What Every Developer Should Know about Floating-point Computing

Yong-Ming Li

Intergraph

Introduction

"*Floating point numbers are like piles of sand; every time you move them around, you lose a little sand and pick up a little dirt.*" – Dr. Brian Kernighan (Princeton professor and contributor of UNIX) and Dr. P. J. Plauger (entrepreneur and author of many programming books).

Introduction

"*Floating point numbers are like piles of sand; every time you move them around, you lose a little sand and pick up a little dirt.*" – Dr. Brian Kernighan (Princeton professor and contributor of UNIX) and Dr. P. J. Plauger (entrepreneur and author of many programming books).

This quote gives a good summary about what we do with respect to floating-point arithmetic: Whenever we move a geometric model or perform geometric operations on it, we would loose some accuracy and pick up some numerical noise.

Introduction

"*Floating point numbers are like piles of sand; every time you move them around, you lose a little sand and pick up a little dirt.*" – Dr. Brian Kernighan (Princeton professor and contributor of UNIX) and Dr. P. J. Plauger (entrepreneur and author of many programming books).

This quote gives a good summary about what we do with respect to floating-point arithmetic: Whenever we move a geometric model or perform geometric operations on it, we would loose some accuracy and pick up some numerical noise.

Sources of numerical noise: floating-point representation and mathematical approximation.

Introduction

Example 1:

What output do we expect from the following simple program?

```
void main()
{
   double a, b;

   a =  0.8485;
   b = 10.0 * a;

   printf("a = %.16lf,  b = %.16lf\n", a, b);
}
```

Introduction

Example 1:

What output do we expect from the following simple program?

```
void main()
{
   double a, b;

   a =  0.8485;
   b = 10.0 * a;

   printf("a = %.16lf,  b = %.16lf\n", a, b);
}
```

```
a = 0.8485000000000000,  b = 8.4849999999999994
```

Introduction

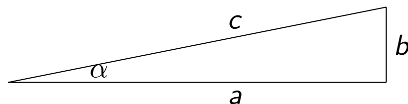
Example 2:

Most developers have computed an angle via trigonometry functions. To obtain the angle α, should we call

$$\alpha = \arccos(a/c),$$

or call

$$\alpha = \arctan(b/a)?$$



Introduction

Floating-point representation and floating-point arithmetic are one of the most confusing areas for most developers.

Introduction

Floating-point representation and floating-point arithmetic are one of the most confusing areas for most developers.

If you are not confident to answer these questions, no need to feel embarrassed since **you are not alone**!

Introduction

In a February, 1998 keynote address entitled *Extensions to Java for Numerical Computing*, Dr. James Gosling, the creator of Java programming language, asserted

"*95% of folks out there are completely clueless about floating-point.*"

Introduction

In a February, 1998 keynote address entitled *Extensions to Java for Numerical Computing*, Dr. James Gosling, the creator of Java programming language, asserted

"*95% of folks out there are completely clueless about floating-point.*"

His assertion may sound far-fetched to many but has good ground in consideration of the following examples:

Introduction

- In 1984 the Vancouver Stock Exchange was in panic because its index was undervalued by almost 50% even though the exchange was setting records for volume and value. Investigation found that index value was computed using 4 decimal places and truncating (not rounding) the result to 3. When this computation problem was corrected, the index sat at 1098.892 (the nominal value is 1,000).

Introduction

- In 1984 the Vancouver Stock Exchange was in panic because its index was undervalued by almost 50% even though the exchange was setting records for volume and value. Investigation found that index value was computed using 4 decimal places and truncating (not rounding) the result to 3. When this computation problem was corrected, the index sat at 1098.892 (the nominal value is 1,000).
- In 1991 an American Patriot missile missed an Iraqi Scud missile and hit an Army barrack, killing 26 people. The cause was later determined to be an inaccurate calculation caused by measuring time in a tenth of a second that cannot be represented exactly since a 24-bit floating-point was used.

