Structured Programming and Applications for Engineers

Lecture notes for
BCEE 231

by K. H. Ha, Ph. D.
Professor of Engineering
Concordia University

©Copyright by K. H. Ha, 2015
Preface

"When I learned to program, you were lucky if you got five minutes with the machine a day. If you wanted to get the program going, it just had to be written right. So people just learned to program like it was carving stone. You sort of have to sidle up to it. That’s how I learned to program."


This set of Lecture Notes is used in the course BCEE 231: Introduction to Computer Applications for Building and Civil Engineers at Concordia University. Its 12 Chapters are divided into two parts:

Part I (Chapters 1 to 4): The methodology of procedural programming (which is independent of computer languages), and the basic elements of the programming languages C & C++ (which are the current languages of choice for software development).

Part II (Chapters 5 to 12): The applications of procedural programming to engineering sciences and mathematics.

This text supplements the lectures, software documentation, and other online instructional material.

1. Objective

The immediate objective of the course is to develop skills for computer programming which is essential for computation in modern engineering design. The long-term objective is to develop the ability to think, organize and synthesize information. These are the skills that ensure success in your engineering study and career.

2. The Art of Learning

Much of university education is devoted to exposing concepts and some limited applications illustrating these concepts. In such an environment, the effectiveness in learning depends on individuals' perception of concepts.
A passive mind: We may look at a rolling rock (Fig. 1.1a) and say: it's just a rolling rock, or we may look at a pile of stone and say: it's just a pile of stone. The observations stop there, dead, not interesting and mostly useless\(^1\). A collection of such observations does not form a cohesive whole. It eventually becomes irrelevant and forgettable.

An active mind: An inquisitive mind may look at exactly the same thing, and try to understand/discover its nature\(^2\), and attempt to associate that with other concepts, synthesize them, leading to new applications (Fig.1.1b, c). That is effective learning\(^3\) where the mind is alive and active. The knowledge gained is alive, active, and then it becomes part of us without being memorized. An active mind can combine and integrate different concepts as shown in Fig. 1.1c.

\(^1\) This is similar to looking at a formula as a collection of symbols where the application is limited to substitution of numbers into symbols.

\(^2\) A round object can roll with minimal force. A pile of rocks stands up because individual rocks are pressed against each other. Deeper pursuit of the nature of things leads to questions such as: Why does a round object roll more easily? Why does a pile of rocks stand up when the rocks press against each other?

\(^3\) There are lots of formulas in mathematics and engineering sciences. To understand the nature of a formula, we find answers to the questions: What is its basis? What are its limitations? How does it relate to other concepts? Why does it make sense? Why are these parameters relevant? Most of the answers are in the proof leading to the formula.
Part I: Fundamentals of Procedural Programming

Chapter 1: Expressions and Programs

If I have ever made any valuable discoveries, it has been owing more to patient observation than to any other reason -- Isaac Newton

The important thing is not to stop questioning -- Albert Einstein

Contents

Preface ii
  1. Objective ii
  2. The Art of Learning ii
Part I: Fundamentals of Procedural Programming 1
Chapter 1: Expressions and Programs 1
  1.1 Introduction 2
    1. Computer Hardware and Software 2
    2. CMAP: The Environment for Learning and Computing 4
  1.2 Computers Do Not Think 4
  1.3 Programs for Simple Computations 6
    Sample Program 1.1 - Simple Calculator-Type Computations 6
    Sample Program 1.2 - Solutions of Equations of One Unknown 8
    Practice Drills 1.1 9
  1.4 Expressions 10
    1. Variable Names 12
    2. Operators 13
    3. Expression Evaluation 13
    4. Functions 14
    5. Subexpressions 15
    Practice Drills 1.2 17
  1.5 Types of Stored Information 19
    1. Data Types in CMAP 19
    2. Internal Representation of float 20
    Practice Drills 1.3 21
  1.6 The Power-Tools: Built-in Functions 24
1.1 Introduction

1. Computer Hardware and Software

Hardware: Hardware is the material things that we can see inside the box. Calculators share many of the following hardware components of computers:

- Input device for receiving input: e.g. keyboard, mouse, pen, and modem.
- Output device for displaying output: e.g. printer, monitor screen, modems.
- Memory storage for storing data and instructions for execution.
- Arithmetic & logic unit for elementary computations and comparison of numbers.
- A central processing unit for coordinating the other components, avoiding conflict of demands.
- Secondary storage: hard disks, flash drives.

Software: Without software, a million-dollar super computer is not as smart as the cheapest toaster\(^4\). A software package is a collection of machine-executable instructions that could be fed, one by one, to the central processing unit for it to act.

---

\(^4\) A toaster has built-in mechanical sensor to pop up the slice of bread when it's hot enough. So it's smart. A computer without software is just about as good as an unplugged toaster.
There are many layers of software running in a computer. Each layer is built on top of others, and ultimately, making computers easier to use.

Languages: To program is to develop software that instructs computer to act. A program uses a language for communication. Common computer languages are high-level languages\(^5\). They need the support of other software, which works behind the scene, for them to work.

At the lowest level is the machine language (directly understood by computers). Few persons could, or should use machine language. Programs in high-level languages must be machine-translated into machine language or interpreted for execution.

Source code: Source code is the original code in a high-level language. Source code of commercial programs is a trade secret.

Executable code: Executable code\(^6\) is the code already translated into machine language, ready to run. The translation process\(^7\) is called "program compilation". Once the translation is done, all program instructions are cast in stone\(^8\).

Interpreters: Instead of translating the source code, an interpreter interprets and executes the source code at the time of execution. Since CMAP is an interpreter, it can execute source code without invoking complex tools such as compiler\(^9\), assembler\(^10\), linker\(^11\) and loader\(^12\). On the other hand, interpreting code is slower because it takes time to analyse each statement in the code each time it is executed.

\(^5\) High-level languages include, to name just a few, FORTRAN, BASIC, Pascal, PL1, COBOL, ALGOL, FORTH, LISP, C, C++, C#.

\(^6\) Executable code is distributed as exe-files.

\(^7\) Translation is done by another software called the "compiler", which is specific to the language of the source code. The output of translation is further processed by two more software called the "assembler" and the "linker". The final product is the executable code.

\(^8\) Some hackers familiar with machine or assembly language can tweak a few crucial machine language instructions to achieve their devilish aims (such as to disable copy protection, or to override authentication check).

\(^9\) A compiler is the software that translates high-level source code into assembly language, which is closer to machine language.

\(^10\) The assembler (another software) translates the assembly source code (output of the compiler) into machine language.

\(^11\) The linker (another software) links the disparate output of the assembler into a cohesive machine executable program.

\(^12\) [https://en.wikipedia.org/wiki/Loader_(computing)](https://en.wikipedia.org/wiki/Loader_(computing))
Learning programming with an interpreter is easier because its environment is more dynamic and interactive\textsuperscript{13}, allowing the user to do things that’s not feasible with machine language code.

2. CMAP: The Environment for Learning and Computing

CMAP is the only software package that provides all of the following advantages:

- The methodology of procedural programming and the basic elements and syntax of the C-language. This will build a solid background of programming for numerical computation.

- A large set of built-in functions for graphics and mathematics. This greatly facilitates the application of programming for solution of significant and interesting problems in engineering mechanics and mathematics, paving the way for other engineering courses.

- A user-friendly graphical environment for quick testing and verification of ideas, hypotheses and programs. This makes learning about programming and its applications so much easier and even fun.

CMAP’s interactive environment provides high-level support that shields us from a multitude of complexities commonly associated with conventional programming. This will be utilized and demonstrated in every lecture, where tips on good practice in programming and development are also presented.

1.2 Computers Do Not Think

A computer may appear to have "intelligence" or "thinking ability" only if it was cleverly programmed to simulate thinking. Without such a program, computers do not have the ability of reorganizing data, synthesizing information in order to reason like human.

Example: Being human, we can think, organize/synthesize information and then take actions in an appropriate sequence, and thus we have no trouble solving a problem like this:

What is the value of $C = A + B$ given that $A$ is 1, $B$ is 2?
And what if $A$ is 100 times bigger and $B$ 10 times bigger?

\textsuperscript{13} Source code in BASIC, FORTH, LISP, etc.. could be run in an interpreter environment or compiled into machine language. The machine language version has fewer features because of its cast-in-stone status.
To solve this problem using a conventional programming language, we need to issue instructions like these:

- Step 1: Put 1 into A
- Step 2: Put 2 into B
- Step 3: Add A and B, and then put the result into C. Print C
- Step 4: Multiply A by 100 and put the result back into A
- Step 5: Multiply B by 10 and put the result back into B
- Step 6: Add A and B, and then put the result into C. Print C

In general, computers will strictly follow the given instructions, no more, no less.

Copy and paste the following program (into a blank document in CMAP) and execute it (Ctrl-E).

```c
main()
{
    A = 1;   // Put 1 into A
    B = 2;   // Put 2 into B
    C = A+B; // Do A+B and put result into C
    view();  // Display current variables
    A = A * 100;   // Do A*100 and put into A
    B = B * 10;   // Do B*10 and put into B
    C = A+B; // Do A+B and put result into C
    cat();  // Memory dump of current variables
    view(); // Inspect current variables
    gotourl("https://en.wikipedia.org/wiki/Compuuter_programming "); // Display a web page
}
```

14 Copy a pdf-format program and paste it into CMAP may introduce mysterious errors. When this happens, watch out for:
   a) A dash or hyphen being used in the place of the minus sign
   b) Smart (curly) quotes being used in the place of straight quotes
   c) A long quote broken into 2 lines.
Thus, to learn programming (e.g. CMAP, C, C++, etc.) is to learn to issue instructions, to play within the rules of the language. We do not assume that the computer can guess, think or reason from the given data. Thus, one needs to know the available instructions, their actions (built-in capability\textsuperscript{15}) as well as limitations.

1.3 Programs for Simple Computations

Sample Program 1.1 - Simple Calculator-Type Computations

Simple computations by calculators can be more quickly and conveniently carried out with a CMAP application program.

\begin{center}
\begin{tabular}{|l|l|}
\hline
Data & Compute\textsuperscript{16} \\
\hline
$L = 2\text{m}$ & $M_y = P\cos(\alpha)L$ \\
$P = 575\text{N}$ & $M_z = -P\sin(\alpha)L$ \\
$b = 80\text{mm}$ & $I_y = \frac{bh^3}{12}$ \\
$h = 140\text{mm}$ & $I_z = \frac{hb^3}{12}$ \\
$\alpha = 30^\circ$ & $\beta = \tan^{-1}\left[\frac{I_z}{I_y}\tan(\alpha + 90^\circ)\right]$ \\
\hline
\end{tabular}
\end{center}

\begin{Verbatim}
main()
{
    /* Sample Program 1.1*/
    Units: N (forces), m (lengths), degree (angles) */
    L = 2; P = 575; b = 80e-3; h = 140e-3; Al = 30;
    useoption("DEGREES"); // useoption("RADIANS");
    My = P*\cos(Al)*L; Mz = -P*\sin(Al)*L;
    \}
\end{Verbatim}

\textsuperscript{15} Simple capability such as "put a value into a variable name", or more complex as "download and display the web page".

\textsuperscript{16} The units of the computed quantities can be established from the units of the values used in the expression. One may guess at the physical nature of the computed quantities from their units. \url{http://physics.nist.gov/cuu/Units/units.html}
\[ I_y = h b^3/12; \quad I_z = b h^3/12; \]
\[ \text{Bet} = \text{atan}(I_z\tan(AL+90)/I_y); \]
\[ \text{Smax} = My*(b/2)/I_y - Mz*(h/2)/I_z; \]
\[ \text{cat}(); \]
\[ } // \quad \text{Ans:} \quad I_y = 5.97e-006 \text{ m}^4 \quad I_z = 1.829e-005 \text{ m}^4 \]
\[ // \quad My = 995.9\text{N.m} \quad Mz = -575\text{N.m} \quad Smax = 8.87e+006 \text{ N/m}^2 \quad (\text{or Pa}) \]

Type the program: Open a new document (click the icon New-Text-Doc or use the menu File > New Text Doc), or use an existing program document. After typing the program into the document, it can be edited and saved.

Execute the program:
- Click the red \texttt{Execute} icon
- or press the key Ctrl+E
- or select the menu Run > Execute

Any of the above actions will do two things:
- Automatic reset: Clear the memory, deleting all existing data and program.
- Automatic execution of the function main(), which is the only function that has this privilege\(^{17}\).

During program execution, the instructions\(^{18}\) in function main() are executed, one by one, left to right, from the top down.

Observe:
- The format & syntax of function main()
- Thanks to the proper use of delimiters such as

\[
() \{ \} , ; \\
/ * .... */
\]

the program is still readable even when the entire program is typed on ONE line with no comments or space between items:

\(^{17}\) If function main() is missing, Cmap only activates the user-defined functions for use in Calc-Mode. Source code to be compiled must have the main() function.

\(^{18}\) A program will blindly execute our instructions, and so, we'll get only what we ask, no more and no less.
main(){L=2;P=575;b=80e-3;h=140e3;Al=30;useoption("DEGREES");My=P*cos(Al)*L;Mz=-P*sin(Al)*L;Iy=h*b^3/12;Iz=b*h^3/12;Bet=atan(Iz*tan(Al+90)/Iy);Smax=My*(b/2)/Iy-Mz*(h/2)/Iz;cat();}

- Every variable in the right-hand-side of the equal sign must have a value in order for the (numerical) computation to be carried out
- The left-hand-side of the equal sign is the name for a container that contains the value of the expression in the right-hand-side
- Use of variable names facilitates subsequent computations, review and corrections
- Expressions written with meaningful names help understanding of relationships among variables better than using numbers.

Sample Program 1.2 - Solutions of Equations of One Unknown

The above approach is good for simple engineering problems. In the following solution presentation\(^19\) we show the data and equations in symbols. Note that each equation has only one unknown which can then be found immediately:

<table>
<thead>
<tr>
<th>Data</th>
<th>Compute the unknowns in the following equations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L = 0.3 \text{ m} ) ( d = 0.1 \text{ m} )</td>
<td>( N_a - P \cos(\beta - \alpha) = 0 \quad \Rightarrow \quad N_a = ? )</td>
</tr>
<tr>
<td>( P = 80 \text{ N} )</td>
<td>( V_a - P \sin(\beta - \alpha) = 0 \quad \Rightarrow \quad V_a = ? )</td>
</tr>
<tr>
<td>( \alpha = 30^\circ ) ( \beta = 45^\circ )</td>
<td>( M_a + P \cos(\beta)L \cos(\alpha) ) ( - P \sin(\beta)[d + L \sin(\alpha)] = 0 \quad \Rightarrow \quad M_a = ? )</td>
</tr>
</tbody>
</table>

\(^19\) This is a great way to present solution in assignments and exams (of most engineering courses). It shows clearly all that is essential. This presentation style is far superior to writing equations in numbers, and it facilitates quick, fair and accurate marking. Numerical values can later be substituted and results evaluated, but their accuracy now plays a small role in grading.
main()
{
    /* Sample Program 1.2
       Units: N (forces), m (lengths), degree (angles) */
    L = 0.3; P = 80; d = 0.1; Al = 30; Bet = 45;
    useoption("DEGREES");
    Na = P*cos(Bet-Al);   Va = P*sin(Bet-Al);
    Ma = -P*cos(Bet)*L*cos(Al) + P*sin(Bet)*(d+L*sin(Al));
    cat();
} // Ans:  Na = 77.3 N   Va = 20.7 N  Ma = -0.555 N.m

Practice Drills 1.1

1. Modify and run the Sample Program 1.1 for this data set:

   \[ L = 10 \text{ ft} \quad P = 325 \text{ lb} \quad b = 4 \text{ in} \quad h = 9 \text{ in} \quad \alpha = 45^\circ \]

   Ans:  \[ I_y = 48 \text{ in}^4 \quad I_z = 243 \text{ in}^4 \quad M_y = 27577.2 \text{ lb.in} \quad M_z = -27577.2 \text{ lb.in} \quad \beta = -78.8^\circ \quad S_{\text{max}} = 1659.74 \text{ psi} \]

2. Given this data set: \[ W = 1 \text{ kN} \quad d = 4 \text{ mm} \quad \alpha = 20^\circ \quad \beta = 48^\circ \]

   Compute:

   \[ T_1 = \frac{W}{\cos \beta \tan \alpha + \sin \beta}, \quad T_2 = T_1 \frac{\cos \beta}{\cos \alpha}, \quad A = \frac{\pi d^2}{4}, \quad \sigma_1 = \frac{T_1}{A}, \quad \sigma_2 = \frac{T_2}{A} \]

   Ans:  \[ S_1 = 57 \text{ MPa} \quad S_2 = 8.1 \text{ MPa} \quad T_1 = 721.68 \text{ N} \quad T_2 = 1013.49 \text{ N} \]
3. Refer to the situation in Sample Program 1.2 While keeping the vertical distance $H = L \sin(\alpha) + d = 0.25 \text{ m}$ constant, find the combination $(\alpha, d)$ that minimizes the maximum stress (at A) given by:

$$\sigma_{\text{max}} = \frac{N_d}{A_d} + \frac{M_a b}{2I_a}$$

where $b = 0.1 \text{ m}$, $t = 0.05 \text{ m}$, $A_a = bt$, $I_a = \frac{tb^3}{12}$

Ans: $\alpha = 33.5^\circ$ $d = 0.084 \text{ m}$

1.4 Expressions

Most statements in program code are expressions which carry out computations and/or invoking functions. Many are assignment expressions where right-hand-side values are assigned to left-hand-side variable names.

Run the following program, observe, and explain the output.

```cpp
main()
{
    /* Program to demonstrate the concept of Variable
Name, and its content that can be a) changed by
assignment expressions, and b) viewed or printed */

    Client = 1;
    PIN = 1234;     // Personal Identification number
    Balance = 2502;  // Account balance
    view();          // Function to view data in memory
    print(" Client No.", Client, "   PIN No:", PIN,
           " Current balance =", Balance, ");

```


21 Program code also has commands (to control execution sequence) and declarations (to specify data type).
Expression

An expression is a combination of constants, variables, operators, and functions. Evaluation of an expression results in a value.
Assignment expression

An assignment expression evaluates the value of the expression and then assign the value to a variable. It has the following format:

\[ \text{Name} = \text{Expression} ; \]

where

- \( \text{Name} \): any valid variable name. Notes: i) best to use a name that suggests a meaning; ii) only a valid \( \text{Name} \) can be in the left-hand-side of the assignment operator =

- \( \text{Expression} \): an algebraic expression that has a value

The \( \text{Name} \) actually identifies the physical memory-storage that contains the \( \text{Expression} \)-value. Think of the physical memory-storage as a box or a container (containing the \( \text{Expression} \)-value) with the \( \text{Name} \) as its label. The \( \text{Name} \) henceforth refers to its content, and the content may be changed if so wished.

1. Variable Names

A variable name must begin with a letter and may have up to 24 additional characters of the following set:

- Letters a to z, A to Z. Note: Upper and lower case letters are considered different (e.g. M and m are different)

- Digits 0 to 9

- The characters $ # _ are permissible but best to be avoided. Any other special character such as [ ] { } * ; " ( ) ’ % space period etc.. is NOT permitted in a variable or function name.

- A name must be completely typed on one line with no space or illegal characters within it.

- Reserved keywords cannot be used for user-defined names.
2. Operators

CMAP is just for numerical computations, and hence it needs only a small sub-set of C/C++ operators\(^\text{22}\).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Precedence</th>
<th>Expression</th>
<th>Same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Function call</td>
<td>1 Highest</td>
<td>A = \sin(30)^*3;</td>
<td>A = (\sin(30))^*3;</td>
</tr>
<tr>
<td>[ ]</td>
<td>Array subscripts</td>
<td></td>
<td>A = 2^*M[3,4]^2;</td>
<td>A = 2*(M[3,4])^2;</td>
</tr>
<tr>
<td>^</td>
<td>Raise to power</td>
<td>2</td>
<td>B = 2^*4^2/8;</td>
<td>B = 2*(4^2)/8;</td>
</tr>
<tr>
<td></td>
<td>Vector cross product</td>
<td></td>
<td>! V = V1^V2 * V3;</td>
<td>! V = (V1^V2) * V3;</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>3</td>
<td>C = 8*6/3;</td>
<td>C = (8*6)/3;</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td></td>
<td>C = 8/6*3;</td>
<td>C = (8/6)*3;</td>
</tr>
<tr>
<td>+</td>
<td>Addition, Subtraction</td>
<td>4</td>
<td>D = 5-1+2;</td>
<td>D = (5-1)+2;</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>Test for:</td>
<td>5</td>
<td>E = 4 + 2 &lt; 6;</td>
<td>E = (4 + 2) &lt; 6;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than</td>
<td></td>
<td>F = 4 + 2 &lt;= 6;</td>
<td>F = (4 + 2) &lt;= 6;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td></td>
<td>G = 4 + 2 &gt; 6;</td>
<td>G = (4 + 2) &gt; 6;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than OR equal to</td>
<td></td>
<td>H = 4 + 2 &gt;= 6;</td>
<td>H = (4 + 2) &gt;= 6;</td>
</tr>
<tr>
<td>==</td>
<td>Equality test</td>
<td>6</td>
<td>I = 4 + 2 == 6;</td>
<td>I = (4 + 2) == 6;</td>
</tr>
<tr>
<td>!=</td>
<td>Not-equal test</td>
<td></td>
<td>J = 4 + 2 != 6;</td>
<td>J = (4 + 2) != 6;</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>7</td>
<td>K = 4 + 1 &amp;&amp; 0;</td>
<td>K = (4+1) &amp;&amp; 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>8 Lowest</td>
<td>N = 2 == 2</td>
<td></td>
</tr>
</tbody>
</table>

3. Expression Evaluation

The evaluation of expressions follows the rules of operator precedence similar to the conventional rules of algebra:

The evaluation is done from left to right UNLESS the next operator has higher precedence than the current one--In such case, the operator of higher precedence is done first.

Operations within parentheses have the highest priority. Thus, in case of doubt, use parentheses to impose top priority.

The value of the following expression

\[
C = 12+3-(A=30-4)*2+4*5^2/8+12/2*5+(6+A)^2/8;
\]

\(\text{22 The complete list of C++ operators can be found here:}\)

is 13.5. Its sequence of evaluation steps is shown below. Note that
the evaluation is done from left to right as long as it does not
interfere with the precedence of arithmetic operators.

\[ C = 12 + 3 - (A = 30 - 4) * 2 + 4 * 5^2 / 8 + 12 / 2 * 5 + (6 + A) * 2 / 8; \]

The following shows the values of all sub-expressions nested in one
big assignment expression. The value of the entire expression is,
finally, \(-6240\).

\[ E = (A = (B = 30 - 4) * 2) * (C = 8 / 2 - 1) * (D = 12 - A); \]

4. Functions

**Built-in functions** are library functions that are already made and are
available for use at any time. They include functions for doing
mathematics, statistics, plotting, graphics, financial computations,
etc... They are the power tools that facilitate problem solution.

Info: CMAP's built-in functions are displayed in the vertical Tab-Window
labelled "Built-in". Double-clicking a function name will bring up
complete documentation on the syntax, options and usage of the function.

Example: Evaluate the following expression in the CALC-Window (Note: angles are specified in radians):

\[ C = (A=0.45)\exp(-A)/\sin(\pi#/6)^2 \]

The preceding expression is evaluated in the following order:

Step 1: Sub-assignment expression to define variable A.
Step 2: The value –0.45 is sent to function \( \exp() \) as its function argument.
Step 3: The exponential function \( \exp() \) "return" the value of \( e^{-0.45} \) for use in the expression. To "return" a value is to bring back the value.
Step 5: \( \pi# \) is the built-in pre-defined constant for \( \pi \). The value of \( \sin(\pi/6) \) is sent to function \( \sin() \) as function argument.
Step 6: The sine function returns (brings back) the value of \( \sin(\pi/6) \) for use in the expression. Since the parentheses following the function name bind strongly to the function name, step 6 precedes step 7.

5. Subexpressions

As seen in the preceding examples, a parenthesised subexpression can be used in the place of a constant or variable name.

Invoking Functions

The following concepts are common to built-in functions as well as user-defined functions [such as \texttt{main()}]:

\[ C = (A=0.45)\exp(-A)/\sin(\pi#/6)^2; \]
Invoking: To use a function, it must be invoked (or called) by name\textsuperscript{23}. When a function is invoked, execution-control is automatically transferred to the function code residing in the memory. Once the function execution is completed, execution-control is automatically transferred back to where it was.

Syntax: A function name must be followed by parentheses within which the required function arguments (if required) are listed and separated by commas.

\[ \text{FunctionName}(\ldots, \ldots, \ldots) \]

Arguments:

Function arguments are the data given to the function for it to process.

Ex. The arguments are underlined in the following example:

\[ B = \text{hypot}(3, 4) \times \text{sin}(A = 30\pi#/180); \]

Function’s Returned Value:

The returned value of \( \text{hypot}(3, 4) \) is \( \sqrt{3^2 + 4^2} = 5 \), and the returned value of \( \text{sin}(A = 30\pi#/180) \) is sine of 30° or 0.5. These returned values are used in expression evaluation \( 5 \times 0.5 = 2.5 \) which is then assigned to ‘\( B \)’.

Function’s Actions:

- To compute and return a value (e.g. common math functions)
- To interact with the user (e.g. print, view, catalog, plot, etc...). Here, the returned values may not be of primary interest, and hence usually not assigned to variables.

\textbf{Exception:} main() function is automatically invoked/executed when the program is executed--after a memory/data reset. Henceforth all current as well as newly added/edited functions are automatically available for use in the Calc Expressions Window, which allows incremental program development (without triggering a memory/data reset).
Practice Drills 1.2

Use the CALC-expressions Dialog to evaluate the values of the following expressions. Confirm the result by mental computation. Consult CMAP documentation for info on built-in functions.

1. Operator precedence

   - $3 + 4 \times 2$; // same as $3 + (4 \times 2)$;
   - $(3 + 4) \times 2$;
   - $3 \times 4 / 2$; // same as $(3 \times 4) / 2$;
   - $3 \times (4/2)$;
   - $(3\times4)/2$;
   - $3\times4^2$; // same as $3\times(4^2)$;
   - $(3\times4)^2$;
   - $2^2^3$; // same as $(2^2)^3$;
   - $2^{(2^3)}$;
   - $-3^2$; // same as $-(3^2)$
   - $(-3)^2$;
   - `cat()`; // Memory dump of variables

2. Press the Reset-button on the Calc-Expressions Dialog and then evaluate the following expressions in the order given.

   - `useoption("RADIANS"); // Rad for all trig. functions`
   - $A = 2\times\sin(\pi/#2)^2$; // $2\times1^2 == A$
   - $B = 2\times\sin((\pi/#2)^2)$; // $2\times0.624 == B$
   - $C = \sqrt{3^2+4^2}$; // hypot(3,4) == C
   - $D = \sqrt{1-\sin(A=60*\pi#/180)^2} - \cos(A)$;
   - $E = \log10(abs(10-110))$;
   - $F = G = H = 10$;
   - `cat()`;

3. Press the Reset-button on the Calc-Expressions Dialog and then evaluate the following expressions in the order given. Explain what is wrong, and fix the errors where necessary.

