i++) { ct++; gel(R, ct) = cp; }
  return shallowconcat1(R);
}
```

Remark: Assume that we re not able to compute the exact size of the vector that we need. In that case we reserve a vector which is guaranteed to be *at least* as long as what is needed. Then, before the final `shallowconcat1` or `gconcat1` we *must* correct for the exact length using the function `setlg`, otherwise there will be garbage on the `Pari` stack leading to a segfault. For instance, at the end of the above program we *know* that `ct` will contain the real length of the vector (1 less than the number of words it occupies). Thus, if initially we had written for instance `R = cgetg(6*n, t_VEC)`, before the final concatenation we would write `setlg(R, ct + 1)` (this wastes `2n` words on the stack, but at least the stack is not corrupted).

Exercises.

1. Same as above, but instead output the character string `s` which printed in `GP` with `print(s)` will print the TEX giving the continued fraction (hint: trivial, simply change the definitions of the variables `pus`, `op`, and `cp`).

2. It is in fact better to use the Library `struct pari_str`, with self-explanatory functions `str_init(&S, 1)`, `str_putc`, `str_puts`, and `str_printf`, and whose `string` content is `S.string`. Rewrite the above programs using these functions instead.

3. Modify these programs so that when $a_0 = 0$ in the continued fraction, it does not print the initial `0+`.

12 A Word about Assignments

There exist assignment functions in the Library, in case you want to put the contents of some `x` into an existing `GEN z` of type `t_INT` or `t_REAL`, for instance `affii`, `affir`, `affrr`, `affsi`, `affsr`, etc..., where the last two letters indicate the types of `x` and `z` respectively (note that for instance `affri` does not exist, if you want to convert a `t_REAL` to a `t_INT` you must use Library equivalents of `floor/ceil/round` programs).

Assignments are not often used, so let me give a somewhat artificial example of use. Recall that one uses `mpfactr` in case we want to avoid a possibly expensive use of `mpfact`, so as to have a real approximation. In the same way, the function `harmonic(n)` (same name in the Library) gives $\sum_{1\le j\le n} 1/n$, and we would like to have a function `harmonicr(n)` which computes this sum as a real number, which is of course much faster as soon as n is large (for simplicity we do not consider `harmonic(n,k)` with $k > 1$).

A trivial `GP` program to do this is simply `harmonicr(n)=sum(j=1,n,1./j)` or `harmonicr(n)=sum(j=1,n,1/j,0.)`, the latter being slightly slower but more accurate. In the Library one would write

```
GEN
harmonicr(ulong n, long prec)
{
  GEN S = real_0(prec);
  long j;
  for (j = 1; j <= n; j++) S = gadd(S, sstoQ(1, j));
  return S;
}
```

plus of course some garbage collecting (recall that `sstoQ` is a very practical function for creating small rational numbers).

As always in a very long loop, one should be careful about the intermediate size of the stack, and as in the `E2` example of Section 3, one should add something like `if (gc_needed(av2, 1)) gc_all(...)` (in fact here `gc_GEN` since only `S` needs to be preserved).

But there is another way using assignments:

```
GEN
harmonicr(ulong n, long prec)
{
  GEN S = cgetr(prec);
  long j;
  pari_sp av = avma;
  affsr(0, S);
  for (j = 1; j <= n; j++)
  { affrr(gadd(S, sstoQ(1, j)), S); set_avma(av); }
```

```
  return S;
}
```

We do not have to worry about the stack since it it cleared at each iteration.

Warning: You may wonder why, unlike in the previous program, we did not initialize `S` directly at the beginning by `S = real_0(prec)`, thus avoiding the additional `affsr(0, S)`. The reason is that whatever `prec` is, `real_0(prec)` will only allocate 2 words, the two codewords, so it cannot contain any other real number than 0. (one could think of changing this behavior). To avoid this problem, we could initialize `S` to `real_1(prec)` instead, and begin the summation at $j = 2$.

Note that there are faster ways to implement this sum, but this is not the purpose of this example. However, I must mention that the use of the logarithmic derivative of the gamma function (`psi(z)` in `GP` and `gpsi(z)` in the Library) is *much* faster (and more precise), as soon as n is more than a few hundred, using the formula $H_n = \psi(n+1) + \gamma$, which is trivial to implement in the Library as a one-liner.

13 Library Use of `sumnum`, `intnum`, etc...

We are now going to see how to translate into C `GP` constructs such as `sumnum`, `intnum`, `derivnum`, etc... To take a specific example, assume that we want to compute the Mellin transform of the Bessel function K_0, in other words

$$M(s) = \int_0^\infty t^{s-1} K_0(t)\, ,dt$$

In `GP` one would write this function as:

```
M(s)=intnum(t=0,[oo,1],t^(s-1)*besselk(0,t));
```

We now want to write this in C, either to export it as a new function of the Library, or because we need to use it inside some other program. A cheat way of doing this is to call the `gp2c` compiler which will do this for you: put the above in a file `file.gp`, say, and type `gp2c file.gp > file.c`. You will see that before writing the function itself, `gp2c` has created two auxiliary functions, probably called something like `anon_0` and `wrap_anon_0`. It is in fact sufficient to have a single auxiliary function as follows. To compute the integral, we need to know argument s, but also the working accuracy `prec` which may or may not be related to the accuracy of s or t. We thus write the following function which computes the integrand:

```
static GEN
_auxM(void *E, GEN t)
{
```

```
  pari_sp av = avma;
  GEN s = gel(E, 1), sm1 = gsubgs(s, 1);
  long prec = itos(gel(E, 2));
  return gc_upto(av, gmul(gpow(t, sm1, prec),
                                kbessel(gen_0, t, prec)));
}
```

(As it happens, the C-name of the K-Bessel function is not `besselk` but `kbessel`. Remember, to find the C-name of a function, in `GP` simply type `??function`, and the C-name or C-names will be at the end). In the above, `E` will in fact be a vector containing the list of auxiliary arguments used by the function, here s and the accuracy `prec`, and `t` is the variable of integration. In general `E` can be absolutely anything (not necessarily a vector of even a `GEN`), and can be put to `NULL` if there are no auxiliary arguments.

It is good programming practice to use names starting with underscore "_" for auxiliary functions such as this, which people do not want to see outside the source code.

The program for the function $M(s)$ (for which we will use a longer name) is then simply

```
GEN
MellinBesselK0(GEN s, long prec)
{
  pari_sp av = avma;
  GEN R = intnum(mkvec2(s, stoi(prec)), _auxM, gen_0,
                 mkvec2(mkoo(), gen_1), NULL, prec);
  return gc_upto(av, R);
}
```

In the above, note `mkoo()` which gives $+\infty$ (`mkmoo()` would give $-\infty$). The prototype for `intnum` (and similar functions) begins with `GEN intnum(void *E, GEN (*eval)(void*, GEN),...)`. The first argument `E` is any C object which will contain all the auxiliary data that the function needs, and in our case we will use a two component vector containing s and the accuracy `prec` which needs to be converted to a `GEN`. The second argument is the name of the function with a single `GEN` argument which will compute the integrand.

To emphasize that `E` can be any valid C object, we could for instance define a `struct` with two entries:

