Computer Numbers and their Precision, I Number Storage

Learning goal: To understand that how computers store numbers leads to limited precision and how limited precision introduces errors into calculations.

Learning objective

Computational and mathematical objectives:

- To understand that exact or whole numbers can be stored as integers.
- To understand that the division of two integers is always rounded down to a smaller integer.
- To understand that numbers can also be stored in scientific or engineering notation (floating point numbers).
- To understand the advantages and disadvantages of floating point numbers.
- To understand that computer errors occur when an integer becomes too large as a positive or negative number.
- To understand that computer errors known as *overflow* occur when the exponent of a floating point number becomes too large.
- To understand that computer errors known as *underflow* occur when the exponent of a floating point number becomes too large in the negative.
- To understand that truncation or roundoff occurs when the mantissa of a floating point number becomes too long.
- To understand some of the consequences of the roundoff of floating point numbers.

Science model/computation objectives:

- To understand computer storage of floating point numbers, and be able to empirically determine *machine precision*.
- To understand that just as laboratory experiments always have limits on the precision of their measurements, so too do computer simulations have limits on the precision of their numbers.
- To understand the difference between “precision” and “accuracy”.
- Students will practice the following scientific skills:
    - Doing numerical experiments on the computer.

Activities

In this lesson, students will:

- Perform calculations on the computer as experiments to determine the computer’s limits in regard to the storage of numbers.
- Perform numerical calculations to see how the rules of mathematics are implemented.
- Sum the series for the sine function to see the effect of error accumulation.

Where's Computational Scientific Thinking and Intuition Development

- Understanding that computers are finite and therefore have limits that affect calculations.
- Being cognizant that uncertainties in numerical calculations are unavoidable.
- Understanding how to obtain meaningful results even with a computer's limited precision.
- Understanding the range of numbers that may be necessary to describe a natural phenomenon.

Background

It is expected students understand some basic aspects of numbers and their properties. In particular, it will help to be familiar with real number systems, integers, rational numbers and irrational numbers, as well as the scientific representation of numbers. A review of these aspects can be found at [UNChem], where there is short Mathematics Review as well as some a useful Calculator Review.

In colloquial usage, the terms *accurate* and *precise* are often used interchangeably. In computational science and engineering, *precision* usually indicates the number of significant figures or meaningful decimal places there are in a measurement or a calculation. In contrast, *accuracy* is more of a measure of how close a measurement of some quantity is to its true value. Likewise, accuracy sometimes refers to how close a theoretical description or model of a phenomena is to what might actually be happening in nature.

You may think of precision as relating to the quality or capability of an "instrument", but not necessarily as to whether the measured value is close to the true value. For example, we may have a watch that always reads exactly 11:00.00 AM when the sun is at its zenith, and does this year after year. Since there are approximately $\pi \times 10^7$ seconds in a year, this means that the watch has a precision of better than one part in a million, which is quite precise as it is quite repeatable. However, since the true time should be noon, the watch cannot be said to be an accurate measure of "the time", although it would be accurate for measuring time differences.

Computer Number Representations (Theory)

Regardless of how powerful a computer may be, it is still finite. This means that the computer takes a finite time to complete an operation and that it has only a finite amount of space in which to store things . Of course as computers and communication networks get faster, and as the available storage gets to be very large, a computer's response may seem to be instantaneous and its memory may seem to be endless, but if you try to do something like predict the weather from first principles or make a copy of a high-definition DVD, the computer's finiteness becomes evident.

In this module we explore one aspect of the finiteness of computers, namely, how they store numbers. And because the basic scheme is used on computers of all sizes, the limitations related to this storage is universal. Knowing the limits of the instrument is an essential element in learning to think computationally and scientifically.

Bit. The ***problem*** faced by computer designers is how to represent an arbitrary number using a finite amount of memory. Most computer memories contain magnetic materials in which elementary magnets are aligned to face either up or down. As a consequence, the most elementary units of computer memory are the two *binary integers* 0 and 1 that form the basis of the binary number system (representing all numbers with just two symbols). The phrase "binary integer" is usually abbreviated as the word *bit*. This in turn means that, ultimately, everything stored in a computer memory is just a long strings of zeros and ones, with each bit representing the number of times a power of 2 is to be added in. For example, the numberl 10 is the binary 1010, that is, $10_{\text{decimal}} = 1010_{\text{binary}}$.

