Introduction to Scientific Computing

Leigh J. Little SUNY Brockport August 28, 2022

Contents

1 Scientific Computing **1**

2 Computer Languages **1**

3 Types of Computer Data **2**
3.1 Number Bases . . . 2
3.1.1 Binary/Hexadecimal Conversion . . . 3
3.2 Integers . . . 3
3.3 Floating Point Numbers . . . 3
3.3.1 Single Precision . . . 3
3.3.2 Double precision . . . 4
3.4 The Special Values Nan and Inf . . . 4
3.5 Character Data . . . 4
3.6 A Short Note on Floating Point Precision . . . 4
3.6.1 Types of Floating Point Errors . . . 5
3.6.2 Why Use Double Precision? . . . 5

1 Scientific Computing

The phrase *scientific computing* has taken on numerous meanings since the first computers were built. Initially it meant using a computer to solve math problems. As computer technology has evolved, the meaning has become more refined. There are many ways to use a computer to solve a math problem (this includes hand-calculation with the aid of a TI calculator). Scientific computing now encompasses

1) Using a symbolic manipulator like Mathematica or Maple to compute exact, algebraic solutions to mathematical equations (when feasible).

2) Computing sufficiently accurate approximate solutions to mathematical equations. This includes

   – Programming a solution technique from scratch using a fundamental language like C.

   – Using a software package that is designed to solve certain types of problems. For example,

      * *Spice* - This is a package that will solve electrical circuits.
      * *Comsol* - This is a package that will solve partial differential equations.

   – Using something in between these two. This means a software tool that will solve many types of basic equations but still has ability to be programmed for more general purposes. Matlab and Python fall in to this category.

As it turns out, only a small fraction of real-world problems have solutions that can be written out in an exact form. This is because many problems rely on the solution of an ordinary differential or partial differential equation. Therefore, in this course we will focus on the approach in Item 2 above. We will be interested in computing approximate solutions to problems that arise in the physical sciences.

2 Computer Languages

This section is not designed to be an exhaustive discussion on languages; rather it is meant to illustrate how programming has evolved from its early life into what we use today.

A hundred years ago, a computer was a person. Their job was to perform elaborate hand calculations. The first modern computer arose from necessity in World War II. Paper copies of guidance tables were critical for field artillery units because they would indicate how the azimuth and elevation angles of a field cannon should be set in order to hit an enemy target.

It was soon discovered that these tables were not very accurate. Better tables were needed however, their computation depended on the solution to a complex system of ordinary differential equations. This system does not have an analytic (*i.e.*, paper and pencil) solution, so numerical methods were used to determine an approximate solution. These numerical methods are conceptually simple, but required a great deal of hand calculation. The modern computer was engineered to perform these calculations more rapidly. This lead to the development of the ENIAC (Electronic Numerical Integrator and Computer) computer in 1945. Other computers (UNIVAC, Whirlwind, Witch) followed soon after.

A common characteristic of early computers is that programs were written by physically rewiring the vacuum tubes that implemented what we now call digital logic. This presented several problems. Vacuum tubes are very sensitive devices and do not like to be jiggled around. This lead to many failures of the tubes. In addition different computers required different wiring to run the same program. Finally, the computer could only be wired for a single program at any one time. Rewiring the computer to run a different program was very time consuming.

This lead to the notion of programming languages. The idea is that a program could be written in a generic language that could be interpreted by any computer. This meant that the wiring could be permanent and the computer could rapidly switch between programs. Finally, the same program could be run on different computers.

The first modern language was called Short Code and it was developed by John von Neumann. This language was not easy to use, but it was a start. The first language that was widely used was Fortran, developed by John Backus at IBM. Fortran is still in use today because it is easy to understand and is probably the best language for high performance number crunching. Unfortunately, it is not a good language for anything else (graphics, operating systems, device drivers). COBOL, developed by Grace Hopper, was another language that emerged around this time and is also still used in business applications.