Introduction

- In 1984 the Vancouver Stock Exchange was in panic because its index was undervalued by almost 50% even though the exchange was setting records for volume and value. Investigation found that index value was computed using 4 decimal places and truncating (not rounding) the result to 3. When this computation problem was corrected, the index sat at 1098.892 (the nominal value is 1,000).
- In 1991 an American Patriot missile missed an Iraqi Scud missile and hit an Army barrack, killing 26 people. The cause was later determined to be an inaccurate calculation caused by measuring time in a tenth of a second that cannot be represented exactly since a 24-bit floating-point was used.
- European Ariane 5's first test flight in 1996 failed, with the rocket self-destructing 37 sec. after launch due to a malfunction in control software. The malfunction was traced to unanticipated overflow in converting 64-bit floating-point number to a 16-bit signed integer.

Introduction

Intergraph has been in software business for over 50 years. Our developers are not immutable! Here are some of our own examples:

Introduction

Intergraph has been in software business for over 50 years. Our developers are not immutable! Here are some of our own examples:

- When PPM started to use Coverity® to check software defects, one of the top five defect criterions was the direct comparison of floating-point values for equality, which, as a rule of thumb, should rarely be used in floating-point computing.

Introduction

Intergraph has been in software business for over 50 years. Our developers are not immutable! Here are some of our own examples:

- When PPM started to use Coverity® to check software defects, one of the top five defect criterions was the direct comparison of floating-point values for equality, which, as a rule of thumb, should rarely be used in floating-point computing.
- It is widely believed that 100 kilometer workspace limit is imposed by CoreMath due to the limitation of their algorithms. The actual reason for this limit is because of the floating-point representation in 32 bit machine!

Introduction

Intergraph has been in software business for over 50 years. Our developers are not immutable! Here are some of our own examples:

- When PPM started to use Coverity® to check software defects, one of the top five defect criterions was the direct comparison of floating-point values for equality, which, as a rule of thumb, should rarely be used in floating-point computing.
- It is widely believed that 100 kilometer workspace limit is imposed by CoreMath due to the limitation of their algorithms. The actual reason for this limit is because of the floating-point representation in 32 bit machine!
- AutoCAD "unitless" models and other foreign models have been directly imported to PPM, even though *Autodesk University*'s article warns that "*this (unitless) selection can only make sense if you never ever switch between metric and imperial, or never exchange files with other companies.*"

Floating-point representation

Computers use a fixed number of bits to represent a character, integer, real number, etc. An n-bit storage location can represent up to 2^n distinct entities. For example, a 2-bit memory location can hold one of these four binary patterns:

$$00, \quad 01, \quad 10, \quad 11.$$

Hence, it can represent at most 4 distinct entities (e.g., integer 0 to 3).

Floating-point representation

Computers use a fixed number of bits to represent a character, integer, real number, etc. An n-bit storage location can represent up to 2^n distinct entities. For example, a 2-bit memory location can hold one of these four binary patterns:

$$00, \quad 01, \quad 10, \quad 11.$$

Hence, it can represent at most 4 distinct entities (e.g., integer 0 to 3).

Modern personal computers have either 32-bit or 64-bit architectures that can hold unsigned integers ranging

32-bit	0 to 4,294,967,295 ($= 2^{32} - 1$).
64-bit	0 to 18,446,744,073,709,551,615 ($= 2^{64} - 1$).

Floating-point representation

These ranges are "huge" for counting but very limited for real world computing.

Floating-point representation

These ranges are "huge" for counting but very limited for real world computing.

For example, the factorial of 21 is

$$21! = 51,090,942,171,709,440,000$$

which means that a 64-bit computer would not be able to compute factorials of 21 or larger if it has only integer representation.

Floating-point representation

These ranges are “huge” for counting but very limited for real world computing.

For example, the factorial of 21 is

$$21! = 51,090,942,171,709,440,000$$

which means that a 64-bit computer would not be able to compute factorials of 21 or larger if it has only integer representation.

