SUBROUTINE arcsum(iin,iout,ja,nrad,nwk,nc)
INTEGER ja,nc,nrad,nwk,iin(*),iout(*)

Used by arcode. Add the integer ja to the radix nrad multiple-precision integer iin(nc..nwk).
Return the result in iout(nc..nwk).

INTEGER j,jtmp,karry
karry=0
DO 11 j=nwk,nc+1,-1
   jtmp=ja
   ja=ja/nrad
   iout(j)=iin(j)+(jtmp-ja*nrad)+karry
   IF (iout(j).GE.nrad) THEN
     iout(j)=iout(j)-nrad
     karry=1
   ELSE
     karry=0
   ENDIF
11 CONTINUE
iout(nc)=iin(nc)+ja+karry
RETURN
END

If radix-changing, rather than compression, is your primary aim (for example to convert an arbitrary file into printable characters) then you are of course free to set all the components of nfreq equal, say, to 1.

CITED REFERENCES AND FURTHER READING:

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

\[ X_0 = \sqrt{2} \]
\[ \pi_0 = 2 + \sqrt{2} \]  \hspace{1cm} (20.6.1)
\[ Y_0 = \sqrt{2} \]
and then, for \( i = 0,1, \ldots \), repeats the iteration

\[
X_{i+1} = \frac{1}{2} \left( \sqrt{X_i} + \sqrt{\frac{1}{X_i}} \right)
\]

\[
\pi_{i+1} = \pi_i \left( \frac{X_{i+1} + 1}{Y_{i+1}} \right)
\]

\[
Y_{i+1} = \frac{Y_i \sqrt{X_{i+1}} + 1}{Y_{i+1}}
\]

(20.6.2)

The value \( \pi \) emerges as the limit \( \pi_\infty \).

Now, to the question of how to do arithmetic to arbitrary precision: In a high-level language like FORTRAN, 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 FORTRAN’s EQUIVALENCE facility, and some initial testing of the order in which bytes are stored in a FORTRAN integer.

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.

SUBROUTINE mops(w,u,v)
CHARACTER*1 w(*),u(*),v(*)
Multiple precision arithmetic operations done on character strings, interpreted as radix 256 numbers. This routine collects the simpler operations.
INTEGER i,ireg,j,n,ir,is,iv,ii1,ii2
CHARACTER*1 creg(4)
SAV
EQUIVALENCE (ireg,creg)
It is assumed that with the above equivalence, creg(ii1) addresses the low-order byte of ireg, and creg(ii2) addresses the next higher order byte. The values ii1 and ii2 are set by an initial call to mpinit.
ENTRY mpinit
ireg=256*ichar('2')+ichar('1')
do :: j=1,4
Figure out the byte ordering.
if (creg(j).eq.'1') ii1=j
if (creg(j).eq.'2') ii2=j
endo ::
return
ENTRY mpadd(w,u,v,n)
Adds the unsigned radix 256 integers u(1:n) and v(1:n) yielding the unsigned integer w(1:n+1).
ireg=0
do :: j=n,1,-1
\begin{verbatim}
  ireg=ichar(u(j))+ichar(v(j))+ichar(creg(ii2))
  w(j+1)=creg(ii1)
enddo 12
  w(1)=creg(ii2)
return
ENTRY mpsub(is,w,u,v,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.
  ireg=256
  do 13 j=n,1,-1
    ireg=255+ichar(u(j))-ichar(v(j))+ichar(creg(ii2))
    w(j)=creg(ii1)
  enddo 13
  is=ichar(creg(ii2))-1
return
ENTRY mpsad(w,u,n,iv)
  Short addition: the integer \(iv\) (in the range \(0 \leq iv \leq 255\)) is added to the unsigned radix 256 integer \(u(1:n)\), yielding \(w(1:n+1)\).
  ireg=256*iv
  do 14 j=n,1,-1
    ireg=ichar(u(j))+ichar(creg(ii2))
    w(j+1)=creg(ii1)
  enddo 14
  w(1)=creg(ii2)
return
ENTRY mpsmu(w,u,n,iv)
  Short multiplication: the unsigned radix 256 integer \(u(1:n)\) is multiplied by the integer \(iv\) (in the range \(0 \leq iv \leq 255\)), yielding \(w(1:n+1)\).
  ireg=0
  do 15 j=n,1,-1
    i=256*ir+ichar(u(j))
    w(j)=char(i/iv)
    ir=mod(i,iv)
  enddo 15
return
ENTRY mpneg(u,n)
  Ones-complement negate the unsigned radix 256 integer \(u(1:n)\).
  ireg=256
  do 17 j=n,1,-1
    ireg=255-ichar(u(j))+ichar(creg(ii2))
    u(j)=creg(ii1)
  enddo 17
return
ENTRY mpmov(u,v,n)
  Move \(v(1:n)\) onto \(u(1:n)\).
  do 18 j=1,n
    u(j)=v(j)
  enddo 18
return
ENTRY mplsh(u,n)
  Left shift \(u(2..n+1)\) onto \(u(1:n)\).
  do 19 j=1,n
    u(j)=u(j+1)
  enddo 19
\end{verbatim}
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 \( 456 \times 789 \):

