# Chapter B20. Less-Numerical Algorithms

f90 Volume 1's Fortran 77 routine machar performed various clever contortions (due to Cody, Malcolm, and others) to discover the underlying properties of a machine's floating-point representation. Fortran 90, by contrast, provides a built-in set of "numeric inquiry functions" that accomplish the same goal. The routine machar included here makes use of these and is included largely for compatibility with the previous version.

```
SUBROUTINE machar(ibeta,it,irnd,ngrd,machep,negep,iexp,minexp,&
    maxexp,eps,epsneg,xmin,xmax)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ibeta,iexp,irnd,it,machep,maxexp,minexp,negep,ngrd
REAL(SP), INTENT(OUT) :: eps,epsneg,xmax,xmin
REAL(SP), PARAMETER :: RX=1.0
```

Determines and returns machine-specific parameters affecting floating-point arithmetic. Returned values include ibeta, the floating-point radix; it, the number of base-ibeta digits in the floating-point mantissa; eps, the smallest positive number that, added to 1.0, is not equal to 1.0; epsneg, the smallest positive number that, subtracted from 1.0, is not equal to 1.0; xmin, the smallest representable positive number; and xmax, the largest representable positive number. See text for description of other returned parameters. Change all REAL(SP) declarations to REAL(DP) to find double-precision parameters.

```
REAL(SP) :: a,beta,betah,one,temp,tempa,two,zero
ibeta=radix(RX)                             Most of the parameters are easily determined
it=digits(RX)                                   from intrinsic functions.
machep=exponent(nearest(RX,RX)-RX)-1
negep=exponent(nearest(RX,-RX)-RX)-1
minexp=minexponent(RX)-1
maxexp=maxexponent(RX)
iexp=nint(log(real(maxexp-minexp+2,sp))/log(2.0_sp))
eps=real(ibeta,sp)**machep
epsneg=real(ibeta,sp)**negep
xmax=huge(RX)
xmin=tiny(RX)
one=RX                                      Determine irnd.
two=one+one
zero=one-one
beta=real(ibeta,sp)
a=beta**(-negep)
irnd=0
betah=beta/two
temp=a+betah
if (temp-a /= zero) irnd=1
tempa=a+beta
temp=tempa+betah
if ((irnd == 0) .and. (temp-tempa /= zero)) irnd=2
ngrd=0                                      Determine ngrd.
```

```
temp=one+eps
if ((irnd == 0) .and. (temp*one-one /= zero)) ngrd=1
temp=xmin/two
if (temp /= zero) irnd=irnd+3          Adjust irnd to reflect partial underflow.
END SUBROUTINE machar
```

⋆ ⋆ ⋆

```
FUNCTION igray(n,is)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,is
INTEGER(I4B) :: igray
   For zero or positive values of is, return the Gray code of n; if is is negative, return the
   inverse Gray code of n.
INTEGER(I4B) :: idiv,ish
if (is >= 0) then            This is the easy direction!
    igray=ieor(n,n/2)
else                         This is the more complicated direction: In hierarchical stages,
    ish=-1                       starting with a one-bit right shift, cause each bit to be
    igray=n                      XORed with all more significant bits.
    do
        idiv=ishft(igray,ish)
        igray=ieor(igray,idiv)
        if (idiv <= 1 .or. ish == -16) RETURN
        ish=ish+ish          Double the amount of shift on the next cycle.
    end do
end if
END FUNCTION igray
```

⋆ ⋆ ⋆

```
FUNCTION icrc(crc,buf,jinit,jrev)
USE nrtype
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(IN) :: buf
INTEGER(I2B), INTENT(IN) :: crc,jinit
INTEGER(I4B), INTENT(IN) :: jrev
INTEGER(I2B) :: icrc
   Computes a 16-bit Cyclic Redundancy Check for an array buf of bytes, using any of several
   conventions as determined by the settings of jinit and jrev (see accompanying table).
   The result is returned both as an integer icrc and as a 2-byte array crc. If jinit is neg-
   ative, then crc is used on input to initialize the remainder register, in effect concatenating
   buf to the previous call.
INTEGER(I4B), SAVE :: init=0
INTEGER(I2B) :: j,cword,ich
INTEGER(I2B), DIMENSION(0:255), SAVE :: icrctb,rchr
INTEGER(I2B), DIMENSION(0:15) :: it = &      Table of 4-bit bit-reverses.
    (/ 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15 /)
if (init == 0) then                          Do we need to initialize tables?
    init=1
    do j=0,255                               The two tables are: CRCs of all characters,
        icrctb(j)=icrc1(ishft(j,8),char(0))      and bit-reverses of all characters.
        rchr(j)=ishft(it(iand(j,15_I2B)),4)+it(ishft(j,-4))
    end do
end if
cword=crc
if (jinit >= 0) then                         Initialize the remainder register.
    cword=ior(jinit,ishft(jinit,8))
```

```
else if (jrev < 0) then                  If not initializing, do we reverse the register?
    cword=ior(rchr(hibyte()),ishft(rchr(lobyte()),8))
end if
do j=1,size(buf)                         Main loop over the characters in the array.
    ich=ichar(buf(j))
    if (jrev < 0) ich=rchr(ich)
    cword=ieor(icrctb(ieor(ich,hibyte())),ishft(lobyte(),8))
end do
icrc=merge(cword, &                      Do we need to reverse the output?
    ior(rchr(hibyte()),ishft(rchr(lobyte()),8)), jrev >= 0)
CONTAINS

FUNCTION hibyte()
INTEGER(I2B) :: hibyte
  Extracts the high byte of the 2-byte integer cword.
hibyte = ishft(cword,-8)
END FUNCTION hibyte

FUNCTION lobyte()
INTEGER(I2B) :: lobyte
  Extracts the low byte of the 2-byte integer cword.
lobyte = iand(cword,255_I2B)
END FUNCTION lobyte

FUNCTION icrc1(crc,onech)
INTEGER(I2B), INTENT(IN) :: crc
CHARACTER(1), INTENT(IN) :: onech
INTEGER(I2B) :: icrc1
  Given a remainder up to now, return the new CRC after one character is added. This routine is
  functionally equivalent to icrc(,,-1,1), but slower. It is used by icrc to initialize its table.
INTEGER(I2B) :: i,ich, bit16, ccitt
DATA bit16,ccitt /Z'8000', Z'1021'/
ich=ichar(onech)                         Here is where the character is folded into the
icrc1=ieor(crc,ishft(ich,8))                 register.
do i=1,8                                 Here is where 8 one-bit shifts, and some XORs
    icrc1=merge(ieor(ccitt,ishft(icrc1,1)), &     with the generator polynomial,
        ishft(icrc1,1), iand(icrc1,bit16) /= 0)   are done.
end do
END FUNCTION icrc1
END FUNCTION icrc
```

The embedded functions `hibyte` and `lobyte` always act on the same variable, `cword`. Thus they don't need any explicit argument.

⋆ ⋆ ⋆

```
FUNCTION decchk(string,ch)
USE nrtype; USE nrutil, ONLY : ifirstloc
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(IN) :: string
CHARACTER(1), INTENT(OUT) :: ch
LOGICAL(LGT) :: decchk
    Decimal check digit computation or verification. Returns as ch a check digit for appending
    to string. In this mode, ignore the returned logical value. If string already ends with
    a check digit, returns the function value .true. if the check digit is valid, otherwise
    .false. In this mode, ignore the returned value of ch. Note that string and ch contain
    ASCII characters corresponding to the digits 0-9, not byte values in that range. Other ASCII
    characters are allowed in string, and are ignored in calculating the check digit.
INTEGER(I4B) :: i,j,k,m
INTEGER(I4B) :: ip(0:9,0:7) = reshape((/ &        Group multiplication and permuta-
    0,1,2,3,4,5,6,7,8,9,1,5,7,6,2,8,3,0,9,4,&        tion tables.
    5,8,0,3,7,9,6,1,4,2,8,9,1,6,0,4,3,5,2,7,9,4,5,3,1,2,6,8,7,0,&
    4,2,8,6,5,7,3,9,0,1,2,7,9,3,8,0,6,4,1,5,7,0,4,6,9,1,3,2,5,8 /),&
```

```
    (/ 10,8 /) )
INTEGER(I4B) :: ij(0:9,0:9) = reshape((/ &
    0,1,2,3,4,5,6,7,8,9,1,2,3,4,0,9,5,6,7,8,2,3,4,0,1,8,9,5,6,&
    7,3,4,0,1,2,7,8,9,5,6,4,0,1,2,3,6,7,8,9,5,5,6,7,8,9,0,1,2,3,&
    4,6,7,8,9,5,4,0,1,2,3,7,8,9,5,6,3,4,0,1,2,8,9,5,6,7,2,3,4,0,&
    1,9,5,6,7,8,1,2,3,4,0 /),(/ 10,10 /))
k=0
m=0
do j=1,size(string)                         Look at successive characters.
    i=ichar(string(j))
    if (i >= 48 .and. i <= 57) then         Ignore everything except digits.
        k=ij(k,ip(mod(i+2,10),mod(m,8)))
        m=m+1
    end if
end do
decchk=logical(k == 0,kind=lgt)
i=mod(m,8)                                  Find which appended digit will check prop-
i=ifirstloc(ij(k,ip(0:9,i)) == 0)-1             erly.
ch=char(i+48)                               Convert to ASCII.
END FUNCTION decchk
```

Note the use of the utility function `ifirstloc` to find the first (in this case, the only) correct check digit.

⋆ ⋆ ⋆

The Huffman and arithmetic coding routines exemplify the use of modules to encapsulate user-defined data types. In these algorithms, "the code" is a fairly complicated construct containing scalar and array data. We define types `huffcode` and `arithcode`, then can pass "the code" from the routine that constructs it to the routine that uses it as a single variable.

```
MODULE huf_info
USE nrtype
IMPLICIT NONE
TYPE huffcode
    INTEGER(I4B) :: nch,nodemax
    INTEGER(I4B), DIMENSION(:), POINTER :: icode,left,iright,ncode
END TYPE huffcode
CONTAINS
SUBROUTINE huff_allocate(hcode,mc)
USE nrtype
IMPLICIT NONE
TYPE(huffcode) :: hcode
INTEGER(I4B) :: mc
INTEGER(I4B) :: mq
mq=2*mc-1
allocate(hcode%icode(mq),hcode%ncode(mq),hcode%left(mq),hcode%iright(mq))
hcode%icode(:)=0
hcode%ncode(:)=0
END SUBROUTINE huff_allocate

SUBROUTINE huff_deallocate(hcode)
USE nrtype
IMPLICIT NONE
TYPE(huffcode) :: hcode
deallocate(hcode%iright,hcode%left,hcode%ncode,hcode%icode)
nullify(hcode%icode)
nullify(hcode%ncode)
nullify(hcode%left)
nullify(hcode%iright)
```

```
END SUBROUTINE huff_deallocate
END MODULE huf_info


SUBROUTINE hufmak(nfreq,ilong,nlong,hcode)
USE nrtype; USE nrutil, ONLY : array_copy,arth,imaxloc,nrerror
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ilong,nlong
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nfreq
TYPE(huffcode) :: hcode
   Given the frequency of occurrence table nfreq of size(nfreq) characters, return the
   Huffman code hcode. Returned values ilong and nlong are the character number that
   produced the longest code symbol, and the length of that symbol.
INTEGER(I4B) :: ibit,j,k,n,node,nused,nerr
INTEGER(I4B), DIMENSION(2*size(nfreq)-1) :: indx,iup,nprob
hcode%nch=size(nfreq)             Initialization.
call huff_allocate(hcode,size(nfreq))
nused=0
nprob(1:hcode%nch)=nfreq(1:hcode%nch)
call array_copy(pack(arth(1,1,hcode%nch), nfreq(1:hcode%nch) /= 0 ),&
    indx,nused,nerr)
do j=nused,1,-1                   Sort nprob into a heap structure in indx.
    call hufapp(j)
end do
k=hcode%nch
do                                Combine heap nodes, remaking the heap at each stage.
    if (nused <= 1) exit
    node=indx(1)
    indx(1)=indx(nused)
    nused=nused-1
    call hufapp(1)
    k=k+1
    nprob(k)=nprob(indx(1))+nprob(node)
    hcode%left(k)=node            Store left and right children of a node.
    hcode%iright(k)=indx(1)
    iup(indx(1))=-k               Indicate whether a node is a left or right child of its par-
    iup(node)=k                       ent.
    indx(1)=k
    call hufapp(1)
end do
hcode%nodemax=k
iup(hcode%nodemax)=0
do j=1,hcode%nch                  Make the Huffman code from the tree.
    if (nprob(j) /= 0) then
        n=0
        ibit=0
        node=iup(j)
        do
            if (node == 0) exit
            if (node < 0) then
                n=ibset(n,ibit)
                node=-node
            end if
            node=iup(node)
            ibit=ibit+1
        end do
        hcode%icode(j)=n
        hcode%ncode(j)=ibit
    end if
end do
ilong=imaxloc(hcode%ncode(1:hcode%nch))
nlong=hcode%ncode(ilong)
```

```
if (nlong > bit_size(1_i4b)) call &       Check nlong not larger than word length.
    nrerror('hufmak: Number of possible bits for code exceeded')
CONTAINS
SUBROUTINE hufapp(l)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: l
  Used by hufmak to maintain a heap structure in the array indx(1:l).
INTEGER(I4B) :: i,j,k,n
n=nused
i=l
k=indx(i)
do
    if (i > n/2) exit
    j=i+i
    if (j < n .and. nprob(indx(j)) > nprob(indx(j+1))) &
        j=j+1
    if (nprob(k) <= nprob(indx(j))) exit
    indx(i)=indx(j)
    i=j
end do
indx(i)=k
END SUBROUTINE hufapp
END SUBROUTINE hufmak
```

```
SUBROUTINE hufenc(ich,codep,nb,hcode)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ich
INTEGER(I4B), INTENT(INOUT) :: nb
CHARACTER(1), DIMENSION(:), POINTER :: codep
TYPE(huffcode) :: hcode
    Huffman encode the single character ich (in the range 0..nch-1) using the code in hcode,
    write the result to the character array pointed to by codep starting at bit nb (whose smallest
    valid value is zero), and increment nb appropriately. This routine is called repeatedly to
    encode consecutive characters in a message, but must be preceded by a single initializing
    call to hufmak.
INTEGER(I4B) :: k,l,n,nc,ntmp
k=ich+1                                      Convert character range 0..nch-1 to ar-
if (k > hcode%nch .or. k < 1) call &             ray index range 1..nch.
    nrerror('hufenc: ich out of range')
do n=hcode%ncode(k),1,-1                     Loop over the bits in the stored Huffman
    nc=nb/8+1                                    code for ich.
    if (nc > size(codep)) codep=>reallocate(codep,2*size(codep))
    l=mod(nb,8)
    if (l == 0) codep(nc)=char(0)
    if (btest(hcode%icode(k),n-1)) then      Set appropriate bits in codep.
        ntmp=ibset(ichar(codep(nc)),l)
        codep(nc)=char(ntmp)
    end if
    nb=nb+1
end do
END SUBROUTINE hufenc
```

```
SUBROUTINE hufdec(ich,code,nb,hcode)
USE nrtype
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ich
INTEGER(I4B), INTENT(INOUT) :: nb
CHARACTER(1), DIMENSION(:), INTENT(IN) :: code
TYPE(huffcode) :: hcode
```

Starting at bit number nb in the character array code, use the Huffman code in hcode to decode a single character (returned as ich in the range 0..nch-1) and increment nb appropriately. Repeated calls, starting with $nb = 0$, will return successive characters in a compressed message. The returned value ich=nch indicates end-of-message. This routine must be preceded by a single initializing call to hufmak.

```
INTEGER(I4B) :: l,nc,node
node=hcode%nodemax                          Set node to the top of the decoding tree.
do                                          Loop until a valid character is obtained.
    nc=nb/8+1
    if (nc > size(code)) then               Ran out of input; return with ich=nch
        ich=hcode%nch                           indicating end of message.
        RETURN
    end if
    l=mod(nb,8)                             Now decoding this bit.
    nb=nb+1
    if (btest(ichar(code(nc)),l)) then      Branch left or right in tree, depending on
        node=hcode%iright(node)                 its value.
    else
        node=hcode%left(node)
    end if
    if (node <= hcode%nch) then             If we reach a terminal node, we have a
        ich=node-1                              complete character and can return.
        RETURN
    end if
end do
END SUBROUTINE hufdec
```

⋆ ⋆ ⋆

```
MODULE arcode_info
USE nrtype
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NWK=20
```

NWK is the number of working digits (see text).

```
TYPE arithcode
    INTEGER(I4B), DIMENSION(:), POINTER :: ilob,iupb,ncumfq
    INTEGER(I4B) :: jdif,nc,minint,nch,ncum,nrad
END TYPE arithcode
CONTAINS
SUBROUTINE arcode_allocate(acode,mc)
USE nrtype
IMPLICIT NONE
TYPE(arithcode) :: acode
INTEGER(I4B) :: mc
allocate(acode%ilob(NWK),acode%iupb(NWK),acode%ncumfq(mc+2))
END SUBROUTINE arcode_allocate

SUBROUTINE arcode_deallocate(acode)
USE nrtype
IMPLICIT NONE
TYPE(arithcode) :: acode
deallocate(acode%ncumfq,acode%iupb,acode%ilob)
nullify(acode%ilob)
nullify(acode%iupb)
```

```
nullify(acode%ncumfq)
END SUBROUTINE arcode_deallocate
END MODULE arcode_info
```

```
SUBROUTINE arcmak(nfreq,nradd,acode)
USE nrtype; USE nrutil, ONLY : cumsum,nrerror
USE arcode_info
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nradd
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nfreq
TYPE(arithcode) :: acode
INTEGER(I4B), PARAMETER :: MAXINT=huge(nradd)
```

Given a table nfreq of the frequency of occurrence of size(nfreq) symbols, and given a desired output radix nradd, initialize the cumulative frequency table and other variables for arithmetic compression. Store the code in acode.

MAXINT is a large positive integer that does not overflow.

```
if (nradd > 256) call nrerror('output radix may not exceed 256 in arcmak')
acode%minint=MAXINT/nradd
acode%nch=size(nfreq)
acode%nrad=nradd
call arcode_allocate(acode,acode%nch)
acode%ncumfq(1)=0
acode%ncumfq(2:acode%nch+1)=cumsum(max(nfreq(1:acode%nch),1))
acode%ncumfq(acode%nch+2)=acode%ncumfq(acode%nch+1)+1
acode%ncum=acode%ncumfq(acode%nch+2)
END SUBROUTINE arcmak
```

```
SUBROUTINE arcode(ich,codep,lcd,isign,acode)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE arcode_info
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: ich,lcd
INTEGER(I4B), INTENT(IN) :: isign
CHARACTER(1), DIMENSION(:), POINTER :: codep
TYPE(arithcode) :: acode
```

Compress ($isign = 1$) or decompress ($isign = -1$) the single character ich into or out of the character array pointed to by codep, starting with byte codep(lcd) and (if necessary) incrementing lcd so that, on return, lcd points to the first unused byte in codep. Note that this routine saves the result of previous calls until a new byte of code is produced, and only then increments lcd. An initializing call with isign=0 is required for each different array codep. The routine arcmak must have previously been called to initialize the code acode. A call with ich=arcode%nch (as set in arcmak) has the reserved meaning "end of message."

```
INTEGER(I4B) :: ihi,j,ja,jh,jl,m
if (isign == 0) then                     Initialize enough digits of the upper and lower
    acode%jdif=acode%nrad-1                  bounds.
    acode%ilob(:)=0
    acode%iupb(:)=acode%nrad-1
    do j=NWK,1,-1
        acode%nc=j
        if (acode%jdif > acode%minint) RETURN       Initialization complete.
        acode%jdif=(acode%jdif+1)*acode%nrad-1
    end do
    call nrerror('NWK too small in arcode')
else
    if (isign > 0) then                  If encoding, check for valid input character.
        if (ich > acode%nch .or. ich < 0) call nrerror('bad ich in arcode')
    else                                 If decoding, locate the character ich by bi-
        ja=ichar(codep(lcd))-acode%ilob(acode%nc)        section.
        do j=acode%nc+1,NWK
```

```
            ja=ja*acode%nrad+(ichar(codep(j+lcd-acode%nc))-acode%ilob(j))
        end do
        ich=0
        ihi=acode%nch+1
        do
            if (ihi-ich <= 1) exit
            m=(ich+ihi)/2
            if (ja >= jtry(acode%jdif,acode%ncumfq(m+1),acode%ncum)) then
                ich=m
            else
                ihi=m
            end if
        end do
        if (ich == acode%nch) RETURN       Detected end of message.
    end if
      Following code is common for encoding and decoding. Convert character ich to a new
      subrange [ilob,iupb).
    jh=jtry(acode%jdif,acode%ncumfq(ich+2),acode%ncum)
    jl=jtry(acode%jdif,acode%ncumfq(ich+1),acode%ncum)
    acode%jdif=jh-jl
    call arcsum(acode%ilob,acode%iupb,jh,NWK,acode%nrad,acode%nc)
      How many leading digits to output (if encoding) or skip over?
    call arcsum(acode%ilob,acode%ilob,jl,NWK,acode%nrad,acode%nc)
    do j=acode%nc,NWK
        if (ich /= acode%nch .and. acode%iupb(j) /= acode%ilob(j)) exit
        if (acode%nc > size(codep)) codep=>reallocate(codep,2*size(codep))
        if (isign > 0) codep(lcd)=char(acode%ilob(j))
        lcd=lcd+1
    end do
    if (j > NWK) RETURN                Ran out of message. Did someone forget to
    acode%nc=j                             encode a terminating ncd?
    j=0                                How many digits to shift?
    do
        if (acode%jdif >= acode%minint) exit
        j=j+1
        acode%jdif=acode%jdif*acode%nrad
    end do
    if (acode%nc-j < 1) call nrerror('NWK too small in arcode')
    if (j /= 0) then                   Shift them.
        acode%iupb((acode%nc-j):(NWK-j))=acode%iupb(acode%nc:NWK)
        acode%ilob((acode%nc-j):(NWK-j))=acode%ilob(acode%nc:NWK)
    end if
    acode%nc=acode%nc-j
    acode%iupb((NWK-j+1):NWK)=0
    acode%ilob((NWK-j+1):NWK)=0
end if                                 Normal return.
CONTAINS

FUNCTION jtry(m,n,k)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: m,n,k
INTEGER(I4B) :: jtry
  Calculate (m*n)/k without overflow. Program efficiency can be improved by substituting an
  assembly language routine that does integer multiply to a double register.
jtry=int((real(m,dp)*real(n,dp))/real(k,dp))
END FUNCTION jtry

SUBROUTINE arcsum(iin,iout,ja,nwk,nrad,nc)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iin
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: iout
INTEGER(I4B), INTENT(IN) :: nwk,nrad,nc
INTEGER(I4B), INTENT(INOUT) :: ja
```

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

```
INTEGER(I4B) :: j,jtmp,karry
karry=0
do j=nwk,nc+1,-1
    jtmp=ja
    ja=ja/nrad
    iout(j)=iin(j)+(jtmp-ja*nrad)+karry
    if (iout(j) >= nrad) then
        iout(j)=iout(j)-nrad
        karry=1
    else
        karry=0
    end if
end do
iout(nc)=iin(nc)+ja+karry
END SUBROUTINE arcsum
END SUBROUTINE arcode
```

