Chapter 20. Less-Numerical Algorithms

20.0 Introduction

You can stop reading now. You are done with *Numerical Recipes*, as such. This final chapter is an idiosyncratic collection of "*less*-numerical recipes" which, for one reason or another, we have decided to include between the covers of an otherwise *more*-numerically oriented book. Authors of computer science texts, we've noticed, like to throw in a token numerical subject (usually quite a dull one — quadrature, for example). We find that we are not free of the reverse tendency.

Our selection of material is not completely arbitrary. One topic, Gray codes, was already used in the construction of quasi-random sequences (§7.7), and here needs only some additional explication. Two other topics, on diagnosing a computer's floating-point parameters, and on arbitrary precision arithmetic, give additional insight into the machinery behind the casual assumption that computers are useful for doing things with numbers (as opposed to bits or characters). The latter of these topics also shows a very different use for Chapter 12's fast Fourier transform.

The three other topics (checksums, Huffman and arithmetic coding) involve different aspects of data coding, compression, and validation. If you handle a large amount of data — numerical data, even — then a passing familiarity with these subjects might at some point come in handy. In §13.6, for example, we already encountered a good use for Huffman coding.

But again, you don't have to read this chapter. (And you should learn about quadrature from Chapters 4 and 16, not from a computer science text!)

20.1 Diagnosing Machine Parameters

A convenient fiction is that a computer's floating-point arithmetic is "accurate enough." If you believe this fiction, then numerical analysis becomes a very clean subject. Roundoff error disappears from view; many finite algorithms become "exact"; only docile truncation error (§1.3) stands between you and a perfect calculation. Sounds rather naive, doesn't it?

Yes, it is naive. Notwithstanding, it is a fiction necessarily adopted throughout most of this book. To do a good job of answering the question of how roundoff error propagates, or can be bounded, for every algorithm that we have discussed would be impractical. In fact, it would not be possible: Rigorous analysis of many practical algorithms has never been made, by us or anyone.

Proper numerical analysts cringe when they hear a user say, "I was getting roundoff errors with single precision, so I switched to double." The actual meaning is, "for this particular algorithm, and my particular data, double precision *seemed* able to restore my erroneous belief in the 'convenient fiction'." We admit that most of the mentions of precision or roundoff in *Numerical Recipes* are only slightly more quantitative in character. That comes along with our trying to be "practical."

It is important to know what the limitations of your machine's floating-point arithmetic actually are — the more so when your treatment of floating-point roundoff error is going to be intuitive, experimental, or casual. Methods for determining useful floating-point parameters experimentally have been developed by Cody [1], Malcolm [2], and others, and are embodied in the routine `machar`, below, which follows Cody's implementation.

All of `machar`'s arguments are returned values. Here is what they mean:

- `ibeta` (called B in §1.3) is the radix in which numbers are represented, almost always 2, but occasionally 16, or even 10.
- `it` is the number of base-`ibeta` digits in the floating-point mantissa M (see Figure 1.3.1).
- `machep` is the exponent of the smallest (most negative) power of `ibeta` that, added to 1.0, gives something different from 1.0.
- `eps` is the floating-point number $\mathtt{ibeta}^{\mathtt{machep}}$, loosely referred to as the "floating-point precision."
- `negep` is the exponent of the smallest power of `ibeta` that, subtracted from 1.0, gives something different from 1.0.
- `epsneg` is $\mathtt{ibeta}^{\mathtt{negep}}$, another way of defining floating-point precision. Not infrequently `epsneg` is 0.5 times `eps`; occasionally `eps` and `epsneg` are equal.
- `iexp` is the number of bits in the exponent (including its sign or bias).
- `minexp` is the smallest (most negative) power of `ibeta` consistent with there being no leading zeros in the mantissa.
- `xmin` is the floating-point number $\mathtt{ibeta}^{\mathtt{minexp}}$, generally the smallest (in magnitude) useable floating value.
- `maxexp` is the smallest (positive) power of `ibeta` that causes overflow.
- `xmax` is $(1-\mathtt{epsneg})\times\mathtt{ibeta}^{\mathtt{maxexp}}$, generally the largest (in magnitude) useable floating value.
- `irnd` returns a code in the range $0\ldots5$, giving information on what kind of rounding is done in addition, and on how underflow is handled. See below.
- `ngrd` is the number of "guard digits" used when truncating the product of two mantissas to fit the representation.

There is a lot of subtlety in a program like `machar`, whose purpose is to ferret out machine properties that are supposed to be transparent to the user. Further, it must do so avoiding error conditions, like overflow and underflow, that might interrupt its execution. In some cases the program is able to do this only by recognizing certain characteristics of "standard" representations. For example, it recognizes the IEEE standard representation [3] by its rounding behavior, and assumes certain features of its exponent representation as a consequence. We refer you to [1] and

Sample Results Returned by `machar`			
	typical IEEE-compliant machine		DEC VAX
precision	single	double	single
`ibeta`	2	2	2
`it`	24	53	24
`machep`	-23	-52	-24
`eps`	1.19×10^{-7}	2.22×10^{-16}	5.96×10^{-8}
`negep`	-24	-53	-24
`epsneg`	5.96×10^{-8}	1.11×10^{-16}	5.96×10^{-8}
`iexp`	8	11	8
`minexp`	-126	-1022	-128
`xmin`	1.18×10^{-38}	2.23×10^{-308}	2.94×10^{-39}
`maxexp`	128	1024	127
`xmax`	3.40×10^{38}	1.79×10^{308}	1.70×10^{38}
`irnd`	5	5	1
`ngrd`	0	0	0

references therein for details. Be aware that `machar` can give incorrect results on some nonstandard machines.

The parameter `irnd` needs some additional explanation. In the IEEE standard, bit patterns correspond to exact, "representable" numbers. The specified method for rounding an addition is to add two representable numbers "exactly," and then round the sum to the closest representable number. If the sum is precisely halfway between two representable numbers, it should be rounded to the even one (low-order bit zero). The same behavior should hold for all the other arithmetic operations, that is, they should be done in a manner equivalent to infinite precision, and then rounded to the closest representable number.

If `irnd` returns 2 or 5, then your computer is compliant with this standard. If it returns 1 or 4, then it is doing some kind of rounding, but not the IEEE standard. If `irnd` returns 0 or 3, then it is truncating the result, not rounding it — not desirable.

The other issue addressed by `irnd` concerns underflow. If a floating value is less than `xmin`, many computers underflow its value to zero. Values `irnd` $= 0, 1$, or 2 indicate this behavior. The IEEE standard specifies a more graceful kind of underflow: As a value becomes smaller than `xmin`, its exponent is frozen at the smallest allowed value, while its mantissa is decreased, acquiring leading zeros and "gracefully" losing precision. This is indicated by `irnd` $= 3, 4$, or 5.

```
#include <math.h>
#define CONV(i) ((float)(i))
```
Change `float` to `double` here and in declarations below to find double precision parameters.

```
void machar(int *ibeta, int *it, int *irnd, int *ngrd, int *machep, int *negep,
    int *iexp, int *minexp, int *maxexp, float *eps, float *epsneg,
    float *xmin, float *xmax)
```
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.

```
{
    int i,itemp,iz,j,k,mx,nxres;
    float a,b,beta,betah,betain,one,t,temp,temp1,tempa,two,y,z,zero;

    one=CONV(1);
    two=one+one;
    zero=one-one;
    a=one;                                  Determine ibeta and beta by the method of M.
    do {                                        Malcolm.
        a += a;
        temp=a+one;
        temp1=temp-a;
    } while (temp1-one == zero);
    b=one;
    do {
        b += b;
        temp=a+b;
        itemp=(int)(temp-a);
    } while (itemp == 0);
    *ibeta=itemp;
    beta=CONV(*ibeta);
    *it=0;                                  Determine it and irnd.
    b=one;
    do {
        ++(*it);
        b *= beta;
        temp=b+one;
        temp1=temp-b;
    } while (temp1-one == zero);
    *irnd=0;
    betah=beta/two;
    temp=a+betah;
    if (temp-a != zero) *irnd=1;
    tempa=a+beta;
    temp=tempa+betah;
    if (*irnd == 0 && temp-tempa != zero) *irnd=2;
    *negep=(*it)+3;                         Determine negep and epsneg.
    betain=one/beta;
    a=one;
    for (i=1;i<=(*negep);i++) a *= betain;
    b=a;
    for (;;) {
        temp=one-a;
        if (temp-one != zero) break;
        a *= beta;
        --(*negep);
    }
    *negep = -(*negep);
    *epsneg=a;
    *machep = -(*it)-3;                     Determine machep and eps.
    a=b;
```

```
for (;;) {
    temp=one+a;
    if (temp-one != zero) break;
    a *= beta;
    ++(*machep);
}
*eps=a;
*ngrd=0;                          Determine ngrd.
temp=one+(*eps);
if (*irnd == 0 && temp*one-one != zero) *ngrd=1;
i=0;                              Determine iexp.
k=1;
z=betain;
t=one+(*eps);
nxres=0;
for (;;) {                        Loop until an underflow occurs, then exit.
    y=z;
    z=y*y;
    a=z*one;                      Check here for the underflow.
    temp=z*t;
    if (a+a == zero || fabs(z) >= y) break;
    temp1=temp*betain;
    if (temp1*beta == z) break;
    ++i;
    k += k;
}
if (*ibeta != 10) {
    *iexp=i+1;
    mx=k+k;
} else {                          For decimal machines only.
    *iexp=2;
    iz=(*ibeta);
    while (k >= iz) {
        iz *= *ibeta;
        ++(*iexp);
    }
    mx=iz+iz-1;
}
for (;;) {                        To determine minexp and xmin, loop until an
    *xmin=y;                          underflow occurs, then exit.
    y *= betain;
    a=y*one;                      Check here for the underflow.
    temp=y*t;
    if (a+a != zero && fabs(y) < *xmin) {
        ++k;
        temp1=temp*betain;
        if (temp1*beta == y && temp != y) {
            nxres=3;
            *xmin=y;
            break;
        }
    }
    else break;
}
*minexp = -k;                     Determine maxexp, xmax.
if (mx <= k+k-3 && *ibeta != 10) {
    mx += mx;
    ++(*iexp);
}
*maxexp=mx+(*minexp);
*irnd += nxres;                   Adjust irnd to reflect partial underflow.
if (*irnd >= 2) *maxexp -= 2;     Adjust for IEEE-style machines.
i=(*maxexp)+(*minexp);
```
Adjust for machines with implicit leading bit in binary mantissa, and machines with radix

```
    point at extreme right of mantissa.
    if (*ibeta == 2 && !i) --(*maxexp);
    if (i > 20) --(*maxexp);
    if (a != y) *maxexp -= 2;
    *xmax=one-(*epsneg);
    if ((*xmax)*one != *xmax) *xmax=one-beta*(*epsneg);
    *xmax /= (*xmin*beta*beta*beta);
    i=(*maxexp)+(*minexp)+3;
    for (j=1;j<=i;j++) {
        if (*ibeta == 2) *xmax += *xmax;
        else *xmax *= beta;
    }
}
```

Some typical values returned by `machar` are given in the table, above. IEEE-compliant machines referred to in the table include most UNIX workstations (SUN, DEC, MIPS), and Apple Macintosh IIs. IBM PCs with floating co-processors are generally IEEE-compliant, except that some compilers underflow intermediate results ungracefully, yielding $\texttt{irnd} = 2$ rather than 5. Notice, as in the case of a VAX (fourth column), that representations with a "phantom" leading 1 bit in the mantissa achieve a smaller `eps` for the same wordlength, but cannot underflow gracefully.

CITED REFERENCES AND FURTHER READING:

Goldberg, D. 1991, *ACM Computing Surveys*, vol. 23, pp. 5–48.

Cody, W.J. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 303–311. [1]

Malcolm, M.A. 1972, *Communications of the ACM*, vol. 15, pp. 949–951. [2]

IEEE Standard for Binary Floating-Point Numbers, ANSI/IEEE Std 754–1985 (New York: IEEE, 1985). [3]

20.2 Gray Codes

A Gray code is a function $G(i)$ of the integers i, that for each integer $N \geq 0$ is one-to-one for $0 \leq i \leq 2^N - 1$, and that has the following remarkable property: The binary representation of $G(i)$ and $G(i+1)$ differ in *exactly one bit*. An example of a Gray code (in fact, the most commonly used one) is the sequence 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, and 1000, for $i = 0, \ldots, 15$. The algorithm for generating this code is simply to form the bitwise exclusive-or (XOR) of i with $i/2$ (integer part). Think about how the carries work when you add one to a number in binary, and you will be able to see why this works. You will also see that $G(i)$ and $G(i+1)$ differ in the bit position of the rightmost zero bit of i (prefixing a leading zero if necessary).

The spelling is "Gray," not "gray": The codes are named after one Frank Gray, who first patented the idea for use in shaft encoders. A shaft encoder is a wheel with concentric coded stripes each of which is "read" by a fixed conducting brush. The idea is to generate a binary code describing the angle of the wheel. The obvious, but wrong, way to build a shaft encoder is to have one stripe (the innermost, say) conducting on half the wheel, but insulating on the other half; the next stripe is conducting in quadrants 1 and 3; the next stripe is conducting in octants 1, 3, 5,