   - $A = 3 \times (4 + 2)$;
   - $C = (B = 2+3)/B^2$;
   - $E = \sqrt{A + c}$;
   - $F = \ln(AB)$;
   - $234 = G$;
   - $H = 0 = K$;
   - `cat()`;

4. Press the Reset-button on the Calc-Expressions Dialog and then evaluate the following expressions in the order given.

   - $A = 3 + 4 \times 2$;
   - $B = (C1 = 2^2^3)/(C2 = 2^(2^3))$;
   - $C = (D = 2 \times 4) / (2D/4)$;
D = (A = 1.5*(3+4))/(B = 3) + (A + 1) * B;
cat();

5. Run the following program and observe the execution.

main()
{
    hypot(3, 4);
cat();
    L = hypot(3, 4);
    view();
    getnum(" Enter an expression");
    view();
    V = getnum(" Enter an expression");
    U = sqrt(5*getnum(" Enter a value", 20));
    view();
    useoption("RADIANS");
    plot(x, 0, 2*pi#, exp(-0.15 * x) * sin(x));
}

6. Use the Calc-Expressions Dialog to evaluate the following expressions, in the given order.

    useoption("DEGREES"); // Degrees in trig functions
    sin(30); // 0.5 ==> ?
    A1 = sin(30)^2; // same as A1 = (sin(30))^2;
    A2 = (sin(30))^2;
    B = sin(30^2);
    C = D = sin(30)*cos(30);
    E = ((F = sin(30))/(G = cos(30)))/(F+G);

7. Every character and key (e.g. 'A', 'a', 'B', 'b', '0', '1', '9', '[]', '+') has an internal numerical value. The code for character 'A' is 65. Consider the following assignment expression:

    Val = 65;

Is Val a character or an integer? Ans: Val is an integer but it will be recognized as the character 'A' when its type is declared as character.

Run the following program and explain the output.

main()
{
    A = 'A'; // Use single quotes to designate character
    B = A + 1; // same as B = 65 + 1;
    B2 = 'B'; // 66 ==> B2
    blank = ' '; // 32: Blank character

---

24 See http://www.asciitable.com/
1.5 Types of Stored Information

- A wide variety of information can be stored in digital form (e.g. characters, integers, decimal values, images, music, maps, program codes, etc.).
- Digital information is stored as binary bits (1 and 0) which are grouped into bytes (8 bits).
- Each byte has an address (e.g. 0, 1, 2, . . . ) for locating the information.
- In CMAP, a decimal value is stored in 8 bytes. A variable-name is actually the address of the first byte where the content of the variable is stored.

Software can access stored data by keeping track of

i) the address of the first byte (i.e. the variable name), and  
ii) the number of bytes for storing the data.

For this scheme to work, the type of data (its length and storage format) need be specified within the program. A complete list of standard basic data types in C/C++ can be found here:


CMAP deals only with decimal values, each stored in 8 bytes. CMAP maintains a table of variable names and associated (first-byte) addresses of their values.

1. Data Types in CMAP

CMAP has 2 data types essential for engineering computations:

- **float** to specify a simple variable having a decimal value.
- **mat** to specify a variable that is a matrix, vector, or complex number. Each is a group of decimal values.25

25 CMAP's internal table associates a matrix-variable name with the address of the first byte of an 1-D array that contains the addresses of the first bytes of the first elements of the rows of the matrix. The figure below illustrates the idea for a matrix M of 4 rows,
While every variable in a C/C++ program must have its type *declared*, CMAP-programs take undeclared variables to be float-type by default as seen throughout this chapter.

2. Internal Representation of float

Binary representation and storage

The memory storage reserved for a CMAP float variable is 8 bytes or 64 bits. A float value can be represented in binary form\(^{26}\):

\[ \pm 1.M \times 2^E \]

where

- \( E \) is the binary signed integer for the exponent, stored in 11 bits.
- \( 1.M \) is the binary mantissa with the fractional part \( M \) stored in 52 bits.
- The sign is stored in 1 bit (0 for plus, and 1 for minus).

Note that the constants 1 and 2 of the binary representation need not be stored--only the parts \( M \) and \( E \) are stored.

Range of values and accuracy of float type\(^{27}\)

\(^{26}\) A number such as -12.34 is first converted into power of 2 format:

-12.34 = -1.5425 x 2\(^3\)

and next, the "mantissa" 1.5425 and the "exponent" 3 are converted and stored in binary format.

\(^{27}\) Why binary representation may not be accurate:
https://www.youtube.com/watch?v=isWMPswI4t4

Glad that we have no need to get deeper into this:
https://www.youtube.com/watch?v=gIkvVHvnkDE
64-bit storage is sufficient for 16 digits of a decimal number. The range of stored values is from $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{+308}$.

Since the internal binary representation of decimal numbers (e.g. 0.1, 0.7) may not be exact\(^{28}\), the result of computation will not be exact. For example, we expect the value of the following expression to be zero:

\[
0.02 - 0.01 - 0.03 + 0.02
\]

but the computed result is only nearly zero: $3.46945 \times 10^{-18}$.

Similarly, the result of $8 - (0.1+0.7) \times 10$ is $8.88178e-016$ instead of 0.

### Rounding values

The result can be rounded to 15 decimal places by using the built-in function `round()` as follows:

```c
round(15, 0.02 - 0.01 - 0.03 + 0.02);
```

which gives 0 as expected.

### Practice Drills 1.3

1. Debug the following program (it contains both syntax and logical errors) so that when it works as expected, it produces this output:

   
   
   
   17.5  19.625  37.125
   

   ```c
   /* This program contains many errors that need to be fixed. */
   
   main( )
   {
       B = A - 1.5;
       A = 5
       AreaRectangle = A B;
       // Radius R = 2.5
   }
   
   28 Similarly, fractions such as 1/3, 1/6, 1/7, 1/9 etc... cannot be exactly represented in the decimal base.

\[\text{© 2015 K. H. Ha - BCEE 231 Chapter 1 - v4.3} 1.21\]
2. The following program requests input of two values and prints their sum. The function \texttt{getnum()} gets the user's input. It presents a dialog that allows you to execute expressions, and it returns the last computed value (on closing the dialog by pressing the OK-button).

```c
main()
{
    A = \texttt{getnum}(" Enter first value", 3);
    B = \texttt{getnum}(" Enter second value", -4);
    C = A + B;
    \texttt{print}(" The sum is ", C);
}
```

- Rewrite the program so that it prints the sum of the absolute values of both \(A\) and \(B\).
- Rewrite the program so that it prints the absolute value of the sum of \(A\) and \(B\).
- Rewrite the program so that it prints the value returned by \texttt{hypot}(A, B).
- Rewrite the program so that it does NOT use any variable name.

3. Evaluate each of the following expressions in the CALC-Window and explain what went wrong and fix it:

- (a) \( A = \texttt{hypot}(3, 4, 5); \)
- (b) \( B = \texttt{hypot}(5, 12); \)
- (c) \( C = \texttt{sqrt}(-3.4); \)
- (d) \( D = \texttt{ln}((-2.45)^2); \)
- (e) \( E = (-2)^2.01; \)
- (f) \( F = \texttt{sqrt}(2-\texttt{sqrt}(2)^2); \)
- (g) \( 2*\text{pi} = A; \)
- (h) \( \text{AVeryLongLongLongLongLongName} = 10; \)
4. For each of the following programs:

- What will be the printed output? Note: You may need to consult CMAP documentation on the reserved words (blue colour).

- After program execution, explain what's listed in the "Globals" Tab-Window.

- Introduce 5 syntax and/or logical errors, and run the program to see the first error message. Correct the error and run the program again to see the next error message. Repeat the previous step until there is no error left.

(a)

```plaintext
main() {
    Radius = 2;
    Area = pi#*Radius^2;
    Length = 2*pi#*Radius;
    print(" Radius = ", Radius, " Area = ", Area,
          " Length = ", Length);
    cat();
}
```

(b)

```plaintext
main() {
    a = 4.5;
    B = 30;
    C = 45;
    A = 180-(B+C);
    useoption("DEGREES");
    b = sin(B)*(aoA = a/sin(A));
    c = sin(C)*aoA;
    Area = a*(h=a*cos(B))/2;
    Length = a+b+c;
    cat();
}
```
1.6 The Power-Tools: Built-in Functions

The following built-in functions have been introduced in this Chapter. Refer to CMAP-documentation for complete information on the purpose, syntax and usage of built-in functions.

It is convenient to use the Calc-Expression Dialog to test the usage of these functions.

- `abs(x)` Returns the absolute value of $x$.
- `atan(x)` Returns the angle whose tangent is $x$.
- `cat()` Displays the catalogue of all current variables (0 is returned).
- `cos(x)` Returns the cosine of $x$.
- `exp(x)` Returns the value of $e^x$.
- `getnum( . . . )` Receives input and returns the value of the last executed expression.
- `hypot(x,y)` Returns the value of $\sqrt{x^2 + y^2}$.
- `ln(x)` Returns the natural log (base $e$) of $x$.
- `log10(x)` Returns the common log (base 10) of $x$.
- `print( . . . )` Prints values and text on screen (0 is returned).
- `round(N,x)` Returns the value of $x$ rounded to $N$ decimal places.
- `saveglobalvars()` Saves all global variables into a file
- `sin(x)` Returns the sine of $x$.
- `sqrt(x)` Returns the square root of $x$.
- `tan(x)` Returns the tangent of $x$.
- `useoption( . . . )` Sets option such as RADIANS or DEGREES (only for built-in trigonometric functions).
- `view( . . . )` Receives input from user, and views/edits variables' contents.
Chapter 2

Program Design and Control Structures

A failure is a man who has blundered but is not able to cash in on the experience -- *Elbert Hubbard*

Contents

Chapter 2  Program Design and Control Structures  1
2.1 Programs and Programming  1
   1. The Algorithm  2
   2. Language Vocabulary and Built-in Capabilities  4
   3. Program Coding  5
   3. Program Debugging  5
2.2 Problem Solving  6
   1. Problem Statement  6
   2. Solution Formulation  6
   3. The Algorithm  7
   4. Implementation  7
   5. Program Testing and Verification  8
2.3 The *if/else* Control Structure for Conditional Actions  9
   Sample Program 2.1 - Test Values before Using Them  10
   Sample Program 2.2 - Multiple Branch of Decision  11
   Practice Drills 2.1  12
2.4 Logical Expressions  13
   1. Simple Logical Expressions  13
   Practice Drills 2.2  14
   2. Compound Logical Expressions  15
   Practice Drills 2.3  15
   3. Mixing Logical Expressions in Arithmetic  18
2.5 Looping: what and why?  20
2.6 The *while* Loop Control Structure  21
   Sample Program 2.3 - Validation of User Input  22
   Sample Program 2.4 - Summation of a Generated Series of Values  22
   Sample Program 2.5 - Iterative Process for Root Finding  23
   Sample Program 2.6 - The Machine Epsilon  25
   Sample Program 2.7 - Finding Limits of Math Functions  26
   Sample Program 2.8 - Factorial of Integers  26
   Sample Program 2.9 - Loss of Significance Digits in Subtraction  27
   Sample Program 2.10 - Summation of Infinite Series  28
   Sample Program 2.11 - Schedule of Loan Payments  29
   Practice Drills 2.4  30
   Exercise 2.1  34
2.7 The *do/while* Loop Control Structure  36
   Practice Drills 2.5  37
2.8 The *for* Loop Control Structure  37
   Practice Drills 2.6  39
2.9 Nested Control Structures  39
   Exercise 2.2  41
2.10 The keyword *break*  42
2.11 The keyword *continue*  43
2.12 The *switch* Structure for Multiple-Branch Selection  44
   Practice Drills 2.7  46
2.13 Useful Built-in Functions  47

© 2015 by K. Ha - BCEE 231 Chapter 2 - v4.3
The programs in Chapter 1 execute left to right, top down, and contain only expressions. The capability of such programs is extremely limited.

The program that controls the ATM banking business must have further capabilities, including the following:

1) To access and update clients' database. This is done using built-in functions (see CMAP-Help topic data file overview).

2) To determine if client's PIN number is correct, and if client's input of dollar amount (e.g. deposit, payment, withdrawal) is valid. This is done with if-else statements.

3) To process any number of clients and transactions per client. This can be done using loop-control structures.

This Chapter introduces standard language features that deal with the items 2, 3 above.

Expressions (Chapter 1) plus if-else statements and loop control mechanism (Chapter 2) constitute the absolute essentials of any programming language.

2.1 Programs and Programming

Programming is the process of organizing, planning, writing, testing, and revising the sequence of commands/instructions/tasks so that the program works as expected.

Computer programming makes use of a computer language to issue commands/instructions/tasks.

1. The Algorithm

The algorithm is the step-by-step procedure that accomplishes the tasks. The steps must be within the rules and power of the available capabilities\(^1\).

---

\(^1\) These capabilities are derived from the defined abilities of (i) Expressions, (ii) key-words, (iii) built-in functions, and user-defined functions which are built with (i, ii, iii)--as described in Chapter 3.
An Example

The task: pick out the biggest apple among many.

The rules: you can only

i) use a hand (left or right) to pick up or put down an apple
ii) feel the size of the apples in the two hands.

The algorithm:

a) pick up one apple with the left hand
b) pick up another apple with the right hand
c) feel the size of the apples in the hands, and put down the one that is smaller
d)  - If there is any apple left, use the free hand to pick up another apple, and then go back to step (c)
    - or else, the biggest apple is now in the hand.

To solve problems means to come up with a feasible strategy--one that falls within the rules².

A Coding Example

The task: input 3 numbers and print the largest one.

The standard rules of programming: We may use only variable names, operators, expressions, key-words and functions, together with delimiters such as , { } ( ) " ; as required by the syntax (format). Note: key-words and names of built-in functions cannot be used for variable names.

The algorithm:

a) Input 3 numbers and store them in A, B, C
b) Assign A to Max
c) Compare B and Max, and if B is larger then assign B to Max
d) Compare C and Max, and if C is larger then assign C to Max
e) Print Max, which now stores the largest number.

² Strategies that use circular reasoning don't work:
How to go to the moon:
a) Get to within 1 m of the moon surface
b) Jump onto the moon surface

Or how to live to the age of 90:
a) Live to the age of 89 plus 364 days
b) Live for one more day
An implementation in CMAP:

```cpp
main()
{
// Purpose: Input 3 values and print the largest one
  // view(A, B, C);
  A = getnum(" First value");
  B = getnum(" Second value");
  C = getnum(" Third value");
  print(^, " Values: ", A, B, C);
  Max = A;  // Assign A to Max
  if(B > Max) { Max = B; }
  if(C > Max) { Max = C; }
  print(^, " Largest value = ", Max);
}
```

*Food for thought:* what is the one key idea\(^3\) in the above program?

### 2. Language Vocabulary and Built-in Capabilities

Evaluation of *expressions* is a crucial ability as seen in Chapter 1, where the sequence of execution is primarily left to right, top down.

This chapter presents commands or key-words (i.e. language vocabulary) which can change the sequence of execution.

The vocabulary of standard C++ language has 84 key-words. The full list is here: [http://en.cppreference.com/w/cpp/keyword](http://en.cppreference.com/w/cpp/keyword)

Since CMAP is primarily for numerical computations, it uses only the following 14 key-words:

```plaintext
break, case, continue, default, do, else, end, float, for, if, mat, return, switch, while.
```

\(^3\) The key idea in this algorithm is to inspect each and every number and save the greater number in a variable name, say Max. It is the basis of the strategy, and is independent of programming languages.
In addition to keywords, programs also make use of built-in functions [e.g. print(), view(), cat(), cos(), sqrt(), ... etc] which provide additional capabilities. While standard C-library for maths has about 22 functions⁴, CMAP's comprehensive math library has over 360. We should be aware of the available capabilities but need to know well only what we use.

3. Program Coding

The rules in program coding are the rules associated with the use of

- Expressions (Chapter 1)
- Key-words (Chapter 2)
- Built-in functions (as documented in CMAP Help)

The above basic elements can be used to construct more complex entities in the form of user-defined functions (Chapter 3) and additional math operations (Chapter 4).

In real world programming, programmers also need a Developer Library which is an extension of the language into the specific areas of application and to facilitate user-interface design⁵. The Developer Library (also called Application Programming Interface) enables programmer to add bells and whistles that make programs user-friendly, powerful, and even intelligent.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Language</th>
<th>Application</th>
<th>Developer Library Application Programming Interface (API)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>Java</td>
<td>Mobile</td>
<td><a href="https://developers.google.com/android/">https://developers.google.com/android/</a></td>
</tr>
<tr>
<td>CMAP</td>
<td>C</td>
<td>Mathematics</td>
<td>CMAP built-in functions</td>
</tr>
</tbody>
</table>

3. Program Debugging

Debugging is the process of detection and correction of errors. CMAP provides some tools to facilitate debugging (e.g. run-time data/program view, function tracing, data dump), but the best tool

⁴ [https://www-s.acm.illinois.edu/webmonkeys/book/c_guide/2.7.html](https://www-s.acm.illinois.edu/webmonkeys/book/c_guide/2.7.html)
⁵ Human-machine communication: windows, menus, icons, mouse, pen, touch, etc..
is still the ability to observe, test hypotheses & scenarios, and to reason.

Thus, programming can be fun (i.e. game playing both as a strategist & a detective) and useful (i.e. problem solving). A well-written program has the beauty and elegance of a piece of art work.

<table>
<thead>
<tr>
<th>A program</th>
<th>VS</th>
<th>An art work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer</td>
<td>Artist</td>
<td></td>
</tr>
<tr>
<td>Language elements</td>
<td>Kaolin clay</td>
<td></td>
</tr>
<tr>
<td>Coding skills</td>
<td>Artist skills</td>
<td></td>
</tr>
<tr>
<td>Other tools:</td>
<td>Other tools:</td>
<td></td>
</tr>
<tr>
<td>- Developer environment,</td>
<td>- Paint</td>
<td></td>
</tr>
<tr>
<td>- Built-in functions</td>
<td>- Brushes</td>
<td></td>
</tr>
<tr>
<td>- API</td>
<td>- Knives, chisels</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Other paraphernalia</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Problem Solving

To develop a program for solving engineering problems requires the following phases:

- Problem statement: Defining the problem and objective
- Problem formulation: The approach for solution
- The algorithm
- Implementation
- Program testing, verification and interpretation of output

1. Problem Statement

This describes the data, objective and requirements of the problem.

**Sample Problem Statement:** Develop a program that determines the real roots of the quadratic equation \( ax^2 + bx + c = 0 \) where the coefficients \( a, b, \) and \( c \) are to be input at runtime.

2. Solution Formulation

This presents the approach for solution. It may involve the use of mathematics, engineering sciences, physics, and computer algorithms, etc.
The formulation:

- If both $a$ and $b$ are zero: There is no root for $x$.
- If $a$ is 0: The root is $x = -\frac{c}{b}$ provided that $b \neq 0$
- Now, $a \neq 0$:

  If $D \equiv (b^2 - 4ac)$ is negative we have only imaginary roots which are not of interest in the present problem;

else, the real roots are

$$x = \frac{1}{2a} \left( -b \pm \sqrt{b^2 - 4ac} \right)$$

3. The Algorithm

1) Input values of $a$, $b$, $c$
2) If $a$ is zero:
   - if $b$ is also zero: print "no solution"
   - or else print the root $x = -c/b$
3) or else (here $a$ is not zero):
   - Compute $D$
   - If $D$ is negative: Print "No real roots"
   - else compute and print the 2 roots

4. Implementation

```c
main() {
    // Real roots of quadratic equations
    view(a, b, c); // Get user input
    print(" a =", a, " b =", b, " c =", c);
    if(a==0) {
        // Here a is zero
        if(b==0) {
            // b is zero
            print(" No solution");
        }
    else
        { // b is not zero, but a is
            x = -c/b;
    
```
```csharp
print(" One root only: ", x);
}
}
else
{
    // a is not zero
    D = b*b-4*a*c;
    if(D < 0)
    {
        // D is negative
        print(" Imaginary roots only");
    }
    else
    {
        // D is not negative: 2 real roots
        x1 = (-b+sqrt(D))/(2*a);
        x2 = (-b-sqrt(D))/(2*a);
        print(" Two roots: ", x1, x2);
    }
}
```

5. Program Testing and Verification

The process of testing is called program debugging. It involves the detection and correction of errors--This will be easier when the structure of the program is made clear with proper indentation of the code segment that forms a block.

There are three types of errors:

Syntax errors

These are errors of typing, grammar or format. CMAP will attempt to identify the nature and location of the errors. Syntax errors must be corrected before the program can be completely executed.

Execution errors

These are errors that occur during computation; e.g. division by zero, or taking logarithm of a negative value. Execution errors will
either interrupt program execution or give results such as NAN (i.e. Not a number), #IND (Indeterminate), #INF (Infinite).

Logical errors

These are errors of logic. While the program may run, the results are erroneous. It’s up to us to detect and correct logical errors.

CMAP’s debugging facilities

Refer to CMAP Help topic: “How do I debug my program?”

We now look at the rules (syntax and actions) of the if-else key words.

2.3 The if/else Control Structure for Conditional Actions

Format and syntax (first part of the rules):

```java
if(Expr) {
    .... // TRUE-block (Expr is non-zero)
}
// Ending brace

// The following else-statement is optional
else {
    ... // ELSE-block (Expr is zero)
}
// Ending brace
...

// OTHER-statement
```

Note that the pair of braces { . . .  } define the body of the TRUE- and ELSE-blocks.

Action (the other part of the rules):

- If the value of `Expr` is non-zero (i.e. TRUE), the statements in the TRUE-block will be executed,

- or else the statements in the ELSE-block will be executed.

Either one of the two blocks may be executed, but never both. The else-clause is optional.
Sample Program 2.1 - Test Values before Using Them

Let $a$, $b$, $c$ be the lengths of the three sides of a triangle. If the sides form a closed triangle, its area can be computed as:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{where} \quad s = (a+b+c)/2$$

The following program gets input data for the three sides, computes the triangle area after verifying that the quantity under the square root sign is not negative.

Observe the usage and the syntax of the `if/else`-statements:

```c
main()
{ /* Input the lengths of 3 sides of a triangle, and compute & print the area of the triangle if the triangle is closed */
    view(a, b, c);
    print(" Lengths: ", a, b, c);
    s = (a+b+c)/2;
    T = s*(s-a)*(s-b)*(s-c);
    if(T > 0) { // Closed triangle
        Area = sqrt(T);
        print(" Area = ", Area);
    }
    else
    {
        notice(" Not a triangle!");
    }
    print(" Finish!");
}
```

The `else`-clause (following `if`) is needed only when dictated by logic.

Observe that the code stays strictly within the rules with proper syntax and use of delimiters.
Food for thought: If we remove the comments and blank space from the above program, and put it in one single line\(^6\), it still works exactly the same way. In fact, we can still read and comprehend it properly, although not as easily. How is that possible?

```c
main(){view(a,b,c);print(^,"Lengths:",a,b,c);s=(a+b+c)/2;T=s*(s-a)*(s-b)*(s-c);if(T>0){Area=sqrt(T);print(^,"Area=",Area);}else{notice("Not a triangle!");}print(^,"Finish!");}
```

Sample Program 2.2 - Multiple Branch of Decision

The following program gets input for "Score" and then prints the Grade per the following table:

<table>
<thead>
<tr>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>80-100</td>
<td>A</td>
</tr>
<tr>
<td>70-79</td>
<td>B</td>
</tr>
<tr>
<td>60-69</td>
<td>C</td>
</tr>
<tr>
<td>50-59</td>
<td>D</td>
</tr>
<tr>
<td>0-49</td>
<td>F</td>
</tr>
</tbody>
</table>

```c
main()
{
Score = getnum(" Enter your Score");
print(^, " Score:", Score);
if (Score >= 80)
{
    print(" Grade A");
}
else
{
    if(Score >=70)
    {
        print(" Grade B");
    }
    else
    {
        if(Score >= 60)
        {
            print(" Grade C");
        }
    }
}```

\(^6\) Maximum of 255 characters on a line
else
{
    if(Score >= 50)
    {
        print(" Grade D");
    }
    else
    {
        print(" Grade F");
    }
}
}

Run the above Sample Program with different data to test its working.

Practice Drills 2.1

1. Write a program to convert a temperature input from Fahrenheit to Celsius or vice-versa per user's wish.

Formulas: °F = \frac{9}{5} °C + 32 \quad °C = \frac{5}{9} (°F - 32)

Notes: Use \( T \) to store the temperature input, and \( \text{FtoC} \) to store the user's wish: \( \text{FtoC} = \text{yesno("Convert Fahrenheit to Celsius?")}; \)

2. Change only one item in a working program, and challenge your friend to fix it should something is wrong.

3. A task can often be coded in different ways. Take a working program and challenge yourself or your friend to use a different approach to achieve the same objectives. Investigate the merits of the approaches in terms of simplicity, clarity, execution speed, and robustness\(^7\).

4. Adapt the Sample Program 1.2 for \( \alpha = 33.5^\circ \quad d = 0.084 \text{ m} \), and furthermore:

   a) Compute the maximum stress (at \( A \)) given by:

   \[
   \sigma_{\text{max}} = \frac{N}{A_a} + \frac{M_a b}{2 I_a}
   \]
   where \( b = 0.1 \text{ m}, \quad t = 0.05 \text{ m}, \quad A_a = bt, \quad I_a = \frac{tb^3}{12} \)

---

\(^7\) The protection against potential program crash or errors caused by bad data input or assumptions.
b) Verify if the computed maximum stress is greater than the allowable stress of 40.2 kPa and issue an appropriate warning, or else print the safety factor: \[ FS = \frac{\sigma_{\text{allow}}}{\sigma_{\text{max}}} \]

c) Find the maximum force \( P \) that can be applied without exceeding the allowable stress.

Ans: \( FS = 2.5 \) \( P_{\text{max}} = 80 \text{ N} \times FS \)

### 2.4 Logical Expressions

#### 1. Simple Logical Expressions

A logical expression may have either the value 1 (for TRUE) or 0 (for FALSE). Comparison of two values can be done with logical expressions using simple logical operators:

<table>
<thead>
<tr>
<th>Simple Logical Operators</th>
<th>Test for</th>
<th>Example Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>( 4 + 2 &lt; 6; ) // 0 or FALSE ( 2 + 3 &lt; 6; ) // 1 or TRUE</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Less than OR equal to</td>
<td>( 4 + 2 \leq 6; ) // 1 ( 3 + 4 \leq 6; ) // 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>( 4 + 2 &gt; 6; ) // 0 ( 3 + 4 &gt; 6; ) // 1</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than OR equal to</td>
<td>( 4 + 2 \geq 6; ) // 1 ( 2 + 3 \geq 6; ) // 0</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
<td>( 4 + 2 = 6; ) // 1 ( 4 + 2 + 1e-15 = 6; ) // 0 ( 4 + 2 + 1e-16 = 6; ) // 1</td>
</tr>
<tr>
<td>!=</td>
<td>Not-equal to</td>
<td>( 4 + 2 \neq 6; ) // 0 ( 4 + 2 + 1e-15 \neq 6; ) // 1 ( 4 + 2 + 1e-16 \neq 6; ) // 0</td>
</tr>
</tbody>
</table>

Logical expressions are useful in if-statements as seen in examples of preceding sections. They have other usage as seen later.
Practice Drills 2.2

1. Execute each of the following simple "logical expressions" in the CALC-Expressions Window and explain their values--or if you wish, make one program containing all expressions.

