1 Introduction

With the invention of the transistor in 1947, computers were transformed from slow application of specific calculators into machines capable of providing general scientific support. Due to the greater reliability, longer life, and faster response time of the transistor, the transistor replaced the vacuum tube as the item of choice in electronic logic circuits. In 1958, Jack Kilby and Robert Noyces co-invented the integrated circuit, allowing the complex combinations of transistors, capacitors and resistors to be united onto a single monolithic block of silicon. In many ways this can be viewed as the beginning of the information revolution, as advances in the integrated circuit enabled more and more transistors to be layered onto the chip. Each advance lead to an increase in the rate at which calculations could be carried out, allowing computers to handle more sophisticated problems.

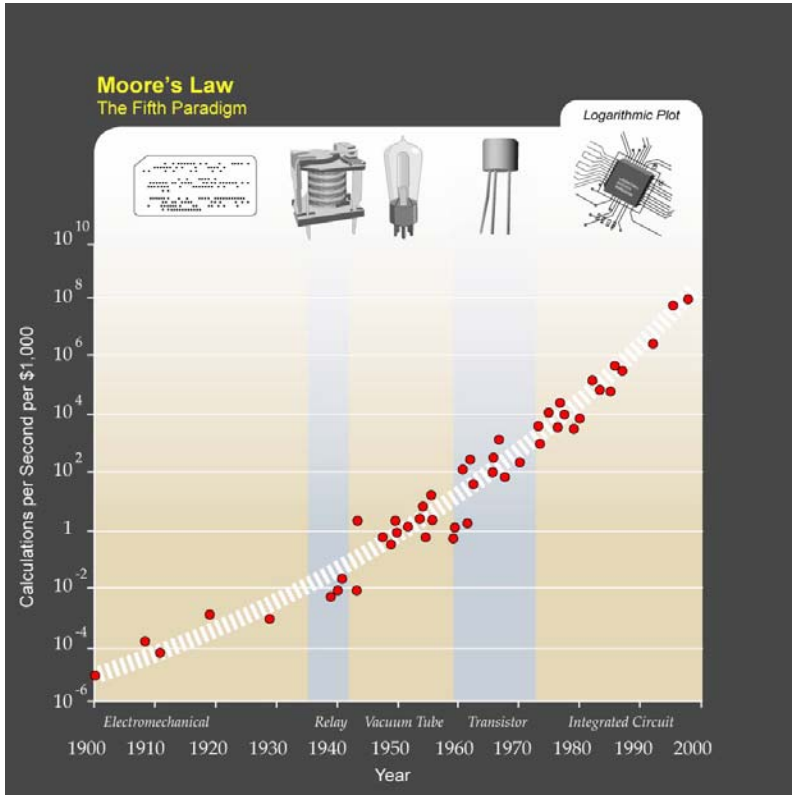

Figure 1: Kurzweil expansion of Moore's law from integrated circuits to earlier transistors, vacuum tubes, relays and electromechanical computers.

The theory of how to automate calculations was originally studied by Charles Babbage in the 1800s as a problem in mathematics. Many other mathematicians contributed to the problem, culminating with the theorems of John von Neumann in the 1930s. These mathematicians helped to develop what is now known as the field of computer science. Computer scientists search for answers to questions that are directly related to efficient use of computers in problem solving, looking at both hardware and software issues. Studying the computer represents an end onto itself.

1.1 Definition of Computational Science

In contrast, the focus of computational science is how to use the computer to solve scientific problems. This is a subtle yet distinct difference. A

computational scientist aims to develop stable, portable and self-consistent algorithms, all within the context of the problem being solved. From this point of view, the algorithms that look to be the best approach possible for this problem may appear to be naive, narrowly focused and a poor use of computer resources when viewed by a computer scientist. Strangely enough, within their respective expertise, both views are correct. Note that this does not imply that a computational scientist ignores the basic tenants of programming. Both the computational scientist and the computer scientist seek generic tools that can apply to a broad range of problems, and computational scientists usually adopt advances in computer science quickly.

Computational physics is the application of computational science to physics problems. Why is this important? Perhaps the best reason has to do with the limitations of traditional analytical approaches. While these tools,

such as integral and differential calculus and complex number theory, have enabled to solve an incredible range of problems, these techniques are primarily suited to study linear problems. However, the 20^{th} century has repeatedly demonstrated that the vast majority of outstanding problems are inherently nonlinear. These problems have tended to resist analytical solutions, and could usually be solved only by limiting the range of applicability so that the equations could be treated as approximately linear. While this has generated many useful insights, the true nature of these problems could not be appreciated. In addition, analytical solutions quickly become tedious if the system involves many degrees of freedom. Interrelationships become obscured and mistakes can creep in.

