CITED REFERENCES AND FURTHER READING:

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990, *Text Compression* (Englewood Cliffs, NJ: Prentice-Hall).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Witten, I.H., Neal, R.M., and Cleary, J.G. 1987, *Communications of the ACM*, vol. 30, pp. 520–540. [1]

20.6 Arithmetic at Arbitrary Precision

Let's compute the number π to a couple of thousand decimal places. In doing so, we'll learn some things about multiple precision arithmetic on computers and meet quite an unusual application of the fast Fourier transform (FFT). We'll also develop a set of routines that you can use for other calculations at any desired level of arithmetic precision.

To start with, we need an analytic algorithm for π. Useful algorithms are quadratically convergent, i.e., they double the number of significant digits at each iteration. Quadratically convergent algorithms for π are based on the *AGM (arithmetic geometric mean)* method, which also finds application to the calculation of elliptic integrals (cf. §6.11) and in advanced implementations of the ADI method for elliptic partial differential equations (§19.5). Borwein and Borwein [1] treat this subject, which is beyond our scope here. One of their algorithms for π starts with the initializations

$$\begin{aligned} X_0 &= \sqrt{2} \\ \pi_0 &= 2 + \sqrt{2} \\ Y_0 &= \sqrt[4]{2} \end{aligned} \tag{20.6.1}$$

and then, for $i = 0, 1, \dots$, repeats the iteration

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\left(\sqrt{X_i} + \frac{1}{\sqrt{X_i}}\right) \\ \pi_{i+1} &= \pi_i \left(\frac{X_{i+1}+1}{Y_i+1}\right) \\ Y_{i+1} &= \frac{Y_i\sqrt{X_{i+1}} + \dfrac{1}{\sqrt{X_{i+1}}}}{Y_i+1} \end{aligned} \tag{20.6.2}$$

The value π emerges as the limit π_∞.

Now, to the question of how to do arithmetic to arbitrary precision: In a high-level language like C, a natural choice is to work in radix (base) 256, so that character arrays can be directly interpreted as strings of digits. At the very end of our calculation, we will want to convert our answer to radix 10, but that is essentially a frill for the benefit of human ears, accustomed to the familiar chant, "three point one four one five nine. . . .” For any less frivolous calculation, we would likely never leave base 256 (or the thence trivially reachable hexadecimal, octal, or binary bases).

We will adopt the convention of storing digit strings in the “human” ordering, that is, with the first stored digit in an array being most significant, the last stored digit being least significant. The opposite convention would, of course, also be possible. “Carries,” where we need to partition a number larger than 255 into a low-order byte and a high-order carry, present a minor programming annoyance, solved, in the routines below, by the use of the macros LOBYTE and HIBYTE.

It is easy at this point, following Knuth [2], to write a routine for the “fast” arithmetic operations: short addition (adding a single byte to a string), addition, subtraction, short multiplication (multiplying a string by a single byte), short division, ones-complement negation; and a couple of utility operations, copying and left-shifting strings. (On the diskette, these functions are all in the single file mpops.c.)

```
#define LOBYTE(x) ((unsigned char) ((x) & 0xff))
#define HIBYTE(x) ((unsigned char) ((x) >> 8 & 0xff))
```

Multiple precision arithmetic operations done on character strings, interpreted as radix 256 numbers. This set of routines collects the simpler operations.

```
void mpadd(unsigned char w[], unsigned char u[], unsigned char v[], int n)
```
Adds the unsigned radix 256 integers u[1..n] and v[1..n] yielding the unsigned integer w[1..n+1].
```
{
    int j;
    unsigned short ireg=0;

    for (j=n;j>=1;j--) {
        ireg=u[j]+v[j]+HIBYTE(ireg);
        w[j+1]=LOBYTE(ireg);
    }
    w[1]=HIBYTE(ireg);
}

void mpsub(int *is, unsigned char w[], unsigned char u[], unsigned char v[],
    int n)
```
Subtracts the unsigned radix 256 integer v[1..n] from u[1..n] yielding the unsigned integer w[1..n]. If the result is negative (wraps around), is is returned as -1; otherwise it is returned as 0.
```
{
    int j;
    unsigned short ireg=256;

    for (j=n;j>=1;j--) {
        ireg=255+u[j]-v[j]+HIBYTE(ireg);
        w[j]=LOBYTE(ireg);
    }
    *is=HIBYTE(ireg)-1;
}

void mpsad(unsigned char w[], unsigned char u[], int n, int iv)
```
Short addition: the integer iv (in the range $0 \le$ iv ≤ 255) is added to the unsigned radix 256 integer u[1..n], yielding w[1..n+1].
```
{
    int j;
    unsigned short ireg;

    ireg=256*iv;
```

```
    for (j=n;j>=1;j--) {
        ireg=u[j]+HIBYTE(ireg);
        w[j+1]=LOBYTE(ireg);
    }
    w[1]=HIBYTE(ireg);
}
```

```
void mpsmu(unsigned char w[], unsigned char u[], int n, int iv)
```
Short multiplication: the unsigned radix 256 integer u[1..n] is multiplied by the integer iv (in the range $0 \leq$ iv ≤ 255), yielding w[1..n+1].
```
{
    int j;
    unsigned short ireg=0;

    for (j=n;j>=1;j--) {
        ireg=u[j]*iv+HIBYTE(ireg);
        w[j+1]=LOBYTE(ireg);
    }
    w[1]=HIBYTE(ireg);
}
```

```
void mpsdv(unsigned char w[], unsigned char u[], int n, int iv, int *ir)
```
Short division: the unsigned radix 256 integer u[1..n] is divided by the integer iv (in the range $0 \leq$ iv ≤ 255), yielding a quotient w[1..n] and a remainder ir (with $0 \leq$ ir ≤ 255).
```
{
    int i,j;

    *ir=0;
    for (j=1;j<=n;j++) {
        i=256*(*ir)+u[j];
        w[j]=(unsigned char) (i/iv);
        *ir=i % iv;
    }
}
```

```
void mpneg(unsigned char u[], int n)
```
Ones-complement negate the unsigned radix 256 integer u[1..n].
```
{
    int j;
    unsigned short ireg=256;

    for (j=n;j>=1;j--) {
        ireg=255-u[j]+HIBYTE(ireg);
        u[j]=LOBYTE(ireg);
    }
}
```

```
void mpmov(unsigned char u[], unsigned char v[], int n)
```
Move v[1..n] onto u[1..n].
```
{
    int j;

    for (j=1;j<=n;j++) u[j]=v[j];
}
```

```
void mplsh(unsigned char u[], int n)
```
Left shift u(2..n+1) onto u[1..n].
```
{
    int j;

    for (j=1;j<=n;j++) u[j]=u[j+1];
}
```

Full multiplication of two digit strings, if done by the traditional hand method, is not a fast operation: In multiplying two strings of length N, the multiplicand would be short-multiplied in turn by each byte of the multiplier, requiring $O(N^2)$ operations in all. We will see, however, that *all* the arithmetic operations on numbers of length N can in fact be done in $O(N \times \log N \times \log\log N)$ operations.

The trick is to recognize that multiplication is essentially a *convolution* (§13.1) of the digits of the multiplicand and multiplier, followed by some kind of carry operation. Consider, for example, two ways of writing the calculation 456×789:

$$
\begin{array}{r}
456 \\
\times\ \underline{789} \\
4104 \\
3648 \\
\underline{3192} \\
359784
\end{array}
\qquad\qquad
\begin{array}{rrrrrr}
 & & & 4 & 5 & 6 \\
 & & \times & 7 & 8 & 9 \\ \cline{4-6}
 & & & 36 & 45 & 54 \\
 & & 32 & 40 & 48 & \\
 & 28 & 35 & 42 & & \\ \cline{2-6}
 & 28 & 67 & 118 & 93 & 54 \\ \hline
3 & 5 & 9 & 7 & 8 & 4
\end{array}
$$

The tableau on the left shows the conventional method of multiplication, in which three separate short multiplications of the full multiplicand (by 9, 8, and 7) are added to obtain the final result. The tableau on the right shows a different method (sometimes taught for mental arithmetic), where the single-digit cross products are all computed (e.g. $8 \times 6 = 48$), then added in columns to obtain an incompletely carried result (here, the list $28, 67, 118, 93, 54$). The final step is a single pass from right to left, recording the single least-significant digit and carrying the higher digit or digits into the total to the left (e.g. $93 + 5 = 98$, record the 8, carry 9).

You can see immediately that the column sums in the right-hand method are components of the convolution of the digit strings, for example $118 = 4 \times 9 + 5 \times 8 + 6 \times 7$. In §13.1 we learned how to compute the convolution of two vectors by the fast Fourier transform (FFT): Each vector is FFT'd, the two complex transforms are multiplied, and the result is inverse-FFT'd. Since the transforms are done with floating arithmetic, we need sufficient precision so that the exact integer value of each component of the result is discernible in the presence of roundoff error. We should therefore allow a (conservative) few times $\log_2(\log_2 N)$ bits for roundoff in the FFT. A number of length N bytes in radix 256 can generate convolution components as large as the order of $(256)^2 N$, thus requiring $16 + \log_2 N$ bits of precision for exact storage. If `it` is the number of bits in the floating mantissa (cf. §20.1), we obtain the condition

$$16 + \log_2 N + \text{few} \times \log_2 \log_2 N < \texttt{it} \tag{20.6.3}$$

We see that single precision, say with $\texttt{it} = 24$, is inadequate for any interesting value of N, while double precision, say with $\texttt{it} = 53$, allows N to be greater than 10^6, corresponding to some millions of decimal digits. The following routine therefore presumes double precision versions of `realft` (§12.3) and `four1` (§12.2), here called `drealft` and `dfour1`. (These routines are included on the *Numerical Recipes* diskettes.)

```
#include "nrutil.h"
#define RX 256.0

void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
    int m)
```
Uses Fast Fourier Transform to multiply the unsigned radix 256 integers u[1..n] and v[1..m], yielding a product w[1..n+m].
```
{
    void drealft(double data[], unsigned long n, int isign);   double version of realft.
    int j,mn,nn=1;
    double cy,t,*a,*b;

    mn=IMAX(m,n);
    while (nn < mn) nn <<= 1;        Find the smallest usable power of two for the trans-
    nn <<= 1;                            form.
    a=dvector(1,nn);
    b=dvector(1,nn);
    for (j=1;j<=n;j++)               Move U to a double precision floating array.
        a[j]=(double)u[j];
    for (j=n+1;j<=nn;j++) a[j]=0.0;
    for (j=1;j<=m;j++)               Move V to a double precision floating array.
        b[j]=(double)v[j];
    for (j=m+1;j<=nn;j++) b[j]=0.0;
    drealft(a,nn,1);                 Perform the convolution: First, the two Fourier trans-
    drealft(b,nn,1);                     forms.
    b[1] *= a[1];                    Then multiply the complex results (real and imagi-
    b[2] *= a[2];                        nary parts).
    for (j=3;j<=nn;j+=2) {
        b[j]=(t=b[j])*a[j]-b[j+1]*a[j+1];
        b[j+1]=t*a[j+1]+b[j+1]*a[j];
    }
    drealft(b,nn,-1);                Then do the inverse Fourier transform.
    cy=0.0;                          Make a final pass to do all the carries.
    for (j=nn;j>=1;j--) {
        t=b[j]/(nn>>1)+cy+0.5;       The 0.5 allows for roundoff error.
        cy=(unsigned long) (t/RX);
        b[j]=t-cy*RX;
    }
    if (cy >= RX) nrerror("cannot happen in fftmul");
    w[1]=(unsigned char) cy;         Copy answer to output.
    for (j=2;j<=n+m;j++)
        w[j]=(unsigned char) b[j-1];
    free_dvector(b,1,nn);
    free_dvector(a,1,nn);
}
```

With multiplication thus a "fast" operation, division is best performed by multiplying the dividend by the reciprocal of the divisor. The reciprocal of a value V is calculated by iteration of Newton's rule,

$$U_{i+1} = U_i(2 - VU_i) \tag{20.6.4}$$

which results in the quadratic convergence of U_∞ to $1/V$, as you can easily prove. (Many supercomputers and RISC machines actually use this iteration to perform divisions.) We can now see where the operations count $N \log N \log \log N$, mentioned above, originates: $N \log N$ is in the Fourier transform, with the iteration to converge Newton's rule giving an additional factor of $\log \log N$.

```
#include "nrutil.h"
#define MF 4
#define BI (1.0/256)

void mpinv(unsigned char u[], unsigned char v[], int n, int m)
```
Character string v[1..m] is interpreted as a radix 256 number with the radix point after (nonzero) v[1]; u[1..n] is set to the most significant digits of its reciprocal, with the radix point after u[1].
```
{
    void mpmov(unsigned char u[], unsigned char v[], int n);
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpneg(unsigned char u[], int n);
    unsigned char *rr,*s;
    int i,j,maxmn,mm;
    float fu,fv;

    maxmn=IMAX(n,m);
    rr=cvector(1,1+(maxmn<<1));
    s=cvector(1,maxmn);
    mm=IMIN(MF,m);
    fv=(float) v[mm];                 Use ordinary floating arithmetic to get an initial ap-
    for (j=mm-1;j>=1;j--) {               proximation.
        fv *= BI;
        fv += v[j];
    }
    fu=1.0/fv;
    for (j=1;j<=n;j++) {
        i=(int) fu;
        u[j]=(unsigned char) i;
        fu=256.0*(fu-i);
    }
    for (;;) {                        Iterate Newton's rule to convergence.
        mpmul(rr,u,v,n,m);            Construct 2 - UV in S.
        mpmov(s,&rr[1],n);
        mpneg(s,n);
        s[1] -= 254;                  Multiply SU into U.
        mpmul(rr,s,u,n,n);
        mpmov(u,&rr[1],n);
        for (j=2;j<n;j++)             If fractional part of S is not zero, it has not converged
            if (s[j]) break;              to 1.
        if (j==n) {
            free_cvector(s,1,maxmn);
            free_cvector(rr,1,1+(maxmn<<1));
            return;
        }
    }
}
```

Division now follows as a simple corollary, with only the necessity of calculating the reciprocal to sufficient accuracy to get an exact quotient and remainder.

```
#include "nrutil.h"
#define MACC 6

void mpdiv(unsigned char q[], unsigned char r[], unsigned char u[],
    unsigned char v[], int n, int m)
```
Divides unsigned radix 256 integers u[1..n] by v[1..m] (with m $\leq$ n required), yielding a quotient q[1..n-m+1] and a remainder r[1..m].
```
{
    void mpinv(unsigned char u[], unsigned char v[], int n, int m);
    void mpmov(unsigned char u[], unsigned char v[], int n);
```

```
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpsad(unsigned char w[], unsigned char u[], int n, int iv);
    void mpsub(int *is, unsigned char w[], unsigned char u[], unsigned char v[],
        int n);
    int is;
    unsigned char *rr,*s;

    rr=cvector(1,(n+MACC)<<1);
    s=cvector(1,(n+MACC)<<1);
    mpinv(s,v,n+MACC,m);                  Set S = 1/V.
    mpmul(rr,s,u,n+MACC,n);               Set Q = SU.
    mpsad(s,rr,n+n+MACC/2,1);
    mpmov(q,&s[2],n-m+1);
    mpmul(rr,q,v,n-m+1,m);                Multiply and subtract to get the remainder.
    mpsub(&is,&rr[1],u,&rr[1],n);
    if (is) nrerror("MACC too small in mpdiv");
    mpmov(r,&rr[n-m+1],m);
    free_cvector(s,1,(n+MACC)<<1);
    free_cvector(rr,1,(n+MACC)<<1);
}
```

Square roots are calculated by a Newton's rule much like division. If

$$U_{i+1} = \frac{1}{2}U_i(3 - VU_i^2) \tag{20.6.5}$$

then U_∞ converges quadratically to $1/\sqrt{V}$. A final multiplication by V gives $\sqrt{V}$.

```
#include <math.h>
#include "nrutil.h"
#define MF 3
#define BI (1.0/256)

void mpsqrt(unsigned char w[], unsigned char u[], unsigned char v[], int n,
    int m)
```

Character string `v[1..m]` is interpreted as a radix 256 number with the radix point after `v[1]`; `w[1..n]` is set to its square root (radix point after `w[1]`), and `u[1..n]` is set to the reciprocal thereof (radix point before `u[1]`). `w` and `u` need not be distinct, in which case they are set to the square root.

```
{
    void mplsh(unsigned char u[], int n);
    void mpmov(unsigned char u[], unsigned char v[], int n);
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpneg(unsigned char u[], int n);
    void mpsdv(unsigned char w[], unsigned char u[], int n, int iv, int *ir);
    int i,ir,j,mm;
    float fu,fv;
    unsigned char *r,*s;

    r=cvector(1,n<<1);
    s=cvector(1,n<<1);
    mm=IMIN(m,MF);
    fv=(float) v[mm];                 Use ordinary floating arithmetic to get an initial ap-
    for (j=mm-1;j>=1;j--) {               proximation.
        fv *= BI;
        fv += v[j];
    }
    fu=1.0/sqrt(fv);
    for (j=1;j<=n;j++) {
        i=(int) fu;
        u[j]=(unsigned char) i;
```

```
        fu=256.0*(fu-i);
    }
    for (;;) {                          Iterate Newton's rule to convergence.
        mpmul(r,u,u,n,n);               Construct S = (3 − VU^2)/2.
        mplsh(r,n);
        mpmul(s,r,v,n,IMIN(m,n));
        mplsh(s,n);
        mpneg(s,n);
        s[1] -= 253;
        mpsdv(s,s,n,2,&ir);
        for (j=2;j<n;j++) {             If fractional part of S is not zero, it has not converged
            if (s[j]) {                    to 1.
                mpmul(r,s,u,n,n);       Replace U by SU.
                mpmov(u,&r[1],n);
                break;
            }
        }
        if (j<n) continue;
        mpmul(r,u,v,n,IMIN(m,n));       Get square root from reciprocal and return.
        mpmov(w,&r[1],n);
        free_cvector(s,1,n<<1);
        free_cvector(r,1,n<<1);
        return;
    }
}
```

We already mentioned that radix conversion to decimal is a merely cosmetic operation that should normally be omitted. The simplest way to convert a fraction to decimal is to multiply it repeatedly by 10, picking off (and subtracting) the resulting integer part. This, has an operations count of $O(N^2)$, however, since each liberated decimal digit takes an $O(N)$ operation. It *is* possible to do the radix conversion as a fast operation by a "divide and conquer" strategy, in which the fraction is (fast) multiplied by a large power of 10, enough to move about half the desired digits to the left of the radix point. The integer and fractional pieces are now processed independently, each further subdivided. If our goal were a few billion digits of π, instead of a few thousand, we would need to implement this scheme. For present purposes, the following lazy routine is adequate:

```
#define IAZ 48

void mp2dfr(unsigned char a[], unsigned char s[], int n, int *m)
```
Converts a radix 256 fraction a[1..n] (radix point before a[1]) to a decimal fraction represented as an ascii string s[1..m], where m is a returned value. The input array a[1..n] is destroyed. NOTE: For simplicity, this routine implements a slow ($\propto N^2$) algorithm. Fast ($\propto N \ln N$), more complicated, radix conversion algorithms do exist.
```
{
    void mplsh(unsigned char u[], int n);
    void mpsmu(unsigned char w[], unsigned char u[], int n, int iv);
    int j;

    *m=(int) (2.408*n);
    for (j=1;j<=(*m);j++) {
        mpsmu(a,a,n,10);
        s[j]=a[1]+IAZ;
        mplsh(a,n);
    }
}
```

Finally, then, we arrive at a routine implementing equations (20.6.1) and (20.6.2):

3.14159265358979323846264338327950288419716939937510582097494459230781640628
620899862803482534211706798214808651328230664709384460955058223172535940812
848111745028410270193852110555964462294895493038196442881097566593344612847
564823378678316527120190914564856692346034861045432664821339360726024914127
372458700660631558817488152092096282925409171536436789259036001133053054882
046652138414695194151160943305727036575959195309218611738193261179310511854
807446237996274956735188575272489122793818301194912983367336244065664308602
139494639522473719070217986094370277053921717629317675238467481846766940513
200056812714526356082778577134275778960917363717872146844090122495343014654
958537105079227968925892354201995611212902196086403441815981362977477130996
051870721134999999837297804995105973173281609631859502445945534690830264252
230825334468503526193118817101000313783875288658753320838142061717766914730
359825349042875546873115956286388235378759375195778185778053217122680661300
192787661119590921642019893809525720106548586327886593615338182796823030195
203530185296899577362259941389124972177528347913151557485724245415069595082
953311686172785588907509838175463746493931925506040092770167113900984882401
285836160356370766010471018194295559619894676783744944825537977472684710404
753464620804668425906949129331367702898915210475216205696602405803815019351
125338243003558764024749647326391419927260426992279678235478163600934172164
121992458631503028618297455570674983850549458858692699569092721079750930295
532116534498720275596023648066549911988183479775356636980742654252786255181
841757467289097777279380008164706001614524919217321721477235014144197356854
816136115735255213347574184946843852332390739414333454776241686251898356948
556209921922218427255025425688767179049460165346680498862723279178608578438
382796797668145410095388378636095068006422512520511739298489608412848862694
560424196528502221066118630674427862203919494504712371378696095636437191728
746776465757396241389086583264599581339047802759009946576407895126946839835
259570982582262052248940772671947826848260147699090264013639443745530506820
349625245174939965143142980919065925093722169646151570985838741059788595977
297549893016175392846813826868386894277415599185592524595395943104997252468
084598727364469584865383673622262609912460805124388439045124413654976278079
771569143599770012961608944169486855584840635342207222582848864815845602850

Figure 20.6.1. The first 2398 decimal digits of π, computed by the routines in this section.

```
#include <stdio.h>
#include "nrutil.h"
#define IAOFF 48

void mppi(int n)
```
Demonstrate multiple precision routines by calculating and printing the first n bytes of π.
```
{
    void mp2dfr(unsigned char a[], unsigned char s[], int n, int *m);
    void mpadd(unsigned char w[], unsigned char u[], unsigned char v[], int n);
    void mpinv(unsigned char u[], unsigned char v[], int n, int m);
    void mplsh(unsigned char u[], int n);
    void mpmov(unsigned char u[], unsigned char v[], int n);
    void mpmul(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    void mpsdv(unsigned char w[], unsigned char u[], int n, int iv, int *ir);
    void mpsqrt(unsigned char w[], unsigned char u[], unsigned char v[], int n,
        int m);
    int ir,j,m;
    unsigned char mm,*x,*y,*sx,*sxi,*t,*s,*pi;

    x=cvector(1,n+1);
    y=cvector(1,n<<1);
    sx=cvector(1,n);
    sxi=cvector(1,n);
    t=cvector(1,n<<1);
    s=cvector(1,3*n);
    pi=cvector(1,n+1);
    t[1]=2;                              Set T = 2.
    for (j=2;j<=n;j++) t[j]=0;
```

```
    mpsqrt(x,x,t,n,n);                     Set X_0 = sqrt(2).
    mpadd(pi,t,x,n);                       Set pi_0 = 2 + sqrt(2).
    mplsh(pi,n);
    mpsqrt(sx,sxi,x,n,n);                  Set Y_0 = 2^{1/4}.
    mpmov(y,sx,n);
    for (;;) {
        mpadd(x,sx,sxi,n);                 Set X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2.
        mpsdv(x,&x[1],n,2,&ir);
        mpsqrt(sx,sxi,x,n,n);              Form the temporary T = Y_i X_{i+1}^{1/2} + X_{i+1}^{-1/2}.
        mpmul(t,y,sx,n,n);
        mpadd(&t[1],&t[1],sxi,n);
        x[1]++;                            Increment X_{i+1} and Y_i by 1.
        y[1]++;
        mpinv(s,y,n,n);                    Set Y_{i+1} = T/(Y_i + 1).
        mpmul(y,&t[2],s,n,n);
        mplsh(y,n);
        mpmul(t,x,s,n,n);                  Form temporary T = (X_{i+1} + 1)/(Y_i + 1).
        mm=t[2]-1;                         If T = 1 then we have converged.
        for (j=3;j<=n;j++) {
            if (t[j] != mm) break;
        }
        m=t[n+1]-mm;
        if (j <= n || m > 1 || m < -1) {
                mpmul(s,pi,&t[1],n,n);         Set pi_{i+1} = T pi_i.
                mpmov(pi,&s[1],n);
                continue;
        }
        printf("pi=\n");
        s[1]=pi[1]+IAOFF;
        s[2]='.';
        m=mm;
        mp2dfr(&pi[1],&s[2],n-1,&m);
        Convert to decimal for printing. NOTE: The conversion routine, for this demonstration
        only, is a slow (∝ N^2) algorithm. Fast (∝ N ln N), more complicated, radix conversion
        algorithms do exist.
        s[m+3]=0;
        printf(" %64s\n",&s[1]);
        free_cvector(pi,1,n+1);
        free_cvector(s,1,3*n);
        free_cvector(t,1,n<<1);
        free_cvector(sxi,1,n);
        free_cvector(sx,1,n);
        free_cvector(y,1,n<<1);
        free_cvector(x,1,n+1);
        return;
    }
}
```

Figure 20.6.1 gives the result, computed with n $= 1000$. As an exercise, you might enjoy checking the first hundred digits of the figure against the first 12 terms of Ramanujan's celebrated identity [3]

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!\,(1103 + 26390n)}{(n!\,396^n)^4} \tag{20.6.6}$$

using the above routines. You might also use the routines to verify that the number $2^{512} + 1$ is not a prime, but has factors 2,424,833 and 7,455,602,825,647,884,208,337,395,736,200,454,918,783,366,342,657 (which are in fact prime; the remaining prime factor being about 7.416×10^{98}) [4].

CITED REFERENCES AND FURTHER READING:

Borwein, J.M., and Borwein, P.B. 1987, *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity* (New York: Wiley). [1]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.3. [2]

Ramanujan, S. 1927, *Collected Papers of Srinivasa Ramanujan*, G.H. Hardy, P.V. Seshu Aiyar, and B.M. Wilson, eds. (Cambridge, U.K.: Cambridge University Press), pp. 23–39. [3]

Kolata, G. 1990, June 20, *The New York Times*. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).