   E1 = (10 == 10);
   E2 = (10 != 10);
   E3 = (10 >= 10);
   E4 = (10 <= 10);
   E5 = (10 > 10);
   E6 = (10 < 10);
   E7 = (8 == (0.1+0.7)*10);
   E8 = (10 == 10.00000000000001);
   E9 = (10 == 10.0000000000000001);
   E10 = (abs(10 - 10.00000000000001) < 1e-12);
   /* Note that E10 provides a safe way to test the equality of two decimal values */
   cat();

2. Since every character (e.g. 'A', 'a', 'B', 'b', '0', '1', '9', '[', '+') has an internal numerical value, characters can be compared by logical expressions. Use CALC-Expressions to determine the values of the following logical expressions:

   'A' > 'B'
   'A' + 1 == 'B'
   ' ' == 32
   'A' + ' ' == 97

This is the basis for dictionary searching, sorting.
2. Compound Logical Expressions

Compound logical expressions test for TRUE or FALSE of multiple conditions. This is accomplished by combining simple logical expressions with the operators && (for AND) and || (for OR).

For example, to test if A is greater than B AND if A is less than C, we write:

```java
if ((A > B) && (A < C)) { ...}
```

Since the operators &&, || have lower precedence than the other logical operators, the inner parentheses in the preceding expression may be omitted:

```java
if (A > B && A < C) { ...}
```

- The AND-compound expression (E1 && E2) is true (value 1) when both expressions E1 and E2 are true.
- The OR-compound expression (E1 || E2) is true (value 1) when either expression E1 or E2 is true (i.e. non-zero).

The following two tables show these results:

| E1 && E2 | E2 | E1 || E2 | E2 |
|---------|----|--------|----|
| E1 = 0  | 0  | 0      | 0  |
| E1 = 1  | 0  | 1      | 1  |
| E1 = 0  | 0  | 0      | 0  |
| E1 = 1  | 0  | 1      | 1  |

Practice Drills 2.3

1. Execute each of the following expressions in the CALC-Expressions Window and observe the output:

```java
E1 = ((a=2+4) < a || a > 0);
E2 = ((a=2+4) != a && a > 0);
```

---

8 This also includes the case when both expressions E1 and E2 are true (non-zero).
2. Run each of the following programs and explain the output.

(a)

```plaintext
main()
{
    A = 2; B = 3; C = 4;
    if( 2*A == C) { print("T"); }
    else { print("F"); }
    cat();
}
```

(b)

```plaintext
main()
{
    A = 2; B = 3; C = 4;
    if( 2*A - C) { print("T"); }
    else { print("F"); }
    cat();
}
```

(c)

```plaintext
main()
{
    A = 2; B = 3; C = 4;
    if( A < B || C < B) { print("T"); }
```
else { print("F"); }
cat();
}

(d)

main()
{
    A = 2; B = 3; C = 4;
    if( 2*A == C && A+B < C) { print("T"); }
    else { print("F"); }
    cat();
}

(e)

main()
{
    if(( A=1) == (B=0 == 0) ) { print("T"); }
    else { print("F"); }
    cat();
}

3. Write a program to find the value of variable \( A \) for each of the following cases. Test the program with assumed values for \( B, C, D \) as required.

   a) \( A = B \) if \( B \geq 0 \) or else \( A = -1/B \).
   b) \( A = B*C \) if \( B*C > 0 \) or else \( A = \sqrt{-B*C} \)
   c) \( A = B \) if \( (B \geq 0 \) and \( C < 0), \) or else \( A = C \)
   d) \( A = B \) if \( (B \geq 0 \) or \( C < 0), \) or else \( A = C \)
   e) \( A = B \) if \( [(B \) is 0 and \( C < -2) \) or \( D < 10], \) or else \( A = C \)

4. Modify the Sample Program 2.2 to remove all else-statements without changing the output.
3. Mixing Logical Expressions in Arithmetic

Logical expressions can be freely mixed with regular arithmetic.

Example (a)

We now write the program to plot the piece-wise continuous function $y(x)$ shown in the following figure.

Note that different x-origins have been used to facilitate writing of the equations of the curves. For plotting the entire function, we need to express $y$ in terms of the common x-origin by the following substitution:

$$x_1 = x - 3$$
$$x_2 = x - 7$$
$$x_3 = x - 12$$

Thus, $y(x)$ can be expressed in piece-wise fashion as follows:

$$y(x) = \begin{cases} 
2x/3 & 0 \leq x \leq 3 \\
-(x-3)/2+3 & 3 < x \leq 7 \\
-0.6(x-7)^2 + 3.2(x-7) - 2 & 7 < x \leq 12 \\
(x-12)/2-1 & 12 < x \leq 14 
\end{cases}$$

Using compound logical expressions, the above 4 conditional expressions can be combined into one:

$$y = \begin{cases} 
(2*x/3) & (x \geq 0 \&\& x \leq 3) \\
-(x-3)/2+3 & (x > 3 \&\& x \leq 7) \\
(-0.6*(x-7)^2 + 3.2*(x-7) - 2) & (x > 7 \&\& x \leq 12) \\
((x-12)/2-1) & (x > 12 \&\& x \leq 14) 
\end{cases}$$

The plot can now be effected by the plot() function.
main()
{
    clearplot();
    setup(M,0);    // Set same scale in x, y
    iter# = 90*3;  // Number of plotting points = 90
    plot(x,0,14,
         y = (2*x/3)   * (x >= 0 && x <= 3)
         + (-(x-3)/2+3) * (x >= 3 && x <= 7)
         + (-0.6*(x-7)^2+3.2*(x-7)-2) *(x>=7 && x<=12)
         + ((x-12)/2-1) * (x >= 12 && x <= 14));
}

The following program plots separately the individual segments, and hence the plot is more accurate at the ends of the segments.

main()
{
    clearplot();
    setup(M,0);            // Set same scale in x, y
    setrange(0,-2, 14, 3); // Data range for the entire plot
    plot(x,0,3, 2*x/3 );
    plot(x,3,7, -(x-3)/2+3 );
    plot(x,7,12, -0.6*(x-7)^2+3.2*(x-7)-2  );
    plot(x,12,14, (x-12)/2-1  );
}

Example (b)

Consider the surface $f(x,y)$ defined by

$$f(x,y) = \begin{cases}
  x + y & x \geq 0, y \geq 0 \\
  x + y^2 & x \geq 0, y < 0 \\
  x^2 + y & x < 0, y \geq 0 \\
  x^2 + y^2 & x < 0, y < 0
\end{cases}$$

Using compound logical expressions, we may express $f()$ as

$$f = (x+y) * (x \geq 0 \&\& y \geq 0)$$
$$+ (x+y^2) * (x \geq 0 \&\& y < 0)$$
$$+ (x+y^2) * (x < 0 \&\& y \geq 0)$$
$$+ (x+y^2) * (x < 0 \&\& y < 0)$$
The 3-D surface of \( f(x,y) \) in the domain \(-2 \leq x, y \leq 2\) can then be plotted by:

```plaintext
sp_surface(x, -2, 2, y, -2, 2,
            f =    (x+y)   * (x >= 0 && y >= 0)
               + (x+y^2) * (x >= 0 && y <  0)
               + (x+y^2) * (x  < 0 && y >= 0)
               + (x+y^2) * (x  < 0 && y < 0) );
```

Consult CMAP-Help on \texttt{sp_surface()} for keys that rotate and move the surface on the screen.

### 2.5 Looping: what and why?

Looping: To repeat the same tasks over and over again--without duplicating the code.

Loops are needed for:

- Finding a phrase in billions of web pages or a word in an electronic dictionary
- Processing billions of transactions in banking, e-commerce
- Solving thousands of unknowns in equations, etc...
- Displaying map sections when dragged by the user or as the car moves

Lesser tasks also need loops:

- Keep asking for user's input until the input value is acceptable
- Sum a series of many terms as needed
- Sum a number of terms in a series until the accuracy is good enough
- Sum all values in a large table
- Repeat the same solution process with a new set of data input
- Repeat the same solution process using a trial solution (trial and error method) until the solution is good enough (random walk process)
- Repeat the same solution process by a systematic search until the solution is good enough (exhaustive search process)
- Repeat the same solution process by improving the previous solution again and again until the solution is good enough (iterative process)
C/C++ language has three ways to implement looping in a program: *while* loop, *do-while* loop and the *for* loop.

### 2.6 The *while* Loop Control Structure

**Format/syntax (the first part of the rules):**

```
while(Expr) // Condition tested here
{
    // Loop body begins (Expr is non-zero)
    . . .
} // loop body ends
. . . // First statement following the loop
```

**Action (the other part of the rules)**

Its action is to execute the loop body *repeatedly as long as the value of Expr is non-zero*\(^9\). Note that the condition is tested before executing the loop body. Furthermore, "Expr" may be any expression including logical expressions\(^{10}\).

---

\(^9\) A float value is non-zero even if it is very nearly zero.

\(^{10}\) If two float values are tested for equality, the test will fail unless the two values are identical to the last digit. The functions `round()`, `floor()`, `ceil()` are often used for truncating digits in float-values prior to comparison.
Sample Program 2.3 - Validation of User Input

Write a program to get user input of a non-zero value. Give the user 3 chances to enter a valid input.

```c
main()
{
    k = 0; // Counter
    A = 0; // Store input
    print("           k           A");
    while(A == 0 && (k = k+1) <= 3)
    {
        A = getnum(" Enter a non-zero value ");
        print(A, k, A);
    }
    print(" Last A = ", A);
    if(A == 0) { notice(" Input is zero! "); }
}
```

Food for thought: is it possible to replace the while-statement by:

```c
while(A == 0)
```

and modify the loop body accordingly?

Sample Program 2.4 - Summation of a Generated Series of Values

Write a program to find the sum

\[ \sum_{k}^{121.5} k = -1.5 - 1 - 0.5 + 0 + 0.5 + 1 + 1.5 + 2 + \ldots + 121.5 \]

Observations:

- The very first term is \( k = -1.5 \). The next term in the series is obtained by adding 0.5 to the preceding term: \( k = k+0.5 \), and so on. This suggests a loop within which 0.5 is added to the previous term.
- We use the variable Sum to store the accumulated sum of the generated k.
- Stop looping when the last term has been added.

```plaintext
main()
{
    Sum = 0;  // Will be the total sum
    k = -1.5;  // Initial k (first term)
    Max = 121.51;  // Last term
    while( k <= Max) {
        Sum = Sum + k;  // Accumulate k into Sum
        print(\^, k, Sum);
        k = k + 0.5;  // Next term
    }
    print(^^" Final sum = ", Sum);
}
```

*Food for thought:* what are the key-points in the above program?

**Sample Program 2.5 - Iterative Process for Root Finding**

Calculators and computers use an algorithm to compute the square root of a given value \( V \). One simple way is to start with an estimate of the root, say

\[
    r_1 = \frac{V}{2}
\]

and then a better root can be found by averaging the two terms:

\[
    r_2 = \frac{1}{2} \left( r_1 + \frac{V}{r_1} \right)
\]

The preceding "iterative" formula is applied again using the previous \( r_2 \) for the estimate \( r_1 \). The process stops when there is little improvement.
The following algorithm\textsuperscript{11} uses the above iterative formula to polish the root repeatedly until the last two estimates differ less than 1% of the computed root.

\begin{center}
\begin{tikzpicture}
  \node (r1) {r_1 = V/2};
  \node (r2) [below right of=r1] {r_2 = (r_1 + V/r_1)/2};
  \node (error) [below of=r2] {Error = |(r_2 - r_1) / r_1|};
  \node (r1again) [below of=error] {r_1 = r_2};
  \node (yes) [right of=error] {Yes};
  \node (no) [below of=yes] {No};
  \node (r2isroot) [below of=no] {r_2 is the root};

  \path[->]
  (r1) edge (r2)
  (r2) edge (error)
  (error) edge (r1again)
  (yes) edge (error)
  (no) -| (error)
  (error) -- (r2isroot);
\end{tikzpicture}
\end{center}

main()
{
  V = abs(getnum(" Enter a value"));
  print(" V = ", V);
  if(V == 0) { return 0; }
  r1 = V/2;
  Error = 1;
  K = 0;
  while( Error > 0.01)
  {
    K = K+1;
    r2 = 0.5*(r1+V/r1);
    Error = abs((r2-r1)/r1); // Hope r1 not 0
    print("K, r1, r2, Error*100");
    r1 = r2;
  }
  return r2;
}

\textsuperscript{11} CMAP’s built-in function sqrt() uses a more robust and efficient algorithm.
Food for thought: what are the key-ideas in the above program? Again, key-idea is the one without which we'll have trouble no matter how well we know the language.

Sample Program 2.6 - The Machine Epsilon

Write a program to divide E (equal to 1 initially) repeatedly by 2 as long as

\[(1 + E/2)\] is greater than 1

Food for thought: Isn't \((1 + E/2)\) always greater than 1?

Once found, E is the smallest value that makes \(1+E\) greater than 1. The program confirms that, using the final computed E, the following logical expressions are TRUE:

\[
\begin{align*}
1+E/2 &\ == \ 1 \\
1+E &\ > \ 1
\end{align*}
\]

E is called the Machine Epsilon. It plays a significant role in numerical analysis.

```c
main()
{    // Find the Machine Epsilon E
    E = 1;     N = 0;
    print(" N   E ");
    while (1+E/2 > 1) // Stop when 1+E/2 equals 1
    {
        E = E/2;   N = N+1;
        // Format to print E with 16 decimal places
        print(", %3.0f", N, "%25.16e", E);
    }
    print(" 1 + E/2 ==  1 ? ", 1+E/2 == 1,
        " 1 + E    >  1 ? ", 1+E > 1);
}
Sample Program 2.7 - Finding Limits of Math Functions

Write a program to find the limit of $\frac{\sin x}{x}$ as $x$ (in radians) tends to zero. Note: To avoid division by zero, we let $x$ tend to the Machine Epsilon instead.

```c
main()
{
    E = 2^(-52); // Machine Epsilon
    x = 1;
    print(", " x sin(x)/x);
    useoption("RADIANS");
    while((x = x/2) >= E)
    {
        print(", %25.16e", x, "%25.16e", sin(x)/x);
    }
}
```

Sample Program 2.8 - Factorial of Integers

This program computes the factorials of the first 10 integers.

```c
main() {
    F = I = 1;
    while(I <= 10) {
        F = F * I;
        print(" Factorial of ", I,
              ", ", " F);
        I=I+1;
    }
}
```

Generally, since CMAP float can retain at most 16 digits, the factorial of large integers quickly loses accuracy.
Food for thought: use the preceding program to find the largest integer whose factorial is exact (http://www.tsm-resources.com/alists/fact.html).

Sample Program 2.9 - Loss of Significance Digits in Subtraction

The following expressions for $E_1$ and $E_2$ look different but are mathematically identical (i.e. one can be derived from the other):

$$E_1 = \sqrt{A + \varepsilon} - \sqrt{A} \quad \text{and} \quad E_2 = -\frac{\varepsilon}{\sqrt{A + \varepsilon} + \sqrt{A}}$$

Write a program that compares the results computed for $E_1$ and $E_2$ with $A = 100$ and decreasing values of $\varepsilon = 1, 0.5, 0.25 \ldots 2^{-52}$

The output shows that the accuracy of $E_1$ deteriorates as $\varepsilon$ gets far too small. This loss of accuracy occurs because the difference between $\sqrt{A + \varepsilon}$ and $\sqrt{A}$ is so small that the result falls out of the range of 15-digit storage as depicted below (with $A = 1$):

```
main() {
  A = 100; Ep = 2; Min = 2^(-52);
  print(" Ep E1 E2", " % Diff");
  while((Ep = Ep/2) >= Min) {
    E1 = sqrt(A+Ep)-sqrt(A);
    E2 = Ep/(sqrt(A+Ep)+sqrt(A));
    print(_, Ep, E1, E2, "%15.4f", (E2-E1)*100/E2);
  }
}
```
Sample Program 2.10 - Summation of Infinite Series

Calculators and computers compute trigonometric functions often by summing sufficient number of terms of an infinite series. The Taylor series for sin(x) is:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

where x must be in radians\(^{12}\).

Larger x would need more terms to be summed. Thus, a loop is needed to sum a sufficient number of terms. Here is an algorithm:

- Assign a value to x
- Initialize: (first term)
  - n = 1;
  - Sum = Term = x;
  - Sign = 1;
- Repeat while the magnitude of Term is still too big
  (Generate the next term & add into Sum)
  - Assign (n+2) to n
  - Assign (- Sign) to Sign
  - Assign [Term * (x*x)/(n*(n-1))] to Term
  - Accumulate (Sign * Term) into Sum

```c
main()
{
    x = pi#/2; print(\n        " x =", x);
    n = 1; Sum = Term = x; Sign = 1;
    while (abs(Term) > 1e-6)
    {
        n = n+2;
        Sign = -Sign;
        Term = Term * (x*x)/(n*(n-1));
        Sum = Sum + Sign*Term;
    }
}
```

\(^{12}\) Radian is dimensionless: https://en.wikipedia.org/wiki/Radian
Food for thought: how is it that the factorial is not explicitly computed, and what are the other key-points in the above program?

Sample Program 2.11 - Schedule of Loan Payments

John got a bank loan of $L = 7200 to buy a used car. The annual interest rate is $I = 5.9\%$, and the monthly payment is $P = 350$.

The monthly interest rate is

$$i = I/12 = 0.00491667$$

The current balance is $B = L = 7200$.

On the first day of the following month, John pays the amount $P$ which includes both interest and capital payments (for the preceding month):

- Interest paid: $Pi = i * B$
- Capital paid: $Pc = P - Pi$
- New balance: $B = B - Pc$

With the new balance $B$, the preceding payment cycle repeats until the final balance is paid.

The following program prints the schedule of payments which shows the monthly status.

```c
main()
{
    L = B = 7200; I = 0.059; P = 350; i = I/12;
    M = 0;
    print("Initial balance = ",B," Month",
          " Interest    Capital    Balance");
    while (B >= P)
{ 
    M = M + 1; // Month
    Pi = i * B;
    Pc = P - Pi;
    B = B - Pc;
    print("", M, Pi, Pc, B);
}
print("", " Final payment = ", B);

Food for thought: what are the key points in the above program?

Practice Drills 2.4

1. Run the following programs (one at a time), and explain the output and why looping never stops. Terminate it by pressing 'Esc' key repeatedly. Next, make only one change in the program so that the loop will terminate.

   a) 

   main()
   {
       I = 0; // Counter
       print("Infinite loop. Press ESC to escape",^);
       while(1) { 
           print(I = I+1);
           if(mod(I, 6) == 0) { print(^); } 
       }
   }

   b) 

   main()
   {
       I = 0.1;
       print("Infinite loop. Press ESC to escape",^);

   ...
2. Make only one change in the following program so that the loop will terminate. Explain the output.

```c
while (I != 1) {
    print(^, I);
    I = I + 0.1;
}
```

3. Make only one change in the following program so that the loop will terminate. Explain the output.

```c
main()
{
    x = 1;
    while(x < 10) {
        print(^, x);
    }
}
```

4. Modify the Sample Program 2.3 to allow the user 5 chances to enter the correct PIN of 5179--which presumably was read off the data base.

5. Modify the Sample Program 2.4 (as needed in order to get the same output) while replacing the first two expressions with:
6. Explain the output of the following program. Observe the use of an arithmetic expression to control looping.

```c
main()
{
    A = 11;
    while( A = A-1 )
    {
        print(", A);
    }
    print(", " Out of the loop. Last A = ", A);
}
```

7. Observe the use of `yesno()` to control looping, and explain the output.

```c
main()
{
    N = 0; // Counter
    while( yesno("Continue?") ) { }
        print(", N = N+1);
    }
}
```

8. Observe the use of a counter to control looping and explain the output.

```c
main()
{
    I = 1; // Initialize counter
    Max = 10;
    while(I <= Max)
```
{  
    print(^,I);  // Example usage of counter  
    I=I+1;  // Increment counter  
}
print(^" Out of loop. Last I = ", I);

9. In the preceding program, initialize the counter I to 0 and then modify the program accordingly so that the loop produces the same output.

10. Modify the program in Drill 8 above so that the loop produces the output in reverse order (i.e. 10, 9, . . . 1).

11. Write a program to compute the following series, with \( M = N = 21 \):

\[
P = 1 \times 3 \times 5 \times \cdots M = \prod_{k=1,3,\ldots}^{M} k
\]

\[
S = 1 + 3 + 5 + \cdots + N = \sum_{k=1,3,\ldots}^{N} k
\]

12. Modify the Sample Program 2.10 to accommodate the following changes

- Initialize:
  - \( n = 3 \);
  - \( \text{Sum} = \text{Term} = x; \)
  - \( \text{Sign} = -1; \)

13. Write a program that keeps requesting a value as long as the user so wishes. The program should print every value, and finally print the value that has the largest magnitude.

Hint: Initialize \( \text{Max} = 0 \). Use a loop to get the input and save the input that has larger magnitude into \( \text{Max} \):

\[
\text{if(abs(Input) > Max)} \{ \text{Max} = \text{Input}; \}
\]

14. Let \( P \) be the price of an item this year. Next year, the price will be \((P + r P)\) where \( r \) is the rate of change\(^{13}\) which usually varies from year to year.

\(^{13}\) If \( r \) is positive, then we have price inflation, while the opposite is called price deflation. The rate of change \( r \) is usually expressed in percentage.
Given \( P = $1000 \) for the first year, and a fixed rate \( r = 5\% \) for the next 10 years. Write a program that prints the price of the item year by year for 10 years.

15. Modify the Sample Program 2.2 to process many "Scores".

**Exercise 2.1**

1. The formula for conversion of temperature in Celsius to Fahrenheit is

\[
\text{°F} = \frac{9}{5} \text{°C} + 32
\]

Write a program to produce the conversion table for temperature ranging from \(-30 \text{°C}\) to \(+35 \text{°C}\) in step of 1 \text{°C}.

2. Let \( a, b, c \) be the lengths of the three sides of a triangle. The area of the triangle is known as:

\[
A = \sqrt{s(s-a)(s-b)(s-c)}
\]

where \( s = (a + b + c)/2 \)

Write a program that will repeatedly prompt the user for \( a, b, c \) and print the area of the triangle. The program should ask the user if s/he wishes to continue with another triangle. The program should also test for the validity of the input.

3. Write and test a program to compute \( \pi \) accurate to within \( ±0.001 \) by summing as many terms as necessary of the following series:

\[
\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)
\]

4. The cube root of \( A \) is \( r = A^{1/3} = \sqrt[3]{A} \) when \( A = r^3 \). It may be found using the following iterative formula based on a previously estimated root \( r_1 \):

\[
r_2 = \frac{1}{3} \left( \frac{2r_1^3 + A}{r_1^2} \right)
\]

Write a program that will prompt the user for a positive value \( A \), and compute the cube root of \( A \) accurate to within 1\% – Adapt the algorithm in problem 6 for this problem, and use \( r_1 = A/3 \) for the initial estimate.
5. Write a program that will compute the approximate value of $\pi$ using Monte Carlo simulation as explained below.

Consider a circle of diameter $d$ that fits snugly inside the square as shown in the following figure. The ratio of their areas is:

$$\frac{A_C}{A_S} = \frac{\pi d^2}{4d^2} \quad \text{Thus:} \quad \pi = 4 \frac{A_C}{A_S}$$

In Monte Carlo method, we randomly throw $N$ darts onto the square board and count the number, $K$, of darts that fall inside the circle. For increasingly large values of $N$, the ratio $K/N$ should approach the ratio $A_d/A_s$. Thus, the approximate value of $\pi$ is then: $\pi \approx 4 \frac{K}{N}$

The computer program simulates dart throwing by generating pairs of (suitably scaled) uniformly distributed random numbers $x_i, y_i$ to represent the hit-point of the $i^{th}$-dart. The point is inside the circle if

$$x_i^2 + y_i^2 \leq \left( \frac{d}{2} \right)^2$$

The built-in function `rand()` generates random numbers uniformly distributed between 0 to 1. Thus, if $d$ is the diameter of the circle, we scale the returned value of `rand()` to fit the range $[-d/2 \text{ to } d/2]$ as follows

$$x_i = (\text{rand()} - 0.5) \times d;$$
$$y_i = (\text{rand()} - 0.5) \times d;$$

6. Refer to the situation in Sample Program 1.2. While keeping the vertical distance $H = L \sin(\alpha) + d = 0.25 \text{ m}$ constant, find the combination $(\alpha, d)$ that minimizes the maximum stress (at $A$) given by:

$$\sigma_{\text{max}} = \frac{N_a}{A_a} + \frac{M_a b}{2I_a}$$

where $b = 0.1 \text{ m}, \ t = 0.05 \text{ m}, \ A_a = bt, \ I_a = \frac{tb^3}{12}$

Hint: Use a loop to generate $\alpha$ varying from $25^\circ$ to $40^\circ$ in step of $0.5^\circ$, compute $\sigma_{\text{max}}$ and keep saving the smallest $\sigma_{\text{max}}$ (see Drill 13 of Practice Drills 2.4).
7. Write a program that computes values of \( e = \lim_{x \to 0} (1 + x)^{1/x} \) for \( x = 1, 0.1, 0.01, 0.001, \ldots, 1.0 \times 10^{-15} \). The program should also print the change in the computed value of \( e \) as compared to its previously computed value.

Run the program and explain why the result deteriorates as \( x \) becomes too small.

8. Write the program that computes the Euler's number:

\[
e = 1 + \frac{1}{1\times2} + \frac{1}{1\times2\times3} + \frac{1}{1\times2\times3\times4} + \cdots + \frac{1}{1\times2\times3\times4\times\cdots\times\infty} = \sum_{k=0}^{\infty} \frac{1}{k!}
\]

by summing sufficient number of terms in the series to obtain an accuracy of at least 6 decimal places. The program should also print the value of each new term added. Note: Avoid the use of the factorial function.

2.7 The do/while Loop Control Structure

The do/while-loop is similar to the while loop except that the loop body is executed before testing the condition. Thus, the loop body is executed at least once.

Format/syntax (the first part of the rules):

```
do
{ // Loop body as a block within curly braces
  
  
} while(Expr); // Mandatory semicolon
  
  
// First statement following the while-loop
```

Action (the other part of the rules): It executes the loop body, then tests the value of \( \text{Expr} \), and as long as \( \text{Expr} \) is non-zero, the loop body is executed. Once \( \text{Expr} \) becomes 0, execution continues from the first statement following the loop body.
Actions of the do-while loop control

Practice Drills 2.5

1. Modify the programs in Section 2.6 to use do-while loop instead of while-loop.

2. Use the do-while-loop to solve the problems in Exercise 2.1.

2.8 The `for` Loop Control Structure

Format/syntax (the first part of the rules):

The for-loop construct contains three expressions separated by semicolons:

```plaintext
for(Expr1; Expr2; Expr3)
{
   // Loop-body as a block within curly braces
   . . .
}
. . . // First expression following the loop
```
Action (the other part of the rules):

When an expression is omitted, its value is 0 by default.

In common usage, a *loop-index variable* appears in the three expressions:

```c
main()
{ // Sum the first 10 integers
    Sum = 0;
    for(I=1; I <= 10; I = I+1) // I: loop-index variable
    {
        Sum = Sum + I;
        print (^,I, Sum);
    }
}
```

The roles of the three expressions are as follows:

- **Expr1**: This expression, if it exists, is always executed, but only once, at the start. In common usage, it initializes the value of the loop-index variable: I = 1;

- **Expr2** provides the condition for executing the loop. If its value is non-zero, the loop body will be executed next, or else, the loop body is skipped. In common usage, it tests the loop-index variable against certain limit: I <= 10; // Execute the loop body while I <= 10
• Expr3 is executed AFTER the loop body has been executed. In common usage, it changes the loop-index variable so that eventually Expr2 will be zero in order to terminate the loop: \( I = I+1; \) // Increase I by 1, and so, eventually 'I' becomes larger than 10.

The following practice drills will hone your skills in using the for-loop.

**Practice Drills 2.6**

1. Modify the programs in Section 2.6 to use the for-loop instead of while-loop. Make sure that the output remains the same.

