

Computer Arithmetic and Computational Errors

2.1. Introduction

Read this section in the book on your own. It may make more sense if you read it after the rest of the chapter.

2.2. Representation of Numbers

It is impossible to cover all kinds of computers. I will concentrate on the ones you are likely to use: The Vax and any computer conforming to the IEEE standard. The latter includes PV and most micros and workstations (IBM PC, Mac, Sun, etc.). When I say “the computer does this”, this means “any computer you are likely to use does this”. There are exceptions to everything.

The standard numerical data types available in most programming languages are *integers*, *reals* and *double precision reals*. Let us take a brief look at how they are represented on the computer. In all of the following, we only look at *positive* numbers. Negative numbers are represented by the two’s complement of the corresponding positive number. If you don’t know what that means, don’t worry about it.

I assume you are familiar with representation of numbers in various bases. For example,

$$\begin{aligned}(123)_{10} &= 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0, \\ (1101)_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (13)_{10}.\end{aligned}$$

For fractional numbers, the same system works

$$\begin{aligned}(1.23)_{10} &= 1 \cdot 10^0 + 2 \cdot 10^{-1} + 3 \cdot 10^{-2}, \\ (11.01)_2 &= 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = (3.25)_{10}.\end{aligned}$$

Most machines use base 2 numbers internally. The HP 41C calculator uses base 10. I will use base 10 examples in class, since that is what everybody is used to, even though your computer really uses base 2.

Most machines set aside a fixed amount of storage per number. For the machines you are likely to use, it is 32 bits = 4 bytes per integer or real, 64 bits = 8 bytes per double precision number. (Wylbur is different, I think).

No matter which system you use, a machine number is stored in a finite space, so only finitely many different numbers can be represented, and the results of computations need to be rounded to fit into the available storage.

2.2.1. Integers. An integer is represented on the computer as a straight binary number of 31 bits, plus one sign bit (0 = positive, 1 = negative). Thus, the largest integer that can be represented is

$$(0111 \dots 111)_2 = 2^{31} - 1 = 2,147,483,647.$$

Since integers are usually used only as array indices and loop counters, this is not likely to cramp your style.

2.2.2. Reals. Early computers worked with *fixed point arithmetic*, where a certain number of places before and after the decimal point are used. This is still useful in some applications such as banking, where fractions of cents are not desired, but for scientific computation *floating point numbers* are used.

A number such as 123 can be written in scientific notation in many different ways:

$$123 = 1.23 \cdot 10^2 = 0.0123 \cdot 10^4 = 12,300 \cdot 10^{-2} = \dots$$

The power of 10 (or 2, or whatever base you are using) is called the *exponent*, the number in front is called the *mantissa*. To standardize things, we define a *normalized number* as one where the exponent has been

adjusted so that the mantissa has exactly one nonzero digit in front of the decimal point. In the above example, that is $1.23 \cdot 10^2$. Each number except zero has exactly one normalized form.

In base 2, the normalized form of the number $5/16 = (0.0101)_2$ is

$$5/16 = (1.01)_2 \cdot 2^{-2}.$$

Note that a normalized binary number always has the digit 1 before the binary point, since that is the only nonzero binary digit there is.

In the IEEE Standard, a real number takes up 32 bits, divided into 1 bit for the sign, 8 for the exponent and 23 for the mantissa. Instead of having a sign for the exponent part, a *bias* of 127 is added to the true exponent. Thus, the stored exponent is the true exponent + 127. The leading digit 1 of the mantissa is not stored, since it is always there. This is called the *hidden bit*.

As an example, the number $5/16$ is represented as

| Sign | Exponent | Mantissa |
|------|----------|--------------------------|
| 0 | 11111101 | 010000000000000000000000 |

The stored exponent is the number 125 in binary form (true exponent of (-2) plus bias 127). The 1. in front of the mantissa is hidden.

The IEEE standard provides for $\pm\infty$ and for NaN (not a number). The result of $1/0$ is $+\infty$, the result of $\sqrt{-1}$ is NaN. (Stored) exponent 255 is reserved for these special non-numbers.

