CornellEngineering
Electrical and Computer Engineering

ECE 4960: Computational and Software Engineering

Spring 2017

Note 2: Sources of Errors in Computing

Reading Assignments:

1. Chap. 2, D. Bindel and J. Goodman, *Principles of Scientific Computing*, 2009.
2. Chap. 1, B. Einarsson, Ed., *Accuracy and Reliability in Scientific Computing*, SIAM 2005.

1. Class logistics

- Setup of your programming platform on your notebook computer
- Reading review 1 (will be multiple choice questions on Blackboard)

2. Setup of programming platform

You are welcome to choose your own platform for programming assignment in this class, as long as debugging, multiple-author projects and version control can be performed (internal to your platform, or external by git or cvs). If you do not have a ready setup, there are a few resources that you can build your own IDE (integrated development environment).

For example, C/C++ run most naturally on Unix-derived systems, although plenty of IDEs on Windows/Mac OS are available as well. So, theoretically you do not really need your own real/virtual/network-connected Linux machine. If you do wish to use genuine Linux and Gnu C/C++ for a standard learnng, two choices are available:

1. You can build a virtual Linux machine. The platform suggested is Oracle Virtual Box on PC or Mac or Android-based platforms. You will need the software downloaded from https://www.virtualbox.org/ and you should choose the Linux OS by Debian or Ubuntu.
2. You can use amdpool.ece.cornell.edu. Xterm from your Windows or MacOS should give you sufficient access of all features we need in this class.

To use an IDE directly on your laptop, code::block is suggested, and a tutorial is available in "Jumping into C++", pp. 13 – 40, along with the download sites. IDE works across platforms, PC, Mac, Linux and RISC mainframe machines, although it is most native to Unix-derived systems and belongs to the general family of Gnu[1]. You will need gcc and gdb prior to install code::block. You can use your own choices of C++ compiler and debugger within code::block. If you have no C++ access yet on your platform, download the Gnu gcc and gdb first, as it is required in the IDE setup.

Alternatively, many large-scale projects still employ generic "makefile" native to any Unix-derive systems. Not only text-based development platform is more portable and scalable[2] than GUI-based IDE, it is also much less error prone. Imagine how annoying it is to programmers when the troubling bug is

[1] Gnu is a free operating system. For more history and philosophy, visit: https://www.gnu.org/.
[2] I am not against visual programming at all, but when the program size is large, showing all controls and objects on a finite-size screen can eventually be inconvenient.

actually in the IDE with no direct access!! If you choose to use the basic "makefile", you will also need a reasonable text editor such as vim and emacs, where automatically indentation and highlights are used in text rendering but not in the source files.

3. Precision estimate and remedy

3.1 Representing integers and real numbers by binary

The digital computer has binary representation in the core hardware, and therefore all computations are subject to finite precision. The real world is however more conventionally represented by integers and floating-point numbers[3]. Table 1 shows the IEEE 754 standard for integer and floating-point representations by binary bits.

Table 1. Integer and floating-point implementation in IEEE Standard.

Data type (C++ declaration)	No. of bits	Attributes	Exception handling	Smallest possible[4]	Largest possible
Integer (`short`)	16	Seldom used now	None	-2^{15}	$2^{15}-1$
Integer (`long`)	32	1 sign bit;	None	-2^{31}	$2^{31}-1$
Single precision floating point (`real`)	32	Seldom used now; 1 sign bit (s); 23 mantissa bits (f); 8 exponent bits (e). Normalized: $x=(-1)^s\cdot(1.f)\cdot 2^{e-127}$ 127 is "the bias".	***NaN:*** $e=255;\ f\neq 0$ ***INF:*** $e=255;\ f=0;\ s=0$ ***NINF:*** $e=255;\ f=0;\ s=1$	Only by e: 2^{-126} Soft landing: $2^{-23}\cdot 2^{-126}=2^{-149}$ $\cong 1.4\times 10^{-45}$	$(2-2^{-23})\cdot 2^{127}\cong$ 3.4×10^{38}
Double precision floating point (`double`)	64	1 sign bit (s); 52 mantissa bits (f); 11 exponent bits (e). Normalized: $x=(-1)^s\cdot(1.f)\cdot 2^{e-1023}$ 1023 is "the bias".	***NaN:*** $e=2047;\ f\neq 0$ ***INF:*** $e=2047;\ f=0;\ s=0$ ***NINF:*** $e=2047;\ f=0;\ s=1$	Only by e: 2^{-1022} Soft landing: $2^{-52}\cdot 2^{-1022}=2^{-1074}$ $\cong 4.9\times 10^{-324}$	$(2-2^{-52})\cdot 2^{1023}\cong$ 1.8×10^{308}

Additionally, underflow is treated explicitly for floating points, signaled by $e = 0$ and $f \neq 0$. We will talk more about this exception handling in the next section.