The increased capabilities of computers dramatically increased the range of solvable problems. In addition to their traditional role of providing numeric solutions to equations, computers are now used in a number of other ways.

Spurred on by the rising costs of experimentation, computational physics is used to conduct "experiments" in the virtual world of the computer memory. Simulations enable experimenters to refine their designs before any equipment is built or purchased, optimizing the detectors for the task at hand. Advanced programming languages allow problems to be solved symbolically as well as numerically, revealing the relationships between physical parameters. Computers oversee the complex arrays of data collectors used in modern experiments, provide motion control in increments finer than humanly possible, and perform analysis and filtering on the vast quantities of data generated by experiments. Advances in graphics enable data to be visualized, catering to the human ability to detect subtle changes and patterns not apparent otherwise.

1.2 Interface Between Theoretical and Experimental Physics

Since the days of Galileo, the primary goal of science has been to obtain data to understand the universe. Here data is defined to be the results of repeatable experiments that, ideally, use procedures and methods that are unassailable. This led to the development of the scientific method, which formalizes methods designed to approach this idea.

Newton's invention of calculus, and his application of it in the formulation of basic laws of motion, provided the mathematical framework necessary to begin exploring the inter-relationships between various measurements. This is considered to be the beginning of theoretical science. Notice that the theory is driven by data. This is one of the primary ways that science differs from philosophy.

During the centuries since Galileo and Newton, experimental physics has progressed from measuring basic data via simple experiments to subtle effects that can only be seen in intricate experiments operating at the extremes of physical measurability. This has led to two new phenomena: the ascendancy of theory over experiment and the rising cost of doing experiments.

Because experiments have become so expensive, they command a great deal of public resources. The demands of accountability have lowered the margin of error to the point that most experiments must be meticulously thought out beforehand - in both their construction and methodology. This need has seen the birth of a third component of physics, namely computational physics.

Computational physics serves a critical role at the interface between experimental and theoretical physics. In order to insure that experiments

are designed properly before construction is ever begun, detailed simulations of the apparatus are carried out. This can only be accomplished by understanding the interaction of the experimental device with the physical phenomenon it is meant to measure. Thus, computational physicists must understand not only the physics that is being studied, but also the capabilities and limitations of the experiment intended to make the measurements. In this way, computational physicists straddle the line between experimentation and theory.

1.3 Computing Platforms

Computational physicists need to be able to work on a variety of different computing platforms. Platforms can be categorized in two ways: by the

type of CPU installed, and by the operating system used. These two are interconnected, but since there are a large number of different CPUs, it is easier to look at operating systems. There are three main operating systems used in the world today: Unix based, Mac-OS based, and MS-Windows based. Each has its own strengths and weaknesses.

1.3.1 Unix Computers

Unix is actually the oldest of the three platforms. Originally developed in the 1969, it was one of the first attempts to develop an operating system that was

1) simple,

2) written in a high level language, and

3) allowed for the re-use of the code.

This enabled the basic operating system, known as the kernel, to be kept small. It also allowed other needs, such as networking and multi-user support, to be layered on top.

There are a number of strengths to computing platforms running Unix. Since it was originally developed in a scientific research environment, it has a large base of support among scientific programmers, especially older ones. It naturally supports a multi-user, multi-threaded environment, allowing different users to run different applications "simultaneously". Unix's network support is more robust than in Windows and Macintosh computers, and it is extremely rare for a problem to bring down the whole system.

Since the operating system is written in C, patching, extending and augmenting the operating system becomes especially simple. Lastly, most Unix workstations use chipsets that employ Reduced Instruction Set Computing (RISC). This typically results in a smaller number of instructions per line of code, with a subsequent reduction in execution time.

Another strength of the Unix system is that almost all of the major workstation producers support Unix. However, this is also one of the major weaknesses since each vendor produced their own variant, or flavor, of Unix. This led to incompatibilities between the different machines, and forced Unix users to learn not only the basic syntax of the Unix operating system, but a plethora of machine specific environments. The lack of a unified operating system standard prevented widespread acceptance of the Unix system, and so it has remained a tool primarily used in the scientific and networking communities. The introduction of Linux in the early 1990s

partially changed this view, but by then Microsoft Windows had such a commanding hold on the market that Linux has not been able to carve out much more than a niche for itself. A further perceived weakness with the Unix operating system was its long-term reliance on command line interfaces. This has been partially alleviated by the use of the X windowing system, but the large majority of tools and applications are still command line based.