Optional: We will leave discussion of the binary representation of numbers to a math class for now, but for the curious:

(1) $1010_{\text{binary}} = 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0,$ where

(2) $2^3 = 8, \quad 2^2 = 4, \quad 2^1 = 2, \quad 2^0 = 1,$ and

(3) thus, $1010_{\text{binary}} = 8 + 0 + 2 + 0$

(4) $10_{\text{decimal}} = 1\times10^1 + 0\times10^0,$ where

(5) $10^1 = 10, \quad 10^0 = 1.$

(6) thus, $10_{\text{decimal}} = 10 + 0$

As a consequence of this scheme, N bits can store positive integers in the range 0 - 2^N.

While it is no longer essential that you know how to convert numbers between different representations in order to compute (we could tell you war stories about that, but maybe another time), it is essential to remember that numbers on a computer are stored using a finite number of bits, and that unless the number is a power of 2, the process of converting it to a decimal number introduces a small uncertainty. In computing, an uncertainty is often called an "*error*", although this does not imply that anyone or any computer has done anything wrong.

Byte. The number of bits that a computer or its operating system uses to store a number is called the *word length.* For example, you might have heard about *32 bit* and *64 bit* versions of the Windows operating systems. Well, these just refer to the length of the words that the computer processes. While we are talking about these sorts of things, we should mention that often word lengths or memory sizes are expressed in *bytes* (a mouthful of bits), where

(5) 1 byte = 8 bits = B.

Conveniently, 1 byte is the amount of memory needed to store a single letter like ``a''. This adds up to a typical printed page requiring approximately 3000 B = 3 KB of storage. Of course with memory sizes continuing to grow, rather than bytes, you hear of kilobytes, megabytes or gigabytes, which mean one thousand, one million or one billion bytes (giga is used for billion since in Europe a billion means a million millions rather than our thousand millions). [If you want to be really technical, the kilobyte of computer memory is really 2^{10} = 1024 bytes, but you do not have to worry about that.]

Integer Storage

In a mathematical sense, integers are real numbers (no imaginary parts) that do not contain a fractional or decimal part. They can be positive or negative and comprise the set {-∞, ..., −2, −1, 0, 1, 2, ... ∞}. They are stored exactly, that is, with no fractional part. Because a finite number of bits used to represent integers, only a finite number of the integers are stored, not the entire range from -∞ to +∞. Just what range they do cover, and what happens when you try to go beyond that range is something we shall leave for the exercises.

Both hardware and software store numbers. If you try to store a number larger than the hardware or software was designed for, what is called an *overflow* results. Likewise, if you try to store a number smaller than the hardware or software was designed for, what is called an *underflow* results. As we shall see, if either an under- or overflow occur in a computation with integers, it is likely that your results are not meaningful ("garbage"). Using 64 bits permits integers in the range $-10^{19} - 10^{19}$. Integers in Python cover the range:

(6) $$-2,147,483,648 \leq \text{Integers} \leq +2,147,483,647$$

While at first this may seem like a large range (4,294,967,294 is an order of 10^9), it really is not when compared to the range of sizes encountered in the physical world. As a case in point, the ratio of the size of the universe to the size of a proton is approximately $\pm 10^{41}$, which is way beyond what you can do with integers.

Order of Operations

Before we run headfirst into actually doing calculations on the computer, it is good to remind you that with multiple operations computers and people may be very powerful, but they still cannot figure out what you are thinking! So even though you may know what you want when you write

(7) $$6 + 3 \times 2 = ?,$$

we may well find this ambiguous. Should the operations be read from left to right, as is done in English, or from right to left, as in done in Hebrew and Arabic? The general rule for order of operations is

(8) $$Please\ Excuse\ My\ Dear\ Aunt\ Sally!$$

that is, Parentheses, Exponentiation, Multiplication or Division, Addition or Subtraction. This means first do any operation in parentheses, then do any exponentiation, etc. Accordingly the operation in equation 7 is equivalent to

(9) $$6 + (3 \times 2) = 6 + 6 = 12.$$

Exercise: Aunt Sally

1) Use your handy calculator, and then computer, to evaluate

i) $6 + 3 \times 2$
ii) $6 + (3 \times 2)$
iii) $(6 + 3) \times 2$
iv) $6 \div 3 + 2$
v) $6 \div (3 + 2)$
vi) $6 + 2 \div 3$

Integer Arithmetic

You can add, subtract and multiply integers on a computer and get just the results you would expect. In contrast, when you *divide* two integers and a remainder results, a decision must be made about what to do with the fractional part since integers do not have fractional parts. Most computer programs just throw away the fractional part, or in other words, the result is always rounded *down*, even if the fraction is greater than 0.5.