3.2 Computation of floating points with finite precision

3.2.1 The quadratic solution: when everything is known

We will use the quadratic solution to illustrate how precision and truncation errors affect your computing, and then we will generalize the observation to broader questions.

[33] Many authors will use "real numbers" here. However, from our understanding of quantum mechanics today, the complex number is actually an essential part in the real world. Imaginary numbers are NOT a mathematical convenience or invention, but part of the physical reality.
[4] Some bit combinations are used for exception handling. Also, very small number has underflow controls.

For a quadratic equation:

$$y = f(x) = ax^2 + bx + c = 0 \tag{1}$$

We know the closed-form solution of the two roots can be expressed in the form of

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

We know the degenerate situation of $a = 0$ as well, which will bring one of the roots to be undefined. For a nearly degenerate situation of:

$$a = 10^{-5};\ b = 10^{3};\ c = 10^{3}.$$

We obtain (with single precision of 10^{-9} in floating-point expression):

$$x_1 = -1.0 \text{ and } x_2 = -3.0518\times 10^{7}.$$

Another equivalent way to write down Eq. (2) is:

$$x_{1,2} = \frac{-2c}{-b \pm \sqrt{b^2 - 4ac}} \tag{3}$$

You can prove analytically that the two are "symbolically" identical, or we can say that the two closed-form solutions are equivalent IF we have infinite precision in the calculation. However, with single precision, we will find:

$$x_1 = -1.0 \text{ and } x_2 = -3.2768\times 10^{7}.$$

Group discussion: What does your calculator say by using Eqs. (2) and (3)? How can you make double precision calculation to show such problems?

3.2.2 The condition number of the formalism

Next we consider the "condition number" κ, which describes how relative input error is related to the relative output function error:

$$\left|\frac{\Delta f}{f}\right| \cong \kappa \left|\frac{\Delta x}{x}\right|, \quad \text{or} \qquad \kappa = \left|\frac{\Delta f}{\Delta x}\cdot\frac{x}{f}\right| \tag{4}$$

For sure, when a function f is "**ill-conditioned**" (very large κ), then the relative error will be magnified. If we use f two times as $f(f(x))$, we will then have uncontrollable error eventually, whatever the finite precision we use. However, we will see that "ill-conditioned" is only a sufficient reason, but not necessary, as there are other erroneous sources as will be shown in the simple quadratic example.

As $f = 0$ for finding the root, we will need to use the average of $f(x)$ and $f(x+\Delta x)$ in Eq. (4) to avoid the apparent degeneracy. The condition number κ at the accurate root –1 is:

$$\left|\frac{f(-1.01)-f(-1)}{0.01\times\left(\dfrac{f(-1.01)+f(-1)}{2}\right)}\right|=2.01$$

The condition number at the other "polluted" root is:

$$\left|\frac{f\left(-3.0518\times10^{7}\right)-f\left(-3.0823\times10^{7}\right)}{0.01\times\left(\dfrac{f\left(-3.0518\times10^{7}\right)+f\left(-3.0518\times10^{7}\right)}{2}\right)}\right|=0.55$$

The condition number looks normal for both cases!!! This is because the error at hand is not the formal solution algorithm that blows up the relative error, but the finite precision has caused the inaccuracy. The condition numbers for both Eqs. (2) and (3) are within reasonable bounds.

3.2.3 Thoughs on formulation conditioning and floating-point precision

The root of the precision error stems from finite precision of the floating-point representation. When a small number is added to a much larger number, the precision of the small number is truncated out. If we substract the large number (or directly the difference of two large numbers has less precision), the answer has much less precision than what is initially set. This is common in many computing applications, including estimation of slope, margin, common-mode rejection, etc. Notice that multiplication, even though taking more time, is more benign to preserve the precision of the operands. However, if we go through the exponential-logarithmic conversion between the addition and multiplication, the problem remains. Truncation errors are built in features for finite precision. Usually you can only do better by extending the precision, such as representing a floating-point number by 128 bits (called quad precision!)

Now, we need to ask: which calculation is more "precise" or "trustworthy", or whether they are any useful? The first thing came tfo mind is to substitue back those two inconsistent roots:

$$f(-3.0518\times10^{7}) = -2.12\times10^{10}; \quad f(-3.2768\times10^{7}) = -2.20\times10^{10}$$

We can immediately recognize both are pretty far away from zero. We are actually fortunate here, as we know the exact form of solution (a "**formal verification**") so that an invese check is straightforward. However, in general situation of a much more complex problem to solve for $\boldsymbol{y} = f(\boldsymbol{x})$ where $\boldsymbol{x}$ and $\boldsymbol{y}$ are large vectors and f is a complex nonlinear function, we are not sure if the back substitution inconsistency is the result of precision error or some other possible errors in the algorithm or code.

