James Madison University
Department of Mathematics & Statistics

Math 248:
Computers and Numerical Algorithms
Part I: Computers and Structured Programming

AUTHORS:
C. David Pruett (pruettcd@jmu.edu)
Anthony Tongen (tongenal@jmu.edu)

August 21, 2013

Math 248, as the course name suggests, has a dual focus on *computers* and *algorithms*. In the course, sophomore-level students will learn 1) structured programming in a high-level programming language, and 2) useful algorithms for performing numerical tasks such as rootfinding, solving systems of linear equations, integrating and differentiating, and interpolating. To our knowledge, Math 248 is unique to JMU. We know of no other university that offers programming and numerical methods seamlessly in one course.

Packaging programming and numerical methods together affords many opportunities. It also carries many pitfalls. The purpose of this coursepack is to structure the course so that students and instructors can take advantage of the opportunities while avoiding the pitfalls.

Contents

1 Part I: Historical Perspectives and Nomenclature

As the title of the course suggests, Math 248 has dual foci: *computers* and *numerical algorithms*. Part I of the course, during approximately the first quarter of the semester, will concentrate on computers and structured programming. During Part II of the course, amid the remaining three quarters of the semester, programming per se will take a back seat. We will concentrate on developing algorithms to accomplish numerical tasks such as rootfinding, integration and differentiation, interpolation, and solving linear systems of equations. However, because the generic algorithms will be converted to programs in a specific programming language and implemented in programming projects, the two components of the course will necessarily be interlaced and the course will toggle back and forth between Parts I and II. Specifically, each week of class will include at least one computer laboratory day in which aspects of programming will be explored in hands-on fashion.

Finally, Math 248 reinforces many concepts learned in the calculus sequence (Math 235 & Math 236) and in Math 238 (Linear Algebra and Differential Equations). No prior programming experience is expected of you. However, Math 248 is inevitably a difficult course. Success in the course will require commitment and consistent hard work. The fruits of that labor will be uncommon mathematical maturity and a depth of understanding, particularly of numerical methods, that distinguishes JMU mathematics and science graduates from the rest of the pack.

1.1 Basic Terminology

1. DEF: A *computer* is a mechanical or electronic device that manipulates *data* to accomplish a task.
2. DEF: *Data* are the numbers, symbols, facts, and figures processed by the computer.
3. DEF: An *algorithm* is a well-defined sequence of instructions to be followed in accomplishing a task on a computer.
4. DEF: A *program* is a list of instructions in a specific programming language (Fortran, MATLAB, C, Java, etc.) that tells the computer how to process the data.

EX: This example demonstrates the sequence of steps used to compute the greatest common integer divisor of the two numbers 1365 and 3654 using Euclid's algorithm.

3654 / 1365 R 924
1365 / 924 R 441
924 / 441 R 42
441 / 42 R 21
42 / 21 R 0
21 is the GCD.

EX: Write out the GCD algorithm in plain English.

QUESTIONS:

1. What is the difference between an *algorithm* and a *program*?
2. What is the language of algorithms?

1.2 Types of Computers

The definition of a computer above is intentionally very general. It encompasses a wide variety of devices one might not normally think of as computers. Under this definition, an abacus is a computer, a slide rule is a computer, a digital watch is a computer, and an MP3 player is a computer.

Mathematics is divided into two distinct specialties: *discrete mathematics* and *continuous mathematics*. Ultimately, discrete mathematics involves integers, and continuous mathematics involves the real numbers. Between any two integers there may or may not be another integer. For example, 3 lies between 2 and 4. But there is no integer between 2 and 3 or between 999 and 1000. In contrast, between any two real numbers, no matter how close, there can always be found another real number. Therefore, the real numbers are assumed to be a *continuum*, with no holes or gaps. Thus, discrete mathematics is characterized by steps or *jumps* and continuum mathematics by smooth *flow*.

Similarly, computers are divided into two major categories, depending upon whether the input and output data are discrete or continuous. *Analog* computers deal with continuous data; *digital* computers, on the other hand, deal with discrete data, which could be mapped onto the integers.

QUESTIONS:

1. What was the first computer?
2. Was the first computer digital or analog?
3. Is a slide rule digital or analog?
4. Is an abacus electronic or mechanical?
5. Is a watch digital or analog?
6. How often is digital device correct?
7. If digital devices are nearly always wrong, why then have digital devices dominated analog ones?
8. OK, so digital devices are favored because they can be programmed, but what do we lose?

1.3 A Brief History of the Digital Computer

Before the age of electronics, computers were rare, and of necessity, they were mechanical. Charles Babbage (1791-1871), a contemporary and compatriot of Charles Darwin, is now regarded by many as the foremost pioneer of computing. Like Isaac Newton before him and Stephen Hawking after him, Babbage held the presitgious Lucasian Chair of Mathematics at Cambridge University. Although he excelled in many areas of mathematics and filled entire notebooks with inventive ideas of all sorts, Babbage was consumed throughout most of his life by the quest to build a mechanical computer. His original model, the Difference Engine, was designed to use a series of gears to evaluate 7th-degree polynomials to 31 significant digits. Celestial navigation tables, which depended upon polynomial interpolation between observations, were notoriously unreliable because of human error both in calculation and in transcription. Babbage's motivation for the Difference Engine was to eliminate the human error so as to produce a reliable table. Far more brilliant than the Difference Engine was the Analytical Engine, one of the great intellectual feats of the 19th Century. The Analytical Engine was a true computer in the modern sense in that it could be programmed to accomplish many different tasks.

Although components of the Difference and Analytical Engines were built and tested in Babbage's lifetime, difficulties with machining to precise tolerances prohibited Babbage from ever delivering a completed working machine. His personality and disputes with his chief machinist also contributed to Babbage's ultimate failure.

For more than a century after his death, it remained a matter of speculation as to whether or not Babbage's Engines would have worked as intended had they been completed. In 1985, the British Science Museum began a mammoth project to construct Babbage's Difference Engine No. 2 from his meticulous drawings. The project was completed in November 1991, just in time to celebrate the 200th anniversary of Babbage's birthday. Upon its completion, Difference Engine No. 2, with 4000 moving parts and powered by a hand crank, worked just as Babbage had designed it to. This magnificent device can be seen on the third floor of the Science Museum in London.

Although there is no direct lineage between Babbage's mechanical devices and today's electronic ones, Charles Babbage remains "a great ancestral figure in the history of computing." Babbage also developed a mentoring relationship with a tempestuous and brilliant young woman, Ada Lovelace, daugther of Lord Byron. Today's programming language ADA honors Mrs. Lovelace for her early contributions to the art of programming.

Those interested in Babbage's inventions are encouraged to read *The Difference Engine: Charles Babbage And The Quest To Build The First Computer* by Doron Swade (2000).

The modern digital computer arose out of necessity during the Second World War. In the United States, digital computing was developed for the Manhattan Project, the code name for the atomic bomb project, and for flight simulators. Early simulators were analog, and when a new aircraft was developed, a corresponding new simulator also had to be produced for training purposes. Digital computing was attractive for flight simulation in that a single simulator could be reprogrammed to mimic the characteristics of virtually any aircraft.

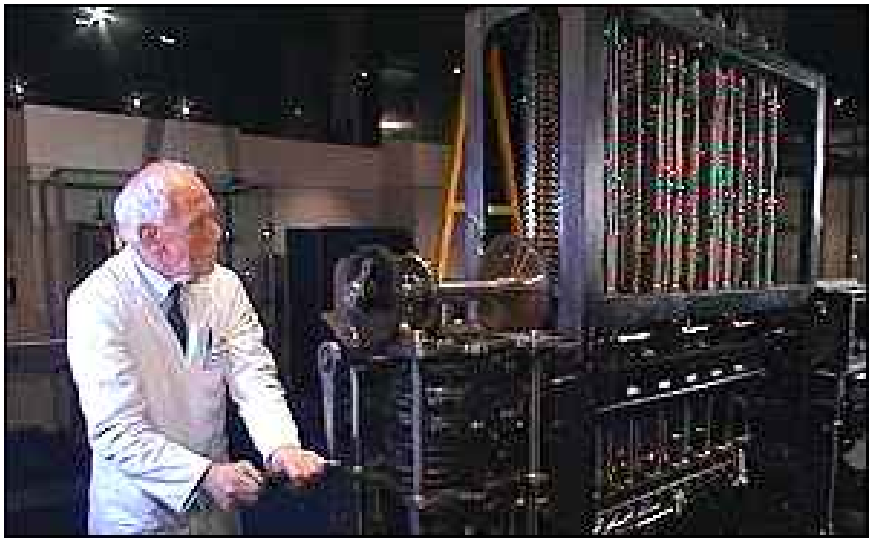


Figure 1: Babbage's Difference Engine No. 2, completed in 1991 for the bicentennial celebration of Babbage's birth, and weighing in at 2.6 tons and 4000 parts, exclusive of the printer, which contains an additional 4000 parts.

In Britain, there was a pressing need for computers for an entirely different reason. Wolf packs of German U-boats were sinking supply ships with virtual impunity. Britain, it was feared, would starve to death. Hope lay in the impossible: breaking the Nazi's secret code by which messages to the German Naval Command and the Luftwaffe were encoded. That task fell largely on the shoulders of a brilliant young logician and mathematician named Alan Turing, who worked for the top-secret Government Code and Cypher School (GCCS) at Bletchley Park. The GCCS had a captured "Enigma" device in its possession, which the Germans used to scramble radio transmissions. The problem, however, was that the number of possible scrambling permutations was impossibly large. Turing conceived an electronic device that could cycle through all possible permutations until the meaningful one was found. Turing's device, the forerunner of the digital computer, worked, and the tide of the war turned.

In 1950, Turing authored one of the most prescient papers of all times, published in the journal *Mind* under the intriguing title "Can a Machine Think?" His article shaped the science of computing for the next 50 years and continues to shape it today. In the article, Turing accomplished two seminal feats. He laid out the structure of the archetypal digital computer. He also pondered the question: When digital computers attain a certain critical mass, can it be said that they are capable of "thought." This question marks the origin of the AI (artificial intelligence) hypothesis, and it remains a question hotly debated.

In honor of Alan Turing, all digital computers, which share the basic structure outlined by Turing (despite their surface differences), are known formally as Universal Turing Machines.