Very small numbers can be represented as denormalized numbers, which have lower accuracy. (Stored) exponent 0 is reserved for that. We won't go into details of denormalized numbers.

Stored exponents therefore go from 1 to 254 and represent true exponents of -126 to 127.

The largest number is $(1.1111\dots)_2 \cdot 2^{254-127}$, which is about $3.4028 \cdot 10^{38}$. The smallest normalized number is $(1.000\dots)_2 \cdot 2^{1-127}$, which is about $1.175 \cdot 10^{-38}$. The accuracy is 24 bits (23 stored and 1 hidden), which corresponds to about seven decimals, since $2^{24} \approx 10^7$. Denormalized numbers can be smaller than 10^{-38} , but they also have less accuracy.

The Vax has no denormalized numbers and a slightly different range: the smallest and largest numbers are about $0.29 \cdot 10^{-38}$ and $1.7 \cdot 10^{38}$. The bias is 128, the accuracy is the same (24 bits \approx seven decimals).

What you should keep in mind from all this is:

For most computers, the range of representable single precision real numbers is approximately from 10^{-38} to 10^{38} . The accuracy is about seven decimals.

Among other things, this means that it is useless to print out numbers to more than seven decimals. Anything past the seventh digit is garbage. Note that your pocket calculator probably has 10 digits (or even 12 or 13 internally), so it is more accurate than a big computer.

2.2.3. Double Precision. If seven decimals are not enough accuracy, you can use double precision numbers. The idea is the same as for reals, but you use 64 bits instead of 32.

Many compilers accept the notation `real*4` for single precision reals, which take 4 bytes of storage each, and `real*8` for double precision numbers, which take 8 bytes of storage each.

Here the IEEE standard and the Vax differ considerably. In the IEEE standard, the exponent increases to 11 bits, the mantissa increases to 53 bits (52 stored + 1 hidden), which gives a range of about 10^{-308} to 10^{308} , and 15 decimals accuracy.

The Vax keeps the 8 bit exponent, which leaves 56 bits for the mantissa. Thus, the range of double precision numbers is the same as for reals (about 10^{-38} to 10^{38}), but the accuracy is 16 decimals.

The Vax also has a kind of double precision numbers similar to the IEEE standard with 11-bit exponent, but has to simulate them in software. This increases execution time by a factor of about 100 (no kidding!). The Vax also has quadruple precision, for those days when double precision just isn't enough. That is even slower. You are discouraged from using these kinds of numbers. If you really want to know more, get URM 146 from the computer center.

What is the penalty for using double precision? It takes twice as much storage and is slower. Runtime on the Vax increases by about 60 %. On a micro with a co-processor, I would assume only 10 – 20 % runtime increase, mostly for extra data moves. The co-processors work internally with more than double precision, anyway. Without a co-processor, my guess is that double precision would take at least 3 times as long as single precision. This is on top of the fact that single precision is already a factor of 20 slower than with a co-processor. (This number I determined experimentally).

For the type of small programs you will write in this course, there is no problem with using double precision. However, you should only use it if single precision fails.

The reason is this: For every numerical routine, no matter how well it is written, you can find some data that it cannot handle. It is part of the learning process to observe how far you can push a program before it fails. Making programs fail is much easier in single precision.

2.2.4. Rounding. Most numbers are represented exactly only by infinite decimal or binary expansions. In order to fit them into a 24-bit mantissa, we need to cut them off somehow. There are two ways of doing this:

- *Truncation* or *chopping*, where all bits past the 24rd are simply thrown away.
- *Rounding*, where we look at the 25th bit first. If it is 0, we truncate; if it is 1, we round up.

This is analogous to representing the number $2/3$ in three decimal digits. Truncated you get 0.666, rounded you get 0.667.

Rounding is normally preferred, since we expect only half as much error as truncation on the average. Also, we expect errors to cancel more, since sometimes we round up and sometimes down. (Truncation is the same as always rounding down).

2.3. Machine Constants

There are three constants that describe the floating point characteristics of a computer.