For these rare cases when the "formal solution" is known, we can indeed do much better by using the perturbation. In general, if we know the perturbation response of a system (or test the perturbation numerically), we can use alternative forms to do much better. For this quadratic solution, we know that the most precision error in Eq. (2) is caused by the subtraction in the numerator. As $4ac << b^2$, we can use the local approximation to improve our solution:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b \pm b\sqrt{1 - \frac{4ac}{b^2}}}{2a} \cong \frac{-b \pm b\left(1 - \frac{2ac}{b^2}\right)}{2a} = -\frac{c}{b} \quad or \quad -\frac{b}{a} + \frac{c}{b} \tag{5}$$

We now obtain the two roots of –1 and $1 - 10^8$! If we perform back substitution, we will have $f(1 - 10^8) = 10^{-5}$, which is indeed a much closer solution than previous calculation of Eqs. (2) or (3) as $f(-3.0518\times10^7) = -2.12\times10^{10}$ and $f(-3.2768\times10^7) = -2.20\times10^{10}$ in terms of the residual of $f(x)$!

This principle can be generalize to expand the precision for degenerate cases! It would be a worthy practice to calculate beyond the double precision, which in practice can be used to get the infinite precision for known irrational number of π and e.

If we do not know the formal solution, the next best thing to know is the **asymptotic** behavior. Do we know the behavior or the function at 0, infinity or at special points? We will cover this more when we have a right context.

For sure, some other coding errors can have the same “coincidence”, and it would be in general hard to tell the exact source of errors. Notice that this is NOT the effect of the local slope, as the slope at the zero-crossing point of the quadratic equation is the same symbolically.

The ill condition and the precision error have different sources, but are very difficult to separate in the complex problems. Another possible way is to resort to even higher precision in numerical representation (such as quad precision to represent a floating point number by 128 bits). However, it would still have some ambiguity, unless the exact operation of large κ or precision error can be identified to the specific line in computational code.

Hacker practice 2.1: Solve the quadratic equation for $a = 10^{-20}; b = 10^3; c = 10^3$ in your platform. Where is the potential problem in precision? Introduce ways to detect and compensate the nearly degenerate condition.

4. Exception definition and handling

In addition to the truncation error from precision to magnify relative errors, there is another consequence for the limited number of digits to represent a floating point number in terms of the smallest and largest numbers representable. IEEE had recognized this issue in the early days of computing, and had developed exception handling rules. There are three main categories, not a number (NaN, where the computer cannot make a guess from the present representation), overflow (the number is too large) and underflow (the number is too close to zero that the precision from rounding can be lost).

4.1 Not a number in floating point representation

In the most straightforward example, the computation of 0.0/0.0 or $\sqrt{-1.0}$ will treated as an unrecoverable error which signals an exception. There are examples where it makes sense for a computation to continue in such a situation. For example, we call an external function *f* with a returned argument, not sure how the computation is handled for events like 0.0/0.0 or $\sqrt{-1.0}$ (can be the quadratic root function before). Instead of halting the entire program, a NaN can be returned when the function cannot figure out the operation, maybe the input is given in the wrong domain by the user.

A list of some of the situations that can cause a NaN are given in Table 2. It is easy to see what the result of combining a NaN with an ordinary floating-point number should be. For example, the function *f* returns the quadratic roots of Eq. (2). If $b^2 - 4ac < 0$, then *f* should return a NaN. Since $b^2 - 4ac <$ 0, $\sqrt{b^2 - 4ac}$ is a NaN, and $-b - \sqrt{b^2 - 4ac}$ should be a NaN, if the sum of a NaN and any other number is a NaN. Similarly if one operand of a division operation is a NaN, the quotient should be a NaN. In general, whenever a NaN participates in a floating-point operation, the result is another NaN.

Table 2. NaN generation and propagation in floating point operations

Operation	NaN produced by
+, –	INF – INF, INF + NINF, etc.
×	0 × INF
/	0/0, INF/INF, INF/NINF, etc.
% or REM	*x*%0, INF%*y*, etc.
$\sqrt{x}$	$x < 0$

When a floating-point operation raises a floating-point exception, the status of the floating-point environment changes (ex. the environmental variable SIGFPE changes from 0 where SIGFPE stands for "signaling" floating-point exception), which can be tested in C++ with std::fetestexcept, but the execution of a C++ program on most implementations continues uninterrupted. Not all platforms implement the SIGFPE signaling the same way, and it is certainly an issue when the program is ported to different platforms.

In IEEE 754, NaNs are often represented as floating-point numbers with the exponent $e_{\max} + 1$ and nonzero mantissa, while INF and NINF have the exponent $e_{max} + 1$ as well but with zero mantissa and the respective sign bit. In double precision, $e_{\max} + 1 = 2047$ (11 bits of "1" in binary form: $(11111111111)_2$). Implementations are free to put system-dependent information into the mantissa. Thus there is not a unique NaN, but rather possibly a whole family of NaNs. When a NaN and an ordinary floating-point number are combined, the result should be the same as the NaN operand. Thus if the result of a long computation is a NaN, the system-dependent information in the mantissa will be the information that was generated when the first NaN in the computation was generated. Actually, there is a caveat to the last statement. If both operands are NaNs, then the result will be one of those NaNs, but it might not be the NaN that was generated first.