1.4 The Structure of Digital Computers

The Universal Turing Machine, the archetype of the digital computer, consists of but three basic components, given here in Turing's words:

> (i) Store. (ii) Executive unit. (iii) Control.
>
> The store is a store of information, and corresponds to the human computer's paper, whether this is the paper on which he does his calculations or that on which his book of rules is printed. In so far as the human computer does calculations in his head a part of the store will correspond to his memory.
>
> The executive unit is the part which carries out the various individual operations involved in a calculation. What these individual operations are will vary from machine to machine. Usually fairly lengthy operations can be done such as 'Multiply 3540675445 by 7076345687' but in some machines only very simple ones such as 'Write down 0' are possible.
>
> We have mentioned that the 'book of rules' supplied to the computer is replaced in the machine by a part of the store. It is then called the 'table of instructions'. It is the duty of the control to see that these instructions are obeyed correctly and in the right order. The control is so constructed that this necessarily happens.

We expand on the analogies that Turing draws between the human computer and the mechanical computer in Fig. 1.1.

QUESTIONS:

1. What is the modern term for "store?"
2. What is the modern name for Turing's "control" function?
3. Where does the data reside? Where does the program reside?

1.5 Trends in Digital Computing

In the 1940's, the word *computer* meant a person, not a machine. Difficult calculations were made by rooms full of "computers," each sitting at a desk with mechanical calculators. Most "computers" were women. Typically, for long calculations, each "computer" would do one *operation*, say an add or a multiply, and then pass the computation along to the next "computer" for the next operation, in assembly-line fashion. Under the best of situations, "computers" could do an operation in one or two seconds.

The acronym FLOPS (floating-point operations per second) is used to measure the performance of a computer. Loosely, a FLOP is a single operation (add or multiply) involving two decimal numbers. Thus, in the 1940s, "computers" could execute at a rate of one-half or one FLOPS at best. A prime motivation for

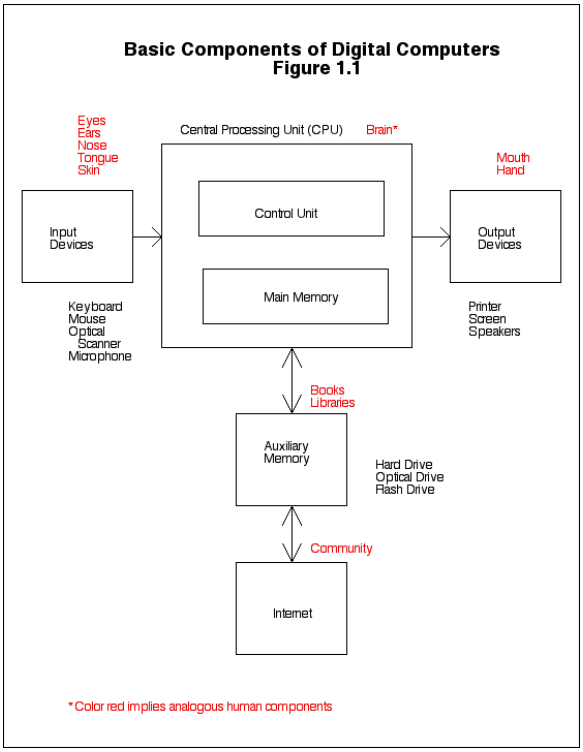


Figure 2: An analogy for the components of a computer

the development of the digital computer was the need for computation rates on the order of 1000 FLOPS or more. Early digital computers used vacuum-tube technology (similar to incandescent light bulbs), filled entire rooms or buildings, and required enormous amounts of electrical power.

Digital computers are essentially long chains of electrical switches, each switch having the *binary* possibility of being either off (0) or on (1). Had the digital computer continued to rely on vacuum tubes as switches, the computers of today would be little faster than the ones of the 1950s. However, one of the most important discoveries of the 20th Century revolutionized computing technology and paved the way for the phenomenal growth in computing performance that we enjoy today. In December 1947 William Shockley, John Bardeen and Walter Brattain of Bell Laboratories succeeded in building the first practical transistor. A transistor is a solid-state switch that is both small and energy efficient. Transistors allowed digital computers to be miniaturized, and with miniaturization comes speed.

The next breakthrough in computing was the development of the integrated circuit, now called the "microprocessor" or the computer "chip," in which large numbers of transistors are imprinted onto a single silicon wafer. Gordon Moore, one of the co-founders of Intel, the pre-eminent maker of computer "chips," was among the first to envision how chip technology would advance into the future. In 1965 Moore predicted that the number of transistors on a microprocessor would double about every two years. Accordingly, the speed of a microprocessor, measured in FLOPS, doubles roughly every other year. In Moore's own words:

The first microprocessor only had 22 hundred transistors. We are looking at something a

million times that complex in the next generations–a billion transistors. What that gives us in the way of flexibility to design products is phenomenal.

Moore's Law, as indicated by Figure 3, has held true for nearly five decades, during which computer speed has also increased exponentially in time. Today's research supercomputers promise a performance in the teraflop range, a teraflop being one million million (10^{12}) FLOPS or one million megaflops! Related to the term flops is MIPS, meaning millions of instructions per second. Thus, teraflop performance is also one million MIPS.

Moore's law cannot hold indefinitely. Eventually quantum effects become important when transistors shrink to the size of a few atoms.

Terms related to the performance of a microprocessor are *clock speed*, *cycle time*, and *execution rate*.

DEF: *Clock speed* is the number of "ticks" in one second for the clock of the microprocessor. EX: The Intel Pentium IV, initially released in 2000, had a clock speed of 2.0 GHz (gigaherz); that is two billion ticks per second.

DEF: *Cycle time* is the time that elapses between ticks of the microprocessor's clock. Cycle time is the reciprocal of the clock speed. EX: Find the cycle time of the Intel Pentium IV above. cycle time = 1.0 / (2.0 x 10^9 cycles/second) = 0.5×10^{-9} seconds.

The cycle time is the shortest amount of time in which the processor can do any useful work. Many processors can perform an operation (add or multiply) in one or two clock cycles. Thus a 2.0GHz processor that can perform an operation in one clock cycle can perform 2 billion operations in one second. That is, it operates at a peak execution rate of 2.0 gigaflops.

1.6 Miscellaneous Jargon

Make sure you know what is meant by each of the following terms:

- Hardware (the physical components of a computer) vs. software (the programs)
- I/O (input/output, as in input/output devices)
- Data terminology
  - Characters
    - Numeric $\{0,1,2,3,4,5,6,7,8,9\}$
    - Alphabetic $\{a,A,b,B,...,z,Z\}$
    - Alphanumeric = Numeric + Alphabetic
    - Special {+,-,@,!,$,etc.}
  - Field (a collection of related characters) EX: social security number

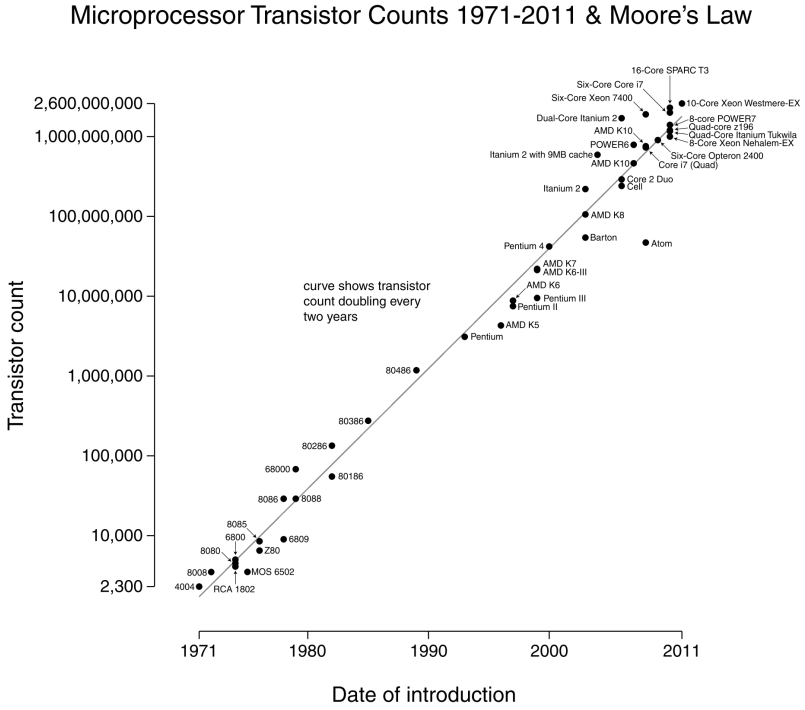


Figure 3: Moore's Law: Exponential growth of microprocessor transistors over time. (From Wikipedia.)

- Record (a collection of related fields, usually on one line) EX: one line of class roster
- File (a collection of related fields) EX: entire class roster

Digital computers make use of *binary* (base 2) arithmetic, primarily for the reason that they are comprised of strings of switches, and the simplest switches have only two positions: off and on. In a binary number system, there are only two digits: 0 and 1. Any number, in any number system, can be represented in binary as strings of zeros and ones. We'll figure out how to do this later. For now, the simplest unit of information that can be stored in a computer is a *bit*, a contraction of "binary digit." Thus a bit is either a zero or a one.

The smallest unit of memory typically is a *byte*, defined as 8 contiguous bits. Most modern PCs have many megabytes of memory. A *megabyte* is one million bytes. One of the great advances in computing, which helped make possible today's fast computers, was the development of *random access memory*, or RAM for short. As a graduate student at M.I.T. in the mid 1940's, Jay Forrester pioneered the development of magnetic core RAM while working with fellow student Robert Everett on Whirlwind, the first real-time digital computer, originally designed as a universal (i.e., programmable) flight simulator. Forrester's invention earned him induction into the National Inventors Hall of Fame in 1979, and Whirlwind, although never used as a flight simulator because of the conclusion of World War II, became the prototype for modern digital computers. The primary advantage of RAM over other types of memory is that RAM allows memory addresses to be accessed randomly, that is, in any order "without the physical movement of the storage medium or a physical reading head (Wikipedia)." Because RAM involves only integrated

circuits without moving parts, data storage to RAM or data retrieval from RAM is many times faster than that from hard drives or tapes.

QUESTIONS:

1. Of Turing's three components, which is/are *hardware* and which is/are *software*?

1.7 Computer Languages

1.7.1 Low-Level vs. High-Level Languages

Low-level languages include *machine language* and *assembly language*. Ultimately, computers run (execute) *machine language*. Machine language is specific to the given computer, resembles the circuitry of the specific computer, and is largely unintelligible to ordinary humans. For example, here is a sequence of four instructions in machine language:

1. 00010000000000000000010000000000
2. 00010010000000000000010000000001
3. 00010001100000000000010000000010
4. 00010000100000000000010000000011

Note: The first 8 bits contain the opcode (operation code) and the remaining 24 bits give the address in memory. Let's translate.