Lack of dynamic range is not the only limitation of integer representation. Another significant drawback of integer presentation is the inability to represent fractional values.

Floating-point representation

The term *floating-point* refers to the fact that its decimal point can "*float*" anywhere with the help of exponents. For example, we can write 3141.59 in *scientific notation* as

$$314.159 \times 10, \quad 31.4159 \times 10^2, \quad 3.14159 \times 10^3, \quad \text{etc.}$$

Floating-point representation

The term *floating-point* refers to the fact that its decimal point can "*float*" anywhere with the help of exponents. For example, we can write 3141.59 in *scientific notation* as

$$314.159 \times 10, \quad 31.4159 \times 10^2, \quad 3.14159 \times 10^3, \quad \text{etc.}$$

In general, a real number in scientific notation is written as

$$m \times 10^n$$

where the exponent n is an integer and m any real number called the *mantissa* or *significand*. If $1 \leq |m| < 10$, the above representation is called the *normalized* scientific notation.

Floating-point representation

The mantissa consists of *significant digits* that carry meaning contributing to its accuracy.

Floating-point representation

The mantissa consists of *significant digits* that carry meaning contributing to its accuracy.

When I say that the diameter of my watch is 27 mm, I imply that it is measured via an ordinary ruler, which is accurate only to millimeter. Therefore, there are only two significant digits.

Floating-point representation

The mantissa consists of *significant digits* that carry meaning contributing to its accuracy.

When I say that the diameter of my watch is 27 mm, I imply that it is measured via an ordinary ruler, which is accurate only to millimeter. Therefore, there are only two significant digits.

If the diameter is given in meter, i.e., 0.027m, there are still only two significant digits. Zeros in this case are simply place holders.

Floating-point representation

The mantissa consists of *significant digits* that carry meaning contributing to its accuracy.

When I say that the diameter of my watch is 27 mm, I imply that it is measured via an ordinary ruler, which is accurate only to millimeter. Therefore, there are only two significant digits.

If the diameter is given in meter, i.e., 0.027m, there are still only two significant digits. Zeros in this case are simply place holders.

If I say the diameter is 27.000 mm, I imply here that the diameter is measured by using a high-precision instrument that is accurate to the micrometer. So, there are five significant digits.

Floating-point representation

The mantissa consists of *significant digits* that carry meaning contributing to its accuracy.

When I say that the diameter of my watch is 27 mm, I imply that it is measured via an ordinary ruler, which is accurate only to millimeter. Therefore, there are only two significant digits.

If the diameter is given in meter, i.e., 0.027m, there are still only two significant digits. Zeros in this case are simply place holders.

If I say the diameter is 27.000 mm, I imply here that the diameter is measured by using a high-precision instrument that is accurate to the micrometer. So, there are five significant digits.

To avoid confusion in counting significant digits, write a real number in normalized scientific notation. In this case, *all digits are significant*!

Floating-point representation

Floating point notation in computers is analogous to the scientific notation of decimal numbers. In computer, a 32-bit is used for the *single-precision* floating-point format (known as **float** in C/C++) and is divided into three fields as follows:

Floating-point representation

Floating point notation in computers is analogous to the scientific notation of decimal numbers. In computer, a 32-bit is used for the *single-precision* floating-point format (known as **float** in C/C++) and is divided into three fields as follows:

- 1 bit for the sign,

Floating-point representation

Floating point notation in computers is analogous to the scientific notation of decimal numbers. In computer, a 32-bit is used for the *single-precision* floating-point format (known as **float** in C/C++) and is divided into three fields as follows:

- 1 bit for the sign,
- 8 bits for the exponent,

Floating-point representation

Floating point notation in computers is analogous to the scientific notation of decimal numbers. In computer, a 32-bit is used for the *single-precision* floating-point format (known as **float** in C/C++) and is divided into three fields as follows:

- 1 bit for the sign,
- 8 bits for the exponent,
- 23 bits for the mantissa (or significand).