For example, $6_c \div 2_c = 3_c$, but $3_c \div 2_c = 1_c$, where we use the subscript c to indicate that this is computer math.

Exercise: Integer Arithmetic

Now try some of these exercises on your computer in excel or python to see the effect of Integer arithmetic

1. $6 \times 2 = ?$
2. $6 \div 2 = ?$
3. $3 \div 2 = ?$
4. $8 \div 3 = ?$
5. $2 \times (6 \div 2) = ?$
6. $2 \times 6 \div 2 = ?$
7. $2 \times (3 \div 2) = ?$
8. $12 \div 2 \div 3 = ?$
9. $12 \div (2 \div 3) = ?$
10. $(12 \div 2) \div 3 = ?$
11. $12 \div (3 \div 2) = ?$

Note that the last 6 exercises are a little tricky in that they involve two arithmetic operations, with the final result possibly depending upon the order in which the operations are preformed. In exercise 5, the parenthesis makes it clear to the computer and you that you want 6 to be divided by 2 *before the multiplication occurs.* Exercise 6 should produce the same result since division is usually performed *before* multiplication, but it is always worth checking. In any case, it is because expressions like those in exercises 6 and 8 are ambiguous, we *recommend using parenthesis to eliminate possible ambiguities.* By the way, in exercise 8, divisions are performed from left to right and so the answer should be 2. However, when the integer division in 9 within the parenthesis is carried out, the result is 0, and as you have seen, computers do not permit division by zero.

Exercise: Determining Range of Integers

It is possible to determine experimentally the range of positive and negative integers on your computer system and for your computer language/software. You can do this by doing successive multiplications by 2 until the results change abruptly, which is your signal that something “not-quite-right” has happened. Write a python program, or use excel, to find when you reach this behavior.

A sample pseudocode is

```
max = 1
min = -1.
do 1 < i < N
        max = max *2.
        min = min * 2.
        write i, max, min
end do
```

Hint: we have indicated above that the integers in Python lies between $\mp 10^{19}$, which is equivalent to approximately $\mp 2^{64}$.

Floating-Point Numbers

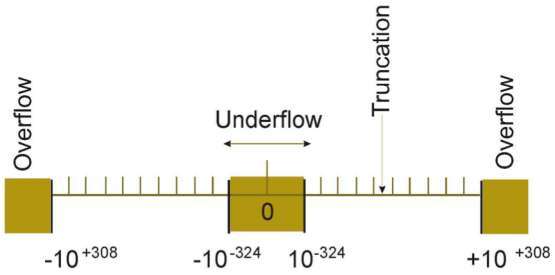

Figure 1 The approximate limits (-10^308, -10^324, 10^324, 10^308; black lines) of double-precision (64 bit) floating-point numbers and the consequences (overflow, underflow) of exceeding these limits. The hash marks abstractly represent the number values that can be stored ; storing a number in between these values leads to round-off error. The shaded areas correspond to over- and underflow.

Because **integers** are represented exactly on a computer, there is no loss of information when using integers within the range that computer can store. Yet we need another way to represent these numbers when the range of integers is greater than the range stored by the computer or when numbers cannot be represented as an integer. (Important numbers like π have fractional parts that cannot be represented as an integer or even as the ratio of integers (rational numbers).) So we also need a more powerful way to represent numbers on a computer. In practice, most scientific computations use the *floating point representation* of numbers, which otherwise is commonly known as *scientific* or *engineering notation*. For example, the speed of light c= + 2.99792458 $\times 10^8$ m/s in scientific notation or +0.299792458 $\times 10^9$ in engineering notation. In each of these cases, the number in front is called the mantissa and contains nine *significant figures*.

The power to which 10 is raised is called the *exponent*, with the plus sign included as a reminder that these numbers may be negative:

$$Floating\ Point\ Number = \ mantissa^{exponent}$$

Often on computers the power is indicated by the letter E meaning exponent; for example, c= + 2.99792458 E08 or 0.299792458 E09. Indeed, deciding whether a number should be represented as an integer or a float is an important aspect of "thinking computationally".