OFL: The *overflow limit*, which is the largest floating point number that can be represented. In single precision, it is near 10^{38} . If the result of a computation gets larger than OFL, and *overflow error* occurs. This is usually fatal to the program.

UFL: The *underflow limit*, which is the smallest positive normalized number that can be represented. In single precision, it is near 10^{-38} . If the result of a computation gets smaller than UFL, an *underflow error* occurs. Usually, the number is rounded to zero, and the program continues (with or without a message).

The machine epsilon ϵ : ϵ represents the accuracy of the machine, in the sense that the relative error from rounding a number to machine accuracy is of order ϵ . One definition is that ϵ is the smallest floating point number such that

$$1 + \epsilon > 1.$$

In single precision, ϵ is approximately 10^{-7} , since we have 7 decimals of accuracy.

The behavior of Vax and PV in response to overflow and underflow errors is explained in the Fortran write-up (appendix B).

2.4. Errors in Scientific Computing

In numerical analysis, errors are defined as follows. Assume that x is some number (for example the number π), and x^* is an approximation to x (for example the machine representation of π). Then

$$\begin{aligned} \text{absolute error} &= x - x^* \quad (\text{or } |x - x^*|) \\ \text{relative error} &= \frac{x - x^*}{x} \quad (\text{or } \left| \frac{x - x^*}{x} \right|) \end{aligned}$$

The first form is called *signed error*, since it can be positive or negative. Sometimes it is more convenient to use signed errors, sometimes it is more convenient to use absolute values. I will switch back and forth between the two. You may do the same, unless otherwise stated.

The signed error has the advantage that you know whether your result is too large or too small. The unsigned error has the advantage that it is often easier to handle.

Note that “absolute error” has nothing to do with absolute values.

Usually, the relative error is a better measure for accuracy than the absolute error. If your absolute error is 0.001, this may be very good (if the true answer is 10,000) or very bad (if the true answer is 0.00000001).

Sometimes the relative error can be misleading, usually when the true result is very small. For example, suppose the true result is 10^{-10} , and your numerical answer is 10^{-8} . The error is 100 times as large as the result, but maybe you only need to know that the value is very small. In these cases, the absolute error is a better measure of accuracy.

Most good scientific subroutines let you specify either absolute or relative error bounds, or a combination of both.

2.4.1. The Condition Number. The difficulty of a problem is measured by the *condition number*. It is defined as

$$\text{condition number} = \max \frac{\text{relative error in output}}{\text{relative error in input}}$$

The fraction is the factor by which (relative) input errors get magnified. The “max” means we consider the worst possible case. In any particular case, the error magnification may be less.

In practice, this means the following: suppose you know that a particular problem has a condition number of 1,000. You are computing in single precision. The input data must be assumed to carry a relative error of 10^{-7} (the machine ϵ). Therefore, the output data may carry a relative error of $1,000 \cdot 10^{-7} = 10^{-4}$. In other words: you can only trust 4 decimals in your answer. In a particular case, your error may be less, but you don’t know that. You have to assume the worst.

The exact value of the condition number is not that important, and is usually not known, anyway. What counts is the *order of magnitude*. A problem with a small condition number (tens or hundreds) is *well-posed*, a problem with a large condition number (millions) is *ill-posed*. A problem with a condition number above 10^7 cannot be solved at all in single precision.

There are really two condition numbers at work: the condition number of the problem, and the condition number of the algorithm. An *algorithm* is a particular sequence of steps the computer follows to get the answer.

The condition number of the problem is the error magnification you get when you do exact calculations. The condition number of a particular algorithm is the error magnification you get when you solve the problem on a computer, using this particular algorithm.

The condition number of any algorithm is always at least as large as the condition number of the problem. There are some problems that simply cannot be solved. For solvable problems, you have to watch out that your algorithm does not make things worse than they already are. Look at the examples in section 2.7 in these notes.

2.4.2. Sources of Errors. There are many sources of error in scientific computation. Some of them are listed here. Note that in practice, the distinction between the types of error is not always clearcut.