Floating-point representation

Floating point notation in computers is analogous to the scientific notation of decimal numbers. In computer, a 32-bit is used for the *single-precision* floating-point format (known as **float** in C/C++) and is divided into three fields as follows:

- 1 bit for the sign,
- 8 bits for the exponent,
- 23 bits for the mantissa (or significand).

Thus, a real number in single-precision is stored in computer as

$$x = \pm m \times 2^{E}$$

where $1 \leq m = (b_0 b_1 b_2 \cdots b_{22})_2 < 2$ and $b_0, b_1, \cdots, b_{22}$ are bits.

Floating-point representation

It was said that the 23 bits in mantissa contributes to the accuracy of the number it represents. Hence, a **float** variable has only 7 significant decimal digits ($\log_{10} 2^{23} \approx 7$). This implies that a **float** variable is incapable of representing a geometry that needs to be as precise as 1.0×10^{-6}!

Floating-point representation

It was said that the 23 bits in mantissa contributes to the accuracy of the number it represents. Hence, a **float** variable has only 7 significant decimal digits ($\log_{10} 2^{23} \approx 7$). This implies that a **float** variable is incapable of representing a geometry that needs to be as precise as 1.0×10^{-6}!

In scientific computation, a *double-precision* format (known as **double** in C/C++) is usually used. The 64-bit is divided into three fields as follows:

- 1 bit for the sign,
- 11 bits for the exponent,
- 52 bits for the significand.

Floating-point representation

It was said that the 23 bits in mantissa contributes to the accuracy of the number it represents. Hence, a **float** variable has only 7 significant decimal digits ($\log_{10} 2^{23} \approx 7$). This implies that a **float** variable is incapable of representing a geometry that needs to be as precise as 1.0×10^{-6}!

In scientific computation, a *double-precision* format (known as **double** in C/C++) is usually used. The 64-bit is divided into three fields as follows:

- 1 bit for the sign,
- 11 bits for the exponent,
- 52 bits for the significand.

Thus, a double-precision variable has $15 \sim 16$ significant decimal digits (52-bit is equivalent to $\log_{10} 2^{52} \approx 15 \sim 16$).

What else to know

We now know that a **float** variable has 7 significant decimal digits, and a **double** has 15 significant decimal digits. What else should we know?

What else to know

We now know that a **float** variable has 7 significant decimal digits, and a **double** has 15 significant decimal digits. What else should we know?

- Any infinite number (e.g., $1/3$, π, $\sqrt{2}$, etc.) will be truncated when it is stored in computer. To retain maximum significant digits, a **double** should always be used.

What else to know

We now know that a **float** variable has 7 significant decimal digits, and a **double** has 15 significant decimal digits. What else should we know?

- Any infinite number (e.g., $1/3$, π, $\sqrt{2}$, etc.) will be truncated when it is stored in computer. To retain maximum significant digits, a **double** should always be used.
- It may surprise many developers that *most finite numbers* cannot be represented precisely in a binary system!

What else to know

We now know that a **float** variable has 7 significant decimal digits, and a **double** has 15 significant decimal digits. What else should we know?

- Any infinite number (e.g., $1/3$, π, $\sqrt{2}$, etc.) will be truncated when it is stored in computer. To retain maximum significant digits, a **double** should always be used.
- It may surprise many developers that *most finite numbers* cannot be represented precisely in a binary system!

  For example, when 0.1 in base 10 is represented in a binary system, it is

  $$(1.100110011\cdots)_2 \times 2^{-4},$$

  which has to be truncated when assigning it to a **float** or **double**. Again, a **double** should be used to retain maximum significant digits.

What else to know

- Geometric models need to be confined in certain dimension.

What else to know

- Geometric models need to be confined in certain dimension.