Remember a fixed number of bits is used to represent the number, thus the precision possible and the range of numbers are limited. When 32 bits are used to store a floating-point number, we have what is called *single precision*, or *singles,* or just *floats*. When 64 bits are used to store a floating-point number, we have what is called *double precision*, or *doubles.* Most scientific computations require and use double precision (64 bits).

Because only a finite number of bits are used to store the *mantissa* (the part in front), some numbers like π will have some of their digits furthest from the decimal point (their least significant parts) *truncated*, and thus will not be stored exactly. In Figure 1 we represent those numbers that can be stored exactly by the vertical hash marks. If the floating point number you try to store on the computer falls between two hash marks, some of the least significant digits are truncated and the number gets stored at the nearest has mark. For example, if you tried to store 1/3 = 0.33333333333… as a single precision float with 7 digit mantissa, it would be stored as 0.3333333, with nothing beyond the last 3.

We have just described how truncation occurs when you try to store a number whose mantissa has more digits than the computer allows. Well, there is also a fixed number of digits allocated to store the exponent of a floating point number, and if you try to store more than that, something untoward happens. If your exponent is positive and is too big, an error condition known as *overflow* occurs. This usually causes your calculation to stop in its tracks, which is a good thing because otherwise the computer would just be producing garbage. If your exponent is a negative number with too large a magnitude (*i.e.* the number is too small), an error condition known as *underflow* occurs. In this case the computer usually sets the full number to zero and continues with the calculation. This may result in some loss of precision, but probably will not produce garbage.

The Institute of Electrical and Electronics Engineers (IEEE) has actually set standards for how computers and computer programs deal with floating point numbers. This means that running the same program on different computers should result in the same answers, even though the hardware and software may differ. As we shall define more carefully soon, the result may actually differ somewhat, but that difference would be smaller than the known precision of the floating point numbers being used. In addition, part of the IEEE standard calls for the symbol "INF" (for infinity) to occur if an overflow occurs, and "NAN" (for not a number) to occur if you try to divide by zero. These symbols cannot be used in any further calculations, but are just given to tell you what the problem is.

The actual limits for single and double precision numbers differ, with the increased length of double precision words leading to a greater range and an increased precision. Specifically**, *single-precision* (32-bit) *numbers have mantissas with 6.5 decimal places*** (not an exact number due to binary to decimal conversion), and exponents that limit their range to

$$1.4 \times 10^{-45} \leq \text{Single Precision Numbers} \leq 3.4 \times 10^{+38}$$

***Double precision* (64 bit*) numbers have mantissas with 16 decimal places*,** and exponents that limit their range to

$$4.9 \times 10^{-324} \leq \text{Double Precision Numbers} \leq 1.8 \times 10^{+308}$$

But, do not believe everything we tell you. We will ask you to verify these limits for yourself.

To repeat, *large scientific calculations almost always require at least 64-bit (double-precision) floats*.

Python and the IEEE 754 Standard

Python is a relatively recent language with changes and extensions occurring as its use spreads and as its features mature. It should be no surprise then that Python does not at present adhere to all aspects of the IEEE standard. Probably the most relevant difference for us is that *Python does not support single (32 bit) precision floating point numbers*. So when we deal with a data type called a *float* in Python, it is the equivalent of a *double* in the IEEE standard. Since singles are inadequate for most scientific computing, this is not a loss. However be wary, if you switch over to Java or C in the future you will need to learn to declare variables in a different way.

Exercise: Over and Underflows

1. Write a program, or sit at a terminal and do this by hand, and test for the **underflow** and **overflow** limits (within a factor of 2) of your computer system. (The simple Python program *Limits.py* can be modified for this purpose.) You can do this by successive multiplication and divisions by 2 until the computer tells you that something is wrong. A sample pseudocode is

```
under = 1.
over = 1.
do 1 < i <N
        under = under/2.
        over = over * 2.
        write out: i, under, over
end do
```

You may need to increase N if your initial choice does not lead to underflow and overflow.

2. If your computer system lets you specify word length, check where under- and overflow occur for single-precision floating-point numbers (floats). Give your answer in decimal.
3. Check where under- and overflow occur for double-precision floating-point numbers (floats in Python).