1.3.2 Macintosh Computers

In 1976, Apple Computers was founded. After going through a series of successes and failures, Apple had been backed into a shrinking niche in the personal computer market. In an attempt to regain the initiative, Apple

introduced the Macintosh in 1984. Built around the Motorola 68000 chip, the Macintosh was the first computer to use a Graphical User Interface (GUI) to communicate with the user.

There are a number of strengths to the Macintosh computers. The GUI provides an intuitive means to interact with the computer, and allows the user to multitask through numerous open windows. The Motorola chip tends to be faster than the equivalent Intel chip, so the Macintosh is usually able to handle a heavier computational load. For many years Macintosh also had superior graphics abilities. Since graphical analysis can provide an intuitive grasp of relationships quickly, this helped propel the Macintosh to a dominant position in scientific laboratories, sharing the workload with Unix machines.

Unfortunately, a number of weaknesses also plague the Macintosh. Apple's decision to keep the manufacturing process proprietary prevents other computer manufacturers from "cloning" the Macintosh. This continues to limit

the extent in which Apple can compete in the computer market. Another problem is the fact that the Macintosh is a single user machine. While this may not appear to be a weakness, single user machines are inherently less secure and make sharing data between collaborators more difficult.

1.3.3 Windows-based Computers

Probably the most common computer today is the personal computer, or PC. While Linux and Apple have made some inroads into this market, the vast majority of PCs today use some form of the Microsoft Windows operating system. Originally introduced in 1985, Windows got off to a slow start. It wasn't until the release of Windows 3.0 in 1990 that Microsoft's operating system began to dominate the market.

The greatest strength of the Windows operating system has to be its prevalence. Over 90% (?) of the computers in the world use some version of Windows, and the number of applications available for use with Windows greatly outstrips those of its competitors. This provides a seamless path for data to flow from acquisition through analysis into presentation/publication format. Finally, Windows provides both single user computers through its Windows 95/98/Me systems, as well as multi user systems via Windows NT/2000/XP/Vista. This allows Windows to operate in a wide range of computing environments.

Windows is not without its drawbacks. It remains tied to the Intel 80×86 chips, which are Complex Instruction Set Computing (CISC). While CISC chips tend to need more instructions on average than a RISC machine, the use of pipelining has reduced the disparity between the two methods considerably. There is also a question of system stability. Of the three main systems, it is the most prone to system crashes, which can devastate long calculations.

1.4 Programming Languages

Just as there are a number of different computing platforms, there are also a wide variety of programming languages available. The languages are typically classified according to the advances that they incorporate, with each major advance represented as a new generation.

The first generation language is machine code, with the program written as a string of numbers that the computer interpreted as instructions. This was the most difficult way to program a computer, but is the closest to how the computer actually works. In fact, all later generation languages are translated into machine code before the CPU processes them.

Second generation languages are considered to be various assembler languages. Since they use English acronyms, these languages are easier to

understand than straight machine code, but they are still arcane compared to later generation languages.

Third generation languages are also known as “high level” languages. A compiler takes the statements of a particular third generation language and converts it into machine code.

Fourth generation languages are similar to third generation languages in that they rely on a compiler or interpreter, but they are usually written in a language which has structure and flow similar to spoken languages.

Fifth generation languages are the latest set of computing languages. These languages use a visual or graphical interface to generate source code that is then compiled via a third or fourth generation compiler.

In general, most computational physicists need expertise in third, or later, generation languages. The main exception to this is if they are working on software that must run in an embedded system or in a real time operating system, and the speed of the code is paramount. In this case, the ability to trim any excess commands out of the code stream becomes important, and assembly languages are the best choice. Of the high level languages, the one that are most commonly found in physics are Fortran, C/C++, MatLab, Mathematica, Maple IDL, and PAW.

1.4.1 Fortran

Fortran is the oldest of the scientific computing languages. Originally designed in 1954, it went through various modifications. Fortran II was

released in 1958. Fortran III was a short-lived release, but Fortran IV, which was released in 1962, became the mainstay of scientific computing. A large number of scientific programs and libraries were written in Fortran IV. When ANSI Fortran (Fortran 77) was released in 1977, these programs and libraries were quickly updated and augmented.