  For 32-bit computer, a **double** has 15 significant digits. If we consider five of the least significant digits to represent round-off errors, there are roughly 10 digits to represent the dynamic range of numbers (smallest and largest numbers) within an object space. Accordingly, the angular tolerance, which is the smallest tolerance, is set to 10^{-10}. When the distance and angular tolerances are defined, the longest line we can model would be

$$\frac{10^{-6}}{10^{-10}} = 10,000\,\text{units}.$$

  This is why ACIS confines the model space to 10K units.

What else to know

- Geometric models need to be confined in certain dimension.

  For 32-bit computer, a **double** has 15 significant digits. If we consider five of the least significant digits to represent round-off errors, there are roughly 10 digits to represent the dynamic range of numbers (smallest and largest numbers) within an object space. Accordingly, the angular tolerance, which is the smallest tolerance, is set to 10^{-10}. When the distance and angular tolerances are defined, the longest line we can model would be

$$\frac{10^{-6}}{10^{-10}} = 10,000 \, \text{units}.$$

  This is why ACIS confines the model space to 10K units.

  CoreMath relaxes the restriction by allowing a model of 100K in dimension and swallows the numerical noises internally.

What else to know

- Foreign data needs to be scaled before importing into our system. It is wrong to import the so-called unitless models directly to our system!

What else to know

- Foreign data needs to be scaled before importing into our system. It is wrong to import the so-called unitless models directly to our system!

  We have seen numerous cases in which models sit as far away as on the Moon (roughly 4×10^8 meters). It requires at least 8 significant decimal digits to represent such models.

What else to know

- Foreign data needs to be scaled before importing into our system. It is wrong to import the so-called unitless models directly to our system!

  We have seen numerous cases in which models sit as far away as on the Moon (roughly 4×10^8 meters). It requires at least 8 significant decimal digits to represent such models.

  When applying geometric operations to these models, our system requires the accuracy to be 10^{-6} meters. This obviously requires additional 6 digits. Therefore, it consumes at least 14 significant decimal digits to represent these unscaled models, leaving no room for round-off errors that occur in arithmetic computations.

What to do

Every developer performs arithmetic operations ($+$, $-$, $\times$, $\div$, etc.) in their code. What should we do to minimize the loss of significant digits?

What to do

Every developer performs arithmetic operations ($+$, $-$, $\times$, $\div$, etc.) in their code. What should we do to minimize the loss of significant digits?

- Avoid dividing a large number by a very small one, as it is equivalent to multiplying a very large number to the numerical noise. If possible, try to divide numbers that have the same relative magnitudes.

What to do

Every developer performs arithmetic operations ($+$, $-$, $\times$, $\div$, etc.) in their code. What should we do to minimize the loss of significant digits?

- Avoid dividing a large number by a very small one, as it is equivalent to multiplying a very large number to the numerical noise. If possible, try to divide numbers that have the same relative magnitudes.
- Avoid subtraction between two similar numbers. Subtracting two similar numbers results in great loss of significant digits!

What to do

Every developer performs arithmetic operations ($+$, $-$, $\times$, $\div$, etc.) in their code. What should we do to minimize the loss of significant digits?

- Avoid dividing a large number by a very small one, as it is equivalent to multiplying a very large number to the numerical noise. If possible, try to divide numbers that have the same relative magnitudes.
- Avoid subtraction between two similar numbers. Subtracting two similar numbers results in great loss of significant digits! This may be understood by looking at the relative error of subtraction. Let $\bar{a} = a(1 + \Delta a)$ and $\bar{b} = b(1 + \Delta b)$ be two approximation numbers to a and b. The relative error caused by small perturbations Δa and Δb is

$$\frac{|\bar{x} - x|}{|x|} = \frac{|a\Delta a - b\Delta b|}{|a - b|},$$

where $\bar{x} = \bar{a} - \bar{b}$ and $x = a - b$. If $a \approx b$, small Δa and Δb can lead to very large relative error in x (known as the *cancellation error*).