Over the years more than 100 programming languages have been developed. Some of these have fallen in to disuse and some are mainly theoretical. Table 1 below summarizes some features of common programming languages. The Out of Box Capability column refers to how many features beyond a minimal programming framework the language provides. The Flexibility column refers to the complexity of programs that can ultimately be written.

Name	**Out of Box Capability**	**Difficulty**	**Flexibility**	**Speed**
Assembly	Almost None	Insane	Complete	Ludicrous
C/C++/FORTRAN	Very Little	Medium	Almost Complete	Great
Java/Javascript/php Matlab/Python/Perl	Better	Medium	Almost Complete	Good
COBOL	Limited	Medium	Limited	Good

Table 1: Summary of features of common programming languages.

3 Types of Computer Data

Throughout the semester, we will encounter various types of data and it is helpful to be familiar with their names. The section below covers only the most common types of data.

3.1 Number Bases

The most common base used to represent computer data is binary (base 2). This base uses only the digits 0 and 1. A *bit* (the short form of *binary digit*) is a single 0 or 1.

Some computers use a decimal (base 10) base. This is most often seen in hand calculators and older cash registers. Older IBM mainframe computers use hexadecimal (base 16) as their base.

Nearly all computers follow the IEEE 754 standard for internal representation of and calculation with

computer numbers (the notable exception is Cray supercomputers). We will only touch on the main features of this standard.

3.1.1 Binary/Hexadecimal Conversion

Numbers are stored in computer memory in binary form. However, large sequences of binary digits are extremely difficult to interpret. Representing a binary number in hexadecimal form allows the same number to be written in a more comprehensible way. Table 2 contains the decimal, binary and hexadecimal representations of the first 16 integers. The binary representations are typically written in 4 digit form to allow

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Table 2: Decimal, binary and hexadecimal representations of the first 16 integers. Hexadecimal requires 16 digits, so it is necessary to use letters A through F for the last 6 digits.

for easier expansion and contraction between numbers in binary and hexadecimal. For example, the binary number

```
0101111101101000101011010100100100100000000111001101110111100001
```

can be written in hexadecimal as

```
5F68AD49201CDDE1
```

To see this, take the digits of the binary number in groups of four and use the information in Table 2 to convert to the appropriate hexadecimal digit.

```
0101 1111 0110 1000 1010 1101 0100 1001 0010 0000 0001 1100 1101 1101 1110 0001
   5    F    6    8    A    D    4    9    2    0    1    C    D    D    E    1
```

3.2 Integers

Integers are numbers without a decimal part (for example, 0, 5, -6, *etc.*). The most common use of integers is for counting and indexing. The basic integer type is represented in computer memory using 32 bits of precision. One of these bits is used to represent the sign of the number and the remaining 31 bits represent the magnitude of the number. One consequence is that the range of integers that can be represented is limited. The smallest allowable integer is $-2^{31} = -2,147,483,648$ while the largest integer is $2^{31}-1 = 2,147,483,647$.

3.3 Floating Point Numbers

In mathematics, numbers that have decimal parts (for example, π) are called rational, real or complex depending on their properties. In the computing world, numbers that have decimal parts are called floating point numbers (real or complex). Floating point numbers are divided into two levels of precision; single and double.

3.3.1 Single Precision

A typical single precision floating point number has a 32-bit binary representation in the computer's memory. This roughly corresponds to 7-8 decimal digits of precision. For IEEE computers, the allowable range of positive single precision floating point numbers is roughly $[10^{-38}, 10^{38}]$. Zero and the negative of this range are also included.

Note that when counting digits, always start with the first nonzero digit. For example, suppose

```
x = 365.23914
y = 0.00053691187
```

The variables `x` and `y` both have 8 digits. The leading zeros in `y` are ignored.

3.3.2 Double precision

A typical double precision floating point number has a 64-bit binary representation in computer memory. This roughly corresponds to 15-16 decimal digits of precision. For many years, double precision values were used only when absolutely necessary because memory was very expensive. In modern scientific computing, double precision is the most common precision used.

For IEEE computers, the allowable range of positive double precision floating point numbers is roughly $[10^{-308}, 10^{308}]$. Zero and the negative of this range are also included.