What made Fortran so widely accepted in the scientific community? The language had a number of built in data types that eased scientific computing, including real, double and complex. It also had a wide variety of controlling and branching structures. These two facts enabled large programs to be created which had the ability to track variables associated with experimental measurements.

Fortran 77 remained the standard scientific language for more than ten years. During this time computer science made a number of significant

advances, prime among them the invention of pointers that allow indirect access to data. Various vendors released patches to Fortran 77 that allowed it to also do this, but it wasn't until the release of Fortran 90 that Fortran gained many of these tools.

Fortran remains an important language in computational physics, both due to the large libraries of code that still exist and due to the strengths still inherent in the language itself. It is an important tool in the computational physicist's toolbox, and one that a student should learn.

1.4.2 C/C++

C is a more recent entry into the field of scientific programming. It was released in 1971 and was designed to provide a computing development tool. Its strengths include the facts that

It has high level constructs,

It can handle low-level activities, and

It produces efficient programs.

From a scientific standpoint, the loose typing that C provided quickly enabled all of the strengths of Fortran to be copied and expanded upon. The rise of Unix workstations, whose kernel is written in C, also aided in the acceptance of C in the scientific community. In addition, the use of header files and longer variable names made documentation of code libraries easier.

As computer science paradigms advanced, the idea of object-oriented programming was developed. Object-oriented programming (OOP) uses a

different approach to problem solving than the method used in older languages such as Fortran and C. These earlier languages use a method known as a "procedural approach". A problem is broken into smaller problems, and this process is repeated until the subtasks can be coded. Thus a library of functions is created, communicating through arguments and variables. This approach is very similar to the problem solving methods taught for analytical problems in freshman physics classes.

In contrast, an object-oriented approach identifies the key ideas in the problem. These key ideas are then connected together to define an internal hierarchy. The key ideas will be the objects in the program and the hierarchy defines the relationship between these objects. From this, an object is seen to be a limited, well-defined structure, containing all information about some entity: those variables associated with physical data, and the functions required to manipulate the data. This approach

mirrors the physical world more than the procedural approach does, and makes programming physical interactions a more natural procedure. The fact that a program object mirrors a physical one also lends itself to the creation of libraries of physical objects, which can be combined together to naturally simulate the interactions of experiments with the physics they are trying to measure.

These ideals were added to the C language though the development of C++. Originally designed to be a pre-processor for C, the differences between C and C++ were soon recognized to be enough that C++ compilers were developed independently. The history of C++ as a pre-processor can be seen in the similarities between the two languages. In fact, C++ forms a superset of the C language and is (almost) completely compatible with C program structure.

1.4.3 IDL, PAW, Mathematica and Maple

IDL, PAW, Mathematica, and Maple are all examples of fourth generation languages. Unlike the third generation languages discussed above, these are interpreted languages. Interpreted languages are ones in which the code is passed directly to a parser, which converts it to machine code and executes it immediately. This allows the program to be written "on the fly" and modified as intermediate results are seen. Interpreted languages do not usually produce a stand-alone executable, and are typically slower than their compiled counterparts.

IDL, which stands for Interactive Data Language, and PAW, which stands for Physics Analysis Workbench, are data analysis and graphics packages. They can read ASCII or binary data files, do a wide range of sophisticated

mathematical operations on the data, and then present the results in either a textual or graphical output. Both packages are widely used as post processing software in experimental support.

In contrast, Mathematica and Maple are symbolic computation programs. While they can both carry out the same duties as IDL and PAW, their greatest strength lies in their ability to perform mathematical calculations using formulae similar to those that would be used in an analytical approach. For example, a user could input a set of linked equations and ask the program to solve them for a specific variable. The result will be presented (if possible) as an analytical formula rather than as a number.

The fact that all of these packages are fourth generation languages makes their use much more intuitive than standard "programming" languages. Users do not necessarily need to know arcane calls or coding techniques to achieve their goals.

2 Computational Methods

2.1 Programming Computers

For our purpose, a computer may be regarded as a digital electronic device capable of carrying out mathematical operations.

At the machine level almost all the computers are constructed to understand only machine language instructions. These are usually expressed in a binary representation, that is, a string of zeros and ones, but are hardly understood by human beings.

To make a machine that is fast in its operations and easy to construct, we want our computers to be able to carry out very complicated tasks with a minimum account of coding.