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 \( \text{it} \) is the number of bits in the floating mantissa (cf. §20.1), we obtain the condition

\[
16 + \log_2 N + \text{few} \times \log_2 \log_2 N < \text{it}
\]

We see that single precision, say with \( \text{it} = 24 \), is inadequate for any interesting value of \( N \), while double precision, say with \( \text{it} = 53 \), allows \( N \) to be greater than \( 10^6 \), corresponding to some millions of decimal digits. The following routine:

```fortran
enddo
return
END
```
therefore presumes double precision versions of realft (§12.3) and four1 (§12.2), here called drealf1t and dfour1. (These routines are included on the Numerical Recipes diskettes.)

SUBROUTINE mpmul(w,u,v,n,m)
INTEGER m,n,NMAX
CHARACTER*1 w(n+m),u(n),v(m)
DOUBLE PRECISION RX
PARAMETER (NMAX=8192,RX=256.D0)
C USES drealf1t DOUBLE PRECISION version of realft.
C 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).
INTEGER j,mn,nn
DOUBLE PRECISION cy,t,a(NMAX),b(NMAX)

mn=max(m,n)
nn=1 Find the smallest useable power of two for the transform.
1 if(nn.lt.mn) then
   nn=nn+nn
end if
if(nn.gt.NMAX)pause 'NMAX too small in fftmul'
do : 11 j=1,n
   a(j)=ichar(u(j))
enddo : 11
do : 12 j=n+1,nn
   a(j)=0.D0
enddo : 12
do : 13 j=1,m
   b(j)=ichar(v(j))
enddo : 13
do : 14 j=m+1,nn
   b(j)=0.D0
enddo : 14

Perform the convolution: First, the two Fourier transforms.
call drealf1t(a,nn,1)
call drealf1t(b,nn,1)
b(1)=b(1)*a(1)
Then multiply the complex results (real and imaginary parts).
b(2)=b(2)*a(2)
do : 15 j=3,nn,2
   t=b(j)
   b(j)=t*a(j)-b(j+1)*a(j+1)
   b(j+1)=t*a(j+1)+b(j+1)*a(j)
enddo : 15

call drealf1t(b,nn,-1)

Then do the inverse Fourier transform.
cy=0.
do : 16 j=nn,1,-1
   t=b(j)/(nn/2)+cy+0.5D0
   The 0.5 allows for roundoff error.
b(j)=mod(t,RX)
cy=int(t/RX)
enddo : 16
if (cy.ge.RX) pause 'cannot happen in fftmul'
w(1)=char(int(cy))
do : 17 j=2,n+m
   w(j)=char(int(b(j-1)))
enddo : 17
return
END

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
20.6 Arithmetic at Arbitrary Precision

\[ V \text{ is calculated by iteration of Newton's rule,} \]
\[ U_{i+1} = U_i (2 - V U_i) \quad (20.6.4) \]

which results in the quadratic convergence of \( U_\infty \) 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 \( \log \log N \).

SUBROUTINE mpinv(u,v,n,m)
INTEGER m,n,MF,NMAX
CHARACTER*1 u(n),v(m)
REAL BI
PARAMETER (MF=4,BI=1./256.,NMAX=8192)
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 \( u(1) \).