Early digital computers were programmed in machine language. Can you think of some problems associated with this?

For this reason, *high-level* languages were developed. A high-level language is machine independent and resembles human language. Here is the same sequence of instructions as a MATLAB assignment statement:

X = A*B + C

Intermediate between machine language and high-level language is assembly language. Here is the same sequence of instructions in assembly language:

1. MOV A, ACC

2. MUL B, ACC
3. ADD C, ACC
4. STO ACC, X

Which do you prefer?

1.7.2 Some High-Level Languages

Here are just a few common examples of the many high-level languages currently in use:

- Ada
- BASIC (Beginners All-Purpose Symbolic Instruction Code)
- C, C++
- FORTRAN (Formula Translation), FORTRAN 77, Fortran 90, Fortran 95 (NOTE: By convention, versions later than 77 are referred to as Fortran rather than FORTRAN.)
- Java
- MATLAB (Matrix Laboratory)
- Perl

1.8 The Role of the Compiler

DEF: A *compiler* is a machine-specific software program that translates from high-level language (e.g. Fortran 90, C, etc.) into low-level language (either assembly language or machine language). The high-level language is generically referred to as *source code* and the machine language as *object code*. Sometimes the translation from source to object code is accomplished in a single step and sometimes in two steps. If in two steps, the first step is *compilation*, by which the compiler translates from high-level into assembly language, and the second step is *assembly*, whereby the assembly language is converted to machine language. Alternately, some compilers translate directly from source code to machine code. For our purposes, we will assume the latter; that is, that the compiler receives source code and produces object code in one fell swoop.

Even if the details of compilation now seem somewhat hazy, please keep the following ever in mind: a compiler is to human-computer communication what a translator is to human-human communication, as illustrated in Figure 4.

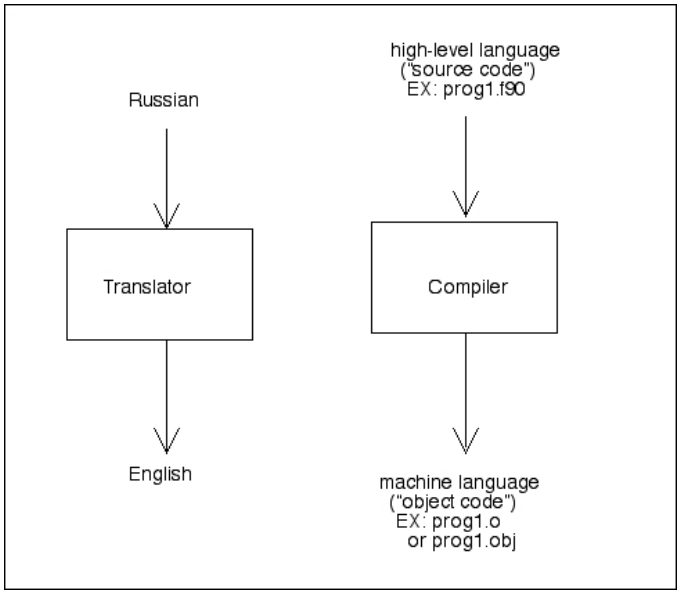


Figure 4: An analogy for the role of the compiler

2 Part I: The Structure of Elementary Programs

The following is an example of an elementary program written in MATLAB. Study it closely for a few moments.

```
%   quadratic_formula.m
%   Compute the roots of a quadratic function

%%\begin{variables}
    a=zeros(1,1); b=zeros(1,1); c=zeros(1,1); two_a=zeros(1,1);
    discriminant=zeros(1,1); root_disc=zeros(1,1);
    root_1 = zeros(1,1); root_2=zeros(1,1);
%%\end{variables}

%   input portion of program

    a=input('Please enter the second order coefficient \n');
    b=input('Please enter the first order coefficient \n');
    c=input('Please enter the zero order coefficient \n');

%   processing portion of program

      discriminant = b*b - 4.0*a*c;
      root_disc = sqrt(discriminant);
      two_a = 2.0*a;
```

```
      root_1 = -b + root_disc;  % root_1 is used as a scalar temporary
      root_1 = root_1 / two_a;
      root_2 = -b - root_disc;  % root_2 is used as a scalar temporary
      root_2 = root_2 / two_a;

%   output portion of program

    real_part1 = real(root_1);  % separate real/imag parts for printing (root_
    complex_part1 = abs(imag(root_1));
    real_part2 = real(root_2);  % separate real/imag parts for printing (root_
    complex_part2 = abs(imag(root_2));

    fprintf(' root_1 = %5.2f + %5.2f i \n', real_part1,complex_part1);
    fprintf(' root_2 = %5.2f - %5.2f i \n', real_part2,complex_part2);
```

What distinct tasks of the program can you identify? Bravo if you said: input, process, and output. Almost all programs, no matter how complicated or in what programming language, involve these three basic tasks, albeit oft repeated.

2.1 The "Flow" of a Computer Program

IMPORTANT: Unless otherwise instructed, the execution of a program proceeds *from the top down*, each statement in succession starting from the top of the program and progressing to the bottom of the program. Only one type of structure (the *loop*) will violate this top-down sequencing, and it will be a couple of weeks before you encounter loops. Students can avoid countless difficulties by keeping this simple fact always in mind: top to bottom in time, top to bottom in time, ... For example, if the natural ordering of events is to input some data, process the data, and then output the result(s) of the process, what do you think happens if you try to process the data *before* it has been input into the computer? It depends on the compiler, but likely a run-time crash. Not good.

In the old days, it was possible to write "spaghetti code," that is, convoluted programs in which the top-down ordering was frequently violated. This in turn required that programmers relied upon extensive flowcharting to lay out the sequence of events. Figure 5 portrays the "flow" of an elementary program by means of a flow chart. Note that different geometric shapes are used for different tasks. What shape is used for I/O tasks? What shape is used for process blocks? What shape is used for decision blocks?

Mercifully, careful *structured programming* has all but eliminated the need for flowcharting.

Alternately, the structure and process of an algorithm or a program can be outlined in *pseudocode*. Pseudocode can be thought of as a generic programming language in which one outlines program structure and function but doesn't worry about specific syntax. Pseudocode, therefore, is written in ordinary English and makes use certain of shorthand devices for decision blocks and loops. Here is the structure of an

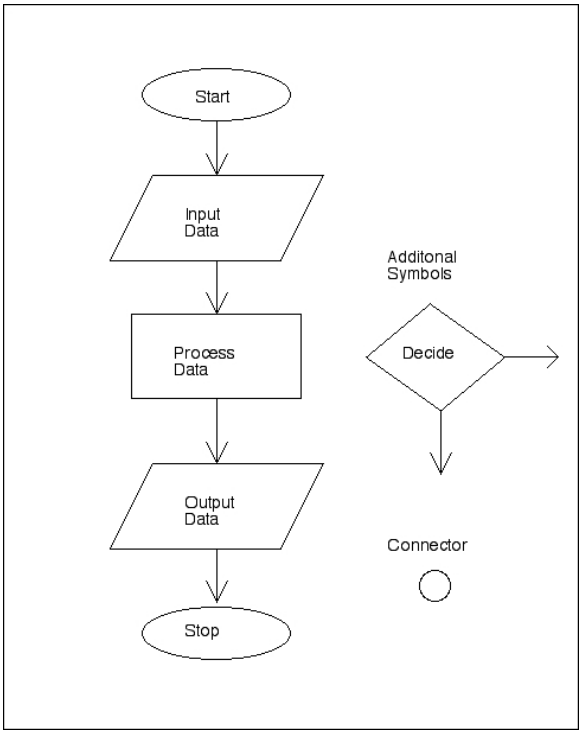


Figure 5: The flow of an elementary computer program.

elementary program in pseudocode:

begin program
    read initial data
    process data
    output result
end program

Here is a two-sided decision structure in pseudocode:

if (something) is true, then
    do this
otherwise
    do this instead
end if

Note that pseudocode makes use of indentations to accentuate the structure and function of an algorithm or a program.

As the art and science of programming matured, *structured programming* became the norm. Structured programming is the use of 1) white space, 2) indentations, 3) descriptive variable names, and 4) liberal documentation to make the flow of a computer program transparent to the programmer and/or the user. A well-structured program not only does its job, it is a work of art and a model of clarity. Whether or not you

ever again program outside of Math 248, the clarity of mind that structured programming develops will benefit you in all kinds of ways. Spaghetti code is the result of muddled thinking. Well-structured code is the product of a lucid and orderly mind.

2.2 Building a Program

What are the odds a program that you write will compile on the first try? Realistically, slim to none. What are the odds that a program that compiles correctly will run correctly on the first try? Slimmer to none. How then does anyone write a successful program? By an iterative process of baby steps, as described below. By taking small steps, one rewards oneself by a sequence of small successes rather than punishing oneself by a massive failure.

repeat until successfully executed
| *repeat until successfully compiled*
| | *create or modify source code using text editor of choice*
| | *compile edited source code [compilation (compile-time) errors?]*
| |
| *execute program [execution (run-time) errors?]*
|

REMARKS:

1. The previous example uses the pseudocode shorthand for a *loop*, which is a repetitive process. Whatever is within the bracketed region is to be repeated (iterated).

2. There are two major categories of errors involved in writing programs: *compile-time errors* and *run-time errors*, also called compilation errors and execution errors. Errors are also known affectionately as "bugs," and the process of finding and correcting errors is "debugging." Many compilers offer debugging tools. While these tools can be very useful, they are no substitute for good programming practices and should not be blindly depended upon at the expense of proper planning and design.

2.3 The Building Blocks of Computer Programs

Computer programs are built from three types of statements: 1) **comments**, 2) **non-executable statements**, and 3) **executable statements**.

2.3.1 Comments

Here is a MATLAB example of a comment:

% This is too easy.

REMARKS: :

1. In MATLAB, comment lines always begin with a percentage sign.
2. Comments are for human consumption only. They are completely ignored by the computer.
3. Nonetheless, comments are *very* important as one of the tools of structured programming. A well-written computer code should contain at least 30% comment lines.
4. In MATLAB, a comment can follow on the same line as an executable or non-executable statement.

2.3.2 Executable Statements

Executable statements cause the computer to take some action during execution. Here are some executable statements in MATLAB. Comments have been added to describe their type or purpose.

```
x = y + 12;        % an assignment statement
x = input('Please enter an x value \n');  % input x from the keyboard
fprintf('%5.2f \n', x);  % to output the value of x to the screen

if(x > 1)  % these three lines comprise a decision block
    fprintf('Too big ');  % note that the guts of decision blocks are indented
end

for j = 1:10  % these three lines comprise a loop
    fprintf('%d ', j); % note that the guts of loops are indented too
end
```

2.4 The Building Blocks of Executable Statements

Almost every line in a MATLAB program is an executable statement. Executable statements are in-turn comprised of the following components:

1. constants: -1, 2.7E08, 'good grief Charlie Brown'
2. variables: x, y, z, number, my_dogs_name
3. arithmetic operators: +,-,*,/,**

4. logical operators: ==,>,<,>=,<=
5. intrinsic functions: cos(X), sin(X), log(X), etc.

2.5 Data Types, Constants, and Variables

MATLAB supports numerous *data types*. Some examples are:

1. double
2. int8, int16, int32, int64
3. uint8, uint16, uint32, uint64
4. char
5. logical

These data types apply both to *constants* and to *variables*.

2.5.1 Constants in MATLAB

The best way to understand constants is by example. Here are some examples of constants:

- integer constants: -1*ones(1,1,'int32'), zeros(1,1,'int32'), 37*ones(1,1,'int16')
- double (real) constants: -1.0, 395.7, 0.0, 1.23E08, 0.00013, 1.5E-16
- character (char) constants: 'good grief', 'my name is', 'z=x+y'

REMARKS:

1. Real constants have a decimal point and may be expressed in scientific notation. For example, 1.23E08 means 1.23×10^8. What does 1.5E-16 mean?
2. Character constants are delineated by single quotes. What is between the delineators is called a *character string.*
3. The ones(m,n) command declares an $m \times n$ matrix of ones. Therefore, ones(1,1) declares a variable of size 1×1, otherwise known as a scalar. If you do not know about matrices, assume that the command ones(1,1) is a way to declare a scalar variable. The zeros(m,n) command is similar except it declares an $m \times n$ matrix of zeros.

2.5.2 Variables in MATLAB

Here are some rules for naming variables in MATLAB:

1. Variable names may include up to 63 alphanumeric characters plus the underscore symbol.
2. Variable names must begin with an alphabetic character (a letter).
3. No other special symbols are permissible for variable names.
4. MATLAB is case sensitive; thus, "a" and "A" are interpreted as different characters.

Given these rules, which of the following variable names is legitmate? Which are illegitimate?

my_name
my-name
fido
my_dog's_name
10fido
BubbleGum10
zippety_doo_dah
supercalifragilisticexpialodoshias

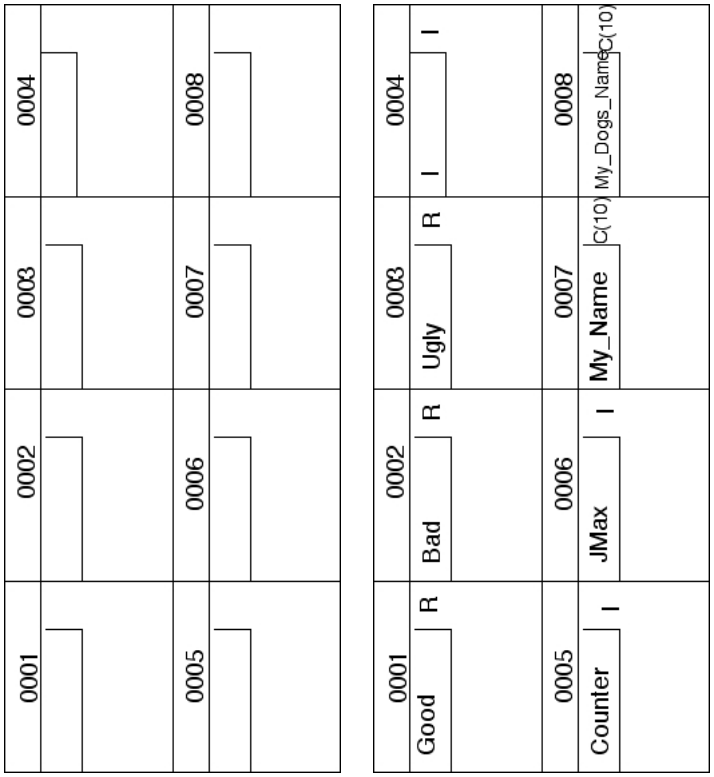
2.5.3 Declaration of Variable Type

Type declaration is an extremely important aspect of computer programming. However, MATLAB does not make the user declare variables before using them. In order to enforce the practice of declaring variables, we will use the following syntax:

```
%%\begin{variables}
    a = zeros(1,1,'int32');  % declare a to be of type int32 (= 0)
    b = zeros(1,1); % declare b to be of type double (= 0.0)
    c = ones(1,1); % declare c to be of type double (= 1.0)
%%\end{variables}
```

REMARKS:

1. Every program will have a section, like the above section, where all of the variables are declared.
2. In MATLAB, variable declaration is more for the reader of your program than for actual use.
3. If you declare a to be of type int32 and later type a=32, MATLAB will by default make a of type double and overwrite your previous declaration. Reread this sentence.



(a) Memory before type declaration

(b) Memory after type declaration

Figure 6:

2.5.4 Correspondence Between Storage Registers and Variable Names

A simple analogy will help you understand the purpose of type declaration statements. Imagine coming to JMU as a freshman. Before you arrive there is a bank of unassigned mailboxes awaiting the freshman class. These unassigned mailboxes are like the storage (memory) locations in a computer. Once you register at JMU, a mailbox is assigned specifically to you, and from that point on, all your mail will be delivered to that specific box and to no others. Similarly, the type declaration command above

```
good=zeros(1,1); bad=zeros(1,1); ugly=zeros(1,1);
```

assigns three memory locations, one to store the value of the variable "good," one to store the value of "bad," and one to store the value of "ugly." However, these locations only accept REAL numbers. It would be inappropriate, for example, to try to store character strings in these locations.

In summary, type declaration associates a variable name with a memory register that will hold the value of that variable.

Figures 6(a) and 6(b) show the first few memory locations of the computer before and after compilation of a program containing the type-declaration commands above.

A couple of fine points. Of course, there are far more than 8 memory registers in a modern computer. Most PCs now come standard with 512MB (megabytes) of random-access memory (RAM), which translates to 128 million 32-bit storage locations. Secondly, memory addresses are encoded in binary rather than decimal notation as shown in the figures.

2.6 Arithmetic Expressions and Assignment Statements

By far, the most common statement in a scientific computer program is the *assignment statement*. In MATLAB, assignment statements have the following syntax:

VariableName = Expression;

The expression on the right hand side of the equal sign is comprised of any or all of the following: constants, variables, arithmetic operators, and intrinsic functions. Here are some assignment statements in MATLAB:

```
y = a*x + b;    % * is the symbol for multiplication in MATLAB
z = y ^ 2;      % ^ is the symbol for exponentiation in MATLAB
w = cos(z);     % cos(z) is an intrinsic function
w = w + 1;      % This makes sense in MATLAB; how can that be?
```

The equality symbol does *not* mean "equal" in the context of assignment statements. It means "replace by." More specifically, the value of the variable on the left is replaced by the value of the expression on the right. This implies that *the right-hand side is evaluated first.* Then the value on the left is replaced. In pseudocode, an assignment statement looks like:

$$\textit{Variable Name} \leftarrow \textit{Expression Value}$$

Try to get in the habit of mentally thinking of the "=" as a left arrow that means "replace by." With this in mind, does the fourth statement above make sense mathematically? Does it make sense in MATLAB? Why the difference? Does the following expression make sense mathematically? Does it make sense in MATLAB?

```
A + B = C + D
```

2.6.1 Arithmetic Operators in MATLAB

Here are the arithmetic operators employed by MATLAB:

+ addition

- subtraction

* multiplication

/ division

^ exponentiation

() grouping symbol

In an arithmetic expression with multiple operations, certain operators have priority. Here are the priorities, in the order of highest to lowest:

1. evaluate grouped quantities from the inside out
2. exponentiate
3. multiply or divide in the order left to right
4. add or subtract in the order left to right

The following shows a useful twist to type declaration:

A=ones(1,1)*4.0; B=ones(1,1)*6.0; C=ones(1,1)*2.0;

Play computer and evaluate the following assignment statement exactly as it would be executed by running the program, showing all steps in the process.

EX1: Z = A + B/C

$A + 6.0/2.0 \rightarrow A + 3.0 \rightarrow 4.0 + 3.0 \rightarrow 7.0 \rightarrow z$

REMARKS: In all programming languages of which we are aware, assignment statements are always evaluated by a sequence of operations on binary pairs; that is, by considering only two operands at any given time.

Now you try it for the following examples.

```
EX2: Z = (A + B) / C;
EX3: Z = 4.0*A*C - B^2;
EX4: A = A + 1.0;
```

Is the following statement wrong? If so, how can it be corrected?

EX5a: Z = A*-B;

In Fortran 90, there cannot be two adjacent arithmetic operators without grouping symbols. However, MATLAB correctly computes the above statement. Even so, statements should be understandable and explicit. Therefore, the correction is EX5a: Z = A*(-B). Explain why this fixes the problem. By the way, the negation of B is a *unary* operation.

2.7 Basic I/O Instructions

There are numerous I/O commands in MATLAB; we will examine the use of input and fprintf. When an input statement is executed, the computer is listening. When a fprintf statement is executed the computer is talking. The two commands, therefore, allow communication between humans and the computer during the execution of a program.

There are two types of I/O: *formatted* and *list-directed*. With formatted output, the programmer decides in which fields the output data will be written. List-directed output is quick and dirty; it makes use of default settings, and the programmer has relatively little control over the appearance of the output. A programmer could easily spend several weeks with all the nuances of formatting in MATLAB. However, we will use formatted output this semester.

The >> sign below is the MATLAB input prompt. Values entered by the user are red. Consider the following MATLAB commands:

```
>> val = input('Please enter a value ')
Please enter a value 10

val =

    10

>> val = input('Please enter a value ');
Please enter a value 10
>> val = input('Please enter a value \n');
Please enter a value
10
>> vals = input('Please enter 3 values \n');
Please enter 3 values
[1 2 4]
>>
```

Notice that the syntax for the input statement to enter a single value or multiple values is the same; although this is an advantage, the programmer should take extra care that the user understands how to

input the specific values. Another formatting issue is the \n in the input command; it prints out a carriage return before the desired value is entered. The final formatting issue is the method by which multiple values are entered. The user is giving the program information in the form of a array (vector) rather than a scalar. We will discuss this topic more in the future.

Consider the following MATLAB commands:

```
>> a = 1;
>> b = [1 2 4];
>> fprintf('%f \n', a);
1.000000
>> fprintf('%4.2f \n', a);
1.00
>> fprintf('%f ', a);
1.000000 >> press enter key
>> fprintf('%f \n', b);
1.000000
2.000000
4.000000
>> fprintf('%f ', b);
1.000000 2.000000 4.000000 >>
```

There are two subtleties about the fprintf command. The first subtlety is the ability to format the output; for instance, using %4.2f rather than %f. The second subtlety is the letter after the % sign. Both of these subtleties will be explored in your first lab. After completing the lab, please fill in the following chart in your notes. Another common difficulty is previously % corresponded to comments and now it corresponds to formatted output; in formatted input the % is in single quotes.

Table to describe behavior of fprintf

Format Code	Description
%d	
%e	
%E	
%f	
%g	

Control Code	Description
\n	Start a new line
\t	

Describe in detail what you think happens in the computer's memory when the sequence of instructions above is executed.

2.8 The Continuation Character

Assignment statements can become quite involved, and may spill off of a record (line). In MATLAB, any type of statement, assignment or otherwise, can be continued on the next line by appending with ... at the end of the line. For example,

```
a = 5*2 + 3*6 + ...
3*4 + 4*2;
```

2.9 Intrinsic Functions

In general, *intrinsic functions* are those that come as standard issue with a compiler. The input value of an intrinsic function is its *argument*. Some intrinsic functions have multiple arguments. Most of the time, the output value of an intrinsic function has the same type (e.g., REAL or INTEGER) as the argument(s).

Mathematical Function	Fortran Syntax	I $\rightarrow$ O Argument Types
$\sin(x)$	sin(x)	R $\rightarrow$ R
$\cos(x)$	cos(x)	R $\rightarrow$ R
$\tan(x)$	tan(x)	R $\rightarrow$ R
e^x	exp(x)	R $\rightarrow$ R
$\ln(x)$	log(x)	R $\rightarrow$ R
$\sqrt{(x)}$	sqrt(x)	R $\rightarrow$ R
$\|x\|$	abs(x)	ouput same as input
$[\|x\|]$ (greatest integer $\leq x$)	floor(x)	R $\rightarrow$ I
$\max\{x_0, x_1, ...x_n\}$	max(x0,x1,...,xn)	output same as input
$\min\{x_0, x_1, ...x_n\}$	min(x0,x1,...,xn)	output same as input

Table 1: A few MATLAB intrinsic functions.

In programming, as in life, there are often many ways to correctly accomplish a task. Of the two sets of assignment statements below, both give the result $y = 4$. Is there any reason to prefer one over the other?

```
x=ones(1,1); y=ones(1,1);
x = 16.0;
y = sqrt(x);
```

```
x=ones(1,1); y=ones(1,1);
x = 16.0;
y = x^0.5;
```

The first option is greatly preferred. On some computers, including the state-of-the-art CRAY supercomputers of the 1990s, exponentiation using fractional real exponents (in this case 0.5) was exceedingly slow because it made use of table lookups from logarithm tables. Thus, on CRAY machines, the second variant ran 40 times slower than the first! This is the first of many examples of *mathematically equivalent statements that are not computationally equivalent*, in the sense that, even though they give the same results, one approach requires far more computational effort than the other. In programming, be mindful of good, better, and best ways to accomplish a goal.

2.10 Integer and Mixed-Mode Arithmetic

MATLAB, with a little bit of work, distinguishes clearly between integers and real numbers and treats operations with reals very differently than operations with integers. It also allows for *mixed-mode* expressions; that is, expressions that contain both integers and reals. Evaluating integer expressions is a bit tricky, and evaluating mixed-mode expressions is *very* tricky. Hence, we'd better spend some time with each type of expression: integer and mixed-mode.

2.10.1 Evaluating Integer Expressions

Integer expressions involve only integer constants and/or integer variables. They appear commonly on the right-hand side of assignment statements. Suppose the following type-declaration statement is in effect.

I=ones(1,1,'int32')*5; J=ones(1,1,'int32')*2; K=ones(1,1,'int32'); L=zeros(1,1);

Find the value assigned to L in each of the following examples:

- EX1: L = ones(1,1,'int32')*1/2;
- EX2: L = ones(1,1,'int32')*1/2 + ones(1,1,'int32')*1/3;
- EX3: L = I/J;
- EX4: L = I*K/J;
- EX5: L = I/J/K;
- EX6: L = 2*I/2;
- EX7: L = 2*(I/2);

Here are the answers: 1) L=1, 2) L=1, 3) L=3, 4) L=8, 5) L=1, 6) L=5, 7) L=6 What's going on? Let's play computer and evaluate L in EX 2.

1/2 + 1/3 $\rightarrow$ 1 + 1/3 $\rightarrow$ 1 + 0 $\rightarrow$ 1 $\rightarrow$ L

Here are the rules for integer expressions.

1. All intermediate evaluations are *binary*; that is, only two operands are considered at any given time, in the order of previously established priorities.
2. The intermediate result of an operation involving two integer operands must also be an integer.
3. In integer arithmetic, fractional parts resulting from division are simply *rounded*. Thus 5/2 = 3, and -5/2 = -3.

What then is the difference between -5/2 and floor(-5.0/2.0)?

2.10.2 Assignment Conversions

Assignment conversions in MATLAB are completely straightforward, unfortunately. The variable to the right of the equal sign takes precedence, completely. The variable on the left becomes the same value and type as the variable on the right.

Consider the following type-declaration commands with compile-time assignments:

```
A = 2.5;
B = ones(1,1,'int32')*3
```

Play computer and explain what happens if the following assignment statement is executed:

- EX1: A = B
- EX2: B = A
- EX3: B = 1.999999

2.11 Mixed-Mode Expressions

A mixed-mode expression is one that contains both real and integer variables and/or constants. Mixed-mode expressions are especially tricky, because the protocol for MATLAB may seem counter-intuitive to humans. To make a long story very short: **avoid mixed-mode expressions like the plague!!!**

To conclude, when should one use integer arithmetic?

1. When counting (as in loops).
2. When dealing only with whole numbers.
3. For whole-number exponents. (Integer exponentiation is much faster than real exponentiation.)

3 Part I: Programming for Decisions

Life involves choices, and so do most computer programs. In this chapter, we will learn structured programming for decisions. In MATLAB, the most complicated of decisions can be accommodated with a single construct: the IF block. The IF block is an amazingly versatile and powerful device.

In a flowchart, decision junctures are indicated by a rhombus, as in the following generic two-branch decision: Here is the same decision juncture represented in pseudocode:

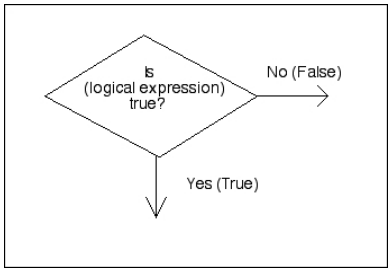


Figure 7: The quintessential decision, represented by flowchart.

if (logical expression) then
    true part
else
    false part
end if

REMARKS: In the same way that computers evaluate complicated operations by considering only two operands at a time, decisions are also *binary*. That is, no matter how complicated the decision structure, at any given time, the choice is always the following: choose one of but two branches based upon some criterion.

3.1 Relational Operators

Most logical expressions involve *relational operators*. Relational operators are to logical expressions what arithmetic operators are to mathematical expressions. Table 2 presents basic relational operators and their symbols in MATLAB.

Meaning	MATLAB
less than	$<$
greater than	$>$
less than or equal to	$<=$
greater than or equal to	$>=$
equal to	$==$
not equal to	$\tilde{}=$

Table 2: Relational operators in MATLAB.

3.2 Logical Constants

In the same way that MATLAB allows for integer, real, and character constants, it also allows for LOGICAL constants. There are only two distinct logical constants. Here are the logical constants in MATLAB:

MATLAB Logical Constants
true
false

3.3 Logical Variables

MATLAB supports logical variables in addition to integer, real, character, complex, and derived variable types. Logical variables are declared below:

```
temp = true;
temp2 = logical(0) % this command makes temp2 false
```

REMARKS: Logical variables are a nice feature of MATLAB, but they are really unnecessary. Why? (Anything that can be done with logical variables could as easily be done with integer variables that are assigned values of either 0 or 1.) For this reason, we will say little more about logical variables.

3.4 Simple Logical Expressions

Logical expressions, like logical constants and logical variables, may assume only the values of true (T) or false (F). Simple logical expressions involve two mathematical expressions bound together by a relational operator, as follows:

(mathematical expression 1) relational operator (mathematical expresson 2)

Consider the following type declaration commands and then evaluate the truth value of each of the simple logical expressions.

```
x = ones(1,1)*3.1; y = ones(1,1)*3.2;
a = ones(1,1)*2.0; b = ones(1,1)*(-6.0); c = ones(1,1)*4.5;
month = ones(1,1,'int32')*12;
```

- EX1: (x < y)
- EX2: (x == y)
- EX3: (B ^ 2 - 4.0*A*C == 0.0)

- EX4: (month = month / 12)

Can you foresee any problems with the logical expression in EX3 above? (Never, never test an equality with real numbers. Why?)

3.5 Logical Operators and Compound Logical Expressions

Compound logical expressions are two (or more) simple logical expressions conjoined by a *logical operator*. The primary logical operators, not to be confused with relational operators, are the conjunction (and), the disjunction (or), and the negation (not). Of these three, the first two are binary operators and the third is unary. The *conjuction* of two logical expressions is true if and only if *both* logical expressions are true. In contrast, the disjunction of a two logical expressions is true if *either* logical expression is true. And the negation of a logical expression is true if the original expression is false, and vice versa. In addition to the conjunction and disjunction, a less frequently used binary logical operator is *equivalence*. The equivalence test is true if and only if the two logical expressions have the same truth value. The *not equivalent* test

Logical Operator	Syntax
conjunction	&
disjunction	|
negation	~
equivalence	==
negated equivalence	~ =

Table 3: Logical operators in MATLAB.

(~ =) has the opposite truth value of the *equivalence* test.

Suppose (p,q) represents a pair of logical expressions, each of which may be either true or false. Thus, there are four possibilities to consider: (T,T), (T,F), etc. The *truth table* below (Table 4) summarizes (in MATLAB) the outcome of all possible logical operations on p and q.

p	q	p & q	p | q	p == q	p ~ = q	~ p
T	T	T	T	T	F	F
T	F	F	T	F	T	–
F	T	F	T	F	T	T
F	F	F	F	T	F	–

Table 4: Truth table summarizing logical operators.

Consider the following type declaration statements and evaluation each of the compound logical expressions accordingly:

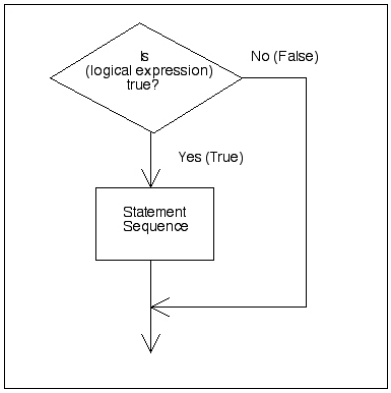


Figure 8: Flowchart of one-branch decision.

I = ones(1,1,'int32')*4; J = ones(1,1,'int32')*3;
Sales = ones(1,1)*5000; Travel = ones(1,1)*600;

- EX1: (I > 0) & (I < J)
- EX2: (I > 0) | (I < J)
- EX3: (~ (I < J))
- EX4: (Sales <= 5000.0) & (Travel < 600.0)

3.6 if Blocks in MATLAB

In this section we consider MATLAB's basic decision structure: the if block. The variations upon this theme are endless, but let's start very simple, with one-branch decisions.

3.6.1 One-Branch Decisions: the if Construct

A one-branch decision structure is shown by flowchart in Fig. 8. Here is the same structure in pseudocode:

if (logical expression) then
    statement sequence
end if

REMARKS: Even though we are calling this a one-branch decision, the choice is still binary: do something or do nothing at all. Can you think of common, everyday situations that involve one-branch decisions? Yes, warning systems. For example, tornado warning sirens, low fuel lights, fire alarms, etc. If trouble is brewing, do something. If all is well, do nothing.

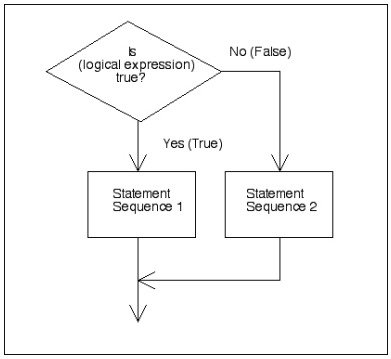


Figure 9: Flowchart of two-branch decision.

Suppose **Gallons_in_Tank** is a double variable that stores the number of gallons of gas remaining in you tank, a value frequently updated by your car's computer. Also suppose that you have a digital dashboard, effectively your car's computer screen. Here is a small MATLAB if block (that should be imbedded inside a loop) to put a warning message on the dashboard whenever fuel reserves run low.

```
if(Gallons_in_Tank < 2.0)
    fprintf(' Stop for Gas Soon!! \n')
end
```

REMARKS: The "innards" of an if block should always be indented for clarity, and the depth of the indentations should be consistent throughout your program.

3.6.2 Two-Branch Decisions: the if-else Construct

A two-branch decision structure is shown by flowchart in Fig. 9.

Here is the same structure in pseudocode:

if (logical expression) then
    statement sequence 1
else (otherwise)
    statement sequence 2
end if

3.6.3 Many-Branch Decisions: the if-elseif Construct

What if one needs to choose between, say, three or more choices? If decisions by computer are always binary choices, that would seem to limit the types of decisions that can be made. Not really. Many-branch decisions are built up from a finite sequence of binary choices.

To allow many-branch decisions, MATLAB provides the if-elseif construct, as shown in the "Courier" example below. Package couriers (UPS, FedEx, DHL, etc.) often charge differentially for small package delivery depending upon how far the destination lies from the point of origin. This is sometimes accomplished by the use of *zones*, whose boundaries are concentric circles of different radii centered at the point of origin. If the destination lies in Zone 1, for example, the lowest rate is available. If the destination is in Zone 2, the next-to-lowest rate applies, and so on. Here is the decision block of **Courier.m**.

```
% Courier.m
Zone = ones(1,1,'int32');
Rate = zeros(1,1);

Zone = input(' Please enter Zone of destination as an integer ');

if (Zone == 1) % local delivery
    Rate = 5.50;
elseif (Zone == 2) % regional delivery
    Rate = 6.50;
elseif (Zone == 3) % moderately distant
    Rate = 8.50;
elseif (Zone == 4) % transcontinental
    Rate = 10.50;
else
    fprintf(' Sorry, you have entered an invalid Zone \n');
end

...
```

How many branches does the decision structure of **Courier.m** have? What happens if the user enters 9 for the Zone?

REMARKS: The ELSE block is optional. However, it is often used as a "safety net" to warn the user of erroneous entries.

Figure 10 shows via flowchart how many-branch decisions can be built up as a sequence of binary choices.

3.6.4 Nested Decision Structures

Decision blocks can be nested inside one another like Russian dolls. The rules of good structured programming require that deeper levels of nesting to be indented more deeply and that indentations are consistent at each level.

The following two MATLAB decision blocks accomplish *exactly* the same task. Both decision blocks

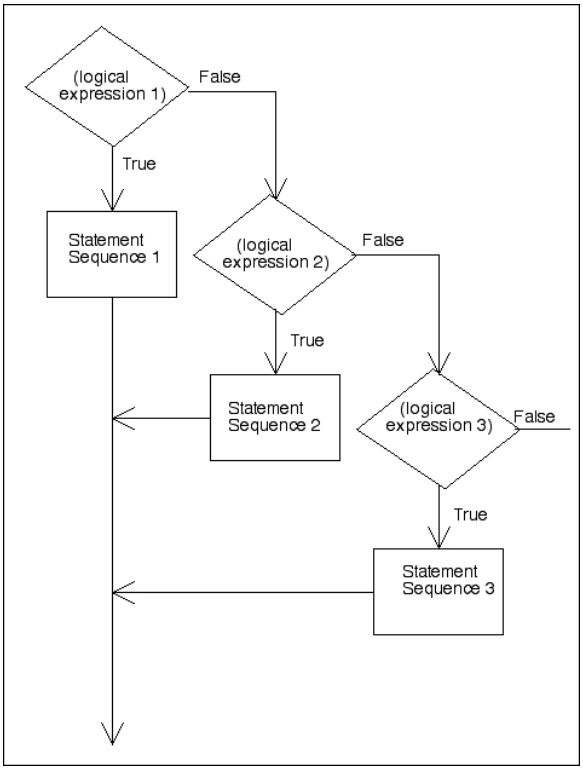


Figure 10: Flowchart of a many-branch decision.

use two logical variables Flag1 and Flag2.

Whereas the first block uses nested if blocks, the latter block uses compound logical expressions. Which do you prefer? Why?

EX1: Nested decision blocks

```
if (Flag1) % Flag 1 is true
    if (Flag2)
        fprintf(' (True,True) case \n');
    else
        fprintf(' (True,False) case \n');
    end
else % Flag 1 is false
    if (Flag2)
        fprintf(' (False,True) case \n');
    else
        fprintf(' (False,False) case \n');
    end
end
```

EX2: Logically equivalent to EX 1 using compound logical expressions

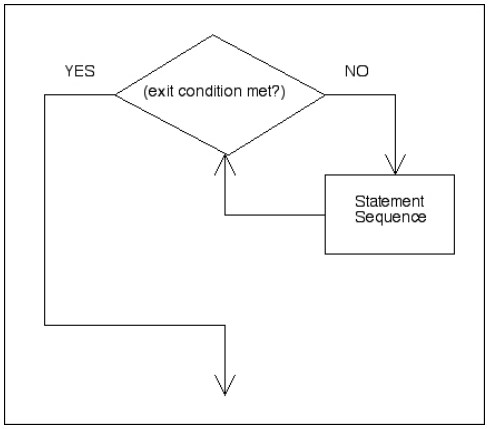


Figure 11: Flowchart showing a loop with the exit condition at the top.

```
if ( Flag1 & Flag2 )
    fprintf(' (True,True) case \n');
elseif ( Flag1 & (~ Flag2) )
    fprintf(' (True,False) case \n');
elseif ( (~ Flag1) & Flag2 )
    fprintf(' (False,True) case \n');
elseif ( (~ Flag1) & (~ Flag2) )
    fprintf(' (False,False) case \n');
end
```

4 Part I: Programming for Repetition

An advantage that computers have over humans is that computers do not tire or get bored by repetitive tasks. Consequently, most computer programs involve repetition or iteration that would be exceedingly tedious if not impossible for humans. The generic term for the structure of repetition in a computer program is a *loop*.

Figure 11 presents one variation of a loop via flowchart. What do you notice that is unusual about the "flow" within a loop? If you are on the ball, you will notice that, if the exit condition is not met, control is transferred back to the top of the loop. This is the first and the only incidence in this course of "upward flow."

In pseudocode, loops are much cleaner. Here is the same loop structure in pseudocode.

repeat until (exit condition) is met
| *statement sequence*
| ___

The statement sequence within the large bracket is executed again and again until the exit condition is satisfied. In general, the exit condition is a simple or compound logical expression.

A subtle variation on the loop structure above is the following:

repeat while (exit condition) is met
| *statement sequence*
| ___

What is the primary difference between these two loops?

A single pass through a loop is called an *iteration*. What happens if the exit condition is NEVER satisfied? This is a very undesirable situation termed an *infinite loop*. Even the fastest computer in the world cannot complete an infinite loop in finite time. All this implies that *something must change* with every iteration. Otherwise, if the exit condition fails on the first iteration, it will fail on every subsequent iteration.

Before we go on to loop syntax in MATLAB, here is a summary of important points to keep in mind about loops.

REMARKS:

1. Most computer programs involve repetitions, tedious to humans, but routinely accomplished by computers via "loops."
2. A loop is the *only* instance of "upward flow" to be encountered in this course.
3. Something must change during each iteration of the loop. Otherwise, the loop may run indefinitely, in which case it is known as an infinite loop.

For reasons that are not entirely clear to the authors, students for whom other aspects of programming come naturally may still encounter conceptual difficulties with loops. One source of difficulty is upward flow. So expect problems, ask lots of questions, and write many, many loops for practice until the idea becomes second nature.

4.1 Loop Structure I in MATLAB: while

The while construct in MATLAB, simply put, is a powerful and versatile loop syntax. Here is MATLAB syntax for the while construct:

```
while (logical expression)
    statements
end
```

The logical expression above is called the *termination condition* or the *exit condition*. Whenever the exit condition is satisfied, control is passed to the *first executable statement following the loop.* In the vernacular, we say the program has exited the loop or "jumped out of the loop." If the exit condition is *not* satisfied, when the end statement is encountered, control will be transferred back to the beginning of the loop, denoted by the while statement, and the "innards" of the loop will be repeated.

We now present two common algorithms that require loops for their accomplishment: 1) a counting algorithm, and 2) a summation algorithm. Each is presented first in pseudocode and then in MATLAB.

EX1: A counting algorithm

Pseudocode:

$i \leftarrow 0$
repeat while $i < 100$
| $i \leftarrow i + 1$
| *print value of* i
| __

Note that, in pseudocode, the left-arrow symbol ($\leftarrow$) denotes "assign," or more specifically, "replace what is on the left by the value of what is on the right."

MATLAB while:

```
Counter=zeros(1,1,'int8');
...
while (Counter < 100)
    Counter = Counter + 1;
    fprintf('Counter = %d \n', Counter);
end
```

EX2: A summation algorithm

ASIDE: Carl Friedrich Gauss (1777-1855) is generally considered the most brilliant (but not the most prolific) mathematician who has ever lived. At the age of seven, he entered elementary school. Like many gifted students, he was bored, and boredom led to minor troubles with his teachers. To keep him occupied, his teachers gave him mathematical tasks, presumed to be time-consuming, like summing the integers from 1 to 100, or 1 to 1000, etc. His mathematical prowess was firmly established when, as an elementary student, Gauss derived a shortcut for summation that works for any upper limit, which flabbergasted his teachers. Here is the derivation of Gauss' trick.

Let S represent the sum of the integers from 1 to n, where $n \geq 1$ is otherwise arbitrary. Thus

$$S = \sum_{i=1}^{n} i \tag{1}$$

Now suppose we twice write out the summation explicitly, once in ascending order, and a second time

in descending order. Because addition is commutative and associative, we may add in any order and still obtain the same sum.

$$S = 1 + 2 + 3 + \ldots + (n-1) + n \quad (2)$$
$$S = n + (n-1) + (n-2) + \ldots + 2 + 1 \quad (3)$$

Gauss noticed that each columnn on the right-hand side above has the same sum, namely $(n+1)$. Moreover, there are n columns. Therefore, summing down the columns, we obtain

$$2S = (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) = n(n+1) \quad (4)$$

The desired result is obtained by dividing both sides by 2. That is,

$$S = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad (5)$$

We can verify Gauss' shortcut. What is the sum of the integers from 1 to 10? $S = 10(10+1)/2 = 55$, which is correct, of course. We can now use Gauss' trick as a check for our summation algorithm. Morever, this result will become important later when we count the operations of selected algorithms.

Summation algorithms require both a counter and an *accumulator*. The purpose of the accumulator is to store the current running total. If we are summing integers, then both the counter and the accumulator should be integer variables.

Pseudocode:

$i \leftarrow 0$
$s \leftarrow 0$
repeat while $i \leq 100$
| $i \leftarrow i+1$
| $s \leftarrow s+i$
| ___
print value of sum s

Here is the pseudocode translated into MATLAB, for summation from 1 to 10.

MATLAB while:

```
Counter=zeros(1,1,'int8'); Accumulator=zeros(1,1,'int8');
...
while (Counter < 10)
    Counter = Counter + 1;
    Accumulator = Accumulator + Counter;
end
fprintf('Sum = %d \n', Accumulator);
```

How would you change the MATLAB code to sum from 1 to 1000? From 1 to n? From 2 to n (even) by two's?

4.2 Loop Structure II in MATLAB: for

The counting and summation algorithms presented above make use of MATLAB's while construct. The while construct requires the programmer to explicitly declare and increment an integer counter variable that keeps track of the number of iterations. In contrast, in MATLAB's *for* construct, the programmer must declare an integer counter variable *but the indexing of the counter variable is done implicitly*. The main problem that students have with loops in MATLAB is confusing the syntax of these two different loop constructs. Here then is the MATLAB syntax of *for* loops:

```
for control_variable = initial_value: increment :final_value
    statement sequence
end
```

The variable **control_variable** must be declared and should be of INTEGER type. The values **initial_value**, **final_value**, and **increment** may be supplied as variables or constants of INTEGER type. If supplied by variables, the variables must be declared. [1]

Here is a simple counting algorithm with the automatic counter-controlled loop construct in MATLAB:

EX3: MATLAB Summation Algorithm with *for* Loop

```
Counter=zeros(1,1,'int8'); Accumulator=zeros(1,1,'int8');
...
for Counter=1:1:10 % normally type Counter=1:10, 1 is the default increment
    Accumulator = Accumulator + Counter;
end
fprintf('Sum = %d \n', Accumulator);
```

EX4: Modify the loop above to sum from 1 to 100.
EX5: Modify the loop above to sum from 1 to 100 by 2's.
EX6: Modify the loop above to sum *backward* from 100 to 1 by 4's.

QUESTION: How would you write the above code in MATLAB? Hint: type help continue

4.3 The continue Command in MATLAB

One pass through a loop is called an *iteration*. The continue command in MATLAB terminates the execution of an iteration of a loop, whereby control jumps back to the top of the loop rather than to the first executable statement following the loop. In practice, all statements *following* the continue command are by-passed for any iteration for which the *logical expression* preceding continue is satisfied. Here is an example, without comments, of a program that exploits the continue command.

[1] MATLAB and some other programming languages allow the use of loop-control variables of type double. The use of double numbers to control loops is a *terrible* programming practice for reasons to be addressed later.

```
count = ones(1,1,'int32');
...
for count = 1:1:30
    if (count == 13)
        continue;
    end
    fprintf(' count = %d \n',count);
end
```

QUESTION: What happens in the above MATLAB code?

4.4 A Detailed Example: Program Fibonacci

The Fibonacci numbers, an integer sequence, are ubiquitous in mathematics and in nature. They show up unexpectedly, as in the spiral pattern of the chambered nautilus or the patterns of seeds in pine cones. The first two Fibonacci numbers are zero and one. Each successive Fibonacci number sums its two immediate predecessors, i.e. $F_{n+2} = F_{n+1} + F_n$. Repeating this process *ad infinitum* generates the following sequence:

$$F_n = \{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...\} \tag{6}$$

QUESTION: If $a_0 = 0$, what is a_{11} in the Fibonacci sequence?

Because sequences are infinite by definition, even the fastest computer cannot generate all Fibonacci numbers. Still, it might be fun to have MATLAB generate, say, the first n Fibonacci numbers, where n is an integer to be read by the computer program. How many variables do we need? What type of syntax will we use?

4.5 Nested Loops

Many programs make use of *nested loops*, a loop within a loop, for example. Here is an example of a simple MATLAB program that uses doubly nested loops to print out the multiplication table for the integers from 1 to 10. How many products are computed in total? Please comment the code below.

```
%%\begin{variables}
operand1=ones(1,1,'int32'); operand2 = ones(1,1,'int32');
product=ones(1,1,'int32');
%%\end{variables}

for operand1=1:10
```

```
    fprintf('\n');

    for operand2=1:10
        product = operand1 * operand2;
        fprintf('%d x %d = %d \t ', operand1, operand2, product);
    end

end
```

It is not uncommon for programs to have triply are even quadruply nested loops. For example, a search over three-dimensional space (say, for a maximum value) requires a triply nest loop: one loop over the x direction, one over the y direction, and one over the z direction. The loops are not independent, but are nested each within the next. Why?

5 Part I: Modular Programming

In addition to the liberal use of white space, indentation, comments, and descriptive variable names, good structured programming depends on *modular programming*. Modular programming is the use of subprograms (also called procedures) tailored to accomplish specific tasks: finding the maximum, finding a root, solving a system of linear equations, inputing necessary data, outputing results, etc. Modular programming exploits the divide and conquer paradigm. No task is too big if it can be broken down into a set of small subtasks. It is not uncommon for computer programs to have tens of thousands or even hundreds of thousands of lines. The proper functioning of such massive programs is possible only because of modular programming, in which each subprogram is at most a few hundred lines long.

Modular programming has many advantages:

1. Functions can be developed and debugged independently of the main program.
2. The logic of the main program is greatly simplified.
3. Highly efficient functions that exist in public-domain libraries can be called for specific tasks, such as solving a system of linear equations.
4. The same function may be called time and again without replication of its source code in the main program.

The following two MATLAB programs accomplish exactly the same tasks. Which do you prefer and why?

Sort_long.m :

```
%Sort_long.m
% Takes as input three real numbers and sorts them in ascending order

%%begin{variables}
x1=ones(1,1); x2 = ones(1,1); x3=ones(1,1); swap = ones(1,1);
%%\end{variables}

x1 = input('Please enter the first number ');

x2 = input('Please enter the next number ');

x3 = input('Please enter the next number');

if (x1 > x2)
    swap = x1;
    x1 = x2;
    x2 = swap;
end

if(x1 > x3)
    swap = x1;
    x1 = x3;
    x3 = swap;
end

if (x2 > x3)
    swap = x2;
    x2 = x3;
    x3 = swap;
end

fprintf('In ascending order, the numbers are: %g  %g  %g \n', x1, x2, x3);
```

Sort_function.m :

```
function Sort_function()
% Takes as input three real numbers and sorts them in ascending order

%%begin{variables}
x1=ones(1,1); x2 = ones(1,1); x3=ones(1,1);
%%\end{variables}

x1 = input('Please enter the first number ');
```

```
x2 = input('Please enter the next number ');

x3 = input('Please enter the next number ');

[x1,x2] = order(x1,x2);

[x1,x3] = order(x1,x3);

[x2,x3] = order(x2,x3);

fprintf('In ascending order, the numbers are: %g  %g  %g \n', x1, x2, x3);


function [a,b] = order(a,b)

%%\begin{variables}
swapping = ones(1,1);
%%\end{variables}

if(a>b)
    swapping=a;
    a=b;
    b=swapping;
end
```

- In order to use subfunctions, the main program must also be a function. The name of the function, in this case Sort_function(), must match the name before the .m in the file name.
- Variables are local to functions, i.e. x1, x2, and x3 do not exist in the subfunction order.

5.1 Syntax for Functions in MATLAB

The writing of functions in MATLAB is extremely straightforward. For instance, recall that we earlier presented a MATLAB code to calculate the sum of the first 10 integers. Here is how to write it as a function:

MATLAB SumFunction.m:

```
function Accumulator = SumFunction(n)
```

```
%%\begin{variables}
n_int = int32(n);
Counter=zeros(1,1,'int32');
Accumulator=zeros(1,1,'int32');
%%\end{variables}


while (Counter < n_int)
    Counter = Counter + 1;
    Accumulator = Accumulator + Counter;
end

fprintf('Sum = %d \n',Accumulator);
```

First, note the syntax of the function declaration. The variable(s) returned, in this case Accumulator, is to the left of the equal sign. The name to the right of the equal sign is the name of the function. The variable(s) in parentheses is the input from the user. So, if the user wanted to call this function to compare the results with the code from earlier in the notes they would type SumFunction(10).