2.1.1 High-level computer languages

High level computer languages are constructed that are closer to the working languages of various applications. The conversion of a computer program written in one of the high level languages to a set of machine instructions is the function of a compiler. The output of a compiler is called the object code of the program so as to distinguish it from the source code, which is written in a high level language.

Before we embark on a project, a few basic considerations must be examined:

To find errors: bug and debug

Readable

Efficient

2.1.2 Structured programs

Independent sections or subprograms: Use subroutines to organize the major task and make the program more readable and understandable. The main program is basically an outline of the program.

Use descriptive name: Choose the names of variables and subroutines according to the problem at hand. Descriptive name make a program easier to understand, as they act as build-in comment statement.

The comments: Include comment statements explain program logic and describe variable.

Take time to make the graphics presentable: In almost all cases, numerical results should be presented graphically. The axes should be labeled clearly and parameter value given directly on the graph.

2.1.3 Libraries of subprograms

Intrinsic functions

Program library

Private library

2.1.4 Choosing a computer languages

The FORTRAN: The language has been developed over the years into a form that is easy to learn and powerful enough for most needs in physics.

The C and C++: The language is perhaps the most powerful. It has the capability of handling characters as well as numerical variables.

The Mathematica/Maple: The languages are aimed at symbolic manipulation and are therefore the languages of choice for algebraic calculation.

Other languages, such as ALGOL, APL, and BASIC, are also used in physics, but not popular.

2.2 Testing your program

Checking a program is a not a trivial task, but there are some general guidelines. After a program has been debugged so that it can run without any complaints about the syntax, there are several things you should do to verify that the results of the program are correct.

Does the output look reasonable?

Does your program agree with any exact result that are available?

Always check that your program gives the same answer for different "step size".

Checking your program should not be viewed as a trivial, last minute job. It is not unreasonable to spend as much time checking and correcting a program as it takes writing it. After all a result is not much good if you don't trust it to be correct.

2.3 Example

2.3.1 Example: Integers and floating numbers

Nearly all computers make a clear distinction between integers and floating numbers. The reason for differentiating between these two types of numbers comes from the structure of the computer memory.

Bit: The basic unit for storing a number in a computer is a *bit*, the state of an electronic component that is either on or off. The two possible state of a bit may be used to represent two numerical values 0 (off) or 1 (on).

Byte: consists of eight-digit binary number. The largest integer in one byte is

$$\begin{aligned} b11111111 &= 2^8 - 1 \\ &= 255 \end{aligned}$$

Representation of number

Binary: based on powers of 2

Octal: based on powers of 8

Decimal: based on powers of 10

Hexadecimal: based on powers of 16

Decimal	Hexa-	Octal	Binary
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

Integer

It is a common practice to designate the first bit of a two-byte or four-byte integer word as a sign bit.

The possible value for a two-byte integer I_2 is

$$-32,768 \leq I_2 \leq 32,767$$

corresponding to -2^{15} to $2^{15} - 1$.

The possible value for a four-byte integer I_4 is

$$-2,147,483,648 \leq I_4 \leq 2,147,483,647.$$

corresponding to -2^{31} to $2^{31} - 1$.

Floating number

Floating numbers broaden the range of values that can be stored in the computer memory by allocating a part of word as the exponent of each number. Each number has a sign, a fraction part or mantissa, and an exponent.

For a four-byte number, 1 bit is devoted to the sign, 8 bits can be assigned to the exponent, and 23 bits are left for the mantissa. In this way, a floating number ranges from 1.2×10^{-38} to $3.4 \times 10^{+38}$.

If we get a number whose absolute value is too small to be represented in a computer, an underflow condition is created. When this happens, the number is replaced by zero. If we get a number whose absolute value is larger than 10^{38}, an overflow condition is created. In this case, the machine is to suspend the calculation.

2.3.2 Example: a calculation for prime number

A prime number is defined as an integer that is divisible only by unity and the number itself. As an example of integer calculation, let us consider a simple way to find a list of prime numbers.

In ordinary calculation,

$$\frac{5}{2} = 2.5.$$

In integer mode,

$$\frac{5}{2} = 2$$

where the fractional part is truncated. Therefore, if an integer n is divisible, we have

$$\frac{n}{m} \times m = n.$$

In n is not divisible by m,

$$\frac{n}{m} \times m < n.$$

Let us assume that we have already found the first n_p prime numbers, p_1, p_2, $\cdots$, p_{max}, arranged in ascending order according to size. To find the next prime number, we start with an integer $N = p_{max} + 1$, where p_{max} is the largest the prime number known so far. We can test whether N is divisible by dividing it with all the members of the first n_p prime numbers. If N is divisible by any of the existing prime number, it is not a prime number. We shall therefore increase N by one and repeat the process.

