20.5 Arithmetic Coding

We saw in the previous section that a perfect (entropy-bounded) coding scheme would use $L_i = -\log_2 p_i$ bits to encode character i (in the range $1 \le i \le N_{ch}$), if p_i is its probability of occurrence. Huffman coding gives a way of rounding the L_i's to close integer values and constructing a code with those lengths. *Arithmetic coding* [1], which we now discuss, actually does manage to encode characters using noninteger numbers of bits! It also provides a convenient way to output the result not as a stream of bits, but as a stream of symbols in any desired radix. This latter property is particularly useful if you want, e.g., to convert data from bytes (radix 256) to printable ASCII characters (radix 94), or to case-independent alphanumeric sequences containing only A-Z and 0-9 (radix 36).

In arithmetic coding, an input message of any length is represented as a real number R in the range $0 \le R < 1$. The longer the message, the more precision required of R. This is best illustrated by an example, so let us return to the fictitious language, Vowellish, of the previous section. Recall that Vowellish has a 5 character alphabet (A, E, I, O, U), with occurrence probabilities 0.12, 0.42, 0.09, 0.30, and 0.07, respectively. Figure 20.5.1 shows how a message beginning "IOU" is encoded: The interval $[0, 1)$ is divided into segments corresponding to the 5 alphabetical characters; the length of a segment is the corresponding p_i. We see that the first message character, "I", narrows the range of R to $0.37 \le R < 0.46$. This interval is now subdivided into five subintervals, again with lengths proportional to the p_i's. The second message character, "O", narrows the range of R to $0.3763 \le R < 0.4033$. The "U" character further narrows the range to $0.37630 \le R < 0.37819$. *Any* value of R in this range can be sent as encoding "IOU". In particular, the binary fraction $.011000001$ is in this range, so "IOU" can be sent in 9 bits. (Huffman coding took 10 bits for this example, see §20.4.)

Of course there is the problem of knowing when to stop decoding. The fraction $.011000001$ represents not simply "IOU," but "IOU. . . ," where the ellipses represent an infinite string of successor characters. To resolve this ambiguity, arithmetic coding generally assumes the existence of a special $N_{ch} + 1$th character, EOM (end of message), which occurs only once at the end of the input. Since EOM has a low probability of occurrence, it gets allocated only a very tiny piece of the number line.

In the above example, we gave R as a binary fraction. We could just as well have output it in any other radix, e.g., base 94 or base 36, whatever is convenient for the anticipated storage or communication channel.

You might wonder how one deals with the seemingly incredible precision required of R for a long message. The answer is that R is never actually represented all at once. At any give stage we have upper and lower bounds for R represented as a finite number of digits in the output radix. As digits of the upper and lower bounds become identical, we can left-shift them away and bring in new digits at the low-significance end. The routines below have a parameter `NWK` for the number of working digits to keep around. This must be large enough to make the chance of an accidental degeneracy vanishingly small. (The routines signal if a degeneracy ever occurs.) Since the process of discarding old digits and bringing in new ones is performed identically on encoding and decoding, everything stays synchronized.

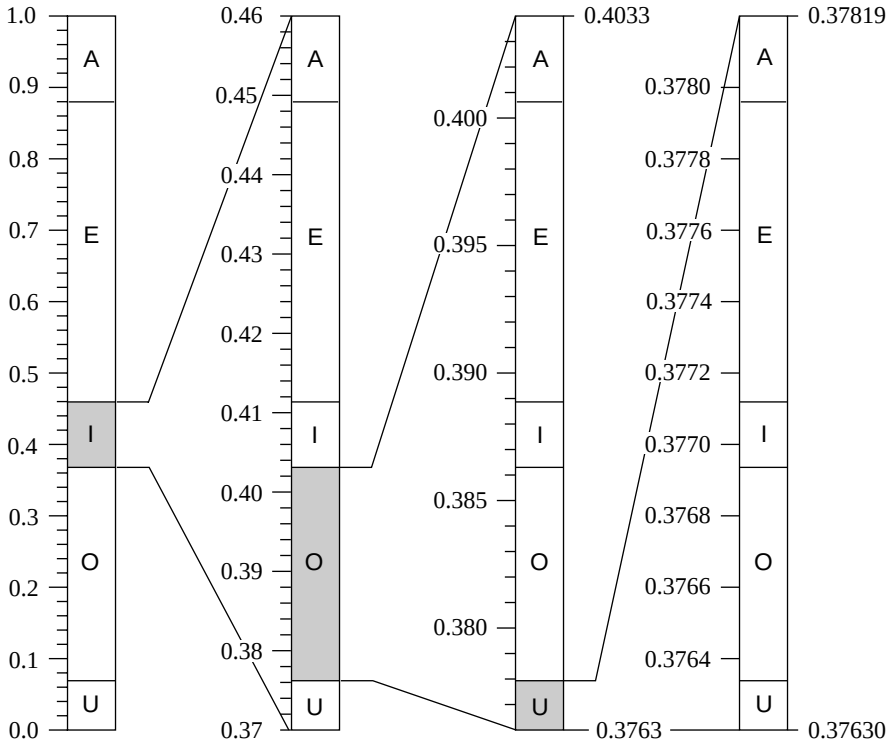

Figure 20.5.1. Arithmetic coding of the message "IOU..." in the fictitious language Vowellish. Successive characters give successively finer subdivisions of the initial interval between 0 and 1. The final value can be output as the digits of a fraction in any desired radix. Note how the subinterval allocated to a character is proportional to its probability of occurrence.

The routine `arcmak` constructs the cumulative frequency distribution table used to partition the interval at each stage. In the principal routine `arcode`, when an interval of size `jdif` is to be partitioned in the proportions of some `n` to some `ntot`, say, then we must compute `(n*jdif)/ntot`. With integer arithmetic, the numerator is likely to overflow; and, unfortunately, an expression like `jdif/(ntot/n)` is not equivalent. In the implementation below, we resort to double precision floating arithmetic for this calculation. Not only is this inefficient, but different roundoff errors can (albeit very rarely) make different machines encode differently, though any one type of machine will decode exactly what it encoded, since identical roundoff errors occur in the two processes. For serious use, one needs to replace this floating calculation with an integer computation in a double register (not available to the `FORTRAN` programmer).

The internally set variable `minint`, which is the minimum allowed number of discrete steps between the upper and lower bounds, determines when new low-significance digits are added. `minint` must be large enough to provide resolution of all the input characters. That is, we must have $p_i \times$ `minint` > 1 for all i. A value of $100N_{ch}$, or $1.1/\min p_i$, whichever is larger, is generally adequate. However, for safety, the routine below takes `minint` to be as large as possible, with the product `minint*nradd` just smaller than overflow. This results in some time inefficiency, and in a few unnecessary characters being output at the end of a message. You can

decrease minint if you want to live closer to the edge.

A final safety feature in arcmak is its refusal to believe zero values in the table nfreq; a 0 is treated as if it were a 1. If this were not done, the occurrence in a message of a single character whose nfreq entry is zero would result in scrambling the entire rest of the message. If you want to live dangerously, with a very slightly more efficient coding, you can delete the max(,1) operation.

```
      SUBROUTINE arcmak(nfreq,nchh,nradd)
      INTEGER nchh,nradd,nfreq(nchh),MC,NWK,MAXINT
      PARAMETER (MC=512,NWK=20,MAXINT=2147483647)
        Given a table nfreq(1:nchh) of the frequency of occurrence of nchh symbols, and given
        a desired output radix nradd, initialize the cumulative frequency table and other variables
        for arithmetic compression.
        Parameters: MC is largest anticipated value of nchh; NWK is the number of working digits
        (see text); MAXINT is a large positive integer that does not overflow.
      INTEGER j,jdif,minint,nc,nch,nrad,ncum,
     *     ncumfq(MC+2),ilob(NWK),iupb(NWK)
      COMMON /arccom/ ncumfq,iupb,ilob,nch,nrad,minint,jdif,nc,ncum
      SAVE /arccom/
      if(nchh.gt.MC)pause 'MC too small in arcmak'
      if(nradd.gt.256)pause 'nradd may not exceed 256 in arcmak'
      minint=MAXINT/nradd
      nch=nchh
      nrad=nradd
      ncumfq(1)=0
      do 11 j=2,nch+1
        ncumfq(j)=ncumfq(j-1)+max(nfreq(j-1),1)
      enddo 11
      ncumfq(nch+2)=ncumfq(nch+1)+1
      ncum=ncumfq(nch+2)
      return
      END
```

Individual characters in a message are coded or decoded by the routine arcode, which in turn uses the utility arcsum.

```
      SUBROUTINE arcode(ich,code,lcode,lcd,isign)
      INTEGER ich,isign,lcd,lcode,MC,NWK
      CHARACTER*1 code(lcode)
      PARAMETER (MC=512,NWK=20)
C     USES arcsum
        Compress (isign = 1) or decompress (isign = −1) the single character ich into or out
        of the character array code(1:lcode), starting with byte code(lcd) and (if necessary)
        incrementing lcd so that, on return, lcd points to the first unused byte in code. 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 code. The routine arcmak must have previously been called to initialize the common
        block /arccom/. A call with ich=nch (as set in arcmak) has the reserved meaning "end
        of message."
      INTEGER ihi,j,ja,jdif,jh,jl,k,m,minint,nc,nch,nrad,ilob(NWK),
     *     iupb(NWK),ncumfq(MC+2),ncum,JTRY
      COMMON /arccom/ ncumfq,iupb,ilob,nch,nrad,minint,jdif,nc,ncum
      SAVE /arccom/
        The following statement function is used to calculate (k*j)/m without overflow. Program
        efficiency can be improved by substituting an assembly language routine that does integer
        multiply to a double register.
      JTRY(j,k,m)=int((dble(k)*dble(j))/dble(m))
      if (isign.eq.0) then                Initialize enough digits of the upper and lower bounds.
        jdif=nrad-1
        do 11 j=NWK,1,-1
```

```
        iupb(j)=nrad-1
        ilob(j)=0
        nc=j
        if(jdif.gt.minint)return    Initialization complete.
        jdif=(jdif+1)*nrad-1
    enddo 11
    pause 'NWK too small in arcode'
else
    if (isign.gt.0) then            If encoding, check for valid input character.
        if(ich.gt.nch.or.ich.lt.0)pause 'bad ich in arcode'
    else                            If decoding, locate the character ich by bisection.
        ja=ichar(code(lcd))-ilob(nc)
        do 12 j=nc+1,NWK
            ja=ja*nrad+(ichar(code(j+lcd-nc))-ilob(j))
        enddo 12
        ich=0
        ihi=nch+1
1       if(ihi-ich.gt.1) then
            m=(ich+ihi)/2
            if (ja.ge.JTRY(jdif,ncumfq(m+1),ncum)) then
                ich=m
            else
                ihi=m
            endif
        goto 1
        endif
        if(ich.eq.nch)return        Detected end of message.
    endif
      Following code is common for encoding and decoding. Convert character ich to a new
      subrange [ilob,iupb).
    jh=JTRY(jdif,ncumfq(ich+2),ncum)
    jl=JTRY(jdif,ncumfq(ich+1),ncum)
    jdif=jh-jl
    call arcsum(ilob,iupb,jh,NWK,nrad,nc)
    call arcsum(ilob,ilob,jl,NWK,nrad,nc)        How many leading digits to output
    do 13 j=nc,NWK                                  (if encoding) or skip over?
        if(ich.ne.nch.and.iupb(j).ne.ilob(j))goto 2
        if(lcd.gt.lcode)pause 'lcode too small in arcode'
        if(isign.gt.0) code(lcd)=char(ilob(j))
        lcd=lcd+1
    enddo 13
    return                          Ran out of message. Did someone forget to encode
2   nc=j                                a terminating ncd?
    j=0                             How many digits to shift?
3   if (jdif.lt.minint) then
        j=j+1
        jdif=jdif*nrad
    goto 3
    endif
    if (nc-j.lt.1) pause 'NWK too small in arcode'
    if(j.ne.0)then                  Shift them.
        do 14 k=nc,NWK
            iupb(k-j)=iupb(k)
            ilob(k-j)=ilob(k)
        enddo 14
    endif
    nc=nc-j
    do 15 k=NWK-j+1,NWK
        iupb(k)=0
        ilob(k)=0
    enddo 15
endif
return                              Normal return.
END
```

```
SUBROUTINE arcsum(iin,iout,ja,nwk,nrad,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
enddo 11
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:

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

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