```
typedef struct
{ GEN z;
  long prec;
} mys_t;
mys_t S;
S.z = s; S.prec = prec;
```

In the calling program we would replace `mkvec2(s, stoi(prec))` by `(void*)&S`, and in _auxM we would replace `s=gel(E,1); prec = itos(gel(E, 2))` by `mys_t *m = (mys_t*)E; s = m->z; prec = m->prec`.

Exercise. Knowing that the prototype of the `suminf` function is `GEN suminf(void *E, GEN (*eval)(void*, GEN), GEN a, long prec)`, write a C Library program implementing the function $E_2(\tau)$ as written in the first `GP` script for E_2. Note that here the auxiliary function will only depend on q and n, so the accuracy `prec` does not need to be added to `E`.

As a second example, we will see how to use the powerful summation program `sumnum` correctly. Let us first see how to use it in a trivial way. In `GP` we can write `Z(s)=sumnum(n=1,1/n^s)`. The `sumnum` algorithm is quite robust, but we cannot expect to have full precision except when s is an integer. In any case, that is not the point, we only want to see how to use it in Library mode.

As usual, we need an auxiliary program, which will be very similar to the previous one:

```
static GEN
_auxZE(void *E, GEN n)
{
  pari_sp av = avma;
  GEN s = gel(E, 1);
  long prec = itos(gel(E, 2));
  return gc_upto(av, gpow(n, gneg(s), prec));
}
```

The driver program is simply:

```
GEN
ZE(GEN s, long prec)
{
  pari_sp av = avma;
  GEN R = sumnum(mkvec2(s, stoi(prec)), _auxZE, gen_1, NULL, prec);
  return gc_upto(av, R);
}
```

We now consider an example where we must tweak `sumnum` so that if gives the correct result, even in `GP`.

Recall that we defined the Pochhammer symbol by $(a)_n = \prod_{0\le j<n}(a+j)$, which is trivial to program, both in `GP` and in C, as we have done above in the Gegenbauer example in Section 4. We would like to compute numerically

$$S = \sum_{n\ge 0} \frac{(-1/2)_n^2}{n!^2}$$

(we have $S = 4/\pi$).

This is a slowly convergent series, so we cannot use `suminf`. To use `sumnum`, it is necessary that the nth term be defined for real n, not only integer n. It is easy to transform the summand using the gamma function since $(a)_n = \Gamma(a+n)/\Gamma(a)$ and $n! = \Gamma(n+1)$. But this is not sufficient since `sumnum` uses large values of n which will overflow the gamma function, and even it it does not, it will entail a huge loss of accuracy. So finally, we use the formula

$$\frac{(-1/2)_n^2}{n!^2} = \exp(2(\log(\Gamma(n-1/2)) - \log(\Gamma(-1/2)) - \log(\Gamma(n+1))))$$

and incidentally recall that `Pari` has the function $\log(\Gamma(z))$ programmed as `lngamma` (`glngamma`) in the Library).

If in `GP` we write `sumnum(n=0,exp(2*(lngamma(n-1/2)...)))` we will get absolute nonsense. The reason is that `sumnum` needs to increase the working accuracy to do its magic, and it does not increase the accuracy of n (which I repeat will not be an integer anymore), which in my opinion is a bug. Thus the final `GP` program which works is the following, where for efficiency we remove the $\exp(-2\log(\Gamma(-1/2))) = 1/(4\pi)$ from the sum:

```
myS()=
{
  B = getlocalbitprec();
  real(sumnum(n0=0,my(n=bitprecision(n0,3*B/2));\
       exp(2*(lngamma(n-1/2)-lngamma(n+1))))/(4*Pi);
}
```

(we use $3B/2$ as a rough guess of the accuracy that we need, if the result is nonsense we can increase it.

Let us see how to do this in the Library. Here we have no additional parameter, so `E` will only contain `prec`, and since `E` can be anything, let us leave it as a `long`. But in the auxiliary program, we must increase the accuracy of `n0` by 50%, but only when `n0` is not exact (i.e., a `t_INT` or a `t_FRAC`). For this, we use the library function `GEN precision(GEN z)` which returns the bitprecision of a real or inexact complex number `z`, and 0 if it is an exact scalar. We first wite a small program which does this accuracy increase if necessary, and then the auxiliary program:

```
static GEN
gprec_extend(GEN n)
{
  long B = precision(n);
  return B ? gprec_w(n, nbits2prec(3*B/2)) : n;
}
static GEN
```

```
_auxS(void *E, GEN n0)
{
  pari_sp av = avma;
  GEN n = gprec_extend(n0), S;
  long prec = (long)E;
  S = gsub(glngamma(gsub(n, ghalf), prec), glngamma(gaddgs(n, 1), prec));
  return gc_upto(av, gexp(gmul2n(S, 1), prec));
}
```

The Library function `gprec_w` (`w` is for *word*) changes the accuracy of a `GEN`, and is roughly the equivalent of the `GP` function `precision`. Here we are sure that when `n` is not exact its accuracy will increase, but in general there is also the function `gprec_wensure` which can increase but never decrease the accuracy of its argument.

The calling program will then simply be:

```
GEN
mysum(long prec)
{
  pari_sp av = avma;
  GEN R = real_i(sumnum((void*)prec, _auxS, gen_0, NULL, prec));
  return gc_upto(av, gdiv(R, Pi2n(2, prec)));
}
```

Note that `sumnum` may (and in this case does) return a `t_COMPLEX`, so we take the real part using `real_i` since we know that the result is a real number.

Note also that for this example the function `sumnumonien` (`sumnummonien0` in the Library) works directly and much faster.

14 Closures

As you may know, `closures` exist in `Pari`, for instance under `GP` you can write `f = (x->sin(x^2+1))`, and `f` will then be a closure, which can primarily first be *evaluated* (such as `f(Pi)`), and second given as an argument of another function since it is an ordinary Pari object, hence in the Library as a `GEN`.

Let us take an example. I want to define a new summation function `riemannsum` which for a given function f and an integer N computes $(1/N)\sum_{1\le n\le N} f(n/N)$, which would be a rough Riemann sum approximation of $\int_0^1 f(t)\,dt$. I could write in `GP`:

```
riemannsum(F,N)=sum(n=1,N,f(n/N),0.)/N;
```

To program this in the Library we can write the following:

```
GEN
```

```
riemannsum(GEN F, long N, long prec)
{
  pari_sp av = avma;
  GEN S = real_0(prec);
  long n;
  for (n = 1; n <= N; n++)
    S = gadd(S, closure_callgen1prec(F, sstoQ(n, N), prec));
  return gc_upto(av, gdivgs(S, N));
}
```

The function `closure_callgen1prec` assumes that `F` is a closure with a single `GEN` argument and needing a `prec` argument to know the precision with which it is going to do the computation (there of course exist more general closure calls for more or less arguments and with or without `prec`).

Once `installed` in `GP` with `install(riemannsum,GLp)` it can be used. But if I want to use it in the Library, I need an additional function. Assume for instance that the function `F` is the function $\sin(ax)$, which would in `GP` be written `fun = ((a, x)->sin(a*x));` (of course `fun(a, x)=sin(a*x)` would also work, but the previous notation emphasizes the fact that `fun` will be used as a closure). We write the following:

```
GEN
fun(GEN x, GEN a, long prec)
{ return gsin(gmul(a, x), prec); }

GEN
riemanntest(GEN a, long N, long prec)
{
  if (!a) a = mppi(prec);
  return riemannsum(strtoclosure("fun", 1, a), N, prec);
}
```

Note: it is essential to put the auxiliary argument(s) (here `a`) *after* the main variable(s) (here `x`), so it would be completely wrong to write `GEN fun(GEN a, GEN x, long prec)` (note: maybe not in this very special case since $ax = xa$, but you get my point!).

The function `strtoclosure` takes as first argument a string which is the name of the function, as second argument a small integer which is the number of auxiliary arguments needed by the function, followed by these arguments in the same order. Here we have 1 auxiliary argument, `a`.

Note that even though `fun` is not supposed to be called from outside, it must not be declared `static`, and must be declared somewhere, i.e., in some header file (usually `paripriv.h`) if it is going to be permanently in the library, or `installed` together with `riemanntest` if the latter is going to be used from `GP`. Here we added an extra quirk, we allow `a` to be

omitted, so that in GP the install commands would be `install(fun,GGp);` `install(riemanntest,DGLp);` (where DG means that the first GEN argument can be omitted), so that you can write if you like `riemanntest(, 100);` (just for fun).

There exist other functions related to closures such as `strtofunction` in case there is no auxiliary argument, and I refer you to the Library manual for more explanations.

We are now going to combine the `riemannsum` program with `limitnum` whose syntax is very similar to `sumnum`. For this, we note that for "nice" functions F, the expression

$$\int_0^1 f(x)\,dx - \frac{1}{N}\left(\frac{f(0)}{2} + \sum_{1\le n\le N-1} f(n/N) + \frac{f(1)}{2}\right)$$

tends to 0 with an asymptotic expansion in *even* powers of $1/N$, so that we can use the function `limitnum` with parameter $\alpha = 2$ (see the documentation under GP to see the meaning of α).

We thus write the following:

```
static GEN
_auxriemann(void *E, GEN gN, long prec)
{
  ulong N = itou(gN);
  GEN F = (GEN)E;
  GEN f0 = closure_callgen1prec(F, gen_0, prec);
  GEN f1 = closure_callgen1prec(F, gen_1, prec);
  GEN S = gdivgs(gsub(f0, f1), 2*N);
  return gadd(S, riemannsum((GEN)E, N, prec));
}

GEN
intnumriemann(GEN F, long prec)
{ return limitnum((void*)F, _auxriemann, gen_2, prec); }
```

This program computes quite accurately the integral from 0 to 1 of nice functions. As an exercise, you can modify all of the above so that it computes the integral on any compact interval $[a, b]$. Of course, the `intnumxxx` programs of `Pari/GP` are much more general and much more efficient, at least once their initialization is done.

15 Parallelism

Thanks to Bill Allombert, one of the most powerful features of `Pari/GP` is the possibility to use parallelism in an essentially trivial way. Let us see how

this is done in the Library. As a first simple example, we will again use the search for Wolstenholme primes, which is typically something that one can do in parallel.

Instead of the iterator `forprime`, we have a parallel iterator of course named `parforprime`, but note that there is no specific `ulong` version. It is used in essentially the same way as `forprime` itself, as follows:

```
void
pardowol(long lim)
{
  pari_sp av = avma;
  parforprime_t S;
  GEN gpres;
  GEN worker = strtoclosure("iswolstenfast", 0);
  parforprime_init(&S, stoi(11), stoi(lim), worker);
  while((gpres = parforprime_next(&S)))
    /* gpres will contain [p,iswolstenfast(p)] as GENs */
    if(!gequal0(gel(gpres, 2))) pari_printf("p = %Ps\n", gel(gpres, 1));
  set_avma(av);
}
```

Several things to note about this example: first, we would have liked to write directly `iswolstenfast` as the last argument of `parforprime_init`. Unfortunately, it requires a `GEN` and not a function. Thus we use the construct `strtoclosure` followed by the name of the function as a string as above, which does exactly what we want: it transforms a function into a `GEN` of type `t_CLOSURE`. Second, since we `install`ed `iswolstenfast` with the prototype `lL`, the closure knows that both the input and the output are `long` (or equivalent such as `ulong` or `int`), and does the appropriate conversions, both on input (transforming the prime, which is a `GEN`, into a `long` using `itos`), and the output (transforming the `int` output into a `GEN` using `stoi` or an equivalent).

Next, note that the 0 which follows the name means that `iswolstenfast` does not need extra data (we will see a slightly more complicated example below).

Finally, note that `parforprime_next` returns a two component vector, the first being the prime, and the second, the evaluation of the given function at that prime, both given as `GEN`s.

Exercise. Install the necessary functions in `GP`, and execute `pardowol(20000)` which should take less than a second and give you the first Wolstenholme prime, then `pardowol(3*10^6)` which will give you the only two known such primes in 5 or 10 minutes depending on your processor and number of threads (assuming of course that you have at least 8 threads, otherwise it will be much longer).

But one can use parallelism more generally without using preinstalled parallel iterators. In that case, the function which will be executed in parallel will be called a `worker`, and we suggest to name all the auxiliary functions used as workers `_myfunction_worker`, so as to make clear (with the initial underscore) that it is an auxiliary function, and (with the ending `_worker`) that it will be used in some parallel program. A typical example is the use of `parsum`, parallel summation, whose library prototype is `GEN parsum(GEN a, GEN b, GEN code)`. For instance, assume that we want to compute as a real number $Z(s,N) = \sum_{1\le n\le N} 1/n^s$ for some complex number s, the partial sum of the Riemann zeta function (this already exists in the library as `dirpowerssum`, but we want to program it from scratch in a naive but parallel way). In `GP` you can simply write `sum(n=1,N,1/n^s,0.)` (the `0.` at the end being necessary if s is an integer), which of course translates trivially in C. But if N is really large, you want to do this in parallel. You can write the following as a worker, which computes the summand:

```
GEN
_parsumpow_worker(GEN gn, GEN s, long prec)
{
  return gpow(gn, gneg(s), prec);
}
```

In the above, the first argument `gn` (which must be a `GEN`, not a `long`) is going to range through the integers `n` as a `t_INT` (in other words `gn=utoi(n)`). Before writing the parallel program using this `worker`, a word about identifiers starting with an underscore. If you try to install the above function in `GP` using `install`, even putting it between double quotes, you will get an error message telling you that `_parsumpow_worker` is not a valid identifier. You must thus give it in addition a valid `GP` name not beginning with an underscore, for instance:

```
install("_parsumpow_worker",GGL,parsumpow_worker);
```

(of course you do not need all these complications if you do not give a name beginning by an underscore).

You can now write the parallel summation using the function `strtoclosure`, which transforms a C function given by its `GP` name (NOT it's C name, see above) as a string into a `GEN` of type `t_CLOSURE`, which will be a closure in `Pari`'s sense.

```
GEN
Z(GEN s, long N, long prec)
{
  pari_sp av = avma;
  GEN worker = strtoclosure("parsumpow_worker", 2, s, utoi(prec));
  GEN S = parsum(gen_1, utoi(N), worker);
```

```
  return gc_GEN(av, S);
}
```

The function `strtoclosure` is easy to use: as first argument you give the worker function name as a string, as second argument the number of auxiliary arguments (do *not* count the summation index `n` or `gn`), here two, and finally the list of arguments in the same order as they appear in the worker, after the summation index.

Important note: you *must* declare the worker to be accessible from any part of the Library. If it is going to be permanent, declare it in a header, and *not* in `paridecl.h` where you declare all the functions which must be known to `GP`, but in the file `paripriv.h`, which is more private. If it is not going to be permanent and you want to use the function `Z` inside `GP`, you must `install` it as explained above.

Technical remark: there is a more efficient way of creating the worker, which essentially avoids unnecessary copies, which uses the `snm_closure` function together with `is_entry`. I do not explain its use, but simply mention that in the above program you can replace the line which creates the worker by the following:

```
  GEN worker = snm_closure(is_entry("_parsumpow_worker"),
                                        mkvec2(s, utoi(prec)));
```

In practice it makes essentially no difference, so for simplicity I suggest only using `strtoclosure`.

You may have noticed that there is a slight omission in the above program: we want to initialize the sum to `0.` and not to `0` so as to avoid computing with rational numbers with huge denominators, and `parsum` does not allow this. A simple possibility is to modify the worker as follows:

```
GEN
_parsumpow_worker(GEN gn, GEN s, long prec)
{ return gequal1(gn) ? real_1(prec) : gpow(gn, gneg(s), prec); }
```