The other new command in this program is the second line of the code, namely `n_int` = int32(n). This command takes the user input n, which in MATLAB is of type double by default, and makes a new int32 variable `n_int`. This programming construct is to emphasize the importance of looping through integers not double precision numbers.

QUESTION: Comment the above MATLAB code in these notes to make sure you understand each line of the program.

6 Part I: Arrays

An *array* is a set of contiguous memory locations all associated with the same variable name. Arrays may be one-dimensional (1D) or multi-dimensional. Another name for a 1D array is a *vector*. A two-dimensional (2D) array is also called a *matrix*. Most programming languages allow arrays of dimensions considerably higher than three; however, we will focus here on 1D and 2D arrays.

6.1 One-Dimensional Arrays: Vectors

Figure 12 illustrates a REAL 1D array named "A" and of length five. In MATLAB, to define a similar array is

```
A=zeros(1,5);
```

How are the various components of a 1D array distinguished from one another? Mathematically speak-

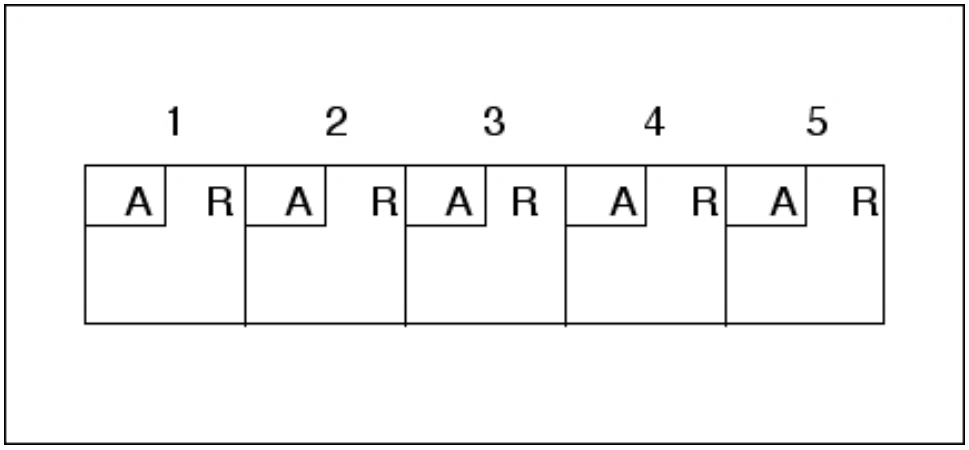


Figure 12: One-dimensional real array "A."

ing, consider the vector $\vec{a} = [1, 2, 4, 6, 9]$. We denote the 4th *component* of $\vec{a}$ by subscript as a_4. In the example, $a_4 = 6$. In general, the ith component is a_i. In MATLAB and many other programming languages, the individual components of vectors are addressed by their respective *indices*. For example, consider the following snippet of MATLAB code:

```
A(5) = 17.;
A(5) = A(5) + 8.;
fprintf('%d \n', A(5));
```

Explain what happens when this sequence of commands is executed.

6.1.1 Loading 1D Arrays

Several methods are available for loading data into 1D arrays. We will discuss two methods: 1) array constants, 2) input loops.

Array Constants

Just as a scalar variable can be initialized by an assignment statement with a constant on the right-hand side, an array can be initialized by an **array constant**. For example, for the real 1D array "A" above, the following assignment statement would distribute the five values in order to the five components of "A." That is, after the array assignment, A(1)=4.7, A(2)=9.6, ..., and A(5)=7.9.

```
A=[4.7, 9.6, 3.2, 1.8, 7.9];
```

Input Loop

A second way to load arrays is by an *input loop*. Consider first the following statement sequence to load array "A" from keyboard input:

```
A(1)=input('Please enter a value for A(1) ');
A(2)=input('Please enter a value for A(2) ');
A(3)=input('Please enter a value for A(3) ');
A(4)=input('Please enter a value for A(4) ');
A(5)=input('Please enter a value for A(5) ');
```

Such a construct is OK for loading an array of relatively short length, but would be very cumbersome for an array of length, say 100. Here is an equivalent construct for loading array "A", that with slight modification would work for an array of any length.

```
i=zeros(1,1,'int32')
...
for i = 1:5
    A(i) = input('Please enter a value ')
end
```

Implied DO Loop

In order to use a single input command in MATLAB, knowledge about how MATLAB handles arrays is extremely important. For instance, the command

```
A(1:5)=input('')
```

will input a vector of length five if entered as

```
[4.7, 9.6, 3.2, 1.8, 7.9]
```

In fact, if you just type A=input(''), A can be an array of any length input by the user.

6.2 Two-Dimensional Arrays: Matrices

MATLAB allows for arrays of one, two, three, and higher dimensions. For our purposes, we will be content to consider only 2D arrays (matrices) in addition to the 1D arrays already considered.

The following type-declaration statement in MATLAB will create a double 2D array (matrix) "A" that has two *rows* and three *columns*, as shown in Fig. 13:

```
A=ones(2,3);
```

Note that the first number in parentheses specifies the number of rows and the second the number of columns.

Like 1D arrays, 2D array *elements* are referenced by their indices. The first index is the *row index* and the second is the *column index*. Thus, for example, the assignment statement below will place the real number 3.7 in the 3rd element of the top row of Fig.13.

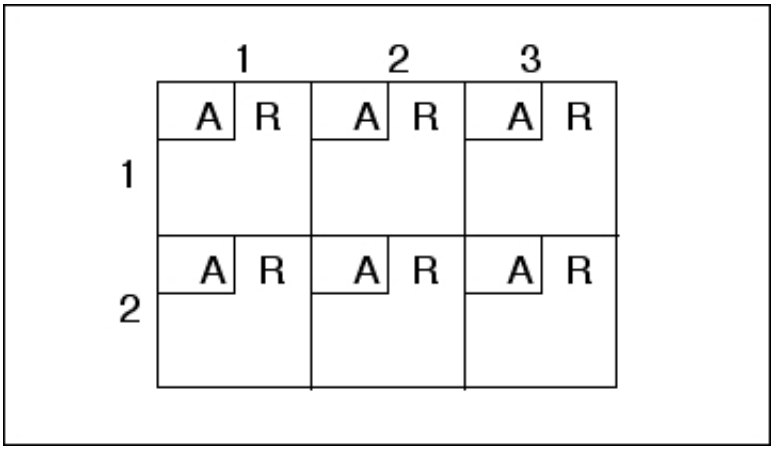


Figure 13: Two-dimensional real array “A”.

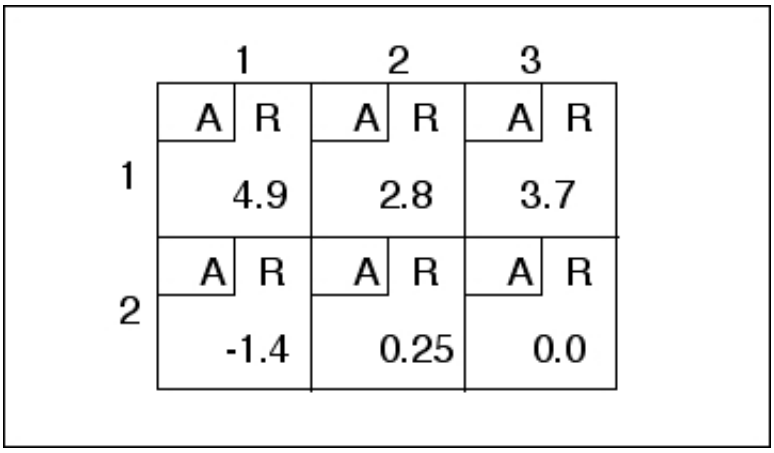


Figure 14: Filled 2D real array “A”.

A(1,3) = 3.7;

6.2.1 Loading 2D Arrays

Like 1D arrays, 2D arrays can be initialized (loaded) in many different ways: by array constants or by loops over row and column indices. Suppose we wish the elements of array “A” to have the values as shown in Fig. 14.

If we wish to fill array “A” by columns, then the data should be ordered as follows:

A = [4.9, 2.8, 3.7; -1.4, 0.25, 0.0];

Here is a nested loop in MATLAB that is a more general technique to obtain input for the matrix A:

```
Irow=ones(1,1,'int8'); Jcol=ones(1,1,'int8');
...
for Irow = 1:2
    for Jcol = 1:3
        A(Irow,Jcol) = input('Please input a number ');
    end
end
```

6.2.2 2D Arrays as Arguments to Subprograms

For the sake of illustration, consider matrix multiplication. The product (AB) of the two matrices A and B is defined if and only if A is $m \times p$ and B is $p \times n$, in which case the product is $m \times n$. That is, the number of columns of the first matrix must match the number of rows of the second matrix.

With this background, now consider the first few lines of a MATLAB function that computes the product of two matrices, C = AB:

```
function C = matrix_multiply (A, B)
...
[m,p1] = size(A);
[p2,n] = size(B);
if (p1 == p2)
```

A nice, but potentially dangerous, feature of MATLAB is that the above function will work when passing A and B as scalars and when passing A and B as vectors. The programmer must take great care to make sure that the correct values (sizes) are being passed between functions. The size command in MATLAB returns the row and column size in the first and second variable, respectively. This is the first time we have encountered to variable to the right of the equal sign; however, it comes as no surprise that MATLAB passes back the arguments as a vector, i.e [a,b].

6.3 A Relative Comparison of Fortran, MATLAB, and C

Before leaving the subject of arrays, we wish to discuss some of the relative advantages and disadvantages of three common high-level programming languages. MATLAB, for example, was specifically designed to facilitate matrix/vector operations. In fact, MATLAB assumes that all mathematical objects are arrays unless told otherwise. As a consequence MATLAB automatically dynamically allocates memory for arrays. This greatly simplifies programming syntax. On the other hand such "transparency" is not necessarily optimal for beginning programmers, who may benefit from learning basic procedures such as type declaration and memory allocation.

Moreover, neither C nor Fortran distinguishes between row vectors and column vectors. In either,

a vector is a vector. In contrast MATLAB treats a vector as a degenerate matrix and distinguishes row vectors from column vectors. In this regard it is more precise and more in keeping with the conventions of linear algebra.

On the other hand, Fortran has an advantage over both C and MATLAB in that array-index defaults can be overridden. In C, array indices begin at 0, a value that cannot be changed. In MATLAB, array indices begin at 1, which also cannot be overridden.

Regarding complex numbers, both MATLAB and Fortran allow for COMPLEX arithmetic; C does not provide the COMPLEX designation as standard equipment.