Arithmetic Error: This comes from rounding numbers to machine accuracy. Every input number, and every result of a calculation acquires a relative error of the order of the machine ϵ .

Discretization Error: Computers can only do computations that take a finite number of steps. Any computation dealing with a continuum needs to be broken into discrete steps. For example, to calculate $\int_a^b f(x) dx$ numerically, you have to sample the function at a finite number of points. This causes an error, because you are ignoring all the other points.

Truncation Error: The same as discretization error, except you cut down a countably infinite operation to a finite one. For example, you cannot add up an infinite series; you have to stop after a finite number of terms. The ignored terms lead to truncation error.

Data Error: Your data may consist of measurements of something and already carry a measurement error.

2.4.3. Arithmetic Errors. Here we consider the effect of roundoff errors. What are the condition numbers of basic operations (addition, multiplication, calculation of exp, sin, sqrt, etc.)?

Fact: *With one exception, all basic operations available in Fortran are well-behaved.* This means the condition number is near 1, so that the result of a calculation has approximately the same relative error as the input. The one exception is *cancellation*, which can occur in addition/subtraction.

Cancellation happens when you subtract two almost equal numbers. Suppose you work in seven digit decimal arithmetic. The number 0.1234567 stands for some number between 0.12345665 and 0.1234567499999 It has a relative error of order 10^{-7} .

Likewise, the number 0.1234576 stands for anything between 0.12345755 and 0.123457649999

If I subtract these numbers, I get 0.0000009. However, the true answer could be anywhere between

$$0.12345764999 \dots - 0.12345665 \approx 0.0000010$$

and

$$0.12345755 - 0.12345674999 \dots \approx 0.0000008,$$

so the relative error in the result could be up to 10 %, depending on what “true” numbers the machine numbers stand for. The original relative error has been multiplied a millionfold.

There is no point in worrying about cancellation every time you add or subtract two numbers. You just have to recognize it when you see it. Look at section 2.7 in these notes examples on how to avoid cancellation in some cases.

2.5. Extrapolation

Many algorithms are based on Taylor series expansions. The (absolute) error can be calculated from the series remainder and is of the form

$$\text{error} = c_p h^p + c_{p+1} h^{p+1} + c_{p+2} h^{p+2} + \dots$$

or similar, so

$$\text{error} = O(h^p).$$

Example: One method to approximate a derivative numerically is

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Taylor series expansion gives

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \dots \\ \frac{f(x+h) - f(x)}{h} &= f'(x) + \frac{h}{2} f''(x) + \frac{h^2}{3!} f'''(x) + \dots \\ \text{error} = \frac{f(x+h) - f(x)}{h} - f'(x) &= \frac{h}{2} f''(x) + \frac{h^2}{3!} f'''(x) + \dots, \end{aligned}$$

so that

$$\text{error} = c_1 h + c_2 h^2 + \dots = O(h).$$

A better method is

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Taylor series expansion gives

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \dots \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \dots \\ \frac{f(x+h) - f(x-h)}{2h} &= f'(x) + \frac{h^2}{3!} f'''(x) + \frac{h^4}{5!} f^{(5)}(x) + \dots \\ \text{error} = \frac{f(x+h) - f(x-h)}{2h} - f'(x) &= \frac{h^2}{3!} f'''(x) + \frac{h^4}{5!} f^{(5)}(x) + \dots, \end{aligned}$$

so that

$$\text{error} = c_2 h^2 + c_4 h^4 + \dots = O(h^2).$$

□

If the order of the error is known, we can eliminate the leading error term. This is called *extrapolation*. Let $A(h)$ be the approximate result calculated with stepsize h . Assume

$$\text{error} = O(h^p) \approx ch^p.$$

Then

$$\begin{aligned} A(h) &\approx \text{true result} + ch^p, \\ A(h/2) &\approx \text{true result} + c(h/2)^p, \end{aligned}$$

so

$$A(h) - A(h/2) \approx ch^p [1 - (1/2)^p],$$

$$\text{true result} \approx A(h) - ch^p = A(h) - \frac{A(h) - A(h/2)}{1 - (1/2)^p},$$

$$\text{true result} \approx \frac{2^p A(h/2) - A(h)}{2^p - 1}$$

After extrapolation, the next term in the error series takes over.

