

Introduction to Computers and Programming using C++ and MATLAB

Alex F. Bielajew
The University of Michigan
Department of Nuclear Engineering and Radiological Sciences
2927 Cooley Building (North Campus)
2355 Bonisteel Boulevard
Ann Arbor, Michigan 48109-2104
U. S. A.
Tel: 734 764 6364
Fax: 734 763 4540
email: bielajew@umich.edu

© 2000—2010 Alex F Bielajew

Revision: October 12, 2010

Preface

This “book” arises out of a course I teach, a four-credit (52 hour) freshman-level course *Introduction to Computers and Programming* being taught in the College of Engineering at the University of Michigan. The book is in reasonably rough shape at this stage. It was assembled from my lecture notes several years ago and is under constant revision. I may never finish it! A wise person once said, “Old age happens when you dwell more on the past than on the future.” By this definition, I have found eternal youth, insofar as this book is concerned.

My educational objectives are quite simple. This is not really a course in computing. It is a course in thinking, technical thinking, logical thinking, about formulating a problem, a mathematical problem, a physics problem, a game, and then executing a solution and making it “work”. I consider the computer and the ability to program it as a kind of laboratory—a laboratory to investigate practically, the theories and ideas of other technical courses. It is possible, for example, to teach a lot of Calculus to students without ever mentioning the word, and there are several examples throughout this book.

This course is not about syntax. Hence, the book introduces the minimum amount syntax to get through a problem. Indeed, I even keep some syntax hidden, to encourage students to discover their own algorithms. So, if you are thinking of using this book as a technical reference in C++ or Matlab, I anticipate that you will be disappointed.

The greatest value in this book, if there is any to be found, is in the exercises, problems and projects at the back of almost every chapter. The ideal way to learn a computer language is to learn a little syntax and then try it out on a computer. The ideal way to think is not to read about it, but to actually do it! The book reflects this. The material in the chapters, separated from the exercises, is worse than useless, for reading the material and not doing the problems is just a waste of time. Do the problems! Moreover, don’t ask me for the solutions! There is much more pedagogical value in a well-posed question than a well-articulated answer.

OK, I’ll step off my soap box now!

Several professors and many students have contributed to the ideas put into this book. Professor James Holloway and I have had endless discussion on the general problem of teaching algorithmic thinking to freshman. We still have not come to any conclusions except

that it is highly challenging and rewarding. As for how it actually gets done, well, to be sure it is a works-in-progress. I spent a very enjoyable term co-teaching this course (the first time I taught it) with Professor Ken Powell. Thanks, Ken, for nursing me through that first year! Both James's and Ken's diverse computational backgrounds are reflected to some degree in this book.

I owe a debt of gratitude to the dozens of Graduate Student Instructors who have taught with me on this course. One of them, Dan Osborne, deserves special recognition. He is one of the most gifted and committed teachers I have ever encountered. Dan and I spent hour after hour discussing the challenges of teaching computing and thinking skills to undergraduates.

To the 3000 or so undergraduates, mostly freshmen, who have taken my course: Every time I teach this course I learn something new—about computing, about teaching, about the joy of making an early “deflection” in a student's career.

Finally, I am one of those people who tends to see the forest, instead of the trees. Consequently, spelling and sentence structure usually yield to an enthusiasm to present the material in an interesting way. It's not my only fault, but maybe my most visible one. To this end, please inform me if you spot any errors. I am also deeply indebted to my wife-to-be, Linda Park, for her unimaginable attention to detail, her proofreading and suggestions. Linda, every time you undertake a proofreading project on my behalf, my mind boggles, truly.

AFB, October 12, 2010

Contents

1	Introduction to the course	1
1.1	What this course is about	1
1.2	Problems	6
2	Data representations	7
2.1	Bits, nibbles, bytes and words	7
2.2	4-bit Binary, Hexadecimal and Decimal Digits	9
2.3	Binary arithmetic	9
2.4	Converting from binary to decimal	10
2.4.1	Whole numbers	10
2.4.2	Real numbers	11
2.5	Binary integer arithmetic on computers	11
2.5.1	Two's-complement integer arithmetic	12
2.6	32-bit Binary, Hexadecimal, Unsigned and Signed Integers	13
2.7	Problems	15
3	Algorithms and Pseudocodes	23
3.1	What is an algorithm?	23
3.2	Flowchart representation	24
3.3	Pseudocode representation	24
3.4	Decisions, conditionals, branch instructions	25
3.5	Looping or jump instructions	25

3.6	Sequencing, branching and looping	26
3.7	A mini-summary before the examples	27
3.8	Some examples	27
3.8.1	The Breakfast algorithm, or, Biela Jew's Sunday Morning Internationally-famous pancakes	28
3.8.2	Solve for x : $Ax^2 + Bx + C = 0$ where A, B, C are arbitrary constants	29
3.8.3	Iteration: A summing loop	33
3.8.4	Iteration: A product loop	36
3.9	An aside on computer architecture	40
3.9.1	What does $S = S + 1$ mean?	40
3.10	Problems	41
3.11	Projects	47
4	Getting started in C++	49
4.1	Simple input/output (I/O): A first program in C++	49
4.2	Compiling, linking, loading and running	52
4.3	Declaring and initializing variables	55
4.4	Integer math in C++	57
4.5	Floating point math in C++	59
4.6	The <code>if/else if/else</code> construct	60
4.7	Logical expressions	63
4.7.1	Logical expressions with AND or OR	66
4.7.2	Mixed arithmetic and logical expressions	67
4.8	Problems	69
4.9	Projects	73
5	Loops	83
5.1	The <code>while</code> loop	83
5.2	The <code>do/while</code> loop	86
5.3	The <code>for</code> loop	89

5.4	Problems	94
5.5	Projects	111
6	Early Abstraction—Functions	123
6.1	Motivation for functions	123
6.2	User-defined functions	129
6.3	Example: A problem tackled with teamwork	134
6.4	Call by value, call by reference, reference parameters	138
6.4.1	The address of a variable	138
6.4.2	Call-by-value <i>vs.</i> call-by-reference	140
6.5	The rules of scope	145
6.6	Problems	153
6.7	Projects	160
7	More Variable Types, Data Abstraction	167
7.1	Representation of floating-point numbers	167
7.1.1	When is one not one? Floating point anomalies.	174
7.2	<code>char</code> : the character variable	175
7.2.1	Character strings; the string class	177
7.3	The vector class	179
7.3.1	Introduction to vectors	179
7.3.2	Declaring and using vectors	180
7.3.3	Vector syntax and rules:	182
7.3.4	Vectors and functions	187
7.4	Problems	189
8	More Data Abstraction, Arrays, Structures	199
8.1	Arrays	199
8.1.1	Declaring a 2-dimensional array	200
8.1.2	Initializing a 2-dimensional array	201

8.1.3	Passing a two-dimensional array to a function	204
8.2	Character arrays	209
8.2.1	One dimensional character arrays	209
8.2.2	Two dimensional character arrays	211
8.3	Structures	211
8.3.1	The structure definition	212
8.3.2	Where can structures be defined?	213
8.3.3	The members of a structure	213
8.3.4	Declaring structure variables	213
8.3.5	Assigning values to structure members	213
8.3.6	Re-assigning values of structure members	214
8.3.7	Function call-by-value of a structure	214
8.3.8	Function call-by-reference to a structure	215
8.3.9	Structure arrays	215
8.3.10	Example using structures and arrays: Ion transport	218
8.4	Problems	222
8.5	Projects	228
9	Miscellaneous Topics	239
9.1	Generating random numbers	239
9.1.1	Example: Integrating functions by random sampling	246
9.2	Simple Sorting: The bubble or sinking sort	248
9.3	Recursion—A function calling itself	249
9.4	Input and Output using files	252
9.4.1	File and stream handling in C++	252
9.4.2	Creating sequential access files and writing to them	255
9.4.3	Reading from sequential access files	258
9.5	Command line arguments	261
9.6	Problems	264

10 A Potpourri of Applications	267
10.1 Finding a zero using the “binary chop”	267
11 Programming in MATLAB	271
11.1 A taste of MATLAB	271
11.2 Arrays in Matlab	273
11.3 Loops	279
11.3.1 The for loop	279
11.3.2 The while loop	282
11.4 The if/else construct	282
11.5 Some Matlab terminology	283
11.6 Operators in MATLAB	284
11.6.1 Math operators in Matlab	284
11.7 Pointwise operators in MATLAB	285
11.7.1 Logical Operators in Matlab	286
11.8 M-files	288
11.8.1 Script M-files	288
11.8.2 Function M-files	292
11.9 Problems	295
11.10 Projects	327
11.10.1 Supplementary material: Trajectories without air resistance	346
11.10.2 Supplementary material: Trajectories with air resistance	346
11.10.3 Supplementary material: Stepping algorithms	347
11.10.4 Supplementary material: Trajectories of objects in the universe	354
11.10.5 Supplementary material: Stepping algorithms	355
12 Graphics	367
12.1 Two Dimensional Plots	367
12.1.1 A basic plot	367
12.1.2 Using different line colors	367

12.1.3	Making titles	370
12.1.4	Axis labels	370
12.1.5	Different line styles	370
12.1.6	Point plots, plotting with point symbols	372
12.1.7	Plotting more than one thing at once	377
12.1.8	Subplots	378
12.2	Three dimensional graphics	379
12.2.1	Plot3 plots	379
12.2.2	Mesh plots	379
12.2.3	Axis labelling	381
12.2.4	meshc and meshz plots	382
12.2.5	Surface plots using surf	382
12.2.6	Surface plots using surfc	383
12.2.7	Surface plots using surfl	383
12.2.8	Contour plots using contour	384
12.2.9	Contour plots using contour3	384
12.2.10	Contour plots using pcolor	384
12.2.11	Contour plots using contourf	385
12.2.12	Contour plots with labels	385
13	Miscellaneous topics	387
13.1	Pitfall review	387
13.1.1	Review of for loops	387
13.1.2	Review of functions	390
13.2	The MATLAB way	395
13.3	Selected Applications	395
13.3.1	Free fall with air resistance	395
13.3.2	Diffusion of charged ions in an electric field	400
13.3.3	Calculation of Electric fields	402

13.4 Summary of the Course	404
14 Programming Style Guide for C++	405
15 Syntax reference for beginning C++	413
16 Syntax reference for more advanced C++	417
17 Syntax reference for MATLAB	421

Chapter 1

Introduction to the course

Computers are stupid. They only give answers.

Computers are blindly faithful. They only do what you tell them to do.

Computers are dumb. They do not do what you want them to do.

1.1 What this course is about

This course is called “ENG101: Introduction to Computers and Programming”.

The College of Engineering Bulletin describes it as follows (emphasis is mine)

*Introduction to the organization of digital computers, computing (i.e. programming) as a **tool** in engineering, and programming in C++ and MATLAB*

Let’s define a few things first...

What is an engineer? What is engineering?

The *American Heritage Dictionary* defines engineering as *The application of science to practical ends, such as the design, manufacture, and operation of structures, machines, and systems. An engineer is defined as someone who practices engineering as a profession.*

Our definition is somewhat broader. Engineering is a broad range of technical endeavor that starts with the basic sciences (trying to characterize and understand the basic laws of nature), integrates and systematizes this knowledge and builds things based on these principles, manufactures these “things” on a large scale, and makes it available for societal benefit. If we broaden our definition of “things” to ideas, methods, algorithms (a prescription for doing something, like a recipe in cooking), as well as physical devices, then that captures the essence of what engineering is today.

In our College of Engineering you will encounter faculty and staff who might otherwise have been called Mathematicians, Physicists, Chemists and Biologists.

What is a computer?

Again, we appeal at first to some standard reference texts.

American Heritage Dictionary

1. One (a person) who computes.

Some of the best examples:

- Richard Feynman's computers—teams of people working out calculations during the Manhattan Project
- Calculation of ballistics tables for WWII
- Calculation of sine/cosine/logarithm tables

2. A device (a machine) that computes, especially a programmable electronic machine that performs high-speed mathematical or logical operations or that assembles, stores, correlates, or otherwise processes information.

Oxford English Dictionary

1. One who computes; a calculator, reckoner; spec. a person employed to make calculations in an observatory, in surveying, etc. (Earliest reference 1646).
2. A calculating-machine; esp. an automatic electronic device for performing mathematical or logical operations; freq. with defining word prefixed, as analogue, digital, electronic computer (see these words). (Earliest reference 1897).

McGraw-Hill Encyclopedia of Science and Technology

A device that receives, processes, and presents information. The two basic types of computers are analog and digital. Although generally not regarded as such, the most prevalent computer is the simple mechanical analog computer, in which gears, levers, ratchets, and pawls perform mathematical operations – for example, the speedometer and the watt-hour meter (used to measure accumulated electrical usage). The general public has become much more aware of the digital computer with the rapid proliferation of the hand-held calculator and a large variety of intelligent devices, ranging from typewriters to washing machines.

Our definition A “device” that

1. accepts some form of input (usually disorganized or of some highly specific nature),
2. responds to this input information in a “pre-determined” (programmed) way,
3. produces some form of output (usually more organized or coherent than the input)

Some common examples of computers

- Personal: Wristwatch, electronic car keys, phone, radio, GPS device, calculator, laptop...
- In the home: Stove, refrigerator, freezer, microwave oven, clocks and timers, toaster, coffee machines, thermostats, washer and dryer, TV's, VCR's, audio equipment...
- Automotive: fuel delivery, air/fuel mixture, spark advance, brakes (ABS), traction control, suspension adjustment, climate control, navigation, computer-controlled driving (coming soon)...
- Mechanical analog computers: Wind-up clocks, toys, mechanical calculators...
- Electrical analog computers: Integrators, differentiators, differential equation solvers, Christmas light flashers (cheap ones!)...
- Digital computers: Calculators, PDAs, Palm computers, laptops, desktops, mini-computers, mainframe computers....

Digital computers?

Digital computers are the simplest of all! They only understand two things, which we label “1” and “0”, or “on” or “off”, but more precisely, states of “high” or “low” voltage. Computers switch between these two states very, VERY quickly. Modern “run-of-the-mill” digital computers can throw about 100 billion switches per second! That’s enough to write about 10 million pages of text in one second.

- The **input** is rarely a collection of 0s and 1s, 011110000110... It can be in the form of text that is converted to 0s and 1s or voltage that is “digitized” with an “analog-to-digital” converter. (Musical recording, for example.)
- The **output** is similarly rarely a collection of 0s and 1s. It can be 1D data (words, numbers), 2D data (functions, charts, pictures, music), 3D data (video, surfaces, time-evolving functions)...
- The **processing** or **computing** or **programming** part is what the bulk of the course is concerned with. The programming responds to the input in some pre-determined way. In some digital computers the programming is fixed, for example, a digital clock or a digital thermometer. In the most interesting cases, the program can be changed. In this course we will learn to program programmable digital computers.

How are digital computers programmed?

You can either start off the computer with the 0s and 1s in the right place, or, you use a programming language. Programming languages provide the prescription, the pre-determined steps that instruct the computer how to respond to the various input it gets. We will learn two languages in this course.

C++ is a very popular (one of the most popular and common) language, a general-purpose language that can be used for all aspects of computing. It is the best language to use if one wants to teach many of the basic operations that a computer can perform.

MATLAB is a language that is suited for scientific and engineering applications, designed for getting answers quickly, with the minimum of effort, with reasonable guarantee that the answer is correct. It also provides graphical output with a minimum of effort.

What this course is REALLY about?

The hidden agenda

This course is all about putting ideas into action. It is about taking understood notions of how things work and building something, developing a numerical solution for some specific problem. This is what engineering is all about. The course will try to put your knowledge of mathematics and physics to practical use. Some of the problems we may solve in this course...

- How can I program a computer to play a simple game?
- How far can a golf ball fly into a 50 mile/hour wind?
- How can one find the shortest distance between two points when the landscape is not flat?
- If I take 10,000 steps in random directions, how far, on average, will I be from where I started?
- How can I do calculus on a computer (integrate, differentiate, find maxima and minima)?
- Where should I place a radiator in my room so that I am warm and do not have to pay too much for the heat?
- How can I make a vacuum?

What this course is NOT REALLY about?***The ugly stuff***

- It is not primarily a course on computer language syntax. It may seem like it at times, but it really isn't. Learning syntax is just a means to an end.
- It is not really a course on operating systems, computer hardware, or the fine detail of how computers work. We will discuss this but only enough to get the job done.

Educational Objectives

Engineering 101, Introduction to Computers and Programming, is intended to give you an exposure to, and a chance to practice, algorithmic thinking. It does so by asking you to actually solve problems using algorithms. This may be different from other courses that you have taken, because we do not tell you exactly how to solve the assigned problems. Rather, we provide you with tools that will allow you to solve them and some examples of how problems are solved. But you must creatively assemble the solution yourself.

Note that neither C++ nor MATLAB languages *per se* are the keys to the course. We teach you these languages so that you will have a means to express algorithms for execution on a computer, but a great many other computer languages could have been selected to support our central purpose of getting you to think about solving problems through a set of procedural steps. Because programming languages such as C++ and MATLAB contain many constructs and demand great attention to detail, it is easy to lose sight of the true intellectual center of the course. But keeping a focus on algorithmic thinking will make writing programs less difficult and less time consuming.

The educational objectives of Engineering 101 include:

- To introduce students to algorithmic thinking in their approach to problem solving.
- To teach students to implement algorithms in ANSI C++.
- To teach students to implement algorithms in MATLAB.
- To have students apply their knowledge of elementary physics and calculus.
- To provide students with a foundation on which to base their later applications of computers in engineering.

Students who successfully complete Engineering 101 (receive a grade of C or better) should:

- Design algorithms to accomplish clearly specified tasks.

- Display knowledge of ANSI C++ syntax and semantics
- Display knowledge of MATLAB syntax and semantics.
- Successfully implement clearly stated algorithms in C++.
- Successfully implement clearly stated algorithms in MATLAB.
- Be prepared to engage in discipline-specific instruction in computing.

1.2 Problems

1. In the broad definition of “computer” in this chapter, how many computers can you find: On yourself? In your room? In a car? In your home?
2. A typical 3rd grader is a lot smarter than any computer. They can read, do simple math, play a musical instrument, memorize poetry and write stories. Generally, with sufficient motivation, they will also do as they are told. How many ways could you instruct a 3rd grader to determine the value of π without the aid of a computer or calculating device?

Chapter 2

Data representations

2.1 Bits, nibbles, bytes and words

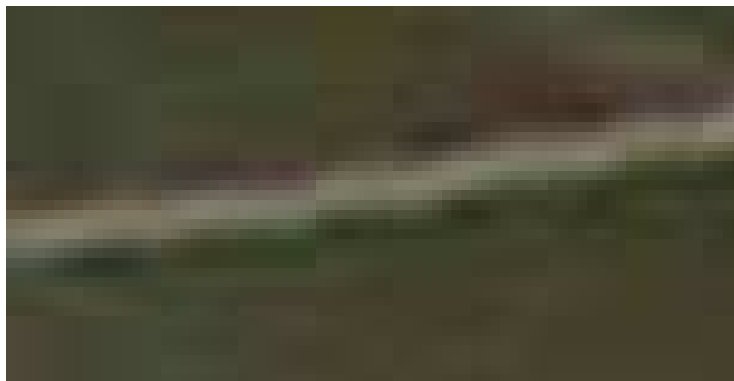
bit The digital computer processing hardware is made up of transistors, each of which can exist in only one of two possible states, a “state” of “high voltage” or “low voltage”. This state of high/low (or on/off) is represented abstractly as 1/0 (one/zero). A “bit” is some piece of microscopic (usually!) hardware that can exist in either of these two states as determined by a programmer or computer designer. In addition to the voltage states in a transistor in a computer’s memory chips or processing chips, there are many others forms: magnetic domains in a floppy disk, hard disk or magnetic tape, “pits” in a CD or DVD, holes in paper tape or computer cards. Nowadays, there are billions or trillions of bits of various forms in a typical computer. A bit may be simple, but it is usually so small, typically well below a micrometer across (1 micrometer = 10^{-6} m), that we can conceive of cramming billions of them into a relatively small volume of space. Some bits, particularly those in the processing chips of a computer, can also be made to change states very rapidly, well under one nanosecond. (1 nanosecond = 10^{-9} s.) So, we can do a lot of different things with lots of bits in a very short span of time.

nibble A “nibble” is a collection of 4 bits. A nibble can be in one of $2^4 = 16$ separate combinations of bits.

byte A “byte” is a collection of 2 nibbles, or 8 bits. A byte can be in one of $2^8 = 256$ separate combinations of bits. This is enough to, for example, associate with each unique pattern one of the characters of the English alphabet (upper and lower case) plus all the common special characters we may use, for example, in writing English prose or a technical document, *e.g.* ‘!\$&(),;:“”’?. And, there would be lots of room left over to make other interesting associations.

word A “word” is a collection of bytes—typically 4 for $2^{32} = 4,294,672,296$ possible combinations. We can do a lot with all the different representations of “words”. We will see, later in the lecture, how whole numbers can be associated with the various word patterns.

The notion that many bits can conspire, in very large numbers, to produce wonderfully intricate objects is illustrated in the following pictures of my dog, Matilda Skip Bielajew II. (Yes, that is her real name!) The first picture was generated from a file that was about 12 Megabytes, containing 96 million bits. It looks like real life. Yet, even when we zoom a little, we start to see the effects of the underlying bit structure. Look at how her whiskers appear jagged. Under extreme magnification (one of her whiskers, again) we see that the picture is made up of flat, uniformly colored “tiles”. Each one of these tiles is represented by 24 bits, 8 bits each for red, green and blue intensity, that, to all appearances, affords an apparently uniform transition of color from tile to tile.



2.2 4-bit Binary, Hexadecimal and Decimal Digits

4-bits of data (a “nibble”) can represent or be represented by several different things. It is common for computer programmers to use “hexadecimal” notation (0, 1, 2...9, A, B...F) as a “shorthand” for (0000, 0001, 0010...1001, 1010, 1011...1111). See the following table.

However, collection of a certain number of bits has to be given an “interpretation”—one that is sensible to humans. One such assignment (which is completely arbitrary and is decided upon by the computer programmer or designer) is to assign “decimal” numbers to the strings of bits. Such an assignment could look like:

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

We will use this particular representation in doing some simple arithmetic in the exercises and the assignments.

2.3 Binary arithmetic

Let us pretend that we have to do some arithmetic (additions and multiplications) using only binary numbers. The numbers 2,3,...9 do not exist! Only 0 and 1 exist. We also assume that the operations of addition and multiplication are *commutative* (*i.e.* independent of the order of the mathematical operation).

The addition of “fictional” binary numbers—base-2 arithmetic looks like:

$0 + ? = ? + 0 = ?$
 $1 + 1 = 10$
 $10 + 1 = 1 + 10 = 11$
 $11 + 1 = 1 + 11 = 100$
 $10 + 10 = 10 + 10 = 100$
 $\cdot$
 $\cdot$
 $1111 + 111 = 111 + 1111 = 10110$
 $\cdot$
 $\cdot$
 $\cdot$

The multiplication of “fictional” binary numbers—base-2 arithmetic looks like:

$0 \times ? = ? \times 0 = 0$
 $1 \times ? = ? \times 1 = ?$
 $10 \times 10 = 10 \times 10 = 100$
 $10 \times 11 = 11 \times 10 = 110$
 $11 \times 11 = 1001$
 $\cdot$
 $\cdot$
 $\cdot$
 $1111 \times 111 = 111 \times 1111 = 1101001$
 $\cdot$
 $\cdot$
 $\cdot$

The arithmetic procedure is analogous to the more familiar base-10 system (that arose simply because of our anatomy—10 fingers). Additions of columns of numbers, subtraction, multiplication, the procedure of long division, can be accomplished by analogous procedures. There is no limit to how long a “fictional” base-2 number can be and we may have to carry along some extra information with each number—its sign.

2.4 Converting from binary to decimal

2.4.1 Whole numbers

Humans are usually quite awful at doing binary arithmetic. If you are doing such a calculation, you can check by converting the binary numbers into their decimal equivalents.

In general, to convert a whole binary number to a whole decimal number, consider a binary number of the form:

$$b_n b_{n-1} b_{n-2} \cdots b_2 b_1 b_0,$$

where the b_i 's are either 0 or 1. We compute a decimal number using the following formula:

$$d_{10} = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} \cdots b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0.$$

For example, the result of the two more difficult calculations above can be checked as follows:

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 2^4 + 2^2 + 2^1 = 16_{10} + 4_{10} + 2_{10} = 22_{10}$$

is the result of one of the additions $[1111_2(15_{10}) + 111_2(7_{10})]$ above, while

$$1101001_2 = 2^6 + 2^5 + 2^3 + 2^0 = 64_{10} + 32_{10} + 8_{10} + 1_{10} = 105_{10}$$

is the result of one of the multiplications $[1111_2(15_{10}) \times 111_2(7_{10})]$ above. Note that the subscript "2" indicates a binary or base-2 number while the subscript "10" indicates a decimal number.

2.4.2 Real numbers

To convert a real(non-whole) binary number to a real decimal number, consider a binary number of the form:

$$b_n b_{n-1} b_{n-2} \cdots b_2 b_1 b_0 . b_{-1} b_{-2} \cdots b_{-(m-2)} b_{-(m-1)} b_{-m},$$

where the b_i 's are either 0 or 1 and a period, ".", separates the whole part, b_i where $i \geq 0$ from the fractional part, b_i where $i < 0$. We compute a real decimal number using the following formula:

$$r_{10} = b_n \times 2^n + b_{n-1} \times 2^{n-1} \cdots b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} \cdots + b_{-(m-1)} \times 2^{-(m-1)} + b_{-m} \times 2^{-m}.$$

It is not a whole lot different from the decimal numbers you are used to. However, sometimes the formulae can look a little messy.

2.5 Binary integer arithmetic on computers

Computers, because of their comprehension of data in binary terms, naturally do their arithmetic using the base-2 representation. The circuitry within a computer is most easily set up this way. However, when it comes to base-2 arithmetic and computers, there is a subtle difference! For practical reasons, computers must deal with integers that are finite in length. 16 bits, representing 65,536 separate possibilities, used to be the norm. Today,

32 bits is common (4294967296 possibilities). Soon, 64 bits (about 18×10^{18}) will be the standard. Some computers nowadays permit hardware 64-bit integer arithmetic.

Let us consider 32-bit computers only—the most common type of hardware available today. In the course of doing an arithmetic calculation, any bits that are set beyond the 32-bit boundary are lost—they fall into the “bit bucket” and disappear.

2.5.1 Two’s-complement integer arithmetic

A string of 32-bits can represent many things but we will focus on hexadecimal, unsigned and signed integers. These signed integers must also carry information as to the sign of the number. Historically, there have been several ways to do this. The “industry” has “settled” on “two’s-complement arithmetic” because it simplifies the process of adding positive, negative or mixed numbers. In this scheme, to negate a signed integer, one takes the complement (change all the 1s to 0s and all the 0s to 1s) of the binary representation and adds 1 to it. This is seen in the table at the end of this chapter.

Let’s give the notation $\bar{i}$ to the complement of the integer i . So, for any i , $\bar{i} + i = 11111111111111111111111111111111$. That is:

$$\begin{array}{r} \text{any 32-bit pattern} \\ + \text{ complement of the above} \\ \hline = 11111111111111111111111111111111 \end{array}$$

and

$$\begin{array}{r} 11111111111111111111111111111111 \\ + 000000000000000000000000000001 \\ \hline = 000000000000000000000000000000 \end{array}$$

since the 33rd bit can not be set! In two’s-complement arithmetic the expression $i - j$ is calculated as $i + \bar{j} + 1$. So you see, computers can not really add and subtract, they can only add (and take complements)!

A few examples help illustrate the simplicity of the two’s-complement scheme. Consider a 4-bit representation of the number $3_{10} = 0011_2$ and $4_{10} = 0100_2$. There are two tricks you should apply here: a) every time you see a negative sign in front of a bit pattern, i , to be used in a calculation, convert it to a positive sign, change i to $\bar{i} + 1$ and do the binary addition, and, b) if a final result has a leading order (leftmost) bit set to 1, it is a negative number, so, extract the minus sign and convert the resultant bit pattern j to $\bar{j} + 1$.

$$3_{10} + 4_{10} = 0011_2 + 0100_2 = 0111_2 = 7_{10}$$

$$3_{10} - 4_{10} = 0011_2 - 0100_2 = 0011_2 + 1100_2 = 1111_2 = -0001_2 = -1_{10}$$

$$-3_{10} - 4_{10} = -0011_2 - 0100_2 = 1101_2 + 1100_2 = 1001_2 = -0111_2 = -7_{10}$$

See how easily it all worked out! This is why the two's-complement approach has been adopted. In order to handle negative integers one can convert them to their two's-complement counterparts and add. Otherwise, special circuitry would have to be developed for handling negative numbers and that would be less efficient.

2.6 32-bit Binary, Hexadecimal, Unsigned and Signed Integers

32-bit computers can represent $2^{32} = 4,294,967,296$ different things but ONLY 4,294,967,296 different things. Now we consider the conventional assignments to “unsigned” and “signed” integers. There is also the hexadecimal representation that is associated with every 4-bit sequence, as indicated in the table below.

The unsigned integers associate the decimal number 0 to a string of 32 0 bits. The rest are formed by the addition by 1 in either binary or decimal. The maximum unsigned integer, therefore, is 4,294,967,295. Adding a one to this causes all the 32 bits to go back to zero, exactly as if you had “cycled” the odometer in your car! It's the same principle. There is no 33rd bit, so you can imagine it just falls off the end.

The signed integers have the same sequence as the unsigned integers up to $2^{31} - 1 = 2147483647$ which is the bit pattern 01111111111111111111111111111111. Adding a one to this gives a bit pattern of 10000000000000000000000000000000 which is interpreted as $-2^{31} = -2147483648$. Note that these two are complements of each other and that they add up to

11111111111111111111111111111111, which has the interpretation “-1”, consistent with the “two's-complement” scheme described above.

Binary	Hexadecimal	Unsigned Int	Signed Int
00000000000000000000000000000000	00000000	0	0
00000000000000000000000000000001	00000001	1	1
00000000000000000000000000000010	00000002	2	2
00000000000000000000000000000011	00000003	3	3
00000000000000000000000000000100	00000004	4	4
.	.	.	.
.	.	.	.
.	.	.	.
00000000000000000000000000000111	0000000F	15	15
00000000000000000000000000001000	00000010	16	16
00000000000000000000000000001001	00000011	17	17
.	.	.	.
.	.	.	.
.	.	.	.
01111111111111111111111111111110	7FFFFFFE	2147483646	2147483646
01111111111111111111111111111111	7FFFFFFF	2147483647	2147483647
10000000000000000000000000000000	80000000	2147483648	-2147483648
10000000000000000000000000000001	80000001	2147483649	-2147483647
.	.	.	.
.	.	.	.
.	.	.	.
11111111111111111111111111111100	FFFFFFFC	4294967292	-4
11111111111111111111111111111101	FFFFFFFD	4294967293	-3
11111111111111111111111111111110	FFFFFFFE	4294967294	-2
11111111111111111111111111111111	FFFFFFF	4294967295	-1

2.7 Problems

1. **octal + octal = octal**

Note: An “octal” is a number represented by only 3 bits.

Consider a fictitious computer, one that can only do 3-bit binary arithmetic. (Anything past the 3’rd bit can not be represented. See the notes for Lecture 2 for some related examples.) Complete the following addition table, that is, add the octal in i ’th row to the octal in the j ’th column and put the resultant octal in the box located at the i ’th row and j ’th column. There are 3 examples worked out for you. For these examples, show your calculations in long-hand.

	000	001	010	011	100	101	110	111
000								
001								
010								
011								
100								
101						010		
110								
111					011			110

2. **octal $\times$ octal = octal**

This is a similar exercise to the above except that you are to multiply the octals. Be sure to show your work for the 3 examples given.

	000	001	010	011	100	101	110	111
000								
001								
010								
011								
100								
101						001		
110								
111					100			001

3. Bits, nibbles and integers. Is it a hex?

Your imaginary computer represents integers using 4 and only 4 bits! The result of an arithmetic operation with two integers always results in a 4-bit result. Higher-order bits are “lost”.

The following table shows the binary, hexadecimal, unsigned integer, and signed integer representations for **all** the possible bit patterns that can be represented by your computer.

Binary	Hexadecimal	Unsigned Int	Signed Int
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	a or A	10	-6
1011	b or B	11	-5
1100	c or C	12	-4
1101	d or D	13	-3
1110	e or E	14	-2
1111	f or F	15	-1

Do your calculations in binary arithmetic and **SHOW YOUR WORK!** Quote your results of your calculations in binary, integer, unsigned and hexadecimal integer representations.

- What is the *sum* of the integers 2 and 3?
- What is the *sum* of the integers -2 and -3?
- What is the *sum* of the integers -8 and -8?
- What is the *product* of the integers 2 and 3?
- What is the *product* of the integers -2 and -3?
- What is the *product* of the integers -8 and -8?

4. **The interbreeding hex:** $\text{hex} + \text{hex} = \text{hex}$

Consider a fictitious computer, one that can only do 4-bit binary arithmetic. (Anything past the 4'th bit can not be represented. See the notes for Lecture 2 for some related examples.) Complete the following addition table, that is, add the hex in i 'th row to the hex in the j 'th column and put the resultant hex in the box located at the i 'th row and j 'th column. Some of the answers are given to you. Fill in the empty boxes. Remember that addition commutes. Therefore the actual work is halved and you need not do the boxes that are filled with X's.

[illegible]

5. Oh, hex, won't it ever stop: $\text{hex} \times \text{hex} = \text{hex}$

This is a similar exercise to the previous one except that you are to multiply the hexes.

[illegible]

6. A fistful of (binary) digits

Your imaginary computer represents integers using 5 and only 5 bits! Fill in the blank spots in the following table. The signed integers follow the two's-complement rule discussed in class.

Binary	Hexadecimal	Unsigned Int	Signed Int
00000	00	0	0
00001	01	1	1
00010	02	2	2
00011	03	3	3
00100	04	4	4
00101	05	5	5
00110	06	6	6
00111	07	7	7
01000	08	8	
01001	09	9	
01010	0A	10	
01011	0B	11	
01100	0C	12	
01101	0D	13	
01110	0E	14	
01111	0F	15	
10000	10	16	
10001	11	17	
10010	12		
10011	13		
10100			
10101			
10110			
10111			
11000			
11001			
11010			
11011			
11100			
11101			-3
11110			-2
11111			-1

7. These digits will give you the fidgets.

Your imaginary computer still represents integers using 5 and only 5 bits! The result of an arithmetic operation with two integers always results in a 5-bit result. Higher-order bits are “lost”.

Do your calculations in binary arithmetic and **SHOW YOUR WORK!** Quote your results of your calculations in binary, signed integer (2s-complement scheme), unsigned and hexadecimal integer representations. You may use the table that you completed in the previous question.

- (a) What is the *sum* of the unsigned integers 3 and 4?
- (b) What is the *sum* of the unsigned integers 16 and 17?
- (c) What is the *sum* of the signed integers -2 and -3?
- (d) What is the *sum* of the signed integers -16 and -16?
- (e) What is the *product* of the unsigned integers 3 and 4?
- (f) What is the *product* of the signed integers -2 and -3?
- (g) What is the *product* of the signed integers -16 and -16?

8. Bits and bytes? Again!

In the following table, the column 1 represents bit patterns on a 6-bit computer.

- (a) In the 2'nd column, write in the complement of the bit pattern in column 1
- (b) In the 3'rd column, write in the two's complement of the bit pattern in column 1
- (c) In the 4'th column, write in the two's complement of the bit pattern in column 3
- (d) In the 5'th column, put a check mark if the bit pattern in column 1 represents a positive signed `int`.
- (e) In the 6'th column, put a check mark if the bit pattern in column 1 represents an odd `unsigned int`.
- (f) In the 7'th column, put a check mark if the bit pattern in column 1 represents an `unsigned int` that is divisible by 2.
- (g) In the 8'th column, put a check mark if the bit pattern in column 1 represents an `unsigned int` that is divisible by 4.
- (h) If the bit pattern in column 1 represents an odd `unsigned int`, write its `unsigned int` representation in column 9.

Column 1	Column 2	Column 3	Column 4	Col 5	Col 6	Col 7	Col 8	Column 9
000010								
111001								
110110								
000001								
001101								
110010								
010010								
110110								
101111								
101010								

9. A bit challenging

In the following table:

- 1) Convert the first 5 32-bit bit patterns to hex
- 2) indicate which bit patterns are divisible by 2 (/2)
- 3) indicate which bit patterns are divisible by 4 (/4)
- 4) indicate which bit patterns represent positive or negative integers in two's-complement representation (+/-?)
- 5) indicate which bit pattern represents the largest integer in two's-complement representation (>?)
- 6) indicate which bit pattern represents the smallest integer in two's-complement representation, negative with the greatest magnitude (<?)

32-bit bit patterns	hex	/2?	/4?	-/+?	>?	<?
1111 0001 0010 0110 1110 0101 0100 0110						
1000 0010 0000 1100 1000 0010 1011 1011						
1011 0001 1000 1011 0100 1111 0000 0111						
0010 0001 1101 1001 0100 1101 0111 1110						
0010 1000 1010 0001 0010 1010 0101 0111						
1110 1100 1000 1000 0011 0101 1110 1101	X					
1111 0111 0000 1100 0011 1010 0010 1101	X					
0110 0111 0100 0000 0101 1000 1011 0000	X					
0011 1110 0000 0011 1100 0110 0000 1101	X					
0000 1110 1101 1001 1111 1011 0101 1000	X					
0110 1111 0001 1010 0010 1111 0110 1000	X					
1010 1000 0000 1110 1111 1010 0010 0011	X					
0000 0010 0011 0010 0100 1100 0001 0101	X					
0011 1100 1010 0111 1011 1011 0111 0000	X					
1100 0011 0100 1011 0110 1011 1111 0011	X					
1101 0001 1011 1110 1011 1010 0000 0001	X					
1101 0011 1110 1011 0100 0101 0100 1111	X					
1111 0011 0001 0111 1110 1110 0011 0000	X					
0100 1101 1110 1110 0000 0011 1011 1011	X					
0111 1011 1010 1111 0100 0010 0010 0101	X					
1101 1001 1101 1111 1101 1101 1010 1101	X					
1000 1101 1010 0101 0110 1010 1111 0110	X					
0100 0100 0110 1111 0001 0010 0111 0001	X					
0011 1110 1010 1100 1100 0111 0010 0111	X					
1110 1111 0111 1100 1001 0101 0001 1101	X					
1001 1000 1010 0101 1000 0010 1011 1010	X					
1011 1111 1000 0100 0011 0101 0011 0111	X					
1001 1010 0000 0011 1111 1010 0110 1111	9A03FA6F					

Chapter 3

Algorithms and Pseudocodes

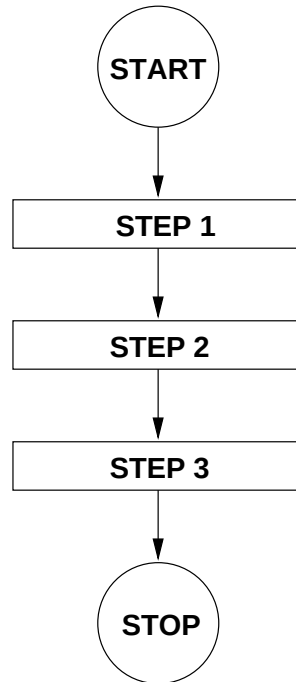
In this chapter we introduce algorithms and ways with which algorithms can be represented. At the end of the chapter, we also talk a little about computer architecture, as a way of illustrating the operations that can happen inside a computer, when a part of an algorithm is executed.

3.1 What is an algorithm?

An algorithm is a set of well-defined, unambiguous *steps* or *instructions* to be carried out in sequence to accomplish some task.

- An algorithm usually has a **START** point and a **STOP** point. Certainly any algorithm we shall encounter in this book will. The **START** may require some *inputs* and the **STOP** may provide some *output*.
- An algorithm is made of individual instructions.
- An instruction consists of a well-defined operation usually on some input (what the instruction receives) and usually producing output (what the instruction produces).
- Each instruction is well-defined and its outcome predictable if the instruction operates on valid input.
- There is a direction of *logic flow* or *sequencing*. Once an instruction is executed, it passes control to another instruction.
- There can only be a finite set of instructions.
- When executed (with valid input) an algorithm is guaranteed to terminate in a sensible way.

3.2 Flowchart representation



The above figure is a representation of a simple sequence of 3 steps with a beginning and an end. The **START** and **STOP** are represented by circles. A sequence step is represented by a rectangle. The logic flow is indicated by lines and arrows. This kind of representation is called a *flowchart* or *schematic diagram*. It looks a lot like a circuit diagram in electronics.

3.3 Pseudocode representation

An algorithm is a well-defined description of actions and their ordering. *Pseudocode* is another way of expressing the actions and ordering to be taken place on a computer in an exact way. Yet, pseudocode does it in an informal way, without getting bogged-down in the details or syntax of a particular computer language like C++, MATLAB or something else. Two programmers should be able to agree on pseudocode and write functionally equivalent computer programs—either in the same computer language or in different ones. Pseudocode can be written in any human language in the form of an ordered (or numbered) list. For example, a pseudocode representation of the above flowchart would be:

START $\implies$ Step 1 $\implies$ Step 2 $\implies$ Step 3 $\implies$ STOP

A more conventional representation has the instruction flow (also called logic flow) going from top to bottom:

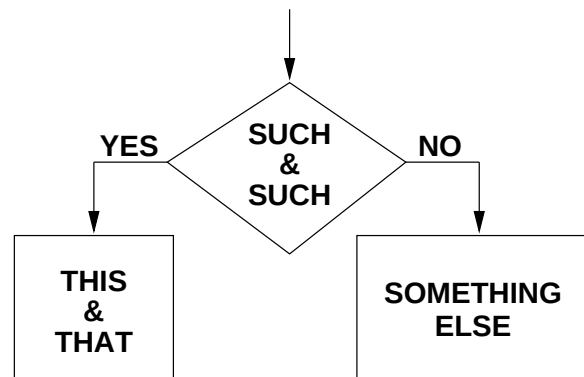
START
Step 1
Step 2
Step 3
STOP

3.4 Decisions, conditionals, branch instructions

A “conditional” instruction or statement, also called a decision instruction, a branch or decision statement, is one in which a decision is made and the logic flow can follow different paths, depending on the answer. In pseudocode it can be represented as follows:

IF *such-and-such* **THEN DO**
 this-and-that
OTHERWISE DO
 something-else

One could do the same thing with a flowchart, and it would look like:



Note that we have introduced a new shape to handle the case where, depending on the input, a decision is made and the logic flow can follow two different paths. This is represented by a “diamond shape”. A branch statement or a *decision statement* is characterized by only one path in and always two paths out.

3.5 Looping or jump instructions

A final ingredient is needed before we complete our abstract discussion of algorithms —the ability for a statement to transfer control not to the next statement, but somewhere else in

the algorithm. This is called a “looping” or “jump” instruction.

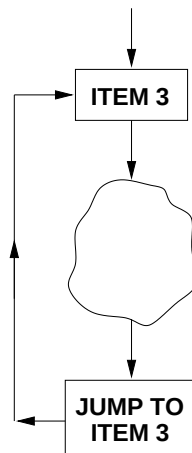
Consider the following pseudocode:

ITEM 3

•
•
a set of other steps
•
•

JUMP TO: ITEM 3

The stuff between **ITEM 3** and **JUMP TO: ITEM 3** inclusive, defines a *loop*. This is represented in a flowchart as follows with the direction of logic flow made clear with the lines and arrows.



3.6 Sequencing, branching and looping

It *can be shown* that the 3 constructs of sequencing, branching and looping, are *all that one needs to construct any computer algorithm!* Thus, the “theory” of algorithmic flow boils down to understanding just these three things. There are many, many ways of expressing these three things in a computer language like C++ or Matlab. But remember that it all boils down to just sequencing, branching and looping. These are **the three pillars of algorithmic design!**

Of course there is a lot of other stuff to be considered to make a computer language useful. We’ll see a lot of this in the course. However, as far as the flow of logic goes, we have done it all at this point! You can now go and write an algorithm to do anything you want and express it in either pseudocode or flowchart form.

3.7 A mini-summary before the examples

PSEUDOCODE: The language and organization of pseudocode is not firmly established. Make any convention for yourself that makes your algorithm clear. Note the use of spacing and indenting, bold and italic fonts, all ways of emphasizing meaning.

FLOWCHARTS: It is also possible to write pseudocode diagrammatically in terms of flow charts that represent the flow of computer instructions, or “logic flow”. Here there are some accepted (but not rigorously adopted) standards. These charts look very much like circuit diagrams. It is considered good practice to describe every variable you use and provide a well-defined starting point and ending point.

Pseudocode and flow charts are indispensable aids in getting a computer program “up and running” with a minimum of effort.

3.8 Some examples

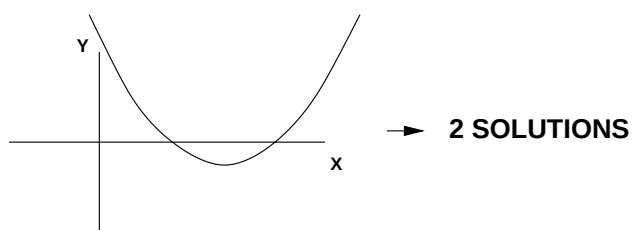
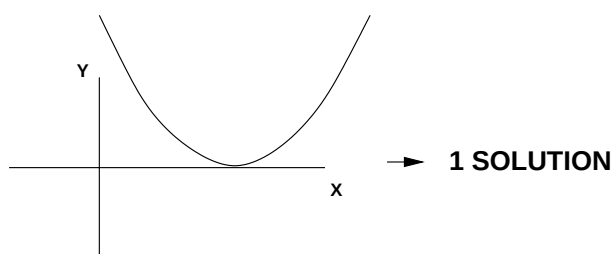
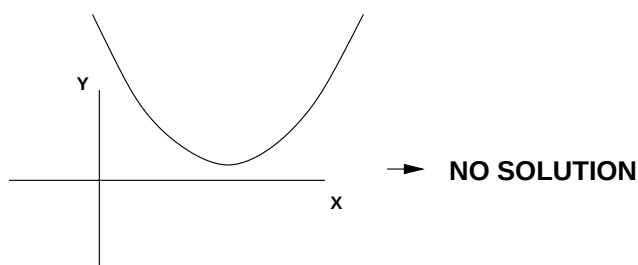
3.8.1 The Breakfast algorithm, or, Bielajew's Sunday Morning Internationally-famous pancakes

1. Start of The Breakfast Algorithm
2. If the day is not Sunday, have your usual boring bowl of Granola. For added excitement, slice in half a banana. Otherwise,
 - (a) In a big bowl, whisk together 1 and 1/2 cups white flour and 2 heaping teaspoons of baking powder.
 - (b) In a separate bowl, beat two whole eggs (no shells please) together for one minute or so.
 - (c) Add 1 cup of milk, preferably skimmed, to the eggs and mix.
 - (d) Add 1 tablespoon of Canola oil, 3 tablespoons of maple syrup, and one teaspoon of vanilla to the milk and eggs and mix.
 - (e) Add the wet stuff to the dry stuff.
 - (f) Take a big spoon and,
 - (g) Mix the stuff in the big bowl one whole turn.
 - (h) If there are lots of lumps in the mixture, go back to step 2.(g), otherwise go to step 2.(i). *Technical note: Don't go overboard and make it completely smooth. Leave it a little lumpy for texture.*
 - (i) Slice in a Granny Smith apple (no peel) and (optionally) a half cup of frozen cranberries (well rinsed).
 - (j) Add a little bit of Canola oil to an electric fry pan preheated to 350°F. Put in just enough to coat the bottom of the pan.
 - (k) Put the pancake dough into the pan, making approximately four 3–4" diameter circles. Turn the heat down to 300°F. Fry for exactly 3 minutes with the pan *covered*, flip and fry for 3 more minutes with the pan *uncovered*.
 - (l) Take them out of the frying pan and place on a napkin to absorb excess oil.
 - (m) Eat with maple syrup and sliced strawberries on top. I DARE you to eat more than 3! (It has never been done...)
3. Clean up your dishes.
4. End of The Breakfast Algorithm

This algorithm produces about 8 pancakes, enough for 4–6 people.

3.8.2 Solve for x : $Ax^2 + Bx + C = 0$ where A, B, C are arbitrary constants

Definitions: x is an unknown “real” number, A, B, C are fixed constants that can take on any valid real value. We are trying to find the values of x for which $Ax^2 + Bx + C$ is zero. If we plot the function $f(x) = Ax^2 + Bx + C$, a parabola, it can look as shown in the figure below, for certain choices of A, B , and C . The x -axis can intersect the curve $f(x) = Ax^2 + Bx + C$ only 0, 1 or 2 times, depending on the values of A, B , and C .



First, we do the math in excruciatingly fine detail. It is essential to understand completely the mathematics of what you are doing if you are to have a chance at writing a successful algorithm.

$$\begin{aligned}
 Ax^2 + Bx + C &= 0 \\
 A\left(x^2 + \frac{B}{A}x + \frac{C}{A}\right) &= 0 \quad \text{N.B.} \quad A \neq 0 \\
 x^2 + \frac{B}{A}x + \frac{C}{A} &= 0 \\
 x^2 + \frac{B}{A}x + \frac{B^2}{4A^2} - \frac{B^2}{4A^2} + \frac{C}{A} &= 0
 \end{aligned}$$

$$\begin{aligned}
x^2 + \frac{B}{A}x + \frac{B^2}{4A^2} &= \frac{B^2}{4A^2} - \frac{C}{A} \\
x^2 + \frac{B}{A}x + \frac{B^2}{4A^2} &= \frac{B^2 - 4AC}{4A^2} \\
\left(x + \frac{B}{2A}\right)^2 &= \frac{B^2 - 4AC}{(2A)^2} \\
x + \frac{B}{2A} &= \pm \frac{\sqrt{B^2 - 4AC}}{2A} \quad \text{N.B. } B^2 - 4AC \geq 0 \\
x &= -\frac{B}{2A} \pm \frac{\sqrt{B^2 - 4AC}}{2A} \\
x &= \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \quad A \neq 0, B^2 - 4AC \geq 0
\end{aligned}$$

So, under some circumstances, ($A \neq 0, B^2 - 4AC > 0$), we have two solutions:

$$x = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \quad \text{and} \quad x = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

corresponding to the last case in the preceding figure.

What if $A \neq 0, B^2 - 4AC = 0$? We have one solution:

$$x = \frac{-B}{2A}$$

corresponding to the middle case in the preceding figure.

What if $B^2 - 4AC < 0$? *No real solution.* This corresponds to the first case in the preceding figure.

What if $A = 0$? Re-analyze!

$$\begin{aligned}
Bx + C &= 0 \quad (y = Bx + C \text{ Is an equation for a straight line!}) \\
x &= \frac{-C}{B} \quad \text{N.B. } B \neq 0
\end{aligned}$$

This is the place where a straight line crosses the x -axis.

What if $A = 0$ and $B = 0$? *No solution for x ! A straight line with zero slope does not intercept the x -axis.* The equation is nonsense unless $C = 0$ as well. If $A = B = C = 0$ it is correct but not usually interesting!

Now we give the pseudo-code for this example, the solution to the quadratic equation.

START A, B, C are inputs, known to be real, fixed constants and x is the output, expected to be real but unknown at the start

```

IF ( $B^2 - 4AC < 0$ )
    PRINT (No solution because  $B^2 - 4AC < 0$ )
    STOP

ELSE (i.e.  $B^2 - 4AC \geq 0$ )
    IF ( $A \neq 0$ )
        IF ( $B^2 - 4AC = 0$ )
             $x = -B/(2A)$ 
            PRINT (One solution:  $x$ , because  $B^2 - 4AC = 0$ )
            STOP
        ELSE (i.e.  $B^2 - 4AC > 0$ )
             $x_1 = (-B + \sqrt{B^2 - 4AC})/(2A)$ 
             $x_2 = (-B - \sqrt{B^2 - 4AC})/(2A)$ 
            PRINT (Two solutions are:  $x_1$  and  $x_2$ )
            STOP
        ELSE (i.e.  $A = 0$ )
            IF ( $B \neq 0$ )
                 $x = -C/B$ 
                PRINT (Only one solution  $x$ , because  $A = 0$ )
                STOP
            ELSE (i.e.  $B = 0$ )
                PRINT (No solution because  $A = 0$  and  $B = 0$ )
                STOP

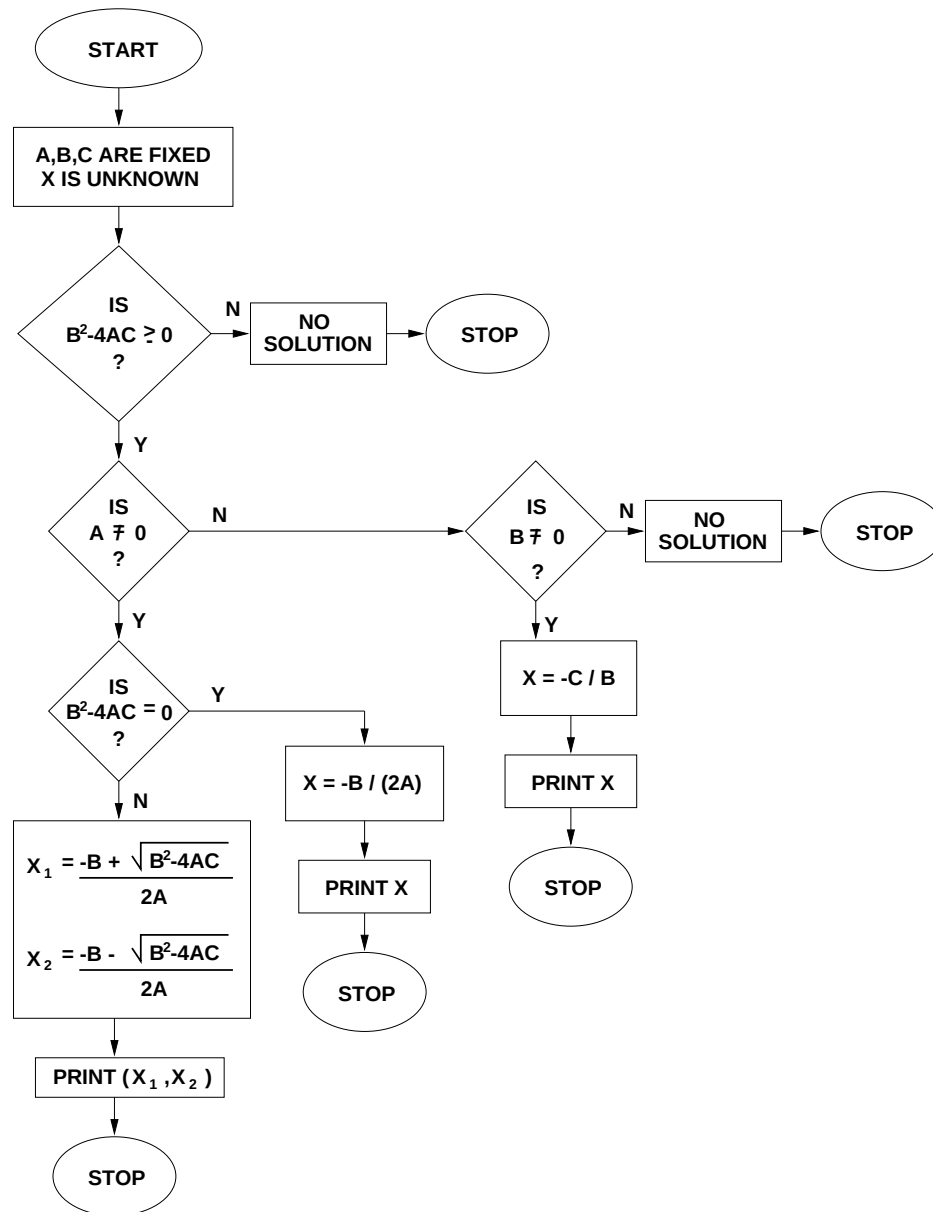
```

END OF ALGORITHM

Note:

- IF's can be nested
- Use of indenting
- Multiple STOP-points!
- Not every piece of pseudocode is used every time
- If enough different A , B , C 's are tried, every piece of pseudocode will be used
- It will work for any value of A , B , C . The algorithm is robust
- Can represent in terms of a graphical flow chart

This is the flow chart for the quadratic formula:



Note that, although there is no reason to expect it, the flowchart designer expects that there will be two real solutions every time. This is reflected in the flowchart by this case running vertically down the page, with the exceptions hanging off to the right. This makes the flowchart particularly easy to read.

3.8.3 Iteration: A summing loop

Before you read this section, if you have never had any experience with programming, the statement $S = S + 1$ should justifiably seem to be completely wrong, since mathematically it is equivalent to saying $0 = 1$! Just cancel the S 's from either side! So, if this is the case, kindly go and read the next section “An aside on computer architecture” and then come back. Although mathematical statements and computer code statement may look the same, they are completely different. Mathematics expresses an idea, while a line of computer code represents an *operation*, or a set of individual operations.

Problem: Sum the positive integers $1, 2, 3 \dots N$ up to N .

A dumb way that works:

START N is an integer (fixed) and S is an integer (unknown)

$S = 1$

$S = S + 2$

$S = S + 3$

.

.

.

$S = S + i$ (for the i 'th statement)

.

.

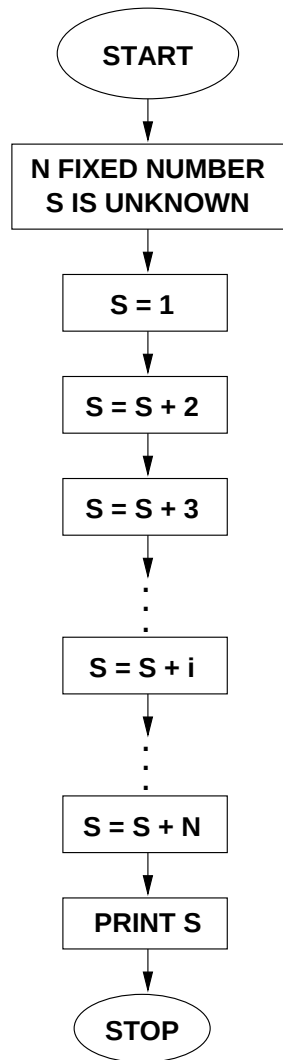
.

$S = S + N$

PRINT(S)

END OF ALGORITHM

This is the flow chart for the dumb way:



N statements—lots of repetition, very long if N is large.

A better way:

START N is an integer (fixed), S is an integer that will contain the sum. At the start, S is initialized to 1, that is, its starting value will be 1. Another integer, i , is employed to contain the value that is to be added to S . It is initialized to 1.

Start of counting loop:

- $i = i + 1$ (get the new value for i)
- $S = S + i$ (add it to S)
- **IF** $i = N$ jump to **End of counting loop:**

- Jump to the first statement in this loop

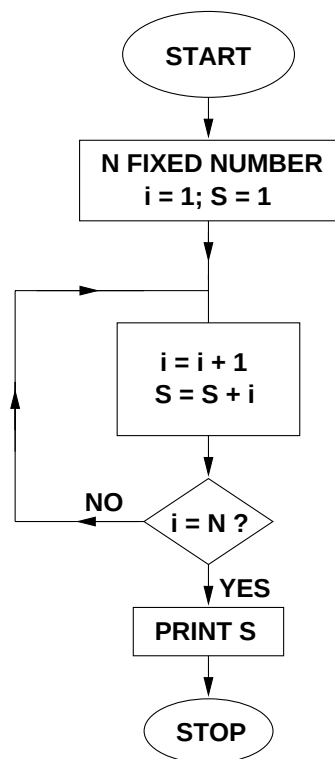
End of counting loop:

PRINT(S)

END OF ALGORITHM

- Only a few “statements” to express the same idea
- Loops can be nested as well
- Can represent in terms of a graphical flow chart

This is the flow chart for the better way:



There is an even better way! There is a mathematical solution to this that was discovered by Gauss (Karl Friedrich Gauss (1777–1855 AD)) as a schoolboy!

$$S = \frac{N(N + 1)}{2}$$

If N is even:

$$\begin{aligned}
 S &= [1, 2, 3, \dots, (N-2), (N-1), N] \\
 &= [1, 2, 3, \dots, (N/2)] + [(N/2+1), (N/2+2), (N/2+2), \dots, N] \\
 &= [1, 2, 3, \dots, (N/2)] + [N, (N-1), (N-2), \dots, (N/2+1)] \\
 &= [(N+1), (N+1), (N+1), \dots, (N+1)] \quad (N/2 \text{ terms}) \\
 &= \frac{N}{2}(N+1)
 \end{aligned}$$

quod erat demonstratum

For odd N the proof is left to the reader.

3.8.4 Iteration: A product loop

Problem: Compute $P = k^N$.

A very, VERY, dumb way (but one that works):

START k is an integer or a real (fixed), N is an integer (fixed) and P is an integer or a real number (unknown)

$P = k$

$P = P \times k$

$P = P \times k$

.

.

.

$P = P \times k$ (for the i 'th statement)

.

.

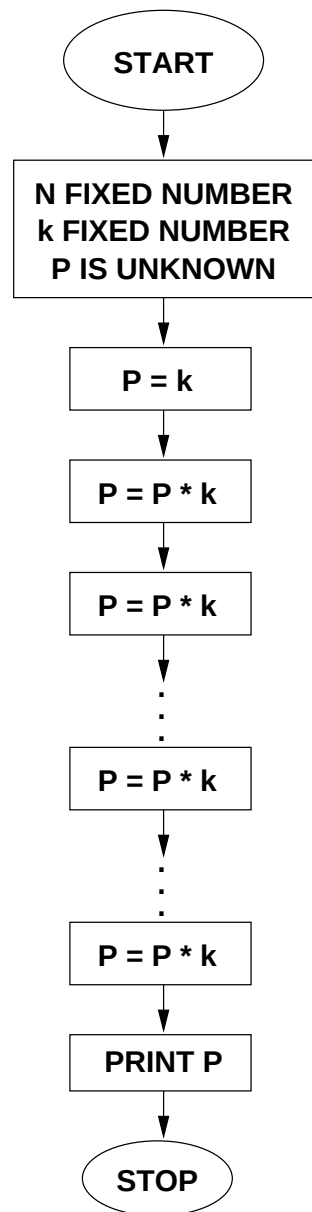
.

$P = P \times k$ (for the N 'th statement)

PRINT(P)

END OF ALGORITHM

Here is the flow chart for the very, VERY, dumb way:



N statements—lots of repetition of the same statement

A better (but still dumb) way:

START k is an integer or a real (fixed), N is an integer (fixed), $P = 1$ is a number that will contain the product (initialized to 1) and i is an integer (initialized to 0) used as an “index” or “counter” to indicate how many times we have passed through the loop.

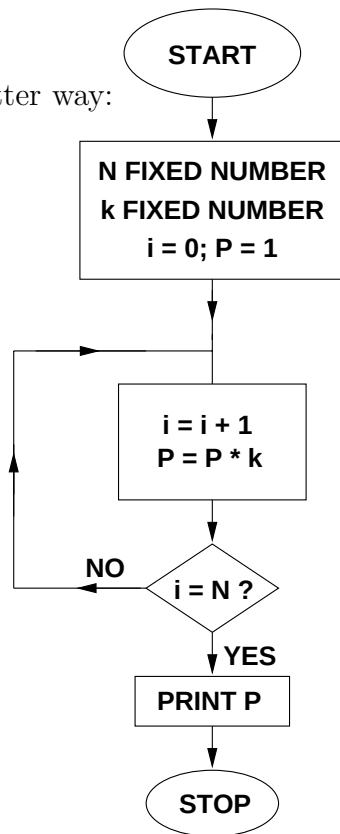
Start of loop:

- $i = i + 1$ (“increment” the counter)
- $P = P \times k$ (“accumulate” the sum)
- **IF** $i = N$ jump to **End of loop**:
- jump to **Start of loop**:

End of loop:
 PRINT(P)

END OF ALGORITHM

Here is the flow chart for the better way:



Only a few statements to express the same idea.

A much, MUCH better way: Al-Kashi (1390 - 1450 AD)

- al’Kashi, Ghiyath al’Din Jamshid Mas’ud
- Born in Kashan (Iran), died in Samarkand (now called Uzbekistan)
- “Self-proclaimed” inventor of decimal fractions
- Calculated π to 16 decimal places
- Theory of numbers and computation
- “First” discovered the binomial theorem
- Inventor of a calculating machine to predict lunar orbits
- Discovered the following algorithm in 1414 AD:

START k is an integer or a real (fixed), N is an integer (fixed), $P = 1$ is a number that will contain the product (initialized to 1)

Start of loop:

IF (N is even)

$N = N/2$

$k = k \times k$

ELSE (*i.e.* N is odd)

$N = N - 1$

$P = P \times k$

IF ($N < 1$) jump to **End of loop:**

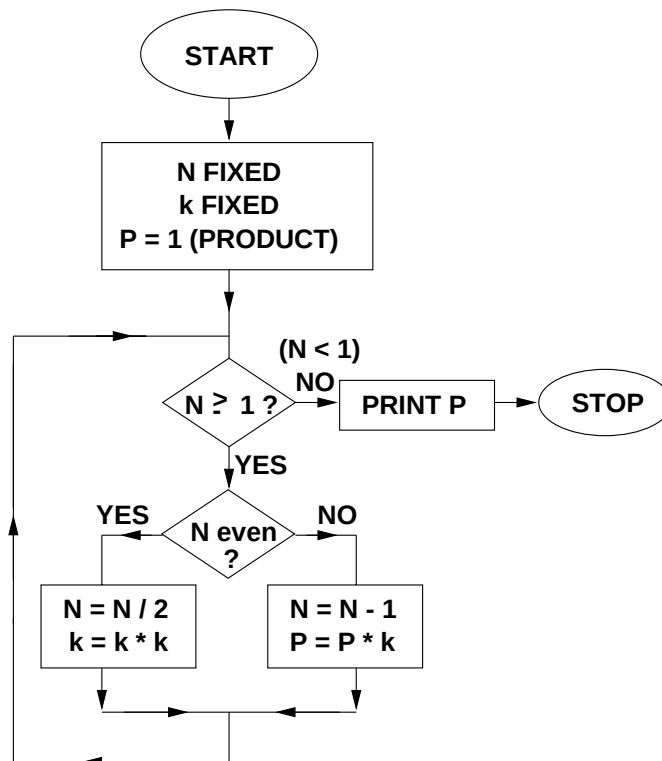
ELSE (*i.e.* $N \geq 1$) jump to **Start of loop:**

End of loop:

PRINT(P)

END OF ALGORITHM

This is the flow chart for the Al-Kashi Method:



Danger, danger: The algorithm redefines N and k .

3.9 An aside on computer architecture

3.9.1 What does $S = S + 1$ mean?

If you're doing math, it means nonsense! It is the same as saying $0 = 1$!

If you're writing a computer program it means something else and describing what it means requires a small “aside” on computer architecture, the real “nuts and bolts” of what is going on inside a computer.

First some definitions. See the figure below.

MEMORY The memory is the physical place in the computer where bit patterns are stored. Memory is best thought of as a stack of “words”, 32-bit bit patterns.

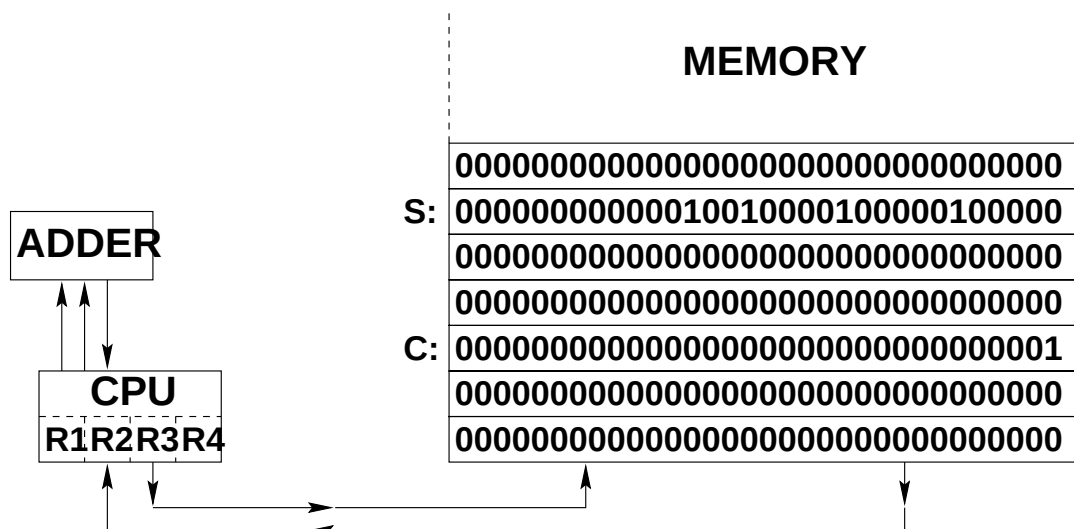
VARIABLE A symbol for a certain bit pattern (a 32-bit integer, say) that the programmer can define and change at will in a program. The “symbol” (S in this case) really refers to the location in memory where the bit pattern is stored.

CONSTANT A symbol for a certain bit pattern that the programmer can define and use but not change.

CPU The Central Processing Unit in a computer. The CPU is the “boss”. It follows the steps in a program and tells all the other parts of a computer what they should be doing. It is the “controller”.

REGISTER A special place in the CPU where bit patterns are stored.

ADDER A functional unit in a computer (usually located right on the CPU chip) that performs additions on 32-bit bit patterns.



In a computer program, the statement: $S = S + 1$ gets “translated” into the following operations that the CPU performs or controls.

FETCH S,R1 Copy the bit pattern that is stored at the location referred to by **S** and put it in the 1st CPU register **R1**.

FETCH C,R2 Copy the bit pattern that is stored at the location referred to by **C** and put it in the 2nd CPU register **R2**. Previously in the program, the constant **C** would have to have been defined and the bit pattern for **C** (00000000000000000000000000000001) would have to have been stored in **C**.

ADD R1,R2,R3 Copy the constants of the bit patterns stored in **R1** and **R2** into the **ADDER**, do the binary arithmetic and copy the result into the 3rd CPU register **R3**.

STORE R3,S make a copy of the bit pattern in the 3rd CPU register **R3** and place it in the memory location referred to by **S**.

So, just the simple increment (add by 1) of a variable results in 4 operations performed by the CPU. The preceding steps are exactly what an “assembly language” program would have to do to interpret the statement $S = S + 1$. Remember that when we write statements like $S = S + 1$ it is really a shorthand for a set of steps that a computer has to carry out. We will be clear in the future as to what is math and what is computer language. Math script will be used for math (*e.g.* $S = S + 1$) while type script will be used for computer instructions (*e.g.* **S = S + 1**).

3.10 Problems

1. The trinity of algorithmic design

What are the three most important capabilities that are needed to construct algorithms?

- _____ The ability to declare variables
- _____ The ability to assign values to variables
- _____ The ability to pass control from one operation to another in a prescribed way
- _____ A way of making a “True” or “False” decision
- _____ The existence of **int**’s and **float**’s
- _____ The ability to do mathematical operations (*e.g.* $+$, $-$, $\times$)
- _____ The ability to go back to a previous operation

_____ A “compiler” that can convert human-readable text into a language (such as binary) that a computer can understand

2. What is an algorithm?

In 35 words or less, answer the question, “What is an algorithm?”.

3. What is an algorithm?

The three “pillars” of algorithm design are 1) sequencing, 2) branching, and 3) looping.

In 25 words or less for each, describe what they mean.

That is:

What is sequencing? (25 words or less)

What is branching? (25 words or less)

What is looping? (25 words or less)

4. Toss your cookies at this one!

In this Chapter, a recipe for Bielajew’s Sunday Morning Internationally-famous pancakes was given—in the form of pseudocode. Describe another recipe for a dish that you like—in the form of pseudocode. It has to have a Start, a Stop, more than 5 Steps, at least one Decision (or branch) point, and at least one Loop.

5. In the 70’s, man, if you were not hip, you were ...

Here is an algorithm in pseudo-code:

```

START
OUTPUT “Enter a positive integer”
INPUT  $N$ 
 $i = 1$ 
WHILE ( $i \times i < N$ )
     $i = i + 1$ 
END WHILE
IF ( $i \times i = N$ )
    OUTPUT “YES”
ELSE
    OUTPUT “NO”
END IF
STOP

```

For which of the following inputs will the algorithm say “YES”?

(a) 28

(b) 36

(c) 54

(d) 81

What property of the input is the algorithm determining?

6. Summing the squares

Describe an algorithm in either pseudocode or flowchart form, to sum the squares of the integers up to and including some N , where N is an input to the algorithm. That is,

$$S = 1^2 + 2^2 + 3^2 + \cdots + N^2$$

where S is the sum. You must express your algorithm in terms of a loop that is traversed N times. (There is a closed mathematical expression $S = N(N+1)(2N+1)/6$ that you must not exploit in your algorithm. Pretend the closed mathematical expression does not exist.) Make sure to declare and initialize all the variables you use.

7. Don't you love triangles?

A number N is called “triangular” if you can take N little asterisks and make an equilateral triangle from them. For example, here are the first few triangular numbers:

*	*	*	*	*
* *	* *	* *	* *	* *
3	* * *	* * *	* * *	* * *
	6	* * * *	* * * *	* * * *
		10	* * * * *	* * * * *
			15	* * * * * *
				21

Here is an algorithm that is attempting to detect triangularity in the input. But it has a bug. Fix the algorithm.

```

START
OUTPUT "Enter a positive integer"
INPUT N
i = 1
S = 0
WHILE (S < N)
    i = i + 1
    S = S + i
END WHILE
IF (S = N)
    OUTPUT "Triangular"
```

```

ELSE
    OUTPUT "Not triangular"
END IF
STOP

```

8. De-Gauss this!

In this chapter we learned Mr. Gauss's formula for the computation of the N th triangular number:

$$1 + 2 + 3 + \dots + N = \frac{N(N + 1)}{2}$$

Here is another algorithm attempting to detect triangularity of the input. Do you think this one is OK? If not, fix it.

```

START
OUTPUT "Enter a positive integer"
INPUT N
i = 1
WHILE [(i × (i + 1))/2 < N]
    i = i + 1
END WHILE
IF [(i × (i + 1))/2 = N]
    OUTPUT "Not triangular"
ELSE
    OUTPUT "Triangular"
END IF
STOP

```

9. The oscillating sign

Write an algorithm in pseudocode that accepts a positive integer N and outputs a 1 (one) if N is even and -1 (minus one) if N is odd. *Hint: -1 multiplied by itself an even number of times is +1, while -1 multiplied by itself an odd number of times is -1.*

10. Summing an oscillating series

Describe an algorithm in both pseudocode and flowchart form, to perform the following sum

$$S = 1 - x + x^2 - x^3 + x^4 \dots + (-x)^N$$

where x and N are inputs and S is the sum, an output. You must express your algorithm in terms of a loop that is traversed N times. Express your algorithm in terms of pseudocode *and* a flowchart.

11. Let's talk about... the factorials of life

The factorial occurs in many places in mathematics and engineering applications and life in general. The symbol for the factorial is the exclamation (!) mark. So, we would call $3!$, “three factorial”. Factorials are easy to calculate.

$$1! = 1$$

$$2! = 2 \times 1$$

$$3! = 3 \times 2 \times 1$$

$$4! = 4 \times 3 \times 2 \times 1$$

and so on.

If N is a whole number,

$$N! = N \times (N - 1) \times (N - 2) \cdots 2 \times 1.$$

Note that $N! = N \times (N - 1)!$, a neat property of factorials.

You will describe an algorithm to calculate the factorial of N where N is any whole number greater than 1. You will describe an algorithm as if you were going to program it on a computer, that is, you will specify a set of instructions that even a machine can understand. The algorithm should include instructions on how to start and end. You can assume that the machine on which the algorithm will be implemented can store variables, loop, do branches, and do simple arithmetic. Express your algorithm in terms of pseudocode *and* a flowchart.

12. Decay and dissipation

You will describe an algorithm to calculate an approximation to the exponential function, e^{-x} . The following sum is a really good approximation to the exponential function, e^{-x} , as long as the right hand side contains enough terms.

$$S = 1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Let x and N , the number of terms on the right hand side, be inputs to the algorithm and S , the sum, be the output. Note the use of the factorials in the above sum. Express your algorithm in terms of pseudocode *and* a flowchart. Note that the algorithm to raise x to a power is given in Chapter 3, Section 5 of “the book”.

Hint: The most efficient way to solve this problem is to rewrite S in the following fashion:

$$S = s_0 + s_1 + s_2 + s_3 + s_4 + \cdots$$

where $s_0 = 1$ and note that

$$s_n = \left(\frac{-x}{n}\right) s_{n-1}$$

13. What's YOUR sine?

You will describe an algorithm to calculate the sine function, as if you were going to program it on a computer, that is, you will specify a set of instructions that even a machine can understand. The algorithm should include instructions on how to start and end. You can assume that the machine on which the algorithm will be implemented can store variables, loop, do branches, and do simple arithmetic.

The following sum is a really good approximation to the sine function, as long as the right hand side contains more than 4 terms in the sequence shown:

$$S = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Let x and the number of terms on the right hand side be inputs to the algorithm and S , the sum, be the output. Note the use of the factorials in the above sum. Express your algorithm in terms of pseudocode *and* a flowchart.

14. The rich get richer, the poor get poorer!

The financial investment firm of Gettridge-Quigg, Ann Arbor, has a special deal for students. If you deposit your money with them they will give you 0.5% interest per month (a large amount these days) but then they will charge you a \$5.00 per month service charge applied against your account. If you have less than \$5.00 in your account, they take the remainder and your account is closed. So, if you deposited \$1000.00 and leave it there, your account will always have the same balance at the end of the month. If you deposit more than \$1000.00 and leave it there you may gain. Deposit less than \$1000.00 and your account may eventually be reduced to zero.

Write an algorithm that starts with an input of how much is to be deposited. You intend to leave that money in the account and never add or withdraw from it. Your algorithm will predict how many months it will take for you to double your money or to reduce your account to zero. If the balance in your account never changes, you have to state that as well. Note that Gettridge-Quigg first applies the interest and then applies the service charge. After every calculation, Gettridge-Quigg rounds the result to the nearest penny. For example, the monthly interest on a balance of \$1001.00 is \$5.01 while the monthly interest on a balance of \$1000.99 is \$5.00. You have to build this fact into your algorithm.

Express your algorithm in terms of pseudocode *and* a flowchart.

3.11 Projects

1. Follow the bouncing ball

You will describe an algorithm as if you were going to program it on a computer, that is, you will specify a set of instructions that even a machine can understand. Your algorithm should specify its purpose, the input data (if any), the output data (if any), and the set of steps that comprise the algorithm itself. The algorithm should include instructions on how to start and end. You can assume that the machine on which the algorithm will be implemented can store variables, loop, and do branches.

The problem is to determine the total distance a ball travels having been dropped from a height h (and traveling a total distance h on its way to the floor) and rebounding to a fraction of h which we will call ch (and traveling a total distance $h + ch$). c is called the “coefficient of restitution”. The ball will then drop to the floor (total distance = $h + 2ch$) and bounce up to a height c^2h (total distance = $h + 2ch + c^2h$) and so on. You can assume for simplicity that the ball is a “point”, that is, its diameter is exactly zero (a mathematical fiction to make our lives easier) and that c falls anywhere in the region $0 < c < 1$. Write an algorithm that determines the distance the ball travels before coming to a rest. You can make any reasonable assumption you like about what “at rest” means for the ball. However, the assumption has to be “reasonable” and justifiable either on the basis of numerics, mathematics or physics. This is a subtle point that is discussed below and is discussed in class.

This problem has an exact mathematical solution:

$$d = h \frac{1+c}{1-c}$$

that you must not employ in your algorithm to determine the total distance traveled. You have to obtain the solution by developing an algorithm that sums individually every bounce of the ball.

Some discussion:

A ball made from putty will have $c = 0$ and an ideal ball (which does not really exist in reality) would have $c = 1$. A fresh “out-of-the-can” tennis ball has $c = 0.5$ or so, an old tennis ball has $c = 0.25$ or so. A ping-pong ball has $c = 0.90$ or so.

This problem is very much like Xeno’s paradox. The paradox is that mathematically the total distance traveled by the ball is finite (as long as c is less than 1) but *mathematically* the total number of bounces is infinite! However, after many bounces the recovery distance becomes so small that accumulating them really does not matter in any practical sense any more. If digital computers could really represent real numbers to infinite precision then it would be possible to write an algorithm that accumulates the distance properly but it would never stop unless we introduced some artificial means of stopping the execution. Part of the purpose of this exercise is to get you thinking

about this paradox, to realize that computers can only provide an approximation to mathematical fiction and physical reality.

Chapter 4

Getting started in C++

4.1 Simple input/output (I/O): A first program in C++

Here is a first program in C++. You can type it in yourself, or download it from the web from the lecture note distribution area for this class.

```
//File: goBlue.cpp

#include <iostream>

using namespace std;

int main(void)
{   cout << "Go Blue!\n";

    // Multiple line statement
    cout <<
        "\n\n\t\"Go\t\t\\n\t\t\t\tBlue!\"\\a\\a\\a\\r";

    return(0);
}
```

There are several common features of every C++ program

Comments Comments start with `/*` and end with `*/`. For example,

```
/* This is a comment */
```

Comments are put there for humans only. They do not affect or modify what the computer does. The computer does not care if you comment or not. But humans do, especially your **Lab Instructors** and other people who **hand out grades** or **pay your salary**. Comments should explain what you are trying to accomplish and how you are going about doing it. It is appropriate and usually correct to assume that the person reading your code is a complete idiot and that you have to explain in detail what you are doing. After you have forgotten how or why you programmed your own code, you will find your own comments very useful to understand what you have done. The start of any program or program “segment” should begin with a few lines of explanation, along with an identification of the programmer and the date it was created. Put comments through the rest of the code wherever you think that the reader (*aka* the idiot) might need some explanation. Less is not more in this case. But, don’t go overboard too far. When in doubt, comment, comment, comment.

Comments can begin and end anywhere and can span more than one line.

```
// This is also a comment, from the "//" to the end of the line
```

which is useful when your comment is only one line or you want a comment to start somewhere in a line and continue to the end of the line.

Use both kinds of comments and make your code readable.

Preprocessor directives Preprocessor directives instruct the compiler in advance of the actual compilation of your code. Preprocessor directives are indicated by a “#” beginning a line. In this case the preprocessor directive

```
#include <iostream>
```

instructs the preprocessor to include the contents of the ANSI standard library file `iostream`. This header file contains other `#include` statements that call other library files that define the standards C++ input and output functions.

There are other common header files. We shall soon encounter `cmath` which is included via the preprocessor directive

```
#include <cmath>
```

which defines a lot of the common math functions. There are other types of preprocessor directives called “symbolic constants” and “macros” which we shall encounter later.

If you want to have a look at these library files, look in the directory called

```
/usr/include/g++-3
```

where they reside. **Boredom alert!** This is not for the faint-of-heart!

using namespace std; Tells the compiler that we will be using that standard “namespace” in the file that contains the line

```
using namespace std;
```

All this really means is that standard library files and definitions are being used. It is possible to define your own namespaces if you want to make sure that your variables and definitions do not conflict with either the standard ones, or someone else’s, someone working on the same project. For this course, we shall be using the standard namespace for everything we do. If you go on to work in a professional programming environment, writing software by teams, you will almost certainly be using your own namespaces.

int main(void){} All C++ programs begin processing at the first executable statement in **main**. The “()”’s indicate that **main** is a block of code called a *function*. The **int** in front of **main** declares **main** to be a function that is expected to “return” an integer. (More on this later.) **main** is a special function that the *operating system* recognizes as the place to begin execution of a C++ program.

The lines of code in between the “{}”’s is called the *function body*. It is considered good programming practice to *indent* the text in the function body by a certain amount of space. A common recommendation is 3 spaces which is reasonably pleasing to the eye. We will adopt this convention for this course. In our example, the function body starts with the statement

```
    cout << "Go Blue!\n";
```

which instructs the C++ standard library function **cout** function to print the series of characters “Go Blue!” (not including the double quotes) to the screen and then positions the cursor at the beginning of the next line. This positioning is effected by a special sequence of characters “\n” called an *escape sequence*. Escape sequences are necessitated when special characters are required or to “undo” the meaning of special characters, like the double quote (”) in the **cout** statement above. Here is a list of some commonly used escape sequences employed in **cout** statements.

\n	Newline. Move cursor to beginning of next line
\t	Horizontal Tab. Move cursor to next tab stop
\r	Return. Move cursor to beginning of current line
\a	Alert. Sound the bell
\\	Print a backslash (\)
\”	Print a double quote (”)

As an extreme example contrived to show all the possibilities,

```
cout <<
    "\n\n\t\"Go\t\t\\n\t\t\t\tBlue!\"\\a\\a\\a\\r";
```

which throws 2 newlines, goes to the first tab stop, prints a double quote followed by “Go”, advances two more tab stops writes a backslash, throws a newline, goes to the third tab stop and prints “Blue!” followed by a double quote, sounds the alarm 3 times (You will probably only hear one beep, depending on your computer.) and issues a carriage return. Because a new line was not thrown, further keyboard input could overwrite the “Blue!” text. This just goes to show how far a professor will go to contrive an example. Here is what the output looks like:

```

    "Go          \
red%                Blue!"
```

Note that the statement starting with `cout` is terminated with a semicolon (;). This is a special character in C++ called a delimiter that signifies the end of a statement. There can be more than one statement per line in a C++ code or a single statement can span several lines. Generally it is considered to be good programming practice to put at most only one statement per line of code. If a statement is long, put it on several lines, space things nicely, and indent the carry-over lines 3 spaces relative to the first line, like in the example above.

return(0); When this line is encountered, processing in the CPU returns to the operating system. Since `main` is a function of the `int` type, an integer value must be “returned” to the process that called it, in this case the O/S. A return value of 0 usually means normal execution, by convention. A return other than 0 can be treated as an error condition by the O/S. For example, if your program had prompted for user input and the user typed in something that the program was not prepared to process, one could signify this to the O/S by returning a 1, for example,

```
return(1);
```

You can have as many `return`’s as you need in a C++ code, just as you can have many stop points in an algorithm.

4.2 Compiling, linking, loading and running

Now that we have created a program, we have to tell the computer what to do with it! After all, computers only understand binary code and our program is written (coded) in terms of characters.

To compile the program, which we will call `goBlue.cpp`, the following statement on a unix machine attempts a compilation:


```
red% g++ goBlue.cpp
```

The C++ compiler is another program that converts text instructions into binary code that a computer understands. Compilers have evolved over a long period of time. The original computers (dating to the middle and late 40s) were actually programmed in binary code that was processed by the computer directly. This was an extremely laborious process. Early on it was recognized that it would be easier to program in a more human-compatible way, in a text-based format, and convert this into machine code using a *compiler*.

The origins of C++ date back to the late 60s and early 70s to the C language. The original intent for C was to develop a language for writing computer *operating systems*. Operating system (OS) code is special software for allowing users and programs to interact with data stored on a computer and instructs the operational units of a computer what to do with this data. Later, in the 80s and 90s, C++ was developed as an evolution of the C language. (Any C++ compiler will compile C code.) The number of improvements is very great—too many to mention.

C++ code is designed to be as hardware-independent as possible so that C++ code can migrate between different computer architectures. C++ code is translated into *assembler* or *assembly language* which represents a text version (*i.e.* human-readable) version of binary code. Assembly language is always machine dependent and addresses (speaks to) the computer hardware and its subcomponents. C++ is a higher-level language that deals with hardware and data in a more abstract way.

C and C++ have evolved beyond their original intent as OS-builders, although this role is still very important. C and C++ are powerful general-purpose languages for most aspects of computing. It is probably true that C and C++ are the most commonly-used computer languages in the world. C++ is certainly the one that is taught the most. There are other high-level languages for special purpose applications in business and numerical computation but probably C++ will supplant these as new business and numerical computation applications evolve.

After an algorithm is developed through the conceptual, pseudocode and flowchart process, a C++ program (or other computer language program) is written. During a C++ compilation and execution, several steps occur:

Edit A human writes a computer program, *e.g.* `goBlue.cpp` Any editor will suffice. Common unix editors are `vi`, `emacs`, `pico`, `nedit` or `kwrite`. Use whatever you like. Your lab instructor will inform you of your options and probably ask you to stick with one of these.

Preprocess, precompile Preprocessor directives, like `#include` and `#define` are handled. This can be inclusion of files or substitution of text throughout the C++ program.

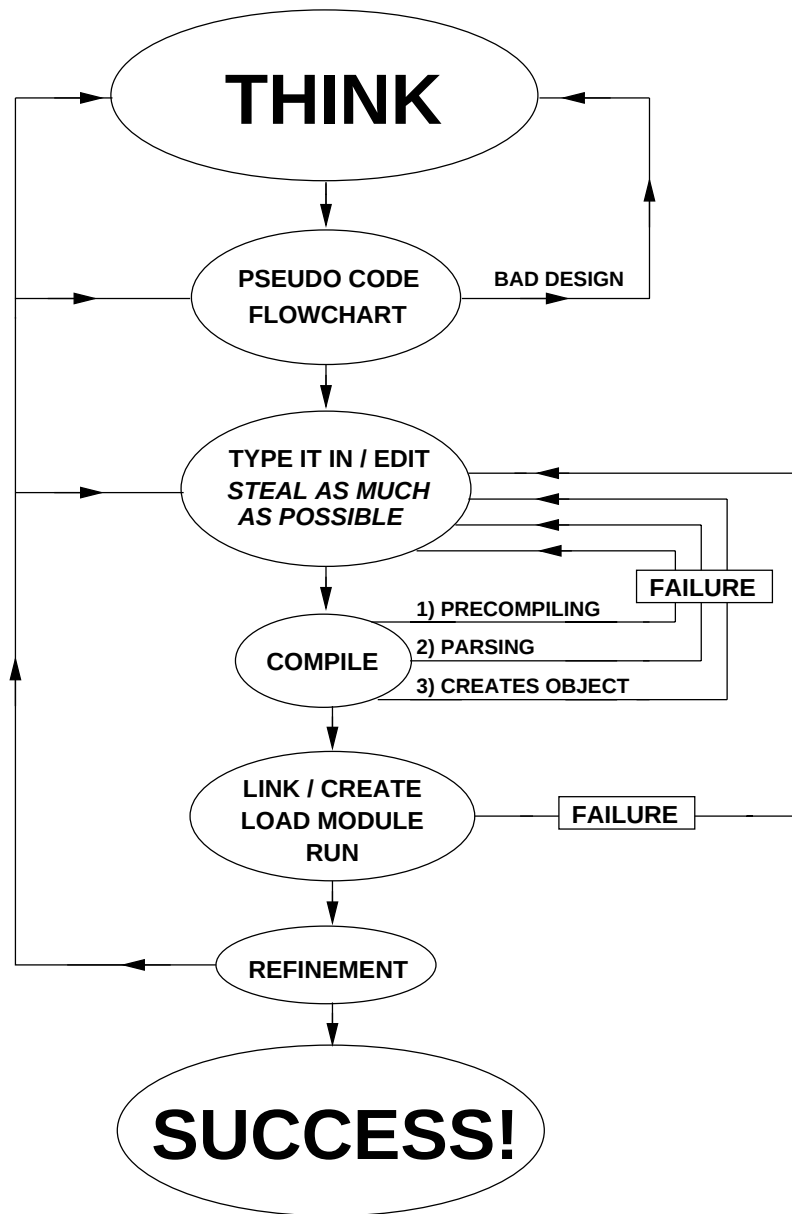
Compile After the preprocessor, the compilation converts the code to *object code* also known as *binary code*. There are actually several stages. The first is the *parsing* stage where the compiler checks the syntax of your code. If this stage is passed successfully the next stage is the production of the assembly code from the C++ code which is syntax-error free. Then the assembly code is converted to binary code, also called object code.

Link After the object code is created the *link* stage is entered. At this point, references to functions that may have been referenced in the C++ program are established. These missing parts, in the form of object code, are either put in place directly where referenced (this is called *static linking*) or their transfer locations are established so that processing can transfer to these locations and be resumed. This is called *dynamic linking*. The file created by the linker is called the *load module*. By default, the load module is called *a.out* by the unix operating system. A user can override this name if desired.

Load The *loader* takes the load module (usually located on disk) and puts it into memory in a place where the operating system expects to see executable code. On a unix system the loading phase is started when a user types the name of the load module resident on disk.

Run In the run phase, the binary code in the memory-resident load module is transferred to the *central processing unit* (CPU) for execution. The load module contains instructions for the CPU that directs it to assign tasks to the various operational units of the computer. The load module also contains data and references to data that are to be operated on. The run phase is where conceptual errors in a computer program are exposed. Code that has incorrect syntax, and syntax errors are quickly found by the compiler's parser, are usually easy to correct. Code that has correct syntax but incorrect design are much harder to find. This is where pseudocode and flow charts can come to the rescue.

This model of editing, preprocessing, parsing, object-code generation, linking, loading and running is followed by all computer code-development environments, from the most simple “Go Blue!” C++ program to the most complex, like the telephone management system, international banking systems and the national security systems—irrespective of the language it is written in. The following figure includes not only the code compilation process, but also describes the whole process of code development. It is general, and mostly independent of the computer language being employed.



4.3 Declaring and initializing variables

Consider the following C++ program that does mathematical operations with two integers:

```
//File: integerMath.cpp
```

```
#include <iostream>
```

```
using namespace std;

int main(void)
{  int i1 = 5;  // Define and initialize
   int i2 = 2;  // Define and initialize
   int iResult; // Define only

   iResult = i1 + i2; // sum
   cout << "5 + 2 = " << iResult << "\n";

   iResult = i1 - i2; // difference
   cout << "5 - 2 = " << iResult << "\n";

   iResult = i1 * i2; // product
   cout << "5 * 2 = " << iResult << "\n";

   iResult = i1 / i2; // division
   cout << "5 / 2 = " << iResult << "\n";

   iResult = i1 % i2; // remainder
   cout << "5 mod 2 is " << iResult << "\n";

   return(0);
}
```

There are some new features of this program.

The statement

```
int i1 = 5;
```

is a *declaration* statement. The name `i1` is an *identifier* for the *variable* `i1`. This declaration statement specifies that `i1` is an `int`, a signed integer. It also provides an *initialization* for `i1`, giving it the *value* of 5.

An identifier is a series of characters you can type at a keyboard that can consist of letters, digits, and underscores (`_`). It is a bad idea to start an identifier with an underscore, even though it is legal, because most identifiers in libraries accessed by C++ use this. An identifier can not start with a digit, however. According to the draft ANSI C++ standard, an identifier may be any length. ANSI [American National Standards Institute] C++ is built upon ANSI C, and in ANSI C only the first 31 characters are required to be recognized by C compilers conforming to the ANSI standard for C. Different C++ compiler writers determine how many characters in an identifier are significant. I recommend NOT using more than

31, for safety. Beyond the limit of 31 or greater, your compiler may treat the characters beyond it as significant or insignificant (*i.e.* ignores them). For absolute safety, use 31 characters or less. Identifiers in C++ are *case sensitive* so that `i1` and `I1` are recognized as different variables. Unless your program is very short, you should name your identifiers in some recognizable way—it cuts down on the amount of commenting you have to do. I also like “humpback” notation, `numberOfStudents`, rather than `number_of_students`. Note that humpback identifiers start with lowercase letters (usually) and new words within the identifier start with an uppercase letter. I find underscores in the middle of identifiers ugly, wasteful and hard to type.

The statement

```
int i2 = 2;
```

is another declaration statement that identifies the name `i2` and declares it to be a signed integer initialized to the value 2.

The statement

```
int iResult;
```

is another declaration statement that identifies the name `iResult` and declares it to be a signed integer. In this case the variable is not initialized.

4.4 Integer math in C++

The statement

```
iResult = i1 + i2;
```

contains two mathematical operators, `=` and `+`. The operator `=` is called the *assignment operator*. **Do not call it the “equal sign”!** The assignment operator assigns the result of the operation(s) to the *right* of it to the value of the identifier to the left of it, `iResult` in this example. `=` is called a *binary* operator because it operates on only two quantities, the variable to the left which gets the assignment and the quantity to the right that provides the assignment (after the other mathematical operations are completed).

The operator `+` is called the *addition operator*. It is also a binary operator and provides the mathematical sum of the quantities to its left and right.

There are 4 other binary operators to consider now, the *subtraction*, `-`, the *product*, `*`, the *division*, `/`, and the *modulus*, `%`. Of these the division requires special mention with regards

to its operation with integers. The integer division returns the integer part of the result. For example, $5/2 = 2$, not 2.5. The remainder of a division can be obtained from the modulus function `%`. Thus $5 \% 2 = 1$.

The various mathematical operations, when employed within the same statement, follow certain ordering rules. These are called the *rules of precedence*. The `*`, `/` and `%` operations take precedence over `+` and `-`. `*`, `/` and `%` have the same precedence, and `+` and `-` have the same precedence as well. For operators with the same precedence, the order of operation is left to right. For example,

$$1 + 10 * 5 - 9 / 3 = 1 + 50 - 3 = 48$$

If a programmer desires a different interpretation of the above statement or wishes to make the operation clearer, *parentheses* may be employed. Parentheses take precedence over `*`, `/`, `%`, `+` and `-`. Inner parentheses are evaluated first. For example,

$$(1 + (10 * 5) - (9 / 3)) = 1 + 50 - 3 = 48$$

is operationally equivalent to the above. However,

$$((1 + 10) * (5 - 9)) / 3 = (11 * (-4))/3 = -44/3 = -14$$

is quite different.

Summarizing, the order of precedence is

Order	Symbol	Operation	Comments
0	<code>()</code>	parentheses	inner ones evaluated first
1	<code>(unary) -</code>	negation (unary minus)	R→L evaluation
2	<code>* / %</code>	product, division, modulus	L→R evaluation
3	<code>+ -</code>	addition, subtraction	L→R evaluation

The statement

```
cout << "5 + 2 = " << iResult << "\n";
```

prints the characters “5 + 2 = ” to the screen then the value of `iResult` and then sends the newline escape sequence. There are more flexible ways of printing things to the screen and we shall encounter these shortly. Note how we can “cascade” the `<<` operator for `cout`. An equivalent way of accomplishing the same thing would be:

```
cout << "5 + 2 = ";
cout << iResult;
cout << "\n";
```

which is a little harder to read. Break up your output statement like this only for neatness and readability.

4.5 Floating point math in C++

Consider the following C++ program that does mathematical operations with two floats:

```
//File: floatMath.cpp

#include <iostream>

using namespace std;

int main(void)
{ cout << "Input the 1st floating point number: ";
  float f1; // Declare the 1st float
  cin >> f1; //and read it in

  cout << "Input the 2nd floating point number: ";
  float f2;
  cin >> f2;

  float fResult = f1 + f2; // sum
  cout << f1 << " + " << f2 << " = " << fResult << "\n";

  fResult = f1 - f2; // difference
  cout << f1 << " - " << f2 << " = " << fResult << "\n";

  fResult = f1 * f2; // product
  cout << f1 << " * " << f2 << " = " << fResult << "\n";

  if (0 == f2)
  { cout << "Division by zero not permitted!\n";
    return(1);
  }
  else
  { fResult = f1 / f2; // division
    cout << f1 << " / " << f2 << " = " << fResult << "\n";
    return(0);
  }
}
```

There are some new features in the above code.

```
float f1;
```

declares the existence of a floating-point number identified as `f1`. A “float” is a representation of numbers like 1.0, 2.5, 0.3333... in (usually) 32-bit representation. How a computer does this we will leave as a mystery for the time being. However, the representation of most floating point numbers is inexact, about 1 part in 10^7 for floats and about 2 parts in 10^{16} for a “double”, a 64-bit representation of real numbers. Float and doubles also have maximum and minimum values defined and this is architecture dependent, that is, on the computer you are using. Common “dynamic ranges” for floats are between about 10^{-38} and 10^{38} and about 10^{-308} and 10^{308} for doubles. We’ll have a lot more to say about floats as we get deeper into the course. Floats and doubles are pretty much central to the issue of computing in engineering and we will discuss the concepts as we require them.

Note that in C++ we can declare variables anywhere. Many programmers like to declare them just before using them for the first time.

The statement:

```
cin >> f1;
```

inputs the float `f1` from the keyboard.

4.6 The if/else if/else construct

The `if/else if/else` construct is one of the most important programming features of C++ and common to many computer languages. It has a single-choice form called `if`, a double-choice form called `if/else` and multiple-choice forms `if/else if` and `if/else if/else`.

The general form of the `if/else if/else` is:

```
if (logical expression 1)
{
    STATEMENT BODY 1
}
else if (logical expression 2)
{
    STATEMENT BODY 2
}
else if (logical expression 3)
{
    STATEMENT BODY 3
}
```



```

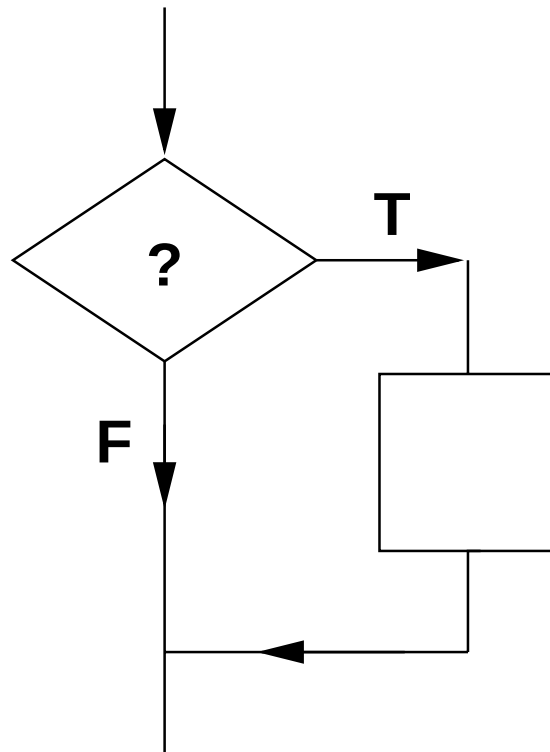
}
.
.
.
else if (logical expression N + 1)
{
    STATEMENT BODY N + 1
}
else
{
    STATEMENT BODY N + 2
}

```

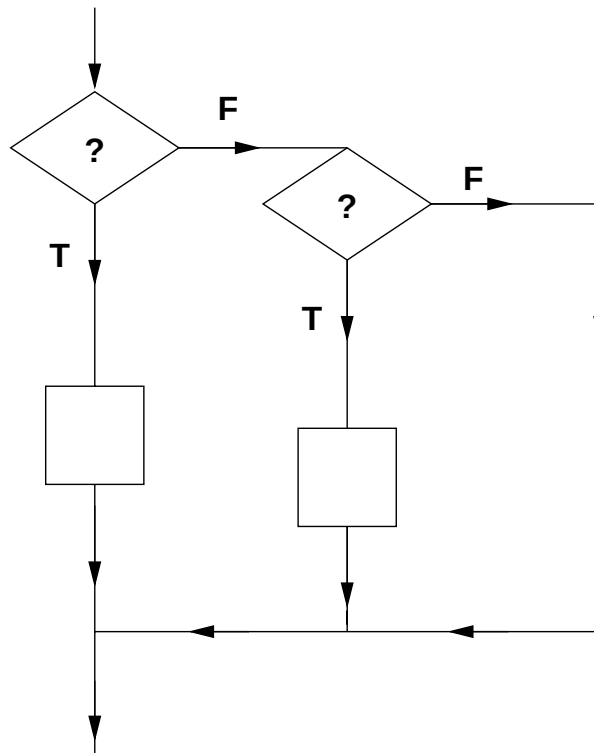
There are many forms of the `if/else if/else` construct:

- The `else if (logical expression) { STATEMENT BODY }` is optional
- The `else { STATEMENT BODY }` is also optional, independent of the above
- If the `STATEMENT BODY` consists of one and only one statement, then the surrounding `{}`'s are optional. (It can help with clarity of programming to keep them, however. Their use is highly recommended.)
- Note the use of indenting, optional but highly recommended
- At most one and only one of the `STATEMENT BODY`'s can execute within an `if/else if/else` construct can be executed
- Processing control passes to the statement immediately following the `if/else if/else` construct after execution of one of the `STATEMENT BODY`'s.
- If the `if/else if` or `if` form is used, it may happen that none of the logical conditions is satisfied. In this case processing control passes to the statement following the construct.
- `if/else if/else` constructs can be nested within each other

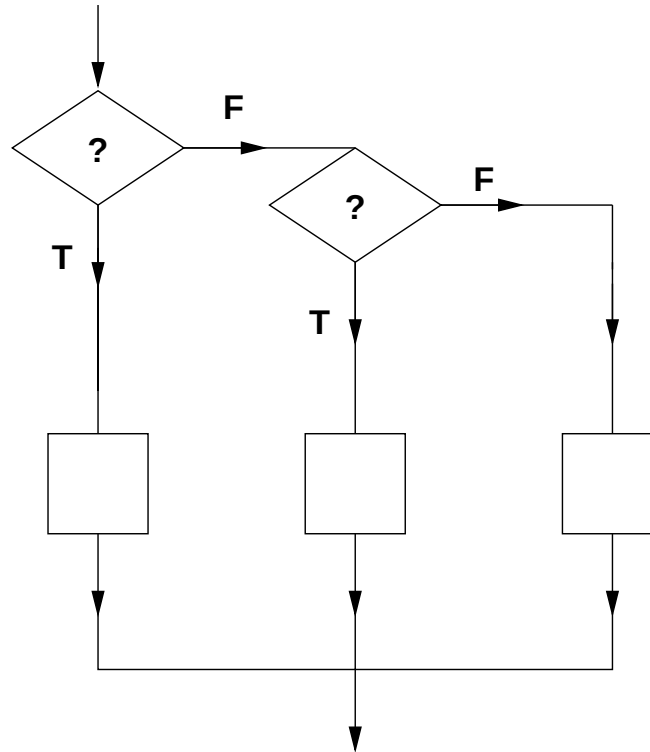
This is the flow chart for the `if` construct:



This is the flow chart for the `if/else if` construct:



This is the flow chart for the `if/else if/else` construct:



4.7 Logical expressions

So, what is a *logical expression*???

A logical expression is something that evaluates to TRUE or FALSE. FALSE has a numeric value of the integer 0. TRUE has a numeric value of integer non-zero.

For example:

```

if (1)
    cout << "This would always print,\n";
else if (0)
    cout << "whereas this would NEVER print\n";
  
```

Here are some of the logical operators and their order of precedence

Order	Symbol	Operation	Comments
0	()	parentheses	inner ones evaluated first
1	(unary) !	negation (unary NOT)	R→L evaluation
2	<=,<	less than or equal, less than	L→R evaluation
2	>,>=	greater than, greater than or equal	L→R evaluation
3	==,!=	equivalent to, not equivalent to	L→R evaluation

Let's return to the example of solving the quadratic equation, $Ax^2 + Bx + C = 0$ that we studied in flowchart and pseudocode form in Lecture 3. If we look at the pseudocode on page 7 of Lecture 3 we see that the `if`'s are nested and we would probably, at first try, write a code that looked something like the following, using only `if`'s and `else`'s:

```
//File: quadratic.cpp

#include <iostream>
#include <cmath>

using namespace std;

int main(void)
{ float a,b,c; // Declare the constants...

  cout << "\n"
        "We will solve for x in Ax*x + B*x + C = 0\n"
        "=====\n"
        "\n";
  cout << "Input A,B,C: ";
  cin >> a >> b >> c; //...and read them in

  cout <<
    "Calculating roots (x) of the equation\n"
    << a << "*x*x + " << b << "*x + " << c << " = 0...\n";

  if (0 > b*b - 4*a*c) //b*b - 4*a*c is less than 0
  { cout << "No solution since b*b-4*a*c < 0\n";
    return(0);
  }
  else // b*b - 4*a*c is >= 0
  { if (0 != a)
    {
      cout <<
        "Sol'n 1 = " << (-b + sqrt(b*b - 4*a*c))/(2*a) << ", "
```

```

        "Sol'n 2 = " << (-b - sqrt(b*b - 4*a*c))/(2*a) << "\n";
    return(0);
}
else // b*b-4*a*c is >= 0 and a = 0
{   if (0 != b) // b*b - 4*a*c is >= 0, a = 0, b != 0
    {   cout << "One sol'n since a = 0. Sol'n = " << -c/b << "\n";
        return(0);
    }
    else // b*b - 4*a*c is >= 0, a = 0, b = 0
    {   cout << "No sol'ns since a and b = 0.\n";
        return(0);
    }
}
}
}
}

```

However, by using `else if`'s we can make the code look less “nested” and complicated. The nesting is really still there and we have to remember that when we pass into an `else if` that the `if` condition above it is still false! However, `else if`'s make the coding look neater and easier to read.

```
//File: quadratic2.cpp
```

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
```

```
int main(void)
```

```
{   float a,b,c; // Declare the constants...
```

```
    cout << "\n"
```

```
        "We will solve for x in Ax*x + B*x + C = 0\n"
```

```
        "=====\n"
```

```
        "\n";
```

```
    cout << "Input A,B,C: ";
```

```
    cin >> a >> b >> c; //...and read them in
```

```
    cout <<
```

```
        "Calculating roots (x) of the equation\n"
```

```
        << a << "*x*x + " << b << "*x + " << c << " = 0...\n";
```

```

if (0 > b*b - 4*a*c) //b*b - 4*a*c is less than 0
{ cout << "No solution since b*b-4*a*c < 0\n";
  return(0);
}
else if (0 != a) // b*b - 4*a*c is >= 0, a != 0
{ cout <<
  "Sol'n 1 = " << (-b + sqrt(b*b - 4*a*c))/(2*a) << ", "
  "Sol'n 2 = " << (-b - sqrt(b*b - 4*a*c))/(2*a) << "\n";
  return(0);
}
else if (0 != b) // b*b - 4*a*c is >= 0, a = 0, b != 0
{ cout << "One sol'n since a = 0. Sol'n = " << -c/b << "\n";
  return(0);
}
else // b*b - 4*a*c is >= 0, a = 0, b = 0
{ cout << "No sol'ns since a and b = 0.\n";
  return(0);
}
}

```

4.7.1 Logical expressions with AND or OR

Here are more of the logical operators and their order of precedence:

Precedence	Symbol	Operation	Comments
0 (first)	(,)	parentheses, paired	L→R evaluation
1	(unary) !	negation (unary NOT)	R→L evaluation
2	<=,<	less than or equal, less than	L→R evaluation
2	>,>=	greater than, greater than or equal	L→R evaluation
3	==,!=	equivalent to, not equivalent to	L→R evaluation
4	&&	logical AND	L→R evaluation
5 (last)		logical OR	L→R evaluation

The AND and OR operators work as follows. Imagine that two logical expressions evaluates to either T (TRUE) or F (FALSE). This is the result of the && and ||'s operating on them.

(T	&&	T)	==	T
(T	&&	F)	==	F
(F	&&	T)	==	F
(F	&&	F)	==	F
(T		T)	==	T
(T		F)	==	T
(F		T)	==	T
(F		F)	==	F

Sometimes these tables are difficult to remember, at least as *abstract* concepts. This is helped somewhat if we substitute numerical values for T (1) and F (0).

(1	&&	1)	==	1
(1	&&	0)	==	0
(0	&&	1)	==	0
(0	&&	0)	==	0
(1		1)	==	1
(1		0)	==	1
(0		1)	==	1
(0		0)	==	0

Once we do this we realize that the result of (A && B) is the lower value of either A or B. The result of (A || B) is the higher value of either A or B. It is no more complex than that! And this is, in fact, how the electronic circuitry works in a computer during the interpretation of logical expressions.

Some examples:

```
int i = 5;

if (0 < i && 10 > i)
    cout << i << " is a positive, single digit integer\n";

if (0 > i || 10 <= i)
    cout << i << " is either negative, or has more than one digit\n";
```

4.7.2 Mixed arithmetic and logical expressions

It is possible to mix arithmetic and logical expressions. Usually it is done where a logical expression is expected, such as within an `if/if else/else` construct. It is also possible to mix these operators elsewhere, where a regular statement is expected. **Don't do it**—unless you really, really have to for some purpose!

When logical operators and arithmetic operators are mixed in a single expression, the order of precedence is as follows. Above all, the contents of parentheses are evaluated starting with the innermost. Then the arithmetic operators get interpreted in the order already discussed. Finally, the logical operators get interpreted in the order already discussed. Here is the table for precedence for both arithmetic and logical expressions.

Precedence	Symbol	Operation	Comments
0 (first)	()	parentheses, paired	L→R evaluation, innermost first
1	(unary) - !	negation	R→L evaluation
2	* / %	multiply, divide, modulus/remainder	L→R evaluation
3	+ -	add, subtract	L→R evaluation
4	<=, <	less than or equal, less than	L→R evaluation
4	>, >=	greater than, greater than or equal	L→R evaluation
4	==, !=	equivalent to, not equivalent to	L→R evaluation
5	&&	logical AND	L→R evaluation
6		logical OR	L→R evaluation
7 (last)	=	assignment	R→L evaluation

When in doubt, use parentheses. When not in doubt, use parentheses anyway. The person reading your code may have forgotten all but the first rule of precedence, parentheses are interpreted first. This will make your code more readable and understandable.

In fact, that is the convention I have adopted in almost all of my programming, put parentheses around every logical sub-expression in any logical expression that contains more than one logical operators. I also put parentheses around any arithmetical sub-expression in a mixed arithmetic/logical expression.

Here's an example of a mixed arithmetic and logical expression. Use the rules of precedence stated above to see how the following statement is evaluated:

```
if (i*j == i*i + j*j || i - j <= i*i - j*j && i + j <= i*i + j*j) ...;
```

The ordering of && and || is also quite important. For example, using the rules of precedence the above could collapse to `1 || 1 && 0` which is 1 or TRUE. However, if you mistakenly thought that the || is evaluated first, you would guess that the answer should be 0 or FALSE, and that is the wrong answer in this case.

4.8 Problems

1. An essay-type question? In a computer¹ class!

Answer the following questions in your own words...

- (a) What is an algorithm?
- (b) Not counting the ability to **START** and **STOP**, what are the 3 functional building blocks that are used to construct any algorithm.
- (c) Give one example each of these 3 functional building blocks
- (d) Name 3 ways in which algorithms may be described.
- (e) Give one example each of the above 3 ways in which algorithms may be described.

2. Basic Arithmetic

Write down the value that is assigned to the variable on the left-hand side of each of these expressions. The variables `i` and `j` are `int`'s; `x` and `y` are `float`'s.

```
i = 1 + 2*3/4+5;
x = 1.0 + 2.0 + 3.0/4.0 + 5.0;
j = (1 + 2)*3/4 + 5;
y = (1.0 + 2.0)*3.0/4.0 + 5;
y = 1 + 2 + 3/4 + 5;
x = 1 - (static_cast<int>(6*5/4.0*3))/2;
```

3. Basic Logical Operations

Do the following logical expressions evaluate to TRUE (1) or FALSE (0)?
`i` is an `int` that can take on any value.

```
(1 || 1 && 0 || 0)
(0 > i && 10 < i)
(0 > i || -10 < i)
```

4. Mixed Logical and Mathematical Operations

Do the following mixed mathematical-logical expressions evaluate to TRUE (1) or FALSE (0)?
`i` is an `int` that can take on any value. (Hint: No matter what `i` is, `2*i + 1` is an odd number and `2*i` is an even number.)

¹Hey! Unless you've been on the moon for the last two weeks...This is a course about thinking and making things work! Writing computer program is just one way (and a good way for Engineers!) to teach this.

```
(2*i + 1)
(4*i%2 && 1)
(2*i + 1 || i/2)
```

5. Mind Your P's, Q's and Semi-Colons

Which of the following are valid statements?

```
int i, j, k = 0;
int i, double j;
int i0 = 4, f = -1, k=0;
int i; float _0j;
int i j k l m;
int The, Wolverines, Are, Rated, Number, 3;
float f, F, _000000, Wolverine;
int i,j ; i == 1 && j || 3%2;
int static_cast<float>(N);
i = -i + -j;
i + j = f;
x = x*x + 3;
y = x^2 + 3;
y = a x + b;
z = x + y123;
z = x + 123y;
for(i = 0,i <= 10,i = i + 1) cout << "OK?\n";
while(thisIsTrue) cout << "OK?\n";
if (((x < 0) && (y > 3) || (z == 2)) cout << "OK?\n";
if (a != b && c > d && e < f && g == h) cout << "OK?\n";
```

6. Predict the Output

The examples of code given below all compile correctly. Predict the output that the code would produce if you compiled and ran it.

- ```
int N = 0;
if (N = 0)
{ cout << "The conditional expression was TRUE\n";
}
else
{ cout << "The conditional expression was FALSE\n";
}
```
- ```
int i = 0;
if (0 != i)
```

```

    cout << "Statement 1\n";
    cout << "Statement 2\n";
    cout << "Statement 3\n";

    • int j = 0;
      if (0 < j);
      {   j = j + 1;
      }
      cout << "j = " << j << "\n";

```

7. This question is iffy, REALLY iffy

Dreaming up your own examples, write syntactically correct C++ code for the following. I have provided the answer to the first question to give you an idea of what I am looking for. You do not have to declare or initialize any of the variables you use.

- An example of the use of the `if`. If the condition is **TRUE** then one and only one statement is executed.
A correct answer is: `if (i < N) f = f*i;`
- An example of the use of the `if`. If the condition is **TRUE** then exactly two statements are executed.
- An example of the use of the `if` and the `else if`. If the `if` condition is **TRUE** then exactly two statements are executed. If the `else if` condition is **TRUE** then one and only one statement is executed.
- An example of the use of the `if` and two `else if`'s.
- An example of the use of the `if`, two `else if`'s and the `else`.

8. Location, location, location

Write a program that prompts its user for two `int`'s, the first representing an x -coordinate and the second representing a y -coordinate. Then the program prints out where the (x, y) point is located, based on the following scheme.

$x > 0$	$y > 0$	upper right quadrant (URQ)
$x > 0$	$y < 0$	lower right quadrant (LRQ)
$x < 0$	$y > 0$	upper left quadrant (ULQ)
$x < 0$	$y < 0$	lower left quadrant (LLQ)
$x = 0$	$y > 0$	upper half (UH)
$x = 0$	$y < 0$	lower half (LH)
$x > 0$	$y = 0$	right half (RH)
$x < 0$	$y = 0$	left half (LH)
$x = 0$	$y = 0$	in the center (C)

Feel free to use abbreviations, like URQ, for upper right quadrant.

Here's an example of how it works:

```
% a.out
Enter two int's, representing x and y: 1 1
Upper right quadrant (URQ)
%
```

4.9 Projects

1. Your first programming problem

You have to create an entire code that will compile correctly, accept inputs and provide outputs and stops. As for coding style, follow Appendix A.

Your program will do the following:

- (a) Ask the user to input two non-negative integers.
- (b) If the user inputs a negative integer, the program issues an error message and stops.
- (c) If the two numbers are equal, output is provided to inform the user. Otherwise the user is informed which number is larger and which is smaller.
- (d) The program outputs the ratio of the larger number to the smaller number unless the smaller number is zero. If the smaller number is zero, a message to this effect is output.
- (e) The program outputs the quadrature sum of the two numbers, that is, `firstInteger * firstInteger + secondInteger * secondInteger`.
- (f) If the smaller number is zero, the program stops. Otherwise, the program checks if the smaller number is a perfect divisor of the larger number. This means there is no remainder after division. For example, 2 is a perfect divisor of 8, but not 7. Output is provided to inform the user whether or not a perfect divisor was found.
- (g) Program stops.

The above set of instructions is really a coarse pseudocode for your program. Read the instructions very carefully.

Here are some example uses of the program. Your program should work exactly the same.

```
> a.out
Input the 1st integer: -1
Input must be non-negative. Try again!
```

```
> a.out
Input the 1st integer: 1
Input the 2nd integer: -1
Input must be non-negative. Try again!
```

```
> a.out
Input the 1st integer: 1
Input the 2nd integer: 1
```

```
The two numbers are the same!
The ratio is 1, the quadrature sum is 2
1 is a perfect divisor of 1

> a.out
Input the 1st integer: 10
Input the 2nd integer: 0
The bigger number is 10, the smaller number is 0
The smaller number is 0, so no ratio is calculated, the quadrature sum is 100

> a.out
Input the 1st integer: 0
Input the 2nd integer: 10
The bigger number is 10, the smaller number is 0
The smaller number is 0, so no ratio is calculated, the quadrature sum is 100

> a.out
Input the 1st integer: 7
Input the 2nd integer: 4
The bigger number is 7, the smaller number is 4
The ratio is 1, the quadrature sum is 65
4 is not a perfect divisor of 7

> a.out
Input the 1st integer: 4
Input the 2nd integer: 7
The bigger number is 7, the smaller number is 4
The ratio is 1, the quadrature sum is 65
4 is not a perfect divisor of 7

> a.out
Input the 1st integer: 8
Input the 2nd integer: 2
The bigger number is 8, the smaller number is 2
The ratio is 4, the quadrature sum is 68
2 is a perfect divisor of 8

> a.out
Input the 1st integer: 2
Input the 2nd integer: 8
The bigger number is 8, the smaller number is 2
The ratio is 4, the quadrature sum is 68
2 is a perfect divisor of 8
```

2. Leap year calculation

The rules to calculate if it is a leap year are:

- (a) A year divisible by 4 is a leap year (2004, 2008...), unless
- (b) it is also divisible by 100 (2100, 2200...) in which case it is not a leap year.
- (c) There is an exception. A year divisible by 400 is a leap year (2000, 2400...).

Write a program that will accept a positive `int`, the year to be tested, from a user. If the user inputs a 0 or negative number, the program quits without doing anything. Otherwise, it will print a message saying whether or not the year is a leap year. Here is an example of how the program works.

```
unix-prompt> a.out
Input a number for the year (>= 1): -1
Sorry, your year must be greater than 1.
unix-prompt> a.out
Input a number for the year (>= 1): 2001
The year 2001 is not a leap year
unix-prompt> a.out
Input a number for the year (>= 1): 2004
The year 2004 is a leap year
unix-prompt> a.out
Input a number for the year (>= 1): 2100
The year 2100 is not a leap year
unix-prompt> a.out
Input a number for the year (>= 1): 2400
The year 2400 is a leap year
```

3. When in Rome...

The following table shows how Roman numerals are formed:

Arabic	Roman	Arabic	Roman	Arabic	Roman	Arabic	Roman
1	I	10	X	100	C	1000	M
2	II	20	XX	200	CC	2000	MM
3	III	30	XXX	300	CCC	3000	MMM
4	IV	40	XL	400	CD		
5	V	50	L	500	D		
6	VI	60	LX	600	DC		
7	VII	70	LXX	700	DCC		
8	VIII	80	LXXX	800	DCCC		
9	IX	90	XC	900	CM		

A Roman number is written in decades starting with the largest decade on the left and then going to the right. For example, the number 1999 would be represented as MCMXCIX. The largest number that can be represented by Roman numerals is 3999, or MMMCMXCIX. Write a program that will accept a positive `int`, between 1 and 3999 inclusive. If the user inputs something outside this range, the program quits without doing anything. Otherwise, it will print the conversion to Roman numerals. Here is an example of how the program works.

```
unix-prompt> a.out
Input an integer between 1 and 3999 inclusive: 0
Sorry, your integer must be between 1 and 3999 inclusive.
unix-prompt> a.out
Input an integer between 1 and 3999 inclusive: 4000
Sorry, your integer must be between 1 and 3999 inclusive.
unix-prompt> a.out
Input an integer between 1 and 3999 inclusive: 1338
MCCCXXXVIII
unix-prompt> a.out
Input an integer between 1 and 3999 inclusive: 3888
MMMDCCCLXXXVIII
unix-prompt> a.out
Input an integer between 1 and 3999 inclusive: 52
LII
```

4. This is a 5-bit calculation

Write a program that will accept 5 bits (0 or 1) in the following order: b_0, b_1, b_2, b_3, b_4 representing a 5-bit bit pattern $b_4b_3b_2b_1b_0$. (Note the ordering.) Write a program that:

- (a) Determines if the 5-bit bit pattern $b_4b_3b_2b_1b_0$ represents an odd or even unsigned integer.
- (b) Determines if the bit pattern represents a negative or positive signed integer in 5-bit two's complement representation.
- (c) Prints the unsigned integer representation.
- (d) Prints the bit pattern's bit complement $\tilde{b}_4\tilde{b}_3\tilde{b}_2\tilde{b}_1\tilde{b}_0$ (1's become 0's and 0's become 1's).
- (e) Prints the signed integer representation assuming 5-bit two's complement representation.

Here is an example of how the program works.


```
unix-prompt> a.out
Bit b0: 0
Bit b1: 1
Bit b2: 0
Bit b3: 1
Bit b4: 0
5-bit bit pattern is: 0 1 0 1 0
Bit pattern represent an even unsigned integer
Bit pattern represent a positive signed integer
Unsigned integer representation is: 10
Complement 5-bit bit pattern is: 1 0 1 0 1
Signed integer representation is: 10
```

```
unix-prompt> a.out
Bit b0: 1
Bit b1: 1
Bit b2: 1
Bit b3: 1
Bit b4: 0
5-bit bit pattern is: 0 1 1 1 1
Bit pattern represent an odd unsigned integer
Bit pattern represent a positive signed integer
Unsigned integer representation is: 15
Complement 5-bit bit pattern is: 1 0 0 0 0
Signed integer representation is: 15
```

```
unix-prompt> a.out
Bit b0: 1
Bit b1: 1
Bit b2: 1
Bit b3: 0
Bit b4: 1
5-bit bit pattern is: 1 0 1 1 1
Bit pattern represent an odd unsigned integer
Bit pattern represent a negative signed integer
Unsigned integer representation is: 23
Complement 5-bit bit pattern is: 0 1 0 0 0
Signed integer representation is: -9
```

```
unix-prompt> a.out
Bit b0: 1
Bit b1: 0
```

```

Bit b2: 0
Bit b3: 0
Bit b4: 1
5-bit bit pattern is: 1 0 0 0 1
Bit pattern represent an odd unsigned integer
Bit pattern represent a negative signed integer
Unsigned integer representation is: 17
Complement 5-bit bit pattern is: 0 1 1 1 0
Signed integer representation is: -15

```

5. Is binary arithmetic iffy

We return to binary arithmetic of 5-bit bit patterns that was considered in one of the problems in the previous chapter. We imagine a machine that can only represent integers in 5-bit bit strings. However, this time you will write a program that *emulates* 5-bit binary arithmetic using actual `int`'s in C. Remember that there is no 6'th or higher bit. The result of any calculation that would carry over into the 6'th bit is thrown away.

The program you will write will do two things, depending on the choice of the person using your program:

- (a) multiply a 5-bit bit pattern by a 2-bit bit pattern, for example, 10101×11 , or
- (b) add two 5-bit bit patterns, for example, $10101 + 10011$.

The user of your program will provide the bit patterns that are to be added or multiplied. Your code will output the 5-bit bit pattern that is the result of the multiplication or addition.

Examples of use:

Here is an example of running the code:

```
> a.out
```

```
What do you want to do?
```

```
Type a 0 if you want to add two 5-bit strings
```

```
Type a 1 if you want to multiply a 5-bit string by a 2-bit string
```

```
(0 or 1): 1
```

```
Enter the 5-bit binary number to be multiplied: 1 0 1 0 1
```

```
Enter the 2-bit multiplier: 1 1
```

```

  10101
x   11
-----

```

...is the same as the addition...

```

  10101
+01010
-----
  11111
>

```

Here is another example of running the code:

```
> a.out
```

What do you want to do?

Type a 0 if you want to add two 5-bit strings

Type a 1 if you want to multiply a 5-bit string by a 2-bit string

(0 or 1): 0

Enter the 1st 5-bit binary number to be added: 1 0 1 0 1

Enter the 2nd 5-bit binary number to be added: 1 0 0 1 1

```

  10101
+10011
-----
  01000
>

```

6. Change for the good

You will write a program that will accept an input, a real number between 0.00 and 19.99 inclusive. This represents the “change” in dollars from the purchase of an item using a \$20 bill. You will write a C++ program that will calculate the most efficient form that the change may be given in terms of \$10, \$5, and \$1 bills plus quarters, dimes, nickels and pennies.

Here’s an example of how it should work. Note that the program was run twice. The first time the input was 19.94, the second time 16.26. Note how the program correctly uses singular (One penny) and plural (Four \$1 bills).

```
unix-prompt> a.out
Input a dollar amount between 0.00 and 19.99 inclusive: 19.94
Change of $19.94 is best given as:
  One $10 bill
  One $5 bill
  Four $1 bills
  Three quarters
  One dime
  One nickel
  Four pennies

unix-prompt> a.out
Input a dollar amount between 0.00 and 19.99 inclusive: 16.26
Change of $16.26 is best given as:
  One $10 bill
  One $5 bill
  One $1 bill
  One quarter
  One penny

unix-prompt>
```

7. Base 10 to binary conversion

Write a C++ program that will accept any integer between 0 and 127, inclusive, and convert it to binary. You may not use loops of any kind and your solution must NOT resemble the following:

```
if (decimal == 0) cout << "0";
else if (decimal == 1) cout << "1";
else if (decimal == 2) cout << "10";
else if (decimal == 3) cout << "11";
.
.
.
else if (decimal == 126) cout << "1111110";
else if (decimal == 127) cout << "1111111";
```

Here's an example of how it should work for an input of 126:

```
unix-prompt> a.out
Input a positive int between 0 and 127 inclusive: 126
```

Decimal 126 converted to base 2 is 1111110

unix-prompt>

Hint: Depending on your algorithm, you may find it useful to use the escape sequence “\b” which causes the cursor to move backwards one space.

8. Something loopy to think about

This is a discussion question, not a programming assignment, because an elegant solution to this requires the syntax introduced in the next chapter. It is meant to make you think about loops, the topic of the next chapter. How would you write a C++ program that could convert any unsigned integer to any base between 2 and 9? Using loops, this actually easier to do than the solution to the previous problem!

Here’s an example of how it should work for an input of 126:

unix-prompt> a.out

Input the base to convert to between 2 <= b <= 9: 7

Input any positive unsigned int: 126

Decimal 126 converted to base 7 is 240

unix-prompt>

Chapter 5

Loops

Loops are the third and final programming element we need to write any algorithm. The C++ language provides three “easy” ways to do this, the **while**, **do/while** and **for** loops.

5.1 The while loop

The general form of the **while** loop is:

```
while (logical expression)  
{  


|                  |
|------------------|
| STATEMENT BODY 1 |
|------------------|

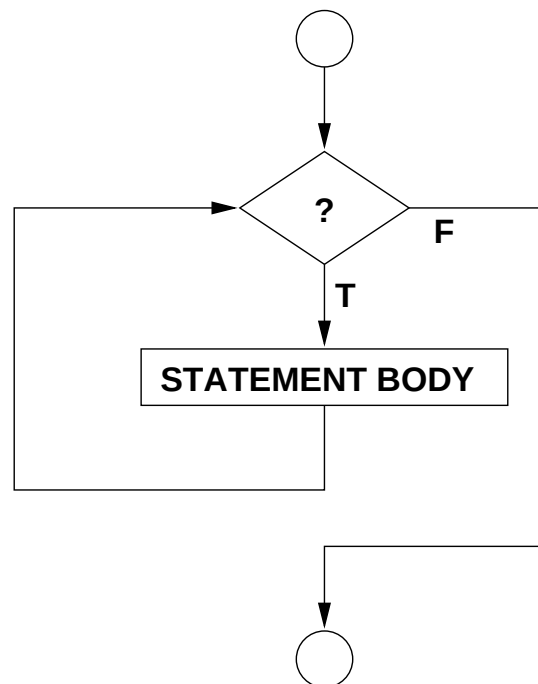
  
}
```

Some features of the **while** loop:

- If the **STATEMENT BODY** consists of one and only one statement, then the surrounding **{}**'s are optional. (It can help with clarity of programming to keep them, however. Their use is highly recommended.)
- Note the use of indenting.
- Processing control passes to the statement immediately following the **while** construct after the condition tested by the logical expression fails.
- **while** loops (in fact, any repetition loop) can be nested within each other
- The *logical expression* can mix arithmetic and logical operators to make for compact coding. (This is not really recommended for novice programmers and often makes code hard to read.)
- If the *logical expression* is false (even the first time), the **STATEMENT BODY** within the **while** loop is never executed.
- If the condition which is tested is **TRUE** and neither the **STATEMENT BODY** nor the *logical expression* modifies it so that it would become false, then it will run forever. This is called a “forever” loop, **while(1){}**. Forever loops are bad, anti-social things.

It puts your computer in a state known formally as “hung”, as in, “Jeepers, I think my computer is hung.” It is NOT a compliment! Hung computers are put out of their misery by a variety of desperate measures, increasing in severity. At the very worst, the power has to be shut down, a really nasty, last resort for a hung computer.

This is the flow chart for the `while` loop:



Here’s an example that tests Gauss’s summation formula:

```
//File: gauss.cpp
#include <iostream>

using namespace std;

int main(void)
{   cout << "Input N: ";
    int N;
    cin >> N;

    int i = 0, Sum = 0;
    while (i < N)
    {   i = i + 1;
```



```

        Sum = Sum + i;
    }

    cout <<
        "Sum of " << N << " digits is " << Sum
        << "...according to Gauss: " << N*(N+1)/2 << "\n";

    return(0);
}

```

Here's a more compact (but dangerous way) to do it:

```

//File: gaussAgain.cpp
#include <iostream>

using namespace std;

int main(void)
{   cout << "Input N: ";
    int N;
    cin >> N;

    int i = 0, Sum = 0;
    while (i = i + 1 <= N) Sum = Sum + i; //NO, NO, NO!

    cout <<
        "Sum of " << N << " digits is " << Sum
        << "...according to Gauss: " << N*(N+1)/2 << "\n";

    return(0);
}

```

This is dangerous because if you forgot the parentheses around the `i = i + 1` part of the conditional, you would generate an infinite loop! Don't believe me? Try it! Um, wait, this is a forever loop! OK, try it, but <CNTRL>-C out of it after a few seconds.

The point is—try to make your coding as easy to interpret as possible. Don't be too clever. Don't save space just for the purpose of saving space at the expense of clarity. In the old days of computing, memory and disk space were very expensive and precious. Programmers prided themselves at making things very compact. Those days are, happily, long gone. However, there still are people who do this. Maybe they think that an electron is as expensive as a Higgs' boson¹.

¹Electrons are essentially free. There are countless numbers of them in everyday things, like water. The

5.2 The do/while loop

The general form of the `do/while` loop is:

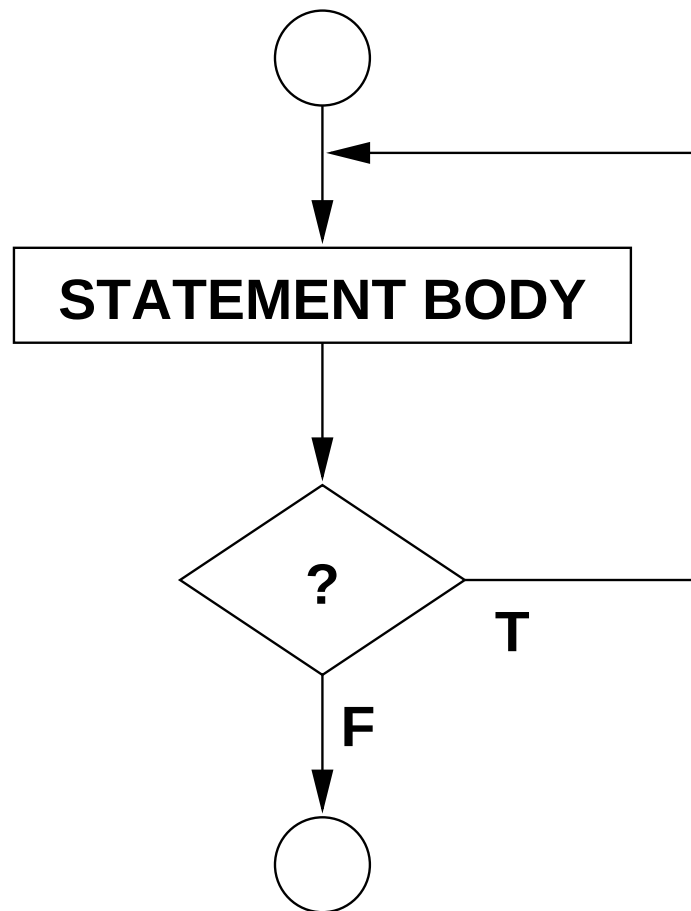
```
do
{
    STATEMENT BODY 1
}while (logical expression);
```

Some features of the `do/while` loop:

- Do not forget the “;” after the condition!
- If the **STATEMENT BODY** consists of one and only one statement, then the surrounding `{}`’s are optional. (It can help with clarity of programming to keep them, however. Their use is highly recommended.)
- Note the use of indenting.
- Processing control passes to the statement immediately following the `do/while` construct after condition tested by the logical expression fails.
- `do/while` loops (in fact, any repetition loop) can be nested within each other.
- The *logical expression* can mix arithmetic and logical operators to make for compact coding. (This is not really recommended for novice programmers and often makes code hard to read.)
- The **STATEMENT BODY** is executed at least once even if the *logical expression* is false initially.
- If the condition which is tested is true and neither the **STATEMENT BODY** nor the *logical expression* modifies it so that it would become false, then it will run forever. This is called a “forever” loop, `do{}while(1);`.

This is the flow chart for the `do/while` loop:

Higgs’ boson is very expensive. Physicists are spending billions to find it. It has not been found yet, but almost! The first person to find it will win the Nobel Prize for physics.



The only thing that really distinguishes a **do/while** loop from a **while** loop is that the **do/while** loop evaluates the conditional at the end of the **STATEMENT BODY** and the **while** loop evaluates it at the beginning. **while** loops are often used when incrementing a counter and using (and usually updating) the counter within the statement body. **do/while** loops are useful if you always want to process the **STATEMENT BODY** at least once. Because of this, loops that accept input from the keyboard (or a file) make good use of the **do/while** loop especially when the input is terminated using a “sentinel”.

Here is an example of sentinel controlled output for calculating the average of student grades. The sentinel for stopping is when the input for a grade is less than zero. Even the cruelest professor does not hand out a negative grades (even though it is legal)! So, this is a good sentinel to use for this application.

```
//File: grades.cpp
#include <iostream>

using namespace std;
```

```

int main(void)
{  int nStudents = 0, grade, sumGrades = 0;

    do
    {  cout << "Input a grade: ";
        cin >> grade;

        if (0 > grade) cout << "Grade < 0, end of input.\n";
        else
        {  nStudents = nStudents + 1;
            sumGrades = sumGrades + grade;
        }
    }while (0 <= grade);

    if (0 < nStudents)
        cout << nStudents
            << " grades were read in. "
            << "The class average is: "
            << sumGrades/nStudents
            << "\n";

    return(0);
}

```

Note that the above calculates the average as `sumGrades/nStudents`, the result of an integer division. This may not be completely fair as an average of, say, 79.9 should not really be reported as 79.

The way to “fix” this is to make the following changes:

1. Add the following `#include` in the pre-processor area of the code:

```
#include <iomanip>
```

2. Modify the `cout` statement as follows:

```

    cout << nStudents
        << " grades were read in. "
        << "The class average is: "
        << setw(5) //Set field width to 5
        << setprecision(3) //3 digits of precision
        << static_cast<float>(sumGrades)/nStudents
        << "\n";

```

The code in its entirety can be found in the lectures area on the web. It is called `grades1.cpp`.

There are two new features in the `cout` statement. One is the introduction to the *unary* “cast” operator `static_cast<float>(expression)` which converts `sumGrades` temporarily (and internally) to a float and the resulting calculation takes place as if it were a divide using floats alone. It is not necessary to make the same conversion for `nStudents`. The division of a float by an integer or an integer by a float always results in a floating point calculation as the integer participating is “promoted” to a float.

The general form of the cast operator is

```
static_cast<type_name>(expression) //Don't forget the parentheses!
```

to change `expression` to the type `type_name`. So, float's can be changed to int's and vice-versa. It is possible to change among other types of variables that we will encounter later.

The other new feature is the use of a “field width” and “precision” to modify the output of the float. The function `setw(m)` sets aside `m` spaces to print. The function `setprecision(n)` prints `n` significant digits. These functions are defined in the `iomanip` (I/O stream manipulator) library file. Hence, we had to include this file, else the compiler would have complained. Note how these functions are employed. `setprecision()` only has to be set once. The stream handlers remember it. However, `setw()` has to be set for every item!

There is another commonly used stream manipulator called `fixed`. We will not use it much.

5.3 The for loop

The `for` loop is similar to the `while` loop except that the syntax makes explicit the initialization, the looping condition and the loop counter modification. This form is generally preferred if the loop is to be performed in a systematic fashion, to a well-defined termination criterion with a regularly-spaced or easy-to-calculate loop counter indexing.

The general form of a `for` loop is:

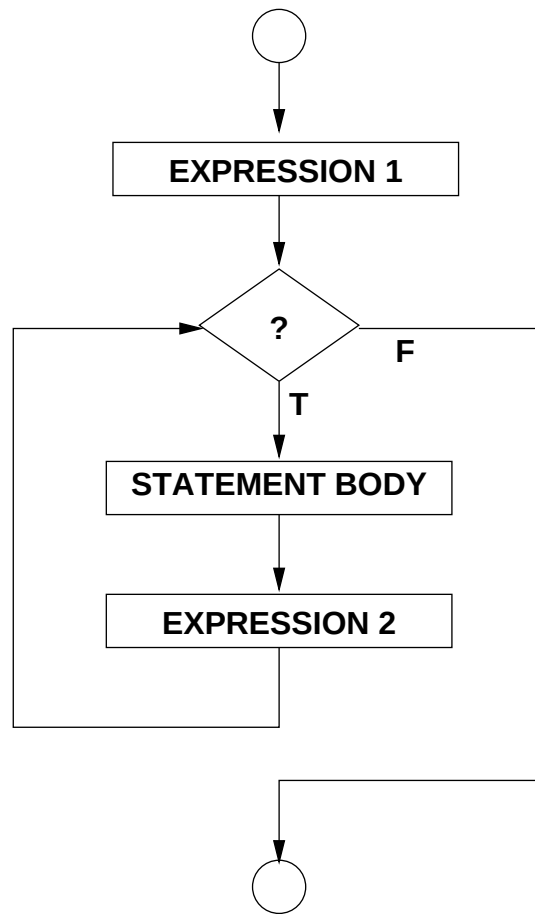
```
for (expression 1; logical expression; expression 2)
{
    STATEMENT BODY
}
```

Some features of the `for` loop:

- If the `STATEMENT BODY` consists of one and only one statement, then the surrounding `{}`'s are optional. (It can help with clarity of programming to keep them, however. Their use is highly recommended.)

- Note the use of indenting.
- Processing control passes to the statement immediately following the `for` construct after condition tested by the logical expression fails.
- `for` loops (in fact, any repetition loop) can be nested within each other.
- The `logical expression` is intended to provide the condition for which the execution of the `for` loop terminates. This `logical expression` can mix arithmetic and logical operators to make for compact coding. (This is not really recommended for novice programmers and often makes code hard to read.)
- The `STATEMENT BODY` will not be executed if the *logical expression* is false initially.
- `expression 1` is intended to initialize the loop counter (or index). It's use is optional. However, in this case the initialization of any counters (should they be used) should be done beforehand.
- The use of the `logical expression` is also optional. If not provided, the `STATEMENT BODY` is always entered. Another way of exiting the loop has to be provided otherwise the loop is infinite.
- `expression 2` is intended to increment the counter. The statement (or statements) are executed within the loop after the `STATEMENT BODY`. It's use is also optional. If not specified, the increments (or decrements) of any counters should be done within the `STATEMENT BODY`.
- The two “;”'s in the definition of the `for` loop are NOT optional. Leaving one or both out will result in a compilation error.

This is the flow chart for the `for` loop:



Here is an example that executes a count down and a “blast off”.

```
//File: blastOff.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main(void)
```

```
{  cout << "Estimate number of counter loops/second: ";
    int loopsPerSecond;
    cin >> loopsPerSecond;
```

```
    cout << "How many characters wide is your screen? ";
    int screenWidth;
    cin >> screenWidth;
```

```
    cout << "How many characters high is your screen? ";
```

```

int screenHeight;
cin >> screenHeight;

for (int i = 10; i >= 0 ; i = i - 1)
{ // Next line should waste about 1 second
    for (int j = 0; j <= loopsPerSecond; j = j + 1) ;
    cout << i << "\n";
}

cout << "\a.\n.\n.\nWe Have ignition!\a\n.\n.\n.\n";

// "Draw" a rocket
cout << " /\n\n"
    << " ||\n"
    << " ||\n"
    << " ||\n"
    << " ||\n"
    << "/|\n\n";

// Start of the vapor trail
cout << " **\n";
for (int j = 0; j <= loopsPerSecond/2; j = j + 1) ;
cout << "*****\n";
for (int j = 0; j <= loopsPerSecond/5; j = j + 1) ;

// Vapor trail expands
for (int i = 6; i <= screenWidth; i = i + 1)
{ for (int j = 1; j <= i; j = j + 1) cout << "*";
  for (int j = 0; j <= loopsPerSecond/i; j = j + 1) ;
  cout << "\n";
}

// Vapor trail at maximum
for (int i = 1; i <= screenHeight; i = i + 1)
{ for (int j = 1; j <= screenWidth; j = j + 1) cout << "*";
  for (int j = 0; j <= loopsPerSecond/screenWidth; j = j + 1) ;
  cout << "\n";
}

// Vapor trail contracts
for (int i = screenWidth; i > 0; i = i - 1)
{ for (int j = 1; j <= i; j = j + 1) cout << "*";

```



```
        for (int j = 0; j <= loopsPerSecond/screenWidth; j = j + 1) ;
        cout << "\n";
    }

    //Clear the screen
    for (int i = 1; i <= screenHeight; i = i + 1)
    {   for (int j = 0; j <= loopsPerSecond/screenWidth; j = j + 1) ;
        cout << "\n";
    }

    return(0);
}
```

5.4 Problems

1. Flowcharts revisited

Draw the flowchart representation (using lines, arrows, rectangles, diamond-shapes, and words) for the following looping constructs:

(a) the `for`-loop:

```
for (i = 10; i > 0; i = i - 1){Sum = Sum + i;}
```

(b) the `do/while` loop:

```
do{cin >> oddNumber; odd = oddNumber%2}while(!odd);
```

(c) the `while` loop:

```
while (different){z = x; y = x; x = temp; different = x - y;}
```

2. Loop tests

(a) How many times is the following loop body executed?

```
int i = 0;
do
{ i = i + 1;
} while (i < 0);
```

(b) How many times is the following loop body executed?

```
int i = 0;
while(i < 0)
{ i = i + 1;
}
```

(c) How many times is the following loop executed?

```
int i = 3;
while(i)
{ i = i - 1;
}
```

(d) How many times is the following loop body executed?

```
int i = 0;
while(i <= 0)
{ i = i - 1;
}
```

(e) How many times is the following loop executed?

```

int j = 1;
do
{   j = -2*j + 1;
}while(j != 0);

```

- (f) How many times is the following loop body executed?

```

for(int i = 0; i < 10; i = i + 2)
{   cout << "i = " << i << "\n";
}

```

3. Predict the output

The examples of code given below all compile correctly. Predict the output that the code would produce if you compiled and ran it.

(a)

```
int N = 0;
if (N = 0)
    cout << "The conditional expression was TRUE\n";
else
    cout << "The conditional expression was FALSE\n";
```

(b)

```
int f;
for(f = 0; f <= 10; f = f + 2)
    cout << f%10;
cout << "\n";
```

(c)

```
int i = 0, Sum = 0, N = 3;
do
{   i = i + 1;
    Sum = Sum + i;
}while(i <= N);
cout << "Sum = " << Sum << "\n";
```

(d)

```
int i = 0;
if (0 != i)
    cout << "Statement 1\n";
    cout << "Statement 2\n";
    cout << "Statement 3\n";
```

(e)

```
int j = 0;
if (0 < j);
{   j = j + 1;
}
cout << "j = " << j << "\n";
```

(f)

```
int i = 2, j = -2;
while (j <= i)
```

```

{   if (j < 0)
        cout << "i is " << i << ".";
        cout << " j is " << j << ".\n";
        i = i - 1;

    j = j + 1;
}

```

4. Loop drills

- (a) Convert the following code containing a **do/while** loop to a code containing a **while** loop in such a way that the execution of the code is identical.

```

int j = 0;
do
{   j = j + 1;
    cout << j << "\n";

}while (j <= 10);
cout << j << "\n";

```

- (b) Convert the following code containing a **while** loop to a code containing a **do/while** loop in such a way that the execution of the code is identical.

```

int j = 10;
while (j > 0)
{   j = j - 2;
    cout << j << "\n";
}
cout << j << "\n";

```

- (c) Consider the following code:

```

#include <iostream>

using namespace std;

int main(void)
{   int x = 1;
    for (int i = 1; i <= 10 ; i = i + 1)
    {   x = x*2;
        cout << x << " ";
    }
    cout << endl;
    return 0;
}

```

What does the above code print to the screen when you run it?

Draw the flow chart for the `for` loop in the above program.

- (d) Consider the following code:

```
#include <iostream>

using namespace std;

int main(void)
{   int i = 5;
    do
    {   i = i - 1;
        cout << i << " ";
    }while(i);
    cout << endl;
    return 0;
}
```

What does the above code print to the screen when you run it?

Draw the flow chart for the `do{}while()`; loop in the above program.

- (e) Consider the following code:

```
#include <iostream>

using namespace std;

int main(void)
{   int i = 13;
    while(i%2)
    {   i = i/3;
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

What does the above code print to the screen when you run it?

Draw the flow chart for the `while(){};` loop in the above program.

- (f) Consider the following code:

```
#include <iostream>

using namespace std;
```

```

int main(void)
{   for (int i = 1; i <= 2; i = i + 1)
        for (int j = 1; j <= 3; j = j + 1)
            cout << i << "," << j << endl;
    return 0;
}

```

What does the above code print to the screen when you run it?

Draw the flow chart for the nested for loops in the above program.

5. Think Like a Computer

Consider the following C++ code.

```

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{   float x;
    cout << "Input x (must be positive): ";
    cin >> x;

    int i = 0;
    const int NMax = 1000;
    const float SMALL = 0.0001;
    float xPower = 1.0, lastTerm, Sum = 0.0;
    do
    {   i = i + 1;
        xPower = x*xPower;
        lastTerm = xPower/i;
        Sum = Sum + lastTerm;
    }while (i < NMax && lastTerm > SMALL);

    if (i == NMax)
        cout << "Series did not converge\n";
    else
    {
        cout << "i          = " << i << "\n"
              << "x          = " << x << "\n"
              << "lastTerm = " << lastTerm << "\n"
              << "Sum          = " << Sum << "\n";
    }
}

```

```
    return 0;
}
```

- (a) What series is being summed by this code? `Sum = x + (you provide the rest)`
- (b) If the user inputs `-1.0` for `x`, what will be printed out by this code?
- (c) If the user inputs `0.0` for `x`, what will be printed out by this code?
- (d) If the user inputs `1.0` for `x`, what will be printed out by this code?
- (e) If the user inputs `0.0001` for `x`, what will be printed out by this code?
- (f) If the user inputs `0.001` for `x`, what will be printed out by this code?
- (g) If the user inputs `0.01` for `x`, what will be printed out by this code?
- (h) If the user inputs `0.1` for `x`, what will be value of `i` that is printed out by this code? (A hand calculator may help.)

6. Think Like a Computer Again

Consider the following C++ code.

```
#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{   float x;
    cout << "Enter x: ";
    cin >> x;

    int n = 0;
    float lastTerm;
    const float EPSILON = 0.001;
    float sign = 1;
    float denominator = 1;
    float total = 1;
    while(n <= 1 || lastTerm > EPSILON)
    {   n = n + 1;
        if (n/2*2 == n)
        {   denominator = denominator*n*(n - 1);
            lastTerm = pow(x,n)/denominator;
            sign = -sign;
            total = total + sign*lastTerm;
            cout << "Term " << n << " is " << lastTerm << "\n";
```

```

    }
}

cout << "The total is " << total << "\n";

return 0;
}

```

- (a) If you input 0.01 for x, what will be printed out?
- (b) If you input 1.0 for x, what will be printed out? (A hand calculator may help.)
- (c) What is another condition for the if statement that gives the same result?
- (d) What is the formula this program is implementing?

7. Predict and convert

- (a) **Prediction part:** If you compiled and ran this program, exactly what would the output look like?

```

#include <iostream>

using namespace std;

int main(void)
{
    const int maxTerms = 8;
    const int x = 2;

    int sign = 1;
    int term = 1;
    for (int i = 1; i <= maxTerms; i = i + 1)
    {
        sign = -sign;
        term = term*x;
        if (0 == i%2 || 2 <= i && 4 >= i)
        {
            cout << i << ":";
            if (0 > sign) cout << " - ";
            else cout << " + ";
            cout << term << endl;
        }
    }

    return 0;
}

```


- (b) **Conversion part:** Convert the program in the first part of this question to use a `do-while` loop rather than a `for` loop. Write your solution in the space below.

8. Predict and convert again

- (a) **Prediction part:** If you compiled and ran this program, exactly what would the output look like?

```
#include <iostream>

using namespace std;

int main(void)
{   const int maxTerms = 8;
    const int x = 2;

    int term = 1;
    int sum = 1;
    cout << "Sum = 1";
    for (int i = 1; i <= maxTerms; i = i + 1)
    {   term = -term*x;
        if (0 != i%3 && 0 != i%4 )
        {   if (0 > term) cout << " - " << -term;
            else cout << " + " << term;
            sum = sum + term;
        }
    }
    cout << " = " << sum << "\n";

    return 0;
}
```

- (b) **Conversion part:** Convert the program in the first part of this question to use a `while` loop rather than a `for` loop. Write your solution in the space below.

9. Fix the bugs

Consider the following C++ code. It is an attempt to do the sum:

$$S = 1 - x^2 + x^4 - x^6 + x^8 - x^{10}$$

The code has errors that the compiler detects as indicated in the comment lines. Fix them. The code also has “conceptual” errors in some lines, also indicated by comments. Conceptual errors are those that compile without error but do not provide correct answers upon execution. Fix these as well.

```

#include <iostream>
using namespace std;

int main(void)
{
    float x;
    cout << "Input x: ";
    cin >> x;

    float Sum, p; // Conceptual errors on this line
    int N = 10;
    do
    {
        i = i + 2; // Compiler detects error on this line
        p = p*x; // Conceptual errors on this line
        Sum = Sum + p;
    }while(i <= N); // Compiler detects error on this line
    cout << "Sum = " << Sum << "\n";

    return 0;
}

```

10. More bugs to fix

Consider the following C++ code. It is an attempt to do the sum:

$$S = 1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!} - \frac{x^{10}}{5!}$$

The code has errors that the compiler detects as indicated in the comment lines. Fix them. The code also has “design” flaws in some lines, also indicated by comments. Design flaws are those that compile without error but do not provide correct answers upon execution. Fix these as well. The corrected code should be able to compile and calculate the above sum.

```

using namespace std;
int main(void)
{
    cout >> "Input x:"; //Compiler detects error(s) on this line
    cin << x; //Compiler error(s) on this line
    double xPower = x; //Design flaw(s)
    double sum; //Design flaw(s)
    for(int term = 0; term <= 12; term = term + 1); //Design flaw(s)
        xPower = xPower + x*term; //Design flaw(s)
}

```

```

        sum = sum * xPower; //Design flaw(s)
    cout << "Sum = " << Sum << "\n"; //Compiler error(s) on this line
    return Sum; //Compiler detects error(s) on this line
}

```

11. Dealing cards

Complete the C++ code started below. The outer for loop deals 5 cards. The statement: `int rank = rand()%13 + 1;` chooses a random number between 1 and 13 inclusive. So, a 1 is an ace, a 2 is a deuce, and so on up to 10, 11 is a jack, 12 is a queen and 13 is a king. Then you must randomly pick the suit from one of 4: spades, clubs, diamonds and hearts. Your code must print out the results as in the example. Do not worry about having two cards the same as the deck your are dealing from is a 6-deck superdeck, six full decks of 52 cards each, so dealing identical cards is possible, but not very likely.

Here is an example hand that was dealt by the program.:

```

Card 1 is a queen of spades.
Card 2 is a 3 of spades.
Card 3 is an ace of diamonds.
Card 4 is a king of diamonds.
Card 5 is a 7 of clubs.

```

```

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main(void)
{
    srand(time(NULL));
    for(int nCards = 1; nCards <= 5; nCards = nCards + 1)
    {
        int rank = rand()%13 + 1;
        // You fill in the rest
        .
        .
        .
        .
    } // End of the for loop over the 5 cards
    return 0;
} // End of the main routine

```

12. Sentinel controlled input loops

Write a complete, standalone (if typed in, it would compile and run) C++ code that prompts a user for an integer but only accepts it if it is a positive, odd integer less than 100. If the user provides an integer that does not satisfy this, an explicit explanation is given and the user is prompted again. Your code has to provide output exactly as follows. Computer output is in **type font**, user input is in *italics*.

```
User! Type in an integer: 0
The integer must be > 0. Try again.
User! Type in an integer: 100
The integer must be < 100. Try again.
User! Type in an integer: 42
The integer must be odd. Try again.
User! Type in an integer: 41
Successful input!
```

13. Count the factors of 2

Write a C++ code that inputs a positive integer and prints out the number of factors of two in it. For example, an odd number has no factors of two, 2 has one factor of 2, $4 = 2 \times 2$ has 2 factors of 2, $6 = 3 \times 2$ has 1 factor of two, $8 = 2 \times 2 \times 2$ has 3, $10 = 5 \times 2$ has 1 factor of 2, $12 = 3 \times 2 \times 2$ has 2 factors of 2 and so on.

14. List the common factors

Write a C++ code that counts all the least common multipliers of a positive integer including 1 and the number itself. For example $8 = 1 \times 2 \times 2 \times 2$, $17 = 1 \times 17$, $123456789 = 1 \times 3 \times 3 \times 3607 \times 3803$. Here are sample uses of the program:

```
Input any positive int : 8
8 = 1 x 2 x 2 x 2
```

```
Input any positive int : 17
17 = 1 x 17
```

```
Input any positive int : 123456789
123456789 = 1 x 3 x 3 x 3607 x 3803
```

15. Compound interest

What happens if you deposit \$1000.00 in a bank account that accumulates interest at 5% per year, compounded annually? After the first year you have $\$1000.00 \times 1.05 = \$1050.00$ After the second year you will have $\$1050.00 \times 1.05 = \$1102.50$. Write a

program that will accept a positive float for the starting balance and output how much money you will have at the end of each year. The program stops when your money has at least doubled.

Here's an example of how it works:

```
% a.out
Input the starting balance: 1000

After year 1, balance = 1050
After year 2, balance = 1102.5
After year 3, balance = 1157.62
After year 4, balance = 1215.51
After year 5, balance = 1276.28
After year 6, balance = 1340.1
After year 7, balance = 1407.1
After year 8, balance = 1477.46
After year 9, balance = 1551.33
After year 10, balance = 1628.89
After year 11, balance = 1710.34
After year 12, balance = 1795.86
After year 13, balance = 1885.65
After year 14, balance = 1979.93
After year 15, balance = 2078.93
%
```

16. Oyster-Shucker's Savings and Loan

Oyster-Shucker's Savings and Loan has a great deal for you! This bank pays 1% interest per month, but only in months that do not have the letter "r" in its name (a rather strange but lucrative deal). Thus, for each month with an "r", the total amount of money is multiplied by 1.01. For months without an "r", the total is unchanged.

Write a program that:

- Reads in an `int` representing the first month the money is put in the bank,
- Reads in an `int` representing the number of months the money is in the bank,
- Reads in a `float` for the amount of money deposited,
- Loops over the number of months that the money is in the bank, and,
 - If the current month has an "r", multiplies the total by 1.01,
- And finally, prints out the final amount of money.

You may find the following useful: The statement

```
currentMonth = (startMonth + i) % 12;
```

calculates the month, given a starting month and a number of months elapsed. For example, if you start in January (Month 1), 13 months later it is February (Month 2).

17. There's Gold in Them Thar Ratios

In the Fibonacci series

1, 1, 2, 3, 5, 8, ...

each number (after the first two) is equal to the sum of the two before it. So, for example, the next number after 8 is $8 + 5 = 13$. One of the properties of the series is that the ratio of a number in the series to the one immediately preceding it converges (when the Fibonacci get very large) to the so-called Golden Ratio.

Write a program to approximate the Golden Ratio. Your program should compute terms in the Fibonacci series, taking the ratio of the most recent term to the one before it. Once the ratio is not appreciably different from the last time it was computed, terminate the loop and print out the result.

18. The Mersenne primes

A “prime number” is a positive whole number that can only be divided by itself and 1 to produce another positive whole number. The first few primes are 1,2,3,5,7,11... In C++ language the test:

```
if (0 == N%i)
```

would never be true for a candidate prime N if i were greater than 1 and less than N .

A Mersenne prime is a prime number that also has the form $2^p - 1$ where p is a positive number greater than 1.

Write a C++ code that calculates candidate Mersenne primes $2^p - 1$ for $2 \leq p \leq 31$. (This step is coded for you below.) Then test the number to see if it is prime. (You'll need a loop to do this.) If you detect that the number is prime, print out the number and the value of p .

19. Nest your for's, inside and out

Write a C++ program that will:

- (a) Prompt the user for a positive odd integer within a do-while construct. If the integer is 0, negative or even, prompt the user again. The algorithm goes to the next step when an odd positive integer is received. Call this odd integer N .
- (b) Output an inverted triangle of asterisks that looks like the following example. The first row has N *'s. Each following row has two *'s less than the one above it and is centered with respect to the row above.

Here's an example of how the program should work:

```
red% a.out
Input an odd positive int: -1
Input an odd positive int: 0
Input an odd positive int: 2
Input an odd positive int: 11
*****
*****
*****
*****
***
*
red%
```

You must use a `do{}while();` loop to obtain N.

You must use for-loops to make the drawing.

You may not use arrays.

You must only use `cout` statements that have only single character strings, for example, `cout << "*" ;`. Something like `cout << "**" ;` or `cout << " *" ;` is **not** permitted. Spaces are counted as a character.

Your program must work for arbitrary odd, positive N.

20. Nest your for's, left to right

Write a C++ program that will output the following:

```

** ** ** ** ** ** ** 
** ** ** ** ** *
*   ** ** ** ** *
*     ** ** ** *
*       ** ** ** *
*         ** ** ** *
*           ** ** ** *
*             ** ** ** *
*               ** ** ** *
*                 ** ** ** *
*                   ** ** ** *
*                     ** ** ** *
*                       ** ** ** *
*                         ** ** ** *
*                           ** ** ** *
*                             ** ** ** *
*                               ** ** ** *
*                                 ** ** ** *

```


In the output that follows, a C++ program prompted a user for two inputs, the height and width of a box and the user responded with “30” for both. Write the C++ program that did this:

[illegible]

You may not use arrays. (Ignore this statement if you do not know what an array is.)

You must only use `cout` statements that have only single character, for example, `cout << "*";`. Something like `cout << "**";` or `cout << " *";` is **not** permitted. Spaces are counted as a character.

22. Base Pythagorean Triples

A base Pythagorean triple is defined as three integers, i, j, k such that $i^2 + j^2 = k^2$ and i, j, k have no common divisors other than 1. For example 3, 4, 5 is a base Pythagorean triple. However, 6, 8, 10, although $6^2 + 8^2 = 10^2$, is not a base Pythagorean triple since 6, 8, 10 are all divisible by 2. Write a program that finds all the base Pythagorean triples such that $i, j, k \leq 1000$.

Here is an example of running the code:

```
Maximum integer to test = 1000
Triple # 1: 3*3 + 4*4 = 5*5 = 25
Triple # 2: 5*5 + 12*12 = 13*13 = 169
Triple # 3: 8*8 + 15*15 = 17*17 = 289
Triple # 4: 7*7 + 24*24 = 25*25 = 625
Triple # 5: 20*20 + 21*21 = 29*29 = 841
Triple # 6: 12*12 + 35*35 = 37*37 = 1369
Triple # 7: 9*9 + 40*40 = 41*41 = 1681
Triple # 8: 28*28 + 45*45 = 53*53 = 2809
Triple # 9: 11*11 + 60*60 = 61*61 = 3721
Triple # 10: 16*16 + 63*63 = 65*65 = 4225
Triple # 11: 33*33 + 56*56 = 65*65 = 4225
Triple # 12: 48*48 + 55*55 = 73*73 = 5329
Triple # 13: 13*13 + 84*84 = 85*85 = 7225
Triple # 14: 36*36 + 77*77 = 85*85 = 7225
Triple # 15: 39*39 + 80*80 = 89*89 = 7921
Triple # 16: 65*65 + 72*72 = 97*97 = 9409
Triple # 17: 20*20 + 99*99 = 101*101 = 10201
.
.
.
```

5.5 Projects

1. Something loopy to do

Write a C++ program that converts any unsigned integer to any base between 2 and 9.

Here's an example of how it should work for an input of 126:

```
unix-prompt> a.out
Input the base to convert to between 2 <= b <= 9: 7
Input any positive unsigned int: 126
Decimal 126 converted to base 7 is 240
```

```
unix-prompt>
```

2. Going nowhere fast

In this problem the program starts up assuming that it is following an object on a 2-dimensional plane as it takes many steps to perform a sort-of “random walk”. The object starts off at the “origin” ($x = 0$ and $y = 0$). You will input two real numbers that will represent the next coordinates of the object. The computer will calculate the straight-line distance that it has to travel to get there and position the object at that point. Then you will input two more real numbers representing a move from that location, then another position (another pair of points) and another position (pair of points) and so on. To stop the program you have to input the identical new coordinates as the previous step—that is, there is no distance to be traveled for that final step. This is a “sentinel” indicating that the loop should terminate. This last zero-step should not be counted in the number of steps. When the program exits the loop at this point make the program print out the total distance traveled, the total number of moves and the average distance per move.

Here is some pseudocode for this problem:

- (a) **START:** Declare $N = 0$ (Counter for the number of steps), x_i, y_i (New coordinates), $x_0 = 0, y_0 = 0$ (Current coordinates), s (Distance of a single step) and $t = 0$ (Total distance).
- (b) Start of the “input/stepping loop”.
 - i. Read in the the new coordinates: x_i and y_i .
 - ii. Test to see if x_i and y_i are valid input. If they are not, go back to (b-i.), otherwise continue on to the next step (b-iii.).

- iii. Test to see if x_i and y_i are identical to x_0 and y_0 . If they are not you will continue with step (b-iv.) Otherwise you will want to transfer control to the the first statement following the “input/stepping loop”, step (c).
 - iv. Increment the step counter N .
 - v. Calculate the distance of the step $s = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2}$.
 - vi. Accumulate s in t , the total distance.
 - vii. Important! Update the current position $x_0 = x_i$, $y_0 = y_i$.
 - viii. Try to take another step, that is return control to the top of the “input/stepping loop”, step (b-i.).
- (c) Report the total distance traveled.
 - (d) Report the total number of steps.
 - (e) Report the average distance traveled per step. Important: make sure that the case of zero steps and zero distance traveled does not cause problems.
 - (f) **END:**

3. The exponential function

(a) Computation of the factorial

Consider the following C++ program:

```
#include <iostream>
using namespace std;

int main(void)
{   cout << "Input N: "; int N; cin >> N;

    if (0 > N)
        cout << "N < 0. Stopping.\n";
    else if (0 == N)
        cout << "0! = 1\n";
    else
    {   int factorial = 1;
        for (int i = 1 ; i <= N; i = i +1)
            factorial = factorial*i;
        cout << N << "! = " << factorial << "\n";
    }

    return 0;
}
```

This program computes the “factorial” of the integer N , input by the user. The factorial of N , represented by the mathematical symbol $N!$, is calculated by:

$$N! = N \times (N - 1) \times (N - 2) \times (N - 3) \cdots 3 \times 2 \times 1$$

By definition, $0! = 1$. Other factorials are $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, $5! = 120$ and so on. Factorials are used a lot in many branches of mathematics.

Your problem is to find out for what values of N input by the user, does the program above not give correct answers. (Here’s a hint: The maximum signed integer has a value $2^{31} - 1$ or 2147483647.) The way you determine this I leave up to you. You can do it mathematically, or you can run the above program for different values of N until you detect that something is wrong. Turn in a description of how you found the answer.

(b) An improved computation of the factorial

Convert the above program so that the variable called `factorial` is a `float`. (Turn in your code.) Now, try again to find out for what values of N input by the user, does the program above not give correct answers. (Here’s another hint: The maximum `float` has a value $3.40282347 \times 10^{38}$.) You may determine this

mathematically, or you can run your program for different values of N until you detect that something is wrong. How do you know that something has gone wrong? Turn in a description of how you found the answer.

(c) **Calculating the exponential of a number**

The exponential function, called e^x is one of the most common functions in all of mathematics. One way of calculating it is to use the following:

$$e^x = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} \cdots + \frac{x^n}{n!} \cdots$$

an infinite series, one that never stops. Mathematically, the above expression for e^x is *exact* for any x from minus infinity to plus infinity. However, to get an accurate calculation from a computer requires some work. Here is what you have to do:

- i. Write a C++ program to compute e^x . Your program should try to compute it by summing the *finite* series

$$e^x = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} \cdots + \frac{x^N}{N!}$$

where N is some integer that you, the programmer, will set. Be careful that you do not set N too high, otherwise, as indicated in the previous two problems, your answer may not make sense. N should also be great enough so that your answer is as accurate as it can be. Turn in your code.

- ii. Discuss over what range of x you expect your program to provide a reasonably accurate answer. Turn in your discussion in such a way that pleases your Lab Instructor.

You can check your results with a calculator. Else, if you read ahead, you can check your program using the C++ math function `exp(x)`.

4. Primal loop therapy

This question is all about loops. A solution for this may involve all of the loops we have encountered: the `do{}while()` loop, the `for(){}` loop, and the `while(){}` loop. A solution also facilitated with the use of the modulus operator (%).

A prime number is a positive whole number that can be divided perfectly only by itself and one. The first 26 prime numbers are 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, and 97. You will write a C++ code that will accept two integers from its user. You will restrict the range of inputs to between 4 and 1,000,000 (one million). The user does not know this, he or she only knows to input integers and so you will keep prompting the user until he or she gives you proper input. (Sentinel-controlled `do{}while()` loops are best for this job.) The two integers can be accepted in any order, bigger one first or smaller one first. You can assume that the user is not “hostile”. The user will only input integers, not floats nor characters nor anything else that `cin` might find disturbing when asking for an `int`.

Once you have two integers you can work with, you will find all the prime numbers between those two numbers, including the starting and ending numbers themselves. You will output the prime numbers as you find them to the screen, and as a last step, print out how many of them you found.

Hint 1: Make sure that the primes your program gives agree with the ones in the list above.

Hint 2: There are 78,496 prime numbers between 4 and one million.

Hint 3: If you are using Hint 2 as a way of checking your program (please do!) and you find that it takes a very, very long time to get your program to complete its task, stop a moment and think! Think about how many potential divisors you have to test to see if a number is prime or not. It should not take more than about a minute or so to find all the primes between 4 and one million, depending on what machine you are using. On a 700 MHz PIII machine a solution takes about 70 seconds if it outputs every number it finds to the screen and only about 10 seconds if it outputs only the number of primes found. To be sure, there are even faster ways of doing it!

Finding prime numbers is an exercise that has intrigued mathematicians for centuries. As soon as computers were invented, this was one of the early problems that mathematicians programmed them for. We know that integers have only 32-bit representations. This is just food for thought, not part of the question: How you would find out if a number bigger than 2^{32} is prime? Now **that’s** a tough computer problem. Here is an example of output from correct solution:

```
> g++ primes.cpp
> a.out
Input the first integer: 1000001
```

```
Input must be > 4 or < 1000000
Keep trying!
Input the first integer: 1000000
Input the second integer: 1
Input must be > 4 or < 1000000
Keep trying!
Input the second integer: 4
```

Looking for all the prime numbers between 4 and 1000000 inclusive

```
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

```
.
.
.
```

```
999953 is prime
999959 is prime
999961 is prime
999979 is prime
999983 is prime
```

There were 78496 prime numbers between 4 and 1000000 inclusive

5. Where's Waldo?

It is a little known fact that Waldo Schtinklebaum, the character of the infamous “Where’s Waldo?” game was a student here at the University Michigan. He started off his career in Civil Engineering. He was notorious for 1) being a lousy dresser, 2) having no sense of direction (why he was always getting lost), 3) and being indecisive. In fact, after partying all night at the Frats, Waldo would often lose his way back to his dorm. He’d leave the party, forget which way his dorm was, take a step in a random direction, change his mind and his direction and take another step. This is not a bad approach to getting where you want to go, except that it can take a very long time to reach your destination. Waldo sometimes ended up walking around randomly until the morning when someone would take pity on him and lead him back home. In fact, Waldo’s meanderings led to his second career. Shortly after graduation, Waldo participated in the infamous Spring running event that takes place in the late evening on the last day of Winter term. On this day, Waldo changed his manner of dress—permanently. Despite his motivation to run in the correct direction, Waldo ended up in the Kresge Eye Center where his vision was corrected and he donned the hospital greens of a medical doctor. Waldo is now an optometrist working in Livonia under an assumed name. His specialty is treating children who have gone cross-eyed staring at the puzzles that bears his former name.

Your project is to write a program that imitates Waldo’s after-party mission to get home. A pseudocode description of the solution is given below.

Pseudocode for Waldo’s mission

- Waldo starts his journey at location $(x, y) = (0, 0)$, the front door of the Frat house, the center of the universe that Waldo is in. Waldo’s universe is a 2-dimensional plane that measures 20×20 , that is $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$. Waldo’s home is a 2×2 square in the upper right hand corner, $8 \leq x \leq 10$ and $8 \leq y \leq 10$.
- Declare a variable called `stillWalking` and give it an initial value of 1 (true), which means that Waldo is still trying to find his way home on foot.
- Declare another variable called `atHome` and give it an initial value of 0 (false) which means that Waldo has not yet found his way home.
- Declare another variable called `stepsToTry` and give it an initial fixed value of 1000. Waldo hopes to get home before 1000 steps. If he doesn’t, you will either tell him to keep going and try another 1000 steps, or to give up and sleep on the nearest park bench.
- **WHILE** (`stillWalking`)
 - **FOR-LOOP** until 1000 steps are taken and while Waldo has not yet reached home.
 - * **DO** (Loop for choosing the direction of the step)

- Take a proposed step of unit length in a random direction. (See note on the next page.)

WHILE (endpoint of the step is outside of Waldo's universe)

- * Take the step.
- * If Waldo's new location is within the square defined as his home, set `atHome` to true.

END FOR-LOOP

- **IF** (`atHome`), print out the total number of steps that Waldo has taken and stop the program.
- **ELSE**, read in a new value for `stillWalking`. If the user inputs a non-zero value for `stillWalking`, Waldo will try to take another 1000 steps, otherwise, if the user inputs a zero value for `stillWalking` the logical condition of the outer **WHILE** loop will be false.

END WHILE

- End the program.

Note 1: Taking a step in a random direction

To use random numbers, you must

```
#include <cstdlib>
```

in the preprocessor area for the code. Then, an expression of the form:

```
rand()
```

will produce a random `int` between 0 and `RAND_MAX` inclusive, where `RAND_MAX` is a big positive number defined in `cstdlib`. This random `int` must be converted to a random `double` that lies between 0 and 2π . This can be done via the following statement:

```
double angle = 2*PI*rand()/RAND_MAX;
```

where `PI` can be declared and initialized (earlier in the program) by

```
const double PI = 4*atan(1.0);
```

Using the above `atan()` function requires the use of

```
#include <cmath>
```

in the preprocessor area. Finally, x and y displacements can be generated by the statements:

```
dx = cos(angle);  
dy = sin(angle);
```

where `dx` and `dy` are double's. This is one of Waldo's steps. **You should use exactly this order of displacement statements in your solution.**

Example use of the program

```
red% a.out  
Will Waldo make it home? Press <RETURN> to continue.
```

```
Waldo is still lost after taking 1000 steps  
Continue playing? (Input "0" to stop): 1
```

```
Waldo is still lost after taking 2000 steps  
Continue playing? (Input "0" to stop): 1
```

```
Waldo made it home after taking 2552 steps!
```

Note 2: Making it really sort of random (optional topic)

If you have followed the above instructions carefully, you will notice that every execution of the program is exactly the same. Moreover, the solution of your colleagues, at least those who followed the instructions carefully, will also get Waldo home in the same number of steps (assuming you all are using the same type of computer). In this case, it is a desirable thing because, it allows your GSI's to grade your assignment quickly. This makes your GSI's happy because most GSI's would volunteer to undergo a frontal lobotomy rather than grade. (I used to be a GSI and I know that I would.) Happy GSI's give out better grades, it's a fact of life. I'm digressing again...

If you want to make each play of the game different, do the following. In the preprocessor area of your code,

```
#include <ctime>
```

Then, in your code before your first use of `rand()`, put in the following expression:

```
srand(time(NULL));
```

That's how I generated the different solutions that are shown in graphical form on the next page. The expression `time(NULL)` gives the number of seconds since the start of the *epoch*, January 1, 1970, 12:00am GMT. This is considered to be the start of the modern computer age.

Note that you can also "seed" the random number generator `rand()` using the following statement:

```
srand(someInt);
```

where `someInt` is an `int` that you can choose. If you choose the same `someInt` every time, the game will be the same. Different `someInt`'s result in different games. Try it!

Some things to think about (optional)

These are some things to think about. Don't hand answers to these to your GSI's. However, feel free to speak to them or me (the Prof) about it.

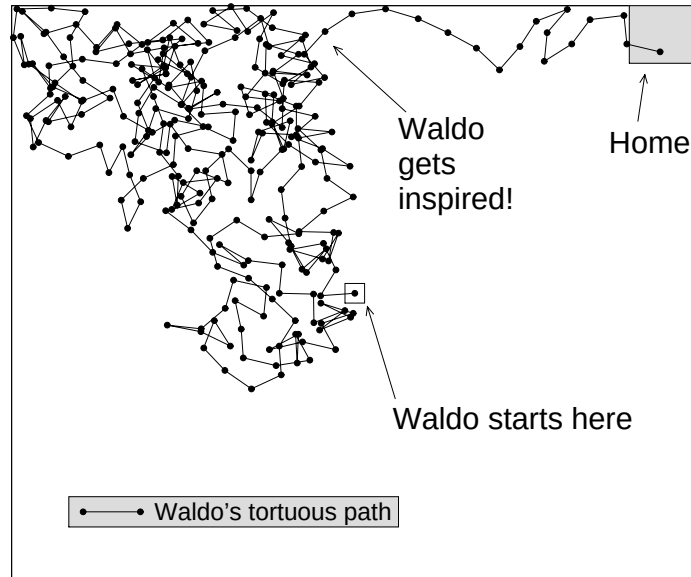
- Why did I confine Waldo to a 20×20 universe?
- Is Waldo always guaranteed to make it home if he tried long enough?
- If I took away the walls of the universe, on average how far away would Waldo be from his starting point after a certain number of steps.
- If you took away the walls, would Waldo always guaranteed to make it home if he tried long enough?

Graphical output of Waldo's meanderings on next page

Waldo is there. Can you see him?

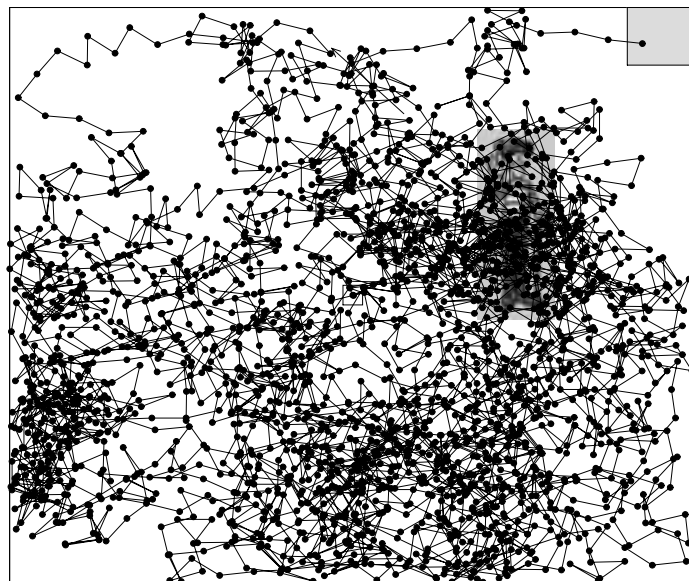
Wandering Waldo

First example



Wandering Waldo

Second example



Chapter 6

Early Abstraction—Functions

6.1 Motivation for functions

Let us consider an implementation of the al-Kashi algorithm. This algorithm computes the X^N where N is any integer, positive, negative or non-zero. Here is what the C++-code looks like (This code is called `alKashi.cpp` on the lecture distribution area on the course website):

```
//File: alKashi.cpp
#include <iostream>

using namespace std;

int main(void)
{   cout << "A calculation of X (float) to the power of N (int) \n";

    float X; // The float to be raised to a power
    int    N; // Power to be raised to
    cout << "Input X (float) and N (int): ";
    cin >> X >> N;

    float result; // The result of the calculation

    { //Open a new "scope" or "code" block

        //An implementation of the al-Kashi algorithm
        int    n = N; //An int used internally in this code block
        float x = X; //A float used internally in this code block
```

```

    if (0 > N)
    { //This "trick" allows us to calculate for negative powers!
        n = -N;
        x = 1/X;
    }

    result = 1; // Initial value
    while(1 <= n)
    { if (0 == n % 2)
        { n = n/2;
          x = x*x;
        }
        else
        { n = n - 1;
          result = result*x;
        }
    }
} // End of the scope block

cout << X << " to the power " << N << " = " << result << "\n";

return(0);
}

```

The above example introduces a new concept in C++-programming, that of a “block”, or “code block”. A “block” is any set of statements enclosed by a set of curly brackets {...}'s. In the above example,

```

{ //Open a new "scope" or "code" block
.
.
.
} // End of the scope block

```

delimit a code block—this one associated with the Al-Kashi algorithm. Variables can be defined within a code block. Those that are defined within them (in this example, `float x` and `int n`) are not known (or defined) outside of the the block. If you tried to do something with `x` or `n` outside of their block, you would get a compilation error, complaining that these variables were not defined. At least, that will be our understanding of code blocks until we get to the section called “scope rules” a little later in the course. We have actually seen code blocks before. The `main` routine has one, as well as the `if/else if/else` construct, the `while` and `do/while` constructs as well as the `for` loop.

Now, suppose that we did not know about the al-Kashi algorithm and coded the algorithm in a more straightforward way (This code is called `dumbPower.cpp` on the lecture distribution area on the course website.):

```
//File: dumbPower.cpp
#include <iostream>

using namespace std;

int main(void)
{   cout << "A calculation of X (float) to the power of N (int) \n";

    float X; // The float to be raised to a power
    int    N; // Power to be raised to
    cout << "Input X (float) and N (int): ";
    cin >> X >> N;

    float result; // The result of the calculation

    { //Open a new "scope" or "code" block

        //An implementation of the dumbPower algorithm
        int    n = N;
        float x = X;

        if (0 > N)
        {   n = -N;
            x = 1/X;
        }
        result = 1; // Initial value

        for (int i = 1 ; i <= n ; i = i + 1)
        {   result = result*x;
        }
    } // End of the scope block

    cout << X << " to the power " << N << " = " << result << "\n";

    return(0);
}
```

In these two examples, we note that there is a lot of repetition in the code outside of the scope block. Repetitious coding is **BAD!** It means that someone has wasted a lot of time!

Indeed, as far as the `main` routine is concerned, it does not really care about the contents of the code block! All it really wants is the answer. It can think of the algorithm to compute the power in a very abstract way. All `main` really needs to do is to give the scope block the variables `X` and `N`.

What we really, **REALLY** want is a routine that is independent of the specific calculational technique that is used to obtain X^N . In fact there is. There is a “function” in the C++ standard math library that does this. It is called `pow`. If we use this information we can write a much more compact code (This code is called `power0.cpp` on the lecture distribution area on the course website.):

```
//File: power0.cpp
#include <iostream>
#include <cmath>

using namespace std;

int main(void)
{   cout << "A calculation of X (float) to the power of N (int) \n";

    float X; // The float to be raised to a power
    int    N; // Power to be raised to
    cout << "Input X (float) and N (int): ";
    cin >> X >> N;

    float result; // The result of the calculation

    result = pow(X,N);

    cout << X << " to the power " << N << " = " << result << "\n";

    return(0);
}
```

Note the use of

```
#include <cmath>
```

that helps with the interpretation of the statement:

```
result = pow(X,N);
```

This is a much more *abstract* and *powerful* implementation of the previous two examples. It allows the main routine to be concerned with only the input and output of the various parameters, X , N , and the answer. The main routine in this case is much more general.

Let us pretend that we are now in the infancy of the C++-language, and that the `pow` function does not exist. However, we require this functionality repeatedly in our applications. But, we do know about al-Kashi's algorithm. Wanting to be efficient, we would code the al-Kashi algorithm into a C++-function and employ it as follows (This code is called `power1.cpp` on the classcodes distribution area):

```
//File: power1.cpp

#include <iostream>

using namespace std;

float pow(float x, int n)
{ float result; // The result of the calculation

    //An implementation of the al-Kashi algorithm

    if (0 > n)
    { n = -n;
      x = 1/x;
    }

    result = 1;
    while(1 <= n)
    { if (0 == n % 2)
      { n = n/2;
        x = x*x;
      }
      else
      { n = n - 1;
        result = result*x;
      }
    }
    return result;
}

int main(void)
{ cout << "A calculation of X (float) to the power of N (int) \n";
```

```

float X; // The float to be raised to a power
int    N; // Power to be raised to
cout << "Input X (float) and N (int): ";
cin >> X >> N;

float result; // The result of the calculation

result = pow(X,N);

cout << X << " to the power " << N << " = " << result << "\n";

return(0);
}

```

Now we note a few important things. There are a lot of things to remember about functions and we are only going to scratch the surface.

- In our use of functions, the function definitions and the main routine go in one file. (They can be put into different files but it becomes more complicated.)
- The function definitions go at the top of the file, after the pre-processor directives but before the definition of the main routine. (They can be put after the main routine but it becomes more complicated.)
- The functions definitions should go before they are used in any other function, including main. (Again, we can use another ordering, but it becomes more complicated.)
- Functions, the way we have used them so far, can return only one thing via the state-
ment

`return expression.`

- The `expression` in the return line must evaluate to a data type that is specified in the definition of the function, in this example, `float pow()`...
- The parameters in the function definition, in this example, `(float x, int n)`, must be declared.
- The variables in the function call, in this example, `X,N` from `main`'s function call `result = pow(X,N);` are not changed by the function. (There is another way to call functions where the parameters can change. This will be taught later.)

- Once the function returns to its calling routine, `main` in this case, the variables in the function's parameter list and any other variables that it declares, are lost. The only thing that `main` has is the return value. (To be absolutely truthful, there are ways that the calling routine can look at the memory where the function stored things temporarily, but...it's complicated! Moreover, it is almost never done.)
- Functions can call other functions.
- `main` is just another function.
- If a function does not return anything, its type is `void`. Example,

```
void returnNothingFunction(void){}
```

- If a function has no parameters, put `void` where the parameter list goes, as in the above example.
- Finally, functions are really, really, REALLY important. So it is important to understand them and use them well. We will spend a lot of effort in this course with functions.

6.2 User-defined functions

In this section, we examine the syntax associated with functions.

There are three things that are associated with the definition of a function in ANSI C++:

- The function prototype: For small applications, where all the source code is found in one file, the function prototype is usually written in the pre-processor area, the area before any function definition (see below). In small applications, where all the source code is found in one file, the function prototype can be omitted if the function definition is located in the file before any function call (see below) is made to it. For large applications, or applications that have more than one source file, function prototypes are usually gathered into an "include file", named, for example, `myIncludes.h` and accessed via an include statement `#include <myIncludes.h>` placed in the preprocessor area.
- The function definition: For small applications, where all the source code is found in one file, the function definition is written after the function prototype and outside of the main routine. For large applications, or applications that have more than one source file, it is conventional to have one function definition per source file.

- The function call: The call to the function can be inside of `main` or another function. In fact, a function can even call itself. This is called “recursion” and is discussed in a later chapter.

Understanding the syntax of these three things is crucial to the successful use of functions.

The general use of a user-defined function is as follows:

```
/* Opening comments */
.
.
.
// Start of pre-processor directives
.
.
.
// function prototype (if required)
(type) myFunction(variable type list separated by , 's);
.
.
.
// end of pre-processor directives
.
.
.
/* function definition */
(type) myFunction(variable type list and variables separated by , 's)
{
    .
    .
    .
    /* Function body with declarations and
       executable statements
    */
    .
    .
    .
}
.
.
.
int main (void)
{ // Start of main routine
```

```

.
.
.
// function usage
...myFunction(list of values separated by ','s)...;
.
.
.
} // End of main routine

```

General comments on functions:

- The function prototype is the declaration of a function. It specifies what the function is expected to provide (return value, see below), what is in the parameter list (inside the parentheses) and its identifier (name). You do not need a function prototype if all your source code is located in one file and if the first time the compiler encounters your function within a source code file is the function definition itself. The compiler will form its own prototype from the definition! This is why we can get away with not using prototypes in most cases when everything is in the same file, the function definitions are in the proper order and main comes last. There are exceptions to this rule but we will probably not encounter them in this course. If the compiler encounters the function for the first time in the function call, it will form its own prototype and assume that the return type is an `int` and can potentially mess up on the argument list as well. Avoid this situation at all costs!
- The `(type)` in the “function prototype” and “function definition” inform the compiler that the function called `myFunction` will return a variable of a certain type. `(type)` could be `float`, *e.g.*

```
float myFunction();
```

which means that `myFunction` is expected to return a `float`.

- Other possibilities for `(type)` could be `int` or any other variable type in C++ (that have not been discussed yet in this course).
- If `(type)` is not defined, then the compiler assumes that it is an `int`.
- The other possibility for `(type)` is `void`. `void` means that the function will not return a value. (It will perform a function on some data but not return any kind of value.)
- `(type)` in the “function prototype” and “function definition” must agree, else there will be a syntax error.

- Forgetting to return a value when `(type)` indicates that one should be returned results in unpredictable behavior.
- Returning a value when `(type)` is `void` causes a syntax error.
- Note that

```
int main (void)
{
    .
    .
    .
}
```

is a function definition! Its `(type)` is `int`. We could have said

```
main ()
{
    .
    .
    .
}
```

because functions are assumed to be `(type) int` unless we state otherwise. `main` is a special function in C++ that the operating system knows is the place where to start processing. Only `main` has this special status. I always `(type)` every function that I define and you should do the same.

- The rules for naming a function, *e.g.* `myFunction`, are the same rules as for variables, *i.e.* the normal *identifier* naming rules (≤ 31 alphanumeric characters and the underscore character “`_`” and NOT starting with a number.)
- The argument list in the function prototype is a comma separated list of variable types, *e.g.*

```
void myFunction(float, int, float);
```

which indicates `myFunction` will be called and defined with a `float`, an `int`, and a `float` in that order. Here’s another example:

```
void realQuadraticRoots(float, float, float);
```

A variable type is required for any non-int. Therefore, declaring a variable position as an int is optional. However, it is recommended that you declare the variable type for all members in the parameter list.

- The parameter list can also be `void`.
- The parameter list in the function definition follows the same rules except that the variables must be named. *e.g.*

```
void realQuadraticRoots(float a, float b, float c){/* Function definition */}
```

- The function call (within the body of the main routine or within the body of another function) as shown here:

```
int main (void)
{
    .
    .
    .
    realQuadraticRoots(a, b, c);
    .
    .
    .
}
```

transmits the *values* of the variables in the parameter list to the function. This is termed “call by value”. There is another way of doing this termed “call by reference” which we will discuss later.

- The variables defined in the function definition (as well as any other variables that the function defines) are internal to that function, even if the name (identifier) is *identical* to a name elsewhere in the program.
- Functions must all be defined in a separate code segment. You can not define a function within another function.
- Function prototypes outside of the body of any function apply to all function calls in that source file.
- Function prototypes inside the body of any function apply only to the associated calls within that function.
- A function can return in 3 ways:
 1. by running into its closing brace `}` (this will work only for `void` functions)
 2. running into the following statement anywhere in the function body

```
return;
```

This will work only for `void` functions.

3. running into the following statement anywhere in the function body

```
return (an expression);
```

This is the way non-`void` functions must return. The parentheses around “**an expression**” are optional. The expression should evaluate to the same variable type that the function is expected to return.

- Functions can be created independently by different programmers. The only information that has to be agreed upon is the return type, the function names and the parameter lists. Otherwise, each programmer can do his or her job however he or she pleases. In a team setting, prototypes are usually decided upon at the design stage. An example follows.

6.3 Example: A problem tackled with teamwork

(Completed version posted on the ENG101 web site in the lectures area called `quadraticRoots.cpp`)

A team of 4 people, Prof (the boss) and his graduate 3 students, Bill, Jane and Eugene (the math expert) are writing a program together.

- Prof to Team: *Team: We will write a code to solve the quadratic equation! We'll patent it and I'll get rich!*
- Team to Prof: *Yeah, right!*
- Prof to Bill: *Bill, I want you to write a `bool` function called `realQuadraticRoots` that will return an integer, and accept the three floats `a`, `b` and `c` in that order. If there are no real roots your function has to return the value `false`. If there are, it has to return the value `true`.*
- Bill to Prof: *Um, Prof, what's a `bool`??*
- Prof to Bill: *It's a variable type that can only have two values, `true` or `false`. It is declared and assigned value as follows:*

```
bool truthOrDare = false;
```

- Bill to Prof: *Gotcha, I'm on it. By the way, what's my cut?*
- Prof to Jane: *Jane, I want you to write a function called `numberQuadraticRoots` that will return an integer, and accept the three floats `a`, `b` and `c` in that order. I will only call your routine if Bill's routine tells me that there are real roots. The integer you return should contain the number of real roots, either “1” or “2”.*

- Jane to Prof: *OK, it's clear what I have to do.*
- Prof to Eugene: *Eugene, you do the math! I want you to...*
- Eugene to Prof: *You probably want me to write a function called something like `twoQuadraticRoots` that will return no value, and accept the three floats `a`, `b` and `c` in that order and print a statement telling you what the two roots are. You will only call your routine if Jane's routine tells you that there are two real roots. Right?*
- Prof to Eugene: *Eugene, stop interrupting! I want you to write a function called `twoQuadraticRoots` that will return no value, and accept the three floats `a`, `b` and `c` in that order and print a statement telling me what the two roots are. I will only call your routine if Jane's routine tells me that there are two real roots.*
- Prof to Team: *Get to work!*
- Team to Prof: *What's in it for us?*
- Prof to Team: *Stop kvetching already! Here's the main routine with function stubs. You guys fill in the blanks. My code compiles so it's absolutely perfect.*

Here's the Prof's main routine with "stub" functions:

```
//File: quadraticRootsStubs.cpp

#include <iostream>
//Do I need anything else, fellow programmers?

using namespace std;

//Function stubs
bool realQuadraticRoots(float a, float b, float c)
{
}
int numberQuadraticRoots(float a, float b, float c)
{
}
void twoQuadraticRoots(float A, float B, float C)
{
}

int main(void)
{ cout << "a, b, c: ";
  float a, b, c;
```

```

    cin >> a >> b >> c;

    if (realQuadraticRoots(a,b,c) && 2 == numberQuadraticRoots(a,b,c))
        twoQuadraticRoots(a, b, c);

    return 0;
}

```

Here is what the team produced.

Here's Bill's code. Note that the Prof did not fully specify his task and so he let him have it. He had to find a way to terminate the program from a subroutine and so he had to go to the library and do some research. (The Prof never answers e-mail from graduate students!) Bill learned how to do this with the `exit` function and other definitions given in `cstdlib`.

```

bool realQuadraticRoots(float a, float b, float c)
/*****
Purpose:          See if there are real roots
Receives:         Quadratic constants a,b,c in  $a*x*x + b*x + c = 0$ 
Returns:          true if real roots, false if none
Programmer:       Bill Boole (with attitude)
Start Date:       09/17/00

Remark1:          Needs <iostream>, <cstdlib>
Remark2:          Prof needs to rethink this thing
*****/
{  if (0 > (b*b - 4*a*c)) return false;
    else if (0 == a && 0 == b)
    {  cout << "Prof, you numbskull!\n"
        << "You didn't tell me what to do if a and b are both 0!\n"
        << "Go back and redesign the code!\n";
        exit(EXIT_FAILURE);
    }
    else return true;
}

```

Here's Jane's code. Her job was made easier by Bill's code letting her know that there is at least one solution.

```

int numberQuadraticRoots(float a,float b,float c)
/*****
Purpose:          See if there are real roots

```

Receives: Quadratic constants a,b,c in $a*x*x + b*x + c = 0$
 Returns: 1 if one real root, 2 if two real roots
 Programmer: Jane Justintime
 Start Date: 09/17/00

Remarks: Enters this routine knowing that there are roots
 *****/

```

{ if (0 == a || (0 == (b*b - 4*a*c))) return 1;
  else return 2;
}
  
```

Eugene, who thinks he's a genius and a super-geek, likes to do things his own way. He likes to store all the values his function receives in uppercase variables. Eugene has his own coding style. Eugene also thinks he is smarter than the compiler and so he played with the quadratic equation to put it into a form that he thinks will execute faster. (You can verify for yourselves that what Eugene did was correct mathematically.)

```

void twoQuadraticRoots(float A, float B, float C)
/*****
Purpose:      See if there are real roots
Receives:     Quadratic constants a,b,c in  $a*x*x + b*x + c = 0$ 
Returns:      The 2 two real roots of the quadratic equation
Programmer:   Eugene Dweebowski (the genius)
Start Date:   09/17/00

Remark1:      Needs <cmath>
Remark2:      Enters knowing that there are two real roots
Remark3:      I dare ya to find something faster and better!
*****/
{ float twoA = 2*A, twoC = 2*C, radical = sqrt(B*B - twoA*twoC) - B;
  cout << "Solution 1: " << radical/twoA << "\n";
  cout << "Solution 2: " << twoC/radical << "\n";
}
  
```

Some comments are in order for the Prof's code. It is incomplete. It does not do anything when there is only one solution. His code needs a little refinement before it can be considered complete. And, he has to decide what to do if all the constants are zero.

As a postscript to this story, function stubs are a good alternative to function prototypes for small projects. Note that the Prof's stub code will compile so that he can at least check his programming. However, for larger projects, prototypes are the way to go.

6.4 Call by value, call by reference, reference parameters

In our present usage of function calls, the parameter lists have contained only *values* that are transmitted to the function and stored in internal variables known only to that function. This is given the name *call by value*. A function that is called “by value” operates on the data it is given (plus whatever else can be transmitted to it by virtue of the rules scope) and can return to its calling routine at most *one value*, the return value, in a return statement of the form:

```
return returnValue;
```

This is unsatisfactory and restrictive if we need to return more than one value. An alternate way of calling functions, a way that can return more than one thing is called *call by reference*. This discussion must start, however, with a discussion of the addresses of variables, that is, where they are stored in a computer’s (virtual) memory.

6.4.1 The address of a variable

We introduce the unary operator called the address operator, `&`. To see its usage, consider the following program :

```
//File: addressLocal.cpp
#include <iostream>

using namespace std;

int main(void)
{ float  x = 1.0f; // Declare and initialize a float
  double y = 3.0;  // Declare and initialize a double
  int    z = 2;    // Declare and initialize an int
  cout <<
    "x's   value: " << x << "   x's address: " << &x << "\n"
    "y's   value: " << y << "   y's address: " << &y << "\n"
    "z's   value: " << z << "   z's address: " << &z << "\n";
  return 0;
}
```

New stuff:

- The address operator `&` applied to a variable, for example, `&y` should be read as “the address of the double `y`”.

Compiling and running it results in the following output:

```
x's   value: 1   x's address: 0xbffff734
y's   value: 3   y's address: 0xbffff72c
z's   value: 2   z's address: 0xbffff728
```

- When `cout` prints an address, it prints it out as `0x...` which is the hexadecimal (hex) representation. The leading `0x` is there just to tell you that what follows is in hex.
- Note the the address “stack” grows from a large address towards smaller addresses.

Note the the operating system (a Linux installation in this case) puts the first variable it encounters (in this case a 32-bit `float`) at address `0xbffff734`. (This address, once converted to an integer number, is something in excess of 3 billion.) This is the address in bytes, one byte being 8 bits. A 32-bit machine has a virtual address space of 2^{32} , or 4,294,967,296 bytes, often abbreviated as 4GB. Above this address, starting at `0xbffff738` in this case (This boundary is machine and O/S dependent) is where the “program environment” resides (we’ll discuss that later) and above that still is where the operating system resides (the kernel) occupying the upper reaches of address space up to the 4GB limit.

The next variable, a `double` is 64-bits or 8 bytes in length. Note that its address is smaller (by 8 bytes). The address space that is provided for local variables is called the “stack frame” and it grows downwards. Note where the next variable is located.

Each function creates its own stack frame for putting its local variables into memory, releasing it when the function terminates. This is what is meant by the stack getting “destroyed” when a function terminates. The next function call will use this space for its own stack frame.

The executable program, the “load module” is at the bottom of the address space, starting at byte 0. This area is called the “text segment”. Above the text segment are stored initialized and uninitialized global variables. Above this is the “heap space”, the address space where dynamic memory (memory allocated by the program) is located. It grows upwards. If your programs and data are so big that your stack frames and heap run into each other, you either have to redesign your program to use less space or purchase a more expensive 64-bit computer, one with 2^{64} bytes of array space. (In a few years, most computers will be 64-bit computers.)

Here another example program that demonstrates that global variables are located at a different place in memory:

```
//File: addressGlobal.cpp
#include <iostream>

using namespace std;
```

```

double y = 3.0; // Declare and initialize a global double
int     z = 2;  // Declare and initialize a global int

int main(void)
{ float x = 1.0f; // Declare and initialize a float
  cout <<
    "x's   value: " << x << "   x's address: " << &x << "\n"
    "y's   value: " << y << "   y's address: " << &y << "\n"
    "z's   value: " << z << "   z's address: " << &z << "\n";
  return 0;
}

```

Compiling and running it results in the following output:

```

x's   value: 1   x's address: 0xbffff734
y's   value: 3   y's address: 0x8049a70
z's   value: 2   z's address: 0x8049a78

```

Note how the global variables, `y` and `z` in this case, are located starting at a lower address `0x8049a70` (something like 1 billion). Note how the global stack grows up. Note also how the double, `y`, occupies 8 bytes.

6.4.2 Call-by-value *vs.* call-by-reference

Imagine for a moment that the `pow()` does not exist and we wish to write a program that computes the square and cube of an integer. Moreover, we wish to write a single function that returns both the square and cube. Using what we have learned so far, we are frustrated to discover that we can not do it! Instead we have to separate the task into two functions. Our code would look something like:

```

//File: returnTwice.cpp
#include <iostream>

using namespace std;

int square(int j)
{ cout << "In function square(), j is located at " << &j << "\n";
  return j * j;
}

```



```

int cube(int k)
{ cout << "In function    cube(), k is located at " << &k << "\n";
  return k * k * k;
}

int main(void)
{ int i = 3, iSquared, iCubed;
  cout
    << "\n\nIn main():\n"
    << "      i is located at " << &i          << "\n"
    << "iSquared is located at " << &iSquared << "\n"
    << "  iCubed is located at " << &iCubed   << "\n\n"
    << "Before the call to square and cube:\n"
    << "      i = " << i << "\n"
    << "iSquared = " << iSquared << "\n"
    << "  iCubed = " << iCubed << "\n\n";

  iSquared = square(i);
  iCubed   = cube(i);

  cout
    << "\n After the call to square and cube:\n"
    << "      i = " << i << "\n"
    << "iSquared = " << iSquared << "\n"
    << "  iCubed = " << iCubed << "\n\n";
  return 0;
}

```

where I have put in a lot of `cout` statements to let me know where the variables are stored. Compiling and running it results in the following output:

```

In main():
    i is located at 0xbffff734
iSquared is located at 0xbffff730
  iCubed is located at 0xbffff72c

Before the call to square and cube:
    i = 3
iSquared = 134520020
  iCubed = 134514715

In function square(), j is located at 0xbffff728

```

In function `cube()`, `k` is located at `0xbffff728`

After the call to `square` and `cube`:

```
    i = 3
iSquared = 9
iCubed = 27
```

Note that `main()`'s local variables `i`, `iSquared`, `iCubed` reside in `main()`'s stack frame. Also note that `iSquared` and `iCubed` are not initialized by `main()` and so the memory locations of these variables contain nonsense bits, junk left over from the previous use of those memory locations, whatever process that happened to be. The first function called is `square()`. It has one local variable called `j` which it locates in its own stack frame, starting at address `0xbffff728`. `square()` does its work happily and returns the result of its computation to `main()` which then called the function `cube()`. `cube()` has one local variable called `k` which it locates in its own stack frame, starting at address `0xbffff728`, *the same address used by `square()`!* Once `square()` returned to `main()`, its stack frame was destroyed and `cube()` did the most efficient thing, locating its stack frame at the next available location, the location just vacated by `square()`.

We now introduce the method of call-by-reference. This is best done by example, so, here is the above program converted to a single function call that has both the usual *value* parameters and also something new—*reference* parameters in the function's parameter list.

```
//File: returnTwoThings.cpp
#include <iostream>

using namespace std;

void squareCube(int j, int& j2, int& j3)
{  cout << "In function squareCube():\n"
    << "j  is located at " << &j  << "\n"
    << "j2 is located at " << &j2 << "\n"
    << "j3 is located at " << &j3 << "\n";
    j2 = j  * j;
    j3 = j2 * j;
    return;
}

int main(void)
{  int i = 3, iSquared, iCubed;
    cout
    << "\n\nIn main():\n"
    << "          i is located at " << &i          << "\n"
```

```

    << "iSquared is located at " << &iSquared << "\n"
    << "    iCubed is located at " << &iCubed    << "\n\n"
    << "Before the call to squareCube:\n"
    << "        i = " << i << "\n"
    << "iSquared = " << iSquared << "\n"
    << "    iCubed = " << iCubed << "\n\n";

    squareCube(i, iSquared, iCubed);

    cout
    << "\nAfter the call to squareCube:\n"
    << "        i = " << i << "\n"
    << "iSquared = " << iSquared << "\n"
    << "    iCubed = " << iCubed << "\n\n";
    return 0;
}

```

In the above example, in the parameter definition list (`int j, int& j2, int& j3`), the variable `j` is a value parameter receiving the value of the first parameter in the call statement, the `i` in `squareCube(i, iSquared, iCubed)`. In `squareCube()`, the variable `j` will be placed in `squareCube()`'s stack frame. The other two parameters in the parameter definition list, `int& j2` and `int& j3`, are reference parameters. What this means is that the `int` variables called `j2` and `j3` within the function `squareCube()` refer to the variables located at the addresses of the variables given to it in the function call. When the compiler calls the function `squareCube()` and sees that the parameter list in the function definition is expecting addresses, it says, *"Aha, squareCube() is expecting addresses for iSquared and iCubed"* and so the addresses are transferred to `squareCube()`. `squareCube()` puts these addresses in its stack frame. When the code inside `squareCube()` refers to the variables `j2` and `j3`, all it is really doing is renaming temporarily the memory locations referred to in `main()` as `iSquared` and `iCubed`.

Compiling and running the example results in the following output:

```

In main():
    i is located at 0xbffff734
iSquared is located at 0xbffff730
    iCubed is located at 0xbffff72c

Before the call to squareCube:
    i = 3
iSquared = 134519980
    iCubed = 134514715

```

```
In function squareCube():
j  is located at 0xbffff720
j2 is located at 0xbffff730
j3 is located at 0xbffff72c
```

After the call to squareCube:

```
    i = 3
iSquared = 9
iCubed = 27
```

Note the memory locations of the variables local to `main()` and the fact that `squareCube()` thinks that `j2` and `j3` live in the same place. Note also that in `squareCube()`, the local variable `j` is located at `0xbffff720`, 12 bytes below the end of `main()`'s stack frame. Yet, `j`, an integer is only 4 bytes long. Can you guess what use `squareCube()` is making of those 8 bytes of memory?

Finally, note that value parameters and reference parameters can be mixed in any order in the parameter list. Just make sure, in the function definition, that all the reference parameters names are preceded by the ampersand, `&` symbol.

About the only real-life example that I can come up with that relates to this is my relationship with my stockbroker. I am the `main()` routine and he, the incompetent bum, is a function called `stockBroker()`. Here is his function definition. I'll let you figure out how I'm doing on the stock market.

```
//File: stockBrokerFunction.cpp
```

```
void stockBroker(double myFee, double& alexMoney)
{ double transactionCost = 200.;
  while (0 < alexMoney && 0 < myFee)
  { myFee = myFee - transactionCost; //Deduct usual transaction charge
    alexMoney = alexMoney * 0.95; //Make a stupid investment
  }
  return;
}
```

However, I do want you to notice that I provide him with a fee, `myFee` which is his to keep. However, he is playing with my money, `alexMoney`, a reference parameter, the value of which, I am unhappy to report, is diminishing rapidly. It is still **MY** money, but I have given the stockbroker the ability to change it.

6.5 The rules of scope

One of the most powerful features of C++ is its implementation of the rules of scope. The philosophy is that information (data) should be hidden from operational code unless it really needs to know about the data. This avoids the danger of a program unit changing data that it is not supposed to—even if it calls the variables by the same name! This is how different people can develop different parts of a program and as long as each programmer stays within his or her assigned duties, data does not get corrupted. C++ has also introduced the concept of **namespace** to keep things separate. For this course we will always use: **using namespace std**; although it is possible to use a different namespace. We will only use the standard namespace in this course.

There are several hierarchies of scope:

file scope An identifier declared outside of any function definition has *file scope*. That is, “global” variables, function definitions and function prototypes (if you use them!) placed outside of any function apply for the entire file. It is perfectly fine to define an **int**, say, outside of any function or **main**. This variable can then be used by all functions and **main**. If separate files are linked together to form a program, the other files will not know about file scope identifiers outside of their own files.

function scope The only identifier that is defined for the entire function is a label (an identifier followed by a colon “:”), *e.g.*

ThisIsALabel:

A transfer to a label can be done with the dreaded **goto** or within the **switch** statement, neither of which will be taught in this course.

Labels are defined for the whole function and undefined outside of it.

block scope A “block” is anything within braces, `{...}`. For example:

```
{ // This is a block
}

{ // This is another separate block
}

{ // This is a "yet another block"
  { // This is still another block nested
    // within "yet another" block
  }
}
```

Note the use of optional spacing to delineate the separate blocks in the above.

We have seen many examples of “block”’s—function blocks, `if/else if/else` blocks, `while/do while/for` blocks. We can also define our own blocks by surrounding statements with braces, `{ ... }` to define a segment of code that has its own scope.

Nested scope is treated in a very special way. Variables defined outside a scope block will carry into a nested scope block unless the nested scope block declares a variable with the identical identifier. In this case the variable from the outer block is “hidden” until the inner block terminates. We will see examples of this.

function-prototype-scope One way of defining a function prototype is to include variable identifiers. For example:

```
int myFunction(float a, float b, float c);
```

is a function prototype that names three identifiers `a`, `b`, and `c`. These identifiers are only known within the function prototype. Often programmers name the variables they are expecting in a function prototype to remember which variables go in which order. These variable names are ignored by the compiler outside of the function prototype statement.

This is all very confusing unless we do some examples. So...let’s do some examples!

Consider the following code called `scope0.cpp`:

```
//File: scope0.cpp
#include <iostream>

using namespace std;

void myFn(float x, float y) // MyFn declares floats x and y
{ float z; // Internal to myFn
  cout << "myFn:    pre-op x = " << x << ", y = " << y << "\n";
  z = x; x = y; y = z; // Variable switch
  cout << "myFn:    post-op x = " << x << ", y = " << y << "\n";
}

int main(void)
{ float x = 1.0, y = 2.0; // Internal to main
  cout << "main:  pre-call x = " << x << ", y = " << y << "\n";
  myFn(x, y); // Main gives values to myFn
  cout << "main:  post-call x = " << x << ", y = " << y << "\n";
  return 0;
}
```

Compiling and running `scope0.cpp` results in the following output:

```
main: pre-call x = 1, y = 2
myFn: pre-op x = 1, y = 2
myFn: post-op x = 2, y = 1
main: post-call x = 1, y = 2
```

`main` declares two floats, `x` and `y` and gives `myFn` their values. `myFn` happens to call these variables by the same name (they could have been different). `myFn`'s job is to switch the values of these two variables. Within its own scope this has been accomplished but the switch has not been effected in `main`. This is because the `x` and `y` in `main` and the `x` and `y` in `myFn` are different.

Consider the following code called `scope1.cpp`:

```
//File: scope1.cpp
#include <iostream>

using namespace std;

float x = 1.0, y = 2.0; // Global variables

void myFn(void) // No declarations
{ float z; // Internal to myFn
  cout << "myFn: pre-op x = " << x << ", y = " << y << "\n";
  z = x; x = y; y = z; // Variable switch
  cout << "myFn: post-op x = " << x << ", y = " << y << "\n";
}

int main(void)
{ cout << "main: pre-call x = " << x << ", y = " << y << "\n";
  myFn(); // Main just calls myFn
  cout << "main: post-call x = " << x << ", y = " << y << "\n";
  return 0;
}
```

Compiling and running `scope1.cpp` results in the following output:

```
main: pre-call x = 1, y = 2
myFn: pre-op x = 1, y = 2
myFn: post-op x = 2, y = 1
main: post-call x = 2, y = 1
```

In this example `x` and `y` are defined globally (outside of any function). Both `main` and `myFn` can access them and change them. `myFn`'s job is to switch the values of these two variables. Within its own scope this has been accomplished and the switch has been effected in `main` as well. This is because the `x` and `y` in `main` and the `x` and `y` in `myFn` are identical.

Consider the following code called `scope2.cpp`:

```
//File: scope2.cpp
#include <iostream>

using namespace std;

float x = 1.0, y = 2.0; // Global variables

void myFn(float x, float y) // myFn declares x,y
{ float z; // Internal to myFn
  cout << "myFn:    pre-op x = " << x << ", y = " << y << "\n";
  z = x; x = y; y = z; // Variable switch
  cout << "myFn:    post-op x = " << x << ", y = " << y << "\n";
}

int main(void)
{ cout << "main:  pre-call x = " << x << ", y = " << y << "\n";
  myFn(x,y); // Main transfers x and y values
  cout << "main: post-call x = " << x << ", y = " << y << "\n";
  return 0;
}
```

Compiling and running `scope2.cpp` results in the following output:

```
main:  pre-call x = 1, y = 2
myFn:   pre-op x = 1, y = 2
myFn:   post-op x = 2, y = 1
main: post-call x = 1, y = 2
```

In this example `x` and `y` are defined globally (outside of any function). `main` can access them and change them. However, `myFn` defines its own `x` and `y` and hence the global ones are hidden and `myFn` cannot access them and change them.

Consider the following code called `scope3.cpp`:

```
//File: scope3.cpp
#include <iostream>
```



```

using namespace std;

int main(void)
{ float x = 1.0, y = 2.0, z = 0.0; // Local to main
  cout << " main:  pre-call x, y, z = "
        << x << ", " << y << ", " << z << "\n";
  { // Start of block
    float z = 42.0; // Local to {} block
    cout << "block:  pre-call x, y, z = "
          << x << ", " << y << ", " << z << "\n";
    z = x; x = y; y = z; // Variable switch
    cout << "block: post-call x, y, z = "
          << x << ", " << y << ", " << z << "\n";
  } // End of block
  cout << " main: post-call x, y, z = "
        << x << ", " << y << ", " << z << "\n";
  return 0;
}

```

Compiling and running `scope3.cpp` results in the following output:

```

main:  pre-call x, y, z = 1, 2, 0
block:  pre-call x, y, z = 1, 2, 42
block: post-call x, y, z = 2, 1, 1
main: post-call x, y, z = 2, 1, 0

```

In this example `x`, `y`, and `z` are defined local to `main`. Within `main` there is a nested block that defines its own scope. Since this block does not define `x` and `y`, these variables pass into the block. However, `z` is defined within the nested block and the external one is hidden. However, the nested block has changed the values of `x` and `y` when control is passed back to `main`.

Consider the following code called `scope4.cpp`:

```

//File: scope4.cpp
#include <iostream>

using namespace std;

void myFn(void)
{ float z = 0; // Internal to myFn
  cout << "myFn:  pre-op z = " << z << "\n";

```

```

    z = 1; // Reset z
    cout << "myFn:    post-op z = " << z << "\n";
}

int main(void)
{  cout << "First   call to myFn\n";
   myFn(); // Main calls myFn
   cout << "Second call to myFn\n";
   myFn(); // Main calls myFn again
   return 0;
}

```

Compiling and running `scope4.cpp` results in the following output:

```

First   call to myFn
myFn:    pre-op z = 0
myFn:    post-op z = 1
Second call to myFn
myFn:    pre-op z = 0
myFn:    post-op z = 1

```

In this example `z` is defined locally local to `myFn` which also initialized it to zero. Every time `myFn` is called, `z` is reset to zero and the executable statements are processed.

Consider the following code called `scope5.cpp`:

```

//File: scope5.cpp
#include <iostream>

using namespace std;

void myFn(void)
{  static float z = 0; // Internal to myFn
   cout << "myFn:    pre-op z = " << z << "\n";
   z = z + 1; // Add to z
   cout << "myFn:    post-op z = " << z << "\n";
}

int main(void)
{  cout << "First   call to myFn\n";
   myFn(); // Main calls myFn
   cout << "Second call to myFn\n";
   myFn(); // Main calls myFn again
}

```

```
    return 0;
}
```

Compiling and running `scope5.cpp` results in the following output:

```
First call to myFn
myFn:    pre-op z = 0
myFn:    post-op z = 1
Second call to myFn
myFn:    pre-op z = 1
myFn:    post-op z = 2
```

In this example `z` is defined locally local to `myFn` which also initializes it to zero but holds it `static`. The specifier `static` indicates that this variable is to retain its value when it leaves a routine, preserving its value for the next entry into the routine. The initialization to zero is only effective the first time the routine is entered.

`static` is called a *storage class* and is a qualifier on the variable type that means that its value is to be saved, not destroyed when the function ends execution.

Consider the following code called `scope6.cpp`

```
//File: scope6.cpp
#include <iostream>
using namespace std;

float max(float u, float v) // function max declares u,v
{ float temp = u;
  u = v;
  v = temp;
  cout << "max: u = " << u << " v = " << v << "\n";
  if (u > v)
  { return u;
  }
  else
  { return v;
  }
}

int main(void){
  float x = 1.0, y = 2.0, maxxy; // Internal to main
  cout << "main: x = " << x << " y = " << y << "\n";
  maxxy = max(x, y);
```

```
    cout << "main: max of x and y = " << maxxy << "\n";  
    return 0;  
}
```

Compiling and running `scope6.cpp` results in the following output:

```
main: x = 1 y = 2  
max: u = 2 v = 1  
main: max of x and y = 2
```

`main` declares two floats, `x` and `y` and gives `myFn` their values. `myFn` happens to call these variables by different names (they could have been the same as in some of the previous examples). `myFn`'s job is to switch the values of these two variables and determine which is greater. Within its own scope this has been accomplished but the switch has not been effected in `main`. This is because the `x` and `y` in `main` and the `u` and `v` in `myFn` are different.

6.6 Problems

1. Another essay question

- (a) In your own words numbering 50 or less, describe when you would want to make a function “call-by-reference”, rather than “call-by-value”.
- (b) Then give an example of a function call-by-reference that accomplishes something that you could **not** do using call-by-value.
- (c) Give an example of
 - i. the function prototype,
 - ii. the function call, and
 - iii. the function definition (with an empty statement block) that you would write for your example.

2. Simple functions

Type in the following main routine exactly as shown and add the definitions of two functions, `cubeit0` and `cubeit1`, each that provides the cube of the argument `x`. You may not use the `pow()` math library function.

```
#include <iostream>

//Function cubeit0 goes here

//Function cubeit1 goes here

int main(void)
{   cout << "Input a number to be cubed: ";
    double x;
    cin >> x;

    cout << cubeit0(x) << endl;

    double result;
    cubeit1(x,result);
    cout << result << endl;

    return 0;
}
```

Here is an example of how the program is intended to work:

```
% a.out
Input a number to be cubed: 3
27
27
%
```

3. Fix the bug(s)

- (a) Why won't the following code compile and/or work as intended? Fix the bug(s).

```
#include <iostream>

using namespace std;

void sumInts(int a, int b)
{   int i;
    i = a + b;
    return i;
}

int main() {
    int i = 1, j = 2;
    cout << "The sum of ints i = " << i << " and j = " << j << " is: "
         << sumInts(i,j) << "\n";
    return 0;
}
```

- (b) Why won't the following code compile and/or work as intended? Fix the bug(s).

```
#include <iostream>
#include <cmath>

using namespace std;

float trigs(float y, &f1, &f2)
{   f1 = sin(y); f2 = cos(y);
    return 0;
}

int main(void)
{   float x = 0.0, cosx, sinx;
    int returnValue = trigs(x, sinx, cosx);
    cout << "sin(" << x << ") = " << sinx << ", "
         << "cos(" << x << ") = " << cosx << "\n";
}
```

```
    return 0;
}
```

4. Predict the output

What does this code print?

```
#include <iostream>

using namespace std;

void cyclic(int i, int& j, int k, int& l)
{   int m;
    m = i;
    i = j;
    j = k;
    k = l;
    l = m;
}

int main(void)
{   int i = 1, j = 2, k = 3, l = 4;
    cyclic(i, j, k, l);
    cout << "i,j,k,l : " << i << j << k << l << "\n";
    return 0;
}
```

5. Fix the design error(s)

- (a) This code attempts to define a function that returns the maximum of two floats and orders the two floats in descending order (bigger first, smaller second). This code only works some of the time. There is one design mistake that occasionally leads to incorrect results. Assume that the `scanf` statement is always executed correctly. You may **not** fix this using a library function that determines maximum.

```
#include <iostream>

using namespace std;

float fMax(float f1, float f2)
{   if (f2 > f1)
    {   float temp = f2; // f2 is maximum
        f2 = f1; f1 = temp; // switch them
        return f1;
    }
}
```

```

    }
    else return f1; // f1 must be maximum, no need to switch
}

int main(void)
{   cout << "Input 2 float's: ";
    float a, b;
    cin >> a >> b;
    cout << "Maximum is: " << fMax(a,b) << "\n";
    cout << "Descending order is: " << a << ", " << b << "\n";
    return 0;
}

```

- (b) The following code compiles error free. Why won't it work as intended?

```

#include <iostream>

using namespace std;

void sort3(int i, int j, int k)
{   int t;
    if (i > j) {t = i; i = j; j = t;}
    if (j > k) {t = j; j = k; k = t;}
    if (i > j) {t = i; i = j; j = t;}
}

int main(void)
{   int i = 5, j = 3, k = 1;
    sort3(i,j,k);
    cout << "In ascending order (smallest->largest) "
          << "the three int's are "
    << i << ", " << j << ", " << k << "\n";
    return 0;
}

```

- (c) The following code compiles error free. Why won't it work as intended?

```

#include <iostream>

using namespace std;

float squareCube(float x, float xx)
{   xx = x*x;
    return xx*x;
}

```



```

    }

    int main(void)
    {   float x = 2.0, xx, xxx;
        xxx = squareCube(x,xx);
        cout << "Square of x: " << xx << ", Cube of x is " << xxx << "\n";
        return 0;
    }

```

6. Roots of the quadratic equation

Write a function that is compatible with the following function call:

```
bool twoRealRoots = twoRoots(a,b,c,root1,root2);
```

The function returns **true** if $b^2 - 4ac > 0$ and $a \neq 0$. Otherwise, it returns **false**. If the function returns **true**, then **root1** and **root2** contain the roots of the quadratic equation $ax^2 + bx + c = 0$, which are

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

7. Write your own $\sqrt{\quad}$ function

The following algorithm may be used to calculate $x = \sqrt{a}$:

- (a) Start with an initial guess: $x = a$
- (b) Refine the guess: $x' = (x + a/x)/2$
- (c) If x' and x are the same, $x = \sqrt{a}$, otherwise, let $x = x'$ and go to step 2.

Write a function that calculates $x = \sqrt{a}$ by the above method.

8. Polar coordinates

Write a function that is compatible with the following main routine:

```

#include <iostream>
#include <cmath>
#include <string>

using namespace std;

//Function definition goes here

int main(void)

```

```

{ float x, y, r, t;
  cout << "x, y: ";
  cin >> x >> y;

  cout << "x = " << x << " y = " << y << " is in the " << polar(x,y,r,t)
        << ". " << "r = " << r << " t = " << t << " degrees\n";

  return 0;
}

```

The function, `polar()`, called in the second `cout` statement above returns a string that indicates

- “upper right quadrant” if $x \geq 0$ and $y \geq 0$
- “upper left quadrant” if $x < 0$ and $y \geq 0$
- “lower left quadrant” if $x < 0$ and $y < 0$
- “lower right quadrant” otherwise

After the function returns, the `float` called `r` contains the radius computed as $r = \sqrt{x^2 + y^2}$. Hint. `r = sqrt(x*x + y*y);` After the function returns, the `float` called `t` contains the angle in degrees computed as $\theta = 180 \tan^{-1}(y/x)/\pi$. Hint. Use `t = 45*atan2(y,x)/atan(1.0);` to calculate this.

Here is an example of how it works:

```

red% a.out
x, y: 1 1 //User types in "1 1" in response to the "x, y: " prompt
x = 1 y = 1 is in the upper right quadrant. r = 1.41421 t = 45 degrees

```

9. How many bits to an unsigned int?

The following algorithm may be used to determine the number of bits that are used to represent an unsigned int:

- (a) Starting with an unsigned int initialized to 1, keep multiplying it by 2 until it equals 0.
- (b) The number of times you multiply it by 2 is related to the number of bits used to represent the unsigned int's.

Write a function that determines the number of bits that are used to represent an unsigned int on the computer that executes the program. Your function has to be compatible with the following main routine.

```
#include <iostream>

using namespace std;

//Function definition goes here

int main(void)
{  cout << "The unsigned int size is " << intSize()
    << " bits on this computer\n";
    return 0;
}
```

Here is how it works:

```
red% a.out
The unsigned int size is 32 bits on this computer
```

6.7 Projects

1. Math library functions

The following table lists some of the commonly used functions in C++’s math library which can be accessed by employing `#include <cmath>` in the preprocessor area of a C++ file:

Math function	Purpose	Example usage
$\log(x)$, $\log_e(x)$, $\ln(x)$	natural logarithm (base e)	<code>double x = 1.; double y = log(x);</code>
$\exp(x)$, e^x	exponential	<code>double x = 1.; double y = exp(x);</code>
$\log_{10}(x)$	logarithm (base 10)	<code>double x = 1.; double y = log10(x);</code>
$\sin(x)$	sine of x	<code>double x = 1.; double y = sin(x);</code>
$\cos(x)$	cosine of x	<code>double x = 1.; double y = cos(x);</code>
$\sqrt{x}$	square root of x	<code>double x = 1.; double y = sqrt(x);</code>
$ x $	absolute value of x	<code>double x = 1.; double y = fabs(x);</code>

Note that the argument to the trigonometric functions above must be in terms of *radians*.

360° (degrees) = 2π (radians).

Write a main program that generates the table on the following page similar to that shown. In other words, you will have to have a `for`-loop that somehow generates the `double`’s -1.0, -0.9, ..., 1.0, calculate and output `x`, `log(x)`, `exp(x)`, `log10(x)`, `sin(PI*x)`, `cos(PI*x)`, `sqrt(x)`, and `fabs(x)`.

Some notes:

- Note the argument of the `sin()` and `cos()`. You were given the method to generate a good value for π in the “Where’s Waldo?” project in the previous chapter.
- You will have to use the `setw()` and `setprecision()` functions as described in Chapter 5.
- You need one other thing to allow the printing of the redundant trailing zero’s. Put in the statement

```
cout.setf(ios::fixed);
```

before your first use of `cout`.

- Note that the outputs `-Infinity` and `NaN` are provided automatically by `cout` when the argument to the math library functions is out of range of its definition.

x	log(x)	exp(x)	log10(x)	sin(PI*x)	cos(PI*x)	sqrt(x)	fabs(x)
-----	-----	-----	-----	-----	-----	-----	-----
-1.0	-Infinity	0.367879	-Infinity	-0.000000	-1.000000	NaN	1.0
-0.9	-Infinity	0.406570	-Infinity	-0.309017	-0.951057	NaN	0.9
-0.8	-Infinity	0.449329	-Infinity	-0.587785	-0.809017	NaN	0.8
-0.7	-Infinity	0.496585	-Infinity	-0.809017	-0.587785	NaN	0.7
-0.6	-Infinity	0.548812	-Infinity	-0.951057	-0.309017	NaN	0.6
-0.5	-Infinity	0.606531	-Infinity	-1.000000	0.000000	NaN	0.5
-0.4	-Infinity	0.670320	-Infinity	-0.951057	0.309017	NaN	0.4
-0.3	-Infinity	0.740818	-Infinity	-0.809017	0.587785	NaN	0.3
-0.2	-Infinity	0.818731	-Infinity	-0.587785	0.809017	NaN	0.2
-0.1	-Infinity	0.904837	-Infinity	-0.309017	0.951057	NaN	0.1
0.0	-Infinity	1.000000	-Infinity	0.000000	1.000000	0.000000	0.0
0.1	-2.302585	1.105171	-1.000000	0.309017	0.951057	0.316228	0.1
0.2	-1.609438	1.221403	-0.698970	0.587785	0.809017	0.447214	0.2
0.3	-1.203973	1.349859	-0.522879	0.809017	0.587785	0.547723	0.3
0.4	-0.916291	1.491825	-0.397940	0.951057	0.309017	0.632456	0.4
0.5	-0.693147	1.648721	-0.301030	1.000000	0.000000	0.707107	0.5
0.6	-0.510826	1.822119	-0.221849	0.951057	-0.309017	0.774597	0.6
0.7	-0.356675	2.013753	-0.154902	0.809017	-0.587785	0.836660	0.7
0.8	-0.223144	2.225541	-0.096910	0.587785	-0.809017	0.894427	0.8
0.9	-0.105361	2.459603	-0.045757	0.309017	-0.951057	0.948683	0.9
1.0	0.000000	2.718282	0.000000	0.000000	-1.000000	1.000000	1.0

2. Strive for Perfection

A “perfect” number is a positive whole number that is the sum of its “proper divisors”. The set of proper divisors of number N does not include N itself, but does include 1. For example, the proper divisors of 6 are 1,2,3 and $1 + 2 + 3 = 6$. So, 6 is a perfect number. In fact, it is the first perfect number. The second perfect number is $28 = 1 + 2 + 4 + 7 + 14$. The first 4 perfect numbers were known to the ancient Greek mathematicians. In fact, only 37 of them are known today! A lot is known about perfect numbers but much more remains a mystery. For example, the following is known.

- (a) The biggest known perfect number, the last discovered, can be written $2^{3021376} \times (2^{3021377} - 1)$, a number that contains 1,819,050 decimal digits.
- (b) All known perfect numbers are even.
- (c) All known perfect numbers are given by Euclid’s formula: $N = 2^{k-1}(2^k - 1)$ where k is some positive integer and $2^k - 1$ also happens to be a prime number. This is a special kind of prime number known as a *Mersenne* prime. It is a **really good idea** to make use of this fact in designing your computer program, described below.

A lot is not known about perfect numbers. It is not known whether or not

- (a) there is an infinite number of perfect numbers.
- (b) all perfect numbers are even.
- (c) all perfect numbers are given by Euclid’s formula.

Your job is to write a program that will:

- (a) find the first 5 perfect numbers and print out their proper divisors.
- (b) be composed of at least 2 functions additional to `main`, one that returns `true` if its integer argument is a perfect number and a second function that prints out the proper divisors of a perfect number. You may define more than 2 additional functions if you wish.
- (c) complete execution in less than one minute of CPU time on a run-of-the-mill, modern computer.
- (d) have output that looks similar to the following:

```
% a.out
```

```
6 is a perfect number
6 = 1 + 2 + 3
```

28 is a perfect number
 $28 = 1 + 2 + 4 + 7 + 14$
.
.
.

3. Random integration

(a) You'll need this later!

Write a function that accepts 3 double's (x , x_0 and x_1 below) and returns the double ($f(x, x_0, x_1)$ below) for the function:

$$f(x, x_0, x_1) = \frac{x \sin(x)}{1 + x} \quad \forall \quad x_0 \leq x \leq x_1$$

If x is outside of the range of definition, a NaN must be returned.

Hint 0: A NaN is IEEE's way of saying "Not a Number". NaN's can be produced when the result of calculation using one of the floating-point representations is undefined. For example, the coding:

```
double zero = 0; cout << zero/zero;
```

produces a NaN. Try it!

Hint 1: Here's a main routine you can use to test your function:

```
#include <iostream>
#include <cmath>

// The function should go here...

int main(void)
{ cout << "Input x, x0, x1: ";
  double x, x0, x1;
  cin >> x >> x0 >> x1;
  cout << "Evaluating the function at x = " << x
        << " between the limits x0 = " << x0
        << " and " << x1 << "\n";
  cout << "f(" << x << ") = " << myFunction(x, x0, x1) << "\n";
  return 0;
}
```

(b) A random way to find an area

- i. Write a function that will find the position, $x_{\max}$, which is the position where the maximum of a function $f(x)$ occurs. This maximum lies between two limits x_0 and x_1 where $x_1 > x_0$. Your value for $x_{\max}$ has to be within a tolerance of $10^{-12}/|x_1 - x_0|$. You can assume that the function has only one local maximum between x_0 and x_1 , that the function $f(x) \geq 0$ between x_0 and x_1 and that the function is not a constant.

Hint 0: Use `double`'s, not `float`'s to work on your solution.

Hint 1: Life is made easy because there is only one local maximum between x_0 and x_1 . That means that you can start with any x such that $x_0 \leq x \leq x_1$ and move a little in one direction, say, $x - \delta$, where δ is some small positive number. If the function is smaller there, that is, $f(x - \delta) < f(x)$, you are testing for the maximum in the wrong direction, so try $x + \delta$.

Hint 2: Life is made difficult because the tolerance is so small. It is not feasible to set your initial delta to $10^{-12}/|x_1 - x_0|$. You will have to start with something much larger, say, $10^{-1}/|x_1 - x_0|$. Once you have found the position of the maximum within this tolerance, lower it to $10^{-2}/|x_1 - x_0|$ and so on. This *incremental* refinement technique is reasonably efficient. This is only a suggestion. You can probably think of a much more efficient way to do it.

- ii. Write another function that determines the area under $f(x)$ between the two limits x_0 and x_1 using a technique called *random sampling*. Random sampling will be discussed in class. You will need to know the position of the function maximum before you can determine the area under the function.
- iii. Complete the problem by using the following function:

$$f(x) = \frac{x \sin(x)}{1 + x}$$

for $0 \leq x \leq \pi$, most of which was coded in the last problem. Use 100,000 samples to estimate the area.

HOW TO WORK ON THIS PROBLEM:

- i. Try a few different examples for $f(x)$ for which you know the answer (without knowing calculus!). Good candidates would be:
 - A. $f(x) = x$, for $0 \leq x \leq 1$.
 - B. $f(x) = 1 - x$, for $0 \leq x \leq 1$.
 - C. $f(x) = x$, for $0 \leq x \leq 1/2$, $f(x) = 1 - x$, for $1/2 \leq x \leq 1$.
- ii. Once you are confident that your algorithm works, then try the assigned function:

$$f(x) = \frac{x \sin(x)}{1 + x}$$

for $0 \leq x \leq \pi$

- iii. Write out a pseudocode to help you design your program.

Chapter 7

More Variable Types, Data Abstraction

7.1 Representation of floating-point numbers

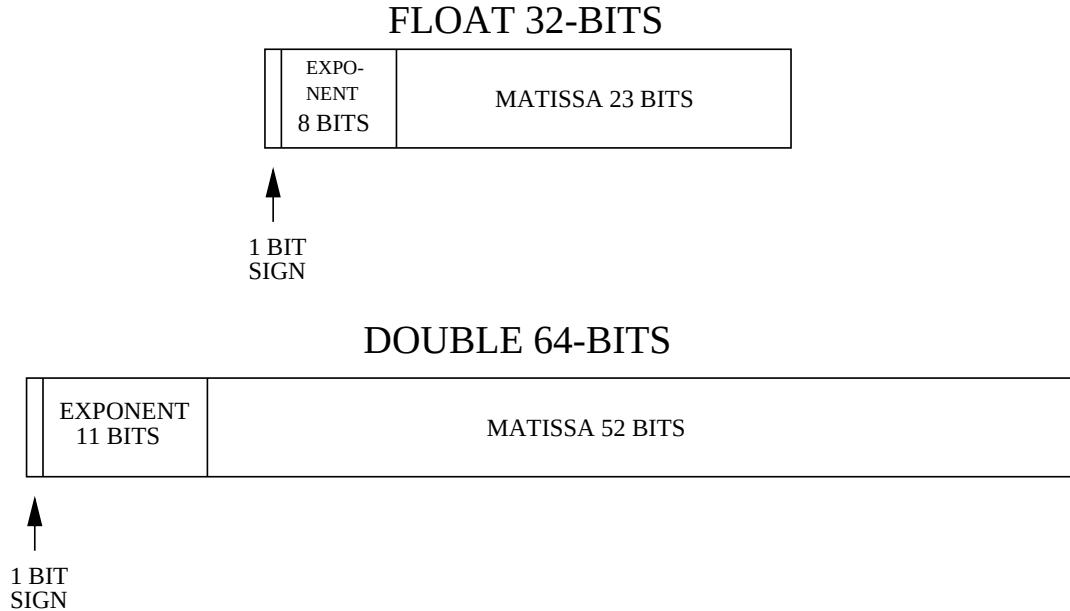
We have seen that `int`'s are represented by 32 bit “words” and that they can only represent a finite range of integral numbers. A signed integer covers the range $-2^{31} \leq i \leq (2^{31} - 1)$. Floating numbers have their own restrictions. One bit is used for the sign of the number, 23 bits represent the mantissa¹ (the precision or granularity of the number) and 8 bits are used to define the exponent. Consequently, `float`'s have a range of about 10^{-45} to about 10^{+38} and a resolution of about 1 part in 10^7 . (The particular floating-point representation can be different on different operating systems.)

`double`'s are represented by 64 bits. 52 for the mantissa, one for the sign and 11 for the exponent. The granularity is about 2 parts in 10^{16} and ranges from about 10^{-324} to about 10^{+308} . (The particular floating-point representation can be different on different operating systems.)

Some CPU manufacturers, for example Intel (which provides the CPU's for your laboratory computers and, if you own them, your home computer or laptop), have adopted an IEEE standard that represents floating point numbers with 80 bits, doing all the internal calculations at precisions higher than both `float`'s and `double`'s. This variable type is called the `long double`. 64 bits are used for the mantissa, one for the sign and 15 for the exponent. The granularity is about 1 part in 10^{19} and ranges from about 10^{-4965} to about 10^{+4932} . Other manufacturers have larger representations. For example, the Sun machines maintained by CAEN have 128-bit `long double`'s.

¹Actually, it is assumed that the leading bit is always 1, so the representation uses 23 physical bits and one “hidden” or “virtual” bit. All the floating-point numbers have a hidden bit.

This is a representation of the difference between `float` and `double`:



Consider the following code called `precisionFloat.cpp`:

```
//File: precisionFloat.cpp
#include <iostream>

using namespace std;

float precision(void)
{ float one = 1.0f, e = 1.0f, onePlus;
  int counter = 0;
  do
  { counter = counter + 1;
    e = e/2.0f;
    onePlus = one + e;
  }while(onePlus != one);
  cout << "Converged after " << counter << " iterations\n";
  return e;
}

int main(void)
{ cout << "Float resolution = " << precision() << "\n";
  return 0;
}
```

New stuff:

1. The statement: `float one = 1.0f` contains the qualifier "f". This means that 1.0 is a float, typically a 32 bit constant. If the "f" were omitted, the constant would be a double, typically a 64 bit constant.

Compiling and running `precisionFloat.cpp` results in the following output:

```
Converged after 24 iterations
Float resolution = 5.96046e-08
```

Consider the following code called `precisionDouble.cpp`:

```
//File: precisionDouble.cpp
#include <iostream>

using namespace std;

double precision(void)
{ double one = 1.0, e = 1.0, onePlus;
  int counter = 0;
  do
  { counter = counter + 1;
    e = e/2.0;
    onePlus = one + e;
  }while(onePlus != one);
  cout << "Converged after " << counter << " iterations\n";
  return e;
}

int main(void)
{ cout << "Double resolution = " << precision() << "\n";
  return 0;
}
```

New stuff:

1. The statement

```
double one = 1.0, e = 1.0;
```

declares and initializes two double's.

Compiling and running `precisionDouble.cpp` results in the following output:

```
Converged after 53 iterations
Double resolution = 1.11022e-16
```

Consider the following code called `precisionLongDouble.cpp`:

```
//File: precisionLongDouble.cpp
#include <iostream>

using namespace std;

long double precision(void)
{ long double one = 1.0L, e = 1.0L, onePlus;
  int counter = 0;
  do
  { counter = counter + 1;
    e = e/2.0L;
    onePlus = one + e;
  }while(onePlus != one);
  cout << "Converged after " << counter << " iterations\n";
  return e;
}

int main(void)
{ cout << "Long double resolution = " << precision() << "\n";
  return 0;
}
```

New stuff:

1. The statement

```
long double one = 1.0L, e = 1.0L;
```

declares and initializes two `long double`'s. The qualifiers "L" specify that the constant 1.0 is to be considered a `long double` constant, an 80-bit number on my computer.

Compiling and running `precisionLongDouble.cpp` results in the following output:

```
Converged after 64 iterations
Long double resolution = 5.42101e-20
```

Note that what is called `Float resolution`, `Double resolution`, or `Long double resolution` in the above example outputs from the programs is the number that can NOT be resolved with respect to 1.

To test the range of a `float` consider the following code called `rangeFloat.cpp`:

```
//File: rangeFloat.cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main(void)
{ float x = 1.0f, x0;
  float y = 1.0f, y0;
  int counter = 0;
  do
  { counter = counter + 1;
    x0 = x; x = x*2.0f; // Multiply by 2
    y0 = y; y = y/2.0f; // Divide by 2
    cout << setw(4) << counter << ": "
          << setw(12) << x << " :: "
          << setw(12) << y << "\n";
  }while(x0 != x || y0 != y);
  return 0;
}
```

Compiling and running `rangeFloat.cpp` results in the following output:

```
1:          2 ::          0.5
2:          4 ::          0.25
3:          8 ::         0.125
4:         16 ::         0.0625
5:         32 ::         0.03125
.
.
.
125: 4.25353e+37 :: 2.35099e-38
126: 8.50706e+37 :: 1.17549e-38
127: 1.70141e+38 :: 5.87747e-39
128:          Inf :: 2.93874e-39
129:          Inf :: 1.46937e-39
130:          Inf :: 7.34684e-40
```

```

.
.
.
147:      Inf :: 5.60519e-45
148:      Inf :: 2.8026e-45
149:      Inf :: 1.4013e-45
150:      Inf :: 0

```

New stuff:

1. The output `Inf` is the IEEE's way of representing a number that is too large to be represented. `Inf` has its own distinctive bit pattern.
2. Note that the `float`'s appear to have more "range" for small numbers than large numbers. IEEE allows for a gradual decrease to zero. These numbers below about `6e-39` are called "sub-normal", with fewer bits of precision. The leading bits of the mantissa are padded with zeros for sub-normal numbers.

To test the range of a `double` consider the following code called `rangeDouble.cpp`:

```

//File: rangeDouble.cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main(void)
{ double x = 1.0, x0;
  double y = 1.0, y0;
  int counter = 0;
  do
  { counter = counter + 1;
    x0 = x; x = x*2.0; // Multiply by 2
    y0 = y; y = y/2.0; // Divide by 2
    cout << setw(4) << counter << ": "
          << setw(12) << x << " :: "
          << setw(12) << y << "\n";
  }while(x0 != x || y0 != y);
  return 0;
}

```

Compiling and running `rangeDouble.cpp` results in the following output:


```

1:          2 ::          0.5
2:          4 ::          0.25
3:          8 ::          0.125
4:         16 ::          0.0625
5:         32 ::          0.03125
.
.
.
1021: 2.24712e+307 :: 4.45015e-308
1022: 4.49423e+307 :: 2.22507e-308
1023: 8.98847e+307 :: 1.11254e-308
1024:          Inf :: 5.56268e-309
1025:          Inf :: 2.78134e-309
.
.
.
1073:          Inf :: 9.88131e-324
1074:          Inf :: 4.94066e-324
1075:          Inf ::          0

```

To test the range of a long double (at least an IEEE 80-bit one as implemented on my machine) consider the following code called `rangeLongDouble.cpp`:

```

//File: rangeLongDouble.cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main(void)
{ long double x = 1.0L, x0;
  long double y = 1.0L, y0;
  int counter = 0;
  do
  { counter = counter + 1;
    x0 = x; x = x*2.0L; // Multiply by 2
    y0 = y; y = y/2.0L; // Divide by 2
    cout << setw(4) << counter << ": "
          << setw(12) << x << " :: "
          << setw(12) << y << "\n";
  }while(x0 != x || y0 != y);
  return 0;
}

```

```
}

```

Compiling and running `rangeLongDouble.cpp` results in the following output:

```

1:          2 ::          0.5
2:          4 ::          0.25
3:          8 ::          0.125
4:         16 ::          0.0625
5:         32 ::          0.03125
.
.
.
16382: 2.97433e+4931 :: 3.3621e-4932
16383: 5.94866e+4931 :: 1.68105e-4932
16384:          Inf :: 8.40526e-4933
16385:          Inf :: 4.20263e-4933
.
.
.
16443:          Inf :: 1.45808e-4950
16444:          Inf :: 7.2904e-4951
16445:          Inf :: 3.6452e-4951
16446:          Inf ::          0

```

Now that's a lot of numbers. Try these example codes on your own machine. If you get output that is different, particularly for the `long double`'s, I'd like to know about it!

7.1.1 When is one not one? Floating point anomalies.

According to mathematics, a very strange way of writing one would be

$$\sum_1^N \frac{1}{N}$$

Is this true for computers? No!

The following code demonstrates the effect.

```
//File: oneNotOne.cpp
#include <iostream>
#include <iomanip>

```

```
using namespace std;

int main(void)
{  cout << "N? ";
   double N;
   cin >> N;
   float fraction = 1.0/N;
   cout << "Will compute " << fraction << " * " << N << " = ";

   float sum = 0;
   for (int i = 1; i <= N; i = i + 1) sum = sum + fraction;
   cout << setprecision(19) << sum << "\n";

   return 0;
}
```

Try running this code for small values of N and big values of N , say $N = 10^7$ or 10^8 . You will get answers near 1 for small N , but something entirely different for big N ! Why? How could you fix this?

Recall that the representation of floating-point numbers is inexact in computers. If you are using `float`'s with a precision of about 1 part in 10^7 , you will start to notice anomalies if you try to add two things together that differ in size by 6 or 7 orders of magnitude. These anomalies can be readily observed with `float`'s. Converting to `double`'s or `long double`'s treats the symptom but does not cure the disease! It is always there but often not noticeable with higher precision numbers. So why not use `double`'s or `long double`'s all the time? Higher precision costs in terms of storage. Sometimes memory requirements dictate the use of lower precision. There are speed/performance issues to consider as well. This is yet another one of the compromises one has to make in designing computer programs.

7.2 char: the character variable

- A single character can be defined as the character type: `char`
- `char` is a data type used to encode single characters
- The following line declares a character called `ch`

```
char ch;
```

- The following assigns the character 'a' to the `char` variable. A literal character is a character between single ' quotes.

```
ch = 'a';
```

- The following line declares a character called `ch` and assigns it the character `'a'`

```
char ch = 'a';
```

- The following statement contains an error. What is the error?

```
ch = "a";
```

- Some special characters:

- `'\n'` is a “newline”
- `'\t'` is a “horizontal tab”
- `'\0'` is a “null”. It means nothing! We have to make a special character that means nothing!
- There are lots of others.

- The inner secret of `char`'s? They are really integers!
But, they are really integers but only one byte (8-bits) long.
Thus there can be at most $2^8 = 256$ of them.
- The following are all valid statements:

```
char c;
int i;
i = 65;
c = i;
c = 99;
```

- Consider the program :

```
//File: chars.cpp
#include <iostream>

using namespace std;

int main(void)
{
    for(int i = 0; i < 256 ; i = i + 1)
    {
        char c = i;
        if (i <= 127)
```

```

        cout << "Int " << i << "-->" << c << " in ASCII\n";
    else
        cout << "Int " << i << "-->" << c << " beyond ASCII\n";
    }
    return 0;
}

```

Try running this program and see what it gives you!

- The characters corresponding to integers 0—127 are the ASCII standard character set. (ASCII = American Standard Code for Information Interchange) which is universally adopted in English-speaking nations. The printing characters are represented by ASCII codes 32–126. Some of the ASCII codes do special things or represent “special” characters. For example, 7 is the “alert” or “bell”, 8 is the “backspace”, 9 is the “horizontal tab”, 10 is a “new line”, 11 is a “vertical tab”, 13 is a “carriage return”, 127 is the “delete”. Note the effect of some of these when running `chars.cpp`.
- The extended ASCII character set corresponding to integers 0—255 is internationally adopted and permits many of the characters from some foreign languages to be represented.
- Movement is afoot to accept the UNICODE standard which codes characters in 2 bytes (16 bits) with $2^{16} = 65536$ possibilities. This is enough to code not only every language that exists, but even every language that has ever existed. Even Egyptian hieroglyphics has a UNICODE representation. There is probably no need to extend beyond 16-bit representation until we start making contact with extra-terrestrial life forms, well, at least ones that want to communicate with us for whatever reason.

7.2.1 Character strings; the string class

- The string class requires the use of `#include <string>`
- You have seen them already!
`cout << "This is a string!\n";`
- Declaring a string called `str` and assigning it a value:
`string str = "This is a string!\n";`
- A character string is really kind of a vector in disguise.
`str[0]` is "T"
`str[1]` is "h"
etc.

- Strings can be read in with `cin` and printed with `cout`. The `cin` operation continues on the string until it sees a space or a new-line character (hitting the Return key). Here is an example that will be discussed in class:

```
//File: string1.cpp
#include <iostream>
#include <string>

using namespace std;

int main(void)
{  string response;
   do
   {  cin >> response;
      cout << response + "\n";
   }while (',' != response[0]); //Note vector notation
   return 0;
}
```

It is important to understand that the space character is considered by the `cin` function to signal the end of a string.

- The fact that a space character is considered by the `cin` operation to signal the end of a string can sometimes be a nuisance if you want to input spaces. For example, you might want an entire sentence to be considered a single string. The `getline()` string class member function (as declared by `<string>`) can be used for this purpose. Here is an example that will be discussed in class:

```
//File: string2.cpp
#include <iostream>
#include <string>

using namespace std;

int main(void)
{  string sentence;
   do
   {  getline(cin,sentence);
      cout << sentence + "\n";
   }while (',' != sentence[0]);
   return 0;
}
```

- The `length()` member function returns the length of a string, the number of characters that it contains. The `substr()` member function extracts a sub-strings from a string. Note that strings can be concatenated with the addition operator, although at least one of the operands must be a string variable. Here is an example that demonstrates these three new things.

```
//File: string3.cpp
#include <iostream>
#include <string>

using namespace std;

int main(void)
{   cout << "Type in a string: ";
    string r1;
    cin >> r1;
    cout << "Your response was " << r1.length() << " chars long\n";

    cout << "Type in another string: ";
    string r2;
    cin >> r2;
    cout << "Your response was " << r2.length() << " chars long\n";

    cout << "Putting them together: " << r1 + " " + r2 << "\n";

    cout << "Slicing and splicing them up: "
          << r1.substr(0,2) + r2.substr(4,7) << "\n";

    return 0;
}
```

7.3 The vector class

7.3.1 Introduction to vectors

So far we have been dealing with individual `float`'s and `int`'s, declaring them individually, naming them individually, manipulating them one at a time, individually. Individuality is nice, but sometimes it is more efficient to refer to a group of similar individuals in a collective manner. In many applications we wish to use data, many of the same type, in some sort of systematic or regular way.

For example, a function of x is called $f(x)$. An operation on $f(x)$, such a multiplication by a constant factor operates on all the elements of $f(x)$, effectively an infinite number of operations! A more tangible example is; say a kind-hearted professor wanted to raise all the grades in the class by 5%. If he had a collection of the grades in an ordered list, it would be a lot easier (and more likely to happen) if he could multiply the whole list at once by 1.05, rather than each grade on an individual basis.

To carry this example a little further, our class list is made up of 217 individuals (as in the case of ENG101/200). Each of these individuals is assigned a different grade at the end of the course. Let's say we wanted to calculate the class average. A computation of a class average would involve accessing all this data in some regular fashion using a computer program.

C++ provides a way of doing this by allowing you to define a *vector*—a symbol that is associated with many objects.

7.3.2 Declaring and using vectors

Suppose I wanted to declare variables associated with the individual grades in this class. I could do the following:

```
int gradeForStudent1 = 98;
int gradeForStudent2 = 79;
int gradeForStudent3 = 88;
.
.
.
int gradeForStudent217 = 91;
```

This is a really dumb, repetitive way to do it! Anything that seems really repetitive for a human can probably be done in a better way by a machine. C++ allows you to define everything at once using the following syntax:

```
vector<int> gradeForStudent(217);
```

which declares, in one fell swoop, 217 `int`'s. (We could also have made them `float`'s or `double`'s. We can make vectors out of any variable type.) So, we can refer to the collection of grades by a single identifier, `gradeForStudent`, in a collective fashion.

But we still have work to do. We still have to enter the numbers somehow into the computer program. One way is through assignment statements:

```
gradeForStudent[0] = 98; /* 1st student */
```



```

gradeForStudent[1] = 79; /* 2nd student */
gradeForStudent[2] = 88; /* 3rd student */
.
.
.
gradeForStudent[216] = 91; /* 217th student */

```

which also seems a little dumb, but indicates that we can also refer to the individuals by a numerical *index*. (For example, 7of9 in the Borg collective. If you are not a Trekkie, please ignore the last statement. However, for some strange reason, 7of9 is the only individual Borg that I can name.)

In fact, for data of this nature, there is always some boring work to be done—typing in all the grades. Unless there is an automatic way to generate the components of the vector, some person has to do the boring job of typing it all in. This is called *data entry*.

The above example, however, shows that vector indices (the integer inside the []'s) start with zero. This leads to “off-by-one” errors that plague all novice (and some experienced) programmers in C++. So, just remember that the *n*'th element in a vector is referred to by the index *n* - 1.

Another way of entering the data to the vector is by using an input loop. For example:

```

#include <iostream>
#include <vector>
.
.
.
const int NumberOfStudents = 217;
vector<int> gradeForStudent(NumberOfStudents);
for(i = 0; i < gradeForStudent.size(); i = i + 1)
{ cout << "Input grade for student number << i << ": ";
  cin >> gradeForStudent[i];
}

```

gets the input done in another way, although someone still has to do all the typing.

There is some new stuff here that needs a little explanation. The vectors that we have just introduced are really a *class* in C++. A class is defined as data and functions that operate on the data within the class. The function `size()` is one of the vector class *member functions*. The syntax to have a member function operate on a data variable in its class is: name the data variable and connect it via the period, “.”, and then name the member function with the appropriate argument list, in this case nothing. The `size()` member function returns the number of elements in the vector, in this case, 217.

There is one other new thing in the above code snippet, the introduction of the `const` qualifier to a variable. All this means is that the variable is to be held constant after its initialization. Any attempt to change it would result in an error. This is a safe way to keep vector sizes safe from programmer error if they are known to be fixed.

The best way to input the numbers is to input them in a separate file and read this file in during the execution of the code. This would make the code more general and the code itself would not have to be concerned with having to enter the data faithfully. And, another program could make use of this data. We will learn how to read and write files later on in the course.

7.3.3 Vector syntax and rules:

- You need to `#include <vector>` to use vectors.
- The name of a vector (*e.g.* `gradeForStudent`) follows the naming convention for identifiers.
- Vectors can be “global” or “local”, just like individual variables, depending on where they are declared.
- The “index” also called the “subscript” is the thing that goes inside the `[ ]`’s.
- What goes inside the `[ ]`’s is an integer or an integer expression.
- If you violate the bounds of a vector (write data either below it or above it), one of two things can happen
 - Either your program will crash, or
 - The data occupying memory adjacent to the vector will be corrupted, the program will carry on and operate on or with corrupted data.

At least in the first situation you know something is wrong. In the second you don’t and that could be dangerous.

- The *i*’th element of vector `someVector` is `someVector[i - 1]`
- `someVector[someIntegerExpression]` denotes a *value*. It can go on the left of an assignment operator “=”
- When a vector is declared all of its elements are set to zero.
- Once declared, it is best to think of vectors as regular variables. For example,

```
vector<float> vector2 = vector1;
```

where `vector1` was previously defined, creates a new vector called `vector2` with the same size as `vector1` and copies all the elements of `vector1` to `vector2`.

Example: Computing averages

Suppose we have a class of 10 students. I want to compute the average grade of a class test. Here is a way to do it.

```
//File: sumGradesGood.cpp
#include <iostream>
#include <vector> //Need this to use the vector "class"

using namespace std;

int main(void)
{   const int MaxNoStudents = 10;
    vector<int> grade(MaxNoStudents);

    int sum = 0;
    for (int i = 0; i < grade.size(); i = i + 1)
    {   cout << "Grade for student #" << i + 1 << ": "; // Note the "+ 1"
        cin >> grade[i];
        sum = sum + grade[i];
    }

    cout << "\nGrade report:\n\n";

    for (int i = 0; i < grade.size(); i = i + 1)
        cout << "Grade for student #" << i + 1 << ": \t" << grade[i] << "\n";

    cout << "Class avg is : " << static_cast<float>(sum)/grade.size() << "\n";
    return 0;
}
```

And here is a sample use of the code:

```
Grade for student #1: 1
Grade for student #2: 2
Grade for student #3: 3
Grade for student #4: 4
Grade for student #5: 5
```

```
Grade for student #6: 6
Grade for student #7: 7
Grade for student #8: 8
Grade for student #9: 9
Grade for student #10: 10
```

Grade report:

```
Grade for student #1: 1
Grade for student #2: 2
Grade for student #3: 3
Grade for student #4: 4
Grade for student #5: 5
Grade for student #6: 6
Grade for student #7: 7
Grade for student #8: 8
Grade for student #9: 9
Grade for student #10: 10
Class avg is : 5.5
```

Example: A more refined way of computing averages

The above example requires that the C++ code know in advance how many students there are in the class. So, if the number of students change, the code has to be recompiled. Why not ask the user of the program to input the number of students, then declare the vector and then do the calculation? Why not? Here is a better way to do it.

```
//File: sumGradesBetter.cpp
#include <iostream>
#include <vector> //Need this to use the vector "class"

using namespace std;

int main(void)
{ cout << "Number of students writing the test: ";
  int nStudents;
  cin >> nStudents;
  vector<int> grade(nStudents);

  int sum = 0;
  for (int i = 0; i < grade.size(); i = i + 1)
```

```

    { cout << "Grade for student #" << i + 1 << ": "; // Note the "+ 1"
      cin >> grade[i];
      sum = sum + grade[i];
    }

    cout << "\nGrade report:\n\n";

    for (int i = 0; i < grade.size(); i = i + 1)
        cout << "Grade for student #" << i + 1 << ": \t" << grade[i] << "\n";

    cout << "Class avg is : " << static_cast<float>(sum)/grade.size() << "\n";
    return 0;
}

```

A example of its use is:

```

Number of students writing the test: 3
Grade for student #1: 6
Grade for student #2: 8
Grade for student #3: 9

```

Grade report:

```

Grade for student #1:    6
Grade for student #2:    8
Grade for student #3:    9
Class avg is : 7.66667

```

Example: An even better way

I'm still dissatisfied with the above program. It requires me to count how many students wrote the test. Why not have the vector simply grow in size as I enter more data? Why not? Here is an even better way to do it. In this case, the input loop terminates with a negative grade or a grade larger than 100.

```

//File: sumGradesBest.cpp
#include <iostream>
#include <vector> //Need this to use the vector "class"

using namespace std;

```

```
int main(void)
{  vector<int> grade;

    double sum = 0, score;
    do
    {  cout << "Input a student's grade: ";
        cin >> score;
        if (0. <= score && 100. >= score)
        {  sum = sum + score;
            grade.push_back(score);
        }
    }while(0. <= score && 100. >= score);

    cout << "\nGrade report:\n\n";

    for (int i = 0; i < grade.size(); i = i + 1)
        cout << "Grade for student #" << i + 1 << ": \t" << grade[i]
            << "\n";

    cout << "Class avg is : " << sum/grade.size() << "\n";
    return 0;
}
```

An example of its use is:

```
Input a student's grade: 79
Input a student's grade: 60
Input a student's grade: 99
Input a student's grade: 57
Input a student's grade: -1
```

Grade report:

```
Grade for student #1: 79
Grade for student #2: 60
Grade for student #3: 99
Grade for student #4: 57
Class avg is : 73.75
```

The new features of the above program is a new member function of the vector class, seen in the statements:

```

.
.
.
vector<int> grade;
.
.
.
grade.push_back(score);
.
.
.

```

Note that the first statement above simply declares a vector without a size! We are free to do this. All it does is bind the identifier **grade** to a vector variable type. The member function **push_back(score)** increases the size of the vector by one and initializes the new location with its argument, in this case, the value of **score**. There is another member function called **pop_back()**, with no argument, that reduces a vector's size by one.

7.3.4 Vectors and functions

Remember that it is usually useful to think of vectors as simple variables. This is especially true for vectors and functions. Take all the rules you have learned about functions and variables and apply them to functions and vectors. To demonstrate this, consider the following example that demonstrates numerical integration. All the algorithm does is integrate a simple function in three different ways. First, it puts the function into a vector with a certain number of points based on an evenly divided mesh, or grid. Then it forms rectangles in 3 different ways, using for its height the left mesh point, the right mesh point and the midpoint. Can you guess why the midpoint method is so much better?

```

//File: integrate.cpp
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

double integrateLeft(vector<double> f, vector<double> x)
{ double sum = 0;
  for (int i = 0; i < f.size() - 1; i = i + 1)
    sum = sum + f[i]*(x[i + 1] - x[i]);
  return sum;
}

```

```

}

double integrateRight(vector<double> f, vector<double> x)
{ double sum = 0;
  for (int i = 0; i < f.size() - 1; i = i + 1)
    sum = sum + f[i + 1]*(x[i + 1] - x[i]);
  return sum;
}

double integrateMiddle(vector<double> f, vector<double> x)
{ double sum = 0;
  for (int i = 0; i < f.size() - 1; i = i + 1)
    sum = sum + 0.5*(f[i] + f[i + 1])*(x[i + 1] - x[i]);
  return sum;
}

int main(void)
{ cout << "Number of points in the integration mesh: ";
  int mesh;
  cin >> mesh;
  vector<double> f(mesh), x(mesh);
  for(int i = 0; i < mesh; i = i + 1)
  { x[i] = static_cast<double>(i)/(mesh - 1);
    f[i] = exp(x[i]);
  }

  double integral = integrateLeft(f, x);
  cout <<
    "left = \t\t" << integral << " cf " << exp(1.0) - 1.0
    << " diff = " << integral - (exp(1.0) - 1.0) << "\n";

  integral = integrateRight(f, x);
  cout <<
    "right = \t" << integral << " cf " << exp(1.0) - 1.0
    << " diff = " << integral - (exp(1.0) - 1.0) << "\n";

  integral = integrateMiddle(f, x);
  cout <<
    "midpoint = \t" << integral << " cf " << exp(1.0) - 1.0
    << " diff = " << integral - (exp(1.0) - 1.0) << "\n";

  return 0;
}

```



```
}
```

7.4 Problems

1. The inner product

Complete the program started below. You must complete the function `innerProduct()` which computes the inner product of two vectors, x and y . The inner product is the sum $x_0y_0 + x_1y_1 + \dots + x_{n-1}y_{n-1}$ where x_i is the $i + 1$ 'th element of x and the number of elements in the vector is n . Here is a function stub and a main routine that you must use to test your function:

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

float innerProduct(const vector<float>& a, const vector<float>& b)
{ //Why did the function use the const qualifiers in the parameter list
  //and pass the vectors by reference?
  //Answer the question and insert your code after this line.
}

int main(void)
{ int vectorLength = 10;

  vector<float> x (vectorLength);
  cout << " x: ";
  for (int i = 0; i < vectorLength; i = i + 1)
  { x[i] = rand()%100;
    cout << "\t" << x[i];
  }
  cout << "\n";

  vector<float> y(vectorLength);
  cout << " y: ";
  for (int i = 0; i < vectorLength; i = i + 1)
  { y[i] = rand()%10;
    cout << "\t" << y[i];
  }
}
```

```

    cout << "\n";

    cout << "Inner product is: " << innerProduct(x,y) << "\n";

    return 0;
}

```

In the comments of your function, answer the question: Why did the function use the `const` qualifiers in the parameter list and pass the vectors by reference?

2. Maximum and minimum

Complete the program started below. You must complete the function `minMax()` which returns both the minimum and the maximum elements of the vector. Here is a function stub and a main routine that you must use to test your function:

```

#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

void minMax(const vector<int>& c, int& min, int& max)
{ //Why were min and max given as "reference parameters"?
  //Answer the question and insert your code after this line.
}

int main(void)
{ int vectorLength = 8;

  vector<int> a(vectorLength);
  cout << " a: ";
  for (int i = 0; i < vectorLength; i = i + 1)
  { a[i] = rand();
    cout << "\t" << a[i];
  }
  cout << "\n";

  int minimum, maximum;
  minMax(a, minimum, maximum);

  cout << "The minimum array element is: " << minimum << "\n";
  cout << "The maximum array element is: " << maximum << "\n";
}

```

```
    return 0;
}
```

In the comments of your function, answer the question: “Why were min and max given as ‘reference parameters’?”

3. Nothing negative

Complete the program started below. You must complete the function `nothingNegative()` which accepts a vector called `bothSigns` containing positive and negative integers and returns a vector of equal or lesser length that contains only the non-negative integers of the vector `bothSigns`. Here is a function stub and a main routine that you must use to test your function:

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

vector<int> nothingNegative(const vector<int>& a)
{ //Why can this program, when compiled with just the function stub,
  //go into what looks like an infinite loop?
  //Answer the question and insert your code after this line.
}

int main(void)
{ int vectorLength = 10;

  vector<int> bothSigns(vectorLength);
  cout << " Input vector: ";
  for (int i = 0; i < vectorLength; i = i + 1)
  { bothSigns[i] = rand()%201 - 100;
    cout << "\t" << bothSigns[i];
  }
  cout << "\n";

  vector<int> noNegs = nothingNegative(bothSigns);

  cout << " Output vector: ";
  for (int i = 0; i < noNegs.size(); i = i + 1)
  { cout << "\t" << noNegs[i];
```

```

    }
    cout << "\n";

    return 0;
}

```

In the comments of your function, answer the question: “Why can this program, when compiled with just the function stub, go into what looks like an infinite loop?”

4. Vector manipulations

Write a function that accepts one called-by-reference vector of int’s and returns another vector of int’s. Upon entry to the function, the vector in the parameter list can be any size and contains int’s that can have any value. Upon return, the vector in the parameter list contains only the positive int’s (> 0) in the same order. Upon return, the returned vector contains only the negative int’s (< 0) in the same order as they appeared in the original called-by-reference vector in the parameter list, when the function was entered.

Here is an example of how it works:

Vector in the parameter list supplied to the function:

```
0 -6 -8 3 4 -8 0 8 -9 -8 8
```

That vector after the function returns:

```
3 4 8 8
```

Returned vector after the function returns:

```
-6 -8 -8 -9 -8
```

5. Push and pop

This is a more challenging version of the previous question. Complete the program started below. Write a `void` function that accepts two called-by-reference vectors of int’s. Upon entry to the function, the first vector can be any size and contains int’s that can have any value. The second vector is empty, its size is 0. Upon return, the first vector contains only the positive int’s (> 0) in the same order. Upon return, the second vector contains only the negative int’s (< 0) in the same order as they appeared in the original first vector, when the function was entered. Important: Do this without declaring any new vectors.

```

#include <iostream>
#include <vector>
#include <cstdlib>

```

```

#include <ctime>

using namespace std;

//Function definition goes here...

int main(void)
{
    srand(time(NULL));
    int size = rand()%20 + 10; //size is a RN between 10 and 29
    vector<int> in(size), out;

    for (int i = 0; i < in.size(); i = i + 1)
        in[i] = rand()%19 - 9; //RN between -9 and +9

    cout << "Original vector is size " << in.size() << endl;
    for (int i = 0; i < in.size(); i = i + 1)
        cout << in[i] << " ";
    cout << "\n\n";

    extractNegs(in, out);

    cout << "Returned vector is size " << out.size() << endl;
    for (int i = 0; i < out.size(); i = i + 1)
        cout << out[i] << " ";
    cout << "\n\n";

    cout << "Modified vector is size " << in.size() << endl;
    for (int i = 0; i < in.size(); i = i + 1)
        cout << in[i] << " ";
    cout << "\n\n";

    return 0;
}

```

6. More vector manipulations

Write a function that returns a `vector<int>` and accepts one called-by-reference `vector<int>`. Upon entry to the function, the vector in the parameter list can be any size and contains int's that are positive, negative and 0. Upon return, the vector that is returned contains the *indices* of the original vector whose original values were negative. Upon return, the original vector in the parameter list is modified so that it only contains its original positive (> 0) int's in the same order that they appeared in the original vector. Write your function below or on the next page. It should be

compatible with the main routine below..

```
//Precompiler code omitted for brevity

//Function goes here

int main(void)
{  srand(time(NULL));
   int size = rand()%11 + 10; //size is random between 10 and 20
   vector<int> in(size);
   for (int i = 0; i < in.size(); i = i + 1)
       in[i] = rand()%19 - 9; //vector element is random between -9 and 9

   cout << "Original vector is size " << in.size() << endl;
   cout << "Original vector: ";
   for (int i = 0; i < in.size(); i = i + 1) cout << in[i] << " ";
   cout << endl;

   vector<int> negs = removeNegs(in);

   cout << "Modified vector is size " << in.size() << endl;
   cout << "Modified vector: ";
   for (int i = 0; i < in.size(); i = i + 1) cout << in[i] << " ";
   cout << endl;

   cout << "Returned vector is size " << negs.size() << endl;
   cout << "Returned vector: ";
   for (int i = 0; i < negs.size(); i = i + 1) cout << negs[i] << " ";
   cout << endl;

   return 0;
}
```

Here is an example of how it works:

```
red% a.out
Original vector is size 13
Modified vector: -9 0 4 9 9 9 0 3 3 -1 -6 5 0
Modified vector is size 7
Modified vector: 4 9 9 9 3 3 5
Returned vector is size 3
Returned vector: 0 9 10
```

7. Perfect squares

Complete the program started below. Write a function that returns a `vector<int>` and accepts one constant called-by-reference `vector<int>`. Upon entry to the function, the vector in the parameter list can be any size and contains int's that are positive (> 0). Upon return, the vector that is returned contains the *indices* of the first vector whose values are perfect squares.

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cmath>
#include <ctime>

using namespace std;

//Function definition goes here...

int main(void)
{  srand(time(NULL));
   int size = rand()%20 + 10; //size is a RN between 10 and 29
   vector<int> in(size);
   for (int i = 0; i < in.size(); i = i + 1)
   {  in[i] = rand()%90 + 10;
      cout << in[i] << " ";
   }
   cout << endl;

   vector<int> out = squareIndex(in);

   cout << "Returned vector is size " << out.size() << endl;

   for (int i = 0; i < out.size(); i = i + 1) cout << out[i] << " ";
   cout << endl;

   return 0;
}
```

8. Sort this out

Complete the program started below. You must complete the function `sort()` that has the `void` return type. This function accepts a vector called `unsorted` containing random int's. Here is a function stub and a main routine that you must use to test your function:

```

#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

void sort(vector<int>& a){}

int main(void)
{   cout << "Input the minimum and maximum random int: ";
    int minRange, maxRange;
    cin >> minRange >> maxRange;

    cout << "Input the length of the vector: ";
    int vectorLength;
    cin >> vectorLength;

    vector<int> unsorted(vectorLength);
    cout << " Vector beforehand: \t";
    for (int i = 0; i < vectorLength; i = i + 1)
    {   unsorted[i] = minRange + rand()%(maxRange - minRange + 1);
        cout << unsorted[i] << " ";
    }
    cout << "\n";

    sort(unsorted);

    cout << " Vector afterward: \t";
    for (int i = 0; i < unsorted.size(); i = i + 1)
    {   cout << unsorted[i] << " ";
    }
    cout << "\n";
    return 0;
}

```

9. A prime example

Complete the program started below. You must complete the function `getPrimes()` that has the `vector<int>` return type. This function accepts an `int` called `NPrimes` that is the number of consecutive prime numbers starting with 2. The return vector contains these prime numbers in order.

You must make your function efficient by using something similar to the following test to eliminate numbers that are not prime:


```
if (0 == n%divisor) nIsPrime = false;
```

where `N` is the candidate prime and `divisor` is a prime number. The `divisor` must not be non-prime.

```
#include <iostream>
#include <vector>
#include <cstdlib>

vector<int> getPrimes(const int NPrimes){}

int main(void)
{  cout << "Input the number of primes to find: ";
    int NPrimes;
    cin >> NPrimes;

    vector<int> primes = getPrimes(NPrimes); //Get the primes
    cout << "The first " <<  primes.size() << " prime numbers: ";

    for (int i = 0; i < primes.size(); i = i + 1)
    {  cout << primes[i] << " ";
        }
    cout << "\n";
    return 0;
}
```

10. No repeats

Complete the program started below. You must complete the function `noRepeats()` that has the `void` return type. This function accepts a vector called `bothSigns` containing random `int`'s and eliminates any values in it that are repeated. Here is a function stub and a main routine that you must use to test your function:

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

void noRepeats(vector<int>& a){}

int main(void)
```

```
{ cout << "Input the minimum and maximum random int: ";
  int minRange, maxRange;
  cin >> minRange >> maxRange;

  cout << "Input the length of the vector: ";
  int vectorLength;
  cin >> vectorLength;

  vector<int> hasRepeats(vectorLength);
  cout << " Vector beforehand: \t";
  for (int i = 0; i < vectorLength; i = i + 1)
  { hasRepeats[i] = minRange + rand()%(maxRange - minRange + 1);
    cout << hasRepeats[i] << " ";
  }
  cout << "\n";

  noRepeats(hasRepeats);

  cout << " Vector afterward: \t";
  for (int i = 0; i < hasRepeats.size(); i = i + 1)
  { cout << hasRepeats[i] << " ";
  }
  cout << "\n";
  return 0;
}
```

Chapter 8

More Data Abstraction, Arrays, Structures

8.1 Arrays

An array is an extension of the vector concept. An array can represent a list of objects with a single index, for example, A_i , for $i = 1, 2, 3 \dots N$. For numerical computation, so called one-dimensional arrays are used to represent lists of objects (*e.g.* a list of grades). Two one-dimensional arrays (*e.g.* $x[i]$, $f[x]$) can represent a function $f(x)$ by specifying a grid of points along the x -axis, $x[i]$, or x_i , and the function evaluated at those grid points, $f[i]$, or $f_i = f(x_i)$. These are the most common uses of one-dimensional arrays.

Vectors, which we encountered in the previous lecture, are really one-dimensional arrays. However, arrays are a *lower level* construct that comes from C. Simply stated, arrays are ordered collections of things you already know about, for example, `int`'s, `float`'s and `double`'s with a few special rules related to indexing, initializing and communicating with functions. A vector, on the other hand, is a standard *class* in C++ and it comes with powerful *member functions* like `size()`, `push_back()` and `pop_back()`, that operate on the data in vectors. For this reason, we will not investigate one-dimensional arrays in detail in this course, choosing instead to focus on the more powerful vector class when dealing with 1-D applications.

The multi-dimensional array is an extension of the one-dimensional array concept. A two-dimensional array can be thought of as a grid or table of values, a set of rows and columns, each with its own value. There are many expressions of these. A two-dimensional array can represent a checkerboard, a chessboard, the board in game of Battleship or Minesweeper, a football field, or the surface of the earth. (*Yes, it is two-dimensional, Christopher; it's just not flat!*) We could think of many examples, there being no limit to human creativity. After this lecture, you will have enough syntax tools and rules to program a game of chess. With a little more understanding of atmospheric physics (not programming), after this lecture

you could write a program that predicts the weather! After only about 20 lectures we have gotten this far! Now this is getting interesting! You have the tools and the rules, now apply your creative thinking to design algorithms which themselves are only composed of three things—the three pillars of algorithm design: sequence, branch and loop. It really just boils down to this. It really is that simple and wonderful.

In more abstract terms, a two-dimensional array can represent a table of objects with two indices, for example, $A_{i,j}$. For numerical computation in science and engineering, two-dimensional arrays are used to represent tables of objects as in the examples above. Two-dimensional arrays (e.g. $x[i][j]$, $f[i][j]$) can represent a function of two variables, $f(x, y)$, by specifying a grid of points in the xy -plane, $x[i][j]$, or $x_{i,j}$, and the function evaluated at those grid points, $f[i][j]$, or $f_{i,j} = f(x_{i,j})$. However, the most interesting ones for scientists and engineers are “square” two-dimensional arrays, also called matrices. In this course, we will restrict our discussion to 2-dimensional arrays. There is nothing new in three and higher dimensions that we will not see in studying 2-dimensional ones. And, the rules for one-dimensional arrays can be obtained easily from two-dimensional arrays.

Unfortunately, C++ has not defined a new class to describe these objects. So we will have to grapple with the more primitive array inherited from C. Later on in the course, we will be studying Matlab. Matlab is built on two-dimensional arrays as the the fundamental variable type and has developed a lot of functionality for it, which we will discover later. Matlab is a very sophisticated program (used by professional engineers worldwide). However, if you can develop the application in C++, it will always run faster, sometimes *much* faster. So, we spend some effort learning how to work with two-dimensional arrays in C++. Besides, doing so will be an excellent exercise in algorithmic thinking—the prime reason for teaching this course.

The ANSI standard calls for arrays of at least 12 dimensions. Believe it or not, there are actually applications for this!

OK, enough with the sermon, already. Let’s dive into the syntax before we learn how to exploit it for creative expression. Some of the syntax was already introduced above.

8.1.1 Declaring a 2-dimensional array

Here’s an example of how to declare and initialize a two-dimensional array:

```
const int NROWS = 3;
const int NCOLUMNS = 2;
int a[NROWS][NCOLUMNS];
```

This is called a “3 by 2 array” or more generally as an NROWS by NCOLUMNS array. It is useful to visualize 2-dimensional arrays with the columns numbered 0 to NCOLUMNS - 1 going from

left to right and the rows numbered 0 to `NROWS - 1` going down the page. Of course, this visualization is completely arbitrary but helps serve as a memory aid.

<code>a[0][0]</code>	<code>a[0][1]</code>
<code>a[1][0]</code>	<code>a[1][1]</code>
<code>a[2][0]</code>	<code>a[2][1]</code>

Another useful way to think about it is as a one-dimensional array of other one-dimensional arrays. Indeed, the notation `a[NROWS][NCOLUMNS]` is very suggestive of this! We can visualize this as vectors of length `NROWS` aligned vertically and columns of length `NCOLUMNS` aligned horizontally. Use whatever visual aid works for you. The above two suggestions work for me.

8.1.2 Initializing a 2-dimensional array

Initializing a 2-dimensional array in the declaration statement can be done in a variety of ways. Here are some examples:

```
const int NROWS = 3;
const int NCOLUMNS = 2;
int a[NROWS][NCOLUMNS] = {0};
```

Sets all of the array elements to zero. That is:

0	0
0	0
0	0

This feature is important to remember as it saves a lot of typing, a particular passion of mine.

The brace notation may also be used:

```
const int NROWS = 3;
const int NCOLUMNS = 2;
int a[NROWS][NCOLUMNS] = {{1,2}, {3,4}, {5,6}};
```

The result is:

1	2
3	4
5	6

The brace notation is nice because it may be used to initialize the array a little more suggestively. For example,

```
const int NROWS = 3;
const int NCOLUMNS = 2;
int a[NROWS][NCOLUMNS] =
{ {1,2},
  {3,4},
  {5,6}
};
```

with the same result.

Note the use of nested braces. If a brace is not filled completely, it is filled with zeros. For example,

```
const int NROWS = 3;
const int NCOLUMNS = 2;
int a[NROWS][NCOLUMNS] = {{1}, {3}, {5}};
```

results in

1	0
3	0
5	0

Note that it is important, when initializing arrays in this fashion, to make the array size declared with `int`'s that are fixed with the `const` qualifier. Otherwise the compiler will complain and not compile. Arrays can be declared anywhere in a program, for example:

```
const int nrows = 10;
const int ncolumns = 20;
float f[nrows][ncolumns];
```

However, the array will have to be initialized with executable statements. Also, in contrast to vectors, arrays are not automatically initialized to zero. It must be done either on the declaration statement or explicitly initialized to zero within executable statements. Otherwise, the array will contain bit patterns that were left in memory following previous use.

Here's a small program that demonstrates some of the declaration, initialization and addressing concepts.

```
//File: whereItsAt.cpp
#include <iostream>

int main(void)
{
    const int NROWS = 3;
    const int NCOLUMNS = 2;
    int a[NROWS][NCOLUMNS] = {{1}, {3}, {5}};

    cout << a[0][0] << ":" << a[0][1] << "\n";
    cout << a[1][0] << ":" << a[1][1] << "\n";
    cout << a[2][0] << ":" << a[2][1] << "\n";

    cout << "&a[2][1]: " << &a[2][1] << "\n";
    cout << "&a[2][0]: " << &a[2][0] << "\n";
    cout << "&a[1][1]: " << &a[1][1] << "\n";
    cout << "&a[1][0]: " << &a[1][0] << "\n";
    cout << "&a[0][1]: " << &a[0][1] << "\n";
    cout << "&a[0][0]: " << &a[0][0] << "\n";

    cout << "a: " << a << "\n";

    return 0;
}
```

which produces the following output:

```
1:0
3:0
5:0
&a[2][1]: 0xbffff72c
&a[2][0]: 0xbffff728
&a[1][1]: 0xbffff724
&a[1][0]: 0xbffff720
&a[0][1]: 0xbffff71c
&a[0][0]: 0xbffff718
a: 0xbffff718
```

Note how this multi-dimensional array is stored. Computer memory is organized as if it were a one-dimensional array. Thus, a multi-dimensional array has to be unfolded into a “line”

and stored. Different languages do this in different ways! C++’s convention is to store it from the outer index to the inner index, reinforcing the “vectors of vectors” concept. In our two-dimensional example this means that all the columns in the first row are stored, then all the columns of the second row and so forth.

It is an important performance issue when writing programs to be aware of this storage order and use the arrays in the most efficient manner. For example, the following program :

```
//File: fastArray.cpp
int main(void)
{  const int SIZE = 1000;
    float a[SIZE][SIZE];

    for(int k = 1 ; k <= 100 ; k = k + 1)
        for(int i = 0 ; i <= SIZE - 1 ; i = i + 1)
            for(int j = 0 ; j <= SIZE - 1 ; j = j + 1)
                a[i][j] = 1;
    return 0;
}
```

executes almost twice as fast on my computer than `slowArray.cpp` which is only different in the ordering of the two for loops:

```
//File: slowArray.cpp
int main(void)
{  const int SIZE = 1000;
    float a[SIZE][SIZE];

    for(int k = 1 ; k <= 100 ; k = k + 1)
        for(int i = 0 ; i <= SIZE - 1 ; i = i + 1)
            for(int j = 0 ; j <= SIZE - 1 ; j = j + 1)
                a[j][i] = 1;
    return 0;
}
```

8.1.3 Passing a two-dimensional array to a function

Here is where multi-dimensional arrays differ significantly from one-dimensional arrays. When a function is passed an array it has to be given enough information to know how to move around in the array. Thus, the dimension of all the outer indices (not the inner-most) must be specified. See how this is done in the following example.


```

//File: 2d.cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>

const int SIZE = 24;
const int MAX_PRINT = 24;

void printArray(float a[][SIZE])
{ if (SIZE <= MAX_PRINT)
  { for(int i = 0; i < SIZE; i = i + 1)
    { for(int j = 0; j < SIZE; j = j + 1)
      cout << setw(2) << static_cast<int>(a[i][j]) << " ";
      cout << "\n";
    }
  }
  else cout << "Array is too big to print.\n";
  return;
}

void productArray(vector<float>& y, float O[][SIZE], vector<float> x)
{ for(int i = 0; i < SIZE; i = i + 1)
  for(int j = 0; j < SIZE; j = j + 1)
    y[i] = y[i] + O[i][j] * x[j];
  return;
}

int main(void)
{ /* Initialize the D array */
  float D[SIZE][SIZE] = {0};
  for(int i = 0; i < SIZE - 1; i = i + 1)
  { D[i][i] = -1;
    D[i][i + 1] = 1;
  }

  // Initialize the I array
  float I[SIZE][SIZE] = {0};
  for(int i = 1; i < SIZE; i = i + 1)
    for(int j = 0; j <= i; j = j + 1)

```

```

        I[i][j] = 1;

// Print out D and I arrays
cout << "The \"D\" array:\n"; printArray(D);
cout << "The \"I\" array:\n"; printArray(I);

// Initialize the f and x vectors
vector<float> f(SIZE), x(SIZE);
for(int i = 0; i < SIZE; i = i + 1)
{
    x[i] = -1 + 2.0*i/(SIZE - 1);
    f[i] = 1.0/(1.01 - x[i]);
}

// Operate on f with D and normalize
vector<float> d(SIZE);
productArray(d,D,f);
for(int i = 0; i < SIZE - 1; i = i + 1)
    d[i] = d[i]/(x[i+1] - x[i]);

// Operate on f with I and normalize
vector<float> F(SIZE);
productArray(F,I,f);
for(int i = 1; i < SIZE; i = i + 1)
    F[i] = F[i]*(x[i] - x[i-1]);

// Print results
cout
    << "      x      "
    << "      f      "
    << "      d      "
    << "      F\n";
for(int i = 0 ; i <= SIZE - 1 ; i = i + 1)
    cout << setiosflags(ios::fixed)
        << setw(11) << x[i] << " "
        << setw(11) << f[i] << " "
        << setw(11) << d[i] << " "
        << setw(11) << F[i] << "\n";

return 0;
}

```

The output of this program is:

[illegible][illegible]

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
x				f				d				F										
-1.000000				0.497512				0.258711				0.000000										
-0.913043				0.520009				0.283216				0.088480										
-0.826087				0.544637				0.311375				0.135840										
-0.739130				0.571713				0.343954				0.185554										
-0.652174				0.601622				0.381930				0.237869										
-0.565217				0.634833				0.426560				0.293072										
-0.478261				0.671925				0.479500				0.351500										
-0.391304				0.713621				0.542947				0.413554										
-0.304348				0.760834				0.619879				0.479714										
-0.217391				0.814736				0.714408				0.550560										
-0.130435				0.876859				0.832347				0.626809										
-0.043478				0.949237				0.982116				0.709351										
0.043478				1.034638				1.176305				0.799320										
0.130435				1.136925				1.434410				0.898183										
0.217391				1.261657				1.787930				1.007892										
0.304348				1.417129				2.290509				1.131120										
0.391304				1.616304				3.039655				1.271669										
0.478261				1.880621				4.228182				1.435201										
0.565217				2.248289				6.283189				1.630704										
0.652174				2.794654				10.317343				1.873718										
0.739130				3.691814				20.073689				2.194744										
0.826087				5.437352				56.080322				2.667557										
0.913043				10.313904				1031.390381				3.564421										
1.000000				100.000000				0.000000				12.260068										

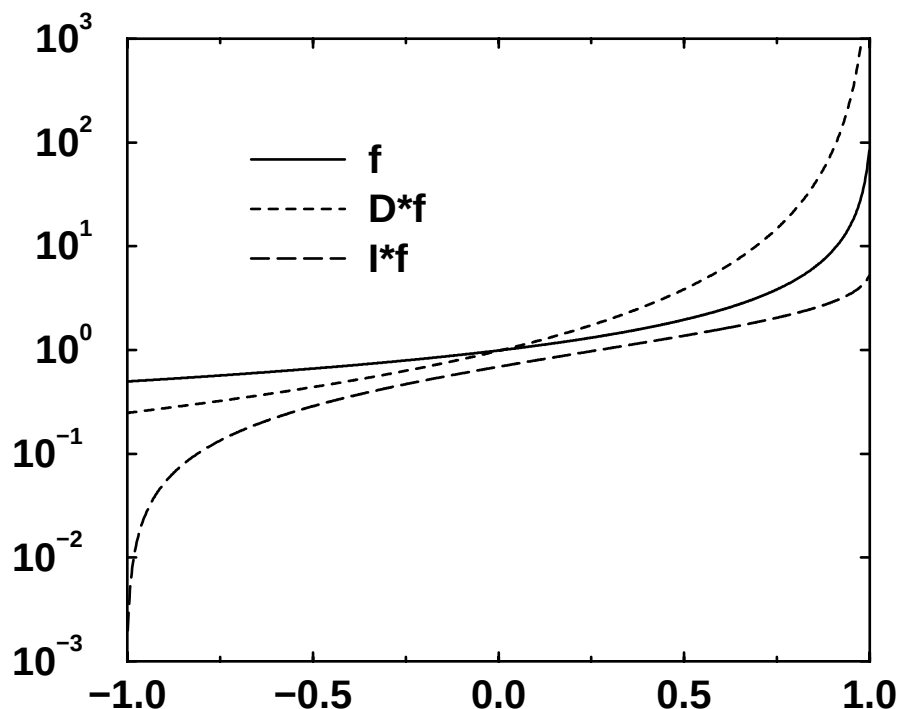
What does this program do?

Using the following formula for the product of a two-dimensional array with a vector

$$y_i = A_{i0}x_0 + A_{i1}x_1 + A_{i2}x_2 \cdots$$

we see that the “D” array differentiates and the “I” array integrates the function $f(x)$. Yes, you can do calculus with arrays as long as the spacing of x is such that the function $f(x)$ does not change too much from element to element. This is the basis of most calculations that solve differential equations. (You won’t be tested on the information in this last paragraph!)

This is what the result looks like:



8.2 Character arrays

8.2.1 One dimensional character arrays

- We have seen previously that the `string` class is really a `vector` in disguise. One can also string together characters in a more basic way using one-dimensional arrays. One should only use this form when it is inconvenient to use the `string` class, as in the example we have just been discussing, because there are some special rules to remember. Moreover, forgetting these rules can lead to problems related to over-running the boundaries of arrays. Indeed, some of these subtle behaviors are exploited by destructive computer hackers (the scourge of the computer industry) to break into computers and cause damage.
- So, a character string may be represented as a one-dimensional array of single characters that terminates with a specific character sentinel, the “null” character, *i.e.* `'\0'`. See! There is a use for something that represents nothing! So, we always have to remember to include this in the one-character array. If we forget to do this and try to interpret a one-dimensional array of characters as a string, then the code, as it is operating on the

string, will look for the end-of-string sentinel until it finds it, possibly reading (or more dangerously changing) memory locations that were not intended to be accessed. (This is a major destructive hacking tool! Don't do it!) The `string` class handles all this automatically for you, so that is why it is preferable to use the `string` class whenever possible.

- The following declares and initializes a string:

```
char s[4];

s[0] = 'H';
s[1] = 'i';
s[2] = '\0'; /* null */
```

Despite the fact that we have allotted 4 bytes for the string, we have only used 3 of them because that's where we put the `'\0'` terminator for that string. After the above statements `s[3]` is undefined.

- Note that the statements

```
s[2] = '\0'; /* null */
s[2] = 0;    /* the same thing */
```

are exactly the same thing.

- Strings define their own length with the `'\0'` terminator which acts as a “sentinel” for completion of the string.
- A simpler way of initializing a character string at the declaration stage is:

```
char s[] = {'H', 'i', '\0'};
```

which is more convenient than the previous one because there is no need to count the number of array elements.

- A simpler way still is:

```
char s[] = "Hi";
```

which uses the double quotes `"` to enclose what is called a *string literal* or a string constant. The above array `s[]` has length 3. The terminating sentinel `'\0'` is put in automatically by the compiler.

- It is a common mistake to forget about the terminating sentinel `'\0'`. For example, say we defined a character string called `mistake` in the following:

```
char mistake[] = {'b','o','o','b','o','o'};
```

As far as C++ is concerned, when the character array `mistake[]` is accessed as a string, this is not considered to be a string whose value is "booboo"! C++ considers this to be a string "booboo.....\0", that is, it keeps going until it runs into the terminating sentinel '\0', or it runs out of memory. This is an error that can easily crash a program or even worse, cause it to behave erratically.

Can you guess how the following program works?

```
//File: charArray.cpp
#include<iostream>
#include<string>

int main(void)
{ char correct[12] =
    {'I', 'A', 'm', 'C', 'o', 'r', 'r', 'e', 'c', 't', '!', '\0'};
  char moreData[] = "Will this print?";
  char someData[5] = {100, 97, 114, 110, 0};
  char mistake[8] = {'A', 'm', 'i', 's', 't', 'a', 'k', 'e'};

  cout << mistake << "\n" << correct << "\n";
  return 0;
}
```

8.2.2 Two dimensional character arrays

One can make arrays out of `char`'s. For example, a checkerboard could be represented by:

```
char checkerboard[8][8];
for (int i = 0; i < 8; i = i + 1)
    for(int j = 0; j < 8; j = j + 1)
        checkerboard[i][j] = ' '; //Initialize with spaces
checkerboard[0][0] = 'R'; //Place a red checker on the board
checkerboard[7][0] = 'B'; //Place a black checker on the board
```

8.3 Structures

So far we have been introduced to many forms of variables that contain only one item of information, `bool`'s, `char`'s, `int`'s, `float`'s and `double`'s. These are *fundamental* variable

types in C++, useful interpretations of bits and bytes that are *pre-defined* in the definition of the C++ language.

We have also seen more complex objects—classes like `vector`'s and `string`'s.

We have also seen arrays of one and two and more dimensions that are made up of only one of the above fundamental variables types but contain many items referred to by indexing within the array.

In many cases, these pre-defined variable types are insufficient to elegantly formulate a solution to an interesting problem. Fortunately, C++ gives the programmer the ability to make new *compound variable types* made up of a collection of the existing variable types. This compound variable type is called a *structure*.

For example, we could define a structure representing a person and the elements of the structure could be a string for the person's name, an `int` for the person's age, `float`'s for the person's height and weight, more strings for the person's address and occupation and so on. Structures are the essential composite data type for databases, which represent a relatively large component of the software industry.

Another example relates to the car racing application of Assignment 6. Let's say that we wanted to introduce new, racing-like features into the game, like oil slicks that reduce traction on the road surface, or the ability to draft behind the opponent's car. Every grid point representing a part of the race track would have to carry more information, the coefficient of friction on the surface, and the direction and number of seconds past the time that a car passed through it (so that the wind speed could be calculated). While these examples could be done by separating the information into different variables, why not put them together in a compound variable? After all, they are all related to one other.

8.3.1 The structure definition

An example of a structure definition:

```
// Define a structure called Student
struct Student
{   string name;
    int section;
    float grade;
};
```

The keyword `struct` begins a structure definition. The identifier `Student` is called the *structure tag*. It is important to note that this is a *definition* and not a *declaration*. All we are doing at this point is informing the compiler that we will be declaring variable types of the form `Student` later on in the program.

8.3.2 Where can structures be defined?

`struct`'s can be defined everywhere although it is conventional to define them outside of any programming unit such as `main` or another function so that all programming units will have access to this variable type.

8.3.3 The members of a structure

The variables within the braces `{}`'s are the *members* of the structure called `Student`. In this case the members defined, as suggested by the naming of the member types, are a string that will contain a name of a student, an `int` that will contain the section number of the student and a float that will contain the student's grade. The use of a `struct` seems quite natural in the application because the various things associated with any given student can be variables of various types.

8.3.4 Declaring structure variables

The *structure tag* is used with the keyword `struct` to declare (and optionally initialize) variables of the *structure type*. For example, if we had defined the above structure type `Student`, in the main routine or elsewhere we could declare and assign as follows:

```
struct Student
    Dan = {"Smith, Dan", 210, 79.8},
    Jan = {"Brown, Jan", 201, 89.3};
```

declares two structures of the `Student` type called `Dan` and `Jan` and assigns them names, section numbers and grades.

8.3.5 Assigning values to structure members

Individual structure members can be assigned values by using the `"."` notation of the form `structureName.structureMember`. For example, if we had not made the assignment in the declaration statement, we could have assigned values as follows:

```
Dan.name  = "Smith, Dan"; Dan.section = 210; Dan.grade = 79.8
Jan.name  = "Brown, Jan"; Jan.section = 201; Jan.grade = 89.3
```

8.3.6 Re-assigning values of structure members

Once defined and declared, we can treat these as regular variables, reassigning them, printing them, manipulating them in various ways. For example, we could make the following redefinitions:

```
Jan.name  = "Smith, Jan"; // Jan and Dan get married!
Jan.section = 210;        // Jan moves over to Dan's section (bad idea)
```

We can also treat a whole structure as a single variable. For example,

```
Dan = Jan;
```

reassigns the entire “Dan” structure, giving each member the values of the “Jan” structure.

Consult the complete working code called `structure0.cpp` that manipulates the “Dan” and “Jan” structures.

8.3.7 Function call-by-value of a structure

Structures can be called by value. This means that the structures are copied locally in the called function and the structures employed by the calling routine remain untouched. Here is an example called `structure1.cpp`:

```
//File: structure1.cpp
#include<iostream>
#include<string>

struct Student{string name; int section; float grade;};

void switchStruct(struct Student a, struct Student b)
{ // I/O statements left out
  struct Student temp = a; a = b; b = temp; // Switch a and b
  // I/O statements left out
  return;
}

int main(void)
{ struct Student
  Dan = {"Smith, Dan", 210, 79.8},
  Jan = {"Brown, Jan", 201, 89.3};
```

```
    // I/O statements left out
    switchStruct(Dan,Jan);
    // I/O statements left out
    return 0;
}
```

8.3.8 Function call-by-reference to a structure

The more conventional way is to use call by reference for structures, saving memory. Here is the same example as `structure1.cpp` but using called by reference. It is called `structure2.cpp`

```
//File: structure2.cpp
#include<iostream>
#include<string>

struct Student{string name; int section; float grade;};

void switchStruct(struct Student& a, struct Student& b)
{ // I/O statements left out
  struct Student temp = a; a = b; b = temp; // Switch a and b
  // I/O statements left out
  return;
}

int main(void)
{ struct Student
  {
    Dan = {"Smith, Dan", 210, 79.8},
    Jan = {"Brown, Jan", 201, 89.3};
    // I/O statements left out
    switchStruct(Dan,Jan);
    // I/O statements left out
    return 0;
  }
}
```

8.3.9 Structure arrays

One of the most common uses of `struct`'s is to make arrays of them. Let's now define an array of `struct`'s of the form `Student` and make each element of the array represent one of the students of the ENG101 course. Here is how it would be done. The code is called `structure3.cpp`. It creates a class of fictitious students, assigns section numbers and grades. Then it sorts them by name, section or grade, whatever the user wants.

```
//File: structure3.cpp
#include<iostream>
#include<iomanip>
#include<string>
#include<vector>
#include<cstdlib>

// Define a structure called Student
const int maxNameLength = 10;
struct Student
{ char name[maxNameLength];
  int section;
  float grade;
};

bool switchStruct(struct Student& a, struct Student& b)
{ struct Student temp = a; a = b; b = temp; // Switch a and b
  return false;
}

void sortStruct(vector<struct Student>& list, bool n, bool s, bool g)
{ bool sorted;
  if (n)
  { do
    { sorted = true;
      for (int i = 0; i < list.size() - 1; i = i + 1)
      { string string1 = list[i].name;
        string string2 = list[i+1].name;
        if (string1 > string2)
          sorted = switchStruct(list[i],list[i+1]);
      }
    }while (!sorted);
  }
  else if (s)
  { do
    { sorted = true;
      for (int i = 0; i < list.size() - 1; i = i + 1)
        if (list[i].section > list[i+1].section)
          sorted = switchStruct(list[i],list[i+1]);
    }while (!sorted);
  }
  else if (g)
```

```

    { do
      { sorted = true;
        for (int i = 0; i < list.size() - 1; i = i + 1)
          if (list[i].grade < list[i+1].grade)
            sorted = switchStruct(list[i],list[i+1]);
        }while (!sorted);
      }
    return;
  }

int main(void)
{ cout << "How many students in the class? ";
  int nStuds;
  cin >> nStuds;
  vector<struct Student> myClass(nStuds);

  for (int i = 0; i < nStuds; i = i + 1)
  { myClass[i].section = 200 + rand()%10;
    myClass[i].grade = 0.1*(rand()%1001);
    int nameLength = 2 + rand()%(maxNameLength - 2);
    for (int j = 0; j < nameLength; j = j + 1)
      myClass[i].name[j] = 97 + rand()%26;
    myClass[i].name[nameLength - 1] = '\0';
    myClass[i].name[0] = myClass[i].name[0] - 32;
    cout << setw(maxNameLength) << myClass[i].name << ", "
          << myClass[i].section << ", "
          << myClass[i].grade << "\n";
  }

  bool sortByName = false;
  bool sortBySection = false;
  bool sortByGrade = false;
  cout << "Sort by name (0 (no) or 1 (yes))? ";
  cin >> sortByName;
  if (!sortByName)
  { cout << "Sort by section (0 (no) or 1 (yes))? ";
    cin >> sortBySection;
    if (!sortBySection)
    { cout << "Sort by grade (0 (no) or 1 (yes))? ";
      cin >> sortByGrade;
    }
  }
}

```

```

    }

    sortStruct(myClass, sortByName, sortBySection, sortByGrade);
    for (int i = 0; i < nStuds; i = i + 1)
        cout << setw(maxNameLength) << myClass[i].name << ", "
            << myClass[i].section << ", "
            << myClass[i].grade << "\n";
    return 0;
}

```

8.3.10 Example using structures and arrays: Ion transport

The following code will be described in class:

```

//File: survivor.cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>

const int NX = 20;           // x-grid dimensions
const int NY = 7;           // y-grid dimensions
const int STEPS = NY;       // # of time steps
const int NCELL = 1000;     // # of pairs per cell
const int TOTAL = (NX*NY*NCELL); // total # of pairs

struct N {int nE; int nP;}; // # electrons, # ions

ofstream resultsOnFile; // Open an output file stream

//-----
//----- Function definitions -----
//-----

// This function initializes the number of e-'s and ions in each cell
void init(struct N grid[][NY])
{
    for (int i = 0; i < NX; i = i + 1)
    {
        for (int j = 0; j < NY; j = j + 1)
        {
            grid[i][j].nE = NCELL;
            grid[i][j].nP = NCELL;
        }
    }
}

```

```

    }
}
return;
}
//-----

// This function counts the number of e-'s
int countE(struct N grid[][NY])
{ int count = 0;
  for (int i = 0; i < NX; i = i + 1)
    for (int j = 0; j < NY; j = j + 1)
      count = count + grid[i][j].nE;
  return count;
}
//-----

// This function counts the number of positive ions
int countP(struct N grid[][NY])
{ int count = 0;
  for (int i = 0; i < NX; i = i + 1)
    for (int j = 0; j < NY; j = j + 1)
      count = count + grid[i][j].nP;
  return count;
}
//-----

// This function recombines the e-'s and ions
void recombine(struct N grid[][NY], int step)
{ for (int i = 0; i < NX; i = i + 1) // Loop over all rows
  { for (int j = step; j < NY; j = j + 1) // Loop over all columns
    { int nE = grid[i][j].nE;
      for (int k = 1; k <= nE; k = k + 1) // Loop over all e's
      { for (int l = 1; l <= grid[i][j].nP; l = l + 1)
        { // Potential annihilation with every ion in the cell
          if (0 == rand()%2000) // Small chance of annihilation
          { // If there is an annihilation, reduce numbers by 1
            grid[i][j].nE = grid[i][j].nE - 1;
            grid[i][j].nP = grid[i][j].nP - 1;
            break; // Break out of loop, only one life to give!
          }
        }
      }
    }
  }
}

```

```

    }
    }
    return;
}
//-----

/* This function shifts the electrons to the right and counts how
   many are collected. Note that, depending on the step, we can start
   at the y index where the number of e-'s is non-zero.
*/
int tStep(struct N grid[] [NY], int step)
{
    int escaped = 0;
    for (int i = 0; i < NX; i = i + 1)
    {
        escaped = escaped + grid[i][NY-1].nE;
        for (int j = NY - 2; j >= step; j = j - 1)
            grid[i][j+1].nE = grid[i][j].nE;
        grid[i][step].nE = 0;
    }
    return escaped;
}
//-----

// This function prints out the grid
void fGrid(struct N grid[] [NY], int step){

    resultsOnFile << "Step: " << step << "\n";
    for (int i = 0; i <= NX - 1; i = i + 1)
    {
        for (int j = 0; j <= NY - 1; j = j + 1)
            resultsOnFile << "|" << setw(5) << grid[i][j].nE
                        << "," << setw(5) << grid[i][j].nP;
        resultsOnFile << "|\n";
    }
    return;
}
//-----

//-----
//----- Main routine -----
//-----

int main(void)
{
    struct N grid[NX][NY]; // grid is an N struct

```



```

init(grid); // Initialize the grid

// Output starting conditions
cout << "Step " << setw(2) << 0
    << " : nE = " << setw(6) << countE(grid)
    << " : nP = " << setw(6) << countP(grid)
    << " : nEsc = " << setw(6) << 0 << "\n";

resultsOnFile.open("N.dat");
fGrid(grid, 0); // Write the grid to file

// Loop over time steps, set = to NY grid size
int tEsc = 0; // Total number of escapees
for (int step = 0; step < STEPS; step = step + 1)
{ recombine(grid, step); // Recombine e- & ions
  int nEsc = tStep(grid, step); // Take a time step
  cout << "Step " << setw(2) << step + 1
      << " : nE = " << setw(6) << countE(grid)
      << " : nP = " << setw(6) << countP(grid)
      << " : nEsc = " << setw(6) << nEsc << "\n";
  fGrid(grid, step + 1); // Write grid, next step
  tEsc = tEsc + nEsc; // Sum total escapes
}

resultsOnFile.close(); // Close the file

cout << "Chance of survival = " << setprecision(3)
    << static_cast<float>(tEsc)/TOTAL << "\n";
return 0;
}
//-----

```

8.4 Problems

1. Multiplying vectors by arrays

Complete the following program. The main routine declares an int array `a[ROWS][COLS]`. Your program should work for any values for `ROWS` and `COLS`, not just the ones provided. Read through `main()` to understand what it is doing. The function `print()` prints the array to `cout`. The function `randomFill()` fills the array with random int's between 0 and 9. *Hint:* Use `rand()%10`. The function `mMult()` returns a vector whose value of its `i`'th element is:

$$y[i] = a[i][0]*x[0] + a[i][1]*x[1] + a[i][2]*x[2] + \dots + a[i][NCOLS - 1]*x[NCOLS - 1];$$

Hint: Use for loops to do this. Supply the parameter lists for the three functions. Answer the following question:

Why is the return vector always full of zeros?

```
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;
const int ROWS = 5; // Set to 5 but can be anything
const int COLS = 4; // Set to 4 but can be anything

void print(/* Supply parameter list*/                                     )
{
    //Supply code for this function...
}

void randomFill(/* Supply parameter list*/                               )
{
    //Supply code for this function...
}

vector<int> mMult(/* Supply parameter list*/                             )
{
    //Supply code for this function...
```

```

}

int main(void)
{
    int a[ROWS][COLS]; // Declare the array
    randomFill(a); // Fills array with rand()%10 int's
    print(a); // Prints the array
    vector<int> x(COLS); // Declare a vector

    // Multiply the vector x by the array a, returning y
    vector<int> y = mMult(a,x);

    // Print the result
    for (int i = 0; i < ROWS; i = i + 1) cout << y[i] << endl;

    return 0;
}

```

2. Making arrays of various types

Consider the following main routine:

```

int main(void)
{
    int I[SIZE][SIZE] = {0}; ident(I); print(I);
    int U[SIZE][SIZE] = {0}; upper(U); print(U);
    int L[SIZE][SIZE] = {0}; lower(L); print(L);
    int H[SIZE][SIZE] = {0}; hatch(H); print(H);

    return 0;
}

```

The 5 functions called by `main()` produce the output shown following, when `SIZE = 10`. Write the 5 functions that complete this program. Your program has to work for any `SIZE` greater than 0.

(Output from the completed program, when `SIZE = 10`)

```

1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0

```

```
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

```
1 1 1 1 1 1 1 1 1 1
0 1 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 1 1
0 0 0 0 0 1 1 1 1 1
0 0 0 0 0 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 1
```

```
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0 0
1 1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1 0
```

3. Vectors and arrays

Complete the program started below.

The main routine declares an int array `a[ROWS][COLS]`. Your program should work for any values for `ROWS` and `COLS`, not just the ones provided. Read through `main()` to understand what it is doing. The function `randomFill()` fills the array with random int's between 0 and 3 inclusive. *Hint:* Use `rand()%4`. The function `sumRowsAndCols()`

sums the rows and columns putting the results into two separate vectors, `sumRows`, a vector `ROWS` in length containing the sum of each row, and `sumCols`, a vector `COLS` in length containing the sum of each column: The function `print()` prints the array to `cout` and the vectors as shown in the example below. Note, pretty formatting is not important, but the numbers have to be correct. Supply the parameter lists for the three functions.

Example use:

```
> a.out
2 2 1 3 3 : 11
3 2 3 0 2 : 10
1 3 0 0 1 : 5
3 1 3 1 2 : 10
- - - - -
9 8 7 4 8

#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;
const int ROWS = 4; // Set to 4 but can be anything
const int COLS = 5; // Set to 5 but can be anything

int main(void)
{   int a[ROWS][COLS];
    randomFill(a);

    vector<int> sumRows(ROWS);
    vector<int> sumCols(COLS);

    sumRowsAndCols(a,sumRows,sumCols);

    print(a,sumRows,sumCols);

    return 0;
}
```

4. More vectors and arrays

Consider the following main routine:

```
const int ROWS = 10; //Constant defined as global
```

```

const int COLS = 8; //Constant defined as global

int main(void)
{  int a[ROWS][COLS] = {0};

    vector<int> r(ROWS);
    for(int i = 0; i < r.size(); i = i + 1) r[i] = rand()%4; // 0 <= r[i] <= 3
    vector<int> c(COLS);
    for(int i = 0; i < c.size(); i = i + 1) c[i] = rand()%4; // 0 <= c[i] <= 3

    outerProduct(a,r,c);
    print(a,r,c);

    return 0;
}

```

Write the two functions `outerProduct()` and `print()`. `outerProduct()` computes the outer product of two vectors, as follows:

```
a[i][j] = r[i]*c[j];
```

where `r[i]` and `c[j]` are vectors. `print()` prints out the vectors and array as shown below. You do not have to concern yourself with pretty formatting, but the numbers should come out in the order shown. Your program has to work for any positive values for `ROWS` and `COLS`.

Here is how it works:

```

red% a.out
v:      1 0 2 2 3 2 1 2
      -----
r[0]: 3| 3 0 6 6 9 6 3 6
r[1]: 1| 1 0 2 2 3 2 1 2
r[2]: 0| 0 0 0 0 0 0 0 0
r[3]: 3| 3 0 6 6 9 6 3 6
r[4]: 1| 1 0 2 2 3 2 1 2
r[5]: 3| 3 0 6 6 9 6 3 6
r[6]: 3| 3 0 6 6 9 6 3 6
r[7]: 1| 1 0 2 2 3 2 1 2
r[8]: 1| 1 0 2 2 3 2 1 2
r[9]: 1| 1 0 2 2 3 2 1 2

```

5. Complex struct's

This problem uses a `struct` defined as follows:

```
struct Complex {double real; double imag;};
```

We may use `struct Complex`'s to define “complex numbers”. A complex number is really a pair of numbers, the “real” part and the “imaginary” part, as suggested by the above `struct` definition. If $a = (a_R, a_I)$ and $b = (b_R, b_I)$ are two complex numbers, their product is also a complex number $ab = (a_R b_R - a_I b_I, a_R b_I + a_I b_R)$. Write a function that accepts two called-by-reference `vector<struct Complex>`'s, both of the same size and returns a `vector<struct Complex>` that contains the product according to the above rule.

6. Using structure lists

Consider the following structure definition:

```
struct Student {char name[10]; int section; float grade;};
```

Write a function compatible with the following function call:

```
adjustGrades(list);
```

where `list` is a `vector<struct Student>` with size > 0 . Your function will do the following:

- (a) Raise all the grades in the class 5% to a maximum of 100.
- (b) Give everyone in the class whose name starts with “Al” a grade of 100.
- (c) Sort the list by grades, highest to lowest.

8.5 Projects

1. Battleship! Sink this one!

You will write C++ code that will allow you to play the game “Battleship!” on your computer following the rules given below.

The game is played on a 10 by 10 array of `char`’s. A battleship is represented by a series of `sizeShip` contiguous (joined) points that lie in a horizontal, vertical or diagonal line. This can be represented by some special character in a 10 by 10 character array, say `'S'`. For example, if the array everywhere was initialized to the character `'.'` and you wrote, `board[2][3] = 'S'; board[3][4] = 'S'; board[4][5] = 'S'; board[5][6] = 'S';` it would represent a battleship on a diagonal occupying the above indices.

For each ship you will ask the computer to generate `sizeShip` such contiguous points on a line starting from a random position within the entire array. For example, if you:

```
#include<cstdlib>
```

in the preprocessor area, the code:

```
i = rand()%10; j = rand()%10; board[i][j] = 'S';
```

picks a random point in the array called `board` and assigns it the value `S`, indicating that it is one of the points representing a ship. This is the first point on the ship. The hard part is to pick `sizeShip - 1` more points on a line that are contiguous with the first point and that remain within the array. This you have to figure out on your own. The orientation of the battleship should be random as well. You have to make the part of the program that determines the ship’s position and orientation a separate C++ function and the array name must be an argument (in the parameter list) of this function. When you come to the part where you are putting more than 1 ship on the board, it gets even harder because no 2 ships can occupy the same space.

Now comes the fun part. You (the user of the program) have `50 + shipSize*nShips/2` “bombs” to toss at the battleship, where `nShips` is the number of enemy ships. The battleship position and orientation are random and are kept secret from you until the code finishes, so you do not know where it is while you are playing. You have to drop the bomb on the array, which is equivalent to selecting an element of the array, by asking the user to provide two indices that lie within the board. If that array element is an `S`, you have hit the ship and the program informs you that you have hit the ship. (See the example below.) You have to hit all locations of a ship to sink it and you have to sink every ship to win the game.

If you hit a ship once, the game strategy simplifies. You look for adjacent points (8 possibilities). If you hit it again, you know the orientation. Now, all you have to do is make sure you get all the points that lie on the line.

Win or lose, print out a representation of the board that shows clearly: the location of the ship, the points of the ship that were struck, and the points where bombs were dropped but missed.

Here is an example of a typical play of the game. Your computer output does not have to look exactly as follows. However, you have to follow the above rules explicitly.

Number of enemy ships? 1

Size of the enemy ships? (≤ 10) 4

Bombs away # 1: i j: 3 4

Bombs away # 2: i j: 7 2

Bombs away # 3: i j: 4 1

Bombs away # 4: i j: 3 6

Bombs away # 5: i j: 9 2

Bombs away # 6: i j: 6 0

Bombs away # 7: i j: 2 6 Ouch, that hurts!

Bombs away # 8: i j: 2 7

Bombs away # 9: i j: 2 5

Bombs away # 10: i j: 3 6

Bombs away # 11: i j: 3 7

Bombs away # 12: i j: 3 5 Ouch, that hurts!

Bombs away # 13: i j: 4 4 Ouch, that hurts!

Bombs away # 14: i j: 5 3 Ouch, that hurts!

Ugh, ya got me! Glub, glub, glub...

```

. . . . .
. . . . .
. . . . . o * o . .
. . . . . o * o o . .
. o . . * . . . .
. . . * . . . . .
o . . . . .
. . o . . . . .
. . . . .
. . o . . . . .

```

A "." is "open sea".

An "*" is a "hit on an enemy ship".

An "S" is a "surviving enemy ship".

An "o" is a "missed bomb".

2. Ladies and Gentlemen: Start your engines!

Write a computer program that simulates a race car negotiating an S-curve track and either crashing or reaching the finish line. The game is played by either one player racing against the clock, or two players competing against each other.

The game starts with a printout of a 2-D grid representing the track made up of a two-dimensional array of characters (`char`'s) with 52 rows and 72 columns. The picture on the following page represents the starting condition for a one-person game. The grid can be printed to the terminal screen by writing the entire `char` grid to `cout`. If you have trouble seeing the whole grid in a single window, try using `<Control>-<right mouse button>` in an XTerm window and resizing the window to the whole screen.

The outer boundary of the track and the two impenetrable barriers are represented by 'X's, the open track by spaces ' ', the finish line by 'F's (lower right hand corner) and the car, pictured at its starting location, `grid[1][1]` (upper left hand corner), by an 'O'. In the following picture, the rows go down the page top to bottom and the columns go left to right.

Here's how the game is played. The user is continually prompted (until the game ends) for a component of horizontal (positive is left to right) and vertical acceleration (positive is top to bottom). (You imagine that this prompt is given once per second, that is, the user has a chance to change the acceleration once/second.) The user responds with either -1, 0, or 1 for each of these, effectively steering, accelerating and braking the car. If the user responds with a number outside of this range, the car crashes and the game is over, because either the engine explodes or the brakes and tires burn up—real concerns for real race cars.

The velocity in both the vertical and horizontal directions starts at 0 and increases or decreases after each prompt, for example, `xVelocity = xVelocity + xAcceleration`, and the same for the *y* direction. Then, both components of the position are changed, for example, `xPosition = xPosition + xVelocity`, and the same for the *y* direction. A car crashes when its position takes it outside the racetrack (except when it crosses the finish line) or into a wall or barrier, or an improper acceleration was input. Crossing the finish line “wins” the race.

The starting grid in a one-person game (note prompt for acceleration at bottom):

[illegible]

To start the programming, write a working main routine and a function called `initRace` that initializes the grid to the given starting configuration, and a function called `printRace` that prints the grid. *The walls and barriers have to be exactly as shown in the figure on the last page.* The barrier on the left is 20 columns wide and 35 rows high. The barrier on the right is 25 columns wide and 35 rows high.

Then, program a one-person and a two-person game. Keep track of the total time for each player. If a player crashes he is out of the race. The first across the finish line wins! Hint: Program a one-person game first, then adapt your program for two people.

Warning: This game is addictive!

Consider modifying the program to do one of the following. These are quite challenging but well within your capabilities.

- In a one-person game, keep track of the real time it takes to respond and adjust the algorithm to work with real time. (Use the standard time library `ctime`. You will have to look up the standard time functions in a different book.)
- Program a one-person-against-the-computer game with multiple plays. Initially, the computer makes random moves. Once the player has crossed the finish line, the computer “learns” the players solution and tries to improve it.
- Program some nice graphics for this game. (Good opportunity to read ahead in Matlab.)
- Program a two-person game but one that generates many barriers randomly in a game with very many columns that scrolls down the screen.
- Dream up some other nice variant of the game.

Sample plays of the race on the following pages

User input: $(0,1),(0,1)(0,1),(0,1),(0,1),(1,1),(1,1),(1,1),(1,1),(1,1)$

[illegible]


```

0 0 0 0    0 0 0 0
0 0 0 0    0 0 0 0
0 0 0 0    0 0 0 0

Enter your guess: 3 4 5 5
  HUMAN      MACHINE
  =====
  ? ? ? 5    ? 2 ? ?
  -----
  1 2 3 4    1 1 1 1
  2 3 4 5    2 2 2 2
  3 4 5 5    3 2 3 3
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0

```

```

Enter your guess: 4 5 6 5
  HUMAN      MACHINE
  =====
  ? 5 ? 5    4 2 ? ?
  -----
  1 2 3 4    1 1 1 1
  2 3 4 5    2 2 2 2
  3 4 5 5    3 2 3 3
  4 5 6 5    4 2 4 4
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0

```

```

Enter your guess: 5 5 7 5
  HUMAN      MACHINE
  =====
  5 5 7 5    4 2 5 ?
  -----
  1 2 3 4    1 1 1 1
  2 3 4 5    2 2 2 2
  3 4 5 5    3 2 3 3
  4 5 6 5    4 2 4 4
  5 5 7 5    4 2 5 5
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0
  0 0 0 0    0 0 0 0

```

Curses, human, you win!

Chapter 9

Miscellaneous Topics

9.1 Generating random numbers

The C++ standard library, `<cstdlib>`, provides a way of producing a random number, actually a “pseudo” random number using the `rand()` function. The `rand()` function produces a pseudo random integer between 0 and `RAND_MAX`, inclusive. The constant `RAND_MAX` is also defined in `<cstdlib>`.

Here is an example program, called `rand0.cpp` that calls `rand()`. Note the inclusion of `<cstdlib>`.

```
//File: rand0.cpp
#include <iostream>
#include <iomanip>
#include <cstdlib> // Need this for the rand() function

int main(void)
{   cout << "RAND_MAX = " << RAND_MAX << "\n\n";
    cout << "The first 10 random numbers are...\n";

    for(int i = 1; i <= 10; i = i + 1)
        cout << "Random number "
              << setw(2)
              << i << " = "
              << rand() // Random int between 0 and RAND_MAX inclusive
              << "\n";
    return 0;
}
```

Compiling and executing this results in:

```
RAND_MAX = 2147483647
```

The first 10 random numbers are...

```
Random number 1 = 1804289383
Random number 2 = 846930886
Random number 3 = 1681692777
Random number 4 = 1714636915
Random number 5 = 1957747793
Random number 6 = 424238335
Random number 7 = 719885386
Random number 8 = 1649760492
Random number 9 = 596516649
Random number 10 = 1189641421
```

Note that `RAND_MAX` and the implementation of `rand()` is system dependent. You should not expect the same values on a different computer or even a different version of the operating system on the same computer.

If you run this program again, you will note that the same random numbers are produced! This sort of goes against the idea that `rand()` produces random numbers! Actually, this is a benefit. Since the behavior is predictable, it makes it easier to debug. Imagine if you were trying to debug a program that always gave different results! You would go bonkers trying to find the cause of the error. So, predictable random numbers are good. But, this is why they are called “pseudo” random numbers. They are made from a mathematical prescription that is remarkably simple. Yet, they pass many different tests of “randomness”. They also fail some tests as well but not ones that we are likely to try.

If you are not happy with the same random sequence every time, you can “seed” the random number to produce another sequence. The `<cstdlib>` function `srand()` does this. All you have to do is provide a positive integer to the argument list of `srand()`. Here’s an example called `rand1.cpp`:

```
//File: rand1.cpp
#include <iostream>
#include <iomanip>
#include <cstdlib> // Need this for the rand(), srand() functions

int main(void)
{   cout << "RAND_MAX = " << RAND_MAX << "\n\n";
    cout << "Input any int to seed the random number generator: ";
    int seed;
```

```

    cin >> seed;
    srand(seed);
    cout << "The first 10 random numbers are...\n";

    for(int i = 1; i <= 10; i = i + 1)
        cout << "Random number "
            << setw(2)
            << i << " = "
            << rand() // Random int between 0 and RAND_MAX inclusive
            << "\n";
    return 0;
}

```

Compiling and executing the above code results in:

```
RAND_MAX = 2147483647
```

```
Input any int to seed the random number generator: 123456789
```

```
The first 10 random numbers are...
```

```

Random number 1 = 1965102536
Random number 2 = 1639725855
Random number 3 = 706684578
Random number 4 = 1926601937
Random number 5 = 71238646
Random number 6 = 1147998030
Random number 7 = 1038816544
Random number 8 = 940714160
Random number 9 = 789063065
Random number 10 = 464968134

```

Note the different sequence because the user provided the seed 123456789 to `srand()`.

One way to make the start-up of a program that uses `rand()` appear to be different every time, is to seed the random number generator with the `int` that is provided by the `time()` function. The `time()` function is part of the standard library `<ctime>`. Here's an example code called `rand2.cpp` that uses it:

```

//File: rand2.cpp
#include <iostream>
#include <iomanip>
#include <cstdlib> // Need this for the rand() and srand() functions
#include <ctime> // Need this for the time() function

```

```

int main(void)
{  cout << "RAND_MAX = " << RAND_MAX << "\n\n";
    srand(time(NULL)); // Seed the random number generator
    cout << "The first 10 random numbers are...\n";

    for(int i = 1; i <= 10; i = i + 1)
        cout << "Random number "
                << setw(2)
                << i << " = "
                << rand() // Random int between 0 and RAND_MAX inclusive
                << "\n";
    return 0;
}

```

Try running `rand2.cpp` and verify that you get different numbers every time. This unique start-up feature is very useful for programming games, in order to produce a game that starts differently every time.

The interesting uses of `rand()` involve making decisions based on the output of `rand()` and this simulates random behavior. In many applications you can use the modulus function to do this. For example,

```
card = rand() % 13 + 1
```

has a one in 13 chance of being a one, a one in 13 chance of being a two, *etc.* Thus, you could use the above statement to simulate the random dealing of cards and you could assign a one to an “ace”, a 2 to a “deuce”, 11 to a “jack”, *etc.* If you were dealing with a finite series of cards, however, you would have to take some care that you do not deal, say, 5 aces and that would require some extra care with the coding. We won’t go into further detail here on this topic except to say that dealing 5 aces can get you into BIG trouble in certain circumstances.

Another way of using random numbers, more common to scientific disciplines, is to convert the random integer to a random double or float, usually converted so that the number lies between 0 and 1 (may be inclusive or exclusive of one or both of the endpoints).

Here is an example code, called `rand3.cpp`, that converts the random integer to a double between 0 and 1 inclusive:

```

//File: rand3.cpp
#include <iostream>
#include <iomanip>
#include <cstdlib> // Need this for the rand() and srand() functions
#include <ctime> // Need this for the time() function

```

```

int main(void)
{  cout << "RAND_MAX = " << RAND_MAX << "\n\n";
   srand(time(NULL)); // Seed the random number generator
   int nRandoms = 10;
   cout << "The first " << nRandoms << " random numbers are...\n";

   for(int i = 1; i <= nRandoms; i = i + 1)
   {  double randomDouble = rand()/static_cast<double>(RAND_MAX);
      cout << "Random number "
           << setw(2)
           << i << " = "
           << randomDouble // Random double between 0 and 1 inclusive
           << "\n";
   }
   return 0;
}

```

Here's an example of its use:

```
RAND_MAX = 2147483647
```

```
The first 10 random numbers are...
```

```
Random number  1 = 0.174225
```

```
Random number  2 = 0.130327
```

```
.
.
.
```

Application: Calculating π by throwing darts

If you are as bad a dart player as I am, throwing a dart at a circular object is truly a random process. Imagine that you have a circle inscribed in a square. It is easy to show that the ratio of the area of the circle to the area of the square is $\pi/4$. This suggests a way to estimate what π is by throwing darts! However, it only works well if you are a bad dart player, like me.

Set up a circular dartboard on a wall. Draw a square on the wall so that the circle defining the dartboard touches each side of the square only once. Now start throwing the darts (badly). Ignore all the darts that land outside of the square. Throw a lot of darts! Count the ones that land on the dartboard and all the ones that land within the square. The ratio should be close to $\pi/4$. And, your estimate gets better the more darts you throw. **Warning:**

Don't do this in your dorms. Don't do this at home. Try it outside against a fence or something and make sure that other people, pets, your mother's favorite vase, *etc.* do not get damaged.

A much safer way is to do it with a computer code. The following code, called `randomPi.cpp` calculates π by random sampling:

```
//File: randomPi.cpp
#include <iostream>
#include <cstdlib> // Need for rand() and srand()
#include <cmath> // Need for pow() and atan()
#include <ctime> // Need for time()

/* Define a conversion function that converts the int returned from
   rand() to a double between -1.0 and 1.0 inclusive
*/
#define MYRAND (2.*rand()/RAND_MAX - 1.)

int main(void)
{ int nDarts = 10000; // Number of darts to throw
  int nInside = 0; // Counter for those that land inside
  srand(time(NULL)); // Seed the random number generator

  for(int i = 1; i <= nDarts; i = i + 1)
  { double xDart = MYRAND;
    double yDart = MYRAND;
    if (pow(xDart,2) + pow(yDart,2) <= 1.0) nInside = nInside + 1;
  }

  double pi = 4.0*nInside/nDarts; // Estimate for Pi
  double Pi = 4.0*atan(1.0); // This is how you can get a very
                             // accurate estimate of the real Pi!

  cout << "The estimated value of Pi is " << pi << "\n";
  cout << "The true value of Pi is " << Pi << "\n";

  return 0;
}
```

There are a few new features in this program that you have not seen before.

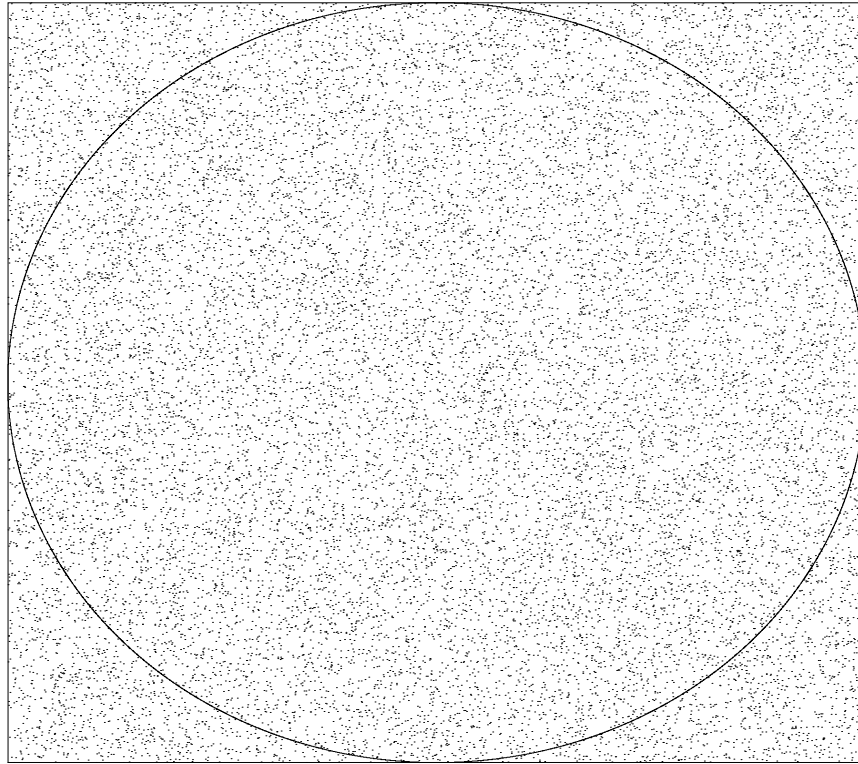
- The statement
`#define MYRAND (2.*rand()/RAND_MAX - 1.)`

in the pre-processor area replaces every occurrence of `MYRAND` throughout the rest of the file with `(2.*rand())/RAND_MAX - 1.)`. This happens before compilation. It is exactly as if you took an editor, looked for every occurrence of `MYRAND` and pasted in its replacement `(2.*rand())/RAND_MAX - 1.)`. You can be as creative as you wish with `#define` 's, but make sure that you understand this as a cut and pasting procedure. Generally, you should avoid `#define` 's because the C++ parser does not help you much with syntax errors. But, `#define` 's are a part of the language and using them can make life a lot easier sometimes. Just be careful with them.

- The expression `(2.*rand())/RAND_MAX - 1.)` produces a random `double` between -1 and 1 inclusive. Why? `rand()` produces an `int` between 0 and `RAND_MAX`, inclusive. Following the *rules of precedence* and the *rules of variable type promotion* multiplying it by “2.” produces a random `double` between “0.” and “2.*`RAND_MAX`”. Dividing by `RAND_MAX` produces a random `double` between “0.” and “2.”. Finally, subtracting “1.” from it produces a random `double` between “-1.” and “1.”. There is a lot going on in this simple-looking expression.
- When you throw a dart on a 2-dimensional board, you are really choosing randomly an x -coordinate and a y -coordinate. So the statements:
`double xDart = MYRAND; double yDart = MYRAND;`
 pick a random point in a square defined by the corners:
 $(x,y) = (-1, -1), (-1, 1), (1, 1), (1, -1)$.
- The `pow()` function, part of the standard library `<cmath>`, takes the power of the first argument to degree of the second argument. For example, `pow(x,3)` computes x^3 .
- So, the statement:
`if (pow(xDart,2) + pow(yDart,2) <= 1.0) nInside = nInside + 1;`
 computes $x^2 + y^2$ and compares it to 1. If $x^2 + y^2 \leq 1$, the dart has landed in the circle and it is counted.
- Finally, the standard library math function `atan()`, also part of `<cmath>` computes the arc tangent of its argument, assumed to be in *radians*. Since $\tan^{-1}(1) = \pi/4$, `4.0*atan(1.0)` is a very accurate determination of π , as good as the math library can provide, at least for a `double`.

Here's what 20,000 points tossed at a circle look like, generated by the program above.

$$n(\text{inside circle})/n(\text{total}) = \pi/4$$



9.1.1 Example: Integrating functions by random sampling

The same idea may be used to integrate functions by random sampling. The following code, called `randomIntegrate.cpp` integrates $\exp(-x)$ for $0 \leq x \leq 1$.

```
//File: randomIntegrate.cpp
#include <iostream>
#include <iomanip>
#include <cstdlib> // Need for rand() and srand()
#include <cmath> // Need for exp(), the exponential function
                // and abs(), the absolute value function

/* Define a conversion function that converts the int returned from
   rand() to a double between 0.0 and 1.0 inclusive
*/
#define MYRAND (static_cast<double>(rand())/RAND_MAX)
```

```

int main(void)
{
    int nDarts = 1000000000, under = 0;
    double Ans = 1 - exp(-1);
    for(int i = 1; i <= nDarts; i = i + 1)
    {
        double xDart = MYRAND;
        double yDart = MYRAND;
        if (yDart <= exp(-xDart)) under = under + 1;
        if (0 == i%1000000) // Every million times
        {
            double ans = static_cast<double>(under)/i;
            cout << setiosflags(ios::fixed) << setprecision(8)
                 << "#" << setw(10) << i
                 << ": I = " << setw(10) << ans
                 << ", Delta = " << abs(1 - exp(-1) - ans) << "\n";
        }
    }
    return 0;
}

```

There are several new features in this code:

- The standard math library functions defined in `<cmath>`, `exp()`, for the exponential e^x and `abs()` for the absolute value, are used.
- `setiosflags(ios::fixed)` allows floating point numbers to print with trailing zeros, nice for formatting. This only has to be done once.
- `setw()` sets the width of the output of the following item. It has to be set every time! `setw()` is defined in `<iomanip>`.
- `setprecision()` set the precision of the following number. `setprecision()` is also defined in `<iomanip>`.

The output of the above program looks like:

```

# 1000000: I = 0.63213700, Delta = 0.00001644
# 2000000: I = 0.63193800, Delta = 0.00018256
# 3000000: I = 0.63197933, Delta = 0.00014123
# 4000000: I = 0.63180525, Delta = 0.00031531
# 5000000: I = 0.63187300, Delta = 0.00024756
.
.
.

```

The program operates very much like the program that found π by random sampling. We find an x and a y randomly. However, the only difference is that we count only those points that fall under the function as contributing to the area under the curve. The area under the curve is the fraction of points that fall under the curve, times the area of the rectangle. In the above example, the area of the rectangle is 1. If the rectangle did not have unit area, how would you know what to multiply by? That's what you have to figure out in assignment 4!

9.2 Simple Sorting: The bubble or sinking sort

This is called the bubble or sinking sort technique because small items “bubble” to the top and large items “sink” to the bottom. The technique involves looping through an array looking at pairs of values in order. If the “upper” item is bigger than the “lower” item, they are switched. Then the next pair is considered. The “lower” item of the previous pair is the “upper” item of the current pair. The iteration continues until you can go through the entire array and no pair is switched.

For example, imagine that you have the following array $\{3, 2, 1\}$ that you want sorted in ascending order. We will use the convention that parentheses surround the pair under consideration.

Here is how it would work:

Loop 1: $\{3, 2, 1\} : \{(3, 2), 1\} \Rightarrow \{2, (3, 1)\} : \{2, 3, 1\}$ Change, continue
 Loop 2: $\{2, 3, 1\} : \{(2, 3), 1\} \Rightarrow \{2, (3, 1)\} : \{2, 1, 3\}$ Change, continue
 Loop 3: $\{2, 1, 3\} : \{(2, 1), 3\} \Rightarrow \{1, (2, 3)\} : \{1, 2, 3\}$ Change, continue
 Loop 4: $\{1, 2, 3\} : \{(1, 2), 3\} \Rightarrow \{1, (2, 3)\} : \{1, 2, 3\}$ No change, stop

Here is some “boiler plate” pseudocode that gets the job done:

```
bool whileThingsAreChanging = true;
while(whileThingsAreChanging)
{ whileThingsAreChanging = false; //Assume that the list is sorted
  for(/* appropriate for-loop parameters */) //Careful!
  { //Loop through the list in an order fashion in one direction
    //Compare adjacent pairs of things
    //If the pair is out of order, switch them and set
    // whileThingsAreChanging to true
  }
  //When the for loop completes, either nothing has changed and
  //whileThingsAreChanging has remained false, or something changed and
  //whileThingsAreChanging was changed to true
```

```

}
//When the execution arrives here, the list is sorted.

```

9.3 Recursion—A function calling itself

Sometimes in mathematics, there are very compact ways of writing rules for generating complicated things. For example, the factorial can be written out explicitly as:

$$N! = N \times (N - 1) \times (N - 2) \cdots 3 \times 2 \times 1$$

or, it can be written more compactly as

$$N! = N \times (N - 1)! \quad \forall N \geq 0 \quad \text{where} \quad 0! \equiv 1$$

where N is an integer.

Mathematicians prefer the second expression because it is better defined (no assumptions about what $\cdots$ means) and it also suggests a mechanism or process for calculating $N!$ C++ mimics this by providing the programmer a mechanism called *recursion*—the ability for a function to call itself. Here is how the factorial can be calculated using this technique.

```

//File: recursiveFactorial.cpp
#include <iostream>

int factorial(int n)
{ if (0 == n) return 1;
  else return (n * factorial(n - 1));
}

int main(void)
{ cout << "N? ";
  int N;
  cin >> N;
  cout << N << "! = " << factorial(N) << endl;
  return 0;
}

```

Note how compact the coding is, and how similar it is to the second mathematical definition. Note also how the $0!$ is coded in the if statement. This is critical to stop the recursion process. Try writing out the stack frames for a simple case, say $5!$, to see how this works.

There are a few simple, general guidelines for using recursion:

- Successive calls to functions should represent progressively simpler cases. (For example, $(N - 1)!$ is “simpler” than $N!$)
- There is an end to the recursion, the “base case”, that does not involve a recursive call. (For example $0! = 1$)
- If a function calls itself more than once, and the “nesting level” gets deep, the calculation can be very inefficient. (See the following example.)
- Recursion is a great way to crash a computer or a program! If you do not program the stopping criterion correctly, you will cause your stack to collide with your heap, and your program will be in a heap of trouble.

To illustrate the third point above, consider the calculation of the Fibonacci numbers defined as follows.

$$f(N) = f(N - 1) + f(N - 2) \quad \forall N > 0 \quad \text{where} \quad f(1) \equiv 1 \quad f(2) \equiv 1$$

where N is an integer. The code can be written using recursive calls as follows:

```
//File: recursiveFibonacci.cpp
#include <iostream>

int fibonacci(int n)
{ if (1 == n || 2 == n) return 1;
  else return (fibonacci(n - 1) + fibonacci(n - 2));
}

int main(void)
{ cout << "N? ";
  int N;
  cin >> N;
  cout << "Fibonacci(" << N << ") = " << fibonacci(N) << endl;
  return 0;
}
```

The `fibonacci` function calls itself twice. This is a **NO, NO!** unless N is quite small, like less than 30. Don't believe it? Try calculating $f(1000)$!

Although the above program is elegant and compact, we sometimes have to give up clarity for efficiency, yet another compromise we must sometimes make. Here is how one can program the Fibonacci numbers using loops.

```
//File: loopFibonacci.cpp
#include <iostream>

int fibonacci(int n)
{ if (1 == n || 2 == n) return 1;
  else
  { int fMinus2 = 1;
    int fMinus1 = 1;
    int f;
    for (int i = 3; i <= n; i = i + 1)
    { f = fMinus2 + fMinus1;
      fMinus2 = fMinus1;
      fMinus1 = f;
    }
    return f;
  }
}

int main(void)
{ cout << "N? ";
  int N;
  cin >> N;
  cout << "Fibonacci(" << N << ") = " << fibonacci(N) << endl;

  return 0;
}
```

It's ugly but it's fast. Try comparing the speed of execution of the above two methods. You will be surprised at the difference.

Recursion relations play a big role in mathematics and consequently there are many examples that could be coded in C++ and used as examples. However, another important use of recursion relations relates to graphics. The following example recursively splits a line into two smaller segments chosen randomly but with the same total length. This is a very simple model of how a tree grows. The output of this program is the total number of branches according to the user's input of the largest branch length and the smallest branch length.

```
//File: tree.cpp
#include <iostream>
#include <cstdlib>

int tree(float L, float LMin)
{ if (L <= LMin) return 0;
```

```

    else
    { float split = static_cast<float>(rand())/RAND_MAX;
      return 2 + (tree(split * L, LMin) + tree((1 - split) * L, LMin));
    }
}

int main(void)
{ cout << "LTotal, LMin? ";
  float LTotal, LMin;
  cin >> LTotal >> LMin;
  cout << "Number of branches = " << tree(LTotal, LMin) << endl;

  return 0;
}

```

There are two companion programs that will be demonstrated in class that provide graphical output from this program. These are called `treeGraphics.cpp` and `anotherTree.cpp`, both posted on the web. In class, the close connection to fractals will be evident from the graphical output. Fractals are interesting not only mathematically, but have great uses in communications, encryption and virtual cinematography!

Before closing the section on recursion, it is worthwhile to note that this technique is also used in numerical simulation, something the author of this lecture is passionate about. In fact, the coding in the tree programs shares a lot in common with, to cite a few examples, scattering of light in the atmosphere, the movement of neutrons in a nuclear reactor, the scattering of electrons and photons in the human body during diagnostic X-ray procedures or treatment of cancer with radiotherapy beams. Just a few of the details are different. In order to find out what these details are, you will have to take a later course in numerical simulation.

9.4 Input and Output using files

9.4.1 File and stream handling in C++

Why files?

- Except for a few cases where we generated random data using quasi-random numbers, so far we have been getting data into our programs by typing input at the keyboard and any output that our programs have generated has gone to the computer screen.
- This is really only useful for limited amounts of data, either on the input side or the output side.

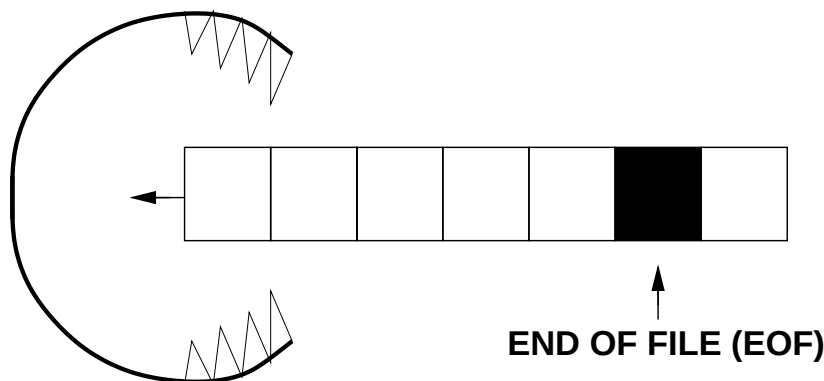
- A large quantity of data is handled more easily and more reliably with files.
- Moreover, the same data files can be used by more than one program.

How is data organized?

- Computers only know about bits - binary digits, 0's and 1's. These are the only fundamental data types understood by computer hardware.
- Humans, on the other hand, are more comfortable with characters - and these, as we have come to know are usually represented by 8 bits, a byte, by computers.
- Characters can be combined into *fields*—combinations of characters that have some unique meaning, like a floating-point number, an integer, a string of characters representing a person's name, *etc*
- Character fields can be gathered into *records*—of related fields, for example, a person's name, social security number, age, salary, *etc* as many fields as are needed for a given application.
- Records can be combined into *files*—collections of records all somehow related, that can be accessed by a computer program for some application.
- Files are collected in directories. A single directory usually contains files with related data. Directories can also contain sub-directories with their own files and sub-directories. It is useful to think of a “tree” structure of directories. Each sub-directory is another branch of the tree. The root directory is the “trunk” of the tree.
- The disk on a computer can contain several such “trees” or “partitions. The “disk controller”, a hardware component in a computer, works with the operating system to organizing or manage the files on the disk.
- A single computer can contain several disks. Some computers, file servers, can house many.
- These disks (or portions of them) may be shared across networks with other computers either nearby (Local Area Networks [LAN's]) or over long distances (Wide Area Networks [WAN's]).
- The ability to connect computers and their data together makes possible the establishment of the World Wide Web [WWW], which permits access to some of the data that resides on a computer connected to the network. This data can be made available to anyone who knows its Uniform Resource Locator [URL]. The URL not only gives the internet address of the data, but also conveys some information about what form the data is in.

What is C++'s view of files?

- C++ views a file as a sequential *stream* of bytes. The end of a file is determined anywhere by an *end-of-file* (EOF) marker, or at a specific byte count that the operating system has been told about in a program.



- Files are stored permanently by the operating system, usually on a hard-disk or some other medium (floppies, tape, CD's, DVD's). The operating system is responsible for the maintenance of files.
- When a file is opened by C++ what it really means is that a request for data is sent to the operating system and the data is put into a data *stream*—and C++ only concerns itself with handling the stream, not the hardware storage device. Once data enters the stream its is the responsibility of C++ to interpret the data using formatting instructions.
- This allows most of C++ to be *operating-system independent*.
- The *standard header files* `iostream` (keyboard and screen-related Input/Output [I/O]) and `fstream` (file-related I/O) contains all the stream-file operating system dependencies and has to be rewritten for each new operating system.
- When a C++ program starts with `#include <iostream>` it knows about three streams:
 - `stdin` (standard input) is (usually) the keyboard which can be read using the “`cin >>`” function.
 - `stdout` (standard output) is (usually) the screen which can be written to using the “`cout <<`” function.
 - `stderr` (standard error) is (usually) the screen which is written to using the “`cerr <<`” function. The `stderr` stream is used by C++ to inform the user, via warning and error messages, that data may contain ambiguities or errors. It is often useful to make a distinction between `stdout` and `stderr` even though they output, by default, to the same hardware device.

- It is possible in C++ to *redirect* any or all of the `stdin`, `stdout` or `stderr` files. We will demonstrate how to do this only as required for some specific purpose. It is operating-system dependent.
- It is possible in C++ to define, create and read other files. Just keep in mind that C++ creates these as streams and lets the operating system handle all the other details of file management.

9.4.2 Creating sequential access files and writing to them

What is a sequential access file?

- A sequential access file is one in which the records do not have a fixed, specified length.
- One usually uses such files for writing and reading data that is not changed later and that is read all at once.
- It is the most efficient way of storing files which have variable length records.
- Changing the contents of records or the numbers of records can be done, but not very efficiently.
- Accessing the contents of some specific records can be done, but not very efficiently.
- Nonetheless, it is a useful form for storage of data with variable record length that is all read into memory, manipulated and then all written out.
- Sequential files contain character data that is human readable and transportable (usually) from machine to machine and (usually) from architecture to architecture as long as all the machines understand ASCII character representation. The only essential difference is the interpretation of the end of record. UNIX uses “line feed”, MacOS uses “carriage return” and DOS/Win uses “carriage return” followed by “line feed”.

There are 6 things to remember when writing to files

1. `#include <fstream>` must go in the pre-processor area.
2. Declare an output file object and name it using an identifier in C++. This is done using the `ofstream` keyword. For example,

```
ofstream myOutputFile;
```

associates the identifier `myOutputFile` with a user-declared output file stream. You should think of this as a declaration, much as a usual variable declaration, or string or vector object declaration. It can go anywhere in a program. It is good practice to locate it close to where you use it for the first time.

3. Before writing to the user-define output stream, it must be “opened” using the `.open()` output file object member function. For example,

```
myOutputFile.open("someData.dat");
```

opens the output file stream object called `myOutputFile`. It also connects the output file identifier `myOutputFile` with a file called `someData.dat`. Data going to `myOutputFile` will be written physically in the file `someData.dat` located on the computer’s hard disk (usually). In this example, the file `someData.dat` would reside in the same subdirectory as the load module that was used to start the program assuming that the user started the program with a statement like:

```
> a.out
```

You can also use relative or absolute path names for the parameter of the `.open()` member function. For example, a typical unix system usage would be:

```
myOutputFile.open("~/Private/someData.dat");
```

and a Dos/Win usage:

```
myOutputFile.open("C:\\eng101\\hw\\hw6\\hw6.dat");
```

Can you guess why there are double backslashes above?

Warning! If the parameter of the `.open()` function refers to an existing file, that file will be *destroyed* unless 1) it is write-protected, or 2), you take special care to see if it exists before opening it. (More on this later.)

4. Always check to see that the file was opened successfully using the `.fail()` member function. The `.fail()` member function returns a `true` when the file open failed. For example,

```
if (myOutputFile.fail())
{ //File open fail
    return 1; // Return to calling function with abnormal termination code.
}
```

5. Write to the output stream. Remember it is just a stream, just like `cout`, but with your own identifier. For example,

```
int i = 1, j = 2;
myOutputFile << i << ", " << j << endl;
```

6. When you are finished writing to the file, close the file using `.close` member function. For example,

```
myOutputFile.close();
```

The `.close()` member function disconnects the output stream object, in this case, `myOutputFile`, with the physical file, in this case, `someData.dat`. Although the termination of a program will properly close all files that it opened, it is a good idea to close files as soon as you are finished with them. This frees up some system resources for other programs and in the event of a system crash, closed files are left intact or may be recovered more easily. Files left unclosed during a system crash may be left incomplete. (This is somewhat system dependent and dependent upon how sudden the crash is. Power failures are particularly nasty in this regard.)

Here is an example called `fileOutput.cpp` that illustrates all these concepts, plus a few other concepts.

```
//File: fileOutput.cpp
#include <iostream>
#include <fstream>

using namespace std;

int main(void)
{ ofstream myOutputFile; // Create an output file object

  cout << "What filename do you want to write to: ";
  char fileName[20]; // 1D array of chars
  cin >> fileName;

  myOutputFile.open(fileName); //Connect the stream to the file

  if (myOutputFile.fail())
  { // File could not be opened
    cerr << "File called " << fileName << " could not be opened.\n";
```

```

        return 1; // Return to O/S with abnormal return code
    }
    else
        cout << "File called " << fileName << " was successfully opened.\n";

    bool keepReading = true; // Keep reading if true
    do{
        cout << "Two floats? ";
        float x, y; // Two input floats
        cin >> x >> y;;
        if (cin.eof()) keepReading = false; // End of file <CNTL>-D detected
        else myOutputFile << x << " " << y << endl;
    }while(keepReading);

    cout << "End of input detected\n";
    myOutputFile.close(); //Disconnect the stream from the file

    return 0;
}

```

This code makes use of the `.eof()` member function which tests to see if an end-of-file (EOF) marker has been received in the stream that it is referring to, in this case, `cin`. `.eof()` returns `true` if the EOF has been received and `false` otherwise. Note how this is exploited in the above example. This is how you can control file input with an EOF sentinel.

On UNIX systems, EOF is signaled with a `<RETURN><CNTL>-d`, on MS systems by `<CNTL>-z`.

Note as well how the filename was obtained by inputting a 1D character array from the keyboard. It would seem sensible to use a string class object to accomplish this, but many compilers do not support the use of a string object as an argument to the `.open()` member function.

9.4.3 Reading from sequential access files

- Reading from sequential access files follows the same 6 steps except for the use of the declaration `ifstream` rather than `ofstream` and the direction of the data as indicated by the symbols `<<` for output and `>>` for input! That's all there is to remember.

Here is an example called `fileInput.cpp` that reads the output created by the previous example:

```

//File: fileInput.cpp
#include <iostream>

```

```
#include <fstream>

using namespace std;

int main(void)
{   ifstream myInputFile; // Create an input file object

    cout << "What filename do you want to read from: ";
    char fileName[20]; // 1D array of chars
    cin >> fileName;

    myInputFile.open(fileName); //Connect the stream to the file

    if (myInputFile.fail())
    { // File could not be opened
        cerr << "File called " << fileName << " could not be opened.\n";
        return 1; // Return to O/S with abnormal return code
    }
    else
        cout << "File called " << fileName << " was successfully opened.\n";

    bool keepReading = true; // Keep reading if true
    do{
        float x, y; // Two floats
        myInputFile >> x >> y;;
        if (myInputFile.eof()) keepReading = false; // End of file detected
        else cout << x << " " << y << endl;
    }while(keepReading);

    cout << "End of input detected\n";
    myInputFile.close(); //Disconnect the stream from the file

    return 0;
}
```

Opening files safely when writing

In order to protect against destroying an existing file when opening a file for writing, you should first open it for reading. then test to see if the file open fails. If it fails, it means that that file does not exist and it is safe to write to that filename. If the open does not fail, it means that the file exists. In this case you should prompt the user to see if the file should

be overwritten. Here's an example:

```
//File: fileSafeOutput.cpp
#include <iostream>
#include <fstream>

using namespace std;

int main(void)
{ ifstream testFile; // Create an output file object
  ofstream myOutputFile; // Create an output file object

  cout << "What filename do you want to write to: ";
  char fileName[20]; // 1D array of chars
  cin >> fileName;

  testFile.open(fileName);
  if (!testFile.fail())
  { cout << "File called " << fileName << " already exists.\n";
    cout << "Overwrite (Input 0 for no, any other int to overwrite): ";
    int clobberIt;
    cin >> clobberIt;
    testFile.close();
    if (!clobberIt) return 0;
  }
  myOutputFile.open(fileName); //Connect the stream to the file

  if (myOutputFile.fail())
  { // File could not be opened
    cerr << "File called " << fileName << " could not be opened.\n";
    return 1; // Return to O/S with abnormal return code
  }
  else
    cout << "File called " << fileName << " was successfully opened.\n";

  bool keepReading = true; // Keep reading if true
  do{
    cout << "Two floats? ";
    float x, y; // Two input floats
    cin >> x >> y;;
    if (cin.eof()) keepReading = false; // End of file <CNTL>-D detected
    else myOutputFile << x << " " << y << endl;
```



```

    }while(keepReading);

    cout << "End of input detected\n";
    myOutputFile.close(); //Disconnect the stream from the file

    return 0;
}

```

9.5 Command line arguments

Most unix commands are written in C or C++. For example, if you have ever listed the contents of a directory, you have probably used the command `ls`. If you just type in `ls` at the unix command prompt, you get a list of files that currently reside in your current working directory. If you type in `ls *.cpp` you will get a listing of all the files that end in `.cpp`. If you type `ls -l`, you get a “long” listing of files with information about ownership and permissions for each file. The command is written in C and so you should wonder, “How did the programmer do that?”.

The programmer did it by using command line arguments. Here’s an example code that exploits command line arguments:

```

//File: commandLine.cpp
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{   cout << "argc = " << argc << endl;

    cout << "This program's name is: " << argv[0] << endl;

    for (int i = 1; i < argc; i = i + 1)
    {   cout << "Argument " << i << " is: " << argv[i] << endl;
    }
    return 0;
}

```

Note the new way of declaring the main routine:

```

int main(int argc, char* argv[])
{   .

```

```

    .
    .
}

```

When `main` is called, `argc` contains the number of “tokens” (think of these as words) on the line that ran the program. For example, if you invoked the program with

```
a.out word anotherWord
```

`argc` would be 3.

`argv[]` is an array of addresses. (The “* qualifier on a variable means that the identifier contains one or more addresses.) These addresses contain the addresses of the starts of the tokens of the command line. Resident at these addresses is the start of the one-dimensional character arrays that contain the tokens. In the above example, the first token is the name of the executable file (usually `a.out` in our examples). The second token would be `word` and the third and last token in this example would be `anotherWord`.

here’s a program that can open a file based on a token given to it on the command line:

```
//File: fileOpen.cpp
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char* argv[])
{ ofstream myOutputFile; // Create an output file object

  char fileName[20];

  if (argc == 1)
  { cout << "What filename do you want to write to: ";
    cin >> fileName;
    myOutputFile.open(fileName);
  }
  else if (argc == 2)
  { myOutputFile.open(argv[1]);
  }
  else
  { cerr << "Usage: a.out [filename]\n";
  }
}
```

```
    if (myOutputFile.fail())
    { // File could not be opened
        cerr << "File could not be opened.\n";
        return 1; // Return to O/S with abnormal return code
    }

    myOutputFile.close(); // Disconnect the stream from the file

    return 0;
}
```

If only one token is detected, the user is prompted to input a filename. If two tokens are detected, the second is used as a filename. If there are more than two tokens, the program stops and prints an error message.

9.6 Problems

1. Double factorials

Write a recursive function that returns the double factorial as a function of its int argument N .

- This is how the double factorial is calculated:

$$N!! = N \times (N - 2) \times (N - 4) \cdots (2 \text{ or } 1)$$
 The algorithm stops when a reduction by 2 leaves a number that is equal to 1 or 2.
- You may assume that $N > 2$ and you do not have to concern yourself with the fact that there is a finite (usually 32) number of bits to represent int's.
- Your function must be compatible with the main routine below.

```
#include <iostream>

using namespace std;

//Function goes here

int main(void)
{   cout << "N: ";
    int N;
    cin >> N;

    cout << N << "!! = " << doubleFactorial(N) << endl;

    return 0;
}
```

Here is an example of how it works:

```
red% a.out
N: 5
5!! = 15
```

2. Sorting struct's

This problem uses a `struct` defined as follows:

```
struct Student {int studentNumber; float finalGrade;};
```

where `studentNumber` is a 9-digit student number and `finalGrade` represents the final grade for that student. Write a void function that accepts one called-by-reference `vector<struct Student>`, and sorts the vector in descending order according to the student's grade.

Chapter 10

A Potpourri of Applications

10.1 Finding a zero using the “binary chop”

Let’s say you were looking for a place where a function is zero in some interval. And let’s also suppose that the only thing you knew about the function was that it is monotonic (either always increasing or decreasing) and that it crossed the axis only once.

In fact, you have seen such a function (in disguise) in Assignment 4. You were asked to find the maximum of the function:

$$f(x) = \frac{x \sin(x)}{1+x} .$$

The derivative of this function is:

$$f'(x) = \frac{\sin(x)}{(1+x)^2} + \frac{x \cos(x)}{1+x} .$$

$f'(x)$ has the properties mentioned above for certain ranges of its argument. So, finding the zero of $f'(x)$ is exactly the same as finding the position of the extremum (in this case the maximum) of $f(x)$. The solution you discovered to find the maximum of $f(x)$ involved a laborious search over the function. The algorithm about to be described, call the *binary chop* or *mid-point bisection method*, works much more efficiently, as we shall see with only about 40 iterations. The pseudocode for the algorithm is:

1. Start with an initial guess for an x_0 and an x_1 such that the function is zero for some $x_0 \leq x \leq x_1$. From the properties of the function, we know that the sign of $f(x_0)$ is different from the sign of $f(x_1)$.
2. Evaluate the function at the mid-point, $x_m = (x_0 + x_1)/2$.
3. If $f(x_m) < \epsilon$, where ϵ is the convergence criterion, x_m is the estimated zero of the function and the algorithm can stop. Otherwise,

4. If the sign of $f(x_m)$ is the same as the sign of $f(x_0)$, assign $x_1 = x_m$.
5. Else if the sign of $f(x_m)$ is the same as the sign of $f(x_1)$, assign $x_0 = x_m$.
6. Go back to step 2.

Here is an example of a working code.

```
//file findZero.cpp
#include <iostream>
#include <cstdlib>
#include <cmath>

double myF(double x)
{ static double zero = 0;
  static double f = zero/zero;
  static double Pi = 4*atan(1.0);
  if (zero <= x && Pi >= x) f = (sin(x)/(1+x) + x*cos(x))/(1 + x);
  return f;
}

int sign(double x)
{ if (0 == x) return 0;
  else if (0 > x) return -1;
  else if (0 < x) return 1;
  else
  { cerr << "Stopping since sign can not be determined\n";
    exit(EXIT_FAILURE);
  }
}

int chop(double x0, double x1, double precision, double& xNew)
{ int counter = 0;
  if (0 == sign(myF(x0))) // Zero at the x0 boundary
  { xNew = x0;
    return counter;
  }
  else if (0 == sign(myF(x1))) // Zero at the x1 boundary
  { xNew = x1;
    return counter;
  }
  while(true){ // Search for new root
    counter = counter + 1;
```



```

        xNew = (x0 + x1)/2;
        if (precision > fabs(myF(xNew))) return counter;
        if (sign(myF(xNew)) == sign(myF(x0))) x0 = xNew;
        else x1 = xNew;
    }
}

bool badLimits(double x0, double x1)
{ if (sign(myF(x0)) == sign(myF(x1))) return true;
  return false;
}

int main(void)
{ double x0, x1, x;
  do
  { cout << "Two limits? ";
    cin >> x0 >> x1;
  }while(badLimits(x0,x1));

  double precision;
  cout << "Precision? ";
  cin >> precision;

  cout << "\n\nFinding root for function between "
        << x0 << " and " << x1 << " to precision = "
<< precision << endl;
  cout << "Number of iterations = "
        << chop(x0,x1,precision,x) << endl;
  cout << "Function is near 0 at x = " << x << endl;
  cout << "Value of function there = " << myF(x) << endl;

  return 0;
}

```

Apart from the idea of the algorithm, there are a few new features in this code. One is the output stream `cerr`. `cerr` is a different output stream than `cout` and it is intended to be the stream to which diagnostic or error messages are sent. For example, the `int` `sign()` function above writes to `cerr` when it can not determine the sign of the function.

The other important use of `cerr` is when the user of a program has redirected `cout` to a file via the unix redirection symbols. For example, output to `cout` could be captured in a file names `myOutput` as follows:

```
red% a.out > myOutput
```

However, `cerr` would still print to the screen so that the user would know something had gone wrong. If the error message had been written to `cout`, the user would not know until he or she opened to file.

Another example is:

```
red% a.out < myInput > myOutput
```

which reads `cin` from the file called `myInput` rather than the keyboard. Note that the file names are arbitrary. The names `myOutput` and `myInput` are just examples.

Chapter 11

Programming in MATLAB

11.1 A taste of MATLAB

- Chapter 11: Basic info, logical operators, logic flow control, functions (read now)
- Chapter 12: 2D and 3D graphics
- Chapter 13: Some applications

What is MATLAB?

- An interpreted computer language
- The name of the program that interprets MATLAB language
- Central idea—all data is represented by 2D arrays of doubles

Data types and variables:

- Names of variables? Same as ANSI C++
- Types of variables and declarations?

C++: Many types, MUST declare

MATLAB: 1 type (2-D arrays of doubles) only and don't declare

- A MATLAB demo:

```
>> i = 10;  
>> disp(i)  
    10  
>>
```

- Statements

ANSI C++: Terminated by a semicolon

MATLAB: Terminated by and end-of-line (carriage return)

unless you put a ... at the end of the line to continue

Can also terminate with a ; which suppresses the echo of the statement

- Flow control—while loops

ANSI C++:

```
i = 0;  
while(i<10)  
{ i = i + 1;  
}
```

MATLAB:

```
i = 0;  
while(i<10)  
    i = i + 1;  
end
```

- Flow control—if/else constructs

ANSI C++:

```
if (i <= 10)  
{ j = 0;  
}  
else  
{ k = 0;  
}
```

MATLAB:

```

if (i <= 10)
    j = 0;
else
    k = 0;
end

```

- Example of summing the integers, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

ANSI C++:

```

int main(void)
{   int i = 0, s = 0;
    while(i < 10)
    {   i = i + 1;
        s = s + i;
    }
    cout << s << "\n");
    return 0;
}

```

MATLAB:

```

i = 0;
s = 0;
while(i < 10)
    i = i + 1;
    s = s + i;
end
disp(s)

```

- Scripts and functions
- **Scripts** (or script M-files) are MATLAB statements in a file that the MATLAB interpreter will execute

Functions (or function M-files) are used to implement functions.
Function names are derived from file names.

11.2 Arrays in Matlab

- The idea: 2-D arrays of doubles are the fundamental data type

- Creating arrays in MATLAB

- `zeros`—Creates and initializes arrays in MATLAB and fills the array with “0.0”’s
- `ones`—Creates and initializes arrays in MATLAB and fills the array with “1.0”’s
- Some examples:

```
>> x = ones(10,1)
```

```
x =
```

```
1
1
1
1
1
1
1
1
1
1
1
```

```
>> whos
```

Name	Size	Bytes	Class
x	10x1	80	double array

```
Grand total is 10 elements using 80 bytes
```

```
>> z = zeros(5,3)
```

```
z =
```

```
0    0    0
0    0    0
0    0    0
0    0    0
0    0    0
```

```
>> whos
```

Name	Size	Bytes	Class
x	10x1	80	double array
z	5x3	120	double array

Grand total is 25 elements using 200 bytes

>>

- The colon operator (`:`) gives a list of “integers”
Example:

```
>> clear % This is how to clear stored variables
>> % This is how to make a comment
>> x = 1:10
```

```
x =
     1     2     3     4     5     6     7     8     9    10
```

```
>> whos
      Name      Size      Bytes  Class

      x         1x10         80  double array
```

Grand total is 10 elements using 80 bytes

>>

So `x = 1:N` creates a row with values 1, 2, 3...N

- You can use two colon operators (`a:Δ:b`) to make a list of the form:
 $a, a + \Delta, a + 2\Delta, a + 3\Delta \cdots a + N\Delta$ where $a + N\Delta \leq b < a + (N + 1)\Delta$.

```
>> y = 1:2:10
y =
     1     3     5     7     9
```

Note that the upper limit, 10, is not reached because the next number in the series would overrun it.

- `linspace`—creates a 1D array with uniformly spaced elements.
`x = linspace(a,b,N)`—makes `N` equally spaced points between `a` and `b` inclusive.

(The statement `x = a:(b-a)/(N-1):b` seems to accomplish the same thing BUT round-off errors may cause the end-point `b` to be excluded! DO NOT USE the `x = a:(b-a)/(N-1):b` form!)

An example:

```
>> clear
>> x = linspace(0,1,5)
x =
      0      0.2500      0.5000      0.7500      1.0000
>>
```

– Literal creation:

* Create an array with one row and 3 columns:

```
>> x = [1.0,5.2,9.7]

x =

      1.0000      5.2000      9.7000

>>
```

* Create an array with 2 rows and 3 columns:

```
>> y = [1.0,5.2,9.7;3.4,9.3,10.7]

y =

      1.0000      5.2000      9.7000
      3.4000      9.3000     10.7000

>>
```

- Array indexing

– Simple indexing

$x(i)$ refers to the i 'th element of array x

There is no off-by-one nonsense!

$x = \text{ones}(1,10) \Rightarrow x = [1,1,1,1,1,1,1,1,1,1]$

$x(1)$ is the first element of array x

$x(10)$ is the last element of array x

Array indices go from 1 to the number of rows/columns

```
>> x = [3,5;7,9]
```

```
x =

      3      5
      7      9
```


$x(1,1) = 3 \implies$ row 1, column 1
 $x(2,1) = 7 \implies$ row 2, column 1
 $x(1,2) = 5 \implies$ row 1, column 2
 $x(2,2) = 9 \implies$ row 2, column 2

– Slice indexing

The “.” symbol as an array index means “the whole” row or “the whole” column.

Example,

```
>> x = [1,2;3,4;5,6]
```

x =

1	2
3	4
5	6

```
>> y = x(:,2)
```

y =

2
4
6

```
>> z = x(1,:)
```

z =

1	2

Partial slices can be extracted by specifying ranges. Example,

```
>> u = x(1:2,1)
```

u =

1
3

```
>> v = x(2:3,:)
```

v =

3	4
5	6

- Saving an array to a file

`sindat` is a 2 row, 1000 column array created and saved this way:

```
%ch11e1.m
sindat = zeros(2,1000);
sindat(1,:) = linspace(0,2*pi,1000);
sindat(2,:) = sin(sindat(1,:)); % Note: array operation!
save sindat -ascii -double
```

- Reading an array from a file

The following reads `sindat` in and plots it.

```
%ch11e2.m
clear
load sindat
x = sindat(1,:);
y = sindat(2,:);
plot(x,y) % Simple plotting function
pause % Pauses until any keystroke is given
close % Closes the figure
plot(x,y.^2) % Note: ".^2" means element by element squaring operation
pause
close
plot(y,x)
pause
close
```

Summary

- **zeros**—Create and initialize an array and fill the array with “0.0”’s
- **ones**—Create and initialize an array and fill the array with “1.0”’s
- **clear**—Clear variables from memory
- **whos**—List current variables, long form
- **linspace**—Create a 1D array with uniformly spaced elements

- Literal creation of arrays
- Simple array indexing
- Slice indexing, full and partial slices
- Writing/reading arrays to/from files.

11.3 Loops

11.3.1 The for loop

The general syntax of a for loop is:

```
for i = array
    % Some loop using i where i can not be re-assigned
end
```

On each successive pass through the loop, *i* is set to the next column of *array*. On the first pass through the loop, *i* is `array(:,1)`, on the second pass *i* is `array(:,2)`, *etc*

This can seem a little confusing unless we remember that the fundamental data type is an array of doubles.

Here are some simple examples:

```
sum = 0;
for i = 1:10
    sum = sum + i;
end
disp(sum);
```

Here's another way of doing the same thing:

```
i = 1:10;
sum = 0;
for j = i
    sum = sum + j;
end
disp(sum);
```

Here's yet another way of doing the same thing:

```
i = 1:10;
sum = 0;
for j = 1:length(i) % length(array) is a MATLAB function that gives the length
    sum = sum + j;
end
disp(sum);
```

Here's a weird way of doing the same thing:

```
i = [1,2,3,4,5;6,7,8,9,10];
sum = 0;
for j = i % j is set to the columns of i
    sum = sum + j(1) + j(2);
end
disp(sum);
```

Why does this work?

The number of columns of *i* is 5 and so, the loop is executed 5 times. But on each iteration, *y* is set to a column of *x*, in this case a column vector with two values which are summed with the statement `sum = sum + j(1) + j(2);`. Weird! But it works and illustrates the more general form of the looping index.

Some useful functions

`size` gives the number of rows and the number of columns as a two-element row vector. For example (using the *i* array above):

```
>> size(i)
ans =
     2     5
```

If you want to “capture” both the number of rows and columns, use the following:

```
>> [nRows,nColumns]=size(i)
nRows =
     2
nColumns =
     5
```

`length` gives the maximum of the number of rows and the number of columns. For example (using the *i* array above):

```
>> length(i)
ans =
     5
```

error('string') displays the 'string' and causes an error exit from an M-file to the keyboard. If the string is empty, no action is taken.

Here's an example of an integration script using **length** and **error**.

```
% Area under f as a MATLAB script
% Assumes that f and x already exist

if (length(x) ~= length(f))
    error(' x and f are different lengths. Stopping.')
end

sum = 0;
for j = 1:(length(f)-1)
    sum = sum + 0.5*(f(j) + f(j + 1))*(x(j+1) - x(j));
end
disp(sum);
```

Here's a 2D plotting example:

```
N = 100;
f = zeros(1,N);
x = linspace(-5*pi,5*pi,N);
for i = 1:N
    if (x(i) == 0)
        f(i) = 1;
    else
        f(i) = sin(x(i))/x(i);
    end
end
plot(x,f);
```

Here's a 3D plotting example (using the above f):

```
g = zeros(N,N);
for i = 1:N
    for j = 1:N
        g(i,j) = f(i)*f(j);
```

```
    end
end
surf(x,x,g);
```

11.3.2 The while loop

The general syntax of a while loop is:

```
while expression
    % Some loop that is executed when expression is true
end
```

Here's an example

```
change = 1;
while change >= 0.001
    change = change/2;
end
disp(change);
```

That's about all you need to know about while loops.

11.4 The if/else construct

The general syntax of an if/else if/else construct is:

```
if expression1
    % Executed when expression1 is true
elseif expression2
    % Executed when expression1 is false and expression 2 is true
elseif expression3
    % Executed when expression1&2 are false and expression 3 is true
.
.
.
elseif expressionN
    % Executed when expression1...N-1 are false and expression N is true
else
```

```
% Executed when expression1...N are false
end
```

Valid syntax requires the `if` and the `end`, zero or more `elseif`'s and zero or one `else`. The design is very much like C and therefore does not really need to be discussed further. The use of indenting make the logical flow a little more comprehensible.

Here's an example that makes a decision based on a random number:

```
x = rand;
if x < 1/3
    disp('x is less than a third');
elseif x < 2/3
    disp('x is greater than or equal to a third but less than two thirds');
else
    disp('x is greater than or equal to a two thirds');
end
```

New function:

rand—provides an array of uniformly distributed random numbers between 0 and 1.

11.5 Some Matlab terminology

scalar A scalar in MATLAB is a 1×1 array. For example:

```
>> whos
  Name      Size      Bytes  Class
  s         1x1         8   double array
Grand total is 1 elements using 8 bytes
```

row vector A row vector of length M in MATLAB is a $1 \times M$ array. For example:

```
>> rv = ones(1,8);
>> whos
  Name      Size      Bytes  Class
  rv        1x8        64   double array
  s         1x1         8   double array
Grand total is 9 elements using 72 bytes
```

column vector A column vector of length N in MATLAB is an $N \times 1$ array. For example:

```
>> cv = zeros(10,1);
>> whos
  Name      Size      Bytes  Class
  cv        10x1         80  double array
  rv         1x8         64  double array
  s          1x1          8  double array
Grand total is 19 elements using 152 bytes
```

matrix An N by M matrix in MATLAB is an $N \times M$ array. For example:

```
>> a = rand(3,4);
>> whos
  Name      Size      Bytes  Class
  a          3x4         96  double array
  cv        10x1         80  double array
  rv         1x8         64  double array
  s          1x1          8  double array
Grand total is 31 elements using 248 bytes
>> a
a =
    0.7382    0.9355    0.8936    0.8132
    0.1763    0.9169    0.0579    0.0099
    0.4057    0.4103    0.3529    0.1389
```

11.6 Operators in MATLAB

11.6.1 Math operators in Matlab

Operation	Symbol	Example	Precedence
Inner expression	(,)	(1+2)/3	Highest
Exponentiation	^	2^3	Medium high
Multiplication, $x \times y$	*	x*y	Moderate
Division, $x/y, y \backslash x$	/, \	y/x = x\y	Moderate
Addition, $x + y$	+	x + y	Medium low
Subtraction, $x - y$	-	x - y	Medium low
Assignment	=	y = x	Lowest

These are all 2-D matrix operations!

If the operands are scalars (a 1×1 array) the functionality is similar to that of C++.

Here are some examples:


```

>> a = 2*ones(2)
a =
     2     2
     2     2
>> b = a/4
b =
    0.5000    0.5000
    0.5000    0.5000
>> a*b
ans =
     2     2
     2     2

>> a+b
ans =
    2.5000    2.5000
    2.5000    2.5000

```

11.7 Pointwise operators in MATLAB

Operation	Symbol	Example	Expansion
Pointwise exponentiation	<code>.^</code>	<code>x.^y</code>	$x(i,j)^y(i,j)$
Pointwise multiplication	<code>.*</code>	<code>x.*y</code>	$x(i,j)*y(i,j)$
Pointwise division	<code>./</code> , <code>.\</code>	<code>y./x = x.\y</code>	$x(i,j)/y(i,j)$

Here are some examples:

```

>> x = 2*ones(2)
x =
     2     2
     2     2
>> y = [0,1;2,3]
y =
     0     1
     2     3
>> x.*y
ans =
     0     2
     4     6
>> x.^y

```

```

ans =
     1     2
     4     8
>> x.\y
ans =
         0    0.5000
    1.0000    1.5000

```

11.7.1 Logical Operators in Matlab

Symbol	Relational operation
< , <=	less than, less than or equal to
> , >=	greater than, greater than or equal to
==	equivalent to
~=	not equivalent to
& ,	AND , OR
~	NOT

Note that these operators work on arrays and can return arrays. The relational operators operating on arrays is not discussed in this course; we will be using them as if they always operate on scalars. Therefore, in this sense, they behave exactly as the analogous operators in C++,

There are some differences in notation with C++ (`~=` *vs.* `!=`, `&` *vs.* `&&`, `|` *vs.* `||`, `~` *vs.* `!`) However, the biggest difference with respect to C++ is the difference in the order of precedence of the operators. Note that even some textbooks are confused on this issue. Here is the current state of the order of precedence of the MATLAB operators. Note that for MATLAB Version 5.3 and lower, the precedence table is different!

Operation	Symbol	Precedence
Inner expression	()	0 (Highest)
Transpose, Exponentiation	^ .^ '	1
Unary plus and negation	+ - ~	2
Multiplication, division	.* .\ ./ * \ /	3
Binary addition, subtraction	+ -	4
Colon operator	:	5
Relational logical operators	< <= > >= == ~=	6
Logical AND	&	7
Logical OR		8
Assignment	=	9 (Lowest)

For more complete information, issue `help precedence`. This is what you get:

```
PRECEDENCE    Operator Precedence in MATLAB
```

MATLAB has the following precedence for the built-in operators when evaluating expressions (from highest to lowest):

1. transpose (.'), power (.^), complex conjugate
transpose ('), matrix power (^)
2. unary plus (+), unary minus (-), logical negation (~)
3. multiplication (.*), right division (./), left
division (.\), matrix multiplication (*), matrix right
division (/), matrix left division (\)
4. addition (+), subtraction (-)
5. colon operator (:))
6. less than (<), less than or equal to (<=), greater than
(>), greater than or equal to (>=), equal to (==), not
equal to (~=)
7. logical AND (&)
8. logical OR (|)

Starting in MATLAB 6.0, the precedence of the logical AND (&) and logical OR (|) operators now obeys the standard relationship (AND being higher precedence than OR) and the formal rules of boolean algebra as implemented in most other programming languages, as well as Simulink and Stateflow.

Previously MATLAB would incorrectly treat the expression:

```
y = a&b | c&d
```

as:

```
y = (((a&b) |c) &d);
```

It now correctly treats it as:

```
y = (a&b) | (c&d);
```

The only case where the new precedence will impact the result obtained is when `|` appears before `&` within the same expression (without parentheses). For example:

```
y = 1 | x & 0;
```

In MATLAB 5.3 and earlier, this statement would yield 0, being evaluated as:

```
y = (1 | x) & 0;
```

In MATLAB 6.0 and beyond, this expression yields a 1 as the result, being evaluated as:

```
y = 1 | (x & 0);
```

We strongly recommend that you add parentheses to all expressions of this form to avoid any potential problems with the interpretation of your code between different versions of MATLAB.

NOTE: A feature has been provided to allow the system to produce an error for any expression that would change behavior from the old to the new interpretation:

```
feature('OrAndError', 0) % warning for any expression that
                          % changed behavior (default)
feature('OrAndError', 1) % error for any expression that
                          % changed behavior
```

11.8 M-files

11.8.1 Script M-files

MATLAB commands may be stored in a file and executed. These files are called “script M-files”. Any commands that you can enter in the MATLAB command window can be entered in a file with a filename that is arbitrary except that it should contain the extension `.m`, for example `myMatlabScript.m`. Script M-files may be created with your favorite editor

or with MATLAB's own editor.

If you type the name, say `myMatlabScript` in the MATLAB command window, the following order of execution is followed:

- MATLAB checks to see if `myMatlabScript` is a variable in the current workspace, the environment of user-defined and pre-defined variables; if not
- MATLAB checks to see if `myMatlabScript` is built-in function; if not
- MATLAB checks to see if `myMatlabScript.m` is a file in the current directory; if not
- MATLAB checks to see if `myMatlabScript.m` exists anywhere along the MATLAB search path;

Some useful functions for use with MATLAB script M-files:

`disp(variable)` displays the value of the variable, *e.g.*.

```
>> n = 10; disp(n)
      10
>>
```

`echo` toggles echoing.

`echo off` turns it off (default value)

`echo on` turns it on and every command executed by a script M-file is written to the screen (useful for debugging purposes)

`echo` switches (toggles) between `echo off` and `echo on`.

`input` prompt the user for input.

```
>> x = input('Input a value for x: ')
Input a value for x: 10
x =
      10
>>
```

`keyboard` gives control to the keyboard. Return to the script M-file by typing `return` at the command window prompt, (`>>`).

`pause` Pause. Continue after user presses any keyboard key.

`pause(x)` Pause for x seconds, then continue.

```
>> for i = 10:-1:1
disp(i)
pause(1)
end
    10
     9
     8
     7
     6
     5
     4
     3
     2
     1
>>
```

waitforbuttonpress Pause. Continue after user presses any keyboard key or mouse key.

sprintf Write formatted data to string.

`S = sprintf(FORMAT,A,...)`

formats the data in the variable `A` (and in any additional arguments), under control of the specified `FORMAT` string, and returns it in the MATLAB string variable `S`. There is a similar function called `fprintf` that writes the MATLAB string variable to a file. (Do a `help fprintf` for more information.)

`FORMAT` is a string containing C++ language conversion specifications. Conversion specifications involve the character `%`, optional flags, optional width and precision fields, optional subtype specifier, and conversion characters `d`, `i`, `o`, `u`, `x`, `X`, `f`, `e`, `E`, `g`, `G`, `c`, and `s`. See the Language Reference Guide or a C++ manual for complete details. The `%g` conversion specifier produces the nicest output for floating point numbers. See the example below.

The special formats `\n`, `\r`, `\t`, `\b`, `\f` can be used to produce linefeed, carriage return, tab, backspace, and formfeed characters respectively. Use `\\` to produce a backslash character and `%%` to produce the percent character.

Here's an example of a script M-file called `russianRoulette.m`:

```
%File: russianRoulette.m
clear
echo off
```

```
prob = 1/6;
```

```
disp('With each spin you could win 1$')
```

```

disp('If you lose, you lose everything!')
disp('It''s best to quit while you''re ahead.')
disp('Press any key to start.')
disp('Good luck')
disp(' ')

pause

money = 0;
repeat = input('Play? (0 = no, 1 = yes) :');

while(repeat)

    disp('Spin....')
    pause(1)
    if (rand < prob)
        disp('BANG! You LOSE!')
        return
    else
        disp('Click! You WIN!')
        money = money + 1;
        disp(sprintf('Your gain is %g',money))
    end
    repeat = input('Repeat? (0 = no, 1 = yes) :');

end

disp(sprintf('Your gain was %g',money))
disp('End of Game')
disp(' ')

```

Here's a typical play of the game:

```

>> russianRoulette
With each spin you could win $1.
If you lose, you lose everything!
It's best to quit while you're ahead.
Press any key or mouse button to start.
Good luck!

Play? (0 = no, 1 = yes) :1
Spin....

```

```
Click! You WIN!
Your gain is $1
Repeat? (0 = no, 1 = yes) :1
Spin....
Click! You WIN!
Your gain is $2
Repeat? (0 = no, 1 = yes) :1
Spin....
Click! You WIN!
Your gain is $3
Repeat? (0 = no, 1 = yes) :1
Spin....
Click! You WIN!
Your gain is $4
Repeat? (0 = no, 1 = yes) :1
Spin....
Click! You WIN!
Your gain is $5
Repeat? (0 = no, 1 = yes) :1
Spin....
Click! You WIN!
Your gain is $6
Repeat? (0 = no, 1 = yes) :1
Spin....
Click! You WIN!
Your gain is $7
Repeat? (0 = no, 1 = yes) :1
Spin....
BANG! You LOSE!
```

11.8.2 Function M-files

Compartmentalizing your code into functional subunits is...yada, yada, yada...

It's the same sermon you got in C++! However, MATLAB makes it easy to design functions for repeated use in many applications.

The general syntax of a function call is:

```
[out1,out2,...,outN] = functionName(in1,in2,inM)
```

and the general syntax of the function definition is to have a separate file called `functionName.m` that has the following form:


```

function [out1,out2,...,outN] = functionName(in1,in2,...inM)
%FUNCTIONNAME A brief one line description
% A more general description of functionName with a description of the inputs and
% outputs and perhaps an example. All this text must fit into contiguous comment
% lines.
%
% See also related list of functions
% After this line starts MATLAB executable functions, important examples below
.
.
.
error('This is an error') % Can use this to return control to the command
                        % windows if an error is detected
.
.
.
warning('This is a warning') % Can use this to issue warnings
.
.
.
return; % Control reverts to calling M-file if a return is seen
.
.
.
% Last line in functionName, control reverts to calling M-file

```

Some general features of functions:

- **functionName** is an identifier, identical to that of variables
- Variables can only be output in the output “string” [out1,out2,...,outN]. There can be zero or one output (in which case the square brackets are not needed).
- There can be zero, one or more inputs. If you change the value of an input variable it is hidden from the calling M-file. If you do not change the value of an input, MATLAB will not make copies for the function’s workspace for the sake of efficiency. If you do change the value of an input, MATLAB will make copies for the function’s workspace and the changes will be hidden from the calling M-file.
- The command window and the functions have their own workspaces, places where variables are stored independently, much like C++’s stack frame handling.
- The first line of the function M-file is called the function declaration line. The function name must agree with the file name. If it does not, MATLAB will use the file name.

This line also serves a similar role as the function prototype in C++, checking the compatibility of the function definition with the function call. (MATLAB is considerably looser, however, than ANSI C++).

- If the second line of the file is a comment, it is used by MATLAB to look for functions and functionality using the `lookfor` MATLAB function.
- Comment lines following contiguously are copied to the screen when you type `help functionName`
- You can have more than one function in a function M-file. However, only the first (the primary function) is available to other M-files and the command window. The others are called subfunctions and cannot be used by other functions, scripts or the command window.

Here is an example called `quadRoots.m`:

```
function [root1,root2] = quadRoots(A,B,C)
%QUADROOTS quadRoots returns the two real roots of A*x^2 + B*x + C = 0
%
% The two roots are root1 = (-B + sqrt(B^2 -4*A*C))/(2*A)
%                      root2 = (-B - sqrt(B^2 -4*A*C))/(2*A)
%
% An error is detected if 4*A*C > B^2 (imaginary roots)
% A warning is issued if 4*A*C = B^2 (identical roots)
%
% The blame? BLIF <bielajew@umich.edu>
% This version 1.1, last modified November 27, 2000
% Warranty? None. Take it or leave it.
% Copyright: Regents of the University of Michigan

if 4*A*C > B^2
    error('Roots would be imaginary. Get real!')
elseif 4*A*C == B^2
    warning('Roots will be the same, degenerate')
end

root1 = (-B + sqrt(B^2 -4*A*C))/(2*A);
root2 = (-B - sqrt(B^2 -4*A*C))/(2*A);

return % This is optional
```

11.9 Problems

1. Check the appropriate circle for the following MATLAB statements.

MATLAB Statement	Valid	Invalid
<code>y=3^2;</code>	☐	☐
<code>if(23==x)</code>	☐	☐
<code>23=x;</code>	☐	☐
<code>for i=-123:234</code>	☐	☐
<code>while(~(~(x<3))))</code>	☐	☐
<code>a(1:10)=1;</code>	☐	☐
<code>b=[1 2 3; 4 5;6 7];</code>	☐	☐

2. Consider the following snippet of code

```
for p = 1:10
    for g = 1:10
        x(g,p) = g*p;
    end
end
```

This piece of code:

- ☐ Fills the entire first row of the array, then moves on to the next row, and so on to fill the array.
 - ☐ Fills the entire first column of the array, then moves on the next column, and so on to fill the array.
3. For each of the following tasks, check the appropriate box to state which construct is the most natural way of implementing that task.

	Counter loop	Sentinel loop	if/else
Initializing the elements of an array of known dimensions	☐	☐	☐
Responding to a user's selection from a menu	☐	☐	☐
Carrying out an iterative process until some measure of convergence is reached	☐	☐	☐
Dealing a hand of cards from a deck	☐	☐	☐
Determining a student's letter grade for this final from his or her numerical score	☐	☐	☐
Computing the average score on this final using the scores of a known number of students	☐	☐	☐

4. Consider the following small script M-file and function M-file.

```
x = 4;
y = [2 3;6 7];
a(1,1) = dumbFunction(x,y(2,1));

function r = dumbFunction(a,b)
r = b/a;
return;
```

Now, complete the following sentence by filling in the blanks.

The numerical values _____ and _____ are passed into the function dumbFunction. The function returns the numerical value _____, which is then stored in the variable _____.

5. **What is an algorithm?**

List and describe the three capabilities that are needed to construct an algorithm.

6. **MATLAB Function M-file design**

Describe the roles of:

- (a) the first line of a MATLAB function M-file,
- (b) the second line of a MATLAB function M-file (assuming it starts with a “%”),
- (c) the third and subsequent contiguous comment lines in a MATLAB function M-file (assuming the second line starts with a “%”).

How do the MATLAB lookfor and help functions interact with these lines?

7. **MATLAB Function M-files**

Write a MATLAB Function M-file that provides the indicated response to the `lookfor` and `help` commands. A human typed the stuff to the right of a `>>` prompt, the computer typed the rest.

```
>> lookfor helpMe
    HELPME Help! I need somebody.
>> help helpMe

    HELPME Help! I need somebody.
    Help! Not just anybody!

>>
```

8. True or false?

(a) The 3 statements

```
clear all; x = 1.0; i = &x;
```

are ALL valid MATLAB statements.

(b) The 3 statements

```
clear all; x = 1; i = i & x;
```

are ALL valid MATLAB statements.

(c) The set of statements

```
clear all; a = 2*ones(2,2); b = a/4; c = a*b; disp(c)
```

```
prints    1 1    to the screen.
          1 1
```

(d) The set of statements (note use of the transpose operator " ' ")

```
clear all; disp(linspace(0,1,5)')
```

```
prints    0 0.2500 0.5000 0.7500 1.0000    to the screen.
```

(e) The statement

```
a = [0,1,2; 2,3; 4,5,];
```

is a valid MATLAB statement.

9. Multiple choice

Circle all the correct answer(s). There may more than one correct answer but there will be at least one choice that is correct.

(a) The following MATLAB code:

```
clear all; N = 5; a = zeros(N);
for i = (2:1:N)
    a(i,i-1) = -1;
    if i ~= N a(i,i+1) = 1; end
end
disp(a)
```

is supposed to print to the screen:

0	1	0	0	0
-1	0	1	0	0
0	-1	0	1	0
0	0	-1	0	1
0	0	0	-1	0

but it does not. The error is

- There is an syntax error in the line: `if i ~= N a(i,i+1) = 1; end`
 - After `a = zeros(5);`, nothing is changed in the first row.
 - There is an syntax error in the line: `for i = (2:1:N)`
 - There is a conceptual error in the statement: `a(i,i-1) = -1;`
 - None of the above, there is a syntax error somewhere else.
- (b) The following code is supposed to sum the squares of the matrix elements of a random 10 by 10 array and then divide the sum by the number of elements. It does not work as intended.

```
clear all;
N = 10;
s = rand(N);
s2 = s*s;
for i = 1:N for j = 1:N sum = sum + s2(i,j); end; end
disp(sum/N^2)
```

- N is not defined properly.
- There is a syntax error in the line that starts with `for`.
- `sum` should be initialized first, before the `for` loops.
- There is a syntax error in the statement `disp(sum/N^2)`.
- There is a conceptual error in the statement `s2 = s*s;`.
- There is no error. The code works as intended.
- There is a conceptual error in the statement `s = rand(N);`.

10. Slice indexing

The following code is written with 4 for loops.

```
N = 100;
a = zeros(N);
for i = 1:N
    for j = 1:N
        a(i,j) = 1;
    end
end
end
```

```

for i = N/4:3*N/4
    for j = N/4:3*N/4
        a(i,j) = 0;
    end
end

```

- (a) Write it without the use of for or while loops.
- (b) If the variable N was set to 4 instead of 100, what would the array look like?

11. Think like a computer

Write the output that the following MATLAB code produces.

```

clear all;
z = ones(6);
z(3:4,:) = 0;
z(:,3:4) = 0;
disp(z);

```

12. Think like a computer again

Consider the following script M-file.

```

sign = 1;
for i = 1:5
    x(i) = i - 3;
    y(i) = sign*i;
    sign = -sign;
end
plot(x,y);

```

Draw the plot that would result from executing this script.

13. Think like a computer again

Write the output that the following MATLAB code produces.

```

clear all;
z = -ones(7); % Don't miss the minus sign!
z(2:3,2:3) = 1; z(2:3,5:6) = 4;
z(5:6,2:3) = 2; z(5:6,5:6) = 3;
z(4,4) = 0;
disp(z);

```

14. Think like a computer again

Consider the following script M-file.

```
a = [1 2];
b = [1 2 3];
c = 1;

for ai = a
    for bi = b
        fprintf('a = %f, b = %f\n',ai,bi)
        if (bi^2 - 4*ai.*c < 0)
            fprintf('No real solution\n\n')
        elseif (bi^2 - 4*ai.*c == 0)
            fprintf('Single root = %f\n\n',-bi/(2*ai))
        else
            fprintf('Root1 = %f\n',(-bi - sqrt(bi^2 - 4*ai.*c))/(2*ai))
            fprintf('Root2 = %f\n\n',(-bi + sqrt(bi^2 - 4*ai.*c))/(2*ai))
        end
    end
end
end
```

What is the output that would result from executing this script?

15. The purpose of the script M-file below is to:

- prompt the user for a number between 0 and 100
- generate a row-vector of 50 random numbers between 0 and 100
- invoke a function called **nearest**
- print out the results returned from the function

The purpose of the function M-file below is to :

- take in a row-vector and a scalar value
- find the entry in the row-vector that is closest to the scalar value
- return the entry that was found, and its index in the array

There are four bugs in the code, at lines 5, 7, 17 and 18. Find them and fix them.

```
% Script M-file testNearest.m
%
x = input('pick a number between 0 and 100: ');
```



```

b=rand(1,50)*100;
[ind,y] = nearest(b(50),x);
fprintf('The nearest value to %f is b(%i)=%f\n',x,ind,b);

% Function M-file nearest.m

function [ind,y] = nearest(a,x)

oldDiff = 1000;
[rows,columns] = size(a);

for i = 1:columns
    newDiff = abs(a(i) - x);
    if(newDiff < oldDiff)
        y = a(i);
        index = i;
        newDiff = oldDiff;
    end
end

return

```

16. Plot like a computer

The code below produces a 2D plot. Indicate what it would look like in the box below.

```

clear all;
x = [0,-0.59, 0.95,-0.95, 0.59, 0];
y = [1,-0.81, 0.31, 0.31, -0.81, 1];
plot(x,y)

```

17. More code debugging

In the following code I want to plot $f(x) = \frac{\sin(x)}{x}$ for $0 \leq x \leq 5\pi$ by making two row vectors, one for x and the other for $f(x)$. The code does not generate any MATLAB errors and produces a plot. However, the plot does not look anything like what I expected. Indicate what needs to be done to fix it by writing a correct version of the code in the box below. The code below has conceptual problems on lines 3 and 5. Take care that your code does not have a “divide-by-zero” problem. (*Hint: $f(x) = 1$ when $x = 0$.*)

```

clear all;                %                               Line 001
N = 1000;                 % Array size                   Line 002
f = zeros(1000);          % Allocate memory for f      Line 003
x = linspace(0,5*pi,1000); % Make a linspace for x    Line 004
f = sin(x)/x;             % Calculate f(x)                Line 005
plot(x,f)                 % Plot it                      Line 006

```

18. Temperature conversion

Write a code that outputs a table of degrees Fahrenheit and degrees Centigrade starting with -40 degrees Fahrenheit, ending with 140 degrees Fahrenheit in increments of 1 degree Fahrenheit. The conversion from degrees Fahrenheit to degrees Centigrade is:

$$C = (5/9) * (F - 32)$$

and must be coded as a separate MATLAB function. The output should look like:

```

-40,-40.0
-39,-39.4
-38,-38.9
.
.
.
138,58.9
139,59.4
140,60.0

```

Hint: The format statement that outputs one line of the table is
`fprintf('%3.0f,%4.1f\n',F(i),C(i)).`

19. Array manipulation

Create a complete set of MATLAB instructions that makes a 1000 by 1000 matrix called “a” that is zero everywhere except for ones in the first column and last column and ones in the first row and last row. There are also ones on the main diagonal of the matrix, ($a(i,i) = 1$) for all i . For full credit you can not use more than one for or while loop. Otherwise, one point will be deducted for each use of a for or a while loop. Do it the MATLAB way! There are five extra credit points on this problem for those who remember and use correctly (from reading the book, or from attending the last lecture) the command for making an array with ones on the main diagonal and zeros everywhere else.

20. Matrix rotation and flipping

Write 4 different functions that:

- Rotates a matrix in a clockwise fashion

- Rotates a matrix in a counterclockwise fashion
- Flips a matrix about its horizontal mid-line
- Flips a matrix about its vertical mid-line

Hints and instructions.

- Recall that `[Rows,Columns] = size(a)` can be used to get the number of rows and columns of the matrix `a`
- Your functions have to work on arrays of any size
- Your functions do not have to provide output to the `help` and `lookfor` functions.

Here's an example:

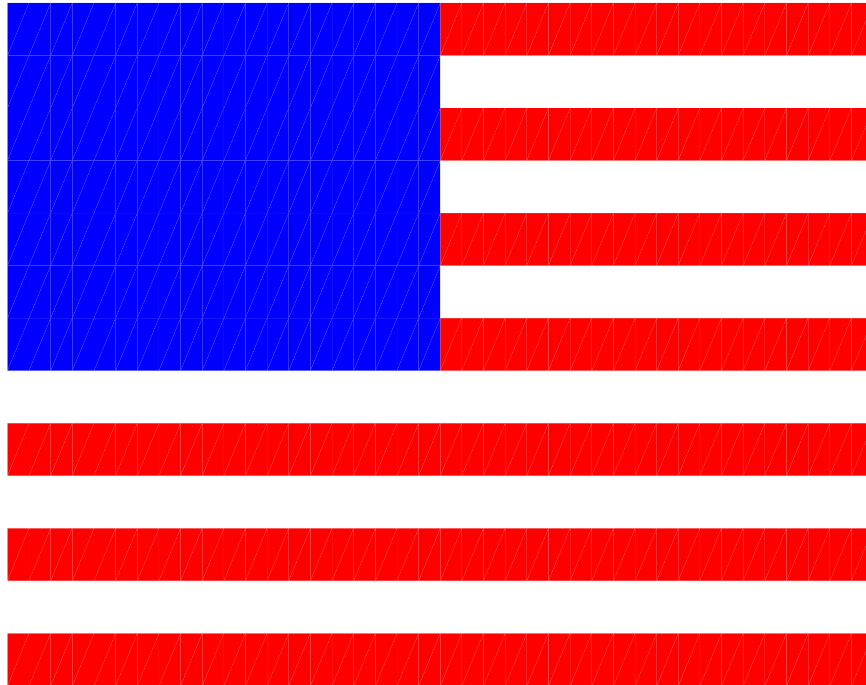
```
>> a = rand(2,3)
a =
    0.9218    0.1763    0.9355
    0.7382    0.4057    0.9169
>> disp(rotateClockWise(a))
    0.7382    0.9218
    0.4057    0.1763
    0.9169    0.9355
>> disp(rotateCounterClockWise(a))
    0.9355    0.9169
    0.1763    0.4057
    0.9218    0.7382
>> disp(flipVertical(a))
    0.9355    0.1763    0.9218
    0.9169    0.4057    0.7382
>> disp(flipHorizontal(a))
    0.7382    0.4057    0.9169
    0.9218    0.1763    0.9355
```

21. Stars and Stripes forever

At the end of this problem description, parts of a script M-file are given. This program may be used to draw the US flag. The US flag has 13 horizontal rows, alternating with red and white. Row 1 is red, row 2 is white, and so on, for 7 red rows and 6 white ones. A solid blue rectangle occupies the upper left corner, from rows 7 to 13 and it is half the width of the flag. There are 50 white stars arranged in the blue patch.

Here's how you complete the code to finish the flag program.

- Put in the code that will make the stripes and the blue patch. (Red value 0.15, blue has value 0.4, and white has value 0.0.) Plotted as indicated it should look like:

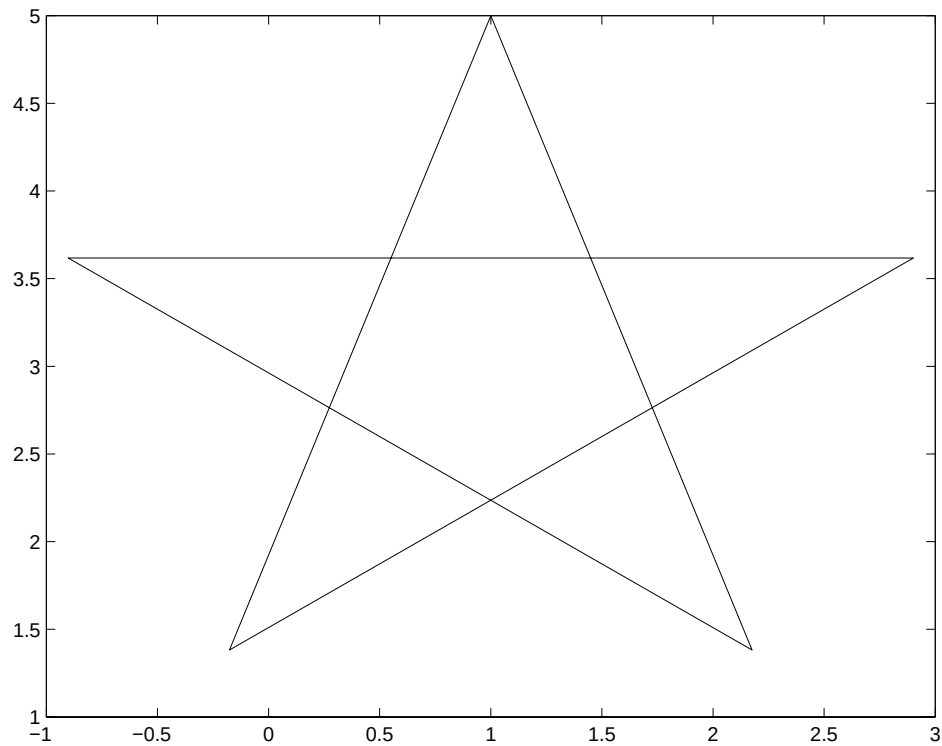


- Read through the rest of the MATLAB code and understand what it is trying to do. The bottom part is for placing the stars.
- Write the function `star(x,y,r)`. `x` and `y` are the points that define the center of the star, and `r` is its radius. Notice that the 5 points on a star are all located on a circle centered at `x` and `y` with radius `r`. If you drew lines from the center to the points of the star, those lines would be separated by 72 degrees, ($72 = 360/5$). To draw the star, imagine that the top point of the star is numbered 1, the one to its right 2, and so on. You can make a star easily by drawing a line from point 1, to 4, to 2, to 5, to 3 and back to 1. Here's an example of `star(1,3,2)`.
- You might find it useful to use the `sin()` and `cos()` functions. The argument to these functions should be expressed in radians. In MATLAB, the conversion from degrees to radians is given by: `rads = pi*a/180`, where `a` is an angle in degrees and `rads` is the same angle expressed in radians.
- You should use the `plot()` function using white solid lines. Here are the details:

PLOT Linear plot. `PLOT(X,Y)` plots vector `Y` versus vector `X`.

Various line types, plot symbols and colors may be obtained with `PLOT(X,Y,S)` where `S` is a character string made from one element from any or all the following 3 columns:

<code>b</code>	<code>blue</code>	<code>.</code>	<code>point</code>	<code>-</code>	<code>solid</code>



g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed

.
.
.

For example, `PLOT(X,Y,'c+:')` plots a cyan dotted line with a plus

- Your function has to respond sensibly to the `lookfor` and `help` functions.

```
%FLAG flag Solution to Stars and Stripes forever
clear all
%Code to make the flag without stars below
```

```

%Show the flag without stars using pcolor
show = flag;
show(end+1,1) = 0; %Hidden point, fixes the lower bound of the colormap
show(end,end+1) = 1; %Hidden point, fixes the upper bound of the colormap
pcolor(1:41,1:14,show)
colormap vga
shading flat
axis off

hold on

%This code prompts the user to place the stars...
putStar = 1;
while putStar
    putStar = input('Place a star? (0 = no, ~0 = yes)');
    if putStar
        response = input('Specify [x, y, size] ');
        x = response(1); y = response(2); r = response(3);
        star(x,y,r)
    end
end

```

22. Turn pseudo-code into real code

Write a MATLAB code to find the zero of a function. Your code will be made up of three M-files:

- a script M-file to drive the process
- a function M-file that finds zeros using a Newton procedure (explained below)
- a function M-file that contains the function for which you are finding the zero.

Pseudo-code for the three pieces is given below. Construct the MATLAB code.

Script M-file

- Prompt the user for an initial guess for the x value at which the zero occurs.
- Call the zero-finding function, with the initial guess as the input argument, and the final answer and number of steps it took to find the final answer as the output arguments.
- Print out the final answer and the number of steps it took to find the final answer in some self-explanatory format.

Function M-file for Finding a Zero

- (a) This function has one input argument, the initial guess for x .
- (b) This function has two output arguments, the final approximation of the x -value at which the zero is found, and the number of steps it took to find it.
- (c) Set a convergence criterion of $\epsilon = 10^{-6}$.
- (d) Initialize the number of iterations to zero.
- (e) Carry out the following steps until the absolute value of $f(x) < \epsilon$ where f is the function you are zeroing, and x is your current approximation of the zero.

- Calculate an approximation to the derivative of the function at this point, from

$$f'(x) = \frac{f(x + \epsilon) - f(x - \epsilon)}{2\epsilon}$$

- Calculate a new approximation to the zero, from

$$x = x - f(x)/f'(x)$$

- Update the number of steps that you have taken through the loop.

- (f) Return to main

Function M-file **f**

- (a) This function has one input argument, x .
- (b) This function has one output argument, y .
- (c) The relation is

$$y = x^2 - \sin(x)/x;$$

23. Temperature along an iron bar

Consider an iron bar 100 cm long. One end of it is fixed at 0°C while the other end is fixed at 100°C. You must plot the temperature as a function of the distance between its end-points by dividing the bar into 1000 elements and using only the following information:

- The temperature of the elements at either end is fixed. That is,
TBar(1) = 0; TBar(1000) = 100;
- Except for the fixed endpoints, the temperature in any element is determined as the average of its neighbors. That is,
TBar(i) = 0.5*(TBar(i-1) + TBar(i+1))
- The solution is obtained by iteration within a **while** loop until the average temperature in the bar, sum(TBar)/1000, changes by less than 0.01°C from the average temperature in the bar from the previous iteration.

Write two versions of MATLAB code that solve this problem. Both versions will employ a **while** loop which computes the new average temperature in the bar.

One version will have a **for** loop (internal to the **while** loop) over the elements of the bar. The other version may not employ any **for** loops, employing instead MATLAB's array indexing method: the colon (**:**) syntax.

As the starting condition for the **while** loop, assume that the bar is 0°C everywhere except at the warmer endpoint, TBar(1000).

24. Code Creation

The MATLAB script M-file below calls a MATLAB function M-file that returns the minimum, maximum, and average value of a 2D array. The average value is the sum of the matrix elements divided by the number of elements in the array. In the box below, write the function. *You are not required to provide any comment lines in the function.*

```
clear all;
Nrows = 37; Ncolumns = 93;
a = rand(Nrows,Ncolumns);
[minValue,maxValue,avgValue] = minMaxAvg(a,Nrows,Ncolumns)
```

25. Code Debugging and Refinement

The MATLAB function M-file below called `quadRoot.m` was discussed in earlier in this chapter.

```
%File: quadRoot.m
function [root1,root2] = quadRoot(A,B,C)
%QUADROOT returns the two real roots of A*x^2 + B*x + C = 0
%
% The two roots are root1 = (-B + sqrt(B^2 -4*A*C))/(2*A)
%                      root2 = (-B - sqrt(B^2 -4*A*C))/(2*A)
%
% An error is detected if 4*A*C > B^2 (imaginary roots)
% A warning is issued if 4*A*C = B^2 (identical roots)

if 4*A*C > B^2
    error('Roots would be imaginary. Get real!')
elseif 4*A*C == B^2
    warning('Roots will be the same, degenerate')
end

root1 = (-B + sqrt(B^2 -4*A*C))/(2*A); % This is line 15
root2 = (-B - sqrt(B^2 -4*A*C))/(2*A); % This is line 16
```


Unfortunately, the Professor wrote an incomplete program! When the function is employed as follows:

```
>> [r1,r2] = quadRoot(0,1,1)
```

the following error is reported by MATLAB.

```
Warning: Divide by zero in quadRoot.m at line 15
Warning: Divide by zero in quadRoot.m at line 16
r1 = NaN
r2 = -Inf
>>
```

A problem occurs when A is zero. When A is zero there should be only one solution, $x = -C/B$ unless B is zero as well. If both A and B are zero, there is no solution at all! Provide a fix-up to these problems by modifying the `if/elseif/else` construct above.

26. John Conway's game of life

Write a MATLAB script M-file that plays John Conway's game of life on a 50 by 50 grid. Here is how to initialize the grid:

- Make a 50 by 50 matrix called `grid` of random numbers between 0 and 1.
- Consider each cell in `grid`. If `grid(i,j)` is less than 0.4, set it to one. Otherwise, set it to zero.

Definitions:

- A cell that contains "0" is "dead".
- A cell that contains "1" is "alive".

Here's how the game is played:

- (a) Make an exact copy of `grid` into the array called `oldGrid`.
- (b) Loop through every cell in `oldGrid`.
- (c) Find the number of neighbors a cell has by summing the values of the eight neighboring cells (left, right, up, down, and the four diagonal ones).
- (d) If the cell is alive and it has less than 2 neighbors, it dies due to loneliness. (Set `grid(i,j)` to zero.)
- (e) If the cell is alive and it has 4 or more neighbors, it dies due to overcrowding. (Set `grid(i,j)` to zero.)
- (f) If the cell is dead and it has exactly 3 neighbors, it comes to life. (Set `grid(i,j)` to one.)

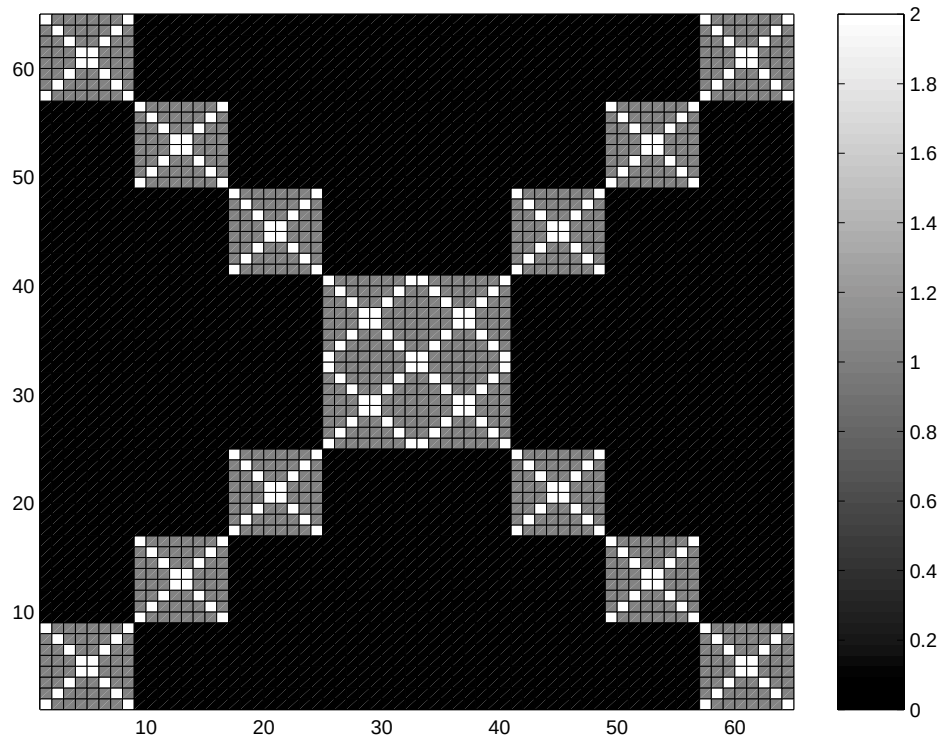
- (g) Any other condition, the cell stays as it was.
- (h) After you have considered every cell, go back to step 1.

You do not have to write any code that produces graphical output. You just have to code the algorithm described above.

Note that this program never terminates. Life goes on...

27. Slice this one up

Write MATLAB code that generates the 64×64 array that produces the following plot.



You may use the following commands to plot the array:

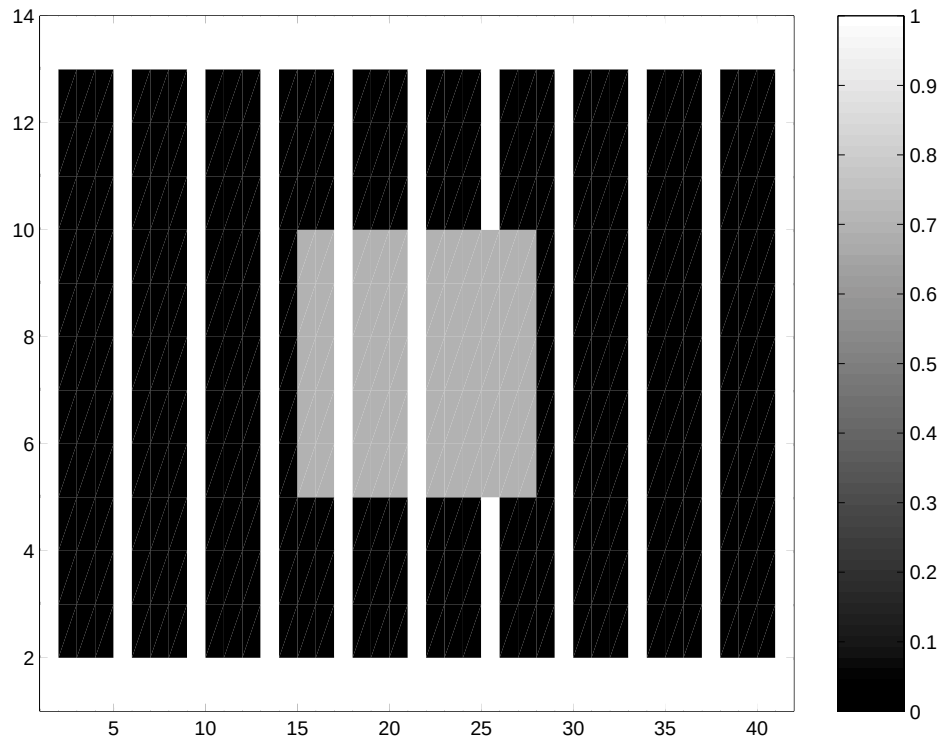
```
plotIt = b; % b is the array that contains the data to be plotted
plotIt(65,:) = 0; % Increase the number of rows by 1
plotIt(:,65) = 0; % Increase the number of columns by 1
pcolor(1:65,1:65,plotIt); % Plot it
colormap(gray)
colorbar
```

Try to make a solution that uses no more than 2 **for**-loops.

The array contains the data 0, 1, 2 in a pattern that produces the above figure. The 64×64 array can be made up of repeated use of an 8×8 array.

28. Any way you slice it

Write two MATLAB codes that generate the 13×41 array that produces the following plot. The first version uses `for`-loops only. The second version contains no `for`-loops but uses slice indexing of the form `iStart:Stride:iEnd`.



You may use the following commands to plot the array:

```
plotIt = a; % a is the array to be plotted
plotIt(14,:) = 0; % Increase the number of rows by 1
plotIt(:,42) = 0; % Increase the number of columns by 1
colormap(gray)
pcolor(plotIt) % Plot it
shading flat
colorbar
```

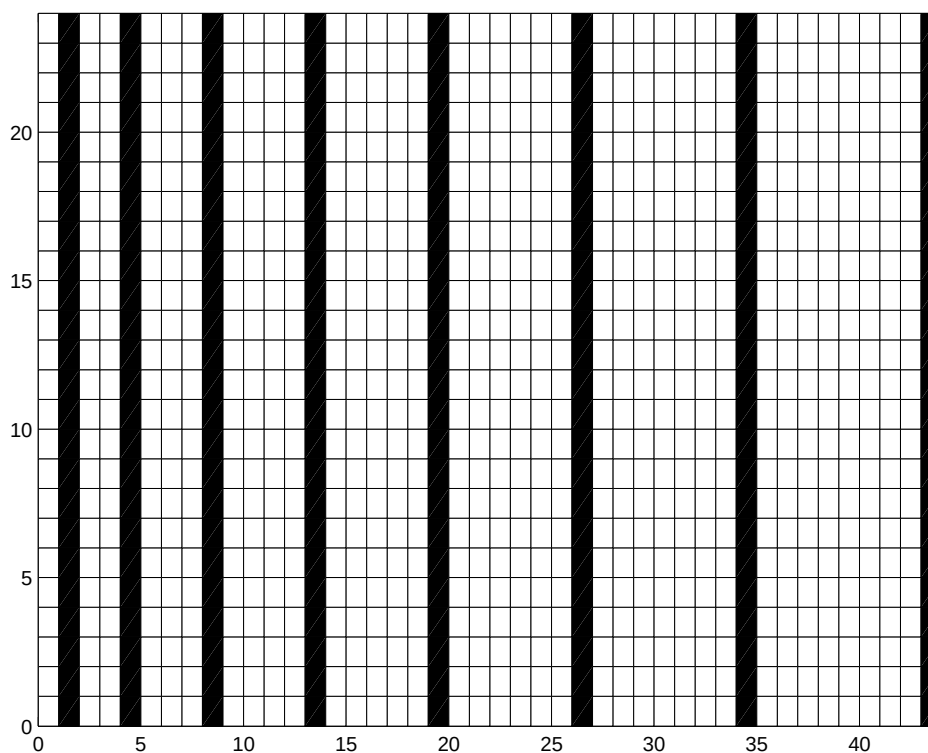
Some helpful hints:

- The vertical bars are 1 unit wide and created with a **Stride** of 4.
- Note that the rectangle is in front of one of the vertical bars and in back of two others.
- The color of the vertical bars and edges are established by giving them a numerical value of 1.0.

- (d) The color of the background is established by giving it a numerical value of 0.0.
- (e) The color of the inner rectangle is established by giving it a numerical value of 0.7.

29. Array striping

Write the MATLAB code that generates the array that produces the following plot. It should work for any response to user inputs (see example below).



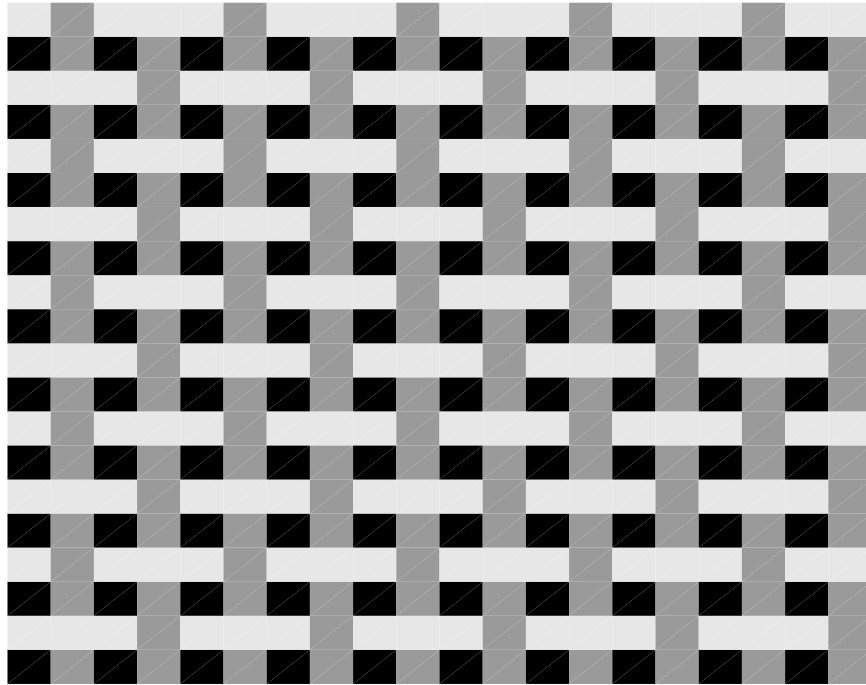
The array contains the data 0 (black stripe) or 1 (white stripe) in a pattern that produces the above figure. The following response to the prompts produced the output shown:

```
Number of stripes: 8
Number of Rows: 24
>>
```

30. Basket weaving

Write the MATLAB code that generates the array that produces the following basket-weave plot.

The array contains the data 0 (background), 0.6 (vertical dark grey stripes), or 1 (horizontal light grey stripes) in a pattern that produces the above figure. The following response to the prompts produced the output shown above:



How many times does the pattern repeat on each side? 5
 >>

Hint: The solution is made a lot easier when you realize that there is a small 4×4 array that is repeated to make the full pattern. Look at `grid(1:4,1:4)`.

31. Minor diagonals and crosses

The built-in function, `eye(M,N)` creates an $M \times N$ matrix with 1's on the main diagonal and 0's elsewhere. Example:

```
>> eye(3,4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
```

- (a) Write a function called `littleEye()` that does the same thing except the 1's are on the minor diagonal. Example:

```
>> littleEye(3,4)
ans =
     0     0     0     1
```

```

0    0    1    0
0    1    0    0

```

You should do this by having `littleEye()` call `eye()` and manipulating array indices using slice indexing.

- (b) Write another function called `cross()` that sums `eye()` and `littleEye()`, except that the maximum element returned can not be greater than 1. Example:

```

>> cross(3,3)
ans =
    1     0     1
    0     1     0
    1     0     1

```

You can avoid using `for` loops by using the `min()` function. For example, if `A` is a scalar and `B` is a matrix, `C = min(A,B)` returns a matrix `C` that has no value greater than `A`.

32. A smooth operation

Consider the following MATLAB script M-file that calls the function called `smooth`.

```

NH = input('Horizontal size of plot: ');
NV = input('Vertical size of plot: ');

a = rand(NV,NH);

avgChng = 1;

while(avgChng > 0.001)

    [a,avgChng] = smooth(a);

    plotIt = a;
    plotIt(NV+1,:) = 0; % Increase the number of rows by 1
    plotIt(:,NH+1) = 1; % Increase the number of columns by 1
    pcolor(0:NH,0:NV,plotIt)
    colormap(gray)
    colorbar
    pause(0.01)

end

```

When a user types `lookfor smooth` the following is received:

```
>> lookfor smooth
SMOOTH smooth   Smooth a 2D array
>>
```

When a user types `help smooth` the following is received:

```
>> help smooth
```

```
SMOOTH smooth   Smooth a 2D array
```

The `smooth` function smoothes a 2D array, its only input parameter, by performing adjacent horizontal and vertical point averaging following the prescription:

Assuming:

`R` is the number of rows in the array
`C` is the number of columns in the array

When $2 \leq i \leq R$ and $2 \leq j \leq C$

$$b(i,j) = (a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))/4$$

However, when the (i,j) 'th point is in the first or last row or column of the array, the algorithm averages only over the available horizontal or vertical array points.

The smoothed array is returned to `b`, the first returned variable.
 The average change is returned to the second return variable using the following algorithm:

$$\text{change} = \text{sum}(\text{sum}(\text{abs}(b - a)))/(R*C)$$

```
>>
```

Write the function `smooth.m`. Your function has to respond to the `lookfor` and `help` functions as described above.

33. Roots of a cubic equation

The roots (values of x such that $f(x) = 0$) of the cubic equation:

$$f(x) = Ax^3 + Bx^2 + Cx + D$$

where A , B , C and D are scalars, are given by the procedure below (attributed to Francois Viète, 1615). You will code this algorithm into a MATLAB function called `cubeRoots.m` where A , B , C and D are arrays:

- If $A = 0$ the above is not a cubic equation, it is a quadratic equation. Assume that there exists a function called `quadraticRoots.m` somewhere on the MATLAB path that operates on arrays in the same way that `cubeRoots.m` does and will return real roots if they exist and NaN's otherwise. Else,
- Define a, b, c as follows: $a = B/A$, $b = C/A$, $c = D/A$.
- Define $Q = (a^2 - 3b)/3$, $R = (2a^3 - 9ab + 27c)/54$.
- If $R^2 > Q^3$ there are no real roots and so the 3 roots are NaN's. Else,
- Define $\theta = \cos^{-1}(R/\sqrt{Q^3})$. (The MATLAB `acos(u)` function returns an array the same dimensions as u but with its elements being the $\cos^{-1}$ of the elements of u .)
- The three roots are:

$$\begin{aligned}x_1 &= -2\sqrt{Q} \cos\left(\frac{\theta}{3}\right) - a/3 \\x_2 &= -2\sqrt{Q} \cos\left(\frac{\theta + 2\pi}{3}\right) - a/3 \\x_3 &= -2\sqrt{Q} \cos\left(\frac{\theta - 2\pi}{3}\right) - a/3\end{aligned}$$

where the MATLAB `cos(u)` and `sqrt` functions return an array the same dimensions as u but with its elements being the $\cos$ and $\sqrt{}$ of the elements of u .)

Some notes:

- Your function `cubeRoots.m` must accept four input **arrays**, A , B , C and D , all with identical dimensions. Consequently, you must be careful with your use of MATLAB operators.
- Your function returns three **arrays** containing either NaN's or roots of the cubic equation.
- Your function must respond sensibly to the MATLAB `lookfor` and `help` functions.
- To make a variable into a NaN, assign it as follows: `root = NaN;`

34. Random, Rotated, Really

Write a MATLAB code that executes the following 5-step pseudocode. Each step is illustrated by a working example.

- (a) Using the `input()` function, interrogate the user for `R` and `C`, a row size and column size, respectively. Example:

```
Number of rows? 4
R =
    4
Number of columns? 3
C =
    3
```

- (b) Using the `input()` function, generate a $R \times C$ random array. Example:

```
a =
    0.0153    0.4660    0.2026
    0.7468    0.4186    0.6721
    0.4451    0.8462    0.8381
    0.9318    0.5252    0.0196
```

- (c) Rotate the array counter-clockwise. The first column becomes the last row, the second column becomes the second-to-last row and so on. Example:

```
a =
    0.2026    0.6721    0.8381    0.0196
    0.4660    0.4186    0.8462    0.5252
    0.0153    0.7468    0.4451    0.9318
```

- (d) Set to zero any array element that has a greater column index than row index. Example:

```
a =
    0.2026         0         0         0
    0.4660    0.4186         0         0
    0.0153    0.7468    0.4451         0
```

- (e) Change the sign of any array element that has a value less than $1/2$. Example:

```
a =
   -0.2026         0         0         0
   -0.4660   -0.4186         0         0
   -0.0153    0.7468   -0.4451         0
```

35. Can you (co)relate?

Write a MATLAB function M-file called `correlate.m` that does the following:

- Accepts a 2-dimensional array of doubles of any size. This array is the only variable in the input list.

- Returns the smallest absolute difference that exists between any two array elements.
- Returns the indices of the two closest elements.
- Responds sensibly to the `lookfor` and `help` functions
- **Note:** `abs(x)` returns the absolute value of all the elements of `x`.

Here's a working example:

```
>> lookfor correlate
CORRELATE correlate, finds the smallest difference between two array elements

>> help correlate

CORRELATE correlate, finds the smallest difference between two array elements

Returns the difference and array indices of the two closest elements

Programmer: Alex Bielajew 12/12/00

Problem on Final Exam for Eng101/200, Fall '00

>> A = rand(3,4)
A =
    0.9501    0.4860    0.4565    0.4447
    0.2311    0.8913    0.0185    0.6154
    0.6068    0.7621    0.8214    0.7919
>> [d,i1,j1,i2,j2] = correlate(A)
d =
    0.0086
i1 =
     2
j1 =
     4
i2 =
     3
j2 =
     1
```

36. Minesweeper initialization

Consider the MATLAB script M-file started below:

```

%MINESWEEPER minesweeper: Initialize the minesweeper game
clear all
echo off
clc

R = input('Number of rows in the grid: ');
C = input('Number of columns in the grid: ');
nMines = input('Number of mines: ');

if (nMines < 1)
    error('Too few mines')
elseif (nMines > R*C)
    error('Too many mines')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Your part goes here.
% Write your code on the following page(s).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp(grid)

```

Here's is what you must do.

- (a) Put `nMines` randomly on the grid by giving that grid point the value -1. For example, `grid(i,j) = -1` means that the `i,j`'th point on the grid contains a "mine". You will need to use both the `rand` and `int32` functions. (`int32()` converts its argument to a 32-bit integer that may be used as an array index.) Be sure that you deposit exactly `nMines` mines on the grid, no more, no fewer.
- (b) For each `grid(i,j)` point that is not a mine, count the number of mines surrounding it, those that are in the range `grid(i-1:i+1,j-1:j+1)`. Be careful with the boundaries of the grid. Note: `sum(sum(a))` sums up all the values on the points of the array called `a`.
- (c) Here is an example use of the program:

```

>> minesweeper
Number of rows in the grid: 5
Number of columns in the grid: 3
Number of mines: 5
    1    -1    1

```

1	2	2
0	2	-1
1	4	-1
-1	3	-1

>>

37. This problem is a minefield

Consider the MATLAB script M-file started below:

```
%LANDMINE landmine: A game of landmine
clear all
echo off
clc

R = input('Number of rows in the grid: ');
C = input('Number of columns in the grid: ');
pMine = input('Probability that a tile contains a mine: ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Your part goes here.
% Write your code on the following page(s).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

showGrid = grid;
showGrid(R+1,:) = -1;
showGrid(:,C+1) = 1;
pcolor(0:C,0:R,showGrid)
header = sprintf('\fontsize{20}There were %d survivors\n',nSurvived);
title(header)
axis off
```

This is a brief description of what you must do.

- (a) Between 1:R and 2:C-1 use the random number function `rand` to decide whether or not that gridpoint contains a landmine. In other words, if `rand < pMine` set the grid point to -1 to indicate the presence of a mine. Otherwise, set it to 0, to indicate that the gridpoint is safe.

- (b) Make the first column of the grid equal to 1. A “1” indicates a person on the grid. These people are trying to make it to the last column on the grid without touching a mine.
- (c) Within a while loop whose logical condition is that there are still people within `grid(:,1:C-1)`, use a double for loop to look at every point within `grid(:,1:C-1)`. If that point has value 1, then do the following:
 - i. Attempt to advance the person one square to the right (horizontally) and with equal chance: one square up, one square down, or no vertical change. Restrict the vertical movements so that the person stays within the grid. Use of the `max()` and `min()` functions, as described in class and in an earlier exam problem, makes this restriction quite efficient.
 - ii. If the proposed new location contains another person, then nothing happens.
 - iii. If the proposed new location contains a landmine, then both the person and the landmine are eliminated.
 - iv. If the proposed new location is within the last column, then that person has safely reached home and should not move anymore.
- (d) The game continues until there are no people remaining within `grid(:,1:C-1)`
- (e) Be careful not to move a person more than once during each execution of the double for loop.

38. War is heck

Write a MATLAB code for the game of War that is described as follows:

- The game is played on an $R \times C$ array, where R and C are inputs provided by the user.
- The array elements are filled with random numbers using the `rand()` function.
- Go through each element of the array. If the element is greater than 1/2 reset the array element to +1, else, set it to -1. Each one of these elements is a player in the game. Those assigned +1 are the Positives. Those assigned -1 are the Negatives. Both the Positives and Negatives think of each other not only as Opposites but also as Opposition—deadly opposition.
- Print out the starting number of Positives and Negatives.
- The Positives and Negatives are at war and here is how the battle is fought.
- An outer while loop is executed under the condition that there are players of both kinds still in the game.
- During each execution of the while loop, the following happens:
 - (a) Each player tries to moves randomly but with equal chance in one of the 4 directions, North(N), South(S), East(E) or West(W) by one array index.

- (b) If a player wants to move outside the bounds of the array, then the player disappears from the game. The penalty for cowardice is severe. (Set that array element to 0.)
- (c) If the array element that the player wants to move to is empty (0), the player moves there. (The +1 or -1 gets written into the new array element. The old array element is set to 0.)
- (d) If the array element that the player wants to move to is a player of the same type, then nothing happens.
- (e) If the array element that the player wants to move contains an Opposite, then they fight. Three things can happen, all with equal chance:
 - i. Mutual destruction—both players perish. (Both array elements get set to 0.)
 - ii. The Positive player wins and the Negative player is eliminated from the game. (The new location gets set to +1 and the old location gets set to 0.)
 - iii. The Negative player wins and the Positive player is eliminated from the game. (The new location gets set to -1 and the old location gets set to 0.)
- When the while loop ends, there are 3 possible outcomes. Provide a message that indicates how the game ended and the number of remaining players.

Here's an example of running the code.

```
>> war
Number of rows in the game: 10
R =
    10
Number of columns in the game: 12
C =
    12
The game starts with 64 +'s and 56 -'s
The -'s have annihilated the +'s. There are 17 -'s remaining.
>>
```

39. **Gaussian sum** Write a MATLAB function M-file, called `gauss()`, that is called as follows:

```
s = gauss(N) % where N is some integer
```

Your `gauss.m` responds to the `lookfor` and `help` functions as follows:

```
>> lookfor gauss
GAUSS gauss The Gaussian sum: gauss(N) = 1 + 2 + 3 + ... + N
>> help gauss
GAUSS gauss The Gaussian sum: gauss(N) = 1 + 2 + 3 + ... + N
>>
```

which gives the algorithm. Feel free to use the “trick” if you remember it.

40. Fibonacci sum

Write a MATLAB function M-file, called `fibonacci`, that is called as follows:

```
s = fibonacci(N) % where N is some integer
```

Your `fibonacci.m` responds to the `lookfor` and `help` functions as follows:

```
>> lookfor fibonacci
FIBONACCI fibonacci Returns the fibonacci number F(N)
>> help fibonacci
FIBONACCI fibonacci Returns the fibonacci number F(N)
The Fibonacci numbers are defined recursively thus:
F(1) = 1; F(2) = 1; F(N) = F(N - 1) + F(N - 2) where N >= 3
```

which gives the algorithm. The `error` function is used to return a message when the parameter (N in the call statement above) is less than or equal to 0. For example...

```
>> fibonacci(0)
??? Error using ==> fibonacci
Fibonacci numbers are defined for N >= 1
```

You may not use recursive function calls.

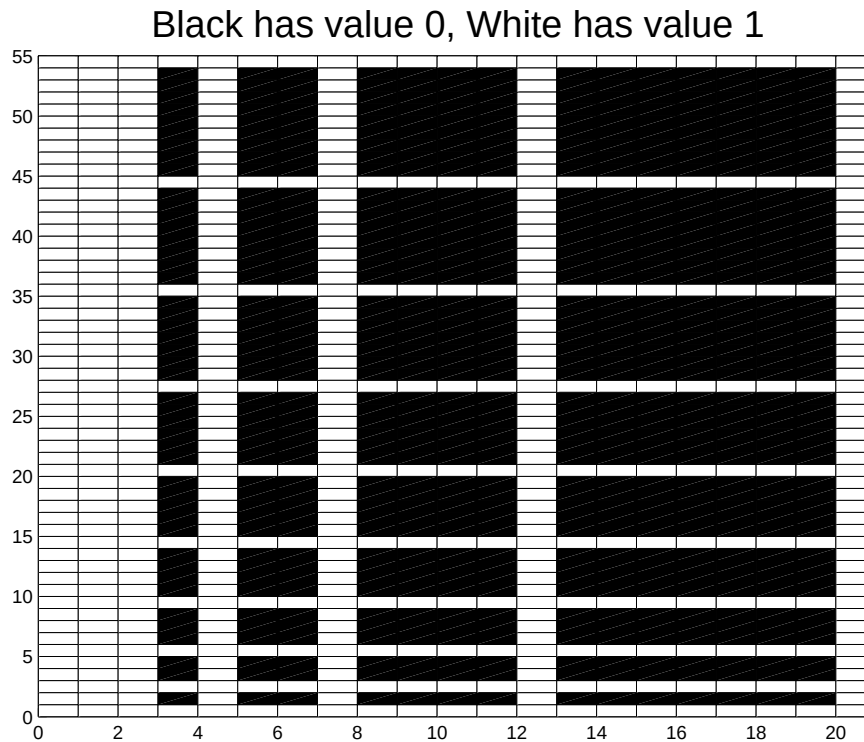
41. Array striping

Complete the MATLAB code below that generates the array that produces the following plot, in response to the following request for inputs:

```
Number of horizontal (Gaussian) white stripes (>0): 10
Number of vertical (Fibonacci) white stripes (>2): 8
```

Your code should work for any positive (> 0) response to the prompts.

The array contains the data 0 (black rectangles) or 1 (white stripes) in a pattern that produces the above figure. The i 'th vertical stripe is situated at the index `fibonacci(i)` (See preceding problem.) while the i 'th vertical stripe is situated at the index `gauss(i)` (See two problems back.) You may assume (if you wish) that the `gauss()` and `fibonacci()` functions, as described in the 2 previous problems, are on the Matlab search path.



```
NH = input('Number of horizontal (Gaussian) white stripes (>0): ');
NV = input('Number of vertical (Fibonacci) white stripes (>2): ');
```

```
%Add your MATLAB code starting here...
```

```
%The code below plots the graph if you supply
%NRows, NCols and a(1:NRows,1:NCols)
```

```
plotIt = a;
plotIt(NRows+1,:) = 0; % Increase the number of rows by 1
plotIt(:,NCols+1) = 0; % Increase the number of columns by 1
pcolor(0:NCols,0:NRows,plotIt)
title('\fontsize{20}Black has value 0, White has value 1')
colormap(gray)
```

42. Waldo wanders again

Write a simplified version of the Waldo program you encountered in Chapter 5. Here are the rules:

- Waldo's universe is `a = zeros(20)`
- Waldo starts off at `a(10,10) = 1`

- As indicated above, the indices of the array where Waldo is has a value 1, and 0 otherwise.
- Waldo is trying to make it to `a(16:20,16:20)`. If he makes it there, the game is over and Waldo has found his way home.
- On each step, Waldo attempts to walk randomly into one of the 8 adjacent squares, up, down, left, right and the 4 diagonal ones.
- If Waldo attempts to walk out of `a(1:20,1:20)`, Waldo is lost and the game is over.
- You do not need to supply any graphical output.

11.10 Projects

1. Is your dorm room cold?—Register your complaint!

Consider that your dorm room is represented by a 32×32 array, with 4 walls fixed at 20°C , a window 16×1 centered on the left wall which is fixed at 0°C (it's freezing outside), and a heat register comprised of 8 contiguous pixels (array elements that all touch another heat register array element on at least one side) that is fixed at some higher temperature. See Figure 1 for an example. Your job is to:

- Locate the register in the room somewhere (not occupying the same space as a wall or a window!) so that the temperature in the room is as uniform as possible,
- Heat the room to between 21 and 22°C by raising the temperature of the register.
- Full marks for algorithm design will be awarded to those who find a location for the heat register that keeps the temperature in the room uniform to within 2 degrees. (How to calculate this is discussed later.)

Here is a step-by-step procedure for this problem:

- Open a file called `a7.m` with any editor. This will be your script M-file. Once in MATLAB, you will run this program by simply typing `a7` at the MATLAB prompt.
- To start the problem, first fill a 32×32 array with zeros:

```
N = 32;
T = zeros(N);
```

- Then, make the first and last pixels of each row and column equal to 20, setting the temperature on the walls, then the 16 pixels of the window (fixed at 0) and finally the 8 pixels of the register (set at $T = 65^\circ\text{C}$) as shown in Figure 1.

The picture in Figure 1 is displayed using the following MATLAB commands:

```
plotT = T;           % Copy the T array
plotT(N+1,:) = 0;    % Increase the number of rows by 1
plotT(:,N+1) = 0;    % Increase the number of columns by 1
pcolor(1:N+1,1:N+1,plotT); % Plot it
colorbar;
```

Increasing the row and columns by 1 is necessary to get MATLAB to show the walls and the window, otherwise only the interior of the room would show.

- Inside a `while` loop, the i,j 'th cell gets its value from its neighbors, via an averaging relation of the form:

$$T(i,j) = (\text{oldT}(i+1,j) + \text{oldT}(i-1,j) + \text{oldT}(i,j+1) + \text{oldT}(i,j-1))/4;$$

where `oldT` is the temperature array after the previous pass through the `while` loop. You have to program this in the appropriate place in the `while` loop and you have to make sure that you do not violate the boundaries of the array. You can use `for` loops to do this although it is much more efficient to use “slice indexing”.

- (e) In the above averaging procedure, you may have overwritten some of the array that was meant to be fixed by the initial conditions, like the register, for instance. So, you had better make sure that you fix these up in the appropriate place before the `while` loop executes another pass.
- (f) The `while` loop is executed until the average change temperature changes less than 0.00001. The average change is calculated from:

```
difference = abs(T - oldT);
change = sum(sum(difference))/(N^2);
```

(Do a “help” on `abs` and `sum` to find out what these functions do.)

- (g) If you get everything correct, the final results look like Figure 2 which is plotted using the commands `pcolor(1:N,1:N,T); shading interp; colorbar;.`
- (h) Calculate and print out the average temperature and its variation:

```
average = sum(sum(T))/(N^2);
variation = sum(sum(abs(T - average)))/(N^2);
```

For the above example, I got the average to be $T_{\text{final}}^{\text{avg}} = 21.1514$ while the variation was $\Delta T_{\text{final}}^{\text{avg}} = 5.39824$. You should verify that you get this result before continuing. While this got the average temperature to within specifications, the variation is way out of spec. You have to do better. The easiest but most boring way to do it is to move the register around by typing in the configuration in the script M-file and running it again until you satisfy both the average and variation specifications. (BLIF’s rule #197 of computer programming: Always do it the easy, boring way first!) In this case, however, you will need some intuition to get the answers to within specifications because there are many possibilities, both in the location of the register and its shape. You should have a look in your dorm or other rooms to see if you find any common relationships between placement of windows and heat registers. You should also hope that the architects and engineers thought about this carefully because the impact on heating costs and human comfort are quite large.

Alternatively, you can try to find a way of automating the optimization procedure. How you do this is up to you! Be creative!

Finally, Figure 3 shows a different starting position with the ending results shown in Figure 4. This is a terrible solution! Not only is the variation worse, but the register is so hot that you would burn yourself if you got close to it!

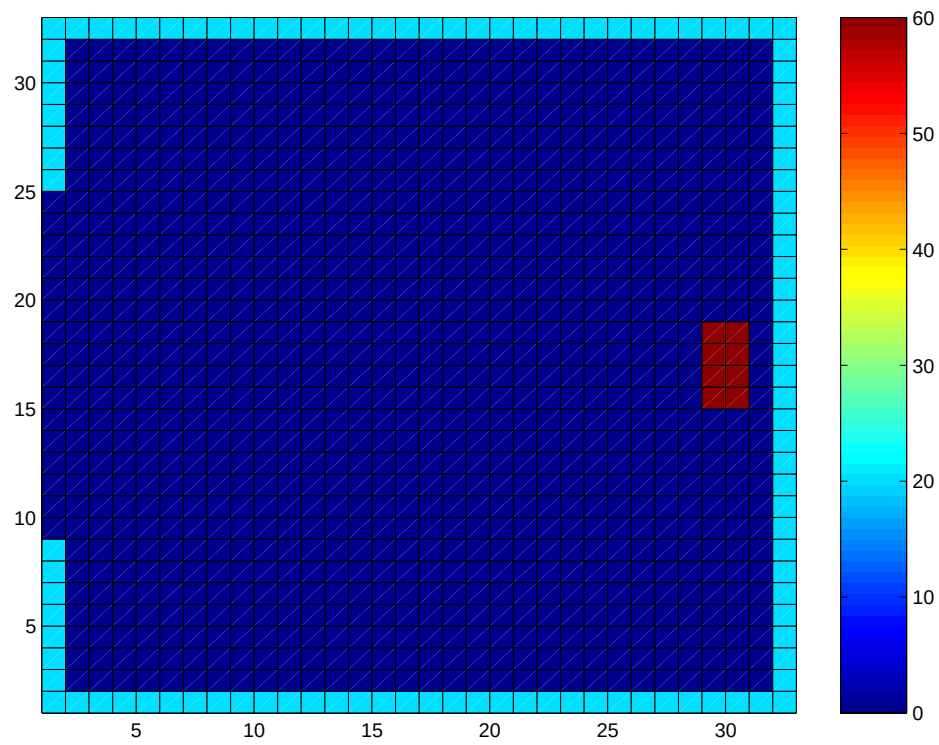


Figure 11.1: Typical starting condition for the 32×32 room with a register on the right wall at $T = 65^\circ\text{C}$.

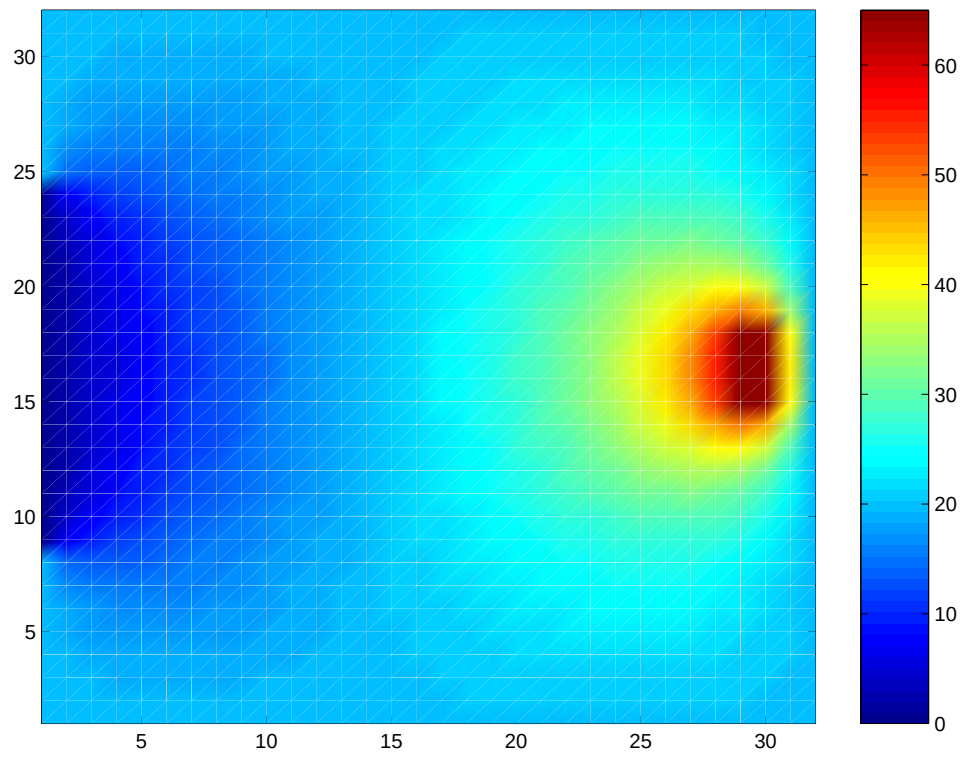


Figure 11.2: Final result for a 32×32 size room starting with the configuration of Figure 1. In this case, $T_{\text{final}}^{\text{avg}} = 21.1514$ and $\Delta T_{\text{final}}^{\text{avg}} = 5.39824$.

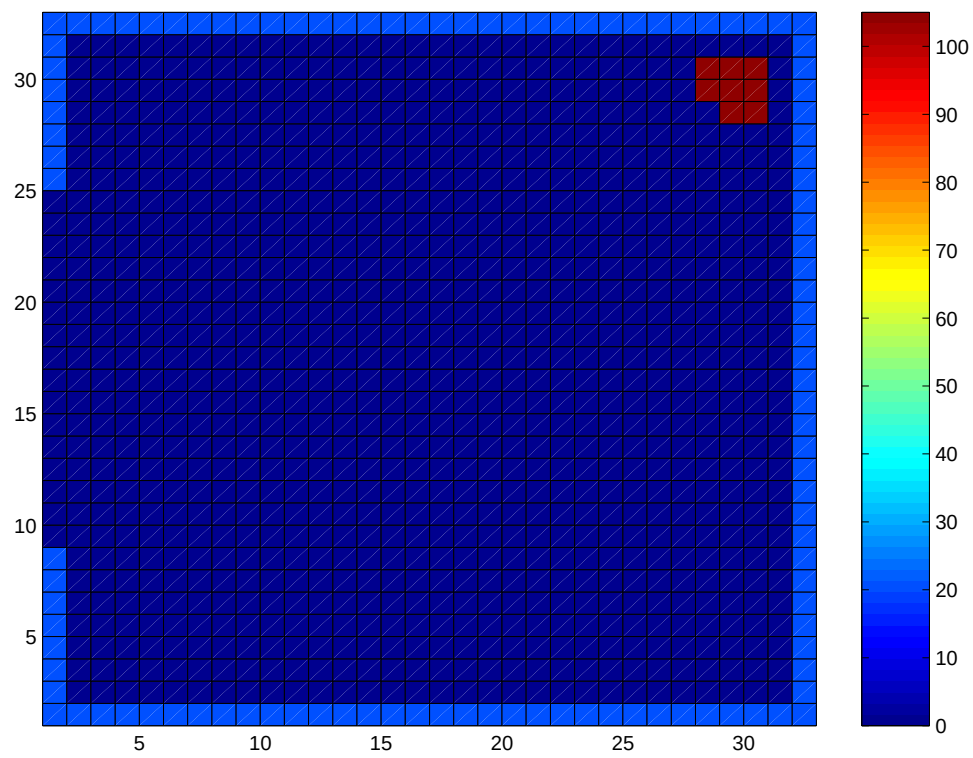


Figure 11.3: Another starting condition for the 32×32 room with a register in one of the corners at $T = 105^\circ\text{C}$.

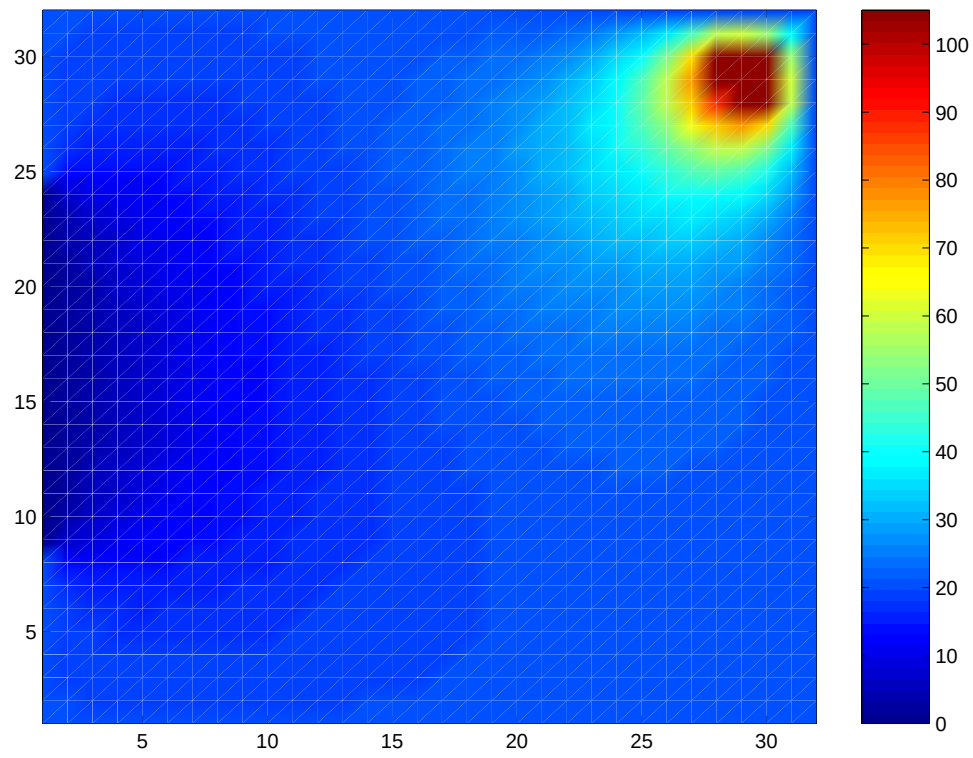


Figure 11.4: Final result for a 32×32 size room starting with the configuration of Figure 3. In this case, $T_{\text{final}}^{\text{avg}} = 21.1643$ and $\Delta T_{\text{final}}^{\text{avg}} = 6.1069$.

2. The plasma arc lamp—Do you get the point?

If you take two electrically conducting objects and connect them to opposite poles of a battery, nothing happens—usually. However, if you touch the objects together you can get a spark. If the voltage difference is great enough, you do not need to touch them together to have something interesting happen. Two conductors, if connected across a voltage that is high enough and brought close enough together, will cause an “arc” to be produced. This is lightning on a small scale!

If causing an “arc” is desirable for some reason, it helps to have the conductors “pointed”. This is why lightning rods are pointed—to attract the lightning in an efficient manner and then divert it into the ground safely away from things that can be damaged by it.

You can also make a brilliant light using high voltage and pointed conductors. These devices are called “plasma arc lamps” or “plasma arc torches”. The point (*pun intended*) of this problem is to make a plasma arc lamp, at least a virtual one, on the computer. The pictures that are produced are a thing of beauty, well, at least to me.

The template for this problem is found at the end of this problem description.

Here is a step-by-step procedure for solving this problem:

- (a) Type the template into a file.
- (b) Start MATLAB and run the template program. The first thing that pops up is a graphical representation of the start to the problem. It should look like:

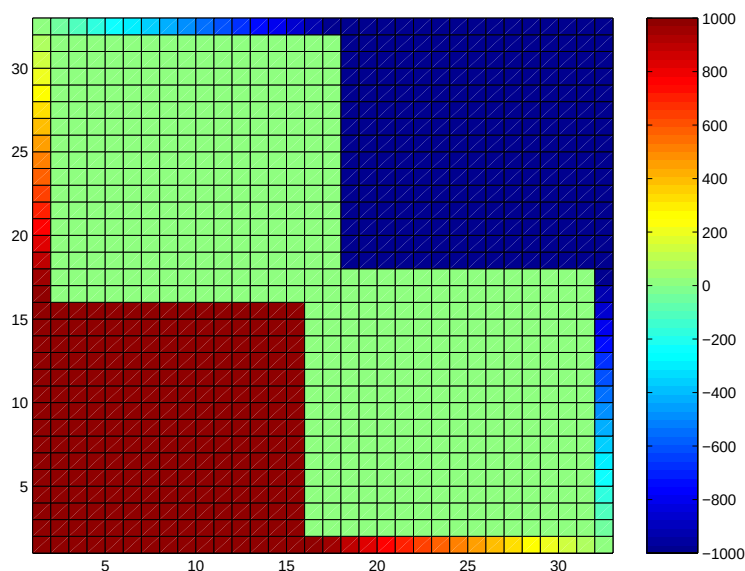


Figure 11.5: Starting conditions for a 32×32 voltage grid.

Do not proceed unless you get this.

The initial conditions for the voltage in a square grid of points (the grid size must be a multiple of 32) is determined as described below. The statement:

```
V = zeros(N);
```

creates an $N \times N$ array to represent the voltage and sets its values to 0. The statement:

```
V(1:15*N/32,1:15*N/32) = 1000;
```

creates the high voltage (+1000 Volts) electrode in the lower left hand corner of the figure, while the statement:

```
V(17*N/32+1:N,17*N/32+1:N) = -1000;
```

creates the low voltage (-1000 Volts) electrode in the upper right hand corner of the figure. The statements:

```
V(1,15*N/32+1:N) = linspace(1000,0,N - 15*N/32);
V(15*N/32+1:N,1) = linspace(1000,0,N - 15*N/32)';
V(N,1:17*N/32) = linspace(0,-1000,N - 15*N/32);
V(1:17*N/32,N) = linspace(0,-1000,N - 15*N/32)';
```

establish a linear change in the voltage from the electrodes to the upper left and lower right hand corners, so that these points at the upper left and lower right are zero. Note the use of the transpose operator “'” in two of the above statements.

- (c) If you then type a <RETURN> in the MATLAB command window, the figure on the previous page is replaced by a **shading interp** version and a second window pops up which will be empty. Now you have to start some programming.
- (d) Inside the **while** loop, the i,j 'th cell gets its value from its neighbors, via a relation of the form:

$$V(i,j) = (\text{oldV}(i+1,j) + \text{oldV}(i-1,j) + \text{oldV}(i,j+1) + \text{oldV}(i,j-1))/4;$$

where **oldV** is the voltage array after the previous pass through the **while** loop. You have to program this in the appropriate place in the **while** loop and you have to make sure that you do not violate the boundaries of the array. You can use **for** loops to do this although it is much more efficient to use “slice indexing” as in the determination of the initial conditions above.

- (e) In the above averaging procedure, you may have overwritten some of the array that was meant to be fixed by the initial conditions. So, you had better make sure that you fix these up in the appropriate place in the **while** loop.
- (f) The **while** loop is executed until the voltage array does not change by a significant amount. Code to test for this is provided.

- (g) After the voltage array is calculated, you calculate the magnitude of the electric field from it. The magnitude of the x -component of the electric field is obtained from:

$$Ex(i,j) = V(i+1,j) - V(i,j);$$

while the magnitude of the y -component of the electric field is obtained from:

$$Ey(i,j) = V(i,j+1) - V(i,j);$$

The magnitude of the electric field is obtained from a relation of the form

$$E(i,j) = \text{sqrt}(Ex(i,j)*Ex(i,j) + Ey(i,j)*Ey(i,j));$$

Be very careful of the array bounds!

- (h) Then the magnitude of the electric field is plotted as provided by the template.
 (i) If you get everything correct, the final results look like:

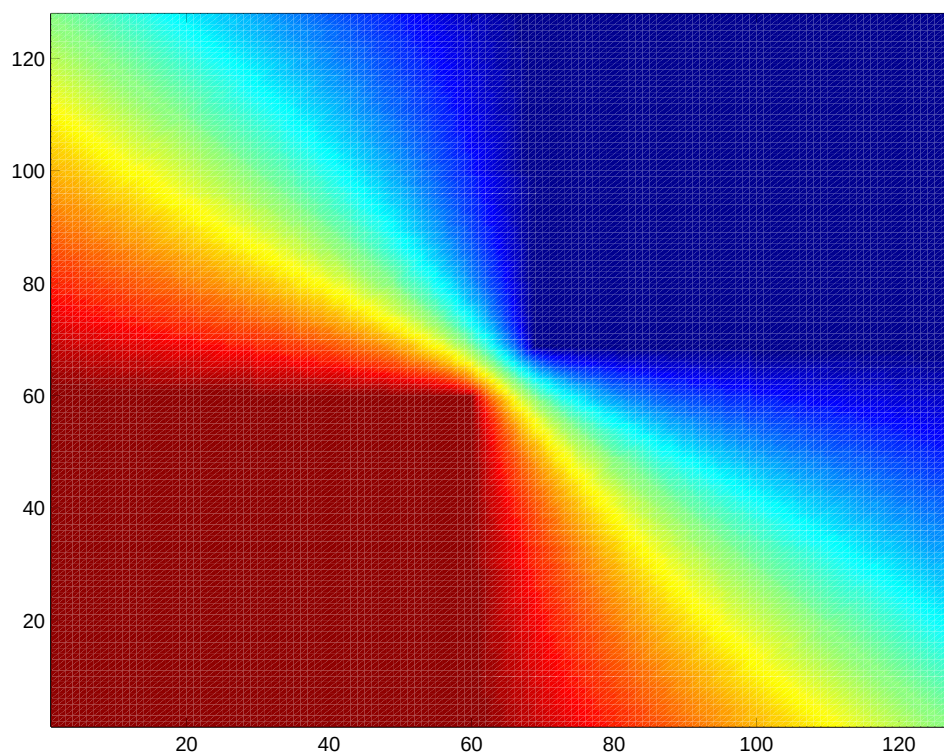


Figure 11.6: Final result for a 128×128 voltage grid.

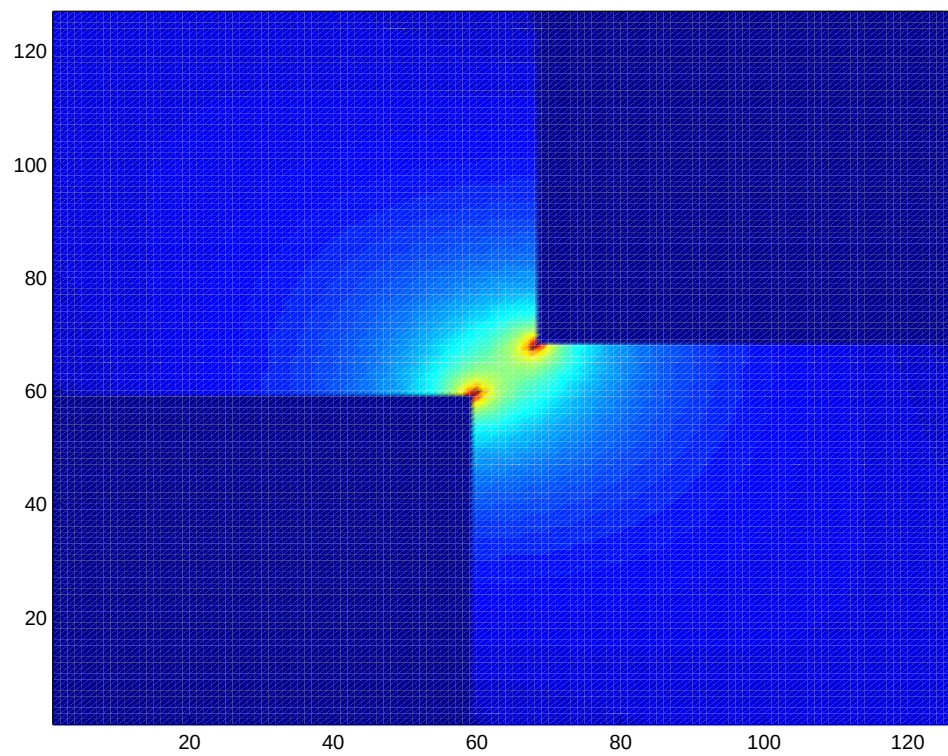


Figure 11.7: Electric field grid corresponding to the voltage in the previous figure.

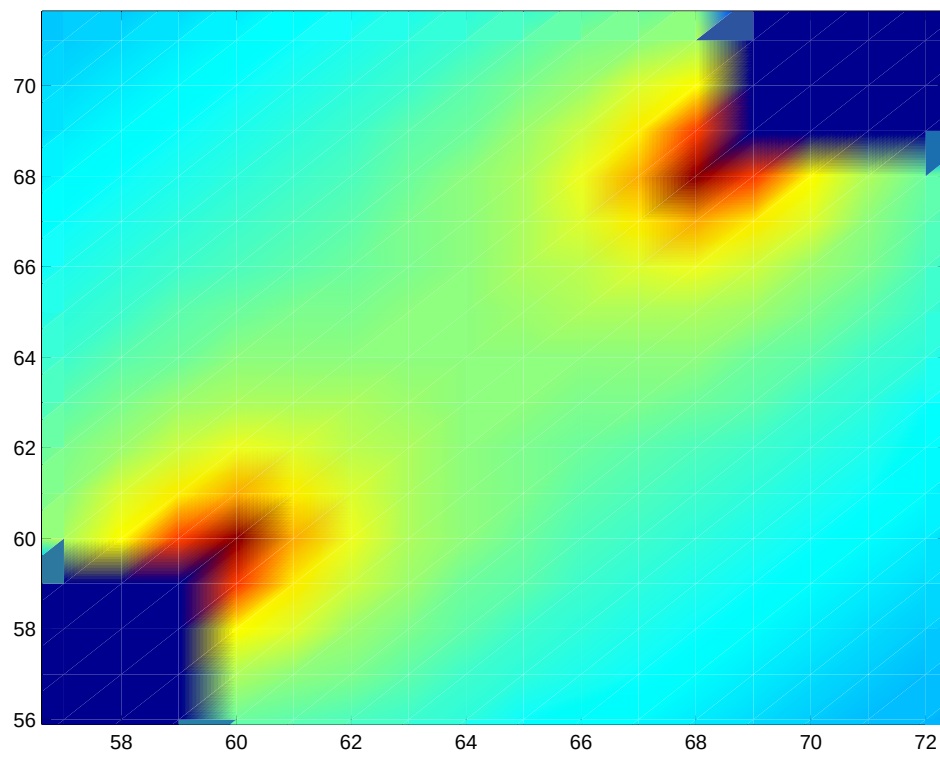


Figure 11.8: A “zoom in” on the previous picture to show the details of the arc. Even on a fine 128×128 grid you can see evidence of the “digitization” of the grid.

The code template...

```

N = 32;          % Array size, should be divisible by 32: 32, 64, 96, 128,...
V = zeros(N); % Creates Voltage array and sets its values to 0

% Sets initial values of the V array
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

V(1:15*N/32,1:15*N/32) = 1000;      % High voltage "point"
V(17*N/32+1:N,17*N/32+1:N) = -1000; % Low voltage "point"

% Linear ramps along edges from the high voltage point
V(1,15*N/32+1:N) = linspace(1000,0,N - 15*N/32);
V(15*N/32+1:N,1) = linspace(1000,0,N - 15*N/32)';

% Linear ramps along edges from the low voltage point
V(N,1:17*N/32) = linspace(0,-1000,N - 15*N/32);
V(1:17*N/32,N) = linspace(0,-1000,N - 15*N/32)';

% Plot the initialized array
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The following lets all the cells of V be printed,
% useful for making sure the array got initialized OK.
% Without this, the N'th row and column of V would not
% be displayed.
% Should NOT do it this way if using shading interp
plotV = V;          % Copy the V array
plotV(N+1,:) = 0; % Increase the number of rows by 1
plotV(:,N+1) = 0; % Increase the number of columns by 1
pcolor(1:N+1,1:N+1,plotV); % Plot it
colorbar;

pause;      % Wait for a keyboard event before continuing

tic;        % Starts the clock

oldV = V; % Save the initial value of the array in "oldV"

% Enter a loop that ends when the change in voltage is
% equal to or less so some value

```

```

change = 1; % Sets change to 1 to get the while loop started
while(0.001 <= change)

    % Take the average voltage of the 4 adjacent cells
    %...need student input here...

    % Resets initial values that should stay constant (see above)
    %...need student input here...

    % Calculates the change in voltage
    difference = abs(V - oldV);
    change = sum(sum(difference))/(N^2);

    % Prints out the change in voltage
    fprintf('Average change = %g\n',change)

    % Gives the oldV array the newly calculated voltage
    oldV = V;

end

time = toc; % Stores how long the program took to run
fprintf('Elapsed time = %g\n',time) % Prints it out

% Plots a representation of the voltage
pcolor(1:N,1:N,V);
shading interp;
colorbar;

% Calculate the magnitude of the electric field
Ex = zeros(N-1,N); % Allocate memory for Ex
Ey = zeros(N,N-1); % Allocate memory for Ey
E = zeros(N-1,N-1); % Allocate memory for E
%...need student input here...

figure(2);
pcolor(1:N-1,1:N-1,E);
shading interp;

```

3. Battleship revisited

You will write MATLAB code that will allow you to play the game “Battleship!” on your computer. There is only one ship, the size of the ship is 4 units and you only have 25 bombs. Otherwise the game rules are exactly the same as for the Battleship program discussed in Chapter 8.

However, you must use graphical output to show the status of the game board (with the ship hidden) and use the mouse to generate the bomb drops.

Hints:

Initializing the game grid

```
N = 10;
showBoard = zeros(N); % Make the board, value of zero for "open sea"
```

For more explanation:

```
>> help zeros
```

Drawing the game grid If you have an array called `showBoard`, here is how to display it:

```
plotBoard = showBoard; % Makes a copy of showBoard into plotBoard
plotBoard(N + 1,:) = 0; % Boosts number of rows in plotBoard by 1, assigns 0's
plotBoard(:, N + 1) = 0; % Boosts number of columns in plotBoard by 1
figure(1); % Working with figure 1
pcolor(0:N,0:N,plotBoard); % Make the plot
axis off % Turn of axis labeling (may want to leave on during debugging)
```

For more explanation:

```
>> help figure
>> help pcolor
>> help axis
```

Choosing the first point of the ship

```
i = 1 + N*rand;
j = 1 + N*rand;
hiddenBoard(int32(i),int32(j)) = 1; %Ship point has value 1
```

Note that you will have to have two boards, one that you show during the game and one that is kept hidden, where the ship points reside.

Note the use of the `int32()` function. The indices of an array should always be integral. The statements `i = 1 + N*rand;` `j = 1 + N*rand;` are intrinsically doubles. You need the `int32()` function to convert from doubles to integers.

For more explanation:


```
>> help rand
>> help int32
```

Dropping bombs with the mouse

```
[x,y] = ginput(1);
jDrop = int32(x + 1);
iDrop = int32(y + 1);
```

The above generates i,j indices on the game board. You can test `hiddenBoard(iDrop,jDrop)` to see if that array elements represents open sea (value 0), a part of the ship (value 1), or something else. In my version of the game, I used a value of 0.25 to represent a bomb dropped on open sea, and a 0.75 to represent a struck ship.

For more explanation:

```
>> help ginput
```

4. Swarm ball

This project was inspired by my 7-year-old daughter's soccer team, *The Muggles*. If you have ever watched a group of 7-year-olds play soccer, it bears very little resemblance to the game called soccer. It resembles more closely what would should be called "swarm ball".

In swarm ball, the object of the game is to surround the ball with as many of your 5 other teammates as possible, for the chances of moving the ball in the direction of the opponents' goal is proportional to the number of your team surrounding the ball. Of course, the other team has a similar strategy. So, around the ball is a swarm of players all kicking at the poor thing. The ball usually blurts out in one direction and the swarm reassembles around the ball. Goaltenders, while still a part of swarm ball, are almost completely irrelevant. The ball is always rolling on the ground. Would you bend down to pick up a soccer ball surrounded by 24 flailing feet? I think not.

So, this project's task is to program a simulation of swarm ball with the following requirements.

- (a) A 2-dimensional array, called `field`, contains the game, the players, the ball and the goalposts. The width and length of the field are defined by:

```
fieldWidth = 31; % Width of the field
fieldLength = 41; % Length of the field
```

- (b) The following values of the elements of `field` determine what that point occupies. Use the following as it gives a sensible rendition when `field` is plotted using `pcolor`:

```
grass = 0.5; % Grass color
ball = 0.62; % Soccer ball's color
red = 0.8; % Red's team color
blue = 0.2; % Blue's team color
redGoal = 0.7; % Red's goal color
blueGoal = 0.4; % Blue's goal color
boundary = 0; % Field boundary
```

- (c) Define the field, the boundaries and the goals as follows:

```
field = zeros(fieldWidth,fieldLength);
field(2:fieldWidth-1,2:fieldLength-1) = grass;
field(10:22,1) = redGoal;
field(10:22,end) = blueGoal;
```

- (d) Define the starting position of the game using the following indices:

```
ballX = 21;
ballY = 16;
```

```
field(ballY,ballX) = ball; % The ball's starting position
```

```
% Red team starting positions
```

```
redX = [17,17,17,11,11,11];
```

```
redY = [13,16,19,10,16,22];
```

```
% Blue team starting positions
```

```
blueX = [25,25,25,31,31,31];
```

```
blueY = [13,16,19,10,16,22];
```

and then place these points on the field.

- (e) The game proceeds in a series of steps, not to exceed 120, when time runs out. During each step, the following happens:

- i. Advance each player towards the ball in either the vertical, horizontal or diagonal directions. A player can only move to an open point (no other players, boundaries, goals or ball). Be sure that you do this in a fair fashion. If you do not and move, say, all the red players first, they would block the blue players from the ball and the red team would have an unfair advantage.
- ii. Count the number of players (`redKick`, `blueKick`) of each team surrounding the ball. You must check all 8 points, vertical, diagonal and horizontal, in the vicinity of the ball. The team that kicks the ball will be determined by the following prescription:

```
% Pick which team gets the kick
if (redKick == 0 & blueKick == 0)
    kick = 0; % Neither team kicks
elseif (rand < redKick/(redKick + blueKick))
    kick = 1; % Red team kicks
else
    kick = -1; % Blue team kicks
end
```

- iii. Pick the direction that the ball will go according to the following prescription:

```
% Pick a direction to kick the ball
if (kick)
    % Add some randomness in the y direction, move towards the goal
    randKick = rand;
    if (randKick <= 0.3333)
        kicky = -1;
    elseif (randKick <= 0.6667)
        kicky = 0;
    else
        kicky = 1;
    end
end
```

```
% Make sure kick is usually in the correct direction
kickx = 0;
if (rand < 0.75)
    if (kick > 0)
        kickx = 1;
    elseif (kick < 0)
        kickx = -1;
    end
end

if (kickx == 0 & kicky == 0)
    kick = false; % No kick, a total miss
end
end
```

- iv. When the kick direction is determined, place the ball in the first available point along the direction. If ends up in a goal, one of the teams wins and the game is over. If the ball hits the boundary, there is no kick. (This is simpler to program than a throw-in from the sidelines, or a goal kick.)
- (f) Use a `pcolor` display after each repositioning and each kick.

5. Going, going, going...Gone!

Baseball, like most sports, is a game of inches. (Except for soccer, of course, which is a game of feet!) However, some things in baseball happen on a grand scale, like the distance a ball travels when hit for a home run.

There are two parts to this problem, both to be solved with MATLAB. You can solve them using any numerical/computational technique you like so long as it is done with MATLAB. You must use a function called `homeRunDistance` which is described in a little detail below.

- (a) A baseball is struck at a height of 3 feet about the ground. It can have any velocity between 0 (Bielajew territory) and 250 feet/second (McGuire/Sosa territory). It can have any angle with respect to the ground from 0 (parallel to the ground) to 90 degrees (straight up in the air).

Using the equations of motion described below, write a MATLAB function that starts something like this. (It can be different.)

```
function d = homeRunDistance(y0,theta,v,G,D)
%HOMERUNDISTANCE returns distance a ball travels (hits the ground)
% homeRunDistance(y0,theta,v,G,D) calculates distance a baseball travels
%       before it hits the ground assuming that:
% y0    is the initial height of the ball (ft)
% theta is the angle in degrees of the start of the trajectory with
%       respect to the ground (90 is straight up)
% v     is the initial speed (ft/s)
% G     is the magnitude of the acceleration due to gravity (ft/s/s)
% D     is the magnitude of the drag factor (1/ft)
%
% The components of acceleration are:
%
% ax = -D*vx*v      (x-direction in units ft/s/s)
% ay = -D*vy*v - G  (y-direction in units ft/s/s)
%
% where
%
% vx = v*cos(pi*theta/180) (x-direction velocity in units ft/s)
% vy = v*sin(pi*theta/180) (y-direction velocity in units ft/s)
```

Using a script M-file to call the function `homeRunDistance`, make a 3D plot of distance versus `theta` and `v` between the limits described above. (A 20 by 20 array should be sufficient.)

- (b) For an initial velocity of 250 ft/s, what angle should the ball be hit for maximum distance? You can determine the maximum in any way you please as long as it is

part of the same script M-file used above and it gives you approximately the right answer! When you obtain this angle make a 2D plot of the trajectory of the ball both with and without air resistance ($D = 0$). “Plotting the trajectory” means making a 2D plot where the y -axis is the height and the x axis is the distance. Without air resistance it would have a parabolic shape.

11.10.1 Supplementary material: Trajectories without air resistance

If a ball starts off at height y_0 , horizontal position $x_0 = 0$, with components of velocity $v_{x,0}$ and $v_{y,0}$ in the presence of constant gravity providing an acceleration towards the ground, $a_y = -G$, the situation is one of constant acceleration.

The trajectory as a function of time may then be described by:

$$\begin{aligned}x &= v_{x,0}t \\y &= y_0 + v_{y,0}t - \frac{1}{2}Gt^2\end{aligned}$$

We can solve for the time of flight, t_{flight} , it takes for a ball to return to the ground using the quadratic equation and obtain the time and distance as follows:

$$\begin{aligned}t_{\text{flight}} &= \frac{v_{y,0} + \sqrt{v_{y,0}^2 + 2Gy_0}}{G} \\d &= v_{x,0}t_{\text{flight}}\end{aligned}$$

For example, if we use $y_0 = 3$ (ft), $v_{x,0} = v_{y,0} = 250/\sqrt{2}$ (ft/s), and $G = 32.174$ (ft/s/s) we get $d =$ about 1950 feet for the distance! The inputs are realistic and so we know that air resistance must play a big role. However, knowing that there is an exact answer in the case of no wind resistance will serve as an important test of our stepping algorithm described later.

11.10.2 Supplementary material: Trajectories with air resistance

In this case, the equations of motion are different because the accelerations are not constant. In this case they are:

$$\begin{aligned}a_x &= -Dv_x\sqrt{v_x^2 + v_y^2} \\a_y &= -Dv_y\sqrt{v_x^2 + v_y^2} - G\end{aligned}$$

where D is a measure of the resistance of the air. The total air resistance is proportional to v^2 and points in the direction opposite to the instantaneous direction of the ball.

The drag factor is related to the physical characteristics of the ball (or projectile) and has the form:

$$D = \frac{1}{2} C_{D,\text{ball}} \rho_{\text{air}} A_{\text{ball}} / m_{\text{ball}}$$

where ρ_{air} is the density of the air, A_{ball} is the cross sectional area of the ball, m_{ball} is the mass of the ball and $C_{D,\text{ball}}$ is the “drag coefficient”, a unitless quantity. For smooth spheres, $C_{D,\text{ball}}$ is always very close to 0.5. However, for a baseball it’s closer to 0.35 because of the seams. A pitcher can also change $C_{D,\text{ball}}$ by adding a little extra to it (saliva, vaseline) or taking a little away (cutting the ball). Baseballs that are newly manufactured have a shine that is rubbed off before the start of the game by the umpires and occasionally by the pitchers. This IS legal. All they are doing is making the $C_{D,\text{ball}}$ a little bit smaller to make the pitches faster.

The D for a major-league baseball can be worked out easily from the above relations. Use $D = 0.0019285$ and $G = 32.174$ for this problem.

11.10.3 Supplementary material: Stepping algorithms

Let’s go back to the equations of motion.

$$\begin{aligned} a_x &= -Dv_x \sqrt{v_x^2 + v_y^2} \\ a_y &= -Dv_y \sqrt{v_x^2 + v_y^2} - G \end{aligned}$$

If we assume that the acceleration is constant over a small time interval, Δt , then make a stepping algorithm as follows:

- (a) Start with the initial conditions $x_0 = 0$, $y_0 = 3$, $v_{x,0} = v_0 \cos \theta$ and $v_{y,0} = v_0 \sin \theta$, $t_0 = 0$, and a step counter i initialized to 0.
- (b) Increment the step counter $i = i + 1$.
- (c) Calculate the acceleration at the start of the step:

$$\begin{aligned} a_{x,i} &= -Dv_{x,i-1} \sqrt{v_{x,i-1}^2 + v_{y,i-1}^2} \\ a_{y,i} &= -Dv_{y,i-1} \sqrt{v_{x,i-1}^2 + v_{y,i-1}^2} - G \end{aligned}$$

(d) Update the positions and velocities assuming constant acceleration.

$$\begin{aligned}x_i &= x_{i-1} + v_{x,i-1}\Delta t + \frac{1}{2}a_{x,i}(\Delta t)^2 \\y_i &= y_{i-1} + v_{y,i-1}\Delta t + \frac{1}{2}a_{y,i}(\Delta t)^2 \\v_{x,i} &= v_{x,i-1} + a_{x,i}\Delta t \\v_{y,i} &= v_{y,i-1} + a_{y,i}\Delta t\end{aligned}$$

(e) Stop if y_i falls below 0, otherwise go back to step 2.

There are a few subtleties you will have to work out. One is to guarantee that the y never becomes negative. The other is to make a reasonable choice for Δt . Justify your choice. Recall that we are assuming that the acceleration is constant over the step. It isn't. However, a proper choice for Δt make sure that the acceleration is nearly constant. This week's lectures will be riddled with hints on the solution of this problem!

6. This problem sucks!

Have you ever wondered how one can make a vacuum?

One can make a good vacuum, comparable to the molecular/atomic density in deep space by employing a multi-stage process. In the first stage would we would take, say, a cylinder with a piston in it (much like a combustion chamber in a car) and a pipe attached to a closed canister that you want to evacuate. With a valve in the canister (to the cylinder) open and a valve in the cylinder (to the outside) closed, withdraw the piston. The pressure in the canister and the pipe is reduced. Now, close the valve to the canister and open the valve to the outside, allow the piston to descend and repeat. One can only go so far with this method. Soon there are too few molecules in the canister and enough molecules in the mechanism of the outside valve that one gets very little or no gain on each stage of the pump cycle.

The second stage (the one that this problem is concerned about) is to introduce a *cold trap*. A separate chamber (attached to the canister) has a cold surface in it to which molecules adhere. This further reduces the number of molecules. A third stage might involve causing the molecules to become ionized and then they may be “swept out” with an electric field. Making a vacuum is an interesting combination of a lot of different technologies—mechanical, electrical, chemical... Let’s focus on the second stage. With MATLAB, first construct two chambers of identical dimensions, connected

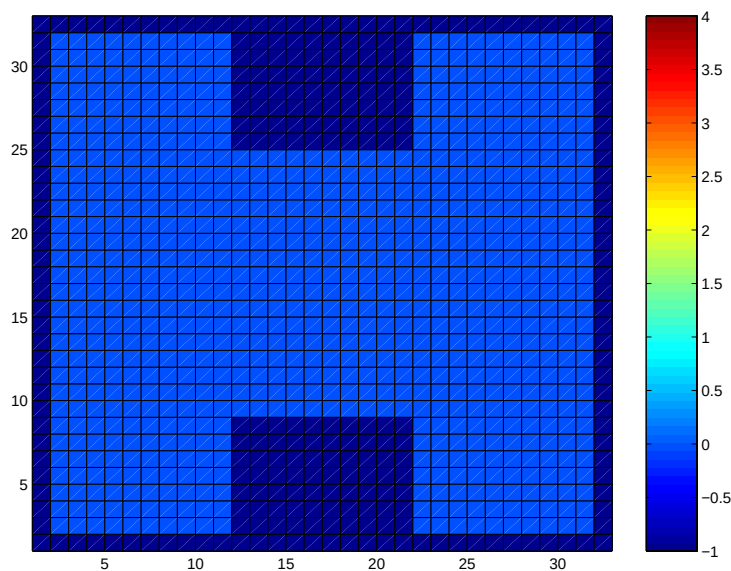


Figure 11.9: Two chambers and a pipe (evacuated, value = 0) surrounded by walls (solid, value = -1).

together with a pipe. The ones I made were made up of 2D arrays and look like those depicted in first figure. In my example I used chambers that were 30 units high and 10 units wide. The pipe was 16 units high and 10 units wide. (A “unit” is the width of

a “cell”.) Thus the entire geometry of the pipe, the two chambers and enclosing walls could be represented by a 32×32 array in MATLAB. The space within the walls is given the value “0” for vacuum and the walls are given the value “-1”.

Now, fill one of the chambers with molecules, placed randomly. In the example in second figure, I placed 100 molecules. Note that I allowed more than 1 molecule per cell. Each cell contains a number that represents the number of molecules within the cell. Since MATLAB’s random number generator seeds differently every time, your distributions will look different.

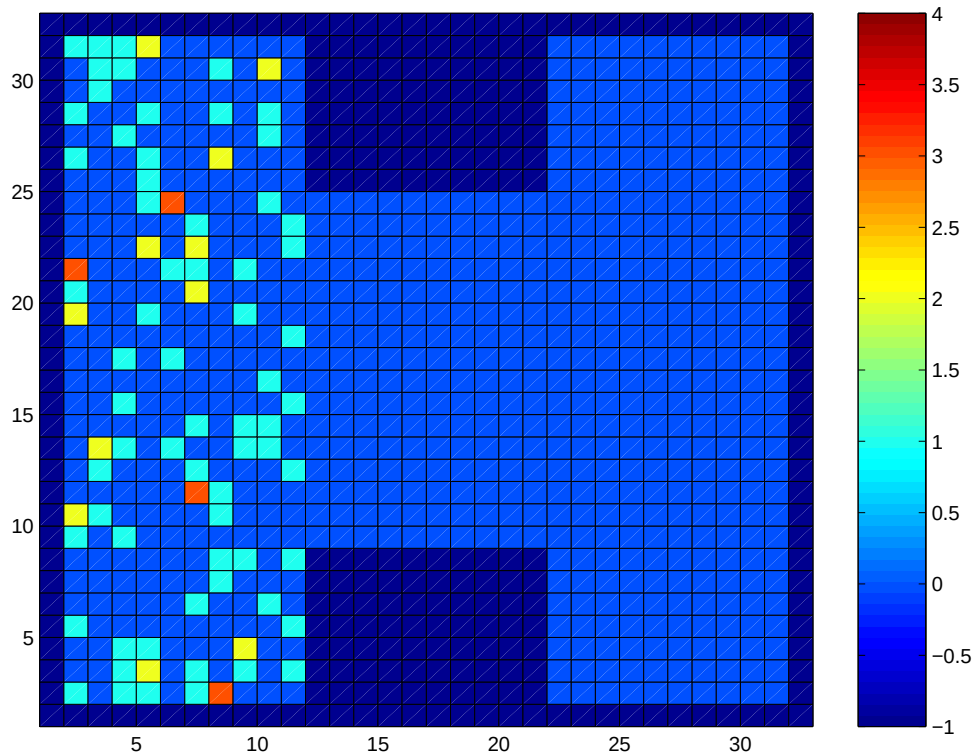


Figure 11.10: The left chamber filled with 100 molecules.

Now, a time-step is executed. During this time-step, each individual molecule can move randomly in any direction by one cell, upwards, downwards, sideways, or diagonally. If the new direction would cause the molecule to hit one of the walls, the proposed step is invalid and a new direction should be picked randomly. If a molecule enters the rightmost column of cells, it “sticks” to the cold wall and can no longer be moved. The third figure shows the distribution of molecules after 2000 time-steps.

Finally, the simulation ends when all the molecules are trapped in the cold trap. The final configuration of my example is shown in the fourth figure.

Here is what you must do to complete this problem.

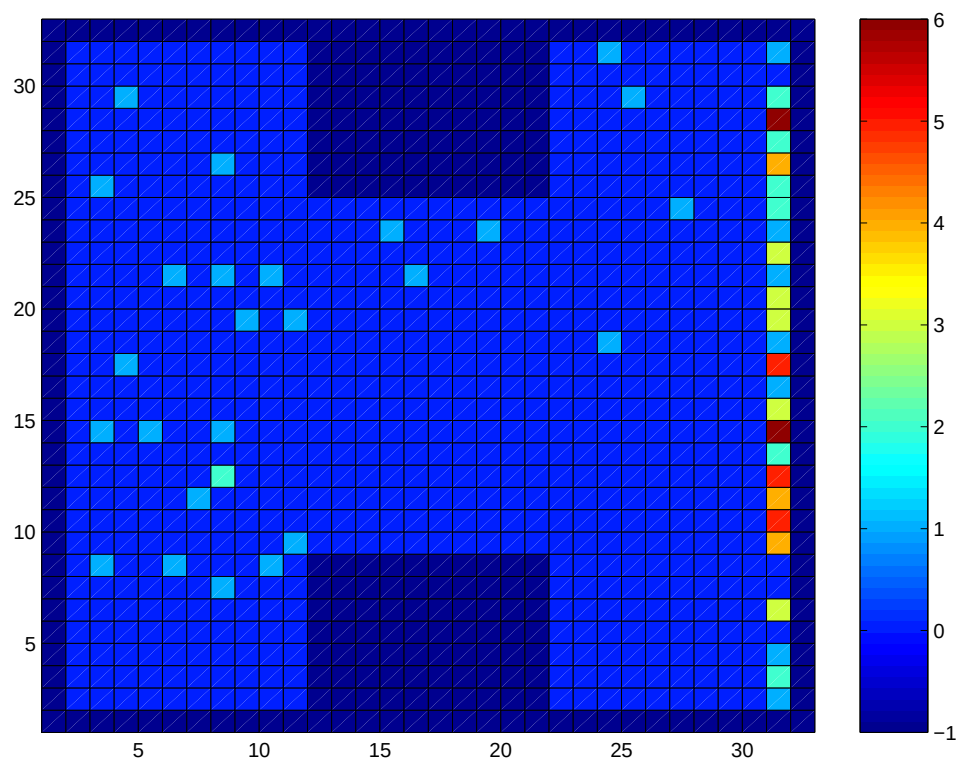


Figure 11.11: After 2000 time steps, 73 molecules have stuck to the cold wall in the cold trap and 27 are still “free”.

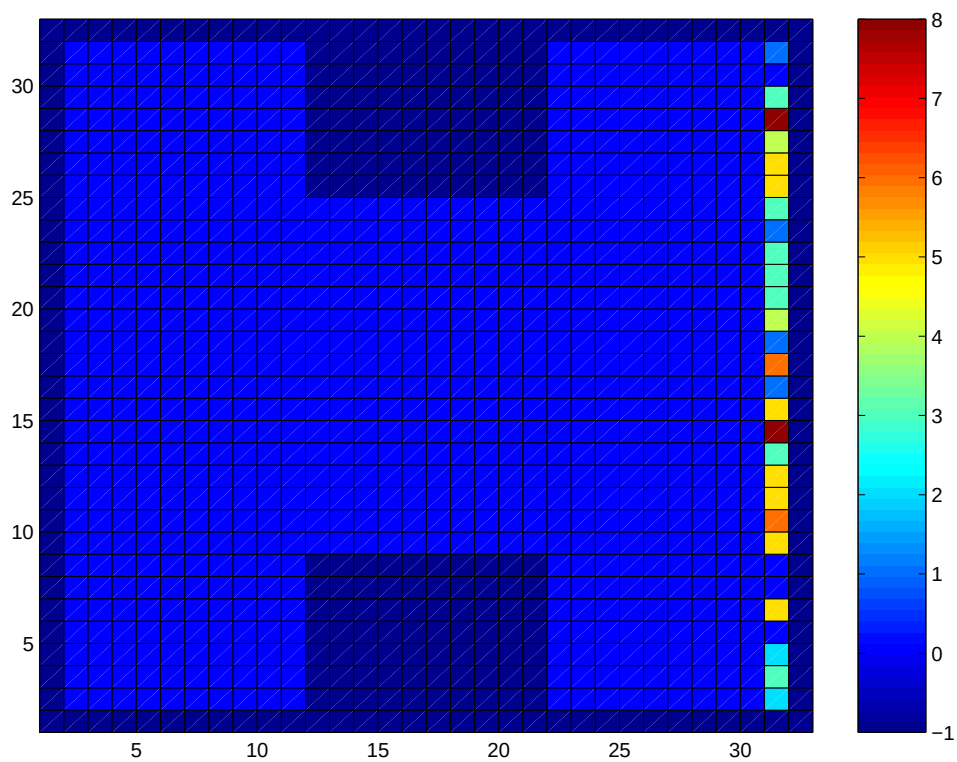


Figure 11.12: After about 8000 time-steps all the molecules have stuck to the cold wall of the cold trap.

You must write a MATLAB script M-file that does the following:

- (a) Prompt the user for:
 - i. Chamber height and width
 - ii. Connecting pipe height and width.
The pipe height must be less than or equal to the chamber height and greater than 0.
 - iii. The number of molecules to simulate.
- (b) Display the initial geometry as in first figure.
- (c) Place the molecules in the left chamber and display the initial configuration as in the second figure.
- (d) Execute the time steps until all the molecules stick to the right most wall. Display the final configuration as in fourth figure as well as the number of time-steps it took to do the simulation.

Some hints:

- When testing your code, use geometries with just a few cells for height and width and just a few molecules. This will allow the simulations to complete quickly and will test the more subtle parts of the code, the wall collisions and the action of sticking to the cold wall.
- When testing your code, make lots of use of visual output. Put in some graphical output for some or all of the time-steps with a `pause` statement following the graphics instruction. For example, the third figure shows some intermediate graphical output. This is a very valuable way of debugging the operation of your code.
- You may need the following MATLAB function in your code (it is not mentioned in the book!) to place the molecules randomly at the start. Using `help uint32`:

`UINT32` Convert to unsigned 32-bit integer.

`I = UINT32(X)` converts the elements of the array `X` into unsigned 32-bit integers. `X` can be any numeric object (such as a `DOUBLE`). The values of a `UINT32` range from 0 to 4294967295. Values outside this range are mapped to 0 or 4294967295. If `X` is already an unsigned 32-bit integer array, `UINT32` has no effect.

·
·
·

See also `...UINT8`, `UINT16`, `INT8`, `INT16`, `INT32`.

7. Solve the universe, plane and simple

Programmers are an ambitious lot, and so this problem is to simulate the evolution of the universe, assuming that you know the position and velocity of every object in it¹. After solving this problem, anything else thrown at you in the rest of your career will seem like a “piece of cake”. For simplicity we will assume that the universe is only 3-dimensional (two for space, one for time) instead of 4-dimensional and that each object in the universe is “point-like” with mass but with no spatial dimensions, and is travelling at a speed much less than the speed of light.

There are two parts to this problem, both to be solved with MATLAB. You can solve them using any numerical/computational technique you like so long as you do them with MATLAB. You must write and use a function M-file called `timeStep` that “evolves” the universe over a small increment of time `DT`. This function and the algorithms that one can use in it are described in some detail below. You will also write a script M-file that will interrogate its user for the following inputs:

- Number of “objects” in your universe (greater or equal to 2),
- Mass of each object in kilograms,
- Starting x and y coordinates of each object in meters,
- Starting x and y velocities of each object in meters/second,
- Time allotted to each timestep, `DT` in seconds, and
- The total number of timesteps to devote to the evolution, `NSteps`.

Note that the magnitude of the velocity, `sqrt(vx^2 + vy^2)` must be much less than the speed of light, `3.0e8` m/s, for the equations described below to work sensibly. So, make sure that the input speed is less than `3.0e6`. The user should be prompted repeatedly for inputs until sensible inputs are provided.

The script M-file will also execute a loop over all the timesteps in the evolution, and provide final graphical output. Examples of the program output will be provided below.

11.10.4 Supplementary material: Trajectories of objects in the universe

The x and y components of acceleration ($a_{i,x}$, $a_{i,y}$) felt by the i 'th object in the universe at position (x_i, y_i) because of gravitational attraction to the other $N - 1$ objects is:

$$a_i^x = \sum_{i \neq j}^N \frac{-Gm_j(x_i - x_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2]^{3/2}}$$

¹Interestingly enough, at the dawn of the 20th century, theoretical physicists actually thought that we could do this. Then, quantum mechanics came along and things got a lot more “uncertain”.

$$a_i^y = \sum_{j \neq i}^N \frac{-Gm_j(y_i - y_j)}{[(x_i - x_j)^2 + (y_i - y_j)^2]^{3/2}}$$

where G is the gravitational constant (6.673×10^{-11} nt-m²/kg²). Note that the sum $\sum_{j \neq i}^N$ excludes the i 'th object, as an object (being point-like) does not exert any gravitational force on itself.

11.10.5 Supplementary material: Stepping algorithms

If we assume that any object feels a constant acceleration over a small time interval, Δt , then we may construct a stepping algorithm as described in the following pseudocode:

- (a) Start with time $t = 0$.

At $t = 0$ the starting conditions:

$x_i(0)$, $y_i(0)$, $v_i^x(0)$, $v_i^y(0)$ for $i = 1, 2, 3 \dots N$

are known.

- (b) Calculate the acceleration at the start of the step for each particle in the system:

$$a_i^x(t) = \sum_{j \neq i}^N \frac{-Gm_j(x_i(t) - x_j(t))}{[(x_i(t) - x_j(t))^2 + (y_i(t) - y_j(t))^2]^{3/2}}$$

$$a_i^y(t) = \sum_{j \neq i}^N \frac{-Gm_j(y_i(t) - y_j(t))}{[(x_i(t) - x_j(t))^2 + (y_i(t) - y_j(t))^2]^{3/2}}$$

- (c) Estimate the positions and velocities at the end of the timestep assuming constant acceleration.

$$x_i(t + \Delta t) = x_i(t) + v_{x,i}(t)\Delta t + \frac{1}{2}a_{x,i}(t)(\Delta t)^2$$

$$y_i(t + \Delta t) = y_i(t) + v_{y,i}(t)\Delta t + \frac{1}{2}a_{y,i}(t)(\Delta t)^2$$

$$v_{x,i}(t + \Delta t) = v_{x,i}(t) + a_{x,i}\Delta t$$

$$v_{y,i}(t + \Delta t) = v_{y,i}(t) + a_{y,i}\Delta t$$

- (d) Increment the time at the end of the timestep, $t = t + \Delta t$.

- (e) If the total time to evolve the universe has been reached, stop. Otherwise go back to step 2.

There are a few subtleties you will have to think about.

- How do you make a reasonable choice for Δt ?

Basically, unless you are prepared to do a lot of extra math and physics to work this out, you have to guess. Leave this up to the user. I'll be happy with an algorithm that works as described above.

- What basic inputs should I start with?

You can dream up your own universe. However, here are some astronomical facts that you can use to set the scale of things:

- G : $6.673 \times 10^{-11} \text{ nt-m}^2/\text{kg}^2$
- The Sun: Mass $1.989 \times 10^{30} \text{ kg}$
- The Earth: Mass $5.9736 \times 10^{24} \text{ kg}$, Earth-Sun orbit radius $149,600,000,000 \text{ m}$
- The Moon: Mass $7.35 \times 10^{22} \text{ kg}$, Moon-Earth orbit radius $384,400,000 \text{ m}$

Velocities? Work these out for yourself assuming that the heavier partner is stationary and the fact that the Earth goes around the Sun once per year and the Moon goes around the Earth once per month. If you want the data on other planets, do a little research. I found the above numbers by surfing the web.

- Is my evolution realistic?

Only in the limit that $\Delta t \rightarrow 0$. You will find that the above equations, the longer you iterate them will increasingly violate the physical laws of the conservation of energy and momentum. There appears to be no satisfactory solution to this problem although there are better algorithms than the one presented above. Discussion of this would take us far beyond the scope of this course.

- How do I check that my algorithm is reasonable?

Here are some suggestions:

- Set $G = 0$ and verify that your objects travel in straight lines.
- The two-object case has an exact mathematical solution. You can use this to check your results. Almost every introductory physics book discusses this solution.

Here is an example of the Earth and Moon executing one orbit around the Sun. In this case, I used a timestep of 3600 seconds and 365.25×24 timesteps.

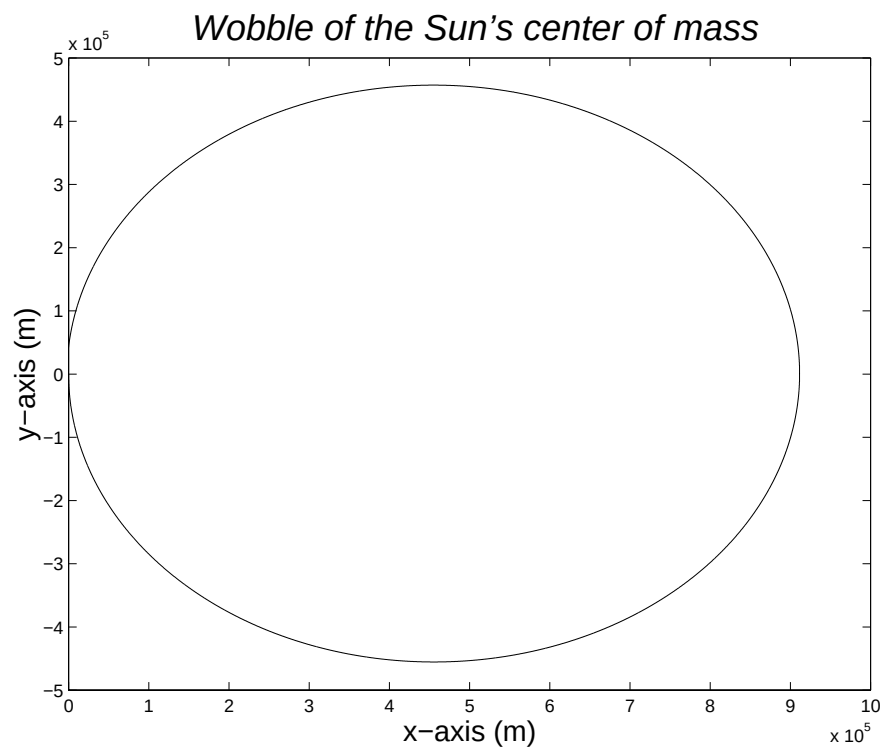


Figure 11.13: This is the wobble of the center of mass of the Sun.

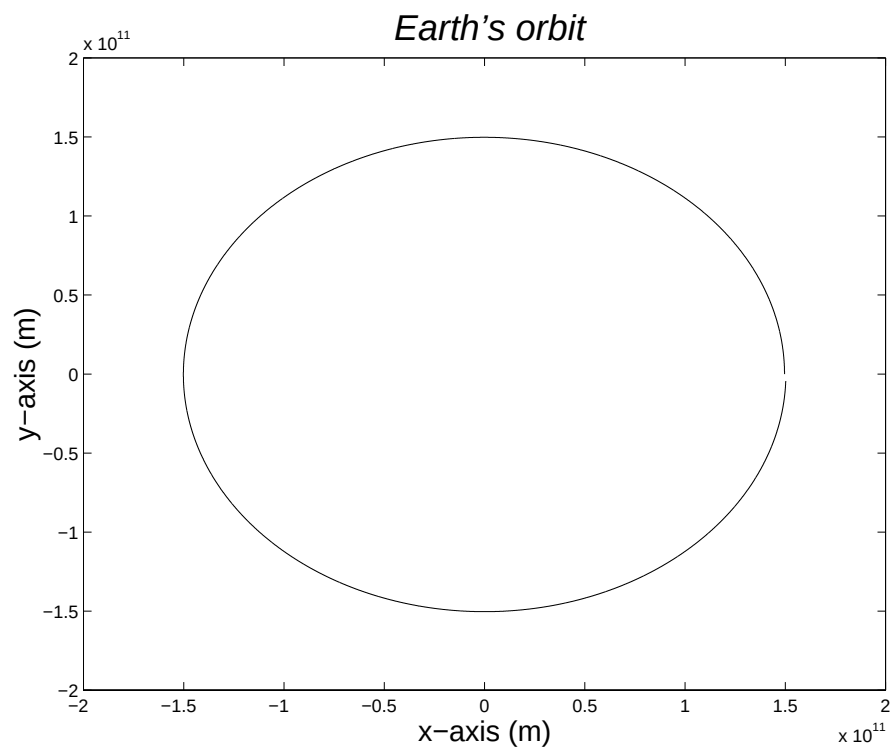


Figure 11.14: This is the Earth's orbit.

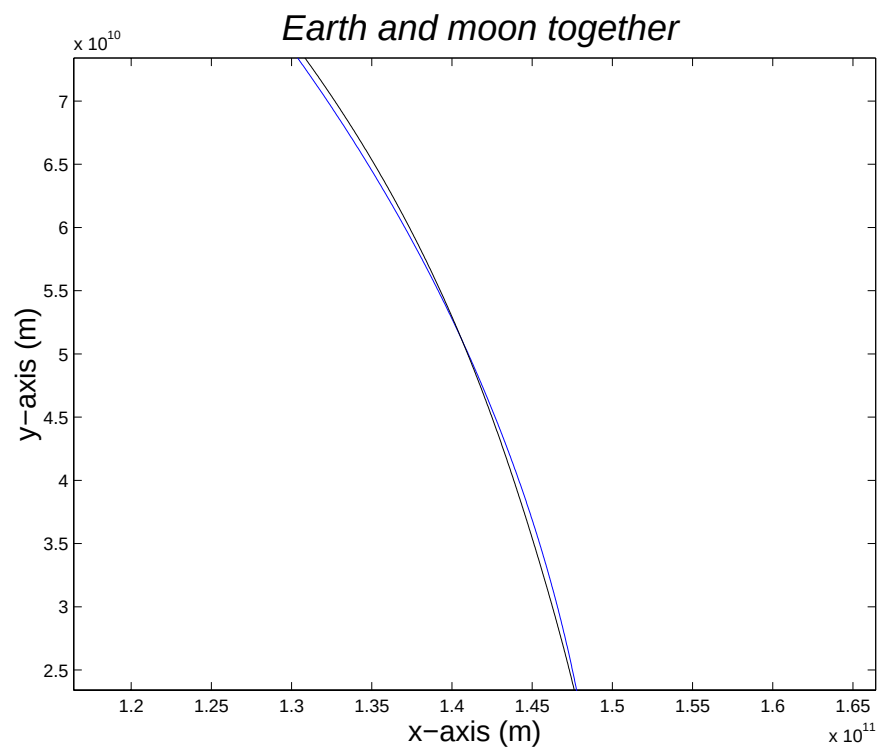


Figure 11.15: This is part of the combined orbit of the Earth and the moon.

8. Re-Mastermind

You will write MATLAB code, a simpler version of the Mastermind program of Chapter 8, one in which you, the user, try to guess the 4 hidden numbers.

You already know the rules of the game. Only, in the case, to make things simpler, the uncovered numbers are represented by '0's rather than '?'s. (Mixing characters and numbers in MATLAB is a bit of a pain.)

And, the game continues until you either quit, or win. Pretty simple, huh?

Hints:

How the computer finds the hidden numbers

```
answer = floor(9*rand(1,4) + 1);
```

Confused? Try:

```
>> help rand  
>> help floor
```

How can you input 4 numbers at once?

```
guess = input('Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: ');
```

The above line prompts the user for input (See next page.) and the user can type in the elements of a row vector.

Confused? Try:

```
>> help input
```

An example play of the game is on the following page.

```

>> MasterMind
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [1,1,1,1]
uncovered =
  0    0    0    0
board =
  1    1    1    1
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [2,2,2,2]
uncovered =
  0    0    2    0
board =
  1    1    1    1
  2    2    2    2
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [3,3,2,3]
uncovered =
  0    0    2    0
board =
  1    1    1    1
  2    2    2    2
  3    3    2    3
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [4,4,2,4]
uncovered =
  0    0    2    0
board =
  1    1    1    1
  2    2    2    2
  3    3    2    3
  4    4    2    4
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [5,5,2,5]
uncovered =
  0    5    2    0
board =
  1    1    1    1
  2    2    2    2
  3    3    2    3
  4    4    2    4
  5    5    2    5
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [6,5,2,6]
uncovered =
  0    5    2    6
board =
  1    1    1    1
  2    2    2    2
  3    3    2    3
  4    4    2    4
  5    5    2    5
  6    5    2    6
Input your guess (1 <= [a,b,c,d] <= 9), [0,0,0,0] to quit: [7,5,2,6]
uncovered =
  7    5    2    6
board =
  1    1    1    1
  2    2    2    2
  3    3    2    3
  4    4    2    4
  5    5    2    5
  6    5    2    6
  7    5    2    6
ans =
You win!

```

9. Can computers cure cancer?

No, they can't. But radiation can kill cancer and computers help. The following project outlines the essential problems of killing cancer with radiation. But first...

Some background

There is about a 50% chance that you will get cancer. You almost certainly know someone in your immediate family who died of cancer. My father died from it 3 years ago. My mother and sister have had it, and, I will most certainly get it if I live long enough. Here are some startling facts:

- In the US alone, there are 1.4 million new cancer patients each year.
- Half of these will get treated with radiation. Cancer treatment using radiation is called *radiotherapy*.
- About 20% of those who are treated will die unnecessarily, even though cancer was well localized. The radiation treatments were not effective enough to cure them. If you do the math, that's 140,000 people per year in the US, about 50% more people than live in Ann Arbor!
- So, if you can make radiotherapy just 1% better, you will save 1,400 people per year.

That's what my research is all about—making radiotherapy better, using computers and computer algorithms. I thought I would give you a flavor of what the main problems are. And, I want to give you a MATLAB project a little more meaningful than Mastermind!

Hints and demonstrations will be given in class.

Here's what you have to do.

- (a) Make a model of a cross section of the human body 30×40 cm that is represented by a 30 row by 40 column array called `body`. This array will represent the energy deposited in the body by the radiation beams.

```
Rows = 30; Cols = 40; body = zeros(Rows, Cols);
```

- (b) Make a working array that pads an extra row or column on the exterior of `body`. This accounts for the outside. `A(2:Rows + 1, 2:Cols + 1) = body;`

```
A(1,:) = 0; A(:,1) = 0; A(Rows + 2, :) = 0; A(:, Cols + 2) = 0;
```

- (c) The tumor is a 4×4 cm square in the center of the body. Use `pcolor` and `plot` to show the body and the outline of the tumor before the radiation is present. Here's a hint:

```
show = A(2:Rows + 2, 2:Cols + 2); %Handles off by one quirk in pcolor
%Some code to scale show so that it's maximum value is 1.
%Hint: help max
```

```
hold on
pcolor(0:Cols,0:Rows,show)
shading flat
plot(...you figure out the arguments...)
```

It should look like this²...

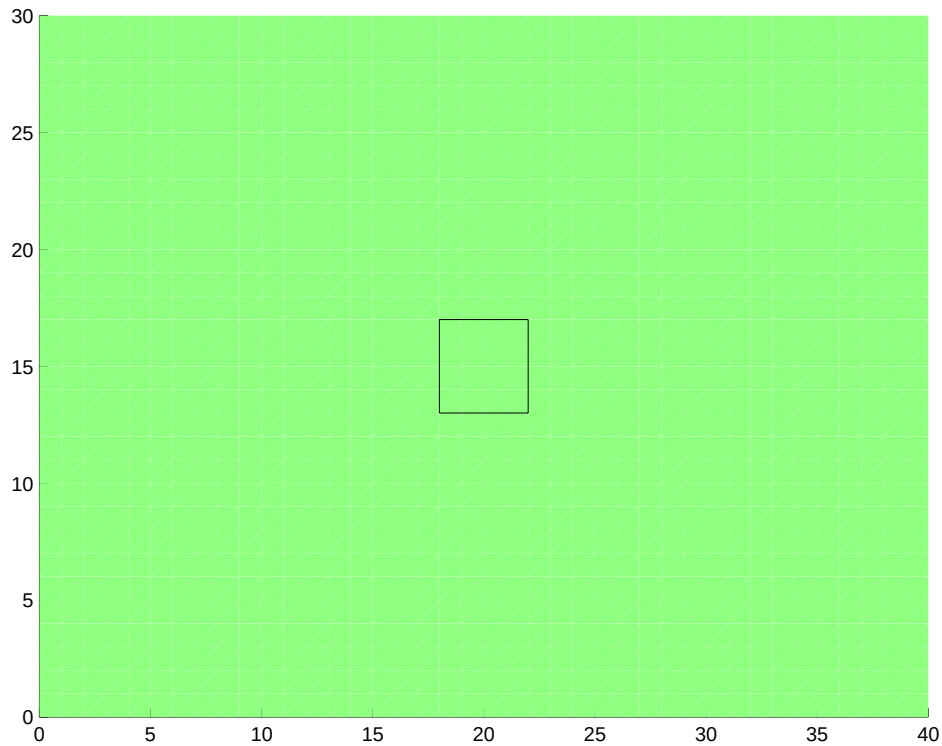


Figure 11.16: Before the radiation

- (d) Give the user of the program the capability of putting up to 4 beams on the patient from the outside, one from the top, one from the bottom, one from the left and one from the right. For example...

```
>> TreatCancer
Place beam top to bottom? (0 = no, 1 = yes) 1
Place beam bottom to top? (0 = no, 1 = yes) 1
Place beam left to right? (0 = no, 1 = yes) 1
Place beam right to left? (0 = no, 1 = yes) 1
```

which should look like this...

²This is much prettier in color!

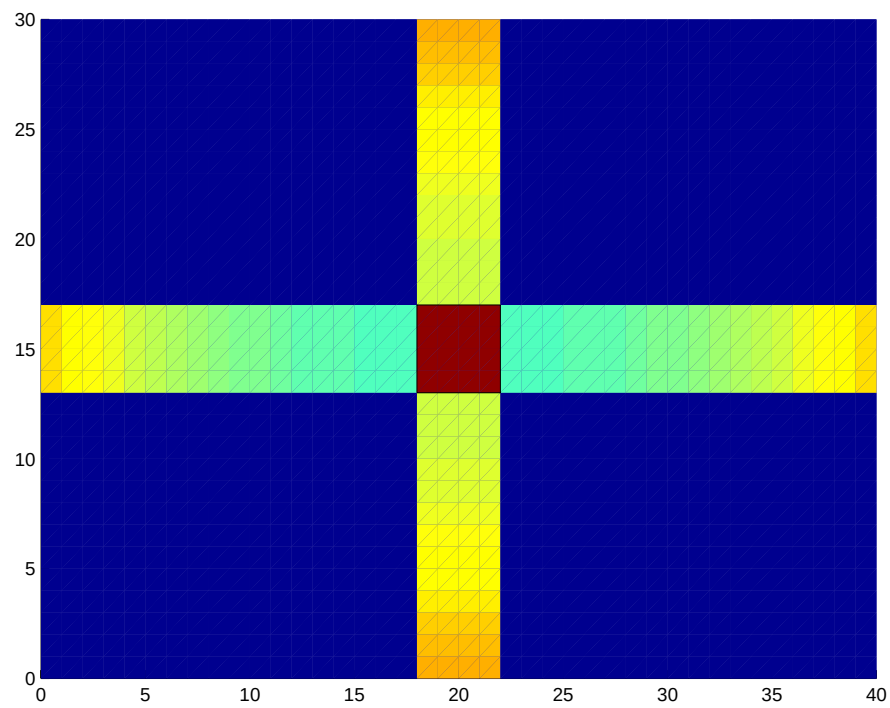


Figure 11.17: After beams are placed.

The beams decay (or attenuate) exponentially from the surface with a decay constant of 15 cm. So, if a surface tile has energy value 1, at depth i , where i represents the depth in centimeters, the value should be $\exp(-i/15)$. Note that the beams should cover the entire extent of the tumor.

- (e) If that was the whole story, cancer would be beaten! Unfortunately, radiation spreads, the tumor does not get a full dose and the healthy tissue can get damaged. A simple model of this spreading is to repeatedly average the energy value as follows:

$$A_{i,j} = (A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})/4$$

Perform this averaging 5 times (perform the operation on the array and then repeat the process 4 more times) to produce a result that looks like the following figure.

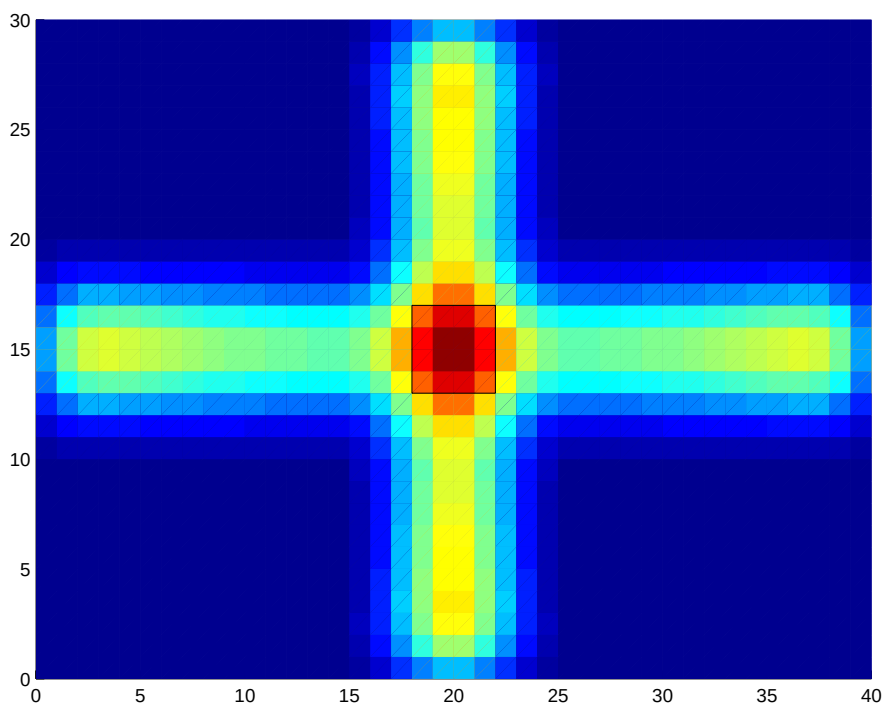


Figure 11.18: What happens after the beams spread.

Open ended questions. (Optional.) Solve these and collect numerous Nobel prizes for medicine and physics!

- Model a real 3-dimensional person rather than the simple geometry above.

- Can you optimize the treatment by using more beams? (The answer is yes. The difficulty is in finding the optimum configuration.)
- What is the best decay constant to use? (This is related to the energy of the beam.)
- Develop the radiation physics to make the above calculation resemble reality.
- Design a better radiation machine that will deliver better beams for radiotherapy.
- Get this calculation done in 5 minutes or less!
- Find out the real reason why radiation kills cells.
- Find out what some radiation is good for you (far in excess of natural radiation exposure). This is called *radiation hormesis*.
- What role does radiation damage play in the development of cancers. This is called *oncogenesis*.

Chapter 12

Graphics

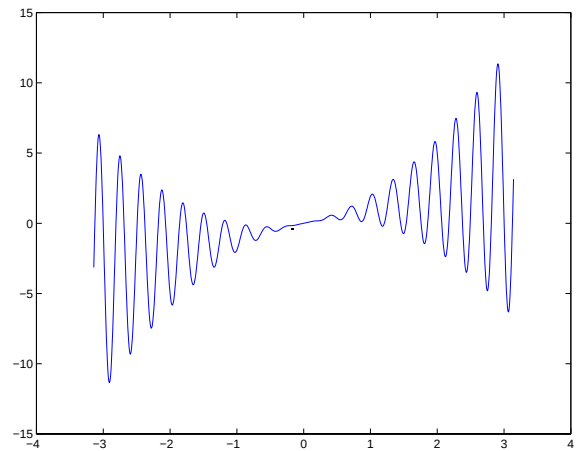
12.1 Two Dimensional Plots

12.1.1 A basic plot

```
% Note the use of pointwise operators

x = linspace(-pi,pi,1000);
y = (x.^2).*sin(20*x) + x;

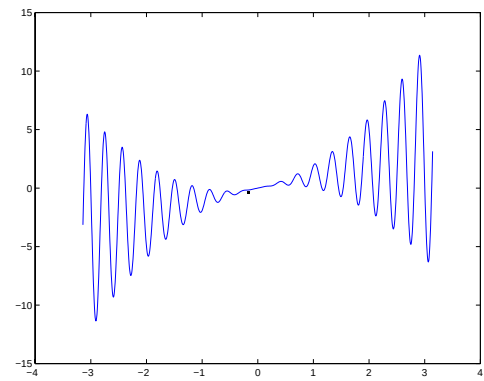
plot(x,y)
pause % Use a "pause" statement to stop
close % Closes the figure
```



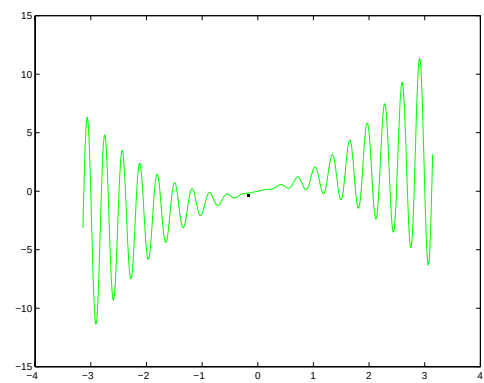
12.1.2 Using different line colors

```
% Choose from:
% b: blue
% g: green
% r: red
% c: cyan
% m: magenta
% y: yellow
% k: black
% w: white
```

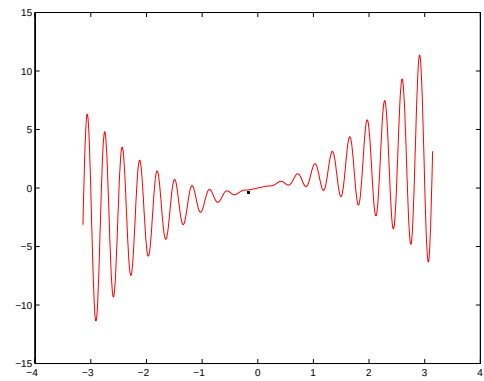
```
plot(x,y,'b') % blue
```



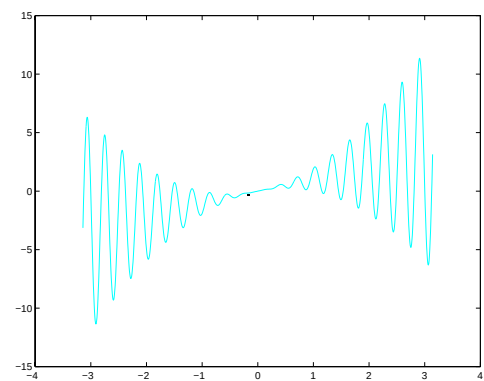
```
plot(x,y,'g') % green
```



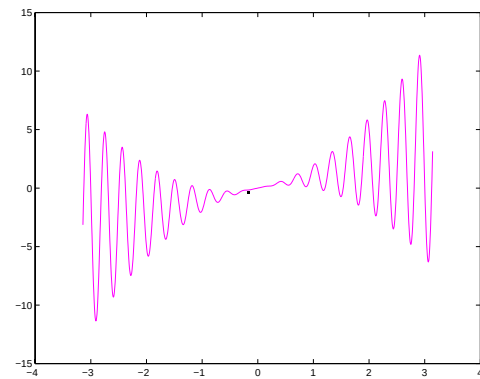
```
plot(x,y,'r') % red
```



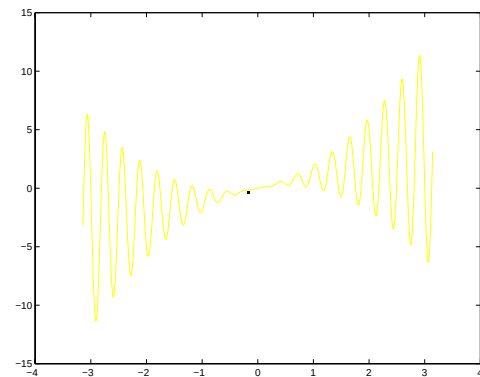
```
plot(x,y,'c') % cyan
```



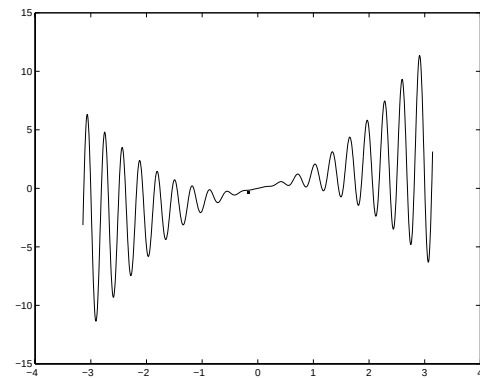
```
plot(x,y,'m') % magenta
```



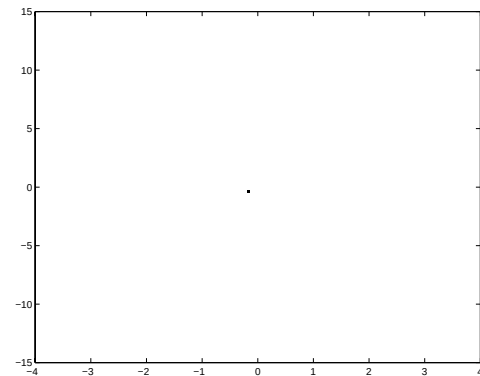
```
plot(x,y,'y') % yellow
```



```
plot(x,y,'k') % black
```

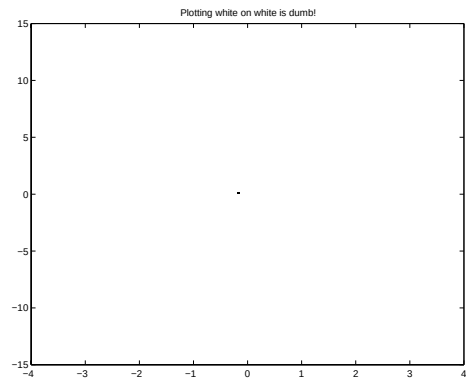


```
plot(x,y,'w') % white  
% White!!!!
```



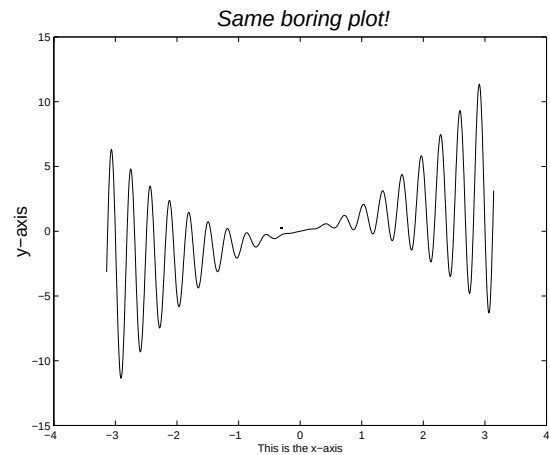
12.1.3 Making titles

```
% Titling your graph
plot(x,y,'w') % white
title('Plotting white on white is dumb!')
```



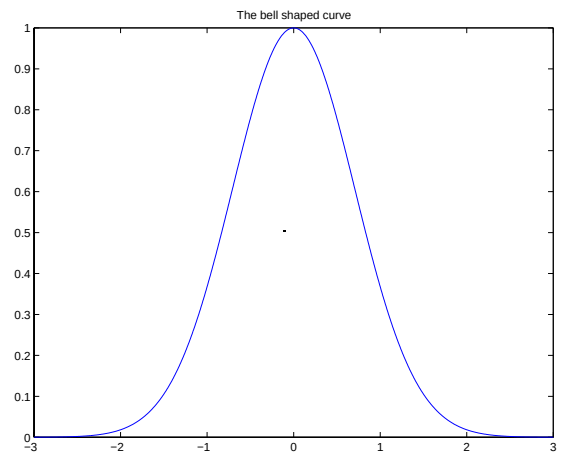
12.1.4 Axis labels

```
% Axis labels
plot(x,y,'k')
xlabel('This is the x-axis');
ylabel('\fontsize{16}y-axis')
title(...
'\fontsize{20}\it Same boring plot!')
```



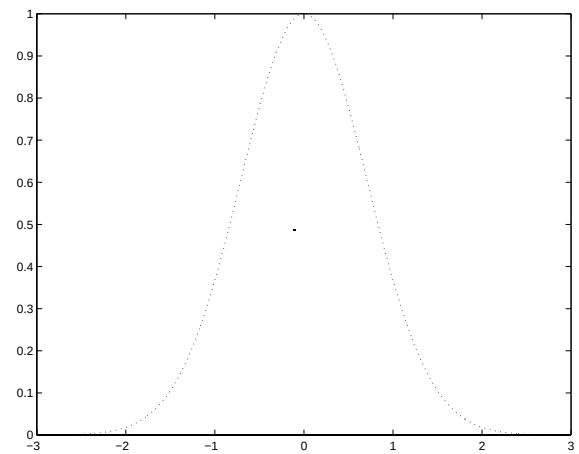
12.1.5 Different line styles

```
% A different 2D example
u = linspace(-3, 3, 1000);
v = exp(-u.^2);
plot(u,v)
title('The bell shaped curve')
```

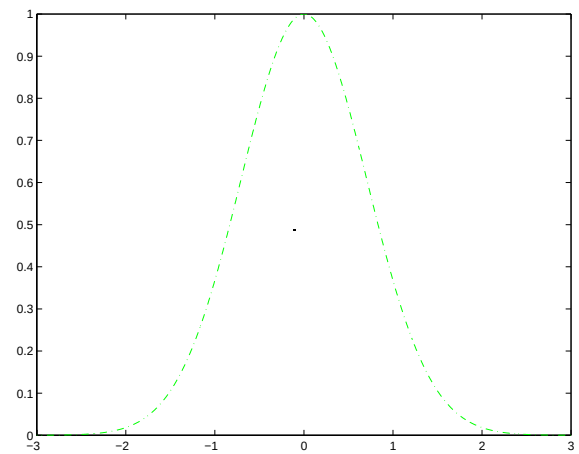


```
% Plotting with different line styles
% Choose from:
% -   solid line (default)
% :   dotted line
% -.  dash-dot line
% --  dashed line
%
% syntax plot(u,v,':')
% Note use of different colors and line styles
```

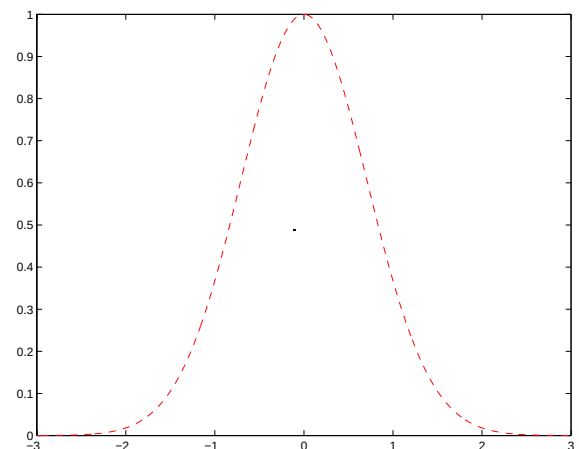
```
plot(u,v,'k:')
```



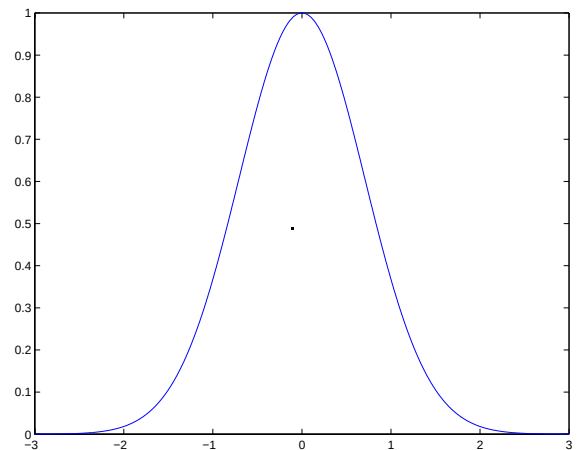
```
plot(u,v,'g-.')
```



```
plot(u,v,'r--')
```

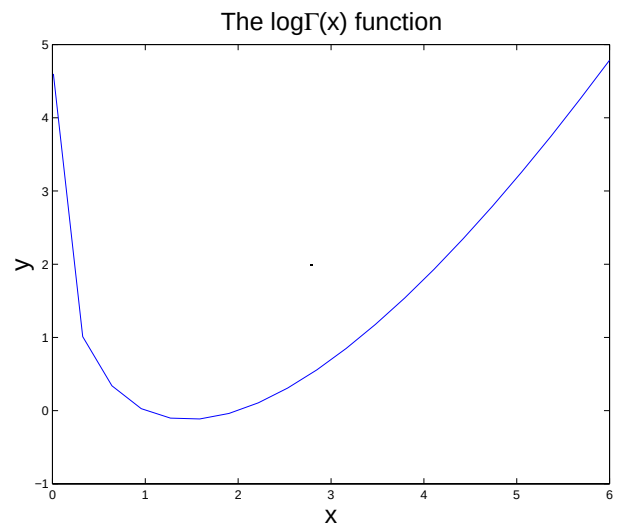


```
plot(u,v,'b-')
```



```
% Example - The LogGamma function
```

```
s = linspace(0.01, 6, 20);
t = log(gamma(s));
plot(s,t)
title(...
'\fontsize{20}The log\Gamma(x) function')
xlabel('\fontsize{20}x')
ylabel('\fontsize{20}y')
```

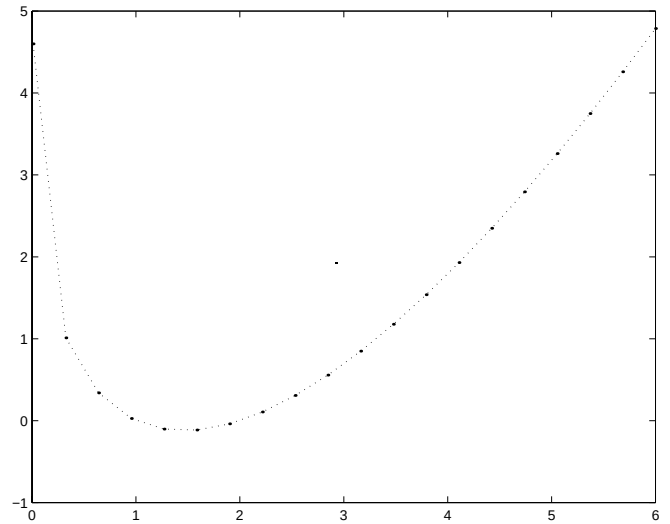


12.1.6 Point plots, plotting with point symbols

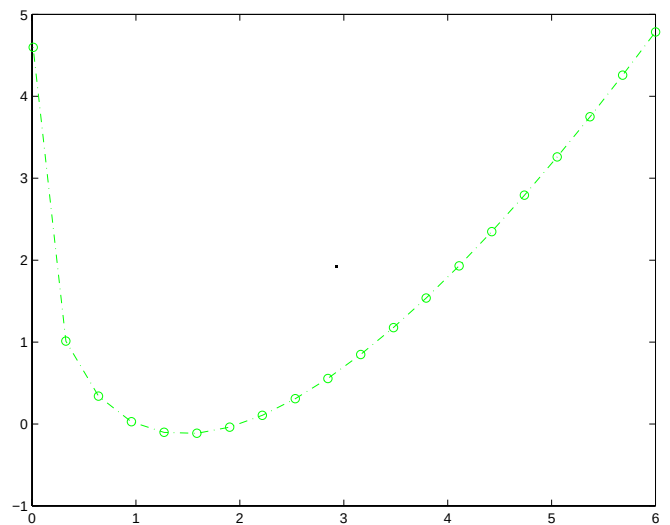
```
% Plotting with point symbols
% Choose from:
% .   point          o   circle
% x   cross          +   plus sign
% *   asterisk       S   square
% d   diamond
% V   triangle down  ^   triangle up
% <   triangle left  >   triangle right
% p   pentagram     h   hexagram
%
% syntax plot(s,t,':')
```


% Note use of different colors and line styles

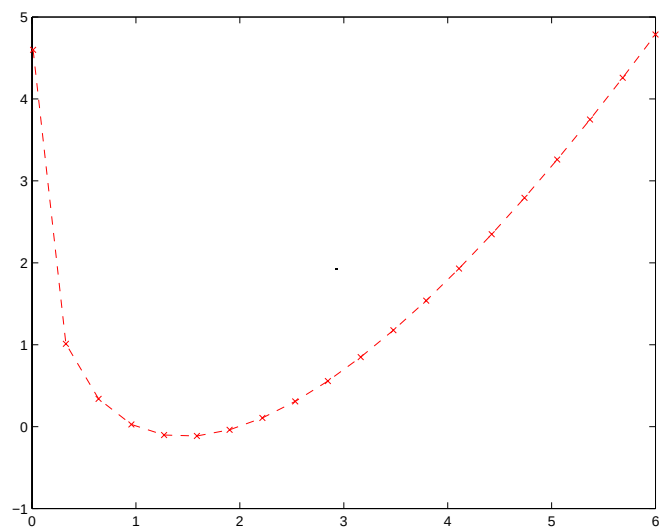
```
plot(s,t,'k:.'
```



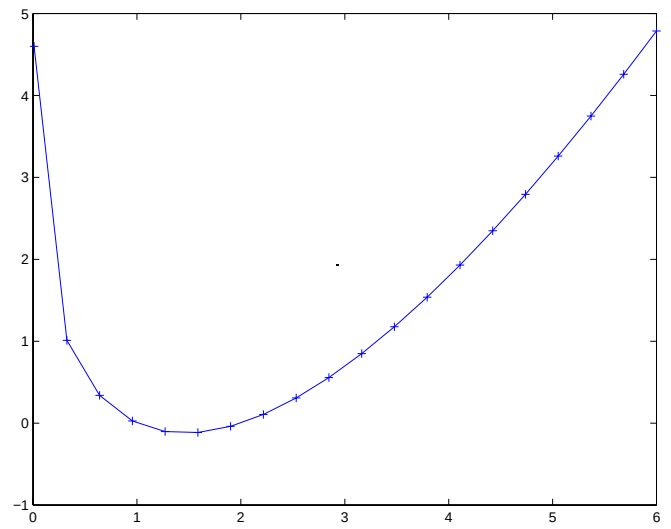
```
plot(s,t,'g-.o')
```



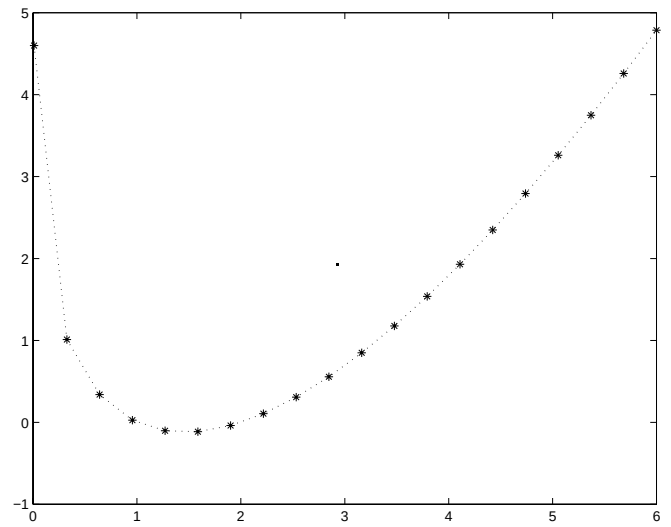
```
plot(s,t,'r--x')
```



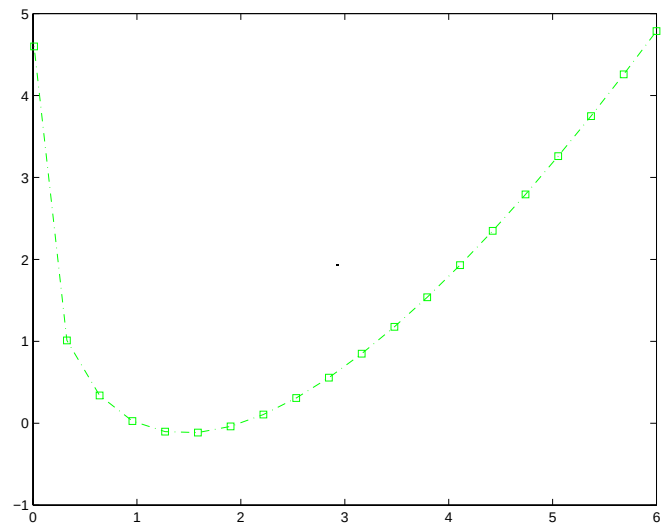
```
plot(s,t,'b-+')
```



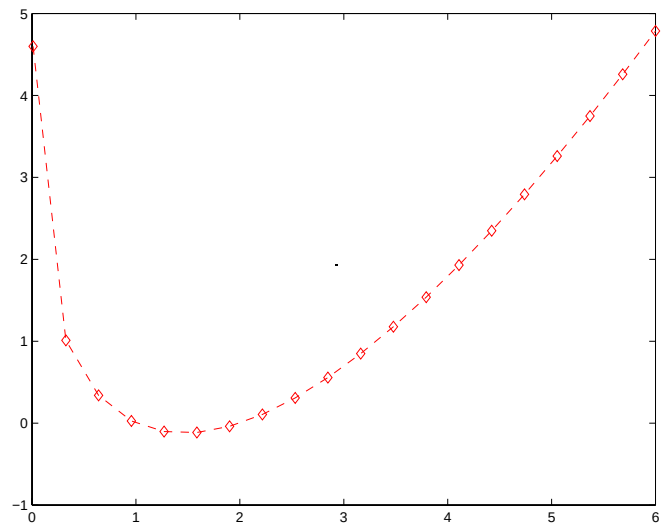
```
plot(s,t,'k:*')
```



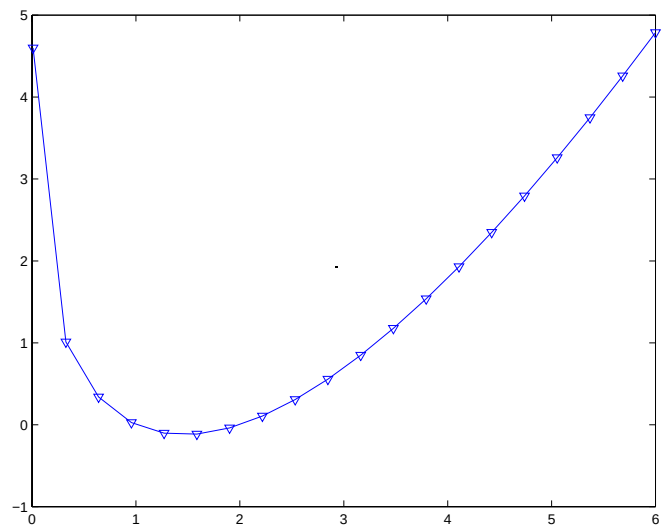
```
plot(s,t,'g-.S')
```



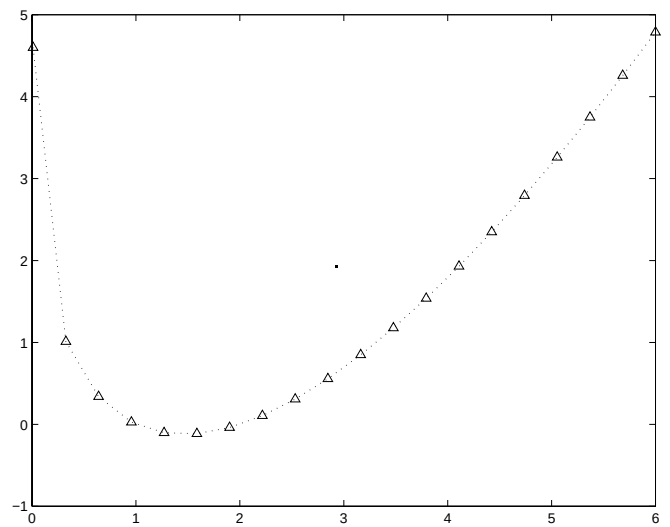
```
plot(s,t,'r--d')
```



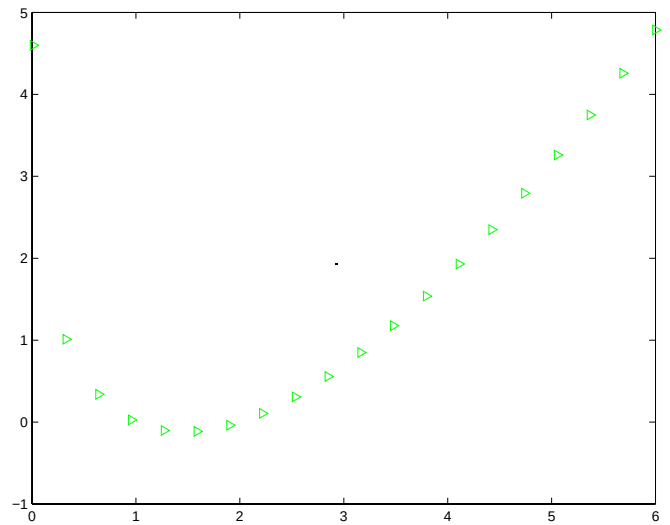
```
plot(s,t,'b-V')
```



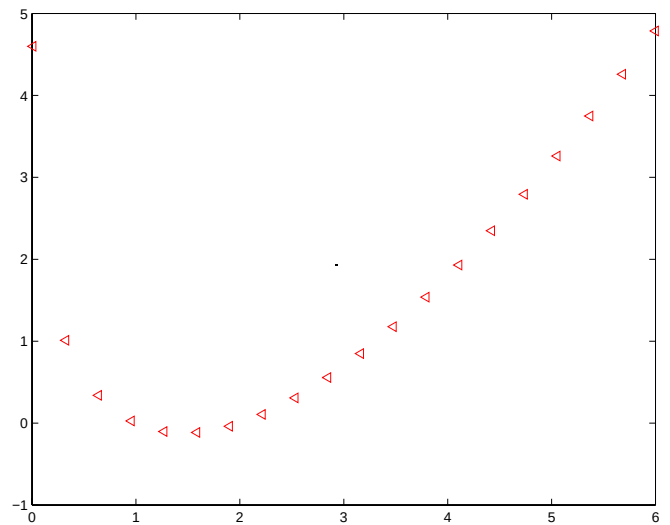
```
plot(s,t,'k:~')
```



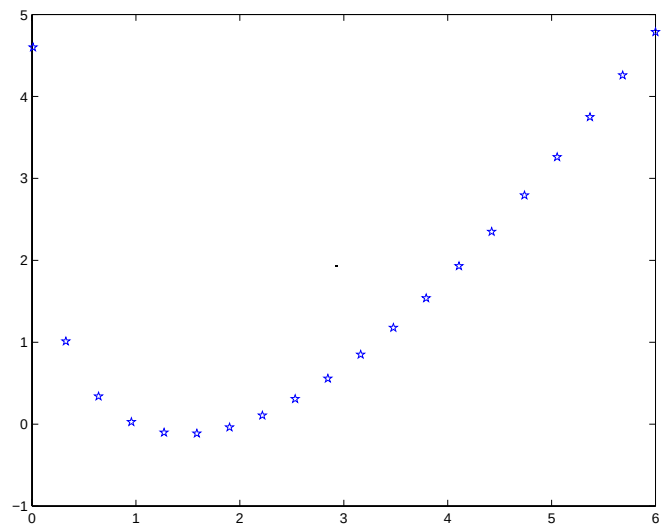
```
plot(s,t,'g>')
```



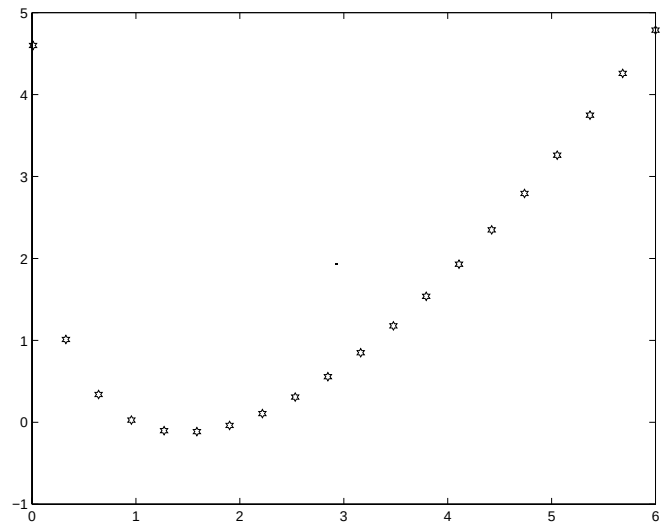
```
plot(s,t,'r<')
```



```
plot(s,t,'bp')
```



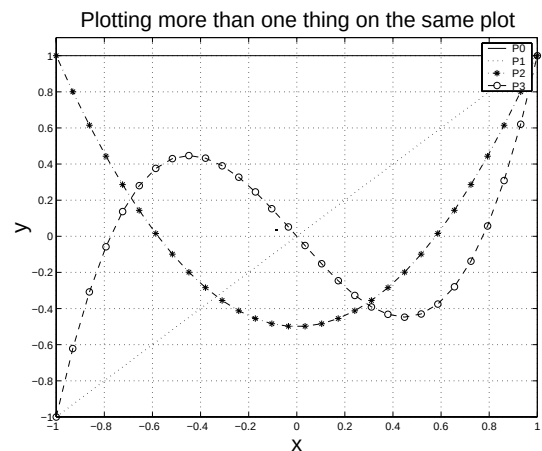
```
plot(s,t,'kh')
```



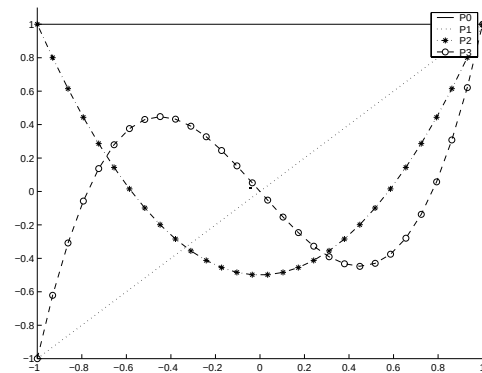
12.1.7 Plotting more than one thing at once

```
% More arguments to the plot function
% Changing axes limits
clear all
x = linspace(-1, 1, 30);
f0 = ones(1,length(x));
f1 = x;
f2 = (1/2)*(3*x.^2 - 1);
f3 = (1/2)*(5*x.^3 - 3*x);

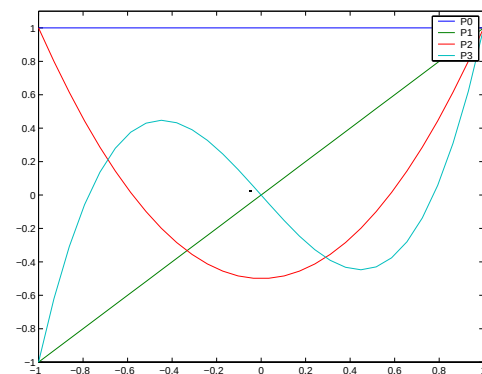
plot(x,f0,'k-',x,f1,'k:',x,f2,'k-.*',x,f3,'k--o')
title('\fontsize{20}Plotting more than one thing on the same plot')
xlabel('\fontsize{20}x')
ylabel('\fontsize{20}y')
axis([-1,1,-1,1.1]) %[xmin,xmax,ymin,ymax]
grid on % Turn the grid on
legend('P0','P1','P2','P3')
```



```
% Another way
hold on % Don't erase when drawing new plot
plot(x,f0,'k-')
plot(x,f1,'k:')
plot(x,f2,'k-.*')
plot(x,f3,'k--o')
axis([-1,1,-1,1.1]) %[xmin,xmax,ymin,ymax]
legend('P0','P1','P2','P3')
```

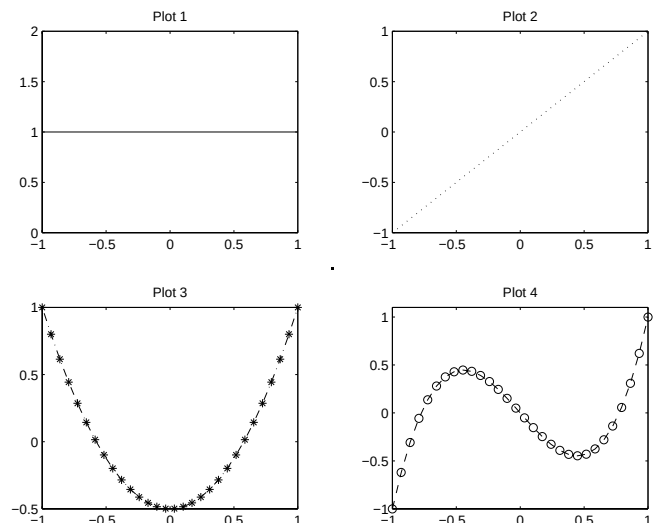


```
% Another way - matrix form
g(1,1:length(x)) = f0;
g(2,1:length(x)) = f1;
g(3,1:length(x)) = f2;
g(4,1:length(x)) = f3;
plot(x,g)
axis([-1,1,-1,1.1]) %[xmin,xmax,ymin,ymax]
legend('P0','P1','P2','P3')
```



12.1.8 Subplots

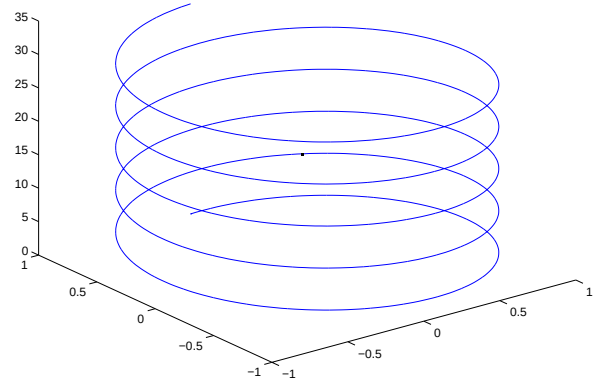
```
% Yet another way, subplots
subplot(2,2,1), plot(x,f0,'k-'), title('Plot 1')
subplot(2,2,2), plot(x,f1,'k:'), title('Plot 2')
subplot(2,2,3), plot(x,f2,'k-.*'), title('Plot 3')
subplot(2,2,4), plot(x,f3,'k--o'), title('Plot 4')
axis([-1,1,-1,1.1]) %[xmin,xmax,ymin,ymax]
zoom on
```



12.2 Three dimensional graphics

12.2.1 Plot3 plots

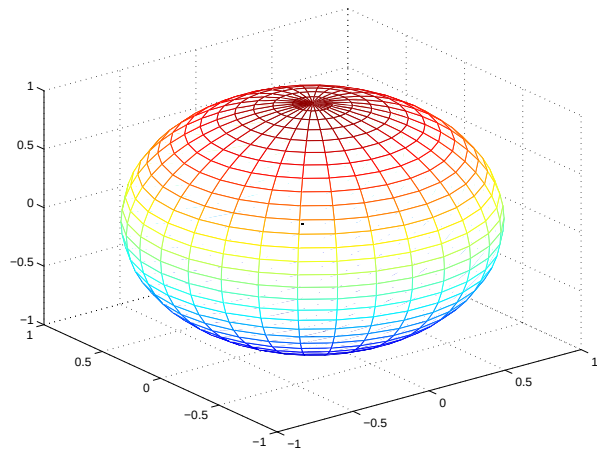
```
% Example using plot3
clear all
t = linspace(0, 10*pi, 1000);
plot3(sin(t),cos(t),t)
```



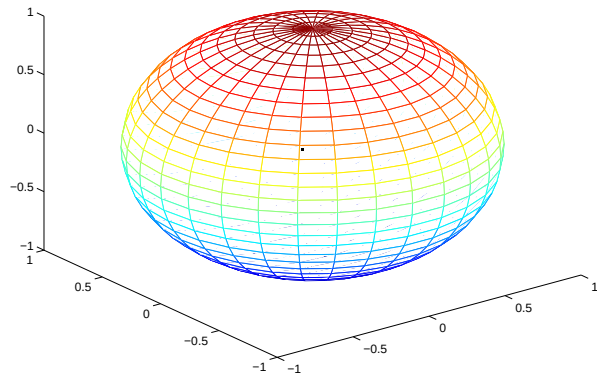
12.2.2 Mesh plots

```
% Mesh plots

[x,y,z] = sphere(30);
mesh(x,y,z)
```

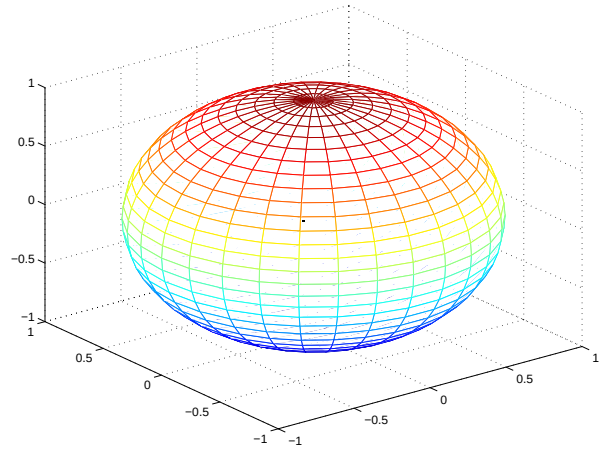


```
grid off % Turn off the grid
```

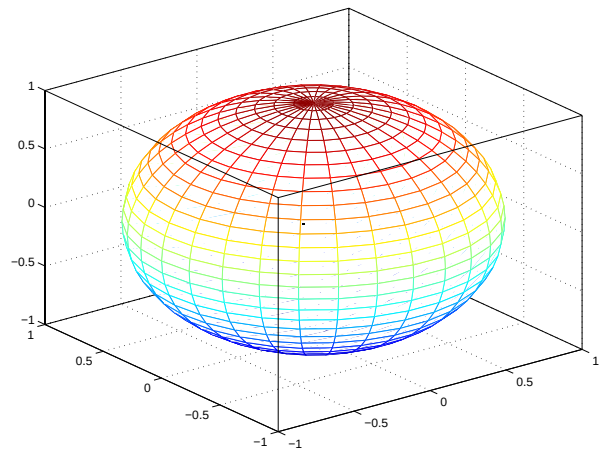


380

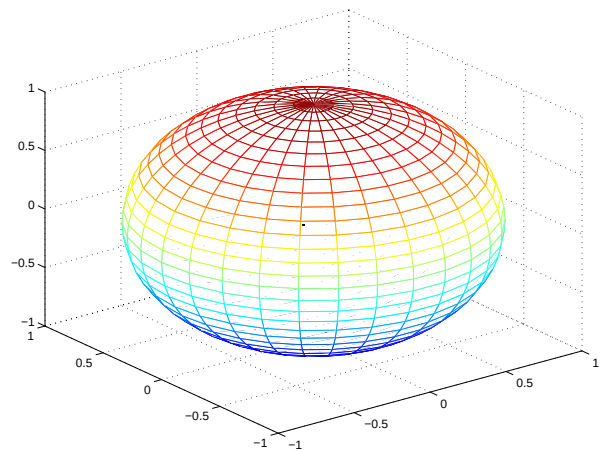
grid on % Turn on the grid



box on % Turn on the box

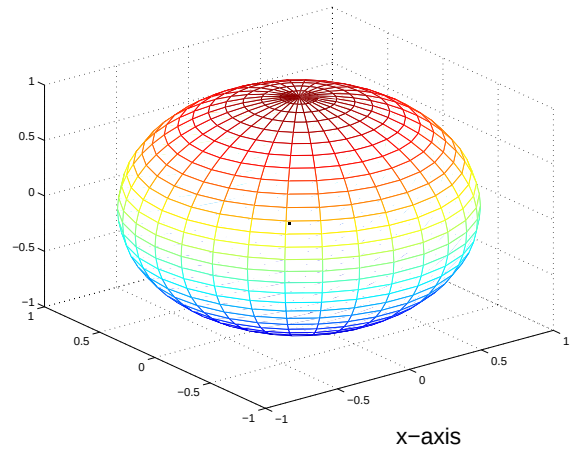


box off % Turn off the box

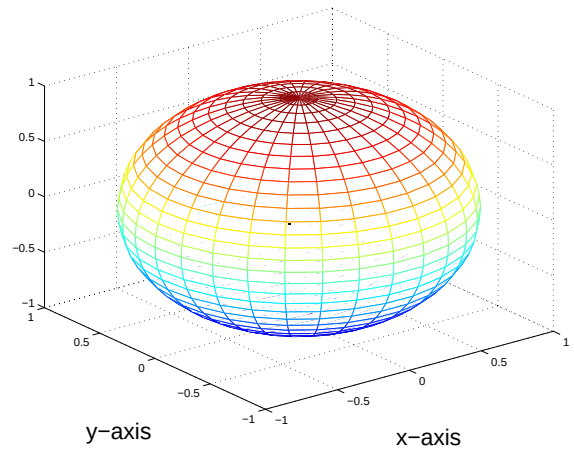


12.2.3 Axis labelling

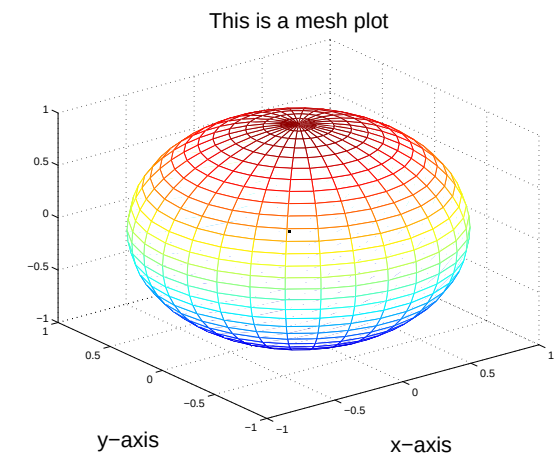
```
% Provide a label for the x-axis
xlabel('\fontsize{20}x-axis')
```



```
% Provide a label for the y-axis
ylabel('\fontsize{20}y-axis')
```

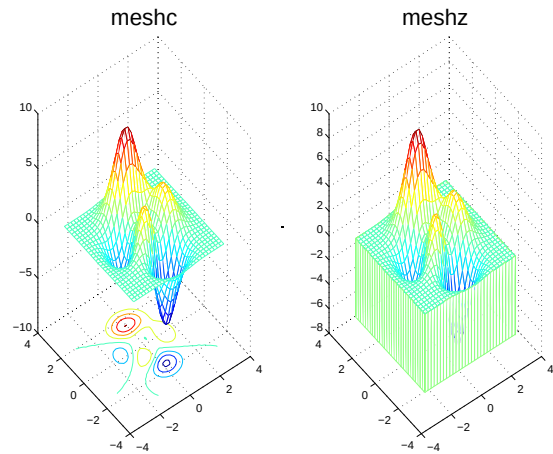


```
% Provide a title
title('\fontsize{20}This is a mesh plot')
```



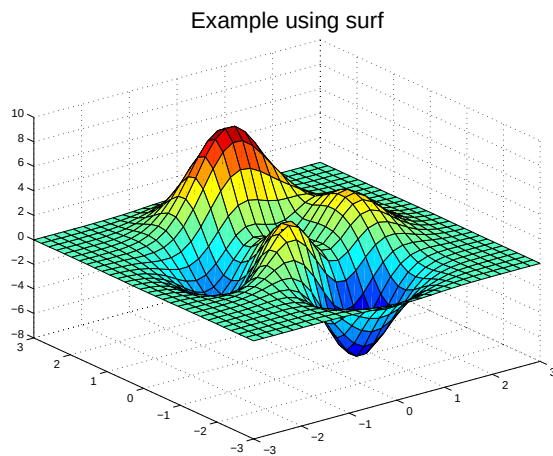
12.2.4 meshc and meshz plots

```
% Two other forms of mesh plots
% meshc and meshz
[x,y,z] = peaks(30);
subplot(1,2,1)
meshc(x,y,z),title('\fontsize{20}meshc')
subplot(1,2,2)
meshz(x,y,z),title('\fontsize{20}meshz')
```

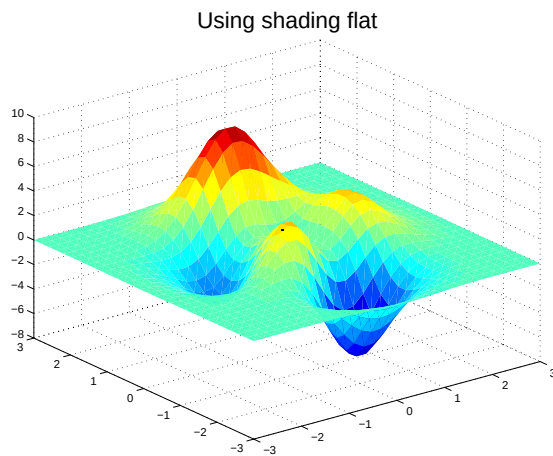


12.2.5 Surface plots using surf

```
[x,y,z] = peaks(30);
surf(x,y,z)
title('\fontsize{20}Example using surf')
```

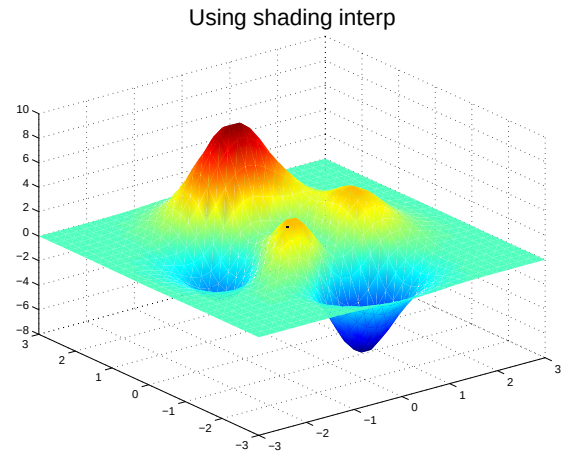


```
shading flat % Flat shading
title('\fontsize{20}Using shading flat')
```



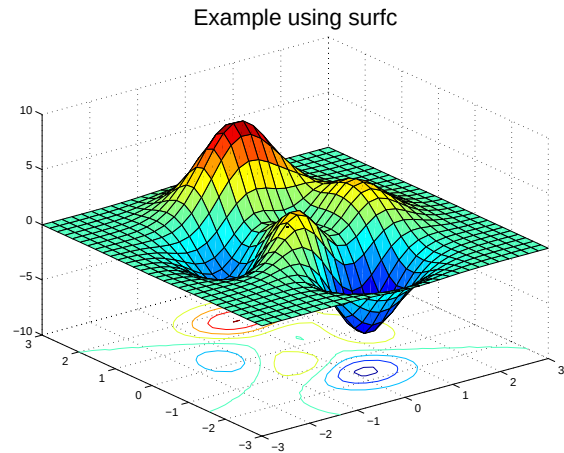
12.2. THREE DIMENSIONAL GRAPHICS

```
shading interp % Interpolated shading
title(...
'\fontsize{20}Using shading interp')
```



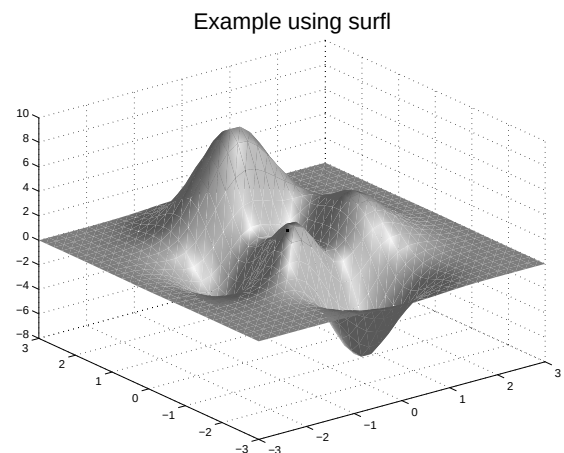
12.2.6 Surface plots using surf

```
surf(x,y,z) % surf with contours
title('\fontsize{20}Example using surf')
```



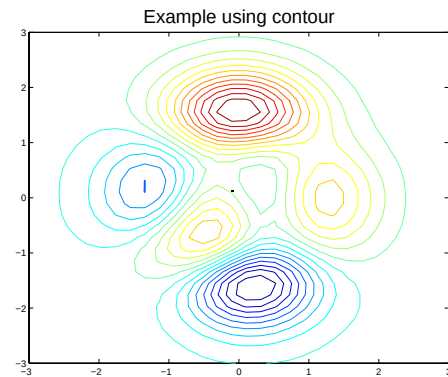
12.2.7 Surface plots using surf1

```
surf1(x,y,z) % surf with lighting
shading interp
colormap gray
title('\fontsize{20}Example using surf1')
```

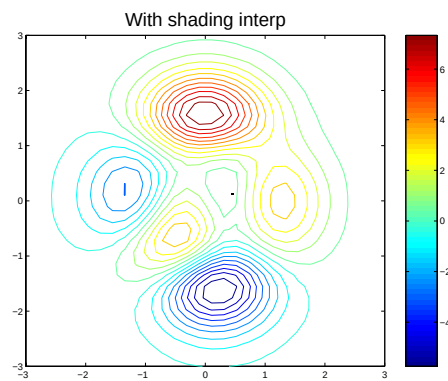


12.2.8 Contour plots using contour

```
[x,y,z] = peaks(30);
contour(x,y,z,20)
title('\fontsize{20}Example using contour')
```

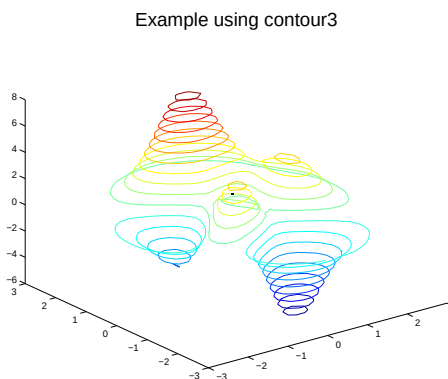


```
colorbar % Show the colorbar
title('\fontsize{20}With shading interp')
```



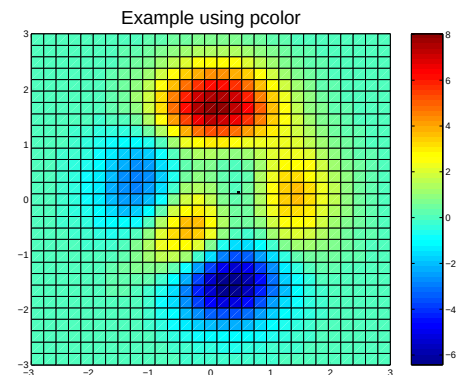
12.2.9 Contour plots using contour3

```
contour3(x,y,z,20)
title('\fontsize{20}Example using contour3')
grid off
```



12.2.10 Contour plots using pcolor

```
% Example using pcolor
pcolor(x,y,z)
colorbar % Show the colorbar
title('\fontsize{20}Example using pcolor')
```



Chapter 13

Miscellaneous topics

13.1 Pitfall review

13.1.1 Review of for loops

This module gives some review on MATLAB's implementation of for loops mostly by way of example.

The general syntax of a for loop is

```
for k = vectorOrColumnList
    % MATLAB statements
end
```

There are several ways of stepping through the elements of a row vector.

Example (Need vector value but not index):

The most common usage is C++-like syntax

```
%File: fex1.m
% For loop requiring vector value but not the index
for k = 1:3
    fprintf('k = %g\n',k)
end
% Prints
k = 1
k = 2
k = 3
```

or

```
%File: fex1a.m
% For loop requiring vector value but not the index
for k = 1:2:8
    fprintf('k = %g\n',k)
end
% Prints
k = 1
k = 3
k = 5
k = 7
```

or

```
%File: fex1b.m
% For loop requiring vector value but not the index
for x = 1.0:-0.1:0.0
    fprintf('x = %g\n',x)
end
% Prints
x = 1
x = 0.9
x = 0.8
x = 0.7
x = 0.6
x = 0.5
x = 0.4
x = 0.3
x = 0.2
x = 0.1
x = 0
```

or

```
%File: fex1c.m
% For loop requiring vector value but not the index
theta = linspace(0,90,3);
for thetai = theta
    fprintf('theta = %g\n',thetai)
end
% Prints
```



```
theta = 0
theta = 45
theta = 90
```

Example (Need vector value and index):

```
%File: fex2.m
% For loop requiring vector value and the index
v = linspace(0,250,3);
for i = 1:length(v)
    fprintf('velocity(%g) = %g\n',i,v(i))
end
% Prints
velocity(1) = 0
velocity(2) = 125
velocity(3) = 250
```

Example (Nested for loops):

```
%File: fex3.m
% Nested for loops
theta = linspace(0,90,2);
v = linspace(0,250,2);
for thetai = theta
    fprintf('theta = %g\n',thetai)
    for i = 1:length(v)
        fprintf('velocity(%g) = %g\n',i,v(i))
    end
end
% Prints
theta = 0
velocity(1) = 0
velocity(2) = 250
theta = 90
velocity(1) = 0
velocity(2) = 250
```

Example (when the index is a column vector):

```
%File: fex4.m
% For loop where the index is a column vector
```

```

a = [1 2 3
     4 5 6];
[rows,columns] = size(a);
fprintf('a is a %g x %g matrix\n',rows,columns)
for b = a
    disp(b);
    disp('-----')
end
% Prints
a is a 2 x 3 matrix
    1
    4
-----
    2
    5
-----
    3
    6
-----

```

Note in this example that **b** is a new column vector on each iteration of the loop.

13.1.2 Review of functions

Some detail is given in Chapter 13 of Austin and Chancogne. However, the definitive reference for this course is given in the Chapter 11 that was handed out and is on the web. This lecture gives some review on MATLAB's implementation of functions mostly by way of example.

Recall:

The general syntax of the function definition is to have a separate file called **functionName.m** that has the following form:

```

function [out1,out2,...,outN] = functionName(in1,in2,...inM)
%FUNCTIONNAME A brief one line description (optional)
% .
% .
% More description (optional)
% .
% .
% .

```

```
% First executable statement follows this line
.
.
.
% Valid executable MATLAB statements and comments
.
.
.
% Last line in functionName, control reverts to calling M-file or command window
```

The function call is made with the following statement:

```
[out1,out2,...,outN] = functionName(in1,in2,...inM)
```

Functions with no input or output arguments

Here's a function called `noEvil.m` that does nothing except printing something silly:

```
function noEvil
%NOEVIL noEvil is a function that does nothing
%
% ENG101 class example

%File: noEvil.m
disp('See no evil')
disp('Hear no evil')
disp('Do no evil')
disp('A function that does absolutely nothing')
```

Functions with one output argument

Here's a function called `square.m`, that computes the square of its argument:

```
function a2 = square(a)
%SQUARE square is a function that squares an argument
%
% It can accept scalars, vectors or matrices
%
% ENG101 class example

%File: square.m
a2 = a.^2;
```

and some examples of its use on a scalar, a vector, a matrix and the element of a matrix:

```
>> square(2)
ans = 4

>> y = linspace(0,10,11)
y = 0    1    2    3    4    5    6    7    8    9    10
>> square(y)
ans = 0    1    4    9   16   25   36   49   64   81   100

>> x = round(10*rand(4))
x =  5   10    3    2
     2    7    4    8
     6    4    9    6
     3    7    7    1

>> square(x)
ans = 25   100    9    4
      4    49   16   64
     36   16   81   36
      9    49   49    1

>> x(3,3) = square(x(3,3))
x =  5   10    3    2
     2    7    4    8
     6    4   81    6
     3    7    7    1

>> z = square(x(2,2))
z = 49
```

Functions with more than one output argument

Here's a function called `squareCube.m`, that computes the square and cube of its argument:

```
function [a2,a3] = squareCube(a)
%SQUARECUBE squareCube is a function that squares and cubes its argument
%
%
% It can accept scalars, vectors or matrices
%
% ENG101 class example
```

```
%File: squareCube.m
a2 = a.^2;
a3 = a.^3;
```

and some examples of its use on a scalar, a vector and a matrix:

```
>> squareCube(2)
ans = 4
>> [u,v] = squareCube(2)
u = 4
v = 8

>> y = linspace(0,10,11)
y = 0    1    2    3    4    5    6    7    8    9   10
>> squareCube(y)
ans = 0    1    4    9   16   25   36   49   64   81  100
>> [u,v] = squareCube(y)
u = 0    1    4    9   16   25   36   49   64   81  100
v = 0    1    8   27   64  125  216  343  512  729 1000

>> x = round(10*rand(4))
x =    10    9    8    9
      2    8    4    7
      6    5    6    2
      5    0    8    4
>> squareCube(x)
ans =   100   81   64   81
       4   64   16   49
       36   25   36    4
       25    0   64   16
>> [u,v] = squareCube(x)
u =   100   81   64   81
      4   64   16   49
      36   25   36    4
      25    0   64   16
v =  1000  729  512  729
      8   512   64  343
     216  125  216    8
     125    0  512   64

>> [a,b] = squareCube(x(2,2))
```

```
a = 64
b = 512
```

Note that if you want to employ all the outputs, you have to assign them, else only the first output is displayed. These functions work on scalars, vectors, matrices and elements of matrices.

Subordinate functions

Let's rewrite `squareCube.m` in the following way:

```
function [a2,a3] = squareCallsCube(a)
% SQUARECALLSCUBE squareCallsCube is a function that .^2's and .^3's its argument
%
% squareCallsCube calls a subordinate function called cube.m
% It can accept scalars, vectors or matrices
%
% ENG101 class example

%File: squareCallsCube.m
a2 = a.^2;
a3 = cube(a);
return

function c = cube(b)
c = b.^3;
```

`squareCallsCube.m` works the same way as if you had called `squareCube.m`. However, the function `cube.m` is only known to the function `squareCallsCube.m`.

```
>> cube(a)
??? Undefined function or variable 'cube'.

>> help cube

cube.m not found.

>>
```

13.2 The MATLAB way

One of the most important things to remember is that MATLAB is organized around the variable type of two-dimensional arrays of doubles. Hence, MATLAB's intrinsic functions are optimized to use this variable type. If you are making use of this aspect of MATLAB, you are doing things “the MATLAB way”.

If you are not (and could be), then your application will almost certainly run slower.

For example, let's consider a temperature diffusion problem. (Examples of a heat diffusion calculation in 1 dimension is given in the example codes `ironbarPointJacobi.m` and `ironbarGaussSeidel.m`. A 2-dimensional example may be found in the example codes `heat.m`, `heatSlow.m`, and `heatFast.m`) The part of the algorithm that takes all the time is:

```
for i = 2:N-1
    for j = 2:N-1
        T(i,j) = 0.25*(oldT(i-1,j) + oldT(i+1,j) + oldT(i,j-1) + oldT(i,j+1));
    end
end
```

Here is a faster solution obtained by using the intrinsic matrix operations of MATLAB. The essential change is made by converting the for loops to

and changing them to

```
T(2:N-1,2:N-1) = 0.25* ...
( ...
    oldT(1:N-2,2:N-1) + oldT(3:N,2:N-1) + oldT(2:N-1,1:N-2) + oldT(2:N-1,3:N) ...
);
```

This will be demonstrated in lecture on a related problem. The speedup is highly dependent on computer architecture. On the laptop the typical speedup is a factor of only about 2 for a 50 by 50 array of temperature. On a high-end workstation, I've obtained as high as 30 times faster.

13.3 Selected Applications

13.3.1 Free fall with air resistance

Let's consider another trajectory problem that we will solve using the stepping method. Consider taking an object at some height, y_0 and dropping it with zero velocity, $v_0 = 0$.

This is a one-dimensional problem and can be solved completely with equations. However, we will execute a computer solution as a model for solving more complex problems, like trajectories of baseballs. The air resistance on a baseball is very significant. If there were no air resistance, the longest home runs (about 600 feet or so) would travel half a mile!

The acceleration on the object is:

$$a = Dv^2 - G \quad (13.1)$$

where D is the drag and G is the acceleration due to gravity.

We can “solve” this in a formal fashion as follows:

$$y(t) = y_0 + v_0 t + \int_0^t dt' \int_0^{t'} dt'' a[v(t'')] \quad (13.2)$$

In fact, even if a were dependent on position y and time t we could write a formal solution as:

$$y(t) = y_0 + v_0 t + \int_0^t dt' \int_0^{t'} dt'' a[y(t''), v(t''), t''] \quad (13.3)$$

But, this is a formal solution only and does not lead to any practical answers, only methods of attack on this problem.

One such attack on the problem is to divide the integral above over a set of time interval Δt and set up an iterative equation as follows.

$$y_i = y_{i-1} + v_{i-1} \Delta t + \int_{t_{i-1}}^{t_{i-1} + \Delta t} dt' \int_{t_{i-1}}^{t'} dt'' a[y(t''), v(t''), t''] \quad (13.4)$$

which is still exact. However, if we make the approximation that the acceleration is constant over the step, that is

$$a_i[y(t''), v(t''), t''] \approx a_i[y(t_{i-1}), v(t_{i-1}), t_{i-1}] \quad (13.5)$$

we get

$$y_i \approx y_{i-1} + v_{i-1} \Delta t + \frac{1}{2} a_i[y(t_{i-1}), v(t_{i-1}), t_{i-1}] \Delta t^2 \quad (13.6)$$

and an algorithm that we can program:

1. Set initial conditions y_0, v_0 at $t = 0$.
2. Initialize a counter $i = 1$.
3. Increment the counter $i = i + 1$
4. Calculate the acceleration $a_i[y(t_{i-1}), v(t_{i-1}), t_{i-1}]$
5. Take a small step $y_i = y_{i-1} + v_{i-1} \Delta t + \frac{1}{2} a_i[y(t_{i-1}), v(t_{i-1}), t_{i-1}] \Delta t^2$

6. Calculate the new velocity $v_i = a_i[y(t_{i-1}), v(t_{i-1}), t_{i-1}]\Delta t$
7. Update the time $t_i = t_{i-1} + \Delta t$
8. Go back to step 3 unless some stopping criterion is reached.

Note that two-dimensional problems are not really more complicated. There would just be more variables to carry around.

Now the central question arises.

How small should I make Δt ????

The answer to THIS question is the art of computer programming and lies at the heart of the general field of “Numerical Methods”. The field of “Numerical Methods” is also concerned with developing better and faster techniques for solving problems than by the “brute force” method outlined above. However, for this problem, we can more or less guarantee that our results are good if the acceleration changes very little over a time step, namely

$$\frac{da}{dt}\Delta t \ll a \quad (13.7)$$

At this point, let’s simplify the discussion and consider just our problem. If we apply the above condition to the problem we get

$$\Delta t \ll \frac{1}{2Dv} \quad (13.8)$$

Does this make sense? It does! If there is no drag, $D = 0$, and we can make Δt anything we like! This is because with no drag there is constant acceleration and we don’t have to make the steps small to get an accurate solution! (We have to be careful that we do not violate our stopping criterion, however. More on this later.)

What if the v is small? In this case we can set Δt to be large because the drag is not affecting the motion very much. If D and/or v are large enough such that Dv is large, we had better use small time steps because drag forces are strongly affecting the trajectory.

How do we know how to set Δt for our problem? The greater the velocity, the greater care we have to take by setting smaller time steps. The greatest velocity we can reach with our problem is called the “terminal” velocity. This is when the drag force and the acceleration force exactly balance each other and there is no net acceleration. For free fall with air resistance, this velocity is

$$v_T = \sqrt{\frac{G}{D}} \quad (13.9)$$

For our baseball the terminal velocity is something like 129 ft/s.

This is the upper limit of the velocity and so we are conservative by saying that

$$\Delta t \ll \frac{1}{2\sqrt{DG}} \quad (13.10)$$

or $\Delta t \ll 2$.

How much smaller than 2 should you make Δt ? Usually, as small as we can get away with! If you want the error to be of the order of 1% or less, you should set it to something below 0.02 seconds. At least we have some indication from a simple analysis.

Now we implement the algorithm with MATLAB coding. Here is what the essential working part of the algorithm would look like:

```
y = 1000; % Starts at 1000 feet
v = 0    ; % with 0 velocity
while (y > 0)
    a = D*v^2 - G;           % Acceleration at beginning of step
    y = y + v*dT + 0.5*a*dT^2; % Position at end of the step
    v = v + a*dT;           % Velocity at end of the step
end
```

Here's what the complete code would look like. It gives some example of organization and plotting:

```
%File: drag.m
clear          % Clears variables
echo off      % Turns off echo
clc           % Clears the screen

G = 32.174;    % Force due to gravity
D = 0.0019285; % Drag on a baseball
dT = 10;      % Initial time step

vT = sqrt(G/D); % Terminal velocity

fprintf('Terminal velocity is %g ft/s\n',vT)

while dT > 0

    dT = input('What delta time to use (0 or less to stop)? >');
    fprintf(' Using dT = %g\n',dT)
    if dT <= 0
        return % dT <= 0 used as an input sentinel
    end
end
```

```

end

y = 1000; % Starts at 1000 feet
v = 0    ; % with 0 velocity

index = 1; % Index into the counting arrays
Y = y; V = v; T = 0; % Plotting arrays

while (y > 0)
    a = D*v^2 - G; % Acceleration at beginning of step
    y = y + v*dT + 0.5*a*dT^2; % Position at end of the step
    v = v + a*dT; % Velocity at end of the step

    % Plotting arrays
    index = index + 1;
    Y(index) = y;
    V(index) = v;
    T(index) = T(index - 1) + dT;
end

subplot(1,2,1)
zoom on
hold on
plot(T,Y,'-')
title('y as a function of t')
xlabel('time (s)')
ylabel('height (ft)')

subplot(1,2,2)
zoom on
hold on
plot(T,V,'-')
title('v as a function of t')
xlabel('time (s)')
ylabel('height (ft/s)')

fprintf('Final velocity = %g\n',V(end));

end

```

When we run this code, as demonstrated in class, we see that the trajectory always “over-

shoots”. We always end up with a negative y at the end. We have to make Δt very small to get the final position close to zero. We can fix this problem relatively easily.

What we can do is check to see if the Δt we are about to use will cause an overshoot and if it will, to truncate it accordingly.

How do we truncate it? Recall that we are assuming that the acceleration over the course of the step is constant. If that is so, we can calculate the Δt required to bring it *exactly* to zero. It must satisfy the quadratic equation:

$$0 = y_{i-1} + v_{i-1}\Delta t + \frac{1}{2}a_i[y(t_{i-1}), v(t_{i-1}), t_{i-1}]\Delta t^2 \quad (13.11)$$

You can solve this with the quadratic equation and convince yourselves that the solution is:

$$dT = -(v + \sqrt{v^2 - 2*y*a})/a;$$

Here is the new working part of the algorithm incorporating this idea. (See the example code `drag0.m`.)

```
y = 1000; % Starts at 1000 feet
v = 0    ; % with 0 velocity

while (y > 0)
    a = D*v^2 - G; % Acceleration at beginning of step
    yTest = y + v*dT + 0.5*a*dT^2; % Position at end of the step
    if yTest < 0
        % Handle the overshoot
        dT = -(v + sqrt(v^2 - 2*y*a))/a;
        y = 0;
    else
        y = yTest;
    end
    v = v + a*dT; % Velocity at end of the step
end
```

So now we have a trajectory that always ends in the right place and an algorithm that is guaranteed to get us close to the right answer. We are now a long way towards solving more complicated problems, like two-dimensional trajectories of baseballs.

13.3.2 Diffusion of charged ions in an electric field

This is the problem of positive and negative ions created somehow (like a pulse of radiation) in a gas drifting in opposite directions under the influence of a constant electric field. Run

the following code called `ion.m` that resides in the `classcodes` area. Note how the leading edge is more populated with ions than the trailing edges since the longer an ion spends in the vicinity of ions of opposite charge the greater the chance that it will “recombine” and neutralize.

```
%File: ion.m
clear all
echo off
clc

NSteps = 100;
NIon = 2000;

for i = 1:NIon
    x1(i) = rand; y1(i) = rand;
    x2(i) = rand; y2(i) = rand;
end

plot(x1,y1,'b*',x2,y2,'r*')
axis([-1 2 0 1])
axis off
title('\fontsize{20}Initial configuration')
pause

for time = 1:NSteps + 3*sqrt(NSteps)
    for i = 1:NIon

        if (rand < 2/NSteps & x1(i) < 1)
            y1(i) = -100;
        end
        theta = 2*pi*rand;
        x1(i) = x1(i) + (1 + cos(theta))*0.5/NSteps;
        y1(i) = y1(i) + sin(theta)*0.5/NSteps;

        if (rand < 2/NSteps & x2(i) > 0)
            y2(i) = -100;
        end
        theta = 2*pi*rand;
        x2(i) = x2(i) - (1 - cos(theta))*0.5/NSteps;
        y2(i) = y2(i) + sin(theta)*0.5/NSteps;

    end
end
```

```

    plot(x1,y1,'b*',x2,y2,'r*')
    axis([-1 2 0 1])
    axis off
    title('\fontsize{20}\leftarrow +ions | -ions\rightarrow')
    pause(0.01)
end

```

13.3.3 Calculation of Electric fields

Constant electric fields as employed in the problem above are really fictional. In this simulation we mock-up a realistic electric field situation. Run the following code called `phi.m` that resides in the `classcodes` area. Note how strong the electric field is near the insulator. (It helps to have come to class to get the full description of this problem!)

```

%File: phi.m
clear all
echo off
clc

N = input('Number of cells = N x N. Input N: ');
delta = input('Convergence criterion delta: ');

V = zeros(N);
V(1,N/2:N) = 1000;
V(N,N/2:N) = 1000;
V(:,N) = 1000;

oldV = V;

change = 10;

tic

counter = 0;
while(change > delta)
    V(2:N-1,2:N-1) = ...
        0.25*( ...
            V(1:N-2,2:N-1) + V(3:N,2:N-1) + V(2:N-1,1:N-2) + V(2:N-1,3:N) ...
        );
    difference = abs(V - oldV);

```

```

    change = sum(sum(difference))/(N^2);
    counter = counter + 1;
    if mod(counter,100) == 0
        fprintf('Iteration %g, average change = %g\n',counter,change)
    end
    oldV = V;
end

toc

pcolor(1:N,1:N,V);
shading interp;
xlabel('\fontsize{20}X-grid');
ylabel('\fontsize{20}Y-grid');
title('\fontsize{20}Voltage in the chamber');

pause

figure

[Ex,Ey] = gradient(V);
E = sqrt(Ex.^2 + Ey.^2);
warning off
pcolor(1:N,1:N,log(E));
shading interp;
xlabel('\fontsize{20}X-grid');
ylabel('\fontsize{20}Y-grid');
title(...
'\fontsize{20}Mag. of electric field in the chamber');

pause

figure

surf(1:N,1:N,log(E));
shading interp;
xlabel('\fontsize{20}X-grid');
ylabel('\fontsize{20}Y-grid');
title(...
'\fontsize{20}Mag. of electric field in the chamber');

```

13.4 Summary of the Course

By now you are conversant in two languages, C++ and MATLAB. You have noted the differences between them. C++ is hard to make work properly. It is a powerful, general-purpose language that runs numerical calculations very fast.

MATLAB is specialized for engineering-type problems. It has lots of engineering-type utility built in and has very nice plotting capabilities, particularly in 3 dimensions. It is easy to get an application “up and running”, but it is relatively slow for large-scale problems.

Not every computer language is suitable for every purpose. Here’s how a research scientist would use both of these computer languages to his or her best advantage.

1. Design a computer algorithm in pseudo-code
2. Convert it to MATLAB
3. Debug the concept of the algorithm using MATLAB
4. Develop graphics output to interpret results
5. Develop a set of test problems that can be completed by MATLAB in a reasonable time.
6. When the MATLAB version is acceptable, re-code the algorithm in C++ starting with the MATLAB code.
7. Run the same set of test problems using the C++ code.
8. Make sure that the C++ results and the MATLAB results of test problems are identical.
9. When the C++ code passes the tests, increase array size or introduce sufficient complexity to solve the problem you really want to solve.
10. Have C++ write data arrays that MATLAB can read in an display, using MATLAB’s excellent graphics capabilities.
11. Publish results!

The upshot is that both ways of doing it has its strengths and weaknesses. Why not use both?

Chapter 14

Programming Style Guide for C++

This programming style guide is a set of mandatory requirements for C++ code layout. These stylistic “laws” are not made for arbitrary reasons. Experience teaches us that conforming to this style, more or less standard through the industry with few exceptions, makes code easy to maintain and allows it to be read less painfully by others. This style guide is mandatory for ENG101, and can only be relaxed under specific instructions from GSIs and Profs.

This style guide is summarized by example:

- Indent 3 spaces relative to the preceding for every new nesting level. Curly brackets must always align vertically. Example,

```
int main(void)
{
    int i = 0, k = 1;
    if (i < k)
    {   int temp = i;
        i = k;
        k = temp;
        if (0 == k)
        {   i = i*i;
            k = -i;
        }
    }
    return 0;
}
```

- Variable and function names are in lowercase, terse but descriptive, enough information conveyed within the programming context, with capital letters to start new words internal to the identifier. Example,

```
numStuds = 211;
degToRad = 180/PI;
```

Don't over do it! Bad example,

```
numberOfStudentsInTheENG101Setion200Class = 211;
conversionFactorForDegreesToRadians = 180/PI;
```

The first encounter with a variable merits a short comment unless the meaning is completely obvious. Example,

```
numStuds = 211; //Number of students in the ENG101 Section 200 class
degToRad = 180/PI; //Conversion factor for degrees to radians.
```

- Never use underscores in function and variable names. Bad example,

```
this_variable_name_is_difficult_to_type = 211;
cute_____but_dumb = true;
```

- Constant names should be in uppercase. Leading underscores are forbidden. Underscores in the middle are acceptable although discouraged. Example,

```
const float PI = 3.14159;
const float EULER_CONSTANT = 0.577216;
```

- A variable should be initialized near where it is first used and should only be defined within the scope for which it is relevant. Examples,

```
const int SIZE = 10;
float x[SIZE], y[SIZE];
for (int i = 0; i < SIZE; i = i + 1) // i only needed with scope of for loop
{
    x[i] = 0;
    y[i] = 0;
}
```

```
float z[SIZE] = {0};
int i; // i needed outside scope of for loop
for (i = 0; i < rand()%SIZE; i = i + 1) z[i] = 1;
cout << "The first " << i << " elements of y were set to one.\n";
```

- There are spaces before and after the binary operators =, +, -, ==, !=, <, <=, >, >=, && and ||. Examples,

```
float result = x + y - z; //Good
float result=x+y-z; //Bad
```

- There are no spaces before and after the binary operators `*`, `/`, and `%`. There is no space following the unary operators `-` and `!`. Examples,

```
float result = x*y + z; //Good
bool decision = ! go; //Bad
```

- No hardwired constants!

Bad example:

```
int a[10][10];
for (i = 0; i == 9; i = i + 1)
    for (j = 0; j == 9; j = j + 1)
        a[i][j] = 1;
```

Good example:

```
const int SIZE = 10;
int a[SIZE][SIZE];
for (i = 0; i < SIZE; i = i + 1)
    for (j = 0; j < SIZE; j = j + 1)
        a[i][j] = 1;
```

- Functions should be small enough so that they fit on the screen. That's about 30 lines of code.
- The use of `goto`, `break` and `continue` is disallowed.
- Use of non-constant global variables is discouraged.

C++ Code layout

Every C++ file, including `main()` is laid out as follows:

- Header comment block
- `#include` statements
- Constants

- Global variables
- Functions, with own headers. Function prototypes are optional unless required by ordering constraints.

Here is an example of code layout:

```
// HEADER COMMENTS GO AT THE TOP *****

/*****
Copyright (C): University of Michigan
Project:      ENG101/200 - Class example
File:        C++example.cpp
Purpose:     Coding example to show code layout
Compiler:    g++
Programmer:  Alex Bielajew
Start Date:  01/18/01
*****/

// INCLUDE statements *****
#include <iostream>

using namespace std;

// CONSTANTS *****

// No constants in this example

// GLOBAL VARIABLES *****

// No global variables in this example

// FUNCTIONS *****

float pow(float x, int n)

/*****
Purpose:  Calculate the integral power of a float using the algorithm
          of al-Kashi
Receives: "x" which is to be raised to the power "n"
Returns:  a float, the result of x raised to the power n
*****/

{ float result; // The result of the calculation

  //An implementation of the al-Kashi algorithm

  if (0 > n)
  { n = -n;
```

```
    x = 1/x;
}

result = 1;
while(1 <= n)
{   if (0 == n % 2)
    {   n = n/2;
        x = x*x;
    }
    else
    {   n = n - 1;
        result = result*x;
    }
}
return result;
}
```

```

int main(void)

/*****
Purpose: To prompt the user for a float x and and int n and call a
        function that computes x to the power of n
*****/

{ cout << "A calculation of x (float) to the power of n (int) \n";

    float x; // The float to be raised to a power
    int    n; // Power to be raised to
    cout << "Input x (float) and n (int): ";
    cin >> x >> n;

    float result; // The result of the calculation

    result = pow(x,n);

    cout << x << " to the power " << n << " = " << result << "\n";

    return(0);
}

```


Chapter 15

Syntax reference for beginning C++

4 Bit Computer Arithmetic

BINARY	HEX	UNSIGNED	SIGNED	BINARY ARITHMETIC (BASE 2)	i (integer), ~i (bit-complement of the integer)
0000	0	0	0	0000 + ? = ? + 0000 = ?	The integer 5 in 4 bit binary is 0101
0001	1	1	1	0001 + 0001 = 0010	The complement of 5 is ~5 = 1010
0010	2	2	2	0010 + 0001 = 0001 + 0010 = 0011	5 + ~5 = 0101 + 1010 = 1111
0011	3	3	3	0011 + 0001 = 0100	5 + ~5 + 1 = 1111 + 0001 = 0000
0100	4	4	4	0010 + 0010 = 0100	-i = ~i (bit-complement of i) + 1
0101	5	5	5	0001 * ? = ? * 0001 = ?	-5 = ~5 + 1
0110	6	6	6	0010 * 0011 = 0110	1011 = 1010 + 0001
0111	7	7	7	0011 * 0011 = 1001	
1000	8	8	-8		
1001	9	9	-7		
1010	A	10	-6		
1011	B	11	-5		
1100	C	12	-4		
1101	D	13	-3		
1110	E	14	-2		
1111	F	15	-1		

32 Bit Computer Arithmetic

BINARY	HEX	UNSIGNED	SIGNED	i =	00000000000000000000000000000001
00000000000000000000000000000000	00000000	0	0	~i =	1111111111111111111111111111101110
00000000000000000000000000000001	00000001	1	1		
00000000000000000000000000000010	00000002	2	2	i	000000000000000000000000000010001
...	...	...	...	+ ~i = +	111111111111111111111111111101110
000000000000000000000000000001111	0000000F	15	15	----	-----
0000000000000000000000000000010000	00000010	16	16		111111111111111111111111111111111
0000000000000000000000000000010001	00000011	17	17		
...	...	...	...		
011111111111111111111111111111111111	7FFFFFFF	2,147,483,647	2,147,483,647		
100000000000000000000000000000000000	80000000	2,147,483,648	-2,147,483,648		
...	...	...	...		
111111111111111111111111111111111110	FFFFFFFE	4,294,967,294	-2		
111111111111111111111111111111111111	FFFFFFF	4,294,967,295	-1		

Algorithms and Pseudocodes

An Algorithm is a set of simple (i.e. well-defined) instructions to be carried out in sequence to accomplish some task. It has a start and a stop and can execute branches and loops.

A Pseudocode is a way of expressing the actions and ordering to be taken place on a computer in an exact yet informal way, without details or syntax of a particular computer language.

e.g. **IF** *such-and-such* **THEN DO**
 this-and-that
 OTHERWISE DO
 something-else

Order of Precedence

Logical Expressions

OPERATOR	ASSOCIATIVITY	
()	left to right	
* / %	left to right	
+ -	left to right	
< <= > >=	left to right	T && T = T, T && F = F, F && T = F, F && F = F
== !=	left to right	T T = T, T F = T, F T = T, F F = F
&&	left to right	!T = F, !F = T
	left to right	False is the same as 0, True is anything else
=	right to left	

If/Else

```
if(logical expression 1)
{ STATEMENT BODY 1
}

else if(logical expression 2)
{ STATEMENT BODY 2
}

// more else if's if needed

else
{ STATEMENT BODY N
}
```

- **else if's** are optional
- **else** is optional independent of the existence of **else if's**
- If there is only one statement in the **statement body**, you *may* ignore the {}, but for clarity it is better to keep them

Loops

While

```
while(logical expression)
{ STATEMENT BODY
}
```

- If the **logical expression** is False the first time, the **statement body** is never executed
- If the **logical expression** is always True, then you have an infinite loop

Do/While

```
do
{ STATEMENT BODY
}while(logical expression);
```

- The **statement body** is evaluated once before the **logical expression** is evaluated at all
- Do NOT forget the ; at the end

For

```
for(expression 1; logical expression; expression2)
{ STATEMENT BODY
}
```

- **expression 1** is optional
- **logical expression** is optional
- **expression 2** is optional
- ; are not optional
- **for(;;){}** is an infinite loop

Example Programs

CODE	OUTPUT	CODE	OUTPUT
#include<iostream>	1	#include<iostream>	1

using namespace std;	2	using namespace std;	2
int main(void)	3	int main(void)	3
{ int i;	4	{ int j = 0;	4
for(i = 1; i <= 10; i = i + 1)	5	do	5
cout << i << "\n";	6	{ j = j + 1;	6
	7	cout << j << "\n";	7
return 0;	8	}while (j <= 10);	8
}	9	cout << j << "\n";	9
	10	return 0;	10
		}	11
			11

```

#include<iostream>
using namespace std;
int main(void)
{ int i = 2,j = -2;
  while(j <= i)
  { if(j < 0 )
    { cout << "i is " << i << ". "
      << "j is " << j << ".\n";
      i = i - 1;
    }
    j = j + 1;
  }
  return 0;
}

```

Chapter 16

Syntax reference for more advanced C++

Addresses

The address of a variable is its location in memory—typically some number from between 0 and $2^{32} - 1$.

The address of a variable is obtained with the unary `&` operator.

The piece of code `{int i; cout << &i << "\n";}` should be read as “i is declared an `int`. Print the address of i in hexadecimal format.”.

Functions, call by value, call by reference

There are **two** important things to remember for functions: 1) the function definition and 2) the function call:

The function definition is written at the top of the C++ file outside of `main` or any other function. A function definition must be given before any code that uses the function via a function call. Generic syntax:

```
[void|int|float|...] functionName(variable type & name list separated by ",") {C++ code}
```

Examples: `int function1(int a, float b){int c;...;return c;}`
`void function2(int& c, float& d){...;return;}`

Note: Variable types and names in parameters lists of function definitions MUST BE INCLUDED.

If the return value is **NOT** `void`, the function must **return** the correct variable type.

If the return value **IS** `void`, the **return;** is **OPTIONAL**.

Functions can “return” at most **ONE VARIABLE**.

The function call is made inside of `main` or another function. Generic syntax:

```
[variable =] functionName(variable name list separated by ",")
```

Example: `int a,c;float b;...;c = function1(a, b);`

Note: Variable names in parameters lists of function calls MUST BE INCLUDED.

Variable types in parameters lists of function calls MUST NOT BE INCLUDED.

Function call by value means that the parameter list of the function definition contains values. Example:

```
float function1(int c, float d){return d/c;} // function definition
x = function1(a, b); // function call
```

Note: Copies of these values are made available to the function. These are lost when the function returns.

Function call by reference means that the parameter list of the function definition contains addresses. Example:

```
function2(int& c, float& d){d = d/c;} // function definition
function2(a, b); // function call
```

Note: Functions called by reference can change the contents of the addresses that are made available to the function in the reference list.

Vectors

Using vectors

```
#include <vector> // Must be included in the pre-processor area
```

Note: Vectors are a “class” defined in standard C++

Declaring `vector<int> vectorName(10);` // An int vector called vectorName of size 10;

Note: Vectors are initialized to 0 when they are declared.

Accessing vector elements

```
for(i = 0; i < 10; i = i + 1){vectorName[i] = i + 1}; // for loop assigns all elements of vectorName
i = vectorName[0]; // i is assigned 1st element of vector vectorName
vectorName[9] = i; // The last (10th) element of vector vectorName is assigned the value of i
```

Note the “off-by-one” syntax in the above three examples

Passing vector to functions

```
double function1(vector<double> b){...} // Typical definition for call-by-value
vector<double> a(10);...;x = function1(a); // Typical call-by-value
double function2(vector<double>& d){...} // Typical definition for call-by-reference
vector<double> c(10);...;x = function2(c); // Typical call-by-reference
```

Vector class functions

```
vector<double> a(10);cout << a.size(); // Prints the size of the a vector
vector<float> b(10);b.pop_back(); // Reduces the size of the b vector by one to size 9
vector<int> c(10); int d = 42; c.push_back(d); // Increases the size of the c vector by one to size 11,
// with value of int d in the last vector element
```

Two dimensional (2D) arrays

Declaring `int arrayName[2][3];` // Array called arrayName of size 2 rows by 3 columns

Declaring and initializing

```
int arrayName[2][3] = {{1,2,3},{4,5,6}}; // Initialize all the elements of arrayName;
int arrayName[2][3] = {{1,2,3}}; // Initialize the first row of arrayName, rest to zero;
```

```
int arrayName[2][3] = {0}; // Initialize all the elements of arrayName to zero;
```

Accessing array elements

```
// for loops below assign all elements of arrayName
for(i = 0; i < 2; i = i + 1) for(j = 0; j < 3; j = j + 1) arrayName[i][j] = 3*i + j;
i = arrayName[0][2]; // i is assigned 1st row/3rd column element of array arrayName
arrayName[1][1] = i; // The last 2nd row/2nd column element of array arrayName is assigned the value of i
```

Note the “off-by-one” syntax in the above three examples

Note: arrayName is the address of arrayName[0][0], that is, arrayName is the same as &arrayName[0][0]

Passing 2D arrays to functions Remember that an array name is the address of the start of the array

```
// Note that column dimension must be specified below
void function(double b[][3], int rowSize, int columnSize){...} // Typical definition
double a[2][3];...;function(a); // Typical call
```

Note that the column size (second index) must be known by the function!

Characters and strings

char's are one-byte integers, strings are arrays of chars that includes a special ending character.

Declaring a character char ch; // Declares a character;

Declaring and initializing a character

```
char ch = 'a'; // Declares a character called ch and initializes it to 'a'. Note single quotes.
```

Declaring a string string name; // Declares a string

Declaring and initializing a character string

```
string name = "Albert Einstein"; // Declares a string
```

Note the use of double quotes in assigning strings.

Printing strings

```
string name = "Barry Manilow"; cout << name;
```

Structures

are compound data types. Structures have to be: 1) defined, 2) declared, 3) initialized, 4) used.

Defining a structure

```
struct AgeAndHeight {int age; float height;}; // Structure tag AgeAndHeight with an int and a float;
```

Declaring a struct

```
struct AgeAndHeight Wilma; // Declares Wilma as an AgeAndHeight struct
```

Declaring and initializing a character

```
struct AgeAndHeight Wilma = {26,65.5}; // Declares Wilma as an AgeAndHeight struct and initializes it
```

Referencing struct elements

```
Wilma.age = 27; // Wilma's age is assigned the value 27  
cout << "Wilma's height is: " << Wilma.height << endl; // Prints Wilma's height
```

Struct vectors

```
vector<struct AgeAndHeight> theFlintstones(4); // A vector of struct's of the AgeAndHeight type  
cout << theFlintstones[0].height; // Prints the height of 1st element
```

Struct arrays

```
struct AgeAndHeight townOfBedrock[100][5]; // A array of struct's of the AgeAndHeight type
```

Struct vectors and arrays are passed to functions in the same way as the standard variable types.

Recursion

Functions can call themselves. Factorial example:

```
int factorial(int n)  
{ if (0 == n) return 1;  
  else return (n * factorial(n - 1));  
}
```

The guidelines are:

- Successive calls to functions should represent progressively simpler cases. (For example, $(N - 1)!$ is “simpler” than $N!$)
- There is an end to the recursion, the “base case”, that does not involve a recursive call. (For example $0! = 1$) Not doing this will lead to a program or computer crash.
- If a function calls itself more than once, and the “nesting level” gets deep, the calculation can be very inefficient.

Chapter 17

Syntax reference for MATLAB

MATLAB Workspace and Variables

The basic variable type in MATLAB is a two-dimensional array of doubles (64-bit representation).

A *scalar* is a 1×1 array.

A *row vector* of length N is a $1 \times N$ array.

A *column vector* of length M is an $M \times 1$ array.

A *matrix* of dimensions M rows and N columns is an $M \times N$ array.

Variable name conventions

MATLAB is case sensitive

Starts with A-Z, a-z

Up to 31 letters, digits and underscores

Default MATLAB variables

<code>ans</code>	result of last unassigned calculation
<code>eps</code>	smallest number that can be added to 1.0 and still be different
<code>Inf</code>	infinity <i>e.g.</i> $1/0 = \text{Inf}$
<code>NaN</code>	Not a Number <i>e.g.</i> $0/0 = \text{NaN}$
<code>pi</code>	Value of Pi (3.1415...)
<code>realmax</code>	Largest real number MATLAB can represent
<code>realmin</code>	Smallest real number MATLAB can represent

User defined variables

<code>clear</code>	clear all or selected variables (or functions) from the current workspace
<code>length</code>	length of a vector or maximum dimension of an array
<code>size</code>	display dimensions of a particular array
<code>who</code>	display current workspace variable names
<code>whos</code>	display current workspace variable names, types and associated sizes

Command/Figure Window control

<code>clc</code>	clear the command window
<code>clf</code>	clear the figure window
<code>figure</code>	start a new figure window
<code>figure(N)</code>	Make figure with index <code>N</code> active. If <code>N</code> is an integer and <code>figure(N)</code> does not exist, create it
<code>close</code>	close current figure window
<code>close(N)</code>	close figure with index <code>N</code>

Useful Workspace functions

<code>help</code>	obtain help generally or for a specific function
<code>lookfor</code>	obtain one-line help if it exists
<code>more</code>	toggles pagination, useful for longs “helps”
<code>load</code>	read variables in from a disk file
<code>save</code>	save all or selected variables to a disk file

MATLAB path and environment

<code>cd</code>	change to a specific directory
<code>dir</code>	list files in the current directory
<code>path</code>	display or modify the function search path

MATLAB punctuation

- . decimal point usage: 325/100, 3.25 and .325e1 are all the same
- ... three or more decimal points at the end of a line cause the following line to be a continuation
- , comma is used to separate matrix subscripts and arguments to functions, also used to separate statements in multi-statement lines
- ; used inside brackets to indicate the ends of the rows of a matrix, also used after an expression or statement to suppress printing
- % begins comments
- ' Quote. 'ANY TEXT' is a vector whose components are the ASCII codes for the characters. A quote within the text is indicated by two quotes. For example: 'Don"t forget.'

Assignment

Individual elements with a row can be delimited by a comma or a space.

Explicit assignment using ;'s to end rows

```
a = [1,2,3;4,5,6;7,8,9]
```

Explicit assignment using "newline" to end rows

```
a = [1,2,3
4,5,6
7,8,9]
```

Explicit assignment using continuation lines

```
b = [1 2 3 4 5 6 ...
7 8 9 10]
```

Useful ways to initialize variables

```
linspace(a,b,N)
```

	Linearly spaced intervals between a and b (inclusive) comprised of N points
<code>zeros(m,n)</code>	An m by n array of zeroes
<code>zeros(n)</code>	An n by n array of zeroes
<code>ones(m,n)</code>	An m by n array of ones
<code>ones(n)</code>	An n by n array of ones
<code>eye(m,n)</code>	An m by n array with ones on the diagonal
<code>eye(n)</code>	An n by n identity matrix
<code>ones(n)</code>	An n by n array of ones
<code>rand(m,n)</code>	An m by n array of random numbers
<code>rand(n)</code>	An n by n array of random numbers

Vector/Matrix operators

These are Vector and Matrix operators

<code>+</code>	addition
<code>-</code>	subtraction
<code>*</code>	multiplication
<code>/</code>	left division
<code>\</code>	right division
<code>^</code>	exponentiation
<code>'</code>	transpose

Point-by-point operators

These operate on matrix elements in point-wise fashion

<code>.*</code>	point-wise multiplication
<code>./</code>	point-wise left division
<code>.\</code>	point-wise right division
<code>.^</code>	point-wise exponentiation

Logical operators

<code><</code>	less than
<code><=</code>	less than or equal
<code>></code>	greater than

<code>>=</code>	greater than or equal
<code>==</code>	equal
<code>~=</code>	not equal
<code>&</code>	logical AND
<code> </code>	logical OR
<code>~</code>	logical NOT

List generation/variable indexing

<code>i:k:l</code>	List generation syntax = 1stValue:Stride:LatValue
<code>v(1)</code>	1st element of vector <code>v</code>
<code>v(end)</code>	Last element of vector <code>v</code>
<code>v(1:2:9)</code>	1st, 3rd, 5th, 7th, 9th elements of vector <code>v</code>
<code>v(2:3:9)</code>	2nd, 5th, 8th elements of vector <code>v</code>
<code>a(2,3)</code>	2nd row, 3rd column of matrix <code>a</code>
<code>a(:,3)</code>	all elements in column 3
<code>a(1,:)</code>	all elements in row 1
<code>a(1:2:end,:)</code>	all odd rows of matrix <code>a</code>
<code>a(1:2,2:4)</code>	sub-matrix of rows 1 and 2, columns 2 through 4
<code>a(1,end)</code>	last element in 1'st row

Script M-files

Sequences of MATLAB commands can be stored in text files with the extension `.m`. The commands can be executed by typing the name of the files (without the extension) or through the file management tools provided by the Command Window menu.

Useful M-file functions

Function M-files

Function definition

Define a separate file called `functionName.m` with the following form:

```
function [out1,...,outN] = functionName(in1,...inM)
%FUNCTIONNAME A brief one line description (optional)
```

```
% .  
% .  
% More description (optional)  
% .  
% .  
% .  
% First executable statement follows this line  
.  
.  
.  
% Valid executable MATLAB statements and comments  
.  
.  
.  
% Last line in functionName
```

Function call

The function call is made with the following statement:

```
[out1,out2,...,outN] = functionName(in1,in2,...inM)
```

Useful M-file functions

<code>disp</code>	display a string
<code>fprintf</code>	write data to screen or file
<code>echo</code>	toggle command echo
<code>error</code>	display message and abort
<code>input</code>	prompt for input
<code>keyboard</code>	transfer control to keyboard
<code>pause</code>	wait for time or user response
<code>return</code>	return to caller
<code>warning</code>	display warning messages

Performance monitoring

<code>tic,toc</code>	stopwatch timer functions
<code>flops</code>	counts floating point operations

Formatting

```
format short    Scaled fixed point format with 5 digits
format long     Scaled fixed point format with 15 digits
format compact  Suppress extra line-feeds
format loose    Puts extra line-feeds in the output
```

Program Flow control

for loops

```
for k = vectorOrColumnList
    % MATLAB statements
end
```

while loops

```
while logicalExpression
    % MATLAB statements
end
```

if/elseif/else construct

```
if logicalExpression1 % Mandatory
    % MATLAB statements
elseif logicalExpression2 % Optional
    % MATLAB statements
elseif logicalExpression3 % Optional
    .
    .
    .
elseif logicalExpressionN % Optional
    % MATLAB statements
else % Optional
    % MATLAB statements
end % Mandatory
```

Plotting

<code>contour</code>	Contour plot on a plane
<code>contour3</code>	3D contour plot with displayed depth
<code>mesh</code>	3D mesh surface
<code>meshc</code>	Combination mesh/contour plot
<code>meshz</code>	3D mesh with curtain
<code>pcolor</code>	Pseudocolor (checkerboard) plot
<code>plot</code>	Basic 2D plots
<code>plot3</code>	Plot lines and points in 3D space
<code>surf</code>	3D colored surface
<code>surfc</code>	Combination surf/contour plot
<code>surf1</code>	3D shaded surface with lighting

Plotting annotation

<code>clabel</code>	Contour plot elevation labels
<code>colorbar</code>	Display color bar (color scale)
<code>legend</code>	Graph legend
<code>title</code>	Graph title
<code>xlabel</code>	X-axis label
<code>ylabel</code>	Y-axis label

Additional plotting functions

<code>box</code>	Toggle the box display
<code>colormap</code>	Color look-up table
<code>grid</code>	Toggle the grid state
<code>hold</code>	Control multiple plots on a single figure
<code>shading</code>	Color shading mode <i>e.g.</i> <code>flat</code> , <code>interp</code>
<code>subplot</code>	Control multiple plots in one window
<code>zoom</code>	Enable mouse-based zooming

Index

- algorithm
 - al-Kashi's power algorithm, 38, 39
 - Breakfast, 28
 - definition, 1, 23–28
 - design, 29, 38, 39
 - the three pillars of**, 26
 - raising a number to an integral power,
 - 38, 39
 - roots of the quadratic equation, 29
- computer
 - architecture, 23, 40
- flowchart, 24
 - simple branching example, 25
 - simple looping example, 26
 - simple sequence example, 24
- Gauss, Karl Friedrich 1777–1855 AD, 35, 44
- instruction
 - branch, 25
 - conditional, 25
 - decisions, 25
 - definition, 23
 - flow, 24
 - jump, 25, 26
 - looping, 25, 26
 - sequencing, 23
- logic
 - flow, 24
- pseudocode, 24
 - simple branching example, 25
 - simple looping example, 26
 - simple sequence example, 24
- sequencing, *see* instruction, sequencing
- statement, *see* instruction