⋆ ⋆ ⋆

***MODULE mpops***

```
USE nrtype
INTEGER(I4B), PARAMETER :: NPAR_ICARRY=64
CONTAINS
SUBROUTINE icarry(karry,isum,nbits)
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: karry
```

Perform deferred carry operation on an array isum of multiple-precision digits. Nonzero bits of higher order than nbits (typically 8) are carried to the next-lower (leftward) component of isum. The final (most leftward) carry value is returned as karry.

```
INTEGER(I2B), DIMENSION(:), INTENT(INOUT) :: isum
INTEGER(I4B), INTENT(IN) :: nbits
INTEGER(I4B) :: n,j
INTEGER(I2B), DIMENSION(size(isum)) :: ihi
INTEGER(I2B) :: mb,ihh
n=size(isum)
mb=ishft(1,nbits)-1                   Make mask for low-order bits.
karry=0
if (n < NPAR_ICARRY ) then
    do j=n,2,-1                       Keep going until all carries have cascaded.
        ihh=ishft(isum(j),-nbits)
        if (ihh /= 0) then
            isum(j)=iand(isum(j),mb)
            isum(j-1)=isum(j-1)+ihh
        end if
    end do
    ihh=ishft(isum(1),-nbits)
    isum(1)=iand(isum(1),mb)
    karry=karry+ihh
else
    do
        ihi=ishft(isum,-nbits)        Get high bits.
        if (all(ihi == 0)) exit       Check if done.
        where (ihi /= 0) isum=iand(isum,mb)      Remove bits to be carried and add
        where (ihi(2:n) /= 0) isum(1:n-1)=isum(1:n-1)+ihi(2:n)    them to left.
        karry=karry+ihi(1)            Final carry.
    end do
end if
END SUBROUTINE icarry
```

```
SUBROUTINE mpadd(w,u,v,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: n
```
Adds the unsigned radix 256 integers u(1:n) and v(1:n) yielding the unsigned integer w(1:n+1).
```
INTEGER(I2B), DIMENSION(n) :: isum
INTEGER(I4B) :: karry
isum=ichar(u(1:n))+ichar(v(1:n))
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpadd
```

```
SUBROUTINE mpsub(is,w,u,v,n)
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: is
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: 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 $0$.
```
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=255+ichar(u(1:n))-ichar(v(1:n))
isum(n)=isum(n)+1
call icarry(karry,isum,8_I4B)
w(1:n)=char(isum)
is=karry-1
END SUBROUTINE mpsub
```

```
SUBROUTINE mpsad(w,u,n,iv)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: 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).
```
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=ichar(u(1:n))
isum(n)=isum(n)+iv
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpsad
```

```
SUBROUTINE mpsmu(w,u,n,iv)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: 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).
```
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=ichar(u(1:n))*iv
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpsmu
```

```
SUBROUTINE mpneg(u,n)
IMPLICIT NONE
```

```
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: n
   Ones-complement negate the unsigned radix 256 integer u(1:n).
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=255-ichar(u(1:n))
isum(n)=isum(n)+1
call icarry(karry,isum,8_I4B)
u(1:n)=char(isum)
END SUBROUTINE mpneg

SUBROUTINE mplsh(u,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: n
   Left shift u(2..n+1) onto u(1:n).
u(1:n)=u(2:n+1)
END SUBROUTINE mplsh

SUBROUTINE mpmov(u,v,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
INTEGER(I4B), INTENT(IN) :: n
   Move v(1:n) onto u(1:n).
u(1:n)=v(1:n)
END SUBROUTINE mpmov

SUBROUTINE mpsdv(w,u,n,iv,ir)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: n,iv
INTEGER(I4B), INTENT(OUT) :: ir
   Short division: The unsigned radix 256 integer u(1:n) is divided by the integer iv (in the
   range 0 ≤ iv ≤ 255), yielding a quotient w(1:n) and a remainder ir (with 0 ≤ ir ≤ 255).
   Note: Your Numerical Recipes authors don't know how to parallelize this routine in Fortran
   90!
INTEGER(I4B) :: i,j
ir=0
do j=1,n
    i=256*ir+ichar(u(j))
    w(j)=char(i/iv)
    ir=mod(i,iv)
end do
END SUBROUTINE mpsdv
END MODULE mpops
```

```
SUBROUTINE mpmul(w,u,v,n,m)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : realft
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,m
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
!   The logical dimensions are: CHARACTER(1) :: w(n+m),u(n),v(m)
REAL(DP), PARAMETER :: RX=256.0
   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(I4B) :: j,mn,nn
REAL(DP) :: cy,t
REAL(DP), DIMENSION(:), ALLOCATABLE :: a,b,tb
mn=max(m,n)
nn=1                              Find the smallest useable power of two for the transform.
```

```
do
    if (nn >= mn) exit
    nn=nn+nn
end do
nn=nn+nn
allocate(a(nn),b(nn),tb((nn-1)/2))
a(1:n)=ichar(u(1:n))               Move U to a double-precision floating array.
a(n+1:nn)=0.0
b(1:m)=ichar(v(1:m))               Move V to a double-precision floating array.
b(m+1:nn)=0.0
call realft(a(1:nn),1)             Perform the convolution: First, the two Fourier trans-
call realft(b(1:nn),1)                 forms.
b(1)=b(1)*a(1)                     Then multiply the complex results (real and imaginary
b(2)=b(2)*a(2)                         parts).
tb=b(3:nn:2)
b(3:nn:2)=tb*a(3:nn:2)-b(4:nn:2)*a(4:nn:2)
b(4:nn:2)=tb*a(4:nn:2)+b(4:nn:2)*a(3:nn:2)
call realft(b(1:nn),-1)            Then do the inverse Fourier transform.
b(:)=b(:)/(nn/2)
cy=0.0                             Make a final pass to do all the carries.
do j=nn,1,-1
    t=b(j)+cy+0.5_dp               The 0.5 allows for roundoff error.
    b(j)=mod(t,RX)
    cy=int(t/RX)
end do
if (cy >= RX) call nrerror('mpmul: sanity check failed in fftmul')
w(1)=char(int(cy))                 Copy answer to output.
w(2:(n+m))=char(int(b(1:(n+m-1))))
deallocate(a,b,tb)
END SUBROUTINE mpmul
```

```
SUBROUTINE mpinv(u,v,n,m)
USE nrtype; USE nrutil, ONLY : poly
USE nr, ONLY : mpmul
USE mpops, ONLY : mpmov,mpneg
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B), PARAMETER :: MF=4
REAL(SP), PARAMETER :: BI=1.0_sp/256.0_sp
```

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).

```
INTEGER(I4B) :: i,j,mm
REAL(SP) :: fu
CHARACTER(1), DIMENSION(:), ALLOCATABLE :: rr,s
allocate(rr(max(n,m)+n+1),s(n))
mm=min(MF,m)
fu=1.0_sp/poly(BI,real(ichar(v(:)),sp))      Use ordinary floating arithmetic to get an
do j=1,n                                          initial approximation.
    i=int(fu)
    u(j)=char(i)
    fu=256.0_sp*(fu-i)
end do
do                                           Iterate Newton's rule to convergence.
    call mpmul(rr,u,v,n,m)                   Construct 2 - UV in S.
    call mpmov(s,rr(2:),n)
    call mpneg(s,n)
    s(1)=char(ichar(s(1))-254)               Multiply SU into U.
    call mpmul(rr,s,u,n,n)
    call mpmov(u,rr(2:),n)
```

```
    if (all(ichar(s(2:n-1)) == 0)) exit       If fractional part of S is not zero, it has
end do                                            not converged to 1.
deallocate(rr,s)
END SUBROUTINE mpinv


SUBROUTINE mpdiv(q,r,u,v,n,m)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : mpinv,mpmul
USE mpops, ONLY : mpsad,mpmov,mpsub
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: q,r
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
!   The logical dimensions are: CHARACTER(1) :: q(n-m+1),r(m),u(n),v(m)
INTEGER(I4B), INTENT(IN) :: n,m
   Divides unsigned radix 256 integers u(1:n) by v(1:m) (with m ≤ n required), yielding a
   quotient q(1:n-m+1) and a remainder r(1:m).
INTEGER(I4B), PARAMETER :: MACC=6
INTEGER(I4B) :: is
CHARACTER(1), DIMENSION(:), ALLOCATABLE, TARGET :: rr,s
CHARACTER(1), DIMENSION(:), POINTER :: rr2,s3
allocate(rr(2*(n+MACC)),s(2*(n+MACC)))
rr2=>rr(2:)
s3=>s(3:)
call mpinv(s,v,n+MACC,m)            Set S = 1/V.
call mpmul(rr,s,u,n+MACC,n)         Set Q = SU.
call mpsad(s,rr,n+n+MACC/2,1)
call mpmov(q,s3,n-m+1)
call mpmul(rr,q,v,n-m+1,m)          Multiply and subtract to get the remainder.
call mpsub(is,rr2,u,rr2,n)
if (is /= 0) call nrerror('MACC too small in mpdiv')
call mpmov(r,rr(n-m+2:),m)
deallocate(rr,s)
END SUBROUTINE mpdiv



SUBROUTINE mpsqrt(w,u,v,n,m)
USE nrtype; USE nrutil, ONLY : poly
USE nr, ONLY : mpmul
USE mpops, ONLY : mplsh,mpmov,mpneg,mpsdv
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w,u
CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B), PARAMETER :: MF=3
REAL(SP), PARAMETER :: BI=1.0_sp/256.0_sp
   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(I4B) :: i,ir,j,mm
REAL(SP) :: fu
CHARACTER(1), DIMENSION(:), ALLOCATABLE :: r,s
allocate(r(2*n),s(2*n))
mm=min(m,MF)
fu=1.0_sp/sqrt(poly(BI,real(ichar(v(:)),sp)))      Use ordinary floating arithmetic
do j=1,n                                                to get an initial approxima-
    i=int(fu)                                           tion.
    u(j)=char(i)
    fu=256.0_sp*(fu-i)
end do
do                                          Iterate Newton's rule to convergence.
    call mpmul(r,u,u,n,n)                   Construct S = (3 − VU^2)/2.
```

```
    call mplsh(r,n)
    call mpmul(s,r,v,n,min(m,n))
    call mplsh(s,n)
    call mpneg(s,n)
    s(1)=char(ichar(s(1))-253)
    call mpsdv(s,s,n,2,ir)
    if (any(ichar(s(2:n-1)) /= 0)) then
        If fractional part of S is not zero, it has not converged to 1.
        call mpmul(r,s,u,n,n)              Replace U by SU.
        call mpmov(u,r(2:),n)
        cycle
    end if
    call mpmul(r,u,v,n,min(m,n))           Get square root from reciprocal and return.
    call mpmov(w,r(2:),n)
    deallocate(r,s)
    RETURN
end do
END SUBROUTINE mpsqrt
```

```
SUBROUTINE mp2dfr(a,s,n,m)
USE nrtype
USE mpops, ONLY : mplsh,mpsmu
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: m
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: a
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: s
INTEGER(I4B), PARAMETER :: IAZ=48
```

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(I4B) :: j
m=int(2.408_sp*n)
do j=1,m
    call mpsmu(a,a,n,10)
    s(j)=char(ichar(a(1))+IAZ)
    call mplsh(a,n)
end do
END SUBROUTINE mp2dfr
```

```
SUBROUTINE mppi(n)
USE nrtype
USE nr, ONLY : mp2dfr,mpinv,mpmul,mpsqrt
USE mpops, ONLY : mpadd,mplsh,mpmov,mpsdv
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), PARAMETER :: IAOFF=48
```

Demonstrate multiple precision routines by calculating and printing the first `n` bytes of $\pi$.

```
INTEGER(I4B) :: ir,j,m
CHARACTER(1), DIMENSION(n) :: sx,sxi
CHARACTER(1), DIMENSION(2*n) :: t,y
CHARACTER(1), DIMENSION(3*n) :: s
CHARACTER(1), DIMENSION(n+1) :: x,bigpi
t(1)=char(2)                          Set T = 2.
t(2:n)=char(0)
call mpsqrt(x,x,t,n,n)                Set X_0 = sqrt(2).
call mpadd(bigpi,t,x,n)               Set pi_0 = 2 + sqrt(2).
call mplsh(bigpi,n)
call mpsqrt(sx,sxi,x,n,n)             Set Y_0 = 2^(1/4).
```

```
call mpmov(y,sx,n)
do
    call mpadd(x,sx,sxi,n)                  Set X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2.
    call mpsdv(x,x(2:),n,2,ir)
    call mpsqrt(sx,sxi,x,n,n)               Form the temporary T = Y_i X_{i+1}^{1/2} + X_{i+1}^{-1/2}.
    call mpmul(t,y,sx,n,n)
    call mpadd(t(2:),t(2:),sxi,n)
    x(1)=char(ichar(x(1))+1)                Increment X_{i+1} and Y_i by 1.
    y(1)=char(ichar(y(1))+1)
    call mpinv(s,y,n,n)                     Set Y_{i+1} = T/(Y_i + 1).
    call mpmul(y,t(3:),s,n,n)
    call mplsh(y,n)
    call mpmul(t,x,s,n,n)                   Form temporary T = (X_{i+1} + 1)/(Y_i + 1).
    m=mod(255+ichar(t(2)),256)              If T = 1 then we have converged.
    if (abs(ichar(t(n+1))-m) > 1 .or. any(ichar(t(3:n)) /= m)) then
        call mpmul(s,bigpi,t(2:),n,n)       Set π_{i+1} = T π_i.
        call mpmov(bigpi,s(2:),n)
        cycle
    end if
    write (*,*) 'pi='
    s(1)=char(ichar(bigpi(1))+IAOFF)
    s(2)='.'
    call mp2dfr(bigpi(2:),s(3:),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.
    write (*,'(1x,64a1)') (s(j),j=1,m+1)
    RETURN
end do
END SUBROUTINE mppi
```