4.2 Overflowing

4.2.1 Overflowing in integers

Integer overflow does not have regulated signaling, and it is subject to the user's own check. For most C/C++ calculation, integers are represented by 32 bits, with one bit as the sign. Therefore the range is between -2^{31} to 2^{31}, which is only around tens of billions. Yes, it is even insufficient for the address of today's memory storage. One of the most common large integer operations is the factorial, where 12! and below are correct but above that the result can be wrong, or even become negative. If you are anticipating large numbers, either tolerate with the limited precision, or you would have to concatenate multiple integers to one, or you will just perform binaries directly. The latest case is the standard practice in cryptography, as factorization of integers represented by more than 4,000 bits is common. This for sure will slow down the integer operations considerably (more than 10 times), although it is relatively straightforward in coding a special class in C++ with operator overloading.

Group discussion: What do you think is the best strategy to implement 1/0 in integers? (1) Give the largest integer; (2) Reserve a symbol in integers and signal the exception; (3) Change to floating points INF and signal the exception; (4) Do not regulate it.

Similarly, NaN is only regulated as a floating-point representation. In integers, there is further ambiguity in NaN implementation. Most platforms do not handle integer exception! Implicit or explicit type conversion is often worse in making the code more robust!

Hacker practice 2.2: Use your platform to see the result:

```
double x = 0.0; y = 0.0; doubleResult1; doubleResult2;
doubleResult1 = 1/m;
doubleResult2 = m/n;
print(doubleResult1, doubleResult2);

long m = 0; n = 0; intResult1; intResult2;
intResult1 = 1/m;
intResult2 = m/n;
print(intResult1, intResult2);

long i = 1; intFactorial = 1;
for (i= 2; i < 30; i++) {
      intFactorial *= i;
      print(i, temp, intFactorial)
      }
```

4.2.2 Overflowing in floating points

The INF and NINF exceptions provide a way to continue code execution when an overflow occurs. This is much safer than simply returning the largest representable number. As an example, consider computing $\sqrt{x^2 - y^2}$ where (double) x =5×10^{160} and y = 4×10^{160}. The best answer will be 3×10^{160}, but during the computation x^2 and y^2 will overflow. If x^2 and y^2 are replaced with the largest number (DBL_MAX), we obtain a false impression of 0.0!!!! It is safer to report and propagate INF for x^2 and NaN for $\sqrt{x^2 - y^2}$.

0.0 divided by 0.0 returns NaN. A nonzero number divided by 0.0, however, returns INF: 1/0 = INF and –1/0 = NINF. The reason for the distinction is this: if $f(x) \to 0$ and $g(x) \to 0$ as x approaches some limit, then $f(x)/g(x)$ could have any value. For example, when $f(x) = \sin x$ and $g(x) = x$, then $f(x)/g(x) \to 1$ as $x \to 0$. But when $f(x) = 1 - \cos x$, $f(x)/g(x) \to 0$. When thinking of 0/0 as the limiting situation of a quotient of two very small numbers, 0/0 could represent anything[5]. Thus in the IEEE standard, 0/0 results in a NaN. In some implementation, you can distinguish between getting INF because of overflow and getting INF because of division by zero by checking the environmental status flags SIGFPE. The overflow flag will often be set in the first case, the division by zero flag in the second. However, this is also platform dependent.

The rule for determining the result of an operation that has infinity as an operand is simple: replace infinity with a finite number x and take the limit as $x \to$ INF. Thus 3/INF = 0.0. Similarly, 4 – INF = NINF, and sqrt(INF) = INF. When the limit doesn't exist, the result is a NaN, so INF/INF will be a NaN. This agrees with the reasoning used to conclude that 0/0 should be a NaN. When a subexpression

[5] Surely, these 0/0 and ∞/∞ cases can be treated symbolically by the L'Hospital's Rule in indeterminate forms. However, this needs to be done by the programmer, not implicitly by the computing platform.

evaluates to a NaN, the value of the entire expression is also a NaN. In the case of INF and NINF, however, the value of the expression might be an ordinary floating-point number because of rules like 1/INF = 0.0.

Here is a practical example that shows some potential problems in infinity arithmetic. Consider computing the function $x/(x^2 + 1)$. This is not a good formula for infinity arithmetic, because not only will it overflow when x is larger than $\sqrt{1.8\times10^{308}} \cong 1.3\times10^{154}$, but INF will give the wrong answer because it will yield 0, rather than a number near $1/x$. However, $x/(x^2 + 1)$ can be rewritten as $1/(x + x^{-1})$. This expression will not overflow prematurely and because of infinity arithmetic will have the correct value when $x = 0$ or INF: $1/(x + x^{-1}) = 1/(0 + \text{INF}) = 1/\text{INF} = 0$. Without infinity arithmetic, the expression $1/(x + x^{-1})$ requires a test for $x = 0$, which not only adds extra instructions, but may also disrupt a pipeline in parallel computing. This example illustrates a general fact: namely that infinity arithmetic often avoids the need for special case checking; however, formulas need to be carefully inspected to make sure they do not have spurious behavior at infinity (as $x/(x^2 + 1)$ did).