This process goes on until we reach a value of N that is not divisible for all the existing prime number. Such a number is a new prime number and may be added to the list as the largest prime number.

Prime number: 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.....

A few improvement may be incorporated into the method to make it more efficient. (1) we recognize that 2 is prime number. As a result all other even numbers are not prime number. For these reason, when we look for the next integer (> 2) to be test we can increase N by 2 each time, rather than 1. In this way the speed of the search is increased by a factor of 2. (2) We do not need to go through the entire list of the existing prime numbers in our test. Obviously, if N is not divisible by any member up to $p_i > N/2$, it will not be divisible of any of the prime numbers in the list that are larger than p_i.

Program DM_PRIME

Generate the first K prime numbers

Initialization

- *– Set K as the total number of prime numbers required.*
  - *– Set N_BEG as the number of prime numbers to start with.*
  - *– Input the first N_BEG prime numbers explicitly*

(1) Store the last prime number in the list as NEW.

(2) Construct a new integer by adding 2 to new.

(3) Test if NEW is prime by dividing it by all existing prime number $\leq$ *NEW/2.*

(a) If divisible, try a new integer by adding 2 to NEW and repeat the test.

(b) If not divisible, it is a new prime number.

(i) Add the new one to the list.

(ii) Increase the number of members by one.

(4)Repeat steps 2 and 3 to find the next prime number in the list.

(5) Stop if the total number is K.

(Write a program as an assigment)

2.3.3 Example: a calculation for the value of π

The ancient Chinese method To estimate the ratio of the circumference to the diameter of a circle, the π value, by measuring to a very high accuracy the ratio of the length of the side of a polygon to its diameter. If the length of the side of a regular polygon with n sides is denoted as l_n

and the diameter is taken as the unit of length, then the approximation of π is given by

$$\pi_n = nl_n.$$

The exact π value is the limit of π_n as $n \to +\infty$. The value of π_n obtained from the measurements on the polygons can be formally written as

$$\pi_n = \pi_\infty + \frac{a_1}{n} + \frac{a_2}{n^2} + \frac{a_3}{n^3} + \cdots.$$

n	π_n
8	3.061467
16	3.121445
32	3.136548
64	3.140331
⋮	

From the four data, one can truncate the expansion at the third order and solve the equation set to obtain the value of π_∞. The approximation of π is $\pi_\infty = 3.141583$. The extrapolated value can be improved if one can accurately measure π_n with higher value of n.

A useful formula to calculate l_n

$$\begin{aligned} \sin\frac{\theta}{2} &= \left(\frac{1-\cos\theta}{2}\right)^{1/2} \\ &= \left(\frac{1-\left(1-\sin^2\theta\right)^{1/2}}{2}\right)^{1/2} \end{aligned}$$

where $\theta \leq \pi/2$. $\sin\frac{\pi}{4} = 2/2$.

(Write a program)

Buffon's Needle Problem: Monte Carlo Simulation Buffon's Needle is one of the oldest problems in the field of geometrical probability. It was first stated by the French naturalist Buffon in 1733. It involves dropping a needle on a lined sheet of paper and determining the probability of the needle crossing one of the lines on the page. The remarkable result is that the probability is directly related to the value of π.

Let's take the simple case first. In this case, the length of the needle is one unit and the distance between the lines is also one unit. There are two variables, the angle at which the needle falls θ and the distance from the center of the needle to the closest line (D). θ can vary from 0 to 180 degrees and is measured against a line parallel to the lines on the paper. The distance from the center to the closest line can never be more than half of the distance between the lines. The graph below depicts this situation.

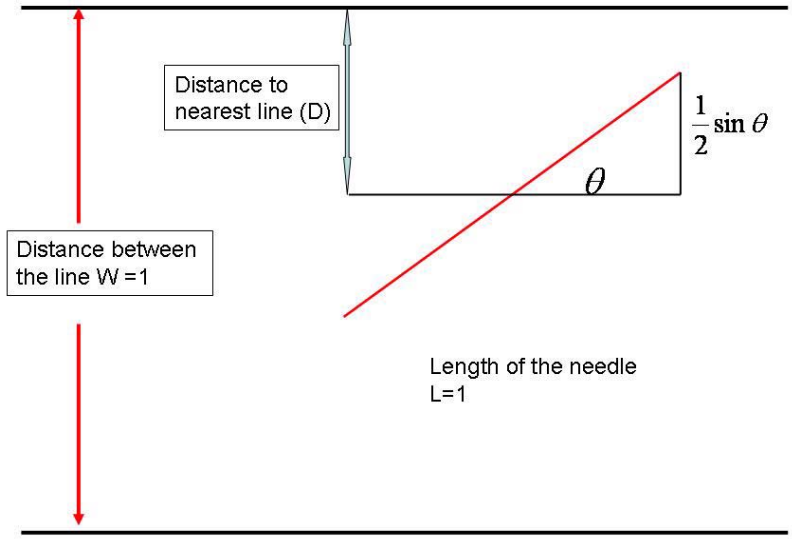
Distance to
nearest line (D)
$\frac{1}{2}\sin\theta$
θ
Distance between
the line W =1
Length of the needle
L=1

Figure 2:

The needle in the picture misses the line. The needle will hit the line if the closest distance to a line (D) is less than or equal to $1/2$ times the sine of theta. That is, $D \leq 0.5 \sin\theta$. How often will this occur?

In the graph below, we plot D along the ordinate and $0.5 \sin\theta$ along the abscissa. The values on or below the curve represent a hit ($D \leq 0.5 \sin\theta$). Thus, the probability of a successful hit is the ratio of the shaded area to the entire rectangle. What is this value?

The shaded portion is found with using the definite integral of $(1/2) \sin\theta$ evaluated from zero to π.

$$\int_0^\pi \frac{1}{2} \sin\theta d\theta = 1$$

The result is that the shaded portion has a value of 1. The value of the entire rectangle is $\pi/2$. So, the probability of a hit is $1/(\pi/2)$ or $2/\pi$. That's approximately 0.6366197.

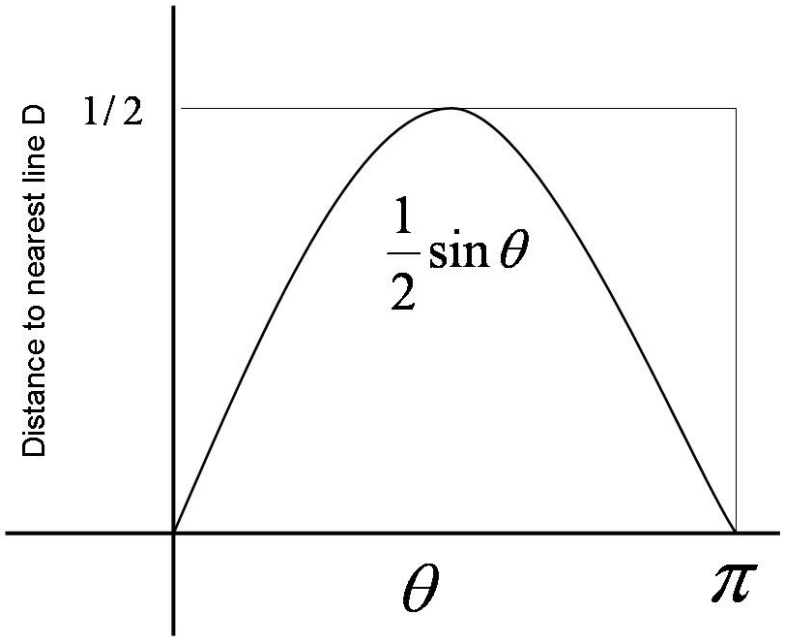

Figure 3:

To calculate π from the needle drops, simply take the number of drops and multiply it by two, then divide by the number of hits, or 2(total drops)/(number of hits) $= \pi$ (approximately).

Numerical Approach:

- N: the number of steps; M: the number of hit
- Generate two random numbers $x \in [0, 0.5]$ and $y \in [0, 1]$. x is for the location of the center of the needle and y is for the angle θ.
- Test $D(x) \leq 0.5 \sin y\pi$: if yes, $M- > M + 1$, $N- > N + 1$; if not, $M- > M$, $N- > N + 1$

- $Pi = 2N/M$

- Repeat

Questions

1. After 1,000 drops, how close would you expect to be to π?

2. After 264 drops, the estimate of π is 3.142857. This estimate is correct to within 2/1000 of the book value of π. Will the next drop:

a. make the estimate more accurate?

b. make the estimate less accurate?

c. make it more or less accurate depending on whether it's a hit or miss?

d. impossible to say.

3. What about the next 10 drops?

Series expansion As an example of computational algorithm, let us calculate the value of π. Starting from the relation,

$$\int_0^1 \frac{dx}{1+x^2} = \tan^{-1} x|_0^1 = \frac{\pi}{4},$$

and

$$\int_0^1 \frac{dx}{1+x^2} = \int_0^1 \sum_n (-1)^n x^{2n} dx$$

we have a series expansion

$$\begin{aligned}\frac{\pi}{4} &= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \\ &= \sum_{n=0}^{+\infty} \frac{(-1)^n}{2n+1}\end{aligned}$$

It is fairly straightforward to program the summation. All we need to do is a simple loop that alternates between adding to and substracting from the sum of inverse of the next odd integer. On the other hand if we examine the number of terms required to achieve a given accuracy in this approach, we will be surprised. For example, if we wish to achieve an accuracy of 10^{-5}, the last term to be included in the summation must be much smaller than 10^{-5}.

Alternatively,

$$\frac{\pi}{4} = 1 - 2\left(\frac{1}{3 \times 5} + \frac{1}{7 \times 9} + \cdots\right).$$

In this way the number of terms required to achieve the same accuracy of 10^{-5} is decreased to 62,500, a factor of 4 reduction.

What do we learn from this example?

- One problem can be solved by different methods: Analytical approach, numerical simulation or experimental measurement, and sometimes exact solution.

- Choose a method suitable for you.

2.3.4 Example: Radioactive decay

Image that we have a sample containing a large number of ^{235}U nucleai which would be usually the case if we were actually doing an experiment to study radioactive decay. If $N_U(t)$ is the number of uranium nuclei that are present in the sample at time t, the behavior is governed by the differential equation,

$$\frac{dN_U(t)}{dt} = -\frac{N_U(t)}{\tau}$$

where τ is the time constant for the decay. The solution of this equation is simple,

$$N_U(t) = N_U(0)e^{-t/\tau}.$$

A numerical approach:

$$N_U(t + \Delta t) = N_U(t) + \frac{dN_U(t)}{dt}\Delta t + \cdots$$

$$N_U(t + \Delta t) = N_U(t) - \frac{N_U(t)}{\tau}\Delta t + \cdots$$

Constructing a working program:

1). The overall structure of the program consists of four basic tasks:

1a. declare the necessary variables■

1b. initialize all variables and parameters■

1c. do the calculation■

1d. store the results.

2). The rest of the work is done in three subroutine, initialize, calculate, and store, which is called in succession.

2a. initialize: the subroutine initialize sets the initial values of variables, such as the initial value of number of nuclei $N_U(0)$, the time Δt and the number of steps for calculation N.

2b. calculate: The real work of computing the number of remaining nuclei is done in the subroutine calculate.

- Begin with $i = 1$, $t_1 = \Delta t$.

- $N_U(t_{i+1}) = N_U(t_i) - N_U(t_i)\Delta t/\tau$

- $t_{i+1} = t_i + \Delta t$

- Repeat $i -> i + 1$

2c. store: The subroutine store writes the result to a file. $\{N_U(0), N_U(t_1), N_U(t_2), \cdots, N$

Suggestion: write a program to calculate $N_U(t)$ yourself. For example, take $N_U(0) = 1000$, $\tau = 1s$, and $\Delta t = 0.01s$.

2.4 Problems

1. If a is any one of the five integers 3, 4, 5, 6 and 7, and b is any one of the three integers 2, 3, and 4, what are the 15 values of c=a/b if the calculation is carried out in integer mode on a computer?

2. Describe a way to calculate the square root of a complex number using a function that can only take the square root of real numbers.

3. Give a way to find on a computer the roots of quadratic equation $ax^2 + bx + c = 0$ for a set of input values of the three coefficient a, b, and c.

4. In the ancient Roman system of notation, the letter I is used to stand for numerical value 1, V for 5, and X for 10, L for 50, C for 100, D for 500, and M for 1000. If a letter is followed by one of equal or lesser value, the two values are added. On the other hand, if one of the above letters is followed by one of greater value, it is subtracted from one of the following. Thus MCMLXXIV means 1974. (a) Design an algorithm to convert Roman numerals to integers and verse versa.

(b) Think of a way to add two integers expressed in Roman numerals without first converting them to their corresponding numerical value.

5. Write a program to solve the differential equation for the radioactive decay, or to find the prime number.