C USES mpmov,mpmul,mpneg
INTEGER i,j,mm
REAL fu,fv
CHARACTER*1 rr(2*NMAX+1),s(NMAX)
if(max(n,m).gt.NMAX)pause 'NMAX too small in mpinv'
mm=min(MF,m)
fv=ichar(v(mm))
Use ordinary floating arithmetic to get an initial approximation.
do : j=mm-1,1,-1
fv=fv*BI+ichar(v(j))
enddo : fv=1./fv
do : j=1,n
i=int(fu)
u(j)=char(i)
fu=256.*(fu-i)
enddo : fu=1./fv
continuedo : j=1,n
i=int(fu)
u(j)=char(i)
fu=256.*(fu-i)
enddo : 1
continuedo : Iterate Newton's rule to convergence.
call mpmul(rr,u,v,n,m)
call mpmov(s,rr(2),n)
call mpneg(s,n)
s(1)=char(ichar(s(1))-254)
call mpmul(rr,s,u,n,n)
call mpmov(u,rr(2),n)
do : 13 j=2,n-1
if(ichar(s(j)).ne.0)goto 1
dono : 13
continuedo : If fractional part of \( S \) is not zero, it has not converged to 1.
call mpmul(rr,u,v,n,m)
call mpmov(s,rr(2),n)
call mpneg(s,n)
s(1)=char(ichar(s(1))-254)
call mpmul(rr,s,u,n,n)
call mpmov(u,rr(2),n)
do : 13 j=2,n-1
if(ichar(s(j)).ne.0)goto 1
dono : 13
continuedo : 1
continuedo
return
END

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.

SUBROUTINE mpdiv(q,r,u,v,n,m)
INTEGER m,n,NMAX,MACC
CHARACTER*1 q(n-m+1),r(m),u(n),v(m)
PARAMETER (NMAX=8192,MACC=6)
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 \( r(1:m) \).
C USES mpinv,mpmov,mpmul,mpsad,mpsub
INTEGER is
CHARACTER*1 rr(2*NMAX),s(2*NMAX)
if(n+MACC.gt.NMAX)pause 'NMAX too small in mpdiv'
Square roots are calculated by a Newton’s rule much like division. If

\[ U_{i+1} = \frac{1}{2} U_i \left( 3 - V U_i^2 \right) \] (20.6.5)

then \( U_{\infty} \) converges quadratically to \( 1/\sqrt{V} \). A final multiplication by \( V \) gives \( \sqrt{V} \).

SUBROUTINE mpsqrt(w,u,v,n,m)
INTEGER m,n,NMAX,MF
CHARACTER*1 w(*),u(*),v(*)
REAL BI
PARAMETER (NMAX=2048,MF=3,BI=1./256.)
C USES mplsh,mpmov,mpmul,mpneg,mpsdv
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.

INTEGER i,ir,j,mm
REAL fu,fv
CHARACTER*1 r(NMAX),s(NMAX)
if(2*n+1.gt.NMAX)pause 'NMAX too small in mpsqrt'
mm=min(m,MF)
fv=ichar(v(mm)) Use ordinary floating arithmetic to get an initial approximation.
do 11 j=mm-1,1,-1
fv=BI*fv+ichar(v(j))
enddo 11
fu=1./sqrt(fv)
do 12 j=1,n
i=int(fu)
u(j)=char(i)
fu=256.*(fu-i)
12 continue
Iterate Newton’s rule to convergence.
call mpmul(r,u,u,n,n) Construct \( S = (3 - V U^2)/2 \).
call mplsh(r,n) call mpmul(s,r,v,n,m)
call mplsh(s,n) call mpmul(s,s,v,n,m)
call mpmul(s,n) call mpmul(s,v,n,m)
s(1)=char(ichar(s(1))-253) call mpsdv(s,s,n,2,ir)
do 13 j=2,n-1
If fractional part of \( S \) is not zero, it has not converged
if(ichar(s(j)).ne.0)goto 2
enddo 13
If fractional part of \( S \) is not zero, it has not converged
if(ichar(s(j)).ne.0)goto 2
1 continue
Get square root from reciprocal and return.
call mpmul(r,u,v,n,m)
call mpmov(w,r(2),n)
return
2 continue
Replace \( U \) by \( SU \).
call mpmul(r,s,u,n,n)
call mpmov(u,r(2),n)
goto 1
END
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 $\pi$, instead of a few thousand, we would need to implement this scheme. For present purposes, the following lazy routine is adequate:

```fortran
SUBROUTINE mp2dfr(a,s,n,m)
INTEGER m,n,IAZ
CHARACTER*1 a(*),s(*)
PARAMETER (IAZ=48)
C USES mplsh,mpsmu
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.
INTEGER j
m=2.408*n
do 11 j=1,m
   call mpsmu(a,a,n,10)
   s(j)=char(ichar(a(1))+IAZ)
   call mplsh(a,n)
11 continue
return
END
```