4.3 Underflowing and soft landing

4.3.1 Signed zero

Zero is represented by the exponent $e_{\min} - 1$ and a zero mantissa. In double precision, $e_{\min} - 1$ is $(e)_2 = (11111111110)_2$ or $-(1022)_{10}$. Notice that $(e)_2$ for $(11111111111)_2$ is reserved for exception of NaN and INF. Since the sign bit in the 64-bit floating point representation can take on two different values, there are two zeros, +0 and –0. If a distinction were made when comparing +0 and –0, simple tests like if (x = 0) would have very unpredictable behavior, depending on the sign of x. Thus the IEEE standard defines comparison so that +0 == –0, rather than –0 < +0. Although it would be possible always to ignore the sign of zero, the IEEE standard does not do so. When a multiplication or division involves a signed zero, the usual sign rules apply in computing the sign of the answer. Thus $3\cdot(+0) = +0$, and $(+0)/(-3) = -0$. If zero did not have a sign, then the relation $1/(1/x) = x$ would fail to hold when x = INF or NINF. The reason is that 1/NINF and 1/INF both result in 0, and 1/0 results in INF, the sign information having been lost. One way to restore the identity $1/(1/x) = x$ is to only have one kind of infinity, however that would result in the disastrous consequence of losing the sign of an overflowed quantity.

Another example of the use of signed zero concerns underflow and functions that have a discontinuity at 0, such as log(). In IEEE arithmetic, it is natural to define log(0) =NINF and log x to be a NaN when $x < 0$. Suppose that x represents a small negative number that has underflown to zero. Thanks to signed zero, x will be negative, so log can return a NaN. However, if there were no signed zero, the log function could not distinguish an underflowed negative number from 0, and would therefore have to return NINF. Another example of a function with a discontinuity at zero is the signum() function, which returns the sign of a number.

Probably the most interesting use of signed zero occurs in complex arithmetic. To take a simple example, consider the equation $\sqrt{1/z} = 1/\sqrt{z}$. This is certainly true when $z \geq 0$. If $z = -1$, the obvious computation gives $\sqrt{1/(-1)} = i$ and $1/\sqrt{-1} = -i$. Thus, $\sqrt{1/z} \neq 1/\sqrt{z}$! The problem can be traced to the fact that square root is multi-valued in the complex plane, and there is no way to select the values so that it is continuous in the entire complex plane. However, square root is continuous if a *branch cut* consisting of all negative real numbers is excluded from consideration. This leaves the problem of what to do for the negative real numbers, which are of the form $x + i0$, where $x < 0$. Signed zero provides a perfect way to resolve this problem. Numbers of the form $x + i(+0)$ have one sign $i\sqrt{|x|}$ and numbers of the form $x + i(-0)$ on the other side of the branch cut have the other sign $-i\sqrt{|x|}$. In fact, the natural formulas for computing sqrt() will give these results.

Hacker practice: Implement $-1 + i(+0)$ and $-1 + i(-0)$ to find out whether $\sqrt{1/z} = 1/\sqrt{z}$. Notice that your platform will either support complex numbers, or you have to use two doubles and the associated rules through the polar coordinates to represent the sqrt() function.

Although distinguishing between +0 and –0 has advantages, it can occasionally be confusing. For example, signed zero destroys the relation $x = y \Leftrightarrow 1/x = 1/y$, which is false when $x = +0$ and $y = -0$. This is because it is mandated that signed zero has equality of $+0 = -0$ to avoid ambiguous statement in ($x ==$ 0), but INF and NINF are not equal. However, the IEEE committee decided that the advantages of utilizing the sign of zero outweighed the disadvantages, so signed zero is kept in the IEEE 754 standard.

4.3.2 Denormalized underflowed numbers and soft landing

With the "normal" or "normalized" expression when $e < 0$ and the mantissa = $(1.f)_2 > 1$, the smallest number representable in double precision is 2^{-1022}. When we have $x = (1.1011)_2\times2^{-1020}$ and $y = (1.1010)_2\times2^{-1020}$ appear to be representable, legal floating-point numbers. They have a strange arithmetic property: $x - y = 0$ even though $x \neq y$!!! The reason is that $x - y = 2^{-1024}$ is too small to be represented as a normalized number, and so *would be* flushed to zero. We now have to ask: How important is it to preserve the property $x = y \Leftrightarrow x - y = 0$?