2. Use the for-loop to solve the problems in Exercise 2.1.

### 2.9 Nested Control Structures

Depending on the problem logic, a control structure may be nested within another control structure, and so on to any number of levels.

Consider the program that produces the multiplication table for integers from 1 to 12 (see output below). The algorithm makes use of two nested loops as follows.

Print integers 1 to 12 on one line
For row \( i \) = 1 to row \( i \) = 12 in step of 1
  
  Print \( i \) on the next line
  For column \( j \) = 1 to column \( j \) = 12 in step of 1
  
  if \( (j<i) \) : leave blank spaces
  else: print \( (i*\text{\textit{j}}) \) on the current line
  
}

**Multiplication Table**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>66</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>56</td>
<td>63</td>
<td>70</td>
<td>77</td>
<td>84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>90</td>
<td>99</td>
<td>108</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>110</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>121</td>
<td>132</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
main() {

N = 12;

print("^" Multiplication Table" ^ "  ");
for(i=1; i<=N; i=i+1) { // Print the top line
    print("%5.0f", i); // Note 1
}
print("  ");
for(i=1; i<=N; i=i+1) { print("_____"); }
for(i=1; i<=N; i=i+1){  // Outer loop. Note 2
    print("%5.0f", i); // Note 1
    for(j=1; j<=N; j=j+1) { // Inner loop. Note 3
        if(j<i) {
            print("  ");
        } else{
            print("%5.0f", i*j);
        }
    }
}
}

Food for thought: what are the key-points in the above program?

Notes:

1. The string "%5.0f " (prefixed by the percentage sign) is a formatting string that specifies how the next value is printed. Here, the value of i is to be printed in a width of 5 spaces with 0 decimal place. Read the specifications of print() for more info.

2. The outer loop controls the output for each row. Here, the index i varies from 1 to 12 in step of 1 (one value per row). Think of i as the row-index.

3. For each value of the row-index i (as defined by the preceding outer loop), this inner loop varies the "column-index" j from 1 to 12 in step of 1, and the product (i * j) is printed.

4. The if-else construct (within the inner for-loop) prints blank spaces in the lower part of the multiplication table. This lower part is detected when the column-index j is less than the row-index i.

© 2015 by K. Ha - BCEE 231 Chapter 2 - v4.3
Exercise 2.2

1. Write a program to produce the multiplication table of integers from 1 to 12 such that only the lower part of the table is printed. Hint: The resulting program should be simpler than the previously given version because trailing blank spaces need not be printed.

2. The probability of rolling a dice to get the 2-face is 1/6. What is the probability of getting 3 times the 2-face in 5 throws of the dice? Study the following program that determines such probability by Monte Carlo simulation.

```c
main()
{
    // Monte Carlo simulation of rolling dice.
    // Probability of getting the Target face nTimes in nThrows
    nSim = 1000;      // Number of simulations
    nThrows = 5;      // Number of throws per simulation
    Target = 2;       // Target face
    nTimes = 3;       // Times of rolling the Target
    nSuccess = 0;     // Total number of successes

    for(i = 0; i < nSim; i++ )
    {
        nHits = 0;
        for(j = 1; j <= nThrows ; j++)
        {
            // Generate a random number from 1 to 6
            Throw = floor(6*rand()+1);
            if(Throw == Target) // Is it the Target
            {
                nHits = nHits + 1;
            }
        }
        if(nHits == nTimes) // Check number of hits
        {
            nSuccess = nSuccess + 1;
        }
    }
    Prob = nSuccess/nSim;
    print("\n, " Probability = ", Prob);
}
```
2.10 The keyword `break`

The `break`-command transfers control out of the innermost `do`, `for`, or `while`-loop in which it appears. The `break`-command should be conditional, i.e. within an `if-else` block, or a `case`-statement. Thus, the `break`-command provides another mechanism for terminating the loop when a certain condition occurs.

**Examples**

Keyword `break` within the `while`-loop

```c
main()
{
    i=1;
    while(i<10) {
        if(i == 7) { break; } // Exit loop when i equals 7.
        print(^,i);
        i=i+1;
    }
    print(" Out of loop");
}
```

Keyword `break` within the `for`-loop

```c
main()
{
    for(i=1; I<= 10; I=I+1) {
        if(I == 7) { break; } // Exit loop when I equals 7.
        print(^,I);
    }
    print(" Out of loop");
}
```
Keyword **break** within the do/while-loop

```c
main()
{
    i=1;
    do {
        if(i == 7) { break; } // Exit loop when i equals 7.
        print(^,i);
        i=i+1;
    } while(i <= 10);
    print(^" Out of loop");
}
```

### 2.11 The keyword **continue**

The **continue**-statement *skips the remaining part of the loop body and starts the next iteration of the innermost do, for, or while-loop in which it appears*. Thus, the keyword **continue** provides the mechanism for excluding the last part of the loop body. This exclusion should be conditional.

Keyword **continue** within the while-loop

```c
main()
{
    i=0;
    while(i<10){
        i=i+1;
        if(i == 7)
        {
            print(^" Skipped");
            continue; // Skip rest of loop body
        }
        print(^,i);
    }
}
```
Keyword continue within the for-loop

```c
main()
{
    for(I=1; I<= 10; I=I+1)   {
        if(I == 7) { print(" Skipped"); continue; }
        print(",I);
    }
    print(" Out of loop");
}
```

Keyword continue within the do/while-loop

```c
main()
{
    i=0;
    do {
        i=i+1;
        if(i == 7) { print(" Skipped"); continue; }
        print(",i);
    } while(i < 10);
    print(" Out of loop");
}
```

2.12 The switch Structure for Multiple-Branch Selection

The switch control construct is more convenient than the if-else construct when there are many cases to test. See syntax and action here: http://beginnersbook.com/2014/01/switch-case-statements-in-c/
Example

Consider the following situation where the value of a variable M determines which block of statements to be executed:

- If M is 1 or 4 or 5 or 10: Execute block #1 only
- If M is 2 or 3 or 8: Execute block #2 only
- If M is 6 or 7 or 9: Execute block #3 only
- Default: M is any other value: Execute block #4 only

This situation is easily implemented with the `switch` construct as follows:

```c
main()
{
    do {
        M = getnum("Please enter an integer");
        switch(M) {
            case 1: case 4: case 5: case 10:
                notice(" Case 1,4,5,10 Block # 1"); // Block #1 here
                break; // Skip the rest. Get out of switch block
            case 2: case 3: case 8:
                notice(" Case 2,3,8. Block # 2"); // Block #2 here
                break; // Skip the rest. Get out of switch block
            case 6: case 7: case 9:
                notice(" Case 6,7,9. Block # 3"); // Block #3 here
                break; // Skip the rest. Get out of switch block
            default:
                notice(" Default: block # 4"); // Block #4 here
        }
    } while(yesno("Continue?") == YES);
}
```

Rules on using `switch`

1. The (optional) keyword `break` transfers control to the statement following the `switch` construct. If it is omitted, execution continues to the next case.

2. The (optional) keyword `default` should be the last one in order to catch all other possibilities that did not fit any of the previous cases.

The following rules are C & C++ standards:
3. The switch–value (following switch-keyword) may be replaced by an expression that evaluates\(^\text{14}\) only to an integer.

4. The case-value (after each case keyword) must be a constant integer\(^\text{15}\).

5. Within the same switch construct, the case values must be distinct\(^\text{16}\).

These last three rules are less restrictive in CMAP as follows:

3. The switch–value may be replaced by any expression.

4. A case-value may be replaced by any expression.

5. While duplicated case values would be a logical error, CMAP does not complain nor flag it as an error.

**Practice Drills 2.7**

1. Remove the break-commands from the preceding program, execute the resulting program and explain the output.

2. Explain the output of the following program:

   ```c
   main()
   {
   M = 2;
   switch(M) {
   case 1:
   case 2: print(^, " case 2");
   case 3: print(^, " case 3");
   case 4: print(^, " case 4");
   break;
   case 5: print(^, " case 5");
   break;
   default: print(^, " default");
   }
   }
   ```

   Repeat with different values for M.

3. Explain the output of the following program:

   ```c
   main()
   {
   for (j = 0; j < 2; j=j+1)
   {
   for (k = 0; k < 5; k=k+1)
   ```

\(^\text{14}\) This happens at runtime.

\(^\text{15}\) This constant integer is built into the executable code, and therefore it is cast in stone.

\(^\text{16}\) The code won't compile if this rule is violated.
{ if (k == 2) { break; }
    print(^, j, k);
}

4. Explain the output of the program in D3 after changing break into continue:

5. Rewrite the Sample Program 2.2 using the switch-structure.

### 2.13 Useful Built-in Functions

The following built-in functions have been introduced in this Chapter. Complete information on them is available in CMAP-Help.

- `inform( . . . )` Displays a window presenting menu items, text, values, etc.. If the user clicks a menu-item (displayed in the window), it will return the user’s choice.
- `mod(a, b)` Returns the remainder of the division a/b
- `notice(". . .")` Displays a message and waits for the user’s response.
- `rand()` Returns a pseudo random number uniformly distributed between 0 and 1.
- `readglobalvars()` Reads from disk the names and values of variables stored in a file.
- `saveglobalvars()` Saves all current global variables into disk file.
- `yesno(". . .")` Displays a Yes/No question and returns the user’s response (1: for Yes-button, 0 for No-button, and -1 for Cancel-button).
Chapter 3

User-Defined Functions

Each problem that I solved became a rule which served afterwards to solve other problems — René Descartes

Contents

Chapter 3 User-Defined Functions ................................................................. 1
3.1 The Fundamentals .................................................................................. 2
  1. Automatic Execution of main() ................................................................ 2
  2. Execution of Functions other than main() ............................................. 2
  3. Sequence of Execution of User-defined Functions ................................. 3
  4. Coding User-Defined Functions ............................................................ 5
3.2 Implementation of Math Functions ........................................................... 5
3.3 Built-in Functions can Use User-Defined Functions ............................... 7
  Practice Drills 3.1 ................................................................................... 9
  Practice Drills 3.2 ................................................................................... 12
3.4 Global Variables vs Local Variables ......................................................... 14
3.5 Sample Applications ............................................................................. 16
  Sample Program 3.1 - Solution of Quadratic Equations ............................ 16
  Sample Program 3.2 - Extreme Values of a Function ............................... 18
  Sample Program 3.3 - Nested if-else: Assign Letter Grades .................... 19
  Sample Program 3.4 - Evaluation of Infinite Series ................................... 20
  Practice Drills 3.3 ................................................................................... 22
  Sample Program 3.5 - Iterative, Bisection Method for Non-linear Equations 25
  Sample Program 3.6 - Graphics, Plotting ................................................. 27
3.6 Program Organization with Functions ...................................................... 30
  Practice Drills 3.4 ................................................................................... 30
3.7 Application: Design of a Simple User-Interface ...................................... 31
3.8 Useful Built-in Functions ....................................................................... 34
3.1 The Fundamentals

1. Automatic Execution of main()

When a program is executed, the *user-defined function* \(^1\) main() is **automatically** executed, and it is always the first function to be executed. Functions **other than** main() must be invoked by name in an expression for it to be executed.

In the following program, OtherFunction() is not invoked in any expression, hence it will not be executed.

```c
main() // Function heading
{ // Function body (instructions)
    notice(" main() is automatically executed");
}

OtherFunction() // Function heading
{ // Function body (instructions)
    notice(" OtherFunction() is executed");
}
```

2. Execution of Functions other than main()

CMAP *built-in functions* and user-defined functions other than main() need be invoked by name for execution.

---

\(^1\) Unlike built-in library functions, the coding of *user-defined functions* appears within the program. They can be typed in arbitrary order, but all must follow the same basic syntax/format:

```
// Coding of function: "Function definition"
FuncName( )
{
    . . . .
}
```

Since CMAP only deals with numerical computation, function coding in CMAP is much simplified compared to C-language ([http://www.c4learn.com/c-programming/c-function-definition/](http://www.c4learn.com/c-programming/c-function-definition/))
3. Sequence of Execution of User-defined Functions

The following program has 4 user-defined functions: main(), fA(), fB(), fC() whose coding (i.e. function definition) may appear in any order.

main() invokes fA() in an expression, which then invokes fB(), which then invokes fC(). When fC() finishes, control returns to (where it left in) fB(), which then returns to (where it left in) fA(), which finally returns to (where it left in) main(). And when main() finishes, the program terminates.
\begin{verbatim}
fC() {
    notice(" fC() is executed. Last in chain. ");
}

main() {
    trace(1); // Trace function execution sequence
    notice(" main() is automatically executed");
    fA() +
    notice(" Back into main()");
    trace(0); // Turn off tracing
}

fB() {
    notice(" fB() is executed => Next fC()");
    fC() +
    notice(" Back into fB()");
}

fA() {
    notice(" fA() is executed");
    fB() +
    notice(" Back into fA()");
}
\end{verbatim}

Note: while viewing data, select the menu item: view > Program Line to see the program line currently under execution. Inserting function view() at strategic locations in the program enables us to follow program execution and inspect data in memory for unexpected behaviour.
4. Coding User-Defined Functions

Coding of user-defined functions can use any feature of the language, namely:

- Expressions and commands (key-words)
- Functions including built-in, user-defined, and even itself (if the logic makes sense).

3.2 Implementation of Math Functions

Math function takes input data and returns a value. For example, \( \sin(\pi/6) \) takes \( \pi/6 \) as input and returns 0.5. Math functions can be used anywhere and as many times as needed.

The following example shows an implementation of the user-defined function \( \text{HYPOT}(a, b) \). On execution, this function takes the arguments \( (a, b) \) and returns the value \( \sqrt{a^2 + b^2} \).

```c
main()
{
    // Test HYPOT() function
    A = 2 * HYPOT(3, 4); // 3, 4: arguments
    B = HYPOT(a=5, b=4)+HYPOT(sin(a),cos(b));
    print(^, a, b, A, B, HYPOT(A, B));
}
```

// Definition of HYPOT()

```c
HYPOT(float a, float b) // a, b: parameters
{
    // Purpose: compute and return \( \sqrt{a^2 + b^2} \)
    float Value; // Declare local variable
    Value = sqrt(a*a+b*b);
    return Value; // Return value
}
```

Parameters:
- \( a \)
- \( b \)

as (temporary) local storage of data sent over by function call.

Concepts:
(a) **Function arguments** are the data sent to the function. Example: In this calling expression:

\[
A = 2 \times \text{HYPOT}(3, 4);
\]

The values 3, 4 are *arguments* of HYPOT().

(b) **Function parameters** are the names 'a', 'b' declared in the heading of the function definition. They are containers for storing the values of the arguments: 'a' contains 3, 'b' contains 4 (Rule: first argument to first parameter, second argument to second parameter, and so on. The type of argument must match the *declared type* of the corresponding parameter).

(c) The **return** keyword: (i) terminates execution of the function; (ii) transfers control back to where it was in the calling expression; and (iii) supplies the returned value for use as the function’s value in the calling expression.

We now improve the Sample Program 2.10 by using user-defined functions. The revised program produces a table of sine of angles.

```c
main()
{
    float A, S; // Local vars of main()
    print("     A     Sin(A)"^);
    for(A = 0; A <= 90; A = A+1) // Angle in degrees
    {  // Convert degrees to radians for series
        S = Sin(A*pi#/180, 1e-8); /* two arguments
                                in function call */
        print(^,"%6.0f", A, S);
    }
}

Sin(float x, float MaxErr) /* parameters x and MaxErr
as local vars of Sin() to store values sent as
function call arguments */
{
    // Compute and return sin of x (radians)
```
```c
float n = 1, Sum = Term = x, Sign = 1; // Local
while (abs(Term) > MaxErr)
{
    n = n+2;
    Sign = -Sign;
    Term = Term * (x*x)/(n*(n-1));
    Sum = Sum + Sign*Term;
    // print(^, n, Sign*Term, Sum);
}
return Sum; // Bring back value Sum as sin of x
```

Observe:

- The data type of every parameter must be declared (e.g. type float for decimal value) in the function heading.
- Function parameters are local variables for storing the values of function arguments (sent by the calling expression).
- One-to-one matching of arguments and parameters.
- How additional local vars are declared in the body of main() and Sin().
- The return statement in Sin() brings back the computed value Sum to the function call. Without return, zero is returned, by default.

These points are crucial for implementing and usage of functions.

### 3.3 Built-in Functions can Use User-Defined Functions

Let F(x, . . ) be a user-defined function (or an expression) that returns a value for any x. Such a function/expression may be used in built-in functions for a variety of purposes:

- Plot the curve F(x) in the interval \( x_1 \leq x \leq x_2 \)

  \[ \text{plot}(x, x_1, x_2, F(x, . . . )); \]

- Find \( x \) (within the interval \( x_1 \leq x \leq x_2 \)) that makes \( F(x) \) equal to 0:

  \[ x_0 = \text{root1}(x, x_1, x_2, F(x, . . . )); \]
• Find \( x \) (within the interval \( x_1 \leq x \leq x_2 \)) to maximize \( F(x) \):

\[
x_0 = \text{max}z_1(x, x_1, x_2, F(x, \ldots));
\]

• Find \( x \) (within the interval \( x_1 \leq x \leq x_2 \)) to minimize \( F(x) \):

\[
x_0 = \text{min}z_1(x, x_1, x_2, F(x, \ldots));
\]

• Find the derivative of \( F \) wrt \( x \) (i.e. \( dF/dx \)) at \( x_0 \):

\[
\text{Der} = \text{deriv}(x, x_0, F(x, \ldots));
\]

• Find the definite integral of \( F(x) \) for the interval \( x_1 \leq x \leq x_2 \):

\[
\text{Int} = \text{integ}(x, x_1, x_2, F(x, \ldots));
\]

**Dummy variables**

The name '\( x \)' is a dummy variable name for use only in the expression; any other name will do. It is a temporary variable to hold a value which will be assigned automatically by CMAP built-in functions. For each such value in \( x \), the user-defined function \( F(x, \ldots) \) returns a value.

A dummy variable is recognized only within the expression that uses it, and it supersedes all other variables (in the program) of the same name. The following program proves this point.

```cpp
main()
{
    float x = 10;  // x: local var of main()
    clearplot();
    useoption("RADIANS");
    plot(x, 0, 2*pi#, f(x)); // x: dummy var
    cat();
}

f(float u)
{
    float x = u^2; // x: local var of f()
    return exp(-0.2*x)*sin(x);
}
```
Practice Drills 3.1

1. Consider the following mathematical functions:

- \( f(x) = \sin(x) e^{-x/2} \)
- \( g(x, y, z) = \exp(xy) / z \)

Instead of variable names \( x, y, z \), we could use other names, and the functions remain essentially the same:

- \( f(u) = \sin(u) e^{-u/2} \)
- \( g(u, v, t) = \exp(uv) / t \)

We can compute functions' values given any valid argument values, for example:

\[
A = f(2) = \sin(2) e^{-2/2} = 0.334512 \\
B = f(-1) - g(0.5, -1, 2) \\
\quad = \sin(1) e^{-1/2} - e^{0.5(-1)} / 2 = -1.69062
\]

In the following program, these math functions are implemented as stand-alone, independent user-defined functions, facilitating usage similar to math functions.

```cpp
f(float x) // Function heading
{
    float Value = sin(x) * exp(-x/2);
    return Value;
}

g(float x, float y, float z)
{
    return exp(x*y) / z;
}
```
```c
main()
{
    useoption("RADIANS");
    float A = f(2),
          B = f(-1) - g(0.5, -1, 2);
    print(^, " A = ", A, "   B = ", B);
    clearplot(); // Delete existing plots
    plot(t, 0, 2*pi#, f(t));
    setop(C, 9); // Set plot 'c'olor 9 (blue)
    plot(t, 0, 2*pi#, g(-1, t, 3));
}
```

After studying the working of the above program, modify the program for each of the following changes and test its working:

a) Eliminate the local var "Value" from the definition of f()

b) Replace x, y, z in g() by u, v, t

c) Replace t in plot(t, 0, 2*pi#, f(t)) by z or any other name (except reserved names such as pi#, e#, etc...)

d) Replace expression plot(t, 0, 2*pi#, g(-1, t, 3)) by plot(t, 0.5, 2*pi#, g(-1, 1, t));

and explain what is being plotted now.

Observe that f(), g() can be used like math functions.

2. Name such as t in plot(t, 0, 2*pi#, f(t)) is a dummy variable name. Function plot() automatically assigns value to t, sends it to f(float x) to be stored in the parameter x.

To find out which values and how many are set for t, we print x within f(float x) as follows:

```c
float K; // Counter of points for plotting
```

```c
main()
{
    clearplot();
    K = 0; // Reset counter to zero
    print(" K x f(x) ");
```
plot(t, 0, 2*pi#, f(t));
}

f(float x) // Function heading
{
    float Value = sin(x)*exp(-x/2);
    print('^, K = K+1, x, Value);
    return Value;
}

Dummy variables provide temporary storage for use only within the expression. They disappear after the expression has been executed.

3. With functions \( f() \), \( g() \) implemented as in Drill D1, rewrite \( \text{main}() \) to perform the following tasks.

**Notes:** Before plotting, you may wish a) to reset \( K \) to zero; b) to use \( \text{clearplot()} \) to erase the current plot, and \( \text{setop(C, . .)} \) to change pen color; c) to open a new draw document \( \text{newdrawdocument();} \) or d) to use \( \text{setrange()} \) to define a data range for plotting.

a. Compute \( D = \frac{d f(u)}{du} \bigg|_{0.4} \) i.e. Differentiate \( f(u) \) wrt \( u \), and then substitute \( u \) by 0.4.
   
   Ans: \( D = \text{deriv}(u, 0.4, f(u)) \); giving \( D = 0.594687 \)

b. Compute \( E = \frac{d g(x, -1, 2)}{dx} \bigg|_{2.1} \) i.e. Differentiate \( g(x, -1, 2) \) wrt \( x \), and then substitute \( x \) by 2.1.
   
   Ans: \( E = \text{deriv}(x, 2.1, g(x, -1, 2)) \); giving \( E = -0.0612282 \)
   \( x \) is a dummy variable name.

c. Compute \( F = \int_{0}^{\pi} f(x)dx \) i.e. Integrate \( f(x) \) wrt \( x \) from 0 to \( \pi \) giving a value
   
   Ans: \( F = \text{integ}(x, 0, \pi#, f(x)) \); giving \( F = 0.966304 \)
   \( x \) is a dummy variable name.
d. Compute $G = \int_0^\pi g(1, t, 2) \, dt$
   
   Ans: $G = \text{integ}(t, 0, \pi, g(1, t, 2))$; giving $G = 11.0703$
   
   $t$ is a dummy variable name.

e. Find $0.1 \leq x \leq 2$ so that $f(x) - g(-1,1,x) = 0$
   
   Ans: $x_0 = \text{root1}(x, 0.1, 2, f(x) - g(-1,1,x))$; // $x_0 = 0.774728$
   
   $x$ is a dummy variable name.

f. Find $2 \leq x \leq 6$ so that $f(x) - g(-1,1,x) = 0$
   
   Ans: $x_1 = \text{root1}(x, 2, 6, f(x) - g(-1,1,x))$; // $x_1 = 2.59599$
   
   $x$ is a dummy variable name.

g. Find $0 \leq x \leq 2\pi$ so that $\int_0^x g(1,t,2) \, dt = 10$
   
   Ans: $x_2 = \text{root1}(x, 0, 2\pi, \text{integ}(t,0,x,g(1,t,2)) - 10)$;
   
   // $x_2 = 3.04454$
   
   $x$ and $t$ are dummy variable names.

h. Plot $h(t) = \left. \frac{d}{du} f(u) \right|_{u=t}$ for $0 \leq t \leq 2\pi$ i.e. Differentiate $f(u)$ wrt $u$, and then substitute $u$ by $t$, giving $h(t)$; next plot $h(t)$ by varying $t$.
   
   Ans: $\text{plot}(t, 0, 2\pi, h = \text{deriv}(u, t, f(u)))$;
   
   or more simply: $\text{plot}(t, 0, 2\pi, \text{deriv}(u, t, f(u)))$;
   
   $t$ and $u$ are dummy variable names.

i. Plot $\int_0^t f(x) \, dx$ for $0 \leq t \leq 2\pi$ i.e. Integrate $f(x)$ wrt $x$, and then substitute $x$ by $t$, giving $h(t)$; next plot $h(t)$ by varying $t$.
   
   Ans: $\text{plot}(t, 0, 2\pi, h = \text{integ}(x, 0, t, f(x)))$;
   
   or more simply: $\text{plot}(t, 0, 2\pi, \text{integ}(x, 0, t, f(x)))$;
   
   $t$ and $x$ are dummy variable names.

Practice Drills 3.2

1. For each of the following mathematical functions of one variable:
   
   • Write a user-defined function to compute and return the value of the math function for any given $x$.
   
   • Write $\text{main}()$ to test the function as follows:
     
     o Compute and print the value of the function for $x$ varying from 0 to 1 in step of 0.1.
     
     o Plot the function for the given interval.
2. For each of the following mathematical functions of two variables $x$, $y$:
   • Write a user-defined function to compute and return the value of the function for any $x$, $y$ given as function parameters.
   • Write main() to test the function as follows:
     o Compute and print the values of the function at the nodes of a 5 by 5 rectangular grid (divided by 6 lines in $x$ and $y$ direction) equally spaced within the rectangular domain defined by the given limits of $x$ and $y$.
     o Plot the surface of the function over the given intervals.

   (a) $f(x, y) = \sin x \sin y \quad 0 \leq x, y \leq \pi$
   (b) $f(x, y) = 0.6xy \quad 0 \leq x, y \leq \pi$

3. The gravitational force $F$ between two masses $m_1$, $m_2$ is given by:

   $$F = G \frac{m_1 m_2}{r^2}$$

   where

   $G = \text{Gravitational constant}^2$
   $m_1, m_2$: The two masses in kg
   $r$: Distance in meters between the two masses.

   $F$ is defined by the user-defined function:

   ```c
   F(float m1, float m2, float r)
   {
       return G# * m1 * m2 / r^2;
   }
   ```

   Given that:

   Radius of the earth: $r = 6.37131e6$ m
   Mass of the earth: $m = 5.98e24$ kg

   Write main() to compute the following, making use of the function $F()$:

   $G = 6.673 \times 10^{-11} \text{ N.m}^2/\text{kg}^2$ is the CMAP pre-defined constant $G#$
4. Write and test the following function of two variables.

\[
f(x, y) = \begin{cases} 
  x + y & x \geq 0, y \geq 0 \\
  x + y^2 & x \geq 0, y < 0 \\
  x^2 + y & x < 0, y \geq 0 \\
  x^2 + y^2 & x < 0, y < 0 
\end{cases}
\]

Plot the 3-D surface of the preceding function \( f(x, y) \) using the following expression:

```
sp_surface(x, -2, 2, y, -2, 2, f(x, y));
```

3.4 Global Variables vs Local Variables

Global variables by default

In CMAP-application programs, undeclared variables are *global* by default. They are available for use (i.e. visible) in every user-defined function (assuming no local variables of same names). This reduces the need to have many arguments/parameters in user-defined functions--convenient for quick, ad hoc programs.

Global variables by explicit declaration

Variables can also be explicitly declared *global* by type-declarations placed outside of ALL functions. See example program below.

Local variables must always be explicitly declared

A type-declaration placed WITHIN a user-defined function declares *local variables* of that particular function.

---

\(^3\) The reduction in weight is a mere 8.8%. Nevertheless, astronauts appeared to float in "weightless" space because they were actually free falling like skydivers. The reduction in elevation is compensated for by the "falling" curvature of the earth surface as the spaceship orbiting the earth.
float B = 2*(A = 10), C = 30, D = 40; // Global vars

main()
{
    cat();
    float A = 100, D = 300, a, d; // Local vars
    C = C * 10; // Global
    A = A * 10; // Local supersedes global
    D = D * 10; // Local supersedes global
    a = 1; d = 2; // Local of main()
    cat();
    view();
    fA();
    cat();
    view();
}

float E = 50, F = 60, G = A + B; // Global vars

fA()
{
    float B = 2, C = 3, a, d; // Local vars
    B = B * 10; // Local supersedes global
    C = C * 10; // Local supersedes global
    A = A * 10; // Global
    E = E * 10; // Global
    a = 10; d = 20; // Local of fA()
    cat();
    view();
}

Local variables of a particular function:

- are available for use only within that function alone (for protection, privacy). Other functions cannot use, see, or change them;
- are deleted when the function returns;
- supersede global variables of the same names.

Because of these characteristics, local variables should always be used for data internal to the function (for data protection, no worry of accidental use of names in other functions, or of other functions' tampering with their values).
3.5 Sample Applications

Sample Program 3.1 - Solution of Quadratic Equations

This example solves the real roots of quadratic equations $Ax^2 + Bx + C = 0$ for any values of $A$, $B$, $C$. It is based on the program in Section 2.2. It has the following additional features:

- Use of do-while loop to process many sets of coefficients $A$, $B$, $C$. To stop the loop, the user must make the request to discontinue.

- Use of the user-defined function Quad($a$, $b$, $c$) to get the real roots of the quadratic equation $ax^2 + bx + c = 0$. Function Quad() prints the solution and returns the number of real roots found.

The flowchart of the major steps is shown below. The chart also demonstrates the transfer of control to and back from function Quad().

Observe that this more complex program is broken down to sub-tasks divided between functions main() and Quad(). Such task division also simplifies the logic within individual functions.

Also note that the program does not create any global variable. This is a good practice of defensive programming in complex programs that has a large number of user-defined functions.

```c
main()
{
   // Roots of quadratic equations $Ax^2 + Bx + C = 0$
```

© 2015 K. H. Ha – BCEE 231 Chapter 3 - v4.3
```c
float A, B, C;      // Local variables of main()
do {
    A = getnum(" Enter value of A", A);
    B = getnum(" Enter value of B", B);
    C = getnum(" Enter value of C", C);
    Quad(A,B,C);       // Note #1 (see below)
} while (yesno(" Continue? "));

Quad(float a, float b, float c)   // Note #2
{ // Real roots of quadratic equations \( ax^2 + bx + c = 0 \)
    float D, x1, x2;       // Note #3
    if(a==0) {
        if(b==0) {
            notice(" No solution");
            return 0;        // Return 0. Note #4
        }
        x1 = -c/b;
        notice(" One root only: ", x1);
        return 1;          // Count of root
    }
    D = b*b-4*a*c;
    if(D < 0) {
        notice(" Imaginary roots only");
        return 0;          // Count of root
    }
    x1 = (-b+sqrt(D))/(2*a);
    x2 = (-b-sqrt(D))/(2*a);
    notice(" First root: ", x1);
    notice(" Second root: ", x2);
    return 2;              // Count of root
}
```
Sample Program 3.2 - Extreme Values of a Function

The following examples illustrate a common technique for searching values returned by a function, looking for its maximum or minimum value.

The tension $T$ in a cable of a system is a function of a certain distance $x$ as given by\(^4\)

$$T(x) = 10 - 3x + 0.9x^2 \quad -2 \leq x \leq 3$$

The program finds the value of $x$ at which the tension $T$ is minimum\(^5\).

We need to examine the function $T(x)$ for a range of $x$ within the interval $-2 \leq x \leq 3$.

To search the value $x$ where $T(x)$ is minimum, we may use the following brute-force strategy: Vary $x$ from $-2$ to $3$ in step of $0.5$. For each $x$, compute $T(x)$, and if $T(x)$ is the minimum so far, save $x$ into $x_0$.

Pseudo code:

(a) Initialize $T_{\text{Min}} = T(-2)$;
(b) for $x$ varies from $-2$ to $3$ in step of $0.5$:
   1. Compute $T_x = T(x)$
   2. If $T_x$ is less than $T_{\text{Min}}$, save $x$ into $x_0$
(c) Print $x_0$ and $T_{\text{Min}} = T(x_0)$.

```c
float a = -2, b = 3, Inc = 0.5, x, xo;  // Declare globals

main()
{
    print(^^" x P"^^);  // Heading
    TMin = T(a);  // Will become minimum T
```

\(^4\) The program written here is not limited to any particular form of $T(x)$.
\(^5\) While the problem is more easily and efficiently solved using the built-in function \texttt{minz1()}, this example illustrates a programming technique that will be useful in many other situations.
for (x=a; x<=b; x=x+Inc)
{
    print(^,x, Tx = T(x));
    if (Tx < TMin) {
        TMin = Tx; // Save approximate Min P
        xo = x;     // Save xo = distance x at TMin
    }
}
print(^ " Approximate: Min P = ", TMin,
    " at xo = ", xo,^^);

T(float x)
{
    return 10-3*x+0.9*x^2;
}

Food for thought: what are the key points in this program?

Sample Program 3.3 - Nested if-else: Assign Letter Grades

The following function Grade(float M) prints a letter grade in according to the following scale:

<table>
<thead>
<tr>
<th>Marks, M</th>
<th>Letter grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 49</td>
<td>FNS</td>
</tr>
<tr>
<td>50 – 59</td>
<td>D</td>
</tr>
<tr>
<td>60 – 69</td>
<td>C</td>
</tr>
<tr>
<td>70 – 79</td>
<td>B</td>
</tr>
<tr>
<td>80 - 100</td>
<td>A</td>
</tr>
</tbody>
</table>

To activate the function, type it in a new document and click the eXecute icon. Now we can test the function in the Calc-Expressions Dialog by executing expressions such as: Grade(67); Grade(81);

Calc-Expressions Dialog is convenient for testing built-in functions as well as current, new or revised user-defined functions.
3.20

```c
{ // This version uses nested if/else constructs
    if(M < 0) { notice(" Error "); return; }
    if(M < 50) { print(" Grade FNS"); }
    else {
        if(M <= 59) { print(" Grade D"); }
        else {
            if(M <= 69) { print(" Grade C"); }
            else {
                if(M <= 79) { print(" Grade B"); }
                else { print(" Grade A"); }
            }
        }
    }
}
```

Sample Program 3.4 - Evaluation of Infinite Series

Special features: Iterative technique, error control

Consider the following power series for computing \( \tan^{-1} x \) or \( \tan^{-1} x \)

\[
\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots
\]

(a) Write the function \( \text{Atan}(\text{float } x) \) that returns the value of \( \tan^{-1} x \) accurate to at least 6 decimal places. The function should also print the number of terms that have been used.

(b) Write the main() function that computes \( \pi \) by the expression:

\[
\pi = 16\tan^{-1}\left(\frac{1}{5}\right) - 4\tan^{-1}\left(\frac{1}{239}\right)
\]

The approach\(^6\)

\(^6\) The given power series for \( \tan^{-1}(x) \) is fast converging for \( x < 1 \). In Chapter 2, we have seen its slow convergence for the case of \( x = 1 \):

\[
\frac{\pi}{4} = \tan(1) = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots
\]
Observing the terms of the power series \( \left( \frac{x}{1}, \frac{x^3}{3}, \frac{x^5}{5}, \frac{x^7}{7}, \ldots \right) \), let each term be represented by the ratio \( \frac{N}{D} \) where \( N \) stands for the numerator, and \( D \) for the denominator:

- The numerators \( N \): starting from the first value \( x \), the subsequent values of the numerators \( (x^2, x^5, x^7, \ldots) \) can be generated successively by multiplying the previous value by \( x^2 \). This process is more efficient than computing \( x^n \) for different \( n \).
- The denominators \( D \): starting from the first value 1, the subsequent values of the numerators \( (3, 5, 7, \ldots) \) can be generated successively by adding 2 to the previous value.

We will use a do-while loop to sum the next term \( \frac{N}{D} \) of the series as long as the current term is greater than \( 10^{-7}/2 \). This ensures that the omitted term does not affect the 6th decimal place of the sum.

```plaintext
main()
{
    float Pi = 16*Atan(1/5) - 4*Atan(1/239);
    print(" Pi = ",Pi, " Error = ", Pi-pi#);
}

Atan(float x)
{
    float  K = 0, // Counter of number of terms
           N = x, // First numerator
           D = 1, // First denominator
           Sum = x, // First term
           Sign = 1, // The sign
           Term; // A term of the series

    do
    {
        N = N * x*x; // Numerator of next term
        D = D+2; // Denominator of next term
        Sign = -Sign; // Switch the sign
        Term = N/D; // The current term
    } while (term > 10^-7/2);
}
```
Sum = Sum + Sign*Term;  // Accumulate term
    K = K+1;  // Increment counter
} while(Term > 1e-7/2);
print(" x = ",x,"  K = ",K," Atan x = ", Sum);
return Sum;  // Return the value of atan(x)
}

Output:

x = 0.2      K = 5      Atan x = 0.197396
x = 0.0041841  K = 1  Atan x = 0.00418408
Pi = 3.14159  Error = -9.73459e-010

Food for thought: what are the key points in this program?

Practice Drills 3.3

Note: Where possible, avoid the use of global variables in these exercise problems.

1. Incorporate the function in Sample Program 3.3 within a complete program where main() contains a loop that inputs Score and invokes Grade(Score) to print out the grades.

2. A function P(x) is defined as follows:

\[
P(x) = \frac{Wds}{(ys + cx)} \quad 0 \leq x \leq \sqrt{L^2 - y^2}
\]

where

\[
W = 98.1 \text{ N}, \quad L = 1 \text{ m}, \quad y = 0.4 \text{ m}
\]

Write a program to determine x where P(x) is maximum. The computed x should be accurate to the fourth decimal place. Print the value of x found, the corresponding P, and the total number of times that the expression for P(x) is evaluated.

Note: The built-in function `maxz1()` may be used to verify the computed solution.
3. (a) Write a user-defined function `Exp(float x)` to compute and return the value of $e^x$ using the following series. Use sufficient number of terms so that the result is accurate to at least 4 decimal places.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \left( \frac{x^{n-1}}{(n-1)!} \right) + \frac{x}{n} \left( \frac{x^{n-1}}{(n-1)!} \right) + \cdots$$

Hint: An efficient implementation would make use of the fact that the $x^n$-term is equal to the previous term times $(x/n)$. Use a do-while loop to generate the terms and accumulate them. The loop stops when the last term added is less than 0.0001.

```c
float Sum = Term = n = 1; // Initialization.
do { // Do as long as Term is still too big
    Term = Term * x/n; // Term x^n
    Sum = Sum + Term;  // Accumulation
    n = n+1;    // Next term
} while (abs(Term) > 0.0001); // Term still too big?
```

(b) Write the `main()` function to print the results for $x$ varying from 1 to 5 in step of 0.5. The output should be in four columns as shown below.

<table>
<thead>
<tr>
<th>x</th>
<th>Exp(x)</th>
<th>Percentage Error</th>
<th>Number of terms</th>
</tr>
</thead>
</table>

Notes: (i) To compute the percentage error, use the built-in function `exp(x)` to get the "exact" value; (ii) Use the format string "%.6f" to print a value with 6 decimal places (see CMAP-Helper, topic print).

4. The cube root of a value $A$ can be found using the following iterative formula based on a previously estimated root $r_1$:

$$r_2 = \frac{1}{3} \left( \frac{2r_1^3 + A}{r_1^2} \right)$$

Write the function `CubeRoot(float A)` that computes and returns the cube root of $A$ accurate to within 1%. Write the function `main()` that computes and prints the cube root of values input by the user.

---

7 In this algorithm, the computation of individual factorial and power term is avoided. Note that the given code fragment is only a part of the desired function.
Write the function \( \text{Cubic(float } a1, \text{ float } a2, \text{ float } a3) \) to solve for the real roots of the following cubic equation

\[
P(x) = x^3 + a_1x^2 + a_2x + a_3 = 0
\]

which can be written in a more computationally efficient form:

\[
P(x) = a_3 + x(a_2 + x(a_1 + x)) = 0
\]

To find the real values \( x_{i=1,2,3} \) that makes \( P(x_{i=1,2,3}) = 0 \), we use the following procedure.

- Compute

\[
Q = \frac{3a_2 - a_1^2}{9}, \quad R = \frac{9a_1a_2 - 27a_3 - 2a_1^3}{54}
\]

\[
D = Q^3 + R^2
\]

- If \( D \) is positive or zero, there are two complex roots which we are not interested in this exercise.

- Considering the case where \( D \) is non-zero negative, we can find the three real roots as:

\[
x_i = 2\sqrt{-Q} \cos\left(\frac{\alpha}{3}\right) - \frac{a_1}{3}
\]

where \( \alpha = \cos^{-1}\left(\frac{-R}{\sqrt{-Q^2}}\right) \)

Test your function \( \text{Cubic()} \) with the equation:

\[
P(x) = (x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6 = 0
\]

The program should also:

- Plot the graph of \( P(x) \) for the range \( 0 \leq x \leq 5 \) in order to verify the solutions. Note that the 3 solutions are obvious (from the given factors): 1, 2 and 3.

- Compute the definite integral \( \int_0^5 P(t) \, dt \) using the built-in function \( \text{integ()} \).

- Plot the integral \( \int_0^5 P(t) \, dt \) for the range \( 0 \leq x \leq 5 \)
- Find the non-zero root of \( \int_0^x P(t) \, dt \) in the range \( 0 \leq x \leq 5 \). Note: use \( \text{root1()} \) function.

Sample Program 3.5 - Iterative, Bisection Method for Non-linear Equations

In an iterative technique, repetitive cycles of computation are done where each cycle leads to a better solution. Iterative technique is necessary for non-linear problems.

For illustration, consider the following non-linear equation:

\[
F(x) = e^{-0.12x^2} - 3\cos x = 0 \quad 0 \leq x \leq \pi
\]

The objective is to find the value(s) of \( x \) (between 0 and \( \pi \)) that makes \( F(x) \) equal to zero. Such a value is called the root\(^8\) of the equation.

We now write a program that computes the root by using the bisection method which is an iterative technique for solution of non-linear equations.

In using the bisection method, we are supposed to know a range \( a \leq x \leq b \) that encloses the root (i.e. \( F(a) \times F(b) \leq 0 \)). Next, we narrow the range by half by taking the mid point \( c \). The reduced range is now either \( ac \) or \( cb \) depending on the value of \( F(c) \).

\(^8\) Solution by CMAP's built-in function: \( x_0 = \text{root1}(x, 0, \pi, \exp(-0.12*x^2)-3*\cos(x)) \);
To develop the algorithm, we will let $ab$ represent the current interval, and then keep moving either the end $a$ to $c$ or the end $b$ to $c$. The complete algorithm is as follows.

i. Given a function $F(x)$, and the initial interval $a \leq x \leq b$ such that $F(a) * F(b) \leq 0$.

ii. Compute the mid-point: $c = (a + b) / 2$

iii. If $F(c) * F(b) \leq 0$ : make $cb$ the new range. i.e. $a = c$ or else make $ac$ the new range. i.e. $b = c$

iv. If $F(c)$ is not nearly zero, repeat from step (ii)

v. Return $c$ as the root.

The algorithm is implemented in the function

$$\text{Bisect}(\text{float } a, \text{ float } b, \text{ float } \text{Err})$$

where $a$, $b$ define the initial interval, and Err is the maximum acceptable absolute error in the root.

```c
main()
{
    \text{clearplot}();
    \text{plot}(x,a = 0, b = \pi#, F(x));
    \text{Root} = \text{Bisect}(a, b, 1e-4);
    \text{print}(\text{^, Root});
}

\text{Bisect}(\text{float } a, \text{ float } b, \text{ float } \text{Err})
{ // Root by bisection method
    \text{float } fa = F(a), fb = F(b), fc, N = 0; // Note 2
    \text{if } (fa*fb > 0.0) {
        \text{notice}("Root must be bracketed for bisection");
        \text{return} 0;
    }
    \text{do } {
        c = (a+b)/2; // Mid-point
        fc = F(c);
```
if (fc*fb <= 0) { a = c; }  // Move a to c
else { b = c; fb = fc; }  // Move b to c
N = N+1;     // Counter
} while (abs(fc) > Err);   // Test for convergence
notice(" Number of iterations: ", N);
return c;

F(float x){
    return exp(-0.12*x^2)-3*cos(x);
}

Food for thought: what are the key points in this program?

Sample Program 3.6 - Graphics, Plotting

The following program plots the trajectories\(^9\) (see figure below) of a ball that is thrown with an initial velocity of \(v_o = 20\) m/s. The angle of throw, \(\alpha\), varies from 5\(^o\) to 85\(^o\) in step of 10\(^o\).

Formulation

The trajectory of the ball is the path that the ball traces in the air. Let x-y be the axis system with the origin at the initial position of the ball. i.e. The ball's initial position: \(x_o = y_o = 0\) at time \(t = 0\). At any time \(t\), the ball's position is at \(x(t)\), \(y(t)\) given by\(^10\):

\[
\begin{align*}
x &= x_o + v_{ox}t = v_{ox}t \\
y &= y_o + v_{oy}t - \frac{g t^2}{2} = v_{oy}t - \frac{g t^2}{2}
\end{align*}
\]

where

\[
\begin{align*}
v_{ox} &= v_o \cos \alpha: \text{Initial horizontal velocity} \\
v_{oy} &= v_o \sin \alpha: \text{Initial vertical velocity} \\
g &= \text{Earth gravitational acceleration} \ (9.81 \text{ m/s}^2) \\
\alpha &= \text{Angle between the initial velocity and the ground surface}
\end{align*}
\]

\(^9\) The sample program in CMAP-Help topic loadbitmap() gives an animation of projectile motion with random initial velocity and inclination.
\(^10\) Neglecting air friction and the curvature of the earth surface.
To draw the trajectory, we plot $y$ versus $x$ using the following expression:

$$\text{plot}(x, 0, xmax, y(x));$$

where $xmax$ is the maximum range of $x$ during which the ball is in the air. This air-time is found from Eq. 2 by setting $y$ to 0:

$$t_{max} = \frac{2v_{oy}}{g} \quad (5)$$

And the $x$-range is, from Eq. 1:

$$x_{max} = v_{ox}t_{max} \quad (6)$$

Since $y$ of Eq. 2 contains $t$ instead of $x$, we need to convert $x$ into $t$ using Eq. 1.

```c
main()
{
  vo = 20;       // Initial velocity
  g = 9.81;      // Gravitational acceleration;
  clearplot();   // Clean the slate
  setrange(0, 0, 45, 25); // Note 1: Range of data for plot
```

© 2015 K. H. Ha – BCEE 231 Chapter 3 - v4.3
```
setup(A, 0); // Note 2: Turn off axis-plotting
useoption("DEGREES");
Co = 1; // Pen colour (1: black, 2: green, etc..)
float A; // Alpha: angle of inclination
for(A=5; A<=85; A=A+10) // Vary the angle
{
    setup(C, Co=Co+1); // Change pen colour
    vox = vo*cos(A); // Eq. 3: Init horz. velocity
    voy = vo*sin(A); // Eq. 4: Init vertical velocity
    tmax = 2*voy/g; // Eq. 5: Air-time
    xmax = vox*tmax; // Eq. 6: Maximum x-range
    plot(x, 0, xmax, y(x)); // Note 3
}
setup(C, 7); // Light grey pen for axes
plotlinearscale('X', 9); // 9 intervals along x-axis
plotlinearscale('Y', 5); // 5 intervals along y-axis
setup(A, 1); // Turn on axis plotting
}

y(float x)
{
    float t = x/vox; // Eq. 1: Convert x to time
    return voy*t - g*t^2/2; // Eq. 2: Height
}
```

**Notes**

1. Since there are many graphs in the page, it is necessary to define a data range suitable for all graphs. If a range is not set, the first plot will automatically set a range, which may not be appropriate for the subsequent plots.

2. The subsequent `plot()` expressions will repeatedly draw the same axes x, y unless the axis-plotting option is turned off. The axes will be plotted at the end using function `plotlinearscale()`, which also draws the grid lines.
3. Different scaling ratios for x- and y-directions are automatically used in order to maximize the size of the plot. To enforce one single scaling ratio, insert the line:

    \texttt{setop(M, 0); // Turn off picture maximization}

just after \texttt{clearplot()}. And CMAP will adjust the subsequent data range to preserve true geometric proportion. The plots will be smaller.

### 3.6 Program Organization with Functions

Use of user-defined functions is effective for splitting up a complex program into simple tasks; each is carried out with a user-defined function.

#### Practice Drills 3.4

1. The following program simulates the working of an ATM which allows only one user to carry out different transactions on an account.

   The function \texttt{main()} is complete as is. Your task is to complete the other functions so that the program works sensibly.

```c
float Balance; // Account balance

main()
{
    // ATM simulator
    Choice = 0;

    Balance = abs(rand(-1))*1000;

    Amount = 0;
    while (Choice != 'Q')
    {
        // Present menu and get user's Choice
        Choice = inform(12, 42, RGB(255, 0, 128),
                       " Main Menu       ", B_Menu(" Quit     ", 'Q'), ^
                       " Account balance: ", $Balance, ^
                       " Transaction amount = ", Amount, ^
                       " ", B_Menu(" Withdraw ", 'W') ^
                       " ", B_Menu(" Deposit ", 'D') ^
                       " ", B_Menu(" Payment ", 'P'));
```
2. Modify the preceding program to process multiple users.

3.7 Application: Design of a Simple User-Interface

We would want our programs to have a menu system for users to define data, revise data, save and open data files as well as to do computation (see Figure).

As shown in the following flow-chart, a menu-driven program has these actions:

(a) Wait until the user makes a choice.
(b) If the choice is “Quit” then quit.
(c) Or else process the choice.
(d) Back to step (a).

The program uses the while-loop to cycle through the actions:

(i) presenting the menu using function inform()
(ii) processing the choice using the switch-statement.

### Conceptual flowchart of menu-driven programs:

- **Choice = 0**
- **Pick a Choice**
- **Process Choice**
- **Choice == “Quit”?**
  - Yes: **End**
  - No: **Choice == “Quit”?**

### Menu produced by inform():

Function inform() displays a window containing a menu of 6 choices: “Define Data”, “Revise Data”, “Analyse”, “Save Data”, “Read DataFile”, and “Quit”. Function inform() waits until the user selects a menu item, and then it closes the window and returns a value indicating the clicked button. This value is used in the switch command for selecting the appropriate branch for action.

```c
float Choice;  // Menu selection to be returned by inform()
main()
{
    float Choice = 0;
    while (Choice != 'Q') // See Note #1 below
    {
        // Present menu and get user's Choice
        Choice = inform(7, 35, RGB(255, 0, 128),
                         "     Main Menu " ^^
                         B_Menu(" Define Data " ', 'D') '', // Note # 2
                         B_Menu(" Revise Data " ', 'V') ^^
```
B_Menu(" Analyse      ", 'A'), "     ",
B_Menu(" Save Data    ", 'S') ^^
B_Menu(" Read DataFile", 'R'), "     ",
B_Menu(" Quit         ", 'Q'));

switch(Choice)
{ // Process the requested Choice
  case 'D': DefineData(); // Define data
    break; // Skip the rest
  case 'V': view();
    break; // View all data
  case 'A': Analyse(); // Compute
    break;
  case 'S': saveglobalvars();
    break; // Save data
  case 'R': readglobalvars(); // Read data
}
// end switch
// end while
// end main()

DefineData()
{ // Data from Sample Program 1.1
  // Units: N (forces), m (lengths), degree (angles)
  view(L, P, b, h, Al); // Data input at run-time
}

Analyse()
{ // Do your actual computation here. Sample Program 1.1
  useoption("DEGREES");
  My = P*cos(Al)*L; Mz = -P*sin(Al)*L;
  Iy = h*b^3/12; Iz = b*h^3/12;
  Bet = atan(Iz*tan(Al+90)/Iy);
  Smax = My*(b/2)/Iy - Mz*(h/2)/Iz;
  cat();
}

Notes
1. The while-loop executes as long as Choice is not 'Q'uit. The ASCII code of 'Q' is 81. Function inform() returns 'Q' or 81 when the user
clicks the Quit-button in the menu. This returned value is assigned to Choice as shown in the next line.

2. The line B_Menu(" Define Data ", 'D') displays the menu-button “Define Data”. When the user clicks this menu item, inform() immediately closes the window, and returns the ASCII code of 'D'. See Help-documentation for complete information on inform().

3.8 Useful Built-in Functions

The following built-in functions have been introduced in this Chapter. Complete information on them is available in CMAP-Help.

deriv(...) Returns first derivative of an expression
inform(...) User-friendly form interface for input, output, menu
integ(...) Returns the definite integral of an expression
plot(...) Plots an expression.
maxz1(...) Returns value which maximizes an expression
minz1(...) Returns value which minimizes an expression
root1(...) Returns root of an expression having one unknown
setop(...) Sets a graphic option such as pen color, font size, etc..
setrange(...) Sets the extent of the data range that defines the plot area
trace() Turns ON/OFF trace of function execution
4.1 Array Variables: One Name for Many Storage Boxes

An array variable has many memory storage boxes associated with that one name. For one-dimensional arrays, the storage boxes are arranged in a long string (Fig. 4.1). For two-dimensional arrays, we may think of a tabular form arrangement as shown in Fig. 4.2.

![Figure 4.1: Examples of 1-D arrays or vectors](image)

1. One-Dimensional Arrays

Figure 4.1 shows 1-D arrays F1 and F2 where each name has three values or boxes.

It's up to us to put whatever values in these boxes (or elements). For instance, we may put into array F1 the three components of a force vector F1 in the sequence shown. The computer is, of course, not aware of the meaning of these values nor of their units.

Creation

There are many built-in functions for creating arrays in CMAP. The function `defmat()` is convenient when the elements' values are known (as the case with arrays F1, F2):

```
defmat(F1[3], -12.5, 23.2, 16.4); // Size = 3
defmat(F2[3], 5.6, -4.7, -8.2);
```

---

1 The tabular form is just for human to see. Actual memory storage boxes for an array are still in one long string, one row after another. Fortran allocates storage column by column.
Note that the sequence of the elements’ values follows the desired order.

Thus, to create a new array, we have to specify:

- A name that serves as the label for the reserved memory storage.

- The array size so that sufficient consecutive storage boxes will be reserved. CMAP reserves 8 bytes for each box (enough for 15 digits). The array size is enclosed within square brackets following the name.

Summing Array Elements

Using proper terminology, we say: the array F1 has three elements, and similarly, the array F2 also has three elements. It is common to refer to 1-Dimensional arrays as vectors irrespective of the number of elements.

A specific element in a vector is indicated by its position index [enclosed in square brackets as shown in Fig. 4.1]. For example, the sum of the three elements of the array F1 may be found by the following expression:

\[
\]

In general, it is far better to use a loop to sum as many elements as needed. The strategy is to scan every element of the array and add it into an accumulator variable. The advantage of the strategy is that it works for arrays of any size.

```c
main()
{
    // Summing array elements
    float I;    // Loop index
    defmat(F1[N=3], -12.5, 23.2, 16.4);
    Sum = 0;    // Will be the Sum
    for(I=1; I<= N; I=I+1)
    {
        Sum = Sum + F1[I]; // Accumulate values
    }
    print(^" Sum = ", Sum);
}
```
2. Two-Dimensional Arrays

Fig. 4.2 shows a 2-D array variable XYZ. Its storage boxes are arranged into a table having 7 rows and 3 columns. A specific element, row, or column can be accessed by its position enclosed within, respectively [ ], ( ), { } as shown below.

![Example of 2-D arrays, tables or matrices](image)

**Figure 4.2: Example of 2-D arrays, tables or matrices**

**Creation**

Similar to the case of 1-D arrays, if the elements' values are known, we may use the function `defmat()`, which will:

- Allocate sufficient storage for the size given in square brackets
- Label it with the given name
- Place values into the elements, row by row

```cpp
// Define array elements, row by row:
defmat(XYZ[7,3], -12.5, 23.2, 16.4,
      5.6, -4.7, -8.2, 1.4, 3.6, 4.6,
      -2.5, 2.32, 3.64, 15.2, -4.23, 3.16,
      5.5, 8.42, -7.42, -9.5, -6.3, -8.4);
```

Note the format for specifying the size: the number of rows is specified first, followed by a comma, and then the number of columns. Both numbers are in square brackets.
Summing Elements

An element of a 2-D array is identified by its row and column position indexes in square brackets\(^2\) (Fig. 4.2):

\[
\text{XYZ[RowIndex, ColIndex]}
\]

For illustration, let's sum all elements of the array XYZ using two nested loops as follows:

```
main()
{ // Summing array elements
    float I, J;         // Loop indexes
    // Define array elements, row by row:
    defmat(XYZ[M=7, N=3], -12.5, 23.2, 16.4,
           5.6, -4.7, -8.2, 1.4, 3.6, 4.6,
           -2.5, 2.32, 3.64, 15.2, -4.23, 3.16,
           5.5, 8.42, -7.42, -9.5, -6.3, -8.4);
    Sum = 0;       // Initialize Sum
    for(I=1; I<= M; I=I+1)   // For each row
    {
        for(J=1; J<= N; J=J+1)  // For each column
        {
            Sum = Sum + XYZ[I,J];
        }
    }
    print(" Sum = ", Sum);
}
```

Use of Arrays

Again, it is up to us to put values into the boxes and to make use of them. For example, we may store in arrays such as XYZ the following types of information:

\(^2\) Standard C-syntax requires this more complex format \text{XYZ[RowIndex][ColIndex]}. Syntax of CMAP's arrays is more similar to Fortran arrays.
(a) The components of 7 forces:
   - x-, y-, z-components of force No. 1 into row No. 1
   - x-, y-, z-components of force No. n into row No. n

(b) The coordinates of 7 points:
   - x-, y-, z-coordinates of point No. 1 into row No. 1
   - x-, y-, z-coordinates of point No. n into row No. n

Such organization of data also gives a clear interpretation of the columns of the array; e.g. column 1 contains the x-components (or x-coordinates) of all forces (or points). This interpretation is up to us.

4.2 Working with Array Elements using Loops

By the convention that A[K] is the K\textsuperscript{th}-element of a one-dimensional array A, we can access different elements of array A by varying the index variable K. The following examples show some standard use of loops for working with arrays.

Sample Program 4.1 - Statistics of data in a 1-D Array

Given the following one-dimensional array A of 10 elements:

\[
A = \{ 65, 42, 76, 89, 94, 78, 34, 66, 48, 52 \}
\]

write a program to compute the mean value as well as the standard deviation of the elements of A using the following formulas:

(a) The mean value:

\[
\mu = \frac{1}{N} \left( A_1 + A_2 + \cdots + A_N \right)
\]

\[
= \frac{1}{N} \sum_{k=1,2,3\ldots}^{N} A_k
\]

(b) The standard deviation:

\[
\sigma = \left[ \frac{1}{N} \sum_{k=1}^{N} (A_k - \mu)^2 \right]^{1/2} = \left[ \frac{1}{N} \left( \sum_{k=1}^{N} A_k^2 \right) - \mu^2 \right]^{1/2}
\]
main()
{
    defmat(A[N=10], 65, 42, 76, 89, 94, 78, 34, 66, 48, 52);
    float K; // Loop index
    Sum = 0; // Will be the sum of all elements
    Sum2 = 0; // Will be the sum of the squares
    for(K=1; K<= N; K=K+1)
    {
        Sum = Sum + A[K];
        Sum2 = Sum2 + A[K]^2;
    }
    Mean = Sum/N;
    StDev = sqrt(Sum2/N - Mean^2);
    print(" Mean = ", Mean,
         " Standard deviation = ", StDev);
}

Sample Program 4.2 - Adding Matrices

Write a program to add the following two arrays A and B:

\[
\begin{bmatrix}
2 & 1 & -3 \\
4 & 0 & 7 \\
\end{bmatrix} + 
\begin{bmatrix}
5 & 8 & 0 \\
-9 & 2 & 5 \\
\end{bmatrix}
\]

Solution

The sum of the arrays A and B is an array C defined by:

\[ C_{ij} = A_{ij} + B_{ij} \]

i.e.

\[
\begin{bmatrix}
2+5 & 1+8 & -3+0 \\
4-9 & 0+2 & 7+5 \\
\end{bmatrix} = 
\begin{bmatrix}
7 & 9 & -3 \\
-5 & 2 & 12 \\
\end{bmatrix}
\]

main()
{
    // Array addition
}
M = 2; N = 3; // Number of rows and columns
defmat(A[M,N],2,1,-3, 4,0,7);
defmat(B[M,N],5,8, 0, -9,2,5);
float i,j; // Row & column indexes
zero(C[M,N]); // Storage for C: Note 1
for(i=1; i<=M; i=i+1) // Each row i=1,2,..
{
    for(j=1; j<=N; j=j+1) // Each column j
    {
        C[i,j] = A[i,j] + B[i,j];
    }
}
print(A, B, C);

Note 1: zero() is a built-in function that creates arrays and initializes the array elements to zero. This line creates array C of 10 zero-elements, serving as the container for the result. Since array C is new, it is automatically created as a global array.

Practice Drills 4.1

1. Let arrays A and B be defined as follows:

defmat(A[N = 10],-12.5, 23.2, 16.4,  5.6, -4.7, -8.2,  
     1.4, 3.6, 4.6, -2.5);
defmat(B[N], 3.64, 15.2, -4.23, 3.16, 5.5, 8.42, -7.42,  
     -9.5, -6.3, -8.4);

Write a program that uses one loop to compute all the following expressions.

(a) \[ S = A_1B_1 + A_2B_2 + \cdots + A_NB_N = \sum_{i=1}^{N} A_iB_i \]
(b) \[ E = \sqrt{(A_1 B_1)^2 + (A_2 B_2)^2 + \cdots + (A_N B_N)^2} = \sqrt{\sum_{i=1}^{N} (A_i B_i)^2} \]

(c) \[ P = (A_1 + A_2 + \cdots + A_N)(B_1 + B_2 + \cdots + B_N) = \left( \sum_{i=1}^{N} A_i \right) \left( \sum_{i=1}^{N} B_i \right) \]

(d) \[ Q = (A_1^2 + A_2^2 + \cdots + A_N^2)(B_1^2 + B_2^2 + \cdots + B_N^2) = \left( \sum_{i=1}^{N} A_i^2 \right) \left( \sum_{i=1}^{N} B_i^2 \right) \]

2. Let the 1-D array \(A\) be defined by:

\[
\text{defmat}(A[N = 10],-12.5, 23.2, 16.4, 5.6, -4.7, -8.2, 1.4, 3.6, 4.6, -2.5);
\]

Use a loop in the following programs. Note: You may wish to write one single program for computing all the desired quantities.

(a) Write a program to compute and print the negative elements (of the array \(A\)), and their column positions.
Strategy: Scan every element of \(A\), and if it is negative, print the value and its index position.

(b) Write a program to compute and print the largest element of the array \(A\).
Strategy: Set \(\text{Max} = A[1]\). Scan every element of \(A\), and if it is greater than \(\text{Max}\), save the value into \(\text{Max}\).

(c) Write a program to compute and print the smallest element of the array \(A\).
Strategy: Set \(\text{Min} = 1.\text{e}60\). Compare \(\text{Min}\) to every element of \(A\), and save the smaller value into \(\text{Min}\).
(d) Write a program to compute and print the smallest positive element of the array A.
Strategy: Set Min = 1.e60. Compare Min to every positive element of A, and save the smaller value into Min.

3. Let the 2-D array XYZ be defined by:

```c
// Define array elements, row by row:
defmat(XYZ[M=7, N=3],-12.5, 23.2, 16.4, 5.6, -4.7, -8.2,
   1.4, 3.6, 4.6, -2.5, 2.32, 3.64,
   15.2, -4.23, 3.16, 5.5, 8.42, -7.42,
   -9.5, -6.3, -8.4);
```

Use two nested loops for the following programs. Note: You may wish to write one single program for computing all the desired quantities.

(a) Write a program to compute and print the negative elements (of the array XYZ), and their row, column positions.

(b) Write a program to compute and print the largest element of the array XYZ.

(c) Write a program to compute and print the smallest element of the array XYZ.

(d) Write a program to compute and print the smallest positive element of the array XYZ.

4. Let the 2-D array XYZ be defined as given in the preceding problem. Write a program to compute and print the average of each row and column of the array XYZ.

### 4.3 Creation of Array Variables

In CMAP, the scope of a variable (both `float` & `mat`) can be either global or local:

- Global: by a declaration placed outside of functions or (in CMAP) by default.

---

3 C-language has more advanced types of scope: http://en.cppreference.com/w/c/language/scope
4 Since the entire CMAP-application program resides within one file, file-scope is truly global.
• Local\textsuperscript{5} to a particular function: only by explicit declaration within that specific function.

Declaring or creation an array or matrix variable means:

• Allocation of storage sufficient for the given size (8 bytes per element).
• Labeling the storage by a name.

1. Declaring Arrays as Global Variables

Global mat-variables are available to all functions but are superseded by \texttt{local} mat-variables of the same names.

Global variables by default\textsuperscript{6}

In CMAP-application program, a NEW (not-previous declared) matrix variable is automatically created \texttt{global} when it is being defined or created by an assignment matrix expression or by built-in functions such as defmat(), zero() or view().

Explicit declaration of global variables\textsuperscript{7}

Global variables may be created and zero-initialized by declarations placed \texttt{OUTSIDE of ALL functions} at the top or before the first function that uses them.

The declaration syntax requires the \texttt{mat}-keyword, name and size:

\begin{verbatim}
mat F[6,3], // Matrix F of 6 rows by 3 columns
    P[4], // Vector P of 4 elements
    Q[1,4], // Vector Q of 4 elements
    R[4,1]; // Vector R of 4 elements
main(){
    DefineMatrices();
    }

DefineMatrices(){
    view(F, P, Q, R); // Edit values of elements
}
\end{verbatim}

\textsuperscript{5} Function-scope in C-language.
\textsuperscript{6} This option can be turned off using the Set Options menu.
\textsuperscript{7} In standard C/C++, every variable must be declared in advance.
Note that the declarations of vectors P, Q, R use three different but equivalent forms. In CMAP, the declaration also initializes all array elements to zero. Such initialization takes place only ONCE when the program is FIRST executed by clicking the eXecute Icon\(^8\).

**Motivation**

Once created, global variables can be used anywhere in the program as well as in the Calc Expressions window. This makes it easy for functions to share data.

**Scope**

Global variables created in one function may be used and redefined in any other functions (unless the name conflicts with a local variable).

**Life**

In CMAP environment, global variables remain available for use during execution and after program termination.

### 2. Declaring Arrays as Local Variables

When a function needs to store temporary data into arrays, it can create local arrays for this purpose. The only way to create an array variable local to a function is to declare it within that function’s body by the type-specifier keyword `mat` followed by the array name and its size\(^9\).

```cpp
main()
{
    Fun();
}

Fun()
{
    mat F1[3], F2[3]; // 1-D arrays local to Fun()
    mat XYZ[7,3];    // 2-D array local to Fun()
    view();          // View and edit variables
}
```

\(^8\) This means that declarations (of global vars) added into a program are not activated until the program is executed again. We can still add the declarations into the program, and define the values in the Calc-Expressions Dialog.

\(^9\) Storage allocation is done for the given size (in CMAP: 8 bytes per element).
These declarations allocate arrays' storage and initializes their elements to zero\(^\text{10}\). Once created, the arrays are available for storing new data.

Local variables are recognized only within the function that declares them, and they supersede global variables of the same names.

3. Releasing Storage of Local Array Variables

Since local variables exist only during function execution, storage allocated to local arrays will be released\(^\text{11}\). The release of the storage of local arrays is, of course, done automatically as the function exits (normally or abnormally).

4. Defining Matrices or Arrays by Built-in Functions

CMAP built-in functions\(^\text{12}\) [e.g. `view()`, `zero()`, `defmat()`, `zeronew()`, `defsym()`] and matrix operators (Section 4.6) can create array or matrix variables.

Rules: The following rules apply:

1. New name: If the specified name did not exist, it will be created as a global variable by allocating sufficient storage for the given size.

2. Existing Name: If the name already exists as an array, the old array will be replaced by the new array having the current scope (either local or global).

   If the matrix size is omitted, the current size will be used.

3. Type Conversion: If the name already exists as a float-variable, `zero()` and `defmat()` will issue an error message, while `view()` will convert the name to mat-type and also delete the previous float-variable.

Returned Value: See CMAP documentation for individual functions.

---

\(^{10}\) Standard C does not ask for arrays to be initialized after allocation.

\(^{11}\) It would be a tremendous waste of memory storage if local arrays stay permanent after being used once.

\(^{12}\) Standard C library does not have functions for dealing with arrays or matrices.
The preceding rules can be observed in the following program.

```c
main()
{
    defmat(ZZ[2], 1, 2); // Rule 1: ZZ is new global array
    defmat(ZZ[3], 1, 2, 3); // Rule 2: New ZZ replaces old ZZ
    UU = VV = 1; // float variables (Global)
    view(UU[3]); // Rule 3: view() converts float
    zero(VV[3]); // Rule 3: zero() can't convert
}
```

5. Resizing Matrices at Runtime

Scenario: Consider the following situation that arises frequently in application programming:

- The user inputs, at runtime, an array F of 4 rows, 3 columns to accommodate 4 forces
- Later on, the user wants to add two more forces to the table.

The following program allows two more forces to be added without destroying the existing four forces:

```c
main()
{
    view(F[4,3]); // Define table F of 4 by 3
    print(F); // Example of using F
    resizemat(F[6,3]); // Expand size to 6 by 3
    view(F); // Let user define the new forces
    print(F); // Example of using F
}
```

In the above program, the built-in function `resizemat()` automatically:

- allocates a new table having size 6 by 3,
- copies the existing data of F into the new table
• gives it the same name \( F \)
• and finally deletes the memory allocated to the initial \( F \).

These steps must be performed separately in conventional C/C++ programming.

4.4 Array Arguments and Array Parameters (Sample Program 4.3)

1. Declaring Array Parameters

Suppose we need a function for adding two vectors \( \mathbf{A}_{1 \times N}, \mathbf{B}_{1 \times N} \) into an existing vector \( \mathbf{C}_{1 \times N} \): \( \mathbf{C} \leftarrow \mathbf{C} + \mathbf{A} + \mathbf{B} \)

The function should have three \texttt{mat}-type parameters: \( \mathbf{A}, \mathbf{B} \) and \( \mathbf{C} \). These parameters must be declared \texttt{mat}-type in the function’s heading as shown below. The names \( \mathbf{A}, \mathbf{B}, \mathbf{C} \) are local names for temporary use within the function body. They supersede global variables of same names.

```c
// Use of arrays or matrices as function parameters
AddArrays(mat A, mat B, mat C, float N)
{
    float I; // local var for loop index
    for(I=1; I<= N; I=I+1)
    {
        // Accumulate A & B into C
    }
}
```

2. Passing Array to Functions

When the function AddArrays() is invoked, array-\texttt{names} must be used as arguments to match with the respective array parameters.

For example, suppose we need to compute the sum of 4 vectors \( \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4 \) whose elements are defined below:
main()
{
    // Sample Program 4.3
    defmat(F1[N=3], -12.5, 23.2, 16.4); // Define F1
    defmat(F2[N], 5.6, -4.7, -8.2); // Define F2
    defmat(F3[N], -0.6, 2.3, 4.5); // Define F3
    defmat(F4[N], 4.3, -0.8, 2.3); // Define F4

    // Sum of 4 vectors R = F1 + F2 + F3 + F4
    zero(R[N]); // Storage for R
    AddArrays(F1, F2, R, N); // R ← R + F1+F2  Note 1
    print(R);
    AddArrays(F3, F4, R, N); // R ← R + F3+F4  Note 2
    print(R);
}

Notes:

1. Array names F1, F2, R are passed to AddArrays(). Within the function AddArrays(), they take the new names 'A', 'B' and 'C', respectively. 'A' refers to the storage of F1; 'B' to F2; and 'C' to R.

2. Array names F3, F4, R are passed to AddArrays(). Within the function AddArrays(), they take the new names 'A', 'B' and 'C', respectively. 'A' refers to the storage of F3; 'B' to F4; and 'C' to R.

3. Memory Allocation for Function Array Parameters

An important question arises: Will array parameters have their own allocated memory for temporary storage of array arguments?

The answer is that no additional memory is allocated for array parameters. They use the same storage of the array arguments--but with their own names temporarily within the function.

This avoids waste of memory and waste of time copying values from array arguments to array parameters.

4. Rules on Array Arguments and Parameters

Here are the rules with respect to array arguments:
1. The function parameter names (such as A, B, C, N) are local, i.e. valid only within the declaring functions. These names are arbitrary, and they supersede global variables of the same names. This rule applies to float-parameters as well.

2. When the function is invoked, its array parameters (e.g. A, B, C) claim the storage of the respective array arguments passed by the calling expression (e.g. F1, F2, F3, F4, R). As a consequence, the function can change the contents of the originally passed array arguments.

3. Since the storage of an array parameter is the storage of the corresponding array argument, this storage will NOT be released when the function terminates.

4.5 Low-Level Array Programming

In conventional programming with arrays, individual elements are operated on as shown in the preceding sections. Programming with individual elements is low-level programming\(^{13}\). Additional examples are given in this section.

1. One-dimensional Arrays

Sample Program 4.4 - Array Elements for float-Arguments

The following program:

- Defines an array R containing 10 values for the radii of 10 circles
- Computes the total area of all 10 circles
- Computes the total circumference of all 10 circles.

The array is used merely as a convenient storage for a series of data of the same nature. Observe how individual elements of the array R are passed as float-argument to functions.

- Run the following program and explain its output.
- What is the purpose of the for-statement?

\(^{13}\) While being lower than CMAP's high-level programming, it is still very high compared to the truly low level in assembly language programming.
main()
{
    trace(1);    // Note #1
    defmat(R[N=10], 2.5, 2.1, 4.2, 2.5, 5.4, 12.5, 4.3, 4.9, 7.2, 0.56);
    TotalArea = 0;  // Initialization & default global
    TotalLength = 0;
    float I;       // Local for loop index
    for (I=1; I<= N; I=I+1)
    {
        TotalArea  = TotalArea + Area(R[I]);   // Note 2
        TotalLength = TotalLength + Len(R[I]); // Note 2
        print(^ " I = ", I, " Partial area = ", TotalArea ,
               " Partial length = ", TotalLength);
    }
    print(^ " Area = ", TotalArea , " Length = ",TotalLength);
    cat();
}

Len(float r) {
    return 2*pi*r;   // Return circumference
}

Area(float r) {
    return pi*r*r;   // Return area
}

Notes
1. Function trace() displays the sequence of function calls and returns.
2. The array element R[I] is passed as float-argument.

Sample Program 4.5 - Polynomial and its Efficient Evaluation

A polynomial of degree (n-1) contains n terms:

\[ P_n(x) = c_1 + c_2x + c_3x^2 + c_4x^3 + \cdots + c_nx^{n-1} \]
The coefficients $c$'s may be stored in a 1-D array as follows:\(^\text{14}\):

$$C_{1xn} = \{c_1, c_2, c_3, c_4, \ldots, c_n\}$$

Our objective is to write the function `Poly(float x, float n, mat C)` that returns the value of the polynomial $P_n(x)$ for a given $x$.

The best way of evaluating polynomials is by Horner's rule as explained below.

<table>
<thead>
<tr>
<th>Polynomial $P_n(x)$</th>
<th>Horner's rule of evaluation $P_n(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_3(x) = c_1 + c_2 x + c_3 x^2$</td>
<td>$P_3(x) = c_1 + x(c_2 + c_3 x)$</td>
</tr>
<tr>
<td>$P_4(x) = c_1 + c_2 x + c_3 x^2 + c_4 x^3$</td>
<td>$P_4(x) = c_1 + x(c_2 + x(c_3 + c_4 x))$</td>
</tr>
<tr>
<td>$P_5(x) = c_1 + c_2 x + c_3 x^2 + c_4 x^3 + c_5 x^4$</td>
<td>$P_5(x) = c_1 + x(c_2 + x(c_3 + x(c_4 + c_5 x)))$</td>
</tr>
<tr>
<td>$P_n(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_{n-1} x^{n-2} + c_n x^{n-1}$</td>
<td>$P_n(x) = c_1 + x(c_2 + x(c_3 + \cdots + x(c_{n-1} + c_n x)\cdots))$</td>
</tr>
</tbody>
</table>

Horner's rule of polynomial evaluation is most efficient because it uses only addition and multiplication operators (i.e. no power operator).

To implement Horner's rule, we start with the $n$\textsuperscript{th}-term as shown in the figure. The pseudo code to accumulate all terms into $S$ is:

\(^\text{14}\) This is also the storage format of CMAP's built-in functions (for polynomials) such as `poly()`, `polycoe()`.

\(\text{poly}(x, \text{float } n)\)
(a) \( S \leftarrow C[n] \quad // \text{S will store the sum} \)

(b) for i = n-1 down to 1 in step of 1:
\[
S \leftarrow C[i] + x S
\]

```c
main()
{
    defmat(P[n=4], -2, 0, 1.5, 0.4);
    plot(x, -2, 3, Poly(x, n, P));
}

Poly(float x, float n, mat C)
{
    float i, S = C[n];
    for(i = n-1; i>= 1; i=i-1)
    {
        S = C[i] + x*S;
    }
    return S;
}
```

**Practice Drills 4.2**

1. Write the function Quad(mat A, mat X) that computes the roots of the quadradic equation \( A_1 + A_2 x + A_3 x^2 = 0 \) and store the roots in the array X.

   Write the function main() that (i) allows the user to input different sets of coefficients A’s and (ii) computes and prints the solutions of the quadradic equations.

2. Write a program that:
   - Defines the following data:

     ```c
     defmat(Marks[nStudents=20], 85, 93, 41, 67, 72, 54, 85, 93, 61, 67, 55, 43, 76, 63, 94, 83, 85, 73, 61, 74);
     ```
- Calls the user-defined function Grade(float M) for each student to determine and output a letter grade corresponding to the marks M in accordance to the following scheme:

<table>
<thead>
<tr>
<th>Marks, M</th>
<th>Letter grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 49</td>
<td>FNS</td>
</tr>
<tr>
<td>50 – 59</td>
<td>D</td>
</tr>
<tr>
<td>60 – 69</td>
<td>C</td>
</tr>
<tr>
<td>70 – 79</td>
<td>B</td>
</tr>
<tr>
<td>80 - 100</td>
<td>A</td>
</tr>
</tbody>
</table>

- Computes and prints the average marks.

3. Write the function MaxElement(mat A, float N) that returns the (algebraic) maximum element of the 1-D array A of size N. Write function main() to test the function MaxElement() with some suitable data.

4. Write the function MinElement(mat A, float N) that returns the smallest positive element of the 1-D array A of size N. Write function main() to test the function MinElement() with the following data.

```c
#define mat (A[N=10], -15, -23, 41, 67, 72, 93, 61, -67, 55, -43);
```

5. Write the function Stats(mat A, float N) that computes and prints the mean value as well as the standard deviation of the elements of the one-dimensional A of size N. Write function main() to test the function Stats() with the following data:

```c
#define mat (Marks[10], 15, 23, 41, 67, 72, 93, 61, 67, 55, 43);
#define mat (Score[5], 45, 32, 46, 76, 87);
```

2. Two-dimensional Arrays

Sample Program 4.6 - Copy Elements of 2-D Arrays to Vectors

In the following program, a 2-D array XYZ is used for storing data in tabular format (i.e. rows & columns). The following program shows how to copy all elements in row 5 of array XYZ into a vector V.
main()
{
    // Define array elements, row by row:
    defmat(XYZ[7,3],-12.5, 23.2, 16.4, 5.6, -4.7, -8.2,
            1.4, 3.6, 4.6,    -2.5, 2.32, 3.64,
            15.2, -4.23, 3.16, 5.5, 8.42, -7.42,
            -9.5, -6.3, -8.4);
    zero(V[3]);  // Create storage for V
    float J;
    for(J=1; J<=3; J=J+1) // Column J = 1,2,3
    {
        V[J] = XYZ[5,J];  // Copy element at row 5, col J
    }
    print(XYZ,V);
    cat();
}

Practice Drills 4.3

(a) Write the function AddRows(mat A, float M, float N, mat S) that adds rows M and N of array A and store the result in array S.

Hint: Makes use of the built-in functions nrow(), ncol() to get the size of an array.

(b) Write the function AddColumns(mat A, float M, float N, mat S) that adds columns M and N of array A and store the result in array S.

(c) Write the main function that

- Defines the 2-D array XYZ as given previously
- Invokes AddRows() to add row 2 and row 5 of XYZ
- Invokes AddColumns to add column 2 and column 3 of XYZ
- Prints the results.
Sample Program 4.7 - Statistics of Data in Rows, Columns of Arrays

The following program computes the average values of individual rows and columns of the array XYZ. The results are stored in vectors RowAve and ColAve.

The program uses nested for-loops to scan all elements of the array. This is a fairly standard technique.

```c
main()
{
    // Define array elements, row by row:
    defmat(XYZ[7,3], -12.5, 23.2, 16.4, 5.6, -4.7, -8.2,
    1.4, 3.6, 4.6, -2.5, 2.32, 3.64,
    15.2, -4.23, 3.16, 5.5, 8.42, -7.42,
    -9.5, -6.3, -8.4);

    zero(RowAve[7], // Create storage for rows' averages
    ColAve[3]);  // Storage for columns' averages
    float I, J;  // Local vars for loop indexes
    float Sum;   // Local var for intermediate data
    for(I=1; I<=7; I=I+1)   // Row I=1,2,..7
    {
        // Average of 3 elements in row I
        Sum = 0;  // Initialized for each new row
        for(J=1; J<=3; J=J+1) // Column J=1,2,3
        {
            Sum = Sum + XYZ[I,J]; // Row I, column J
        }
        RowAve[I] = Sum/3;
    }
    for(J=1; J<=3; J=J+1)   // Column J=1,2,3
    {
        // Average of 7 elements in column J
        Sum = 0;  // Initialized for each new column
        for(I=1; I<=7; I=I+1) // Row I=1,2,3,4,5,6,7
        {
```
Sum = Sum + XYZ[I,J];
}

ColAve[J] = Sum/7;
}

print(XYZ, RowAve, ColAve);
cat();

Practice Drills 4.4

1. Reorganize the above Sample Program into 4 functions that have NO function arguments:
   - main();
   - GetData();
   - GetRowAverages();
   - GetColumnAverages();

2. Reorganize the above Sample Program into 4 functions with appropriate function arguments/parameters such that ALL variables are local variables.
   - main();
   - GetData(......);
   - GetRowAverages(......);
   - GetColumnAverages(......);

4.6 High-Level Programming with CMAP Arrays

Low-level manipulation of individual elements of arrays as shown in the preceding section is completely general, but tedious and difficult in complex problems.

CMAP has built-in functions and operators that work with an entire row, column, or array, and thus, substantially reducing the effort in using arrays\textsuperscript{15} and matrices. In addition, CMAP provides automatic memory management and error handling, which are complex issues in low-level C-programming.

\textsuperscript{15} This is the original raison d’être of CMAP (C-based Matrix Application Package).
1. Expressions of Vectors and Matrices

Scenario: Consider the problem of adding a series of arrays of equal size, where each array is scaled by a factor $c$:

$$\mathbf{R} = c_1 \mathbf{F}_1 + c_2 \mathbf{F}_2 + c_3 \mathbf{F}_3$$

Example of application: As a special case where $c_1 = c_2 = c_3 = 1$, the array $\mathbf{R}$ may represent the resultant vector of three forces.

Solution: We can solve this problem in one matrix/vector expression\(^{16}\):

$$\mathbf{R} = c_1 \mathbf{F}_1 + c_2 \mathbf{F}_2 + c_3 \mathbf{F}_3;$$

Note: Operations for scaling, addition and subtraction of arrays or matrices are performed element-by-element for all elements. And they work for both 1-D and 2-D arrays of any order.

Matrix Expression: The exclamation character '!' in front of the expression forces CMAP to use matrix/vector/array operations for computation. The result of such expression is always a matrix or an array.

CMAP expressions for arrays or matrices have the conventional mathematical form:

$$\mathbf{R} = (\mathbf{F} = 2.5 \mathbf{F}_1 + 4 \mathbf{F}_2) / 3.4 - 2(6 \mathbf{F}_4 - 2 \mathbf{F}_3);$$

There are a large number of built-in functions and operators dealing with arrays, vectors and matrices. In this course, we'll use only a few of them.

2. Array Operators

Vector and matrix expressions perform computation according to the rules of vector and matrix algebra. Array operations are useful for processing of tabular data, but they are not part of vector and matrix algebra.

The following program demonstrates the use of the array operators $\&*$ and $\&/$ on the rows and columns of tabular data.

```c
main()
{
```

\(^{16}\) CMAP array and matrix expressions work for both 1-D and 2-D arrays.
defmat( A[5,6], 1,2,4:0, 3,4,4:0, 5,6,*); view(A);

// Multiply columns 1 and 2, and place result into column 3

// Divide row 2 by row 1, add row 3, and place result into row 4

3. Memory Management

Scenario: To illustrate the issue of memory management, consider the following situation that arises frequently:

- Define an array F of 4 rows, 3 columns to accommodate 4 forces
- Later on, the array F need be expanded to accommodate 6 forces.

The following program does just that:

```c
main()
{
    view(F[4,3]); // Allocate table of 4 by 3
    print(F);    // Example of using F
    view(F[6,3]); // Allocate table of 6 by 3
    print(F);    // Example of using F
}
```

Before allocating the second table, CMAP de-allocates the storage of the first table, or else the system eventually runs out of memory resource, and it will hang.

CMAP provides automatic memory management\(^\text{17}\) for deletion, allocation, expansion, and contraction of the actual storage as well as of all intermediate data storage required during the evaluation of matrix expressions.

\(^\text{17}\) In low-level programming, the program must take care of memory management.

© 2015 by K. Ha - BCEE 231 Chapter 4 - v4.
4. Passing Array Arguments

In low-level programming, use of array arguments often needs passing both the array names as well as their sizes. In CMAP high-level programming using matrix expressions, only the array names need to be passed.

Two functions are available for getting the array size: \texttt{nrow()} and \texttt{ncol()}, but it's rare that we need them.

5. Handling Errors

Computer software is plagued with numerous potential errors that may cause program/system crash. The following shows the two most troublesome and frequent errors associated with the use of arrays.

"Buffer overflow"\textsuperscript{19}: use of un-allocated memory.

- An array F of 3 elements was allocated:
  
  \begin{verbatim}
  zero(F[3]);
  \end{verbatim}

- Later on, a value is assigned to the fourth element of F:
  
  \begin{verbatim}
  F[4] = 1;
  \end{verbatim}

Since only 3 boxes were allocated for array F, placing a value into the fourth box would destroy something in the computer memory. This "something" might be a piece of the program, or of another program, or the Windows system software itself, thus causing system crash.

"Memory leak"\textsuperscript{20} occurs when allocated memory is not released, thus depleting available memory. Consider the sequence of function calls: main() \Rightarrow fA() \Rightarrow fB() \Rightarrow fC() \Rightarrow fD() crashes. When the program crashes in fD(), any local & global matrices (allocated along the way) if not released will deplete available memory.

Solution: CMAP automatically prevents these errors and a host of others.

\textsuperscript{18} Unlike standard low-level programming, CMAP actually remembers the size of all arrays so that it can perform validity check on element indexes. This also explains why CMAP's built-in functions never need arguments for array size.

\textsuperscript{19} \url{https://en.wikipedia.org/wiki/Buffer_overflow}

\textsuperscript{20} \url{https://en.wikipedia.org/wiki/Memory_leak}
4.7 Operations on 3-D Vectors – Vector Algebra

3-D vectors

Vector algebra operations are carried out using Cartesian components for 3-D vectors. The $x,y,z$-components of a typical vector $\mathbf{V} = V_x \mathbf{i} + V_y \mathbf{j} + V_z \mathbf{k}$ are stored in a 1 by 3 matrix in the following order:

- First element $\mathbf{V}[1]$: $x$-component
- Second element $\mathbf{V}[2]$: $y$-component
- Third element $\mathbf{V}[3]$: $z$-component

By assigning 0 to the $z$-components, vector operations give results for 2-D vectors\(^\text{21}\).

Defining Vectors

The following vectors will be used in subsequent examples.

- $\mathbf{F} = -3 \mathbf{i} + 5 \mathbf{j} + 2 \mathbf{k}$
- $\mathbf{P} = \mathbf{j} + 4 \mathbf{k}$
- $\mathbf{Q} = 2 \mathbf{i} + 6 \mathbf{k}$
- $\mathbf{R} = \mathbf{F} + \mathbf{P} + \mathbf{Q}$

Scalar expressions:

```c
main()
{
    defmat(F[3], -3, 5, 2); // returns 0
    defmat(P[3], 0, 1, 4);
    defmat(Q[3], 2, 0, 6);
    !R = F + P + Q;
    print(F, P, Q, R);
}
```

Note that `defmat()` returns 0 in these scalar expressions.

\(^{21}\) Even when 2-D vectors are used, operations such as the cross product will give result as 3-D vectors. For this reason, all vectors in vector operations must be allocated as 3-D vectors.
Matrix expressions:

When being used within a matrix expression, defmat() returns the matrix being defined:

```cpp
main()
{
    ! R = defmat(F[3], -3, 5, 2) + defmat(P[3], 0, 1, 4) + defmat(Q[3], 2, 0, 6);
    print(F, P, Q, R);
}
```

The returned matrix can be used for further computation within the same expression.

1. Length of a Vector

The length (or magnitude) of a general vector $\mathbf{F}$ is $L = \sqrt{\sum_{i=1}^{2,1} F_i^2}$

For the special case of 3-D physical vectors: $L = \sqrt{F_x^2 + F_y^2 + F_z^2}$

Built-in function `hypot()` can be used to compute the length $L$ of a matrix or vector, but the matrix argument/expression must be prefixed by '!':

```cpp
main()
{
    // Length of vector R
    LR = hypot(! R = defmat(F[3], -3, 5, 2) + defmat(P[3], 0, 1, 4) + defmat(Q[3], 2, 0, 6));
    cat();
}
```
2. Scaling, Addition, and Subtraction of Vectors

To evaluate the vectorial expression \( \mathbf{R} = 0.4\mathbf{F} + 3\mathbf{P} - 5\mathbf{Q} \), we execute the following matrix expression:

\[
! \mathbf{R} = 0.4 * \mathbf{F} + 3 * \mathbf{P} - 5 * \mathbf{Q};
\]

It gives \( \mathbf{R} = <-11.2, 5, -17.2> \) i.e. \( \mathbf{R} = -11.2\mathbf{i} + 5\mathbf{j} - 17.2\mathbf{k} \)

Unit vectors

The unit vector \( \mathbf{u} \) parallel to a given vector \( \mathbf{F} \) is obtained by scaling \( \mathbf{F} \) by its length:

\[
! \mathbf{u} = \mathbf{F}/\text{hypot}(\mathbf{F}); // \text{Prefix '!'} \text{ appears twice}
\]

Similarly, the unit vector parallel to \( \mathbf{R} = 0.4\mathbf{F} + 3\mathbf{P} - 5\mathbf{Q} \) may be obtained in two steps:

\[
! \mathbf{R} = 0.4 * \mathbf{F} + 3 * \mathbf{P} - 5 * \mathbf{Q};
! \mathbf{u} = \mathbf{R}/\text{hypot}(\mathbf{R});
\]

or in one step:

\[
! \mathbf{u} = (\mathbf{R}=0.4*\mathbf{F}+3*\mathbf{P}-5*\mathbf{Q})/\text{hypot}(\mathbf{R});
\]

3. Scalar Product of Vectors

Physical 3-D vectors

In vector algebra, the dot or scalar product of two vectors:

\[
\mathbf{P} = P_x\mathbf{i} + P_y\mathbf{j} + P_z\mathbf{k} \quad \text{and} \quad \mathbf{Q} = Q_x\mathbf{i} + Q_y\mathbf{j} + Q_z\mathbf{k}
\]

is the scalar \( D = \mathbf{P} \cdot \mathbf{Q} = P_xQ_x + P_yQ_y + P_zQ_z \)

This operation is evaluated by matrix expression as follows

\[
D = ! \mathbf{P} * \mathbf{Q}; // D \text{ is a scalar}
\]

\* Components of a unit vector \( \mathbf{u} \) are the cosines of the angles between the axes and \( \mathbf{u} \). They are the direction cosines of any line parallel to \( \mathbf{u} \).
If we write instead

\[ d = P \times Q; \quad // \quad d \text{ is a 1x1 matrix} \]

then the result 'd' is a 1x1 matrix.

Any vectors

In the general case of vectors of arbitrary size N: \( P_{1xN}, \ Q_{3xN} \), the scalar (or inner) product is defined in the same way:

\[ D = P \cdot Q = P_1Q_1 + P_2Q_2 + \ldots + P_NQ_N = \sum_{i=1}^{N} P_iQ_i \]

This operation is evaluated as follows

\[ D = ! P \times Q; \quad // \quad D \text{ is a scalar} \]

or

\[ d = P \times Q; \quad // \quad d \text{ is a 1x1 matrix} \]

A special case: The dot product of two identical vectors is equal to the square of the length of the vector:

\[ D = P \cdot P = P_1^2 + P_2^2 + \ldots + P_N^2 = \sum_{i=1}^{N} P_i^2 \]

\[ D = ! P \times P; \quad // \quad D \text{ is a scalar} \]

Thus, the length of P can also be computed by:

\[ L = \sqrt{D = ! P \times P}; \]

### 4. Cross Product of Physical 3-D Vectors

The cross product of two 3-D vectors \( P \) and \( Q \) is a vector \( C \) given as:

\[
C = P \times Q = \begin{vmatrix}
i & j & k \\
P_x & P_y & P_z \\
Q_x & Q_y & Q_z \\
\end{vmatrix} = (P_yQ_z - P_zQ_y)i + (P_zQ_x - P_xQ_z)j + (P_xQ_y - P_yQ_x)k
\]

The operator \( ^\wedge \) in matrix expression gives the desired result:
4.32

5. Mixed Product of Physical Vectors

Any expression in vector algebra can be translated directly into an equivalent matrix expression for evaluation. For example, the following mixed product \( T = F \cdot P \times Q \) is a scalar, which can be evaluated as:

\[
T = F \cdot (P \times Q) \quad // \text{Resulting } T \text{ is a scalar}
\]

where the resulting \( T \) can be subsequently used as a scalar float-value.

Alternatively, we can also write

\[
M = F \cdot P \times Q \quad // \text{Resulting } M \text{ is a 1 by 1 matrix}
\]

and the resulting \( M \) is a 1 by 1 matrix that can be subsequently used as a matrix.

Practice Drills 4.5

Write a program that (a) defines the physical vectors \( A = 3i + 2j - 4k \), \( B = -2i + 3k \), \( C = i - 4j + 2k \), \( D = -i + 2j - 3k \); (b) uses a single expression for each of the following questions.

1. Find the magnitude of \( P = A + B + C + D \)
2. Find the magnitude of \( Q = 2A - 4B + 3C - D \)
3. Find the unit vector parallel to the resultant of \( A, B, C, D \).
4. If \( A \) and \( B \) are the position vectors of two points \( A \) and \( B \), find the vector from \( A \) to \( B \), its length, and the unit vector parallel to line \( AB \).
5. Verify that the length of \( A + B + C + D \) is less than the sum of the lengths of the four vectors.

\[
\text{Hint: if(hypot(V = A+B+C+D) < (hypot(A)+hypot(B)+hypot(C)+hypot(D)))}
\]
\[
\{ \text{print("Inequality 5 is verified."); } \}
\]
6. Verify that $A \cdot (B + C) = A \cdot B + A \cdot C$

   Hint$^{22}$: if(hypot($!V = A \cdot (B+C) - (A \cdot B + A \cdot C)) < 1e-10)$
   { print('^', "Equality 6 is verified."); }

7. Verify that the length of $A$ is also given by the square root of the dot product $A \cdot A$.

8. Verify that $A \times B = -B \times A$

   Hint$^{23}$: if(hypot($!V = A^B + B^A) < 1e-10)$
   { print('^', "Equality 8 is verified."); }

9. Verify that $A \times (B + C) = A \times B + A \times C$

10. Verify that $A \cdot (B \times C) = B \cdot (C \times A) = C \cdot (A \times B)$

11. Verify that $A \times (B \times C) = B(A \cdot C) - C(A \cdot B)$

12. Verify that $(A \times B) \cdot (C \times D) = (A \cdot C)(B \cdot D) - (A \cdot D)(B \cdot C)$

13. Verify that $(A \times B) \cdot (B \times C) \cdot (C \times A) = (A \cdot B \times C)^2$

---

### 6. Use of Vectors Stored in 2-D Arrays

In application programming, it is convenient to store vectors as rows of a 2-D array:

```
[ -3  5  2 ]
[  0  1  4 ]
[  2  0  6 ]
[ -5  4.5  3 ]
[  2.8 -2.6  9.3 ]
```

```c
defmat(A[5,3], -3, 5, 2, 0, 1, 4, 2, 0, 6,
       -5, 4.5, 3, 2.8, -2.6, 9.3);
```

The following operations on rows of a 2-D array are available.

**Rows**

- Copy rows 1, 2, 3 of array $A$ into vectors $F$, $P$, $Q$ respectively:

  ```c
  ! F = A(1);    // A(1): row 1 of array A
  ! P = A(2);    // A(2): row 2
  ! Q = A(3);    // A(3): row 3
  ```

$^{22}$ Since only scalars can be compared in logical expressions, we test for the near-zero length of the 1x1 vector storing the scalar $A \cdot (B + C) - (A \cdot B + A \cdot C)$.

$^{23}$ The vector $A \times B + B \times A$ is a null vector if and only if its length is zero.
The usual operations of vector algebra can also be carried out using the rows of array A directly:

\[
R = 0.4 \times A(1) + 3 \times A(2) - 5 \times A(3);
\]

\[
D = A(2) \times A(3); // Dot product of rows 2 & 3
\]

\[
C = A(2) \wedge A(3); // Cross product of two rows
\]

\[
T = A(1) \times A(2) \wedge A(3); // Scalar triple product
\]

Columns

To denote a column, say column 3, of array A, we write:

\[A(3)\] // Column 3 of array A

Sample Program 4.8 - Use of Row, Column Operators

The following program demonstrates the use of row/column operators within loops. It computes the average values of individual rows and columns of array XYZ. The results are stored in vectors RowAve and ColAve.

```c
main()
{
    // Define array elements, row by row:
    defmat(XYZ[7,3],-12.5, 23.2, 16.4, 5.6, -4.7, -8.2,
           1.4, 3.6, 4.6,     -2.5, 2.32, 3.64,
           15.2, -4.23, 3.16, 5.5, 8.42, -7.42,
           -9.5, -6.3, -8.4);
    print(XYZ);
    zero(RowAve[7], // Create storage for rows' averages
         ColAve[3]); // Storage for columns' averages
    float I, J; // Local vars for loop indexes
    for(I=1; I<=7; I=I+1){ // Row I=1,2,..7
        !ColAve = ColAve+XYZ(I); // Add row I into ColAve
        print(" Partial sum up to Row ",I, ",", ColAve);
    }
}
```

24 Many common operations on arrays such as mean value, maximum, minimum, statistics of elements, etc.. are handled by built-in functions.

© 2015 by K. Ha - BCEE 231 Chapter 4 - v4.
!ColAve = ColAve/7;

println(" Column Averages", ColAve);

for (J=1; J<=3; J=J+1) // Column J=1,2,3
{
    !RowAve = RowAve+XYZ{J}; // Add column J into RowAve
    println(" Partial sum up to Column ",J, ", RowAve);
}

!RowAve = RowAve/3;

println(" Row Averages", RowAve);
cat();

4.8 Passing Arrays to Built-in Functions

Many built-in functions require array arguments. When invoked, array names must be supplied for those arguments. Some examples:

invert(A); // Matrix inverse in place
invert(!A); // ! is ok, but not really needed
invert(!A = C * D); // ! needed for expression

Common math functions such as sin(), cos(), exp(), abs(), hypot(), etc.. normally expect only scalar arguments. Most of them, however, also accept array arguments prefixed by ! to signal a matrix name or matrix expression.

Sample Program 4.9 – Array Arguments in common math functions.

Most common math built-in functions can accept mat-argument, and they operate on every element of the array. The mat-argument need be preceded by the character '!' in such case.

1.

main()
{
    // Exponentials of array elements
defmat(A[5], 0.2, 3, -2, 0, 1.5);
2. main() // Sines of array elements
{
    defmat(A[5], 0.2, 3, -2, 0, 1.5);
    sin(!A); // Replace elements of A by their sines
    Len = hypot(!A); // Length of matrix A
    print(A, ^, " Length = ", Len);
}

3. main() // Arc-cosine of array elements
{
    defmat(A[5], 1, sqrt(3)/2, sqrt(2)/2, 0.5, 0);
    acos(!A); // Replace A by its acos in radians
    !A = A*(180/pi#); // A in degrees
}

Practice Drills 4.6

1. Define in CALC the following two vectors (1-D arrays) of size $N = 6$:

   $A = \{1.2, -2.9, -4.16, 0.65, 6.45, 3.35\}$
   $B = \{1.4, -3.1, -3.78, 1.02, 5.97, 3.13\}$

   and write one single self-contained expression to compute each of the followings:

   (a) Vector $C = B - A$ \(i.e.\) $C_i = B_i - A_i$ \(i = 1,2,\cdots,N\)

   (b) Vector $D_i = A_iB_i$ \(i = 1,2,\cdots,N\)

   Hint: Use array operation: \(!D = A \&* B;\)
(c) Vector $E_i = A_i (B_i - A_i), \ i = 1, 2, \ldots, N$
Hint: Use array operation: $!E = A \&* (B - A)$;

(d) Vector $G_i = 100 \left| \frac{A_i - B_i}{B_i} \right|, \ i = 1, 2, \ldots, N$
Hint: $!G = 100 \times \text{abs}(!C = (A-B) \&/ B)$;

(e) Scalar value:
$$L = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2 + \cdots + (B_N - A_N)^2}$$
$$= \sqrt{\sum_{k=1}^{N} (B_k - A_k)^2}$$
Hint: $L = \text{hypot}(!C = B-A)$;

(f) Vector $F_i = \frac{A_i (B_i - A_i)}{\sqrt{\sum_{k=1}^{N} (B_k - A_k)^2}}, \ i = 1, 2, \ldots, N$
Hint: $!F = (A \&* (C = B-A)) / \text{hypot}(!C)$;

2. Define in CALC the following two vectors (1-D arrays) of size $N = 6$:
\[ A = \{ 3, 2, 4, 3.3, 4.3, 2.6 \} \]
\[ B = \{ 1, -3, 2, -2, 4, 7 \} \]
And then write one single self-contained expression to compute each of the following scalar values:

(a) $S = A_1 B_1 + A_2 B_2 + \cdots + A_N B_N = \sum_{i=1,2,3,\ldots}^{N} A_i B_i$
Ans: 33.8

(b) $E = \sqrt{(A_1 B_1)^2 + (A_2 B_2)^2 + \cdots + (A_N B_N)^2} = \sqrt{\sum_{i=1,2,3,\ldots}^{N} (A_i B_i)^2}$
Hint: Use array operation: $E = \text{hypot}(!C = A \&* B)$;
Ans: 27.922

(c) $P = (A_1 + A_2 + \cdots + A_N)(B_1 + B_2 + \cdots + B_N) = \left( \sum_{i=1,2,3,\ldots}^{N} A_i \right) \left( \sum_{i=1,2,3,\ldots}^{N} B_i \right)$
Hint: Use summat()
Ans: 172.8
(d) 

\[ Q = (A_1^2 + A_2^2 + \cdots + A_N^2)(B_1^2 + B_2^2 + \cdots + B_N^2) = \left( \sum_{i=1}^{N} A_i^2 \right) \left( \sum_{i=1}^{N} B_i^2 \right) \]

Hint: Use vector dot products: \( Q = !(A*A)*(B*B) \);
Ans: 5406.62

(e) 

\[ Sd = \left[ \frac{1}{N} \left( \sum_{k=1}^{N} A_k^2 \right) - \left( \frac{1}{N} \sum_{k=1,2,3\ldots} A_k \right)^2 \right]^{1/2} \]

Ans: \( Sd = \sqrt{!(A*A)/6 - (\text{summat}(A)/6)^2} \);
\[ = 0.785281 \]

Practice Drills 4.7

1. Define the following matrix:

\[
\begin{bmatrix}
-3 & 5 & 2 \\
0 & 1 & 4 \\
2 & 0 & 6 \\
-5 & 4.5 & 3 \\
2.8 & -2.6 & 9.3
\end{bmatrix}
\]

And then write one single self-contained expression to compute each of the following scalar values:

(a) The average of the elements in row 4 of \( A \)
Ans: \( A = \text{summat}(A(4))/\text{ncol}(A); \)
(b) The average of the elements in column 2 of \( A \)
(c) The dot product of rows 1 and 5 of \( A \)
(d) The cross product of rows 1 and 5 of \( A \)

2. Reorganize the Sample Program 4.8 into 4 functions with NO function arguments/parameters:

- main();
- GetData();
- GetRowAverages();
- GetColumnAverages();

3. Reorganize the Sample Program 4.8 into 4 functions with appropriate function arguments & parameters such that ALL variables are local
variables.

- main();
- GetData(......);
- GetRowAverages(......);
- GetColumnAverages(......);

### 4.9 Functions Returning Matrices

When a function returns a matrix, its name (in the calling matrix expression) represents the returned matrix, which may be directly used within the matrix expression. We'll show application examples of matrices returned by built-in and by user-defined functions.

#### 1. Built-in Functions

Built-in functions such as `defmat()`, `zero()` return 0 to scalar expressions. However, when used in matrix expressions, they will return the defined matrices.

```plaintext
main()
{
  S = 10 * defmat(P[3], 1, 2, 3);  // Scalar expression
  V = 0.1 * defmat(P[3], 1, 2, 3); // Matrix expression
  useoption("DEGREES");
  R = cos(!q = defmat(Q[3], 30, 45, 60));
  print(P, V, Q, q, R);
  cat();
}
```

### Practice Drills 4.8

1. Consult CMAP-documentation on the various options available in function `defmat()`, and then execute the following expressions in CALC--in the given sequence:

```plaintext
! defmat(A[3], 1::6::2);
! defmat(B[4], 2::8::2);
! defmat(C[7], A, B);
```
! defmat(D[5], !defmat(E[3], 8, 9, 10), !defmat(F[2], 11, 13));
! defmat(G[7,2], A, B, C);
! defmat(g[2,7], A, B, C);
! defmat(H[7,2], A, B, C);

2. Matrix returned by User-defined Function

A user-defined function may also return a matrix expression using any of the following formats:

\[
\text{return } \text{MatName}; // \text{Return matrix MatName}
\]
\[
\text{return } \text{MatName} = \text{Expr}; // \text{Return matrix MatName}
\]

where MatName must be either a global matrix or a mat-parameter of the function. It cannot be a local matrix because its storage is deleted when the function returns.

Note: Value of a local float-variable can be returned even though the local-var's storage is deleted.

Sample Program 4.10 – Function Returning Array

The following program is similar to Sample Program 4.3 with the following changes:

- AddArrays() returns the matrix sum.
- main() makes use of the above new feature to sum 4 vectors.

```plaintext
main()
{
    // Sum of 4 vectors R = F1 + F2 + F3 + F4
    defmat(F1[N=3], -12.5, 23.2, 16.4); // Define F1
    defmat(F2[N], 5.6, -4.7, -8.2); // Define F2
    defmat(F3[N], -0.6, 2.3, 4.5); // Define F3
    defmat(F4[N], 4.3, -0.8, 2.3); // Define F4
    zero(R[N], R1[N], R2[N]); // Create storage
    !R = AddArrays(F1, F2, R1) + // Returns R1
        AddArrays(F3, F4, R2); // Returns R2
    print(F1, F2, F3, F4, R1, R2, R);
}
```
AddArrays(mat A, mat B, mat C)
{
    // Add C <= A+B, both A & B must be of same size
    return !C = A+B;  // Return matrix sum in C
}

4.10 Arrays Versus float-Variables

The ability of arrays to hold many values simultaneously in memory is the principal reason for using array storage. Array storage is indispensable for many situations where

- The quantity is associated with many values (e.g. vectors, matrices).
- The entire set of values must reside in memory in order to simplify the programming (e.g. system of equations, sorting).

Array storage should not be used in situations where the required data could be input, generated and used within a loop. Use of array storage may complicate programming in such cases.

4.11 Useful Built-in Functions

The following built-in functions have been introduced in this Chapter. Complete information on them is available in CMAP-Help.

- defmat(...) Creates an array and defines array elements.
- hypot(!X) Returns the length of vector X; i.e. $\sqrt{X_1^2 + X_2^2 + \cdots + X_N^2}$
- resizemat(...) Resize a matrix while keeping its values where possible.
- sin(!X) Replaces the elements of X by their sines, and similarly for cos(!X), tan(!X), acos(!X), etc..
- view(...) Creates variables and gets user-input
- zero(...) Creates arrays and initializes elements with zero.
Part II: Application of Procedural Programming

Chapter 5: Application to Statics Problems

The purpose of computing is insight, not numbers.
-- Hamming
5.1 Resultant of Forces

In this section, we write small special-purpose programs to solve specific problems of engineering statics. More general programs are presented in later sections.

1. The Resultant of 2-D Forces

Sample Program 5.1 - Resultant force in 2-D

Given 4 forces in the xy-plane:

<table>
<thead>
<tr>
<th>Force No. i</th>
<th>Magnitude, F_i (N)</th>
<th>Angle with x-axis, A_i (°)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>150</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>-90</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>-15</td>
</tr>
</tbody>
</table>

determine the resultant \( \mathbf{R} \) of these forces (i.e. \( \mathbf{R} = \sum_i F_{ix} + j \sum_i F_{iy} \)).

Note: This is Sample Problem 2.3 on p.27 of "Vector Mechanics for Engineers - Statics" by F. P. Beer and E. R. Johnston Jr, Third SI-Metric Edition, McGraw-Hill.

Solution:

Data structure

We store the forces in a 2-D array \( \mathbf{FA} \) where the \( i^{th} \)-row is for the \( i^{th} \)-force (i.e. Column 1 contains the forces’ magnitudes, and column 2 the angles).

```c
 ///////////// main() /////////////
{ // Version 1: Low-level programming
     useoption("DEGREES");
     defmat(FA[N=4,2], 150, 30, // Store data in table. Note 1
             80, 110, 110, -90, 100, -15);
     zero(R[2]); // Create & initialize resultant vector
     float i;   // Loop index
```
```
print(^^, " Force Fx Fy");
for(i=1; i<= N; i=i+1) {  // Loop over N forces
    Fi = FA[i,1];   // Magnitude
    Ai = FA[i,2];   // Angle
    R[1] = R[1]+ (Fx = Fi* cos(Ai)); // Sum x-component
    print(^, i, Fx, Fy);
}
print(^^" Resultant force", R, ^
    " Magnitude =", mR = hypot(!R),
    " Angle =", atn2(R[1], R[2]));
```

Note 1: This line may be replaced by the following two lines that will request data input during run time.

```java
N = getnum("Enter number of vectors", 4);
view(FA[N,2]);
```

2. Components of a Force Directed Along a Line in Space

A vector \( \mathbf{F} \) can be represented by its components:

\[
\mathbf{F} = F_x \mathbf{i} + F_y \mathbf{j} + F_z \mathbf{k} = F(c_x \mathbf{i} + c_y \mathbf{j} + c_z \mathbf{k}) = \mathbf{F} \mathbf{u}
\]

where

- \( F \) : the magnitude of \( \mathbf{F} \)
- \( c_x, c_y, c_z \) : cosines of the angles between \( \mathbf{F} \) and the axes \( x, y, z \), respectively (i.e. The direction cosines of \( \mathbf{F} \))
Sample Program 5.2 - Unit vector and force components

Given two points A (40, 0, -30) and B (0, 80, 0) and a force $F = 2500$ N directed from A to B. Determine:

- The unit vector $u$ along AB from A to B.
- The components of $F$.
- The angles between $F$ and the axes.

Note: This is the Sample Problem 2.7 of Beer & Johnston text, p. 46

```cpp
main()
{
    defmat(A[3], 40, 0,-30);  // Coordinates of A
    defmat(B[3],  0,80,  0);  // Coordinates of B

    ! AB = B-A;                // Vector AB
    ! u = AB/hypot(!AB);      // Unit vector along AB
    ! F = 2500*u;             // Components of F

    useoption("DEGREES");
    acos(!Theta = u);        // Angles in Theta
    print(A, B, u, F, Theta);
}
```

3. The Resultant of Forces in Space

Sample Program 5.3 - Resultant force in 3-D

Given:

- Three points A (16, 0, -11), B (0, 8, 0) and C (0, 8, -27)
The force \( T_{AB} = 8.4 \text{ kN} \) directed from A to B

The force \( T_{AC} = 12 \text{ kN} \) directed from A to C.

Determine the resultant of these two forces, its magnitude and direction.

Note: This is the Sample Problem 2.8 of Beer & Johnston text, p. 47.

Solution

```c
main()
{
   defmat(A[3], 16, 0,-11); // Coordinates of A
   defmat(B[3], 0, 8, 0); // Coordinates of B
   defmat(C[3], 0, 8,-27); // Coordinates of C
   ! AB = B-A;    // Vector AB
   ! uab = AB/hypot(!AB); // Unit vector along AB
   ! Tab = 8.4*uab;  // Components of Tab
   ! AC = C-A;    // Vector AC
   ! uac = AC/hypot(!AC); // Unit vector along AC
   ! Tac = 12*uac;  // Components of Tac
   ! R = Tab + Tac;  // Resultant
   mR = hypot(!R);  // magnitude of R
   ! uR = R/mR;    // Unit vector along R
   useoption("DEGREES");
   acos(!Theta = uR);  // Angles
   print(R, " Magnitude", mR, Theta);
}
```

Practice Drills 5.1

1. Revise Sample Program 5.1 so that, instead of defining the array FA, the data is stored in two 1-D arrays as shown below:

   ```c
   defmat(F[N=4], 150, 80, 110, 100); // Magnitudes
   defmat(A[N], 30, 110, -90, -15); // Angles
   ```
2. Revise Sample Program 5.1 so that, instead of defining array FA, the data for a force is read within the loop as shown below.

```c
for(i=1; i<= N; i=i+1) {  // Loop over N forces
    view(Fi[2]);   // Define force Fi
    ...
}
```

3. Write the function `UnitVect(mat A, mat B, mat u)` that accepts two points A and B in 3-D space and returns the unit vector along AB.

Write the function `main()` to test `UnitVect()` with some suitable data.

4. The coordinates of two points are stored in arrays A and B. Write only one expression to compute each of the following:

- The components of the vector from A to B.
- The unit vector along AB.
  
  Ans: \( \mathbf{u}_{AB} = \frac{(\mathbf{B} - \mathbf{A})}{\text{hypot}(AB)} \);

- The angles that the vector from A to B makes with the coordinate axes.
  
  Ans: \( \Theta = \cos^{-1}\left(\frac{(\mathbf{B} - \mathbf{A})}{\text{hypot}(AB)}\right) \);

5.2 General Program for Resultant Force Problems

In this section, you will learn to:

- (a) Pose an engineering problem
- (b) Formulate it in a concise mathematical form
- (c) Outline the steps for solution
- (d) Select the data structure for the convenience of input and programming
- (e) Implement a computer program for solution.
1. Problem Statement

In Fig. 5.1, the particle $A_o$ is subject to $n$ known forces $F_i$, $i = 1, 2, \ldots, n$. The line of action of the force $F_i$ is specified by the directed line $A_oA_i$. Positions of all points are given (see Fig. 5.1).

The objective is to determine the resultant of all given forces. Here, all forces are known. Note: This is Problem 2.C4 of Beer & Johnston text, p. 66.

2. Formulation

Given: $n$ forces $F_i = F_i u_i$, $i = 1, 2, \ldots, n$

where the unit vectors $u$ can be found from the positions of the points. The direction cosines of the force $F_i$ are the components of the unit vector $u_i$:

$$u_i = c_{x_i} i + c_{y_i} j + c_{z_i} k$$

i.e. $F_i = F_i \left(c_{x_i} i + c_{y_i} j + c_{z_i} k\right)$

![Diagram](image)

Figure 5.1: A typical force $F$

Solution: The resultant force $R_k$ is the (vectorial) sum of all given forces:

$$R = \sum_{i=1}^{n} F_i u_i$$
3. Pseudocode for Task Division

1. Get input data and allocate storage

2. Initialize the resultant vector \( \mathbf{R}_k \) of known forces to 0

3. For each force vector \( \mathbf{F}_i \), \( i=1,2,...n \), do:
   - Get the unit vector \( \mathbf{u} \) along the force \( \mathbf{F}_i \)
   - Add the vector \( \mathbf{F}_i \mathbf{u} \) to vector \( \mathbf{R}_k \)

4. Output results

4. Data Structure

The actual implementation depends on how we store the data input. To allow data inspection after execution, the following variables are global.

float-variables:
- \( n \): total number of known forces.
- \( nD = 2 \) (for 2-D forces) or 3 (for 3-D forces)

Arrays (mat-variables):
- Position of point \( \mathbf{A}_0 \) is stored in 1-D array: \( \mathbf{A}_0[nD] \)
- Position of \( n \) points \( \mathbf{A}_i \) are stored in 2-D array: \( \mathbf{xyz}[n, nD] \)

Table \( \mathbf{xyz} \) has \( n \) rows and \( nD \) columns. The \( i^{th} \)-row contains the coordinates of the point \( \mathbf{A}_i \) on the line of action of force \( i \). Note that the direction from \( \mathbf{A}_0 \) to \( \mathbf{A}_i \) is the positive direction for the force vector \( \mathbf{F}_i \). It is also the true sense if the magnitude \( F_i \) is positive.
• The magnitudes of all known forces are stored in 1-D array: \( F[n] \). This 1-D array contains the algebraic magnitude of \( n \) forces. If the magnitude \( F_i \) is positive, the force's true sense is from \( A_0 \) to \( A_i \).

### 5. Implementation

The tasks are implemented in user-defined functions. The function `GetResultant()` computes the components of each force and adds up the forces to get the resultant force. It prints the components as soon as they are available. It also prints proper headings for easy interpretation of the output.

```c
main()
{
    DefineData(); // Get data at runtime
    GetResultant(); // Compute and print results
}

DefineData()
{
    // Global data, runtime input
    n = getnum(" Total number of forces ", 4);
    nD = getnum(" Enter 2 for 2-D or 3 for 3-D problem ", 3);
    view(Ao[nD], xyz[n,nD], F[n]);
}

GetResultant()
{
    // Temporary storage
    float i, j; // Local vars for loop indexes
    mat u[nD], // Unit vector
        Ai[nD], // Vector from Ao to Ai
        Fi[nD]; // Storage for components of \( F_i \)

    print(^^, " Force Fx Fy Fz");
    zero(Rk[nD]); // Initialize resultant of known forces
    for(i=1; i<= n; i=i+1) // Loop through all \( n \) forces \( F_i \)
    {
        Ai = xyz(i) - Ao; // Vector from Ao to Ai
    }
```
5.3 Issues Related to Equilibrium Problems

1. What is Really in Equilibrium?

Should we say:

- The forces are in equilibrium\(^1\), or
- The body is in equilibrium?

We should prefer the second statement for the following reasons:

- First, the body has to be identified (as a free-body diagram);
- ONLY then that the forces on the body could be identified. These forces (known and unknown) must cancel themselves so that the body is in equilibrium.

Correct FBDs (free-body diagrams) become essential for solving statics problems.

2. When is a Body a Particle?

A body (e.g. a plane, boat, swimmer, train, car, etc...) is the same as a particle, as far as its equilibrium is concerned when the (relevant) forces acting on it are concurrent.

---

\(^1\) This loose statement is meaningful only when the body on which the forces act is clear. For clear thinking, it is best to avoid it.
3. Storage of Unknown Forces

(a) Forces with Known Lines of Action

The unknown tension in a fixed cable has the direction along the cable. In such case, only the magnitude $U$ of the force is unknown. The components of $\mathbf{U}$ can be expressed in terms of the magnitude $U$ and the known components of the unit vector along the cable:

$$\begin{align*}
\mathbf{U} &= U \left( c_x \mathbf{i} + c_y \mathbf{j} + c_z \mathbf{k} \right)
\end{align*}$$

where the direction cosines $c_x$, $c_y$, $c_z$ must satisfy the relation:

$$c_x^2 + c_y^2 + c_z^2 = 1.$$ 

The direction cosines may be computed from the position of any two points on the line of action of the force.

The required data for such an unknown force may consist of:

- A scalar variable to contain the magnitude.
- A vector to contain the known direction cosines.

Example: Unknown tension in the cable AB

```plaintext
U = 0; // Allocate storage for the unknown magnitude
defmat(A[3], ?, ?, ?); // Position of point A
defmat(B[3], ?, ?, ?); // Position of point B
!uAB = (B-A)/hypot(!AB); // Unit vector along AB
// Once U found, the vector of cable tension is
! TAB = U*uAB; // Components of cable force
```

In case of many forces, we may use:

- 2-D arrays to store the positions of the points or the direction cosines of the forces.
- 1-D array to store the force magnitudes.

---

2 Components of a unit vector $\mathbf{u}$ are the cosines of the angles between the axes and $\mathbf{u}$. They are often called the direction cosines, and denoted by $\{ c_x, c_y, c_z \}$.

3 The length of a unit vector is 1, and therefore the sum of the squares of its components is 1.
(b) Forces with Unknown Lines of Action

When the line of action of $U$ is unknown (e.g. the reaction force at a hinge support), each component of $U$ becomes an independent unknown force that has:

- Unknown magnitude
- Known direction (along $x$, $y$ or $z$ axes).

Thus this type of unknown forces is converted into a number of unknown forces of type (a) above.

5.4 Linear Simultaneous Equations

Solution of an equilibrium problem is formulated, in a general way, as solution of a system of linear simultaneous equations\(^4\). The unknowns are the magnitudes of the forces (the lines of actions of the forces are known).

1. Problem Statement

Consider the following equations that contain constants $A_{ij}$, $B_i$ and first order terms of the unknowns $X_i$:

\[
\begin{align*}
A_{11}X_1 + A_{12}X_2 + A_{13}X_3 &= B_1 \\
A_{21}X_1 + A_{22}X_2 + A_{23}X_3 &= B_2 \\
A_{31}X_1 + A_{32}X_2 + A_{33}X_3 &= B_3
\end{align*}
\]

where the coefficients $A_{ij}$, $B_i$ are known. We want to determine the values of $X_i$ that ensure the equality in Eq.1.

2. Matrix Form

We now use matrix algebra to regroup the information in Eq.1 in order to separate known values from the unknowns $X$:

\[^4\text{This topic of system of linear equations is of great importance in many sciences, and will be more fully treated in Chapter 7.}\]
where the product of matrix $\mathbf{A}$ and vector $\mathbf{X}$ is equal to vector $\mathbf{B}$. Note that the product $\mathbf{AX}$ is a vector of order 3, the same as the order of $\mathbf{B}$.

3. Solution by the Built-in Function solve()

Given the coefficient matrix $\mathbf{A}$ and the right-hand-side vector $\mathbf{B}$, the solution vector $\mathbf{X}$ may be found using option 'S' of function solve():

```plaintext
solve(S, A, B);
```

Sample Program 5.4 - Solution of simultaneous linear equations

Solve for $X_i$ so that the following matrix equation is satisfied:

$$
\begin{bmatrix}
3 & 2 & 0 \\
-1 & -2 & 4 \\
2 & -1 & -3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= 
\begin{bmatrix}
20 \\
2 \\
0.01
\end{bmatrix} 
\Rightarrow \mathbf{A}_{3\times 3}\mathbf{X}_{3\times 1} = \mathbf{B}_{3\times 1}
$$

Program

```plaintext
main()
{
    defmat(A[3,3], 3, 2, 0, -1, -2, 4, 2, -1, -3);
    defmat(B[3], 20, 2, 0.01);
    print(" Data", A, B);
    solve(S, A, B); // Note 1
    print(" After solution by solve",
          A, "^ Solution", B);
}
```

---

5 Two matrices (or vectors) $\mathbf{C}$ and $\mathbf{D}$ are equal if and only if each and every element of $\mathbf{C}$ is equal to the corresponding element of $\mathbf{D}$. That is $C_{ij} = D_{ij}$ for all indexes $i, j$.  

© 2015 by K. Ha - BCEE 231 Chapter 5 - v4.3
**Note 1:** Option S of `solve(S, A, B)` has the following actions (as documented in CMAP-Help):

- It decomposes the original matrix A and replaces the original matrix by the decomposed matrix\(^6\).
- It replaces the original vector B by the solution vector.
- It returns 0 to scalar expressions.
- It returns the solution vector B to matrix expressions.

In order to save the original A and B, we may submit their copies to `solve()`:

```plaintext
solve(S, !a = A, !X = B);
```

and the solution is in X.

### 5.5 General Program for Equilibrium of Particles

1. **Problem Statement**

   In addition to the known forces as stated in the preceding Resultant Force Problem in Section 5.2, the particle Ao is also subject to nD forces\(^7\) having unknown magnitudes\(^8\). We want to determine these nD unknown magnitudes which ensure equilibrium of the particle Ao.

2. **Formulation**

   Given:
   - The resultant \(R_k\) of ALL known forces. It is found as in the preceding Resultant Force program.
   - The unit vectors \(u_j\) along the unknown forces (magnitudes \(U_j\)).

   Solution:
   
   The vector of unknown forces can be written in terms of the direction cosines (i.e. components of the unit vector \(u_j\)) as:

\(^6\) Matrix decomposition is a computationally intensive operation. Once A has been decomposed, a new solution for a new right hand side vector B will be much faster. That's why the decomposed matrix is saved just for this possibility. Furthermore, by stuffing the decomposed matrix back into A, we save memory. In many applications, the size of matrix A can be really large.

\(^7\) There must be nD unknown forces because there are nD equations of equilibrium, which can be satisfied only by exactly nD forces to be determined.

\(^8\) If the line of action is unknown, resolve the force into components along the axes. Each component's magnitude is an unknown force to be included in nD (see Section 5.2.3a).
\[
U_j = U_j \mathbf{u}_j \quad j = 1,2,..,nD
\]

where \( \mathbf{u}_j = c_{xj} \mathbf{i} + c_{yj} \mathbf{j} + c_{zj} \mathbf{k} \) for 3-D case.

For equilibrium of the particle, the vectorial sum of all forces, known and unknown must be zero:

\[
\mathbf{R} = \mathbf{R}_k + \sum_{i=1,nD} U_j \mathbf{u}_j = 0
\]

The matrix form of the preceding equation, making use of the direction cosines (i.e. the components of the unit vectors \( \mathbf{u}_j \)) of the unknown forces, is:

\[
\begin{bmatrix}
R_x \\
R_y \\
R_z
\end{bmatrix} = \begin{bmatrix}
R_{kx} \\
R_{ky} \\
R_{kz}
\end{bmatrix} + \begin{bmatrix}
c_{x1} & c_{x2} & c_{x3} \\
c_{y1} & c_{y2} & c_{y3} \\
c_{z1} & c_{z2} & c_{z3}
\end{bmatrix} \begin{bmatrix}
U_1 \\
U_2 \\
U_3
\end{bmatrix} = 0
\]

or

\[
\{ \mathbf{R} \}_{nDx1} = \{ \mathbf{R}_k \}_{nDx1} + [D_C]_{nDxnb} \{ \mathbf{U} \}_{nDx1} = 0
\]

Note that a column \( j \) of matrix \( D_c \) contains the direction cosines of the unknown force \( U_j \). The unknown magnitudes \( \{ U_1 \ U_2 \ U_3 \} \) can now be found using the built-in function solve() as shown in Section 5.3

### 3. Pseudocode for Task Division

As compared to the Resultant Force Problem, there are two additional steps that must be done for an equilibrium problem:

- Form the matrix \( D_C \) to contain the direction cosines of the unknown forces
- Solve the system of equilibrium equations.

### 4. Data Structure

The actual implementation depends very much on how we store the data input. The options include:
- Store the information related to known forces separately from that of unknown forces. We need new variable names for the unknown forces as well as for related intermediate data.

- Use the same data structure of the Resultant Force Problem and have a way to tell which force is unknown.

We will use the first option as it is easier to understand. The data for a sample problem is shown below.

---

**Data for Sample Problem 2.9** *(Beer & Johnston text, p. 54)*

A 200-kg mass (in the plane XY) is supported by cables AB and AC. A force P is required to keep the mass in equilibrium at the position shown. Determine the force P as well as the cable tensions.

<table>
<thead>
<tr>
<th>n</th>
<th>1       // One known force, which is the weight of 200kg-mass.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nD</td>
<td>3       // 3-D problem</td>
</tr>
<tr>
<td>{Ao}</td>
<td>{1.2 2.0.} // Coordinates of A</td>
</tr>
</tbody>
</table>

**Known forces:**

Array \text{xyz}[n, nD]: xyz of point on the line of action of each known forces

\[
\text{Array: } \begin{bmatrix} 1.2 & 0. & 0. \end{bmatrix} \quad // \text{downward from A}
\]

Vector \text{F}[n] = \{200 * 9.81\} // Weight of the 200kg-mass

**Unknown forces:**

Array \text{XYZ}[nD, nD]: XYZ of point on the line of action of each unknown forces

\[
\text{Array: } \begin{bmatrix} 2. & 2. & 0. \end{bmatrix} \quad // \text{Force P: Point just right of A}
\]

\[
\begin{bmatrix} 0. & 12. & 8. \end{bmatrix} \quad // \text{Point B for tension in cable AB}
\]

\[
\begin{bmatrix} 0. & 12. & -10. \end{bmatrix} \quad // \text{Point C for tension in cable AC}
\]

*Tip: To correct errors in data input:*

- Click the button \text{PrtToggle} (in the Calc Expression Window) in order to avoid automatic printing of matrices by Calc
• Execute main() in Calc Expressions Window (so that existing data remains for editing).

5. Implementation

In the following code listing, the function SolveEquilibrium() is new while the rest of the program is the same as in section 5.2. The program is now able to solve both types of problems.

• To set up matrix DC, the program places the direction cosines of the unknown force $U_j$ into column $j$ of array DC.

```cpp
main()
{
    DefineData();   // Get data at runtime
    GetResultant();  // Get resultant force $R$
    SolveEquilibrium(); // Solve for unknown forces
}

DefineData()
{
    // Global data, runtime input
    n  = getnum(" Total number of forces ", 4);
    nD = getnum(" Enter 2 for 2-D or 3 for 3-D problem", 3);
    view(Ao[nD], xyz[n,nD], F[n]); // Known forces
    view(XYZ[nD,nD]);    // Line of action of unknown forces
}

GetResultant()
{
    // Temporary storage
    float i, j;   // Local vars for loop indexes
    mat  u[nD],  // Unit vector
         Ai[nD],  // Vector from Ao to Ai
         Fi[nD];  // Storage for components of Fi

    print(^^, " Force Fx Fy Fz");
    zero(Rk[nD]); // Initialize resultant of known forces
    for(i=1; i<= n; i=i+1) // Loop through all n forces Fi
```
5.18

\{ 
! Ai = xyz(i) - Ao;  // Vector from Ao to Ai  
! u = Ai/hypot(!Ai);  // Unit vector u  
! Fi = F*[i]*u;  // Force vector Fi  
! Rk = Rk + Fi;  // Add to resultant Rk  
print(^, i);  // Output components of Fi  
for(j=1; j<= nD; j=j+1)  // Loop through nD components  
{ print(Fi[j]); } 
\}

print(^" Resultant of known forces", Rk);
\}

SolveEquilibrium()
{  // Temporary storage  
  float i;  // Local var for loop indexes  
  mat u[nD],  // Unit vector  
    Ai[nD];  // Vector from Ao to Ai  
  zero(DC[nD,nD]);  // Storage for direction cosines matrix  
for(i=1; i<= nD; i=i+1)  // Loop through unknown forces  
{  
  ! Ai = XYZ(i) - Ao;  // Vector from Ao to Ai  
  ! u = Ai/hypot(!Ai);  // Unit vector u  
  !DC{i} = u;  // Place u into column i of DC  
}
  solve(S, !dc = DC, !U = -Rk);  // Solve for U  
  print(U, DC);  // Output U and original DC
}

5.6 A Better User-interface

In this section, we adapt the user-interface of Section 3.7 for the equilibrium problem program.

1. Menu System

Function main() presents and processes a menu of 6 choices: “Define Data”, “View all”, “Analyze”, “Save Data”, “Read DataFile”, and
“Quit”. The program must be able to detect user’s choice and then process the request accordingly.

![Main menu presented by inform()](image)

Figure 5.2 Main menu presented by inform()

Function main() invokes the built-in function `inform()` to display the menu in a window as shown in Fig. 5.2. When the user selects a menu item, inform() closes the window and returns a value indicating the clicked button. This value is used in the switch command for selecting the appropriate code for action.

```c
main()
{
    float Choice = 0;  // Menu selection returned inform()
    while (Choice != 'Q') // See Note #1 below
    {
        // Present menu and get user's Choice
        Choice = inform(6, 35, RGB(255, 0, 128),
                        "     Main Menu " ^^
                        B_Menu(" Define Data ", 'D'), "     ", // Note # 2
                        B_Menu(" View all ", 'V') ^^
                        B_Menu(" Analyse ", 'A'), "     ",
                        B_Menu(" Save Data ", 'S') ^^
                        B_Menu(" Read DataFile", 'R'), "     ",
                        B_Menu(" Quit ", 'Q'));
        switch(Choice)
        {
            case 'D': DefineData();    // Define data break; // Skip the rest
            case 'V': view(); break;  // View all data
        }
    }
}
```
case 'A': Analyse(); break;
case 'S': saveglobalvars(); break; // Save data
case 'R': readglobalvars(); break; // Read data
}
} // end switch
} // end while
} // end main()

Notes

1. This loop is executed as long as the choice is not 'Q'uit. The ASCII code of 'Q' is 81, but that does not concern us here.

2. When the user clicks this menu item, inform() immediately closes the window, and returns the ASCII code of 'D'. See Help-documentation for complete information on inform().

2. User-Interface for Data Input

The following replacement code for function DefineData() presents a window containing explanations for data input (Fig. 5.3).

![Figure 5.3 Input window displayed by inform()](image)

This window is displayed by invoking the function `inform()`. Observe that the information displayed in the window parallels the parameters listed in the command `inform()`. Function `inform()`
accepts a large variety of parameters for building a useable user interface. See the CMAP documentation for complete descriptions of the features offered by `inform()`.

```cpp
float n = 2, nD = 3; // Default data
mat Ao[nD], F[n], xyz[n,nD]; // Default size
mat XYZ[nD,nD];
DefineData()
{
    inform(12, 61, RGB(255, 0, 0), " Data Input"^^
    " General Data"^
    " Enter 2 for 2-D or 3 for 3-D, nD = ", nD,^" Point Ao: position of particle ",
        (view(Ao[nD])), Ao,^" Data for known forces"^" Number of known forces, n = ", (n >= 1), n,^" Position of second point on line of action ",
        (view(xyz[n,nD])),xyz,^" Magnitude of n known forces ", (view(F[n])), F,^" Data for nD unknown forces"^" Position of second point on line of action ",
        (view(XYZ[nD,nD])),XYZ
    );
}
Analyse()
{
    GetResultant(); // Get resultant force R
    if(nD == 2 || nD == 3)
        { SolveEquilibrium(); } // Solve for unknown forces
}
GetResultant()
{
    // Temporary storage
    float i, j; // Local vars for loop indexes
    mat u[nD], // Unit vector
        Ai[nD], // Vector from Ao to Ai
        Fi[nD]; // Storage for components of Fi
```
print(", " Force Fx Fy Fz");
zero(Rk[nD]); // Initialize resultant of known forces
for(i=1; i<= n; i=i+1) // Loop through all n forces Fi
{
    Ai = xyz(i) - Ao; // Vector from Ao to Ai
    u = Ai/hypot(!Ai); // Unit vector u
    Fi = F[i]*u; // Force vector Fi
    Rk = Rk + Fi; // Add to resultant Rk
    print(i); // Output components of Fi
    for(j=1; j<= nD; j=j+1) // Loop through nD components
    { print(Fi[j]); }
}
print( Resultant of known forces", Rk);
}

SolveEquilibrium()
{
    // Temporary storage
    float i; // Local var for loop indexes
    mat  u[nD], // Unit vector
         Ai[nD]; // Vector from Ao to Ai
    zero(DC[nD,nD]); // Direction cosines matrix
    for(i=1; i<= nD; i=i+1) // Loop through unknown forces
    {
        Ai = XYZ(i) - Ao; // Vector from C to Ai
        u = Ai/hypot(!Ai); // Unit vector u
        DC[i] = u; // Place u into column i of DC
    }
    solve(S, !dc = DC, !U = -Rk); // Solve for U
    print(U, DC); // Output U and original DC
}

Practice Drills 5.2

1. Revise the program for Particle Equilibrium Problems to:
   (a) Accept input data for the direction cosines of the forces (instead of the positions of the second points A’s). The program should make
sure that the sum of the squares of the direction cosines\(^9\) is equal to 1.

(b) Include a new menu item to view the computed solutions.

2. Use either the given program or your revised program to solve the following problems:

(a) Find the resultant of the two cable forