Computing one million digits of racine de 2

Xavier Gourdon, Bruno Salvy

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Computing one million digits of $\sqrt{2}$

Xavier GOURDON
Bruno SALVY

N°155
Mai 1993

PROGRAMME 2

Calcul symbolique
Programmation
Génie logiciel

Rapport technique

Computing one million digits of $\sqrt{2}$

Xavier Gourdon and Bruno Salvy

Abstract

We describe how the knowledge of Maple's internal data structure can be used to compute the first million digits of $\sqrt{2}$ efficiently.

Calcul du premier million de décimales de $\sqrt{2}$

Résumé

Nous montrons comment la connaissance des structures de données internes de Maple peut être utilisée pour calculer efficacement le premier million de décimales de $\sqrt{2}$.

To appear in *The Maple Technical Newsletter.*

Computing one million digits of $\sqrt{2}$

Xavier Gourdon and Bruno Salvy [1]

Introduction

Maple can be used as a high precision calculator to effectively compute a large number of digits of some constants. For example, an interesting survey of the computation of π with Maple can be found in the 8th issue of this newsletter. The most natural way to proceed to compute a constant C is to call the function `evalf(C,n)` which returns the first n digits of C. However, when n becomes large (n around 100,000), `evalf` becomes very time-consuming. In this note, we use the computation of the first million digits of $\sqrt{2}$ as an incentive to the understanding of Maple's floating point numbers. Our aim is to show that this understanding is profitable to gain much efficiency in the computations. The program we describe computes the first million digits of $\sqrt{2}$ in about 3.5 c.p.u. hours on a Sparc10. Here are some timings comparing the speeds of our algorithm and of Maple's evalf.

Nb digits	our time	evalf	ratio
4096	1.1s	4.8s	4.3
8192	3.5s	19.2s	5.4
16384	11.3s	77.8s	6.9
32768	35.4s	311.3s	8.8
65535	111.5s	1266.6s	11.3

In the second column, our time is multiplied by 3 each time the number of digits doubles, hence a growth of $O(n^{\log 3/\log 2})$, while evalf's time grows in $O(n^2)$. The time needed by evalf to compute one million digits of $\sqrt{2}$ would thus be roughly 80 c.p.u. hours on this machine (to be compared with our 3.5 hours).

Since all the computations of `evalf` are performed in the kernel, which is in compiled C, they cannot be overtaken by mere Maple programming. Our improvements over `evalf` thus have to be algorithmic. These fall into two classes: improvement of the algorithm used for the computation of $\sqrt{2}$ itself and improvement of the underlying arithmetic. Also, Maple integers (hence floats) are limited to $2^{19} = 524288$ digits on a 32-bits machine. To overcome this limitation, the final program will output two numbers whose concatenation yields the right result.

Computation of $\sqrt{2}$

The most well known algorithm to compute $\sqrt{2}$ consists in iterating the sequence

$$u_{n+1} = u_n/2 + 1/u_n, \quad u_0 = 3/2, \tag{1}$$

[1] Algorithms Project, INRIA, 78153 Le Chesnay Cedex, France.
`Xavier.Gourdon@inria.fr` and `Bruno.Salvy@inria.fr`

obtained by applying Newton's method to the polynomial x^2-2. This sequence converges to $\sqrt{2}$ quadratically, which means that the number of correct digits doubles at each iteration. This follows from substituting $u_n = \sqrt{2}+\ell_n$ in (1) and rewriting $1/u_n$ as $(\sqrt{2}-\ell_n)/2+\frac{1}{2}\ell_n^2/(\sqrt{2}+\ell_n)$, which yields

$$\ell_{n+1} = \frac{1}{2}\frac{\ell_n^2}{\sqrt{2}+\ell_n}.$$

From this one notes that this convergence is also *stable*, meaning that a small error at one stage will not prevent the sequence from converging. This property can be used profitably by computing only 2^n digits of u_n; or in Maple parlance by doubling `Digits` at each step.

A shortcoming of (1) is that it requires the computation of the inverse of numbers with a large number of digits. This is not a necessity and we shall rather rely on the following sequence

$$u_0 = 0.7071067811865475, \qquad u_{n+1} = \frac{3}{2}u_n - u_n^3. \tag{2}$$

This sequence is obtained by applying Newton's method to $2-\frac{1}{x^2}$ and by similar arguments as above, it converges quadratically to $\sqrt{2}/2$. More precisely, one can prove that $|u_n - \sqrt{2}/2| \le 10^{2^{n+4}}$ for all non negative integer n. Hence the first million digits of $2\,u_{16}$ (more precisely the first $2^{20} = 1,048,566$ digits) are those of $\sqrt{2}$.

The advantage of (2) over (1) is that it requires only additions, products and divisions by 2. Addition and division by 2 of large numbers (numbers with thousands of digits) are not expensive in Maple, but multiplication is. This is discussed in our next sections.

Large integers and floats in Maple

Floating point numbers are stored by Maple as

$$\texttt{Float}(a,b),$$

where a and b are integers. This represents $a\cdot 10^b$. Then, except when `Digits` is smaller than `evalhf(Digits)`, all the operations performed on floating point numbers are reduced internally to operations on integers.

Consider now the division by 2 which is needed in (2), and suppose `Digits` = 20000. What happens is that 1/2 is first computed to 20000 digits, and then Maple computes the product of two 20000 digits integers. To avoid this waste of time, it is therefore desirable to transform (2) into a recurrence over integers:

$$u_0 = 7071067811865475, \qquad u_{n+1} = \frac{3u_nL_n}{2} - \frac{u_n^3}{L_n}, \tag{3}$$

where $L_n = 10^{2^{n+4}}$, and the divisions in (3) are considered as Euclidean divisions computed by `iquo`. Division and multiplication by L_n are not difficult to compute because Maple stores its integers in base 10000 (on a 32-bit machine). Therefore these operations reduce to shifts. We show in our next section how this can be done efficiently.

To conclude this section, this is the procedure we use to compute $\sqrt{2}$ with a large number of digits:

```
# n is the number of digits. This procedure works for n < 2^18 = 262144 digits.
sqrt2 := proc(n)
local u, l;
   u:=7071067811865475;
   l:=4;
   while 8*l < n do
         u:=iquo(3*shift(u,l),2)-trunccube(u,l);
         l:=2*l;
   od;
   RETURN(evalf(Float(3*shift(u,l)-2*trunccube(u,l),-8*l),n))
end:
```

The procedures `shift` and `trunccube` are detailed below.

Shifting integers in Maple

Although Maple stores its integers in base 10000 (on a 32-bit machine), the product of an integer by 10^{4n} takes the same time as the product by any $4n$-digits integer. It turns out that using Maple's internal data structure, the cost of such operations can be considerably reduced. This is achieved thanks to Maple's "hackware package" commands: `assemble`, `disassemble`, `pointto`, `addressof`.

We illustrate these commands on an example. We take the integer `n:=123456789`. Calling the function `ptrn:=addressof(n)` returns the internal address of the Maple integer `n`. The `disassemble` function, applied to the address `ptrn`, disassembles the Maple integer `n` into its components and returns the sequence of their addresses:

```
> n:=123456789:
> ptrn:=addressof(n):
> addrseqn:=disassemble(ptrn);
                        addrseqn := 2, 6789, 2345, 1
```

The first integer in `addrseqn` represents the type of the object `n` (here a positive integer), the following ones are the 4-digits blocs of `n` in reverse order. Suppose we want to calculate `m=n*10000`. The Maple representation of `m` being

```
                           2, 0, 6789, 2345, 1
```

we compute

```
> m:=pointto(assemble(2, 0, subsop(1=NULL,addrseqn)));
                            m := 1234567890000
```

The `assemble` function assembles the sequence of integers into a Maple object and returns the address of this object. Calling then the function `pointto` returns the Maple expression to which it points, here `m`. Thus multiplying or dividing an integer by 10^{4n} is reduced to shift operations on the n 4-digits blocs of this integer, and these are very fast.

The code that does it is the following:

```
# If a macro could have an argument, this would be a macro.
inttolist:=proc(n) RETURN([disassemble(addressof(n))]) end:
```

```
# This returns a · 10^(4k)
shift:=proc(a,k)
   if a=0 then RETURN(0) fi;
   RETURN(pointto(assemble(op(subsop(1=(2,0$k),inttolist(a))))))
end:
```

Fast multiplication

We now concentrate on the product of two large integers, which is the most expensive operation in this algorithm. Maple computes the product of two n-digits integers like one does by hand (except that it does it in base 10000). This requires $O(n^2)$ elementary operations. This cost can be reduced to $O(n^{\log 3/\log 2})$ operations using Karatsuba's algorithm. To multiply two $2n$-digits integers A and B, Karatsuba's algorithm consists in cutting A and B into two parts,

$$A = a_1 L + a_0, \qquad B = b_1 L + b_0$$

where $L = 10^n$, a_1, a_0, b_1 and b_0 being n-digits numbers, and then to compute the product AB as

$$AB = (a_1 b_1)L^2 + [(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0]L + a_0 b_0. \tag{4}$$

Only three products of n-digits numbers have to be computed instead of four with the usual formula $AB = (a_1 b_1)L^2 + (a_0 b_1 + a_1 b_0)L + a_0 b_0$. A recursive application of (4) leads to the bound $O(n^{\log 3/\log 2})$.

As seen in the previous section, cutting integers and multiplying them by $L = 10^n$ can be efficiently achieved using Maple's internal data structures. Some care is required though, since the representation of an integer cannot end by a zero. Consider for instance the integer `n:=1234500006789` and suppose we want to compute `m`, the last 8 digits of `n`. We compute

```
> l1:=disassemble(addressof(n));
                        l1 := 2, 6789, 0, 2345, 1
> l2:= 2, op(2..3,[l1]);
                           l2 := 2, 6789, 0
> m:=pointto(assemble(l2));
                                m := 0
```

This means that we have to test the last 4-digits blocs of the representation of `m` and delete them if they are zero, before calling the function `pointto(assemble())`. Taking this into account, we write the following procedure:

```
# Returns both the first and last part, assigning them to the last arguments
firstlastpart:=proc(n,k,first,last)
local l, i;
   l:=inttolist(n); # this is slightly non-linear (hence expensive)
   if k+2>nops(l) then first:=0;last:=n;RETURN() fi;
   first:=pointto(assemble(op(1,l),op(k+2..nops(l),l)));
   l:=[op(2..k+1,l)];
   for i from k by -1 to 1 while op(i,l)=0 do od;
```

```
    if i = 0 then last:=0 else last:=pointto(assemble(2,op(1..i,l))) fi;
    RETURN()
end:
```

Using this and our previous routines, Karatsuba's algorithm fits in five lines:

```
# n × m by Karatsuba's algorithm (n ≥ m)
karaprod:=proc(n,m)
local k, k2, a, b, c, d, ac, bd;
    if length(n)<6000 or length(m)<2000 then RETURN(n*m) fi;
    k:=ceil(length(n)/4); k2:=iquo(k,2);
    firstlastpart(n,k2,a,b); firstlastpart(m,k2,c,d);
    ac:=karaprod(a,c); bd:=karaprod(b,d);
    RETURN(shift(ac,2*k2)+shift(karaprod(a+b,c+d)-ac-bd,k2)+bd)
end:
```

Note that the practical efficiency of this algorithm depends on the value of the thresholds where we let the underlying arithmetic (here Maple's internal multiplication) compute the products. In practice, this program is more efficient than Maple's product as soon as the integers have more than about ten thousand digits.

Computation of u_n^3

Now that we have an efficient multiplication, it is a simple matter to compute u_n^3 with two multiplications. However, a few shortcuts can be used to make this computation even faster.

First, when $n = m$, the splitting `firstlastpart` in `karaprod` can be done once instead of twice. One can use this to write a second procedure `karasqr` which is slightly faster than `karaprod` in this special case.

Second, in our algorithm u_n^2/L_n is close to $L_n/2$. This can be taken into account to spare one long multiplication (u_n^2 by u_n): once we have computed $z = u_n^2$, we cut z into two blocs $z = u_n^2 = AL_n + B$, and rewrite it as $z = (A - L_n/2)L_n + B + L_{n+1}/2$. We now have three products to examine

$$(i) \quad (A - L_n/2) \times u_n, \quad (ii) \quad B \times u_n \quad \text{and} \quad (iii) \quad L_{n+1}/2 \times u_n.$$

Product (i) is quickly achieved, because $A - L_n/2$ is a short integer; product (iii) is not costly since we only have to shift and divide by 2 the integer u_n. This way, there is only one large product (ii) to calculate instead of two ($A \times u_n$ and $B \times u_n$).

This leads us to the following program:

```
# Compute the first 2k blocs of n^3, using the fact that n^2 is close to 1/2.
# k is the number of blocs of n (it is a power of 2)
trunccube:=proc(n,k)
local u, v;
    firstlastpart(karasqr(n),k,u,v);
    RETURN(iquo(shift(n,k),2)+firstpart(karaprod(n,v),k)+(u-shift(5000,k-1))*n)
end:
```

In this procedure, `firstpart` is an analogous to `firstlastpart` which returns only the first half of the number.

Huge numbers

Because of the limit size of the integers in Maple, we have to avoid creation of integers with more than 524288 digits. Thus, in the actual program, we use two extra procedures to compute the last two terms of the sequence. These procedure are rather long and unilluminating and we omit them here. Basically, one has to cut the numbers into several pieces, perform the "small" products avoiding numbers with more than 524288 digits in the intermediate computations, propagate the carry and reuse the local variables to spare memory. These procedures can be obtained by e-mail from the authors.

Conclusion

We expect that this experience convinces the reader that being an interpreted language is not an obstacle to the speed of such computations in Maple. A program written in C would not have been really more efficient than ours; moreover, it would have required big numbers packages and would have been more tedious to write. Nevertheless, some fast product algorithm should be implemented in Maple's kernel. The program would have been 10 lines long. While an FFT-based product will be efficient only for very large numbers of digits (about one million) and thus is not absolutely necessary, Karatsuba's algorithm, if implemented in the kernel, would be faster than the existing product for numbers of about 300 digits, which is a common size for coefficients appearing in large computations.