Machine Precision (Model)

A major concern of computational scientists is that the floating-point representation used to store numbers is of limited precision. In general, *single-precision numbers are precise to 6-7 decimal places, while doubles are precise to 15-16 places*. To see how limited precision affects calculations, consider the simple computer addition of two single-precision words with 6 place precision:

$$7 + 1.0 \times 10^{-7} = ?$$

Because there is no room left to store any digit beyond the sixth, they are lost, and after all this hard work the addition just gives 7 as the answer (truncation error in Figure 1). In other words, because a single precision number stores only 6 decimal places, it effectively ignores any changes beyond the sixth decimal place.

As you may have realized already, a number stored on a computer may not equal the mathematical value of that number. For example, mathematically the decimal representation of 2/3 has an infinite number of digits, 2/3 = .6666666666…. . In contrast, the computer representation is finite, for example, $2/3_c$ = 0.666667, where we use the subscript c to indicate the computer representation. In other words, except for powers of 2 that are store exactly, we should assume that all single-precision numbers contain an error in the sixth decimal place and that all doubles have an error in the fifteenth or sixteenth place.

The preceding loss of precision is categorized by defining a number known as the *machine precision* ϵ_m. Although this may sound like a contradiction at first, ϵ_m is the *largest* positive number in a computer computation that can be added to the number stored as 1 without changing the value of that stored 1:

$$1_c + \epsilon_m = 1_c \, .$$

Here the subscript c is a reminder that this is a computer representation of 1. Note that the machine precision is related to the number of decimal places used to store the mantissa of a floating-point number, specifically, $10^{-\,number\ of\ places}$. Because we have told you several times already that the number of places for single and double precision numbers, you can predict a value for the machine precision by a simple substitution. However, as we ask you to do next, we prefer that you use your computer as a laboratory and experiment to determine its machine precision.

Exercise: Determine Your Computer's Machine Precision

We want you to either write a program or just enter numbers on your computer ~~terminal~~ that will permit you to determine the machine precision ϵ_m of your computer system. You do not have to be really precise, so getting the answer within a factor of 2 will be just fine. The basic idea is very simple: start with an arbitrary value of ϵ_m, say $\epsilon_m = 1$, and add that to 1.0. If the answer is not 1.0, then make ϵ_m smaller and again add it to 1.0. Keep repeating the process until you get 1.0 as an answer. The largest value of ϵ_m for which this happens is your machine precision.

Implementation

Note, if you are entering the values of ϵ_m manually you may want to divide by 10 each time until you get a gross answer, and then do some fine tuning. If you are writing a program, you can divide by 2 and just let the computer churn away for a long time. Here is a sample pseudo code (the basic elements of a computer program not specific to any language):

eps = 1.

```
do N times
        eps = eps/2.                    # Make smaller
        one = 1. + eps
        output one, eps         # Write loop number, one, eps
end do
```

Python Implementation

A model Python program than needs to be extended to actually determine the precision is *Limits.py:*

```
# Limits.py: Increase N to determine approximate machine precision

N = 3
eps = 1.0
for  I  in range(N):
        eps = eps/2
        one = 1.0  +  eps
        print('eps = ', eps,  'one = ', one)
print("Enter and return a character to finish")
s=raw_input()
```

Excel: Excel Implementation. This workbook has three worksheets – one for each exercise in this module.

Summary and Conclusions

You must know something about how computers store numbers in order to judge how reliable the results are. Computed numbers are never exact unless you are carrying out symbolic manipulations (mathematics that does not require specific numbers and not discussed here). But numbers can still be used to carry out calculations within a stated level of precision. Specifically, floating point numbers can give up to 16 places of precision. Integers can be exact, but have a much more limited range of values. Integers are thus appropriate for counting items such as loops in a program.

Where's Computational Scientific Thinking

- Understanding that computers are finite and therefore have limits.
- Being cognizant that uncertainties in numerical calculations are unavoidable.
- Understanding how it is possible to work within the limits of a computer to obtain meaningful results.
- Understanding that the range of numbers that may be necessary to describe a natural phenomenon may not be stored properly in a given computer or computer language.

References

[CP] Landau, R.H., M.J. Paez and C.C. Bordeianu, (2008), *A Survey of Computational Physics,* Chapter 5, Princeton Univ. Press, Princeton.

[UNChem] *UNC-Chapel Hill Chemistry Fundamentals Program, Mathematics Review,* http://www.shodor.org/unchem/math/; *Using Your Calculator,* http://www.shodor.org/unchem/math/calc/.