What to do

- Avoid adding a very small number to a large number. Floating point computation has limited precision. If a large float is added to a small float, the small float may be too negligible to change the value of the larger float. In performing a sequence of additions, the numbers should be added in the order of smallest in magnitude to largest in magnitude. This ensures that the cumulative sum of many small arguments is not negligible.

What to do

- Avoid adding a very small number to a large number. Floating point computation has limited precision. If a large float is added to a small float, the small float may be too negligible to change the value of the larger float. In performing a sequence of additions, the numbers should be added in the order of smallest in magnitude to largest in magnitude. This ensures that the cumulative sum of many small arguments is not negligible.
- Reduce arithmetic computations to minimize cumulative error. Each arithmetic operation adds numerical noise at least linearly if not exponentially (Please note some numerical noises may cancel each other out due to different sign). Therefore, reduction of arithmetic operations generally improves numerical stability.

What to do

- Select the most stable algorithm when several approaches are available. A big problem with floating-point arithmetic is that it does not follow the standard rules of algebra. Normal algebraic rules apply only to infinite precision arithmetic.

What to do

- Select the most stable algorithm when several approaches are available. A big problem with floating-point arithmetic is that it does not follow the standard rules of algebra. Normal algebraic rules apply only to infinite precision arithmetic.
- Transform geometric models to the origin before doing complex geometric computations. It is not uncommon for CAD system users to create their models far away from the origin of the global coordinate system. In this case, any geometry will have large (x, y, z) components and hence leave fewer decimal digits for computations. Transforming them close to the origin will greatly improve the stability of numerical computation.

Example 1

In school, we were all taught that solutions to a quadratic equation $ax^2 + bx + c = 0$ is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Example 1

In school, we were all taught that solutions to a quadratic equation $ax^2 + bx + c = 0$ is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

The above solutions can be numerically unstable if $b^2 \gg 4|ac|$ because of subtraction between two similar numbers.

Example 1

In school, we were all taught that solutions to a quadratic equation $ax^2 + bx + c = 0$ is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

The above solutions can be numerically unstable if $b^2 \gg 4|ac|$ because of subtraction between two similar numbers.

In implementation, it is better to use the identity $x_1 x_2 = c/a$ to calculate one root from another.

- If $b < 0$, calculate $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}, \quad x_2 = \frac{c}{ax_1}$.
- If $b > 0$, calculate $x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}, \quad x_1 = \frac{c}{ax_2}$.

Example 1

If you are not convinced, try the following code fragment

```
float x2, x1_bad, x1_good, delta;
delta = (float)sqrt(b * b - 4.0 * a * c);
x2 = (-b - delta)/(2.0 * a);
x1_bad = (-b + delta)/(2.0 * a);
x1_good = c / (a * x2);
```

with

$$
\begin{aligned}
a &= 1.0, \\
b &= 200.0, \\
c &= -1.5 \times 10^{-3}, -1.5 \times 10^{-4}, -1.5 \times 10^{-5}.
\end{aligned}
$$

You will get the following three sets of solutions:

Example 1

```
c = 1.0e-3:
    x2      -200.00000
    x1_bad  -7.6293945e-006
    x1_good -7.5000003e-006

c = 1.0e-4:
    x2      -200.00000
    x1_bad   0.00000000
    x1_good -7.5000003e-007

c = 1.0e-5:
    x2      -200.00000
    x1_bad   0.00000000
    x1_good -7.4999996e-008
```

Example 2

Evaluation of many transcendental functions such as $\sin x$, $\cos x$, e^x and so on is often performed via evaluation of a truncated *Taylor series*. For example, expanding the natural exponential function gives

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots$$

Example 2

Evaluation of many transcendental functions such as $\sin x$, $\cos x$, e^x and so on is often performed via evaluation of a truncated *Taylor series*. For example, expanding the natural exponential function gives

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots$$

Let $P_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$. The evaluation terminates if

$$|P_n(x) - P_{n-1}(x)| = \frac{x^n}{n!} < \epsilon.$$

Example 2

Evaluation of many transcendental functions such as $\sin x$, $\cos x$, e^x and so on is often performed via evaluation of a truncated *Taylor series*. For example, expanding the natural exponential function gives

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots$$

Let $P_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$. The evaluation terminates if

$$|P_n(x) - P_{n-1}(x)| = \frac{x^n}{n!} < \epsilon.$$

A simple implementation might be:

- Compute the ith term and sum it to the function value at each iteration.
- Check if the ith term is less than the given tolerance. If it is, terminate the iteration. Otherwise, repeat the iteration.

Example 2

The implementation is given below:

```
void EvaluateExpX(double x, double epsilon, double &f)
{
   int      i;
   double   term;

   f = 1.0;
   i = 1;
   while (true)
   {
      term = pow(x, i) / Factorial(i);
      f += term;
      if (fabs(term) < epsilon)
         break;
      i++;
   }
}
```

Example 2

The computation of factorials is implemented as follows:

```
inline long long Factorial(long long n)
{
   long long Result = 1;
   while (n > 1)
   {
      Result *= n;
      n--;
   }
   return Result;
}
```

This implementation has at least two problems: *overflow* of **long long** when $n > 20$ and *inefficiency* in computation.

Example 2

We can avoid the overflow and improve the performance with the following implementation:

```
void apiEvalExp(double x, double epsilon, double &f)
{
   int         i = 1;
   double      term = 1.0;
   f = 0.0;
   while (true)
   {
      f += term;
      if (fabs(term) < epsilon)
         break;
      term = term * x / i;
      i++;
   }
}
```

Example 2

If $x > 0$, calling `apiEvalExp` yields a converged value w.r.t. the one obtained by calling the standard C function `exp(x)`.

Example 2

If $x > 0$, calling `apiEvalExp` yields a converged value w.r.t. the one obtained by calling the standard C function `exp(x)`.

However, it may diverge when $x < 0$. For example, calling `apiEvalExp` with $x = -25.0$, we obtain

$$-7.1297804036720779e - 007.$$

But the correct answer should be $1.3887943864964021e - 011$.

Example 2

If $x > 0$, calling `apiEvalExp` yields a converged value w.r.t. the one obtained by calling the standard C function `exp(x)`.

However, it may diverge when $x < 0$. For example, calling `apiEvalExp` with $x = -25.0$, we obtain

$$-7.1297804036720779e - 007.$$

But the correct answer should be $1.3887943864964021e - 011$.

Two results are not only significantly different but also have different signs. How could a converging series lead to a diverging result?

Example 2

If $x > 0$, calling `apiEvalExp` yields a converged value w.r.t. the one obtained by calling the standard C function `exp(x)`.

However, it may diverge when $x < 0$. For example, calling `apiEvalExp` with $x = -25.0$, we obtain

$$-7.1297804036720779e - 007.$$

But the correct answer should be $1.3887943864964021e - 011$.

Two results are not only significantly different but also have different signs. How could a converging series lead to a diverging result?

Referring to the Taylor series, the even terms are positive and odd terms are negative when $x < 0$. Numerical instability is caused by subtraction of similar numbers. One remedy is to treat x as positive to compute f. If $x < 0$, we have $e^{-x} = 1/e^{x}$. Accordingly, $1/f$ is the result for e^{-x}.

Example 3

The angle between two unit vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ may be computed via a *dot-product* of two vectors as follows:

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\| \times \|\mathbf{v}_2\| \cos\theta = \cos\theta \quad \rightarrow \quad \theta = \arccos(\mathbf{v}_1 \cdot \mathbf{v}_2)$$

noting that $\|\mathbf{v}_1\| = 1$ and $\|\mathbf{v}_2\| = 1$.

Example 3

The angle between two unit vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ may be computed via a *dot-product* of two vectors as follows:

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\| \times \|\mathbf{v}_2\| \cos\theta = \cos\theta \quad \rightarrow \quad \theta = \arccos(\mathbf{v}_1 \cdot \mathbf{v}_2)$$

noting that $\|\mathbf{v}_1\| = 1$ and $\|\mathbf{v}_2\| = 1$.

