Comparison of three public-domain multiprecision libraries: BigNum, Gmp and Pari

Paul Zimmermann
Paul.Zimmermann@inria.fr

February 12, 1998

Abstract. We describe the use of three multiprecision libraries on two sample problems: compute digits of π and compute a recursive sequence of integers. Our conclusion is that (i) a multiprecision library is about ten times faster than a general computer algebra system on specific problems and that (ii) the smaller program is usually the faster one.

A lot of tools are nowadays available for the mathematician or computer scientist. There are two main families: general computer algebra systems on the one hand, and other systems, designed for specific applications, on the other hand. The main advantages of general computer algebra systems like MACSYMA, MAPLE, MATHEMATICA, SCRATCHPAD are:

- you have *all in one*,
- they are interactive,
- they are easy to program.

Besides the fact they are not cheap, general computer algebra systems are usually not very efficient for large problems, first because they use general data structures that require a lot of memory space, secondly because the programs are interpreted and not compiled.

Therefore in the last few years several people have developed special-purpose applications. Most of these packages are public-domain and thus easily available. But they are not enough widely used, perhaps because the potential users think they are too difficult to program. With the example of multiprecision libraries written in the C language, we prove that using special-purpose systems is not that difficult, and moreover may be very efficient.

For our comparison, we have chosen the following multiprecision libraries: the BIGNUM package developed jointly by people at Inria and Dec Prl (version from July 1991); GMP or GNU MP, version 2.0.2, the package of the Free Software Foundation developed by T. Granlund, TMG Datakonsult; and PARI, version 1.36, the library developed by people at the University of Bordeaux, France, which is the basis of the GP system. These three libraries were also compared to the Maple system, version V.3. The compilers and optimization levels used are as follows (for GMP, the `-O3` level gives a significant speedup on the R8000 with respect to `-O2`, but only a small speedup on the DecAlpha and nothing on the UltraSparc):

	SGI R8000 90 Mhz	Dec Alpha 500 333 Mhz	Sparc Ultra 170 167 Mhz
GMP 2.0.2	cc -O3	gcc -O3	gcc -O3

Please note that the UltraSparc does not produce true 64-bit code, but 32-bit code. We have chosen three simple test-problems that can be directly solved using the functions provided by each library.

Domain	BigNum	GMP	Pari
$\mathbb{Z}$	•	•	•
$\mathbb{Q}$		•	???
$\mathbb{R}$		•	???
$\mathbb{C}$			???
modular			???
intervals			

Figure 1: Domains available in each library.

Operation	BigNum	GMP	Pari
$a+b$	•	•	•
$a \times b$	•	•	•
a/b	•	•	•
gcd	???	•	???
random	???	•	???

Figure 2: Operations available in the $\mathbb{Z}$ domain.

Operation	GMP	Pari
$a+b$	•	•
$a \times b$	•	•
a/b	•	•
random	???	???
FFT		
exp		
log		
sin, cos, tan		
sinh, cosh, tanh		
asin, acos, atan		
asinh, acosh, atanh		
ζ		
Γ		

Figure 3: Operations available in the $\mathbb{R}$ domain.

1 Available domains and operations

2 Digits of π

Our first test is the computation of the constant π to several digits, by the means of the following formula[1], due to Gauss [3]

$$\pi = 48 \arctan \frac{1}{18} + 32 \arctan \frac{1}{57} - 20 \arctan \frac{1}{239}.$$

The arctan function will be computed by its well-known series expansion around zero:

$$\arctan \frac{1}{n} = \frac{1}{n} - \frac{1}{3n^3} + \frac{1}{5n^5} - \frac{1}{7n^7} + \cdots$$

Instead of computing with real numbers, we compute an integer approximation to $10^N \pi$, and thus work only with big integers. The algorithm, written in Maple, is as follows:

```
main := proc()
    N := readstat(`N=`);
    S := 0;
    Arctg(18,48,1); Arctg(57,32,1); Arctg(239,20,-1);
    S
end:

Arctg := proc(n,c,eps)
    X := iquo(c*10^N,n);
    S := S + eps*X;
    for i from 1 while X>0 do
        X := iquo(X,n^2);
        Y := iquo(X,2*i+1);
        if i mod 2 = 1 then S := S - eps*Y
        else S := S + eps*Y fi;
    od;
end:
```

where the function $\mathrm{Arctg}(n, c, \epsilon)$ adds $\epsilon\, c \arctan(1/n)$ to the variable S (the above Maple description is just a sketch for the C programs that will follow, not a beautiful program).

[1] This formula was used by J. Guilloud and M. Bouyer in 1976 to compute for the first time $1,000,000$ digits of π. This computation was done on a CDC 7600 in 22 hours and 11 minutes.

With Gmp. The following C program uses the GMP library to compute N digits of π. The type `MP_INT` stands for *multiprecision integer*. Each variable of this type must be initialized by the function `mpz_init`. All functions take as arguments pointers to `MP_INT`s, so we have to put the character & before each variable.

```
#include <stdio.h>
#include "gmp.h"

MP_INT S,X,Y;
int N,i;

main()
{
    printf("N="); scanf("%d",&N);
    mpz_init(&S); mpz_init(&X); mpz_init(&Y);
    Arctg(18,48,1);
    Arctg(57,32,1);
    Arctg(239,20,0);
    mpz_out_str(stdout,10,&S); putchar('\n');
}

Arctg(n,c,b)
int n,c,b;
{
    mpz_set_ui(&X,10);
    mpz_pow_ui(&X,&X,N);
    mpz_mul_ui(&X,&X,c);
    mpz_div_ui(&X,&X,n);
    if (b) mpz_add(&S,&S,&X);
    else mpz_sub(&S,&S,&X);
    for (i=1;mpz_cmp_ui(&X,0)>0;i++) {
        mpz_div_ui(&X,&X,n*n);
        mpz_div_ui(&Y,&X,2*i+1);
        if (i%2==b) mpz_sub(&S,&S,&Y);
        else mpz_add(&S,&S,&Y);
    }
}
```

With GMP, the programmer does not have to worry about the size of the objects or their allocation. All operations can be done in place (for example `mpz_pow_ui(&X,&X,N)` replaces X by X^N).

With Pari. Here the program is not so simple because the garbage collection is not automatic in Pari. In fact, there are two ways to program: either use in-place operations ending with the letter `z`, or use operations that put their result on the Pari stack. In the first case, the programmer must compute the size needed for each operation; in the latter case, he must clean the Pari stack from time to time, otherwise he will get a message saying "the PARI stack overflows !!!". In both cases though, the programmer must define the size of the Pari stack with the function `init`.

```
#include <pari/genpari.h>

GEN S,X,Y;
int N,i,ltop,lbot;

main()
{
    printf("N="); scanf("%d",&N);
    init(500000,2); ltop = avma;
    S = gzero;
    Arctg(18,48,1);
    Arctg(57,32,1);
    Arctg(239,20,0);
    brute(S,'g',-1); putchar('\n');
}

Arctg(n,c,b)
int n,c,b;
{
    X = gpuigs(stoi(10),N);
    X = mulsi(c,X);
    X = divis(X,n);
    if (b) S = addii(S,X); else S = subii(S,X);
    for (i=1;signe(X)>0;i++) {
        X = divis(X,n*n);
        Y = divis(X,2*i+1);
        lbot = avma;
        if (i%2==b) S = subii(S,Y);
        else S = addii(S,Y);
        S = gerepile(ltop,lbot,S);
    }
}
```

We have chosen here the second way of programming in PARI, with the stack. The garbage collection is done by the function `gerepile`, which is described in the Pari manual [1, p. 69].

With BigNum. There is no garbage collection provided with the BigNum library, whence the programmer has to use in-place operations and must carefully determine the size of the variables to define. Here for example, one has to compute the number l of 32-bit digits occupied by the integers S, X and Y. In addition, there is no power function in the package, therefore we had to program it.

```
#include <stdio.h>
#include <math.h>
#include <bignum/BigNum.h>
#include <bignum/BigZ.h>

#define new(l,t) malloc((l)*sizeof(t))
#define BN(l) (BigNum) new(l,BigNumDigit)

BigNum S,Y,X;
int N,l;

Power(T,a,n)
BigNum T;
int a,n;
{
    int i,tl,logn;

    BnnSetToZero(T,l); BnnSetDigit(T,a);
    logn = (int)(log((float)n)/log(2.0));
    for (i=logn-1; i>=0; i--) {
        tl = BnnNumDigits(T,l);
        BnnSetToZero(Y,2*tl);
        BnnMultiply(Y,2*tl,T,tl,T,tl);
        BnnAssign(T,Y,2*tl);
        tl = BnnNumDigits(T,l);
        if ((n>>i)%2)
            BnnMultiplyDigit(T,l,T,tl,a-1);
    }
}

main()
{
    printf("N="); scanf("%d",&N);
    l = 1+(int)N*log(10.)/log(2.)
      /BN_DIGIT_SIZE;
    Y = BN(l+1); X = BN(l+2); S = BN(l+1);
    BnnSetToZero(S,l+1);
    Arctg(18,48,1);
    Arctg(57,32,1);
    Arctg(239,20,0);
    printf("%s\n",
      BzToString(BzFromBigNum(S,l),10));
}

Arctg(n,c,b)
int n,c,b;
{
    int i,xl;

    Power(X,10,N); BnnSetToZero(X+l,2);
    BnnMultiplyDigit(X,l+1,X,l,c-1);
    BnnDivideDigit(X,X,l+2,n);
    xl = l+1;
    if (b) BnnAdd(S,l+1,X,xl,0);
    else BnnSubtract(S,l+1,X,xl,1);
    for (i=1;!BnnIsZero(X,xl);i++) {
        BnnSetToZero(X+xl,1);
        BnnDivideDigit(X,X,xl+1,n*n);
        xl = BnnNumDigits(X,xl);
        BnnSetToZero(X+xl,1);
        BnnDivideDigit(Y,X,xl+1,2*i+1);
        if (i%2==b) BnnSubtract(S,l+1,Y,xl,1);
        else BnnAdd(S,l+1,Y,xl,0);
    }
}
```

In this program, we have used at several places the trick that `BnnMultiplyDigit(X,l+1,X,l,c-1)` multiplies in place the BigNum X by c (it replaces X by $X + (c-1)X$).

Figure 4 gives the number of seconds used by each program to compute N digits of π, on three different machines.

Remark: the above programs work only for "small" N, because we have used the division of a multi-precision integer by a small integer (`mpz_div_ui` in Gmp, `divis` in Pari, `BnnDivideDigit` in BigNum). More precisely, in the function `Arctg`, the number $2i+1$ should not exceed the capacity of the C-type `unsigned long`. On a 32-bit machine, the limit for N is about 5 billions.

R8000				
N	Maple	BigNum	Gmp	Pari
1000	0.9		0.1	
10000	67		13	
100000	9490		1337	
DecAlpha 500/333				
N	Maple	BigNum	Gmp	Pari
1000	0.2		0.02	
10000	11		1.0	
100000	1146		92	
UltraSparc				
N	Maple	BigNum	Gmp	Pari
1000	0.9		0.1	
10000	70		11	
100000	10034		1092	

Figure 4: Time needed to compute N digits of π.

3 A recursive sequence

This second test was suggested to me by J.-M. Steyaert. The problem is to compute the sequence (C_n) of rational numbers defined recursively by

$$C_0 = 1, \quad C_n = \frac{1}{n^2} \sum_{j=0}^{n-1} C_j C_{n-1-j} \text{ for } n \geq 1, \qquad (1)$$

up to $n = 1000$. The origin of these numbers is that C_n is the probability that two binary search trees of n nodes each have the same shape (you obtain the same binary trees when you replace labels by unlabelled nodes). The function $C(z) = \sum C_n z^n$ is a solution of the differential equation

$$zC'' + C' = C^2.$$

The numbers C_n are hard to compute: from formula (1) and with classical algorithms, the computation of C_n up to $n = N$ costs $\Theta(N^4 \log^2 N)$ basic operations. Thus we really need a *fast* program to reach $N = 1000$.

Instead of computing C_n, we will compute the integers $B_n = (n!)^2 C_n$ that satisfy the recurrence relation

$$B_0 = 1, \quad B_n = \sum_{j=0}^{n-1} \binom{n-1}{j}^2 B_j B_{n-1-j}$$

using the following algorithm (written in Maple):

```
main := proc(N)
local s,n,j,p;
    B(0) := 1:
    for n to N do
        s := 0;  p := 1;
        for j from 0 to n-1 do
            s := s + p*B(j)*B(n-1-j);
            p := p * (n-1-j)^2/(j+1)^2;
        od;
        B(n) := s;
    od;
end:
```

The variable s contains the partial sum up to j, and p is the square of the current binomial coefficient $\binom{n-1}{j}$.

With Gmp. The very short program is the direct translation of the algorithm written in Maple:

```
#include <stdio.h>
#include "gmp.h"

main()
{
    int N,n,j;
    MP_INT *B,P,T;

    printf("N="); scanf("%d",&N);
    B = (MP_INT*)
        malloc((N+1)*sizeof(MP_INT));
    mpz_init(B); mpz_set_ui(B,1);
    mpz_init(&P); mpz_init(&T);
    for (n=1; n<=N; n++) {
        mpz_init(B+n); mpz_set_ui(B+n,0);
        mpz_set_ui(&P,1);
        for (j=0; j<n; j++) {
            mpz_mul(&T,B+j,B+(n-1-j));
            mpz_mul(&T,&P,&T);
            mpz_add(B+n,B+n,&T);
            mpz_mul_ui(&P,&P,(n-1-j)*(n-1-j));
            mpz_div_ui(&P,&P,(j+1)*(j+1));
        }
    }
```

```
        mpz_out_str(stdout,10,B+N); putchar('\n');
    }
```

The auxiliary variable T is used to store the product of B_j by B_{n-1-j}, and then by P.

With Pari. The program is very short too,

```
#include <pari/genpari.h>

main()
{
    int N,n,j,ltop,lbot,dec;
    GEN *B,P,T;

    init(500000,2);
    printf("N="); scanf("%d",&N);
    B = (GEN*) malloc((N+1)*sizeof(GEN));
    B[0] = gun;
    for (n=1; n<=N; n++) {
        ltop = avma;
        B[n] = gzero;  P = gun;
        for (j=0; j<n; j++) {
            T = mulii(B[j],B[n-1-j]);
            T = mulii(T,P);
            P = mulis(P,(n-1-j)*(n-1-j));
            lbot = avma;
            P = divis(P,(j+1)*(j+1));
            B[n] = addii(B[n],T);
            dec = lpile(ltop,lbot,0)/4;
            B[n] += dec;  P += dec;
        }
    }
    brute(B[N],'g',-1);putchar('\n');
}
```

but the garbage collection is not as simple as in the previous section. The problem is that we have to keep more than one variable at each iteration, namely P and B_n. Thus a simple call to `gerepile` is no longer possible. However, the solution is given in the PARI manual [1, p. 70].

With BigNum. The program is longer here, as we must precompute the memory necessary to store the B_n and all auxiliary variables. For example the binomial coefficients are less than 2^N, thus their square needs $2N$ bits, and therefore the variable P must contain at least $\lceil 2N/\texttt{BN\_DIGIT\_SIZE}\rceil$ BigNumDigits.

```
#include <stdio.h>
#include <math.h>
#include <bignum/BigNum.h>
#include <bignum/BigZ.h>

#define new(l,t) malloc((l)*sizeof(t))
#define BN(l) (BigNum) new(l,BigNumDigit)
#define min(a,b) (((a)<(b)) ? a : b)

main()
{
  int N,n,j,*lb,lp,lt,ltmax;
  BigNum *B,P,T;

  printf("N="); scanf("%d",&N);
  B = (BigNum*) new(N+1,BigNum);
  lb = (int*) new(N+1,int); lb[0] = 1;
  B[0] = BN(1); BnnSetDigit(B[0],1);
  lp = 1+(2*N)/BN_DIGIT_SIZE; P = BN(lp+1);
  ltmax = 2 +
    (int)(2*N)/BN_DIGIT_SIZE*log(N*1.)/log(2.);
  T = BN(ltmax);
  for (n=1; n<=N; n++) {
    lb[n] = lb[n-1]+1+lp;
    B[n] = BN(lb[n]);
    BnnSetToZero(B[n],lb[n]);
    BnnSetToZero(P,lp); BnnSetDigit(P,1);
    for (j=0; j<n; j++) {
      lt = min(ltmax,lb[j]+lb[n-1-j]);
      BnnSetToZero(T,lt);
      BnnMultiply(T,lt,B[j],lb[j],B[n-1-j],lb[n-1-j]);
      BnnMultiply(B[n],lb[n],T,lt,P,lp);
      BnnMultiplyDigit(P,lp,P,lp,(n-1-j)*(n-1-j)-1);
      BnnDivideDigit(P,P,lp+1,(j+1)*(j+1));
      BnnSetToZero(P+lp,1);
    }
    lb[n] = BnnNumDigits(B[n],lb[n]);
  }
  printf("%s\n",BzToString(
          BzFromBigNum(B[N],lb[N]),10));
}
```

Figure 5 gives the number of seconds used to compute B_n up to $N = 100$ and $N = 200$.

4 Fermat's quotients

In this section we test the different libraries on small-size arithmetic, but where the precision of C long

R8000				
N	Maple	BigNum	Gmp	Pari
100	2.8	0.1		
200	28	0.6		
500	830	17		
DecAlpha				
N	Maple	BigNum	Gmp	Pari
100	0.5	0.03		
200	4.4	0.25		
500	87	8.5		
UltraSparc				
N	Maple	BigNum	Gmp	Pari
100	2.5	0.3		
200	33	5.0		
500	1256	247		

Figure 5: Time needed to compute $B_1 \dots B_N$.

R8000				
N	Maple	BigNum	Gmp	Pari
$F_2(10^6)$	5721	48		
$F_{3391}(2 \cdot 10^7)$		1227		
$F_{999959}(10^6)$	6002	52		
DecAlpha				
N	Maple	BigNum	Gmp	Pari
$F_2(10^6)$	1594	27		
$F_{3391}(2 \cdot 10^7)$		652		
$F_{999959}(10^6)$	1675	27		
UltraSparc				
N	Maple	BigNum	Gmp	Pari
$F_2(10^6)$	3551	56		
$F_{3391}(2 \cdot 10^7)$		1528		
$F_{999959}(10^6)$	3808	61		

Figure 6: Time needed to compute $F_p(q_{\max})$.

integers (64 bits) or of long floating-point numbers (96 bits) is not enough. The problem we choose is the computation of the set $F_p(q_{\max})$ defined by:

$$F_p(q_{\max}) = \{q \leq q_{\max}, p^{q-1} = 1 \bmod q^2\}.$$

Usually one wants q prime, but here for simplicity one considers all odd numbers. These sets were defined by Maurice Mignotte and Yves Roy [7] to prove that Catalan's equation $x^q - y^q = 1$ admits no solution for $\min(p, q) \leq 10^5$ apart from the trivial one $2^3 - 3^2 = 1$. For $p = 2$, Fermat's quotients base 2 are related to Wieferich primes. Namely, the set $F_2\infty$ contains all Wieferich primes, i.e. the prime numbers q such that $2^{q-1} = 1 \bmod q^2$. The only known Wieferich primes are 1093 and 3511. Lehmer found no other Wieferich prime up to $6 \cdot 10^9$ [6]. Richard Crandall, Richard McIntosh, Carl Pomerance and other people found no other Wieferich prime up to $4 \cdot 10^{12}$ [2]. The Maple program to compute $F_p(q_{\max})$ is the following:

```
F := proc(p,qmax) local q;
   for q from 3 by 2 to qmax do
      if Power(p,q-1) mod (q^2)=1
      then lprint(`p=`,p,`q=`,q) fi
   od
end:
```

With Gmp.

```
#include <stdio.h>
#include "gmp.h"

#define T(q) {mpz_set_ui(Q,q);mpz_mul(QQ,Q,Q);\
        mpz_powm_ui(R,P,q-1,QQ);\
        if (mpz_cmp_ui(R,1)==0)\
   printf("p=%u q=%u\n",p,q);}

main(argc,argv) int argc; char *argv[];
{
   int p,qmax,q; mpz_t P,Q,QQ,R;

   p=atoi(argv[1]); qmax=atoi(argv[2]);
   mpz_init_set_ui(P,p);
   mpz_init(Q);
   mpz_init(R);
   mpz_init(QQ);
   T(3); T(5); T(7); T(11); T(13); T(17); T(19);
   for (q=23;q<=qmax;q+=2)
      if (q%3 && q%5 && q%7 && q%11 && q%13 &&
          q%17 && q%19 && q%23 && q%29 && q%31 &&
          q%37 && q%41 && q%43 && q%47 && q%53 &&
          q%59 && q%61 && q%67 && q%71 && q%73 &&
          q%79 && q%83 && q%89 && q%97) T(q)
}
```

5 Conclusion

The following table summarizes the characteristics of the three systems. The criterion "automatic allocation" means that the results of operations may be automatically stored in memory, so that the programmer does not have to compute their size. The BigNum package includes some functions that do automatic allocation (level `BigZ`), but without garbage collection, these functions are useless for *big* problems as considered here. The garbage collection is "automatic" when the user does not even have to think of it, "manual" when he must call special functions like `gerepile` in Pari to clean the memory. The "programming facility" reflects the easiness to write a program with the corresponding library, which is often related to the program length.

criteria	BigNum	Gmp	Pari
automatic allocation	yes	yes	yes
garbage collection	no	auto	manual
programming facility	0	++	+

For what concerns speed, another table indicates which package is the fastest one on a given machine and for a given type of operations. For example, the entry **1**/3 for Pari on a XXXXXX tells us that on this kind of machine, Pari is the fastest one for operations between multiprecision integers and small integers, but the slowest one for operations between two multiprecision integers. We conclude from this table that the speed ranking of the packages depends both on the type of machine (some packages have special assembly code) and on the compiler used.

speed	BigNum	Gmp	Pari
	3/2	**1**/3	2/**1**
	3/2	2/**1**	**1**/3
	2/**1**	**1**/2	3/3
	[big × small]/[big × big]		

How to obtain the packages. The BigNum package can be obtained by sending an e-mail message to `doc-server@prl.dec.com` with `send package bignum` in the subject line. The Gmp package can be obtained by anonymous ftp from `prep.ai.mit.edu`. The Pari package can be obtained by anonymous ftp from `nuri.inria.fr` or `math.ucla.edu`.

Acknowledgment. I would like to thank François Morain for his useful comments.

References

[1] C. Batut, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*, December 1991. Available by anonymous ftp from `megrez.ceremab.u-bordeaux.fr` or `math.ucla.edu`.

[2] Richard Crandall, Richard McIntosh, Carl Pomerance, and al. Search for wieferich primes up to $4 \cdot 10^{12}$. http://www.loria.fr/ zimmerma/records/Wieferich.status, 1996. private communication.

[3] C. F. Gauss. *Werke*, volume 2. Göttingen, second edition, 1876. First edition in 1863.

[4] Torbjörn Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 1.2 edition, December 1991. Available by anonymous ftp from `sics.se`.

[5] J.-C. Hervé, B. Serpette, and J. Vuillemin. BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic. Technical Report 2, Digital Paris Research Laboratory, May 1989.

[6] D. H. Lehmer. On Fermat's quotient, base two. *Mathematics of Computation*, 36(153):289–290, January 1981.

[7] Maurice Mignotte and Yves Roy. L'équation de Catalan. Prépublication 513/P-299, Institut de Recherche Mathématique Avancée, Strasbourg, 1992.