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

```fortran
SUBROUTINE mppi(n)
INTEGER n,IAOFF,NMAX
PARAMETER (IAOFF=48,NMAX=8192)
C USES mpinit,mp2dfr,mpadd,mpinv,mplsh,mpmov,mpmul,mpsdv,mpsqrt
Demonstrate multiple precision routines by calculating and printing the first n bytes of $\pi$.
INTEGER ir,j,m
CHARACTER*1 x(NMAX),y(NMAX),sx(NMAX),sxi(NMAX),t(NMAX),s(3*NMAX),
* pi(NMAX)
call mpinit
  t(1)=char(2)
do 11 j=2,n
     t(j)=char(0)
11 continue
  call mpsqrt(x,x,t,n,n)
Set $T = 2$.
call mpadd(pi,t,x,n)
Set $\pi_0 = 2 + \sqrt{T}$.
call mplsh(pi,n)
call mpsqrt(sx,sxi,x,n,n)
Set $Y_0 = 2^{1/4}$.
call mpmov(y,sx,n)
call mpadd(x,sx,sxi,n)
Set $X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2$.
call mpadd(x,2,sx,2,ir)
call mpsqrt(sx,sxi,x,n,n)
Form the temporary $T = Y_iX_{i+1}^{1/2} + X_{i+1}^{-1/2}$.
call mpmul(t,y,x,n,n)
call mpadd(t(2),t(2),sx,n)
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press.
Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for personal use and for educational use in courses at
universities to make one copy per student in any computer class.
Further reproduction or any copying of machine-readable files (including this one) is strictly prohibited.
To order Numerical Recipes books or software, visit our website:
http://www.nr.com or call 1-800-872-7423 (North America only) or send email to trade@cup.cam.ac.uk (outside North America).
Figure 20.6.1. The first 2398 decimal digits of π, computed by the routines in this section.

\[
x(1) = \text{char(ichar}(x(1)) + 1) \quad \text{Increment } X_{i+1} \text{ and } Y_i \text{ by 1.}
\]

\[
y(1) = \text{char(ichar}(y(1)) + 1) \quad \text{Set } Y_{i+1} = T/(Y_i + 1).
\]

\[
\text{call mpinv}(s, y, n, n) \quad \text{call mpmul}(t, x, s, n, n)
\]

\[
\text{Form temporary } T = \left( X_{i+1} + 1 \right)/(Y_i + 1). \quad \text{If } T = 1 \text{ then we have converged.}
\]

\[
m = \text{mod}(255 + \text{ichar}(t(2)), 256) \quad \text{do } j = 3, n \quad \text{if (ichar}(t(j)) \neq n) \text{ goto 2}
\]

\[
\text{enddo} \quad \text{if (aba(ichar}(t(n+1)) = -m) \text{ goto 2}
\]

\[
\text{write (*,'(1x,64a1)') (s(j)),j=1,m+1) \quad \text{return}
\]

\[
\text{2 continue}
\]

\[
\text{call mpmul}(s, p, t, n, n) \quad \text{Set } \pi_{i+1} = T \pi_i.
\]

\[
\text{call mpmov}(p, s, n) \quad \text{goto 1}
\]

\[
\text{END}
\]
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]

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

(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 $7.416 \times 10^{98}$) [4].

CITED REFERENCES AND FURTHER READING: