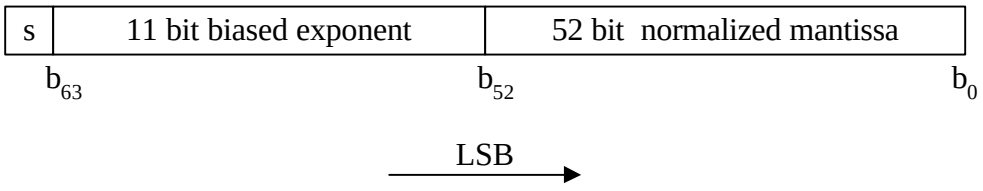
Chapter 1: Floating Point Numbers

Not all real numbers (denoted here as $\mathcal{R}$) are representable on a digital computer. In operations involving the real numbers, a computer uses a subset $\mathcal{F}$, $\mathcal{F} \subset \mathcal{R}$, known as the *floating point numbers*.

The Intel-based PC utilizes floating point numbers based on the IEEE floating point standard. This standard utilizes both *single precision* (Intel's short real format) and *double precision* (Intel's long real format) formats. For numerical calculations, MatLab uses double precision floating points exclusively; in fact, MatLab does not support the single precision format. For this reason, we will concentrate only on the double precision format.

IEEE Floating Point Format

There are both *normalized* and *denormalized* floating point numbers. We discuss normalized numbers first. Double precision normalized numbers are 64 bits (8 bytes) in length, and they have the format



b_{63} is the sign bit: s = 0 if the number is nonnegative, s = 1 if negative.

b_{62} .... b_{52} is the biased exponent E′ (b_{52} is the LSB of the biased exponent)

b_{51} .... b_0 is the mantissa (b_0 is the LSB of the mantissa).

The sign bit is either 0 or 1. If 0, the number is non-negative. If 1, the number is negative.

The 11 bit biased exponent E′ contains a bias of 1023. Hence, the actual binary exponent is E′ - 1023. Inclusion of a bias allows for negative exponents. With the exception of all 1s and

all 0s, all possible bit patterns are allowed for the exponent (all 0s and 1s are reserved for special purposes). Hence, with 11 bits, the range for E′ is $1 \le E' \le 2^{11}-2 = 2046$, and the range for the actual binary exponent is $-1022 \le E' - 1023 \le 1023$.

The mantissa is normalized. That is, the exponent is adjusted so that the mantissa has a MSB of 1. Since the mantissa's MSB is always 1, there is no need to store it in memory. With the implied 1 for its MSB, the mantissa has 53 bits of precision.

The base-ten value of a normalized double precision number can be computed easily. It is

$$(-1)^S * [1 + b_{51}\left(\frac{1}{2}\right) + b_{50}\left(\frac{1}{4}\right) + \cdots + b_0\left(\frac{1}{2^{52}}\right)] * 2^{E'-1023}$$

MatLab's HEX format can be used to decode double precision floating point numbers. For example, consider entering x = -1.5 at the MatLab prompt. You will get what is shown below.

> **format hex**

> **x = -1.5**

```
x = bff8000000000000
```

(keyboard entries are preceded by >, and they are shown in **boldface**. The computer's response is shown in the `Courier New` font). The HEX number bff8000000000000 is converted easily to the binary number

1011 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Clearly, the sign bit is s = 1, the biased exponent is $E' = 2^{10} - 1 = 1023$, and the mantissa is 1 + 1/2. Based on these values, we conclude that the number has the base 10 value of $-1.5 \times 2^0 = -1.5$

Numeric Range of Double Precision Normalized Floating Points

Floating point numbers have a finite range. Normalized, double precision floating point numbers must lie in the range

$$2^{-1022} \le |\text{Normalized Double Precision Floating Point}| \le (2 - 2^{-52}) * 2^{1023}, \qquad (1\text{-}1)$$

or they have a magnitude between $2.225073858507202 \times 10^{-308}$ and $1.797693134862316 \times 10^{+308}$. These numbers are returned by MatLab's **REALMAX** and **REALMIN** statements.

On the real line, the floating point numbers are *not uniformly dense*. An equal number of floating point numbers fall between successive powers of 2. For example, the number of floating point values between 2 and 4 is equal to the number of floating point numbers between 65,536 and 131,072 (both powers of 2). Between adjacent floating point numbers, the "gaps" become larger as the biased exponent increases (the density of floating points decreases with increasing exponent value).