3.4 The Special Values Nan and Inf

NaN (Not a Number) and Inf (Infinity) are special cases of numbers that are used to indicate something unusual happened with a calculation. For example, if you ask a computer to compute 1/0, the result will be Inf. If you try to compute 0/0, the result will be NaN. A NaN will also be generated if the result of a calculation exceeds the allowable range for a number.

These special values are relatively new introductions to computing. Not that long ago if you tried to compute 1/0, your program would immediately terminate with a division by zero error. Similarly, exceeding the allowable range for a number would result in an overflow error.

3.5 Character Data

Not all computer data is numerical in nature. Data such as names, addresses, social security numbers and phone numbers are text based. These types of data are called character data (or string data).

3.6 A Short Note on Floating Point Precision

When talking about computing in this course, we are specifically excluding symbolic manipulators such as Maple and Mathematica. These programs are capable of manipulating algebraic expressions exactly and can exactly solve equations of various types (when feasible).

One of the consequences of the IEEE standard is that there is a limit to the precision of the calculations that can be performed. We have seen that integers and floating point values have limited ranges and precisions.

Consider the rational number $\frac{1}{3}$. The computer can't represent this number in rational form and instead will use the decimal equivalent

$$\frac{1}{3} = 0.\overline{3}.$$

Representing the decimal value exactly would require an infinite number of bits. This is not possible, so this representation must be truncated at some point. In double precision, this number would look like

$$\frac{1}{3} \approx 0.333333333333333.$$

In a practical sense, this means that every calculation involving a floating point number will almost certainly result in a calculation error, but hopefully this error will be small. For example, if you were to compute the double precision sum $z = x + y$ where

```
x = 0.123456789012345
y = 0.000123456789012345
```

the resulting value of z would not be exact. The exact value of z requires 18 digits and the computer can only store 15-16 of these. The error in the calculation would be small though (about 1 digit in the last place in the answer).

3.6.1 Types of Floating Point Errors

There are two types of floating point computing errors; storage errors and rounding errors.

A storage error occurs whenever a number is stored on a computer. This is more significant than it sounds because most decimal numbers that have digits after the decimal point have infinite binary expansions. For example,

$$7.6_{(10)} = 111.\overline{1001}_{(2)}.$$

The binary representation must be truncated at some point and this truncation is the storage error. Storage errors occur only once (when the number is first stored).

A rounding error occurs whenever the result of a a calculation has more digits than can be stored and must be truncated to fit into 64 bits. Rounding errors can accumulate over numerous calculations.

3.6.2 Why Use Double Precision?

Double precision floating point values seem to contain many more digits than is needed to perform a typical calculation, however there is a good reason that most scientific computing uses double precision.

A single calculation (add, subtract, multiply or divide) has the potential to cause a rounding error of 1 digit in the last place of the result. This effect is cumulative. This means that the result of 2 calculations can result in a rounding error of 2 digits in the last place, 3 calculations can be off by 3 digits in the last place, *etc.*. Extrapolating this relationship, 10 calculations can result in a rounding error of 10 digits in the last place (which is 1 digit in the next to last place).

```
x = 0.123456789012345
                   ^------ 10 calculations can cause the round-off error to
                           creep up to here
```

Similarly, 100 calculations can cause the round-off error to creep into the third-to-last digit in the worst possible case.

```
x = 0.123456789012345
                  ^------ 100 calculations can cause the round-off error to
                          creep up to here
```

Extrapolating this out to a trillion (10^{12}) calculations, it is possible for accumulated rounding errors to creep up into the position indicated below:

```
x = 0.123456789012345
         ^--------------- 10^12 calculations can cause the round-off error to
                          creep up to here
```

For a large-scale calculation (for example, performing a weather simulation or computing the flow of air around an aircraft), a trillion calculations is not unusual. Simulations involving 10^{15} to 10^{17} calculations are becoming very common. The use of double precision values means that the accumulated round-off errors are contained (most of the time) to the lower 8-15 digits and that the first 5-6 digits are reliable.