It is very easy to imagine writing the code fragment, if ($x \neq y$) then $z = 1/(x - y)$, and much later having a program fail due to a spurious division by zero. Tracking down bugs like this is frustrating and time consuming. On a more philosophical level, computer science textbooks often point out that even though it is currently impractical to prove large programs correct as *formal verification*, code testing/validating always results in much better and reliable code against programmer bugs and attacking virus. For example, introducing invariants is quite useful in code verification, even if they aren't going to be used as part of a formal proof. The floating-point code is just like any other code: it helps to have provable facts on which to depend. Similarly, knowing that $x = y \Leftrightarrow x - y = 0$ is most usually true makes writing reliable floating-point code easier. If it is not true for all numbers, it cannot be used to prove anything. The IEEE standard uses denormalized numbers (aka, "**denormals**") to implement soft landing to approach zero when the computational result is below 2^{-1022}, which can then guarantee $x = y \Leftrightarrow x - y = 0$ to a much larger range, as well as other useful relations.[6] The idea behind denormals for soft landing to approach 0 is very simple. When the exponent is $e_{\min}$ ($e_{\min} = -1022$ for double precision), the Mantissa is denoted as denormalized. The exponent $e_{\min}$ is used to represent denormals. More formally, if the bits in the mantissa field are $b_1, b_2, ..., b_{p-1}$, and the value of the exponent is e:

- When $e > e_{\min} - 1$, the number being represented is $1.b_1b_2...b_{p-1} \times 2^e$;
- When $e = e_{\min} - 1$, the number being represented is $0.b_1b_2...b_{p-1} \times 2^{e+1}$. [7]

[6] The "denormals" are the most controversial part of the standard and probably accounted for the long delay in getting IEEE 754 approved in 1984. Many high performance hardware that claims to be IEEE compatible does not support denormalized numbers directly, but rather traps (use throw and catch) when consuming or producing denormals, and leaves it to software to simulate the IEEE standard.

[7] Here I use $b_1b_2...b_{p-1}$ to conform to Bindel's book with $p = 53$ in double precision. $p = s + f$ with s=1 sign bit and f=52 mantissa bits.

The +1 in the exponent is needed because denormals have an exponent of $e_{\min}$, not $e_{\min} - 1$. With denormals, $x - y$ does not flush to zero but is instead represented by the denormalized number. This behavior is called gradual *underflow* or **soft landing**. It is easy to verify that $x = y \Leftrightarrow x - y = 0$ always holds when using gradual underflow.

Notice that the gap between 0 and the smallest normalized floating-point number is 2^{-1022} when denormals are not used. If the result of a floating-point calculation falls into this gap, it can be flushed to zero and cause an artificial discontinuity. When denormals of soft landing are used, the "gap" is filled in, and when the result of a calculation is less than 2^{-1022}, it is represented by the nearest denormal, which avoids the "flush to zero" by another 52 bits! Because of this, many algorithms that can have large relative error for normalized numbers close to the underflow threshold are well-behaved in this range with soft landing.

5 Data structures in C++

5.1 Conventional data structures of array and struct

To handle large data, it is often desirable to create a data "structure". Data structure needs to be thought very carefully before designing the program, as a modification to the data structure will change almost every part that uses the given structure, which means the least code-reuse and the most danger in accidental errors. One of the main reasons to migrate from C to C++ is to avoid pinning down the data structure too early. An "object" (an improved version of struct) has pre-defined methods of access (aka procedural interface), instead of the defined data structure.

The simplest data structure, assuming all elements are of basic types, is the "array" in most languages. Array has a regular "pointer" structure and fixed element size, and access to arbitrary elements can be done very fast by performing integer arithmetic to the index. This is true for any dimension of arrays, which makes it extremely efficient in execution and in memory usage for both serial and parallel computing. It is of no surprise that most operations of linear algebra and physical simulation will prefer arrays. However, the real world may not be that "regular", and there are concerns that may lead to inconvenience or even large inefficiency:

- Array size often needs to be pre-determined. Changing an array size will be very inefficient.
- There is only a physical sequence. Multiple topological sequences cannot be easily expressed. For example, element at a[2][3] will only have neighbors of a[1][3]; a[3][3]; a[2][1]; a[2][4], but there is no ready way to build a topological link.
- Elements are homogeneous, and often they are required to be the supported basic types, instead of user defined types.

To remedy this, the conventional C supports "struct", as an inhomogeneous combination of different data, and pointers to build more topologically flexible data structures like linked list, stack, and trees. For example, to create a linked list of ID cards:

```
struct idCard
{
      string name;
      int yearBirth;
      …
      idCard* nextCard;
};
```

Or a tree structure for faster sorting of car positions in 2D:

```
struct     carPosition
{
      double xCarPos;
      double yCarPos;
      …
      carPosition* leftBranch;
      carPosition* rightBranch;
}
```

5.2 Extension in C++ to vector and class

C++ extends the homogeneious data set for array and the inhomogeneous data set for struct in two ways: `vector` (as a resizable array) and `class` (methods and security extension).

The new type `vector` is similar to an array in concept, and is implemented in C++ as a standard template library (STL)[8]. In comparison with a standard array implementation, `vector` relaxes two constraints: the element does not need to be a system-defined type, and the size can change during execution. In C/C++ convention, an array is declared by:

```
int an_array[ 10 ];
```

By using vector in STL:

```
#include <vector>
using namespace std;
vector<int> a_vector( 10 );
```

Loosely speaking, `vector` is just a built-in class that we will introduce immediately next. There are built-in interface to vector, almost similar to a function call, such as:

```
a_vector.size();
```

To readjust the size of the declared a_vector, consider the following two code segments:

```
vector<int> a_vector( 10 );
a_vector[ 10 ] = 10; // the last valid element is 9, giving a compiler error

vector<int> a_vector( 10 );
a_vector.push_back( 10 ); // .push_back resizes avector to have 11 elements
a_vector[ 10 ] = 10;      // Valid statement
```

In original array representation, you would need to declare a new array and then make the entire copy, and then free the old array. This is rather inconvenient for programmers if the resizing is only occasional.

5.2.1 Philosophical change in data access from struct to class

[8] STL contains a large collection of templates, and is beyond the scope here. In my humble opinion, except a few standard templates like vector, many are too complicated and hard for general purposes.

For about 25 years, the data structure in C have implemented most operating systems and library packages. Around the start of 1990s, software researchers started to introduce the concepts of Object-Oriented Programming (OOP) to give the programmer better abilities in modular programming, which is very critical for debugging and validating large codes. OOP has three essences on supported data structures[9]:

- **Data hiding**: Two fundamental reasons behind data hiding. First, experiences told us that change of data structure is often needed when codes evolve with new specification but need backward compatibility. Many programmers implement code with a specific data structure in mind[10]. Change of struct will not only change most of the codes (which makes them less reliable), but also problematic for backward compatibility. Second, in a large program development, all parts of the codes have to been writable to all programmers, which is not only dangerous, but also much more difficult to manage. Data hiding means the access of the "object" or "class" will be ONLY through a *public* method[11] or procedural interface, instead of exposing the *private* data directly. To use an object (or call a function), as long as the procedural interface does not change, the upstream programmer will NOT touch the implementation of the object. If you find a mistake in your logic, a higher chance you only need to modify the implementation of the object access method in one place, instead of everywhere the data structure is used. Moreover, if you need to change your private data structure later either for storage efficiency or porting to different platforms, you only need to change the private part of the class, and NOTHING else.
- **Inheritance**: Inheritance means that one class (subclass) inherits its traits from another class (superclass). This is mainly to ease up the procedural interface and collect all identical traits in various subclasses into the feature in the superclass. When the common feature needs to be changed, the superclass is modified in one place to avoid redundant code change. When individual feature of subclass needs to be changed, other subclasses belonging to the same superclass do not need to be touched. By creating "inheritance", we can maximize code reuse and minimize code change during revision. We will cover more of this feature when practical situation arises.
- **Polymorphism**: Polymorphism means the same procedural interface can handle multiple different classes (you can see the same purpose here for maximize code reuse and minimize code change). This is especially convenient in revision to meet the new specification while keeping backward compatibility. For an older version, the class has an established interface and implementation. For the next version, we can add in a new class (without touching the old class implementation), and employ polymorphism in the programming language to handle both. Again, we will cover more of this feature when practical situation arises.

5.2.2 Implementing functional abstraction in OOP

5.2.2.1 Implementation in conventional language by self discipline

To implement data hiding in the functional call, you can follow the conventional C struct. An example to implement a chess board is shown below:

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* and others */ };
```

[9] There are other features in OOP not directly related to data structure, such as operator overloading. We will talk about it later.
[10] This is often related to "redundant codes", which can be a redundant code segment for a repeated algorithm, or can be a repeated assumption of data structure.
[11] "Functional abstraction" is another name for accessing the objects by a function instead of data or data pointers, as it most critical for upstream functions to know how you use the data, instead of how you store the data.

```
enum PlayerColor { PC_WHITE, PC_BLACK };

struct ChessBoard
{
      ChessPiece board[ 8 ][ 8 ];
      PlayerColor whose_move;
};
```

You can create functions that operate on the board, taking the board as a parameter to the function. You need to remember yourself that you will always access the struct ChessBoard through a function call, instead of directly like ChessBoard.board.

```
ChessPiece getPiece (const ChessBoard *p_board, int x, int y)
{
      return p_board->board[ x ][ y ];
}

PlayerColor getMove (const ChessBoard *p_board)
{
      return p_board->whose_move;
}

void makeMove (ChessBoard* p_board, int from_x, int from_y, int to_x, int
to_y)
{
      // normally, we need some code that validates the move first
      p_board->board[ to_x ][ to_y ] = p_board->board[ from_x ][ from_y ];
      p_board->board[ from_x ][ from_y ] = EMPTY_SQUARE;
}
```

You can use `getMove` and `makeMove` just like any other function, but this is mainly designed to access the ChessBoard struc. You can say this is OOP, but by self discipline instead of language requirements.

5.2.2.2 Implementation by enforced OOP language features

Alternatively, you can choose to enforce data hiding by implementing the class with `public` and `private` parts.

```
ChessBoard b;
// first we'd need some code to initialize board
// then we can use it, like so...
      getMove( & b );
      makeMove( & b, 0, 0, 1, 0 ); // move a piece from 0, 0 to 1, 0


enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* and others */ };
enum PlayerColor { PC_WHITE, PC_BLACK };

class ChessBoard
{
public:
      ChessPiece getPiece (int x, int y);
      PlayerColor getMove ();
      void makeMove (int from_x, int from_y, int to_x, int to_y);
private:
      ChessPiece _board[ 8 ][ 8 ];
      PlayerColor _whose_move;
};

// Method definitions are exactly the same!
ChessPiece ChessBoard::getPiece (int x, int y)
{
      return _board[ x ][ y ];
}

PlayerColor ChessBoard::getMove ()
{
      return _whose_move;
}

void ChessBoard::makeMove (int from_x, int from_y, int to_x, int to_y)
{
      // normally, we'd want some code that validates the move first
      _board[ to_x ][ to_y ] = _board[ from_x ][ from_y ];
      _board[ from_x ][ from_y ] = EMPTY_SQUARE;
}
```

Notice that this class declaration looks a lot like the conventional `struct`, but the method is in the public area and the data structure is in the private area. You can surely implement the method within the class, but often this will clutter the object definition and expose too much[12] about your code implementation details. Following the same syntax of functional name declaration, you can just declare the method with its arguments, and implement the method somewhere else (which will not be shown to common programmers just using this object). To help the compiler know which class or struct to associate the method with, the *scoping* operator `::` is shown above.

5.2.3 Initialization in class by object constructor by constructor

One of the common bugs is lack of proper initialization (memory allocation and default values) when a variable or object is declared. This is in general called the "constructor". Surely within the lifecycle of the variable or object, when everything is done for, the memory needs to be returned. A specific way to initialize a class object is by a special function call with the same name with the class and has no "return" within its scope.

```
enum ChessPiece { EMPTY_SQUARE, WHITE_PAWN /* and others */ };
enum PlayerColor { PC_WHITE, PC_BLACK };

class ChessBoard
{
public:
      ChessBoard (); // <-- no return value at all!  For initialization.
      PlayerColor getMove ();
      ChessPiece getPiece (int x, int y);
      void makeMove (int from_x, int from_y, int to_x, int to_y);
```

[12] Remember that we are implementing "hiding" now. One of the guarding principles to prevent people from relying on a data structure or details or from trying to change the structure or details is: Do not let them know!!! We practice this principle in our social life and even in government/organization management often, but OOP actually formalize a method!

```
private:
      ChessPiece _board[ 8 ][ 8 ];
      PlayerColor _whose_move;
};

// This is the special function for the object constructor
//
ChessBoard::ChessBoard () // <-- still no return value, and no return.
{
      _whose_move = PC_WHITE;
      // start off by emptying the whole board, then fill it in with pieces
      for ( int i = 0; i < 8; i++ )
      {
            for (int j = 0; j < 8; j++ )
            {
                  _board[ i ][ j ] = EMPTY_SQUARE;
            }
      }
      // other code to initialize the board...
}
```

5.2.4 Deleting an object by destructor

As C/C++ handles the memory directly (this is the most reliable way to handle large data set, instead of relying on the compiler or OS as in Java or Python), it is a good habit to free or destroy the object when you know that you have finished using it. This is called the "destructor". If you do not do this, it can cause memory leak, which can eventually accumulate and make your hardware perform more slowly (allocated memory does not have use, but cannot be re-used either). The built-in recursive destructor is:

```
class LinkedList
{
public:
      LinkedList (); // constructor
      ~LinkedList (); // destructor, notice the (~)
      void insert (int val); // adds a node
private:
      LinkedListNode *_p_head;
};

LinkedListNode::~LinkedListNode ()
{
      delete p_next;  // like the constructor, no explicit return
}
```

If you do not trust the built-in recursive destructor, you can alternatively do the recursion explicitly:

```
LinkedList::~LinkedList ()
{
      LinkedListNode *p_itr, *p_tmp;

      *p_itr = _p_head;       // _p_head is in the private data of LinkedList
                              // The scope :: will inform the compiler
      while ( p_itr != NULL )
      {
            LinkedListNode *p_tmp = p_itr->p_next;
            delete p_itr;
            p_itr = p_tmp;
        }
    }
```