To illustrate a simple normalized floating point system, consider a system with

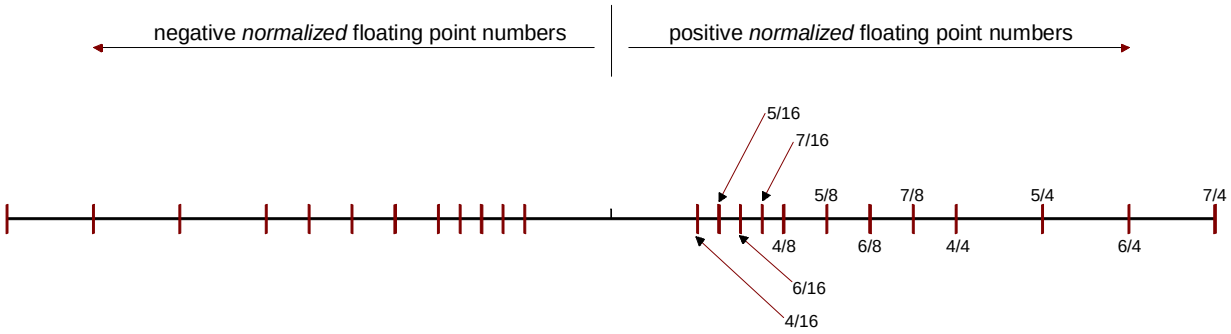
i) a normalized (MSB is always 1) two bit mantissa,
ii) base two exponent range of $-2 \le E \le 0$ and
iii) a sign bit: s = 0 (s = 1) for non-negative (negative) numbers.

In this system, normalized numbers take the form $(-1)^s(1.b_1b_0)2^E$. And, the possible numbers fall on the real line as shown by Figure 1-1.

On Figure 1-1, notice the "large" gap centered at the origin. This is the result of normalization of the mantissa. All normalized floating point systems have a "gap" centered at the origin. The denormalized floating point numbers fill this gap, they are discussed after we cover overflow and underflow.

Numeric Overflow and Underflow in MatLab

In MatLab, suppose a normalized, double precision IEEE floating point variable exceeds $(2 - 2^{-52}) * 2^{1023}$ in magnitude. What does MatLab do when a numeric *overflow exception* occurs?



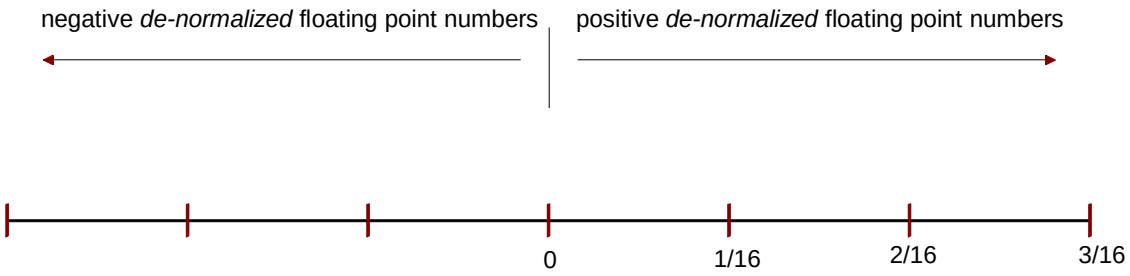
Notes:
1) Two bits of mantissa
2) -2 ≤ Base Two Exponent ≤ 0
3) 1 Sign Bit
4) Gap from -4/16 to 4/16 in the normalized floating point numbers

Figure 1-1: A hypothetical normalized floating point system

Well, MatLab sets the variable to either *inf* or - *inf* (*inf* is a special number in MatLab). In MatLab, you can use *inf*'s calculations to obtain "very natural results". For example, for any real finite number x, MatLab calculates *inf* = x∗*inf*, 0 = x/*inf*, etc. However, some operations involving *inf* result in "not a number", which is termed a "NAN" in MatLab. For example, MatLab returns NAN if you type *inf* - *inf* or *inf*/*inf*. Note that MatLab uses the *affiine closure* model (as opposed to the *projective closure* model) for infinity since it allows both + *inf* and - *inf*.

On an Intel-based PC, when the magnitude of a normalized double precision floating point falls below 2^{-1022} and *underflow exception* occurs. Well written software will institute some "reasonable" strategy to deal with numeric underflows.

When an underflow occurs, MatLab allows the result to "denormalize" (*i.e.*, become a "*denormal*"), and the program "trades" accuracy (significant mantissa bits) for numeric range. When underflow occurs, MatLab keeps -1022 as the base 2 exponent, but allows the mantissa to become *unnormalized* with leading zeros. By allowing the mantissa to have leading zeros, the effective range of negative exponents can be extended by the number of mantissa bits. However, each leading mantissa zero is a loss of one bit of precision, so extended exponent range is achieved at the expense of lost mantissa accuracy.



Notes:
1) Two bits of mantissa
2) -2 ≤ Base Two Exponent ≤ 0
3) 1 Sign Bit

Figure 1-2: Denormals for a hypothetical floating point system.

The denormals fill the gap, centered at the origin, left by the normalized floating point numbers. Again, consider the hypothetical floating point system described by Figure 1-1. In this system, the mantissa is 2 bits long, and the binary exponent ranges from -2 ≤ E ≤ 0. For this system, the "gap-filling" denormals are illustrated by Figure 1-2.

When a floating point number "denormalizes", the precision of the mantissa is traded for exponent range. This tradeoff of accuracy for range is illustrated by the following MatLab script.

```
>pi
ans = 3.14159265358979
>pi*1E-308*1E-0
ans = 3.141592653589792e-308
>pi*1E-308*1E-1
ans = 3.141592653589791e-309
>pi*1E-308*1E-2
ans = 3.141592653589806e-310
>pi*1E-308*1E-3
ans = 3.141592653589954e-311
>pi*1E-308*1E-4
ans = 3.141592653588472e-312
>pi*1E-308*1E-5
ans = 3.141592653588472e-313
>pi*1E-308*1E-6
ans = 3.141592653786098e-314
>pi*1E-308*1E-7
```

```
ans = 3.141592653292032e-315
>pi*1E-308*1E-8
ans = 3.141592643410720e-316
>pi*1E-308*1E-9
ans = 3.141592495191026e-317
>pi*1E-308*1E-10
ans = 3.141590518928442e-318
>pi*1E-308*1E-11
ans = 3.141615222210734e-319
>pi*1E-308*1E-12
ans = 3.141763441904488e-320
>pi*1E-308*1E-13
ans = 3.142257507550328e-321
>pi*1E-308*1E-14
ans = 3.162020133383978e-322
>pi*1E-308*1E-15
ans = 2.964393875047479e-323
>pi*1E-308*1E-16
ans = 4.940656458412465e-324
>pi*1E-308*1E-17
ans = 0
```

Here, $pi*10^{-308}*10^{-x}$ is printed for $0 \le x \le 17$. For x = 0, full accuracy is maintained. But as x increases, accuracy is lost until the result is set to zero at x = 17. As can be seen, the result "degrades" in a "gentle", "gradual" manner.

Allowing the floating point to denormalize, as MatLab does, has both good and bad features. As a good feature, it allows MatLab to use larger magnitudes. On the bad side, with denormalized numbers, MatLab produces less accurate results. In a lengthy calculation, it is possible for an intermediate result to denormalize and loose most of its precision, but the final result may appear "OK". In all likelihood, the user will not know (MatLab will not tell him) that denormalization occurred, and he will mistakenly believe that his answer contains full precision.

Machine Epsilon

Machine Epsilon, known as eps in MatLab, is the distance from 1.0 to the next valid floating point number. This value is used in determining the numerical rank of a matrix and in determining the pseudo inverse of a matrix. On an Intel based machine, $eps = 2^{-52} \approx 2.22x10^{-16}$.

Rounding and Round Off Error

More often than not, the result of an arithmetic operation cannot be represented exactly as a valid floating point number. Suppose the "correct answer" falls between adjacent normalized floating point numbers. The computer must round to one of the numbers, and this produces a *round off error*. Such errors can produce serious problems in algorithms that involve a large number of repetitive calculations.

On an Intel based machine, MatLab programs the computer's floating point unit (FPU) to round to the nearest valid floating point value. When the exact result lies exactly midway between two valid floating point numbers, the FPU rounds towards the nearest *even* floating point number. A floating point number is even (off) if the least significant bit, of LSB, of its mantissa is zero (one).

To illustrate the rounding procedure described above, assume that f_2, f_3, f_4, f_5 and f_6 represent consecutive positive floating point numbers. Also, assume that f_2, f_4, and f_6 are even so that f_3 and f_5 are odd (floating point numbers alternate between even and odd). The "halfway" values will be rounded as shown by Table 1-1.

Halfway Between	**Round Towards**
f_2 and f_3	Down to f_2
f_3 and f_4	Up to f_4
f_4 and f_5	Down to f_4
f_5 and f_6	Up to f_6

Table 1-1: Round to nearest even scheme.

The hope is that half the time the system will round up (down) and some rounding error will cancel out. This method is sometimes called *unbiased round to nearest even*.

Rounding to the nearest even is illustrated by the following MatLab script. Note that MatLab's output is given in HEX. By inspection, the mantissa can be decoded from the HEX

output, and the number's even/odd nature determined.

```
> 1
ans = 3ff0000000000000
```

% from 1+.1eps to 1+.5eps, the result rounds down to the nearest even (which is 1.0)

```
> 1+.1*eps
ans = 3ff0000000000000

> 1+.2*eps
ans = 3ff0000000000000

> 1+.3*eps
ans = 3ff0000000000000

> 1+.4*eps
ans = 3ff0000000000000

> 1+.5*eps
ans = 3ff0000000000000
```

% from 1+.6eps to 1+.9eps, the result rounds up to the nearest floating point (which is 1.0 + eps)

```
> 1+.6*eps
ans = 3ff0000000000001

> 1+.7*eps
ans = 3ff0000000000001

> 1+.8*eps
ans = 3ff0000000000001

> 1+.9*eps
ans = 3ff0000000000001

> 1+eps
ans = 3ff0000000000001
```

Notice that the midpoint 1 + .5eps rounds to the nearest *even* (which is 1.0 in this case).

Round off errors accumulate with increasing amounts of calculation. After N calculations, your ***might*** be so lucky as to have a total round off error on the order of $\sqrt{N}$ eps, if it is just as likely to round up as it is to round down, and the round off errors occur independently of one another (in this case, the total round off error can be modeled as a random walk with a variance

proportional to N∗eps^2. However, such an outcome is unlikely; usually, things are much worse. There are at least two pitfalls that can cause significant round off.

1) At each calculation, the round off error is **not** independent of the round off error at previous calculations. Instead, round off errors may accumulate preferentially in one direction. In this case, the total round off error is on the order of N∗eps.

2) If you are really unlucky, a single calculation (or just a few) can introduce a found off error that is many powers of 2 greater than eps. This will be the case if the single calculation forces an underflow so that the result denormalizes. The denormalized result will have a "shortened", less precise, mantissa forcing a significant (possibly many orders of magnitude larger than machine eps) round off error.

Sometimes, an algorithm can be rendered unusable by round off error. The calculation of e^x by power series expansion is an example that is given by many authors. Recall the series

$$e^x = 1 + x + x^2/2! + x^3/3! + ... \tag{1-2}$$

Suppose we wanted to calculate $e^{-5.5}$ on a base 10 machine that uses 5 digits in ***all*** of its calculations and rounds to nearest. We compute

$$\begin{aligned} e^{-5.5} = \\ &+1.0000 \\ &-5.5000 \\ &+15.125 \\ &-27.730 \\ &+38.129 \\ &-41.942 \\ &+38.446 \\ &\ \ \vdots \\ &\overline{\qquad\qquad} \\ &+.0026363 \end{aligned} \tag{1-3}$$

The sum .0026363 is terminated after 25 terms because adding more terms no longer changes the computed result (due to the 5 digit limitation). This "answer" has no valid significant figures. The true answer is closer to $e^{-5.5} \approx .00408677$, so the computed result has no correct significant

digits. What happened here? The small final "answer" results from the difference of many larger terms. Terms like 38.129 contain round off error which is nearly as large as the final answer. This phenomenon, where a small final answer is the difference of large intermediate results, is known as *catastrophic cancellation*, and it is all too common in algorithm development.

Of course, better results from the series expansion could be obtained by using more significant figures. However, increasing the number of significant figures is not always possible. Often, we must search for a better algorithm, one that does not suffer from *catastrophic cancellation*. In the case under study, we can calculate

$$e^{-5.5} = 1/e^{5.5} = \frac{1}{1+5.5+15.125+\cdots} = .0040865, \tag{1-4}$$

where all calculations are performed using 5 decimal digits. With this algorithm, *catastrophic cancellation* is avoided, and the total error is reduced to .007%.

Floating point round off is not the only problem that one can run into during a calculation. In fact, sometimes, each step in an iterative algorithm can be accomplished without introducing *and* round off error at all. But yet, the algorithm may fail. This may happen when the basic algorithm is *numerically unstable*.

Algorithm Stability

An algorithm may be extremely sensitive to errors in the specification of the problem to be solved (i.e., the algorithm may be extremely sensitive to modeling errors). Or, the algorithm may be extremely sensitive to errors in the initial data (initial conditions) used to start the algorithm. In these cases, we say that the algorithm has *stability problems*. Instabilities are a problem when small initial errors grow rapidly and "swamp out", or dominate, the result. In an unstable algorithm, this can happen even if each iteration of the algorithm is carried out with no round off error.

Example

Suppose we wish to compute the integrals

$$E_n = \int_0^1 x^n e^{x-1} dx, \quad n = 1, 2, 3, \cdots \tag{1-5}$$

Using integration by parts, we obtain

$$\begin{aligned} E_n &= \int_0^1 x^n e^{x-1} dx = x^n e^{x-1}\Big|_0^1 - \int_0^1 n x^{n-1} e^{x-1} dx \\ &= \begin{cases} e^{-1} & , n = 1 \\ 1 - nE_{n-1} & , n = 2,3,\cdots \end{cases} \end{aligned} \tag{1-6}$$

This iteration can be programmed easily. Suppose our computer has 6 decimal digits of accuracy (i.e., we are using the IEEE single precision format) so we compute $E_1 = e^{-1} \approx .367879$ as our initial value. Let's use this approximation to E_1 and compute E_2 through E_9 by iterating (1-6).

$E_1 = 0.367879$

$E_2 = 1 - 2*E_1 = 0.264242000000000$

$E_3 = 1 - 3*E_2 = 0.207274000000000$

$E_4 = 1 - 4*E_3 = 0.170904000000000$

$E_5 = 1 - 5*E_4 = 0.170904000000000$

$E_6 = 1 - 6*E_5 = 0.127120000000000$

$E_7 = 1 - 7*E_6 = 0.110160000000000$

$E_8 = 1 - 8*E_7 = 0.118720000000000$

$E_9 = 1 - 9*E_8 = -0.068480000000000$

Obviously, something is wrong! The integrand $x^9 e^{x-1}$ is positive over [0, 1], but our calculated E_9 is negative! Notice that starting with the calculation for E_2, each iteration of the algorithm is done

without round off (We used double precision in MatLab to do the calculations. But note that no round off would occur in E_2 through E_9 on a machine that uses only 6 decimal digits). And, our integration by parts (Equation (1-6)) is exact. Hence, our problem *must* lie with an error in the initial value E_1. This algorithm is very sensitive to initial errors!

To see how an error of $\approx 4.412\text{x}10^{-7}$ in E_1 becomes so large, notice that it is multiplied by -2 to form the error in E_2. Then the error in E_2 is multiplied by -3 when E_3 is calculated, and so on. The error in E_9 is 9! (9 factorial) times the error in E_1. Initial errors are propagating through the iteration, and they are dominating the correct answer. As implemented above, this algorithm is unstable. Fortunately, for the problem under consideration, there is a simple "fix".

To obtain a stable algorithm for calculating the E_k, we iterate (I-6) in a *backwards* direction. Just write

$$E_{n-1} = \frac{1 - E_n}{n}, \quad n = \cdots, 3, 2. \tag{1-7}$$

Note that any error in E_n is decreased by 1/n to produce the error in E_{n-1}. If we start with a value for E_n, $n >> 1$, and iterate backwards, any initial error will be decreased at each step. For this problem, backward iteration is stable!

To obtain a starting value, we note that

$$E_n = \int_0^1 x^n e^{x-1} dx \le \int_0^1 x^n dx = \frac{1}{n+1} \quad . \tag{1-8}$$

Thus, $E_n \to 0$ as $n \to \infty$. For example, if we approximate E_{20} by 0 and iterate backwards, our initial error is less than 1/21. The error in E_{19} is less than (1/21)*(1/20) $\approx$ .0024. Iterating downwards, the initial error in E_{20} decays to less than $4\text{x}10^{-8}$ by the time E_{15} is computed, and this amount of error is less than the precision of our imaginary 6 digit computer. We compute the values

$E_{20} \approx 0$ $E_{14} \approx .0627322$

$E_{19} \approx .0500000$ $E_{13} \approx .0669477$

$E_{18} \approx .0500000$ $E_{12} \approx .0717733$

$E_{17} \approx .0527780$ $E_{11} \approx .0773523$

$E_{16} \approx .0557190$ $E_{10} \approx .0838771$

$E_{15} \approx .0590176$ $E_{9} \approx .0916123$

By the time we reach E_{15}, the initial error in E_{20} has completely damped out by the stability of the algorithm. The computed E_{15} ... E_9 are accurate to at least 6 decimal digits.