Example: Calculate $f'(x)$ numerically for $f(x) = e^x$, $x = 1$. Using the first formula, we know the error is $O(h)$, so $p = 1$.

I got the following values on my calculator:

| h | $\frac{f(x+h) - f(x)}{h}$ | extrapolated values |
|--------|---------------------------|------------------------------|
| 0.1 | 2.858841955 | 2.715929629 |
| 0.05 | 2.787385792 | 2.718296492 |
| 0.025 | 2.752545284 | 2.717704776 2.718281791 |
| 0.0125 | 2.735342100 | 2.718138916 |

| h | error | error of extrapolated values |
|--------|-------------|-------------------------------|
| 0.1 | 0.140560126 | -0.002352199 |
| 0.05 | 0.069103964 | 0.000014663 -0.000000037 |
| 0.025 | 0.034263456 | -0.000577052 0.000001801 |
| 0.0125 | 0.017060272 | -0.000142912 |

The first column of extrapolated values is calculated from the derivate estimates by

$$\text{true result} \approx \frac{2A(h/2) - A(h)}{1}.$$

The more accurate answer $A(h/2)$ is the value from the row below, the less accurate answer $A(h)$ is the value from the row above. After the leading error term c_1h has been eliminated, the leading error term is now c_2h^2 (see above). Thus, the extrapolated values have error $O(h^2)$.

We can extrapolate them again, this time with

$$\text{true result} \approx \frac{4A(h/2) - A(h)}{3},$$

to get a new column with error $O(h^3)$, and for the last value with

$$\text{true result} \approx \frac{8A(h/2) - A(h)}{7}.$$

If you look at the corresponding errors, you can verify that the error in each column behaves as it should: In the column where the error is $O(h)$, each error value is approximately 1/2 of the one before. In the column where the error is $O(h^2)$, each error value is approximately 1/4 of the one before, etc. \square

Example: Repeat the previous example, using the better derivative formula. This time, the original error is $O(h^2)$, and subsequent columns have error $O(h^4)$, $O(h^6)$, etc.

| h | $\frac{f(x+h) - f(x-h)}{2h}$ | extrapolated values | | |
|--------|------------------------------|---------------------|-------------|-------------|
| 0.1 | 2.722814564 | 2.718281261 | | |
| 0.05 | 2.719414587 | 2.718281794 | 2.718281830 | 2.718281829 |
| 0.025 | 2.718564992 | 2.718281829 | | |
| 0.0125 | 2.718352618 | 2.718281827 | | |

| h | error | error of extrapolated values | | |
|--------|-------------|------------------------------|--------------|--|
| 0.1 | 0.004532735 | -0.000000567 | | |
| 0.05 | 0.001132759 | 0.000000001 | 0.000000001 | |
| 0.025 | 0.000283163 | 0.000000001 | -0.000000002 | |
| 0.0125 | 0.000079789 | | | |

Again, compare the error columns to the predicted behavior. \square

2.6. Historical Perspective

Read this on your own.

2.7. Examples

Let us look at some examples of the principles mentioned in this chapter.

2.7.1. Avoiding Ill-conditioned Algorithms. As mentioned above, some problems are ill-conditioned, and no clever algorithm will save you. All you can do is to avoid ill-conditioned algorithms for well-conditioned problems.

To determine whether your problem is ill-conditioned or not, ask yourself the question: if I change the data just a little bit, how will that affect the result?

For example, in numerical integration a small change in the function will produce a small change in the area under the curve: numerical integration is well-conditioned. In numerical differentiation, a small change in the function can produce a large change in the result (see example 2.2 in NMS): numerical differentiation is ill-conditioned.

If you get bad results for a well-conditioned problem, there is usually cancellation in your algorithm somewhere. All algorithms break down into basic additions, multiplications, etc. at the lowest level. Unless you are doing billions and billions of these, only cancellation would produce a noticeable error in the result.