In Cartesian coordinates, $\mathbf{v}_1 = (x_1, y_1, z_1)$ and $\mathbf{v}_2 = (x_2, y_2, z_2)$. Hence,

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1 x_2 + y_1 y_2 + z_1 z_2.$$

Example 3

The angle between two unit vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ may be computed via a *dot-product* of two vectors as follows:

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\| \times \|\mathbf{v}_2\| \cos\theta = \cos\theta \quad \rightarrow \quad \theta = \arccos(\mathbf{v}_1 \cdot \mathbf{v}_2)$$

noting that $\|\mathbf{v}_1\| = 1$ and $\|\mathbf{v}_2\| = 1$.

In Cartesian coordinates, $\mathbf{v}_1 = (x_1, y_1, z_1)$ and $\mathbf{v}_2 = (x_2, y_2, z_2)$. Hence,

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1 x_2 + y_1 y_2 + z_1 z_2.$$

If we use the above formula to compute θ, we will get noticeable numerical noise when two vectors are almost parallel (i.e., θ is very small).

Example 3

To illustrate this instability issue, let's try the following code fragment:

```
double alpha = 1.7453292519943297e-007;
double cos_alpha = cos(alpha);
double theta = acos(cos_alpha);
```

Example 3

To illustrate this instability issue, let's try the following code fragment:

```
double alpha = 1.7453292519943297e-007;
double cos_alpha = cos(alpha);
double theta = acos(cos_alpha);
```

We expect to obtain $\theta = \alpha$ with respect to the machine tolerance. However, $\theta = 1.7441362009524762e{-}007$.

Example 3

To illustrate this instability issue, let's try the following code fragment:

```
double alpha = 1.7453292519943297e-007;
double cos_alpha = cos(alpha);
double theta = acos(cos_alpha);
```

We expect to obtain $\theta = \alpha$ with respect to the machine tolerance. However, $\theta = 1.7441362009524762e{-}007$.

A stable way to compute θ is to compute also the *cross product* of two vectors:

$$\|\mathbf{v}_1 \times \mathbf{v}_2\| = \|\mathbf{v}_1\| \, \|\mathbf{v}_2\| \sin\theta = \sin\theta.$$

Then, the angle is obtained by either $\theta = \arctan\left(\sin\theta / \cos\theta\right)$ or $\theta = \mathrm{atan2}(\sin\theta, \cos\theta)$.

Example 3

To illustrate this instability issue, let's try the following code fragment:

```
double alpha = 1.7453292519943297e-007;
double cos_alpha = cos(alpha);
double theta = acos(cos_alpha);
```

We expect to obtain $\theta = \alpha$ with respect to the machine tolerance. However, $\theta = 1.7441362009524762e{-}007$.

A stable way to compute θ is to compute also the *cross product* of two vectors:

$$\|\mathbf{v}_1 \times \mathbf{v}_2\| = \|\mathbf{v}_1\| \, \|\mathbf{v}_2\| \sin\theta = \sin\theta.$$

Then, the angle is obtained by either $\theta = \arctan\left(\sin\theta / \cos\theta\right)$ or $\theta = \mathrm{atan2}(\sin\theta, \cos\theta)$.

It can be proven that arccos is *ill conditioned* when the angle is small.

The presentation is extracted from Chapter 2 of my book:

A Practical Guide to Developing Computational Software.

It is sold at Amazon.com or Barnes & Noble.

The presentation is extracted from Chapter 2 of my book:

A Practical Guide to Developing Computational Software.

It is sold at Amazon.com or Barnes & Noble.

You can also read:

What Every Computer Scientist Should Know About Floating Point Arithmetic, ACM Computing Surveys, Vol 23, No 1, March 1991, by DAVID GOLDBERG.