Sometimes, the errors can be avoided by some simple rearrangement of the algorithm. Details depend on the individual case.

Example: Find $f(x) = 1 - \cos(x)$ for $x = 10^{-5}$ with eight correct decimals on your hand calculator. If you simply punch it out, you get zero. The reason is cancellation: $\cos(x) \approx 1$ for small x . But

$$\begin{aligned} \cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - + \dots, \\ f(x) &= \frac{x^2}{2!} - \frac{x^4}{4!} + - \dots, \\ f(10^{-5}) &= \frac{1}{2} \cdot 10^{-10} - \frac{1}{24} \cdot 10^{-20} + - \dots \\ &= 5 \cdot 10^{-11} \end{aligned}$$

to 10 places of accuracy. \square

Example: Calculate $f(x) = \sqrt{x+1} - \sqrt{x}$. Again, you get zero for large x . With a little algebra

$$\begin{aligned} f(x) &= (\sqrt{x+1} - \sqrt{x}) \cdot \frac{\sqrt{x+1} + \sqrt{x}}{\sqrt{x+1} + \sqrt{x}} \\ &= \frac{1}{\sqrt{x+1} + \sqrt{x}} \end{aligned}$$

and you can calculate this very accurately. \square

Example: See example 2.7 in NMS (power series of e^x for large negative x). \square

2.7.2. Interaction of Errors. It happens frequently that the total error is a sum of two or more errors which depend on some parameter h . When h changes, one of the errors decreases and another one increases. The result is that there is a certain minimum error that you cannot get below.

This happens especially in ill-posed problems; a standard example is numerical differentiation (examples 2.2, 2.8 in NMS). The following example is basically example 2.8 in NMS. I am just repeating it here for completeness.

Example: We approximate the derivative of f by the formula

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

There is a discretization error, due to replacing the derivative by a difference quotient. From the Taylor series expansion, the (absolute) discretization error is

$$\text{discretization error} = f'(x) - \frac{f(x+h) - f(x)}{h} \approx \frac{1}{2}f''(\xi)h \approx \frac{1}{2}f''(x)h,$$

since ξ must be very close to x .

There is also arithmetic error, since $f(x+h)$ and $f(x)$ are close together for small h . This cancellation error can be estimated as follows: Assume the relative error in $f(x+h)$, $f(x)$ is each about ϵ , so the absolute error in each case is about $f(x) \cdot \epsilon$. (This estimate is good for both, since $f(x+h) \approx f(x)$). If one error is positive, the other negative, they could add up to

$$\text{absolute error} \approx 2f(x)\epsilon/h.$$

The total error is therefore about

$$\frac{f''(x)h}{2} + \frac{2f(x)\epsilon}{h}.$$

Using calculus, we find that the minimum occurs when

$$\text{optimal } h = 2\sqrt{\frac{\epsilon f(x)}{f''(x)}},$$

and the minimal total error is about

$$\text{minimum total error} = 2\sqrt{\epsilon f(x)f''(x)}.$$

We can do this much faster by using some rules of thumb. We have no idea what $f(x)$, $f''(x)$ are in general, anyway, so we set all constants except h and ϵ to 1. The minimal total error is approximately achieved when the discretization and cancellation errors are the same. This leads to

$$\begin{aligned} \text{discretization error} &\approx h, \\ \text{arithmetic error} &\approx \frac{\epsilon}{h}. \end{aligned}$$

The total error is minimal when $h \approx \epsilon/h$, or $h \approx \sqrt{\epsilon}$. For seven digits of accuracy, this means h should be near 10^{-3} or 10^{-4} .

Note that the minimal total error is also of order $\sqrt{\epsilon}$. This means that you cannot calculate a numerical derivative to more than three or four places in single precision.

In example 2.2 in NMS, the actual value for the optimal h was more like 10^{-5} , but the above is just a crude estimate. Theory and practice are closer in example 2.8, which was done on a Cyber. The Cyber has 60-bit numbers, so $\epsilon \approx 10^{-14}$, $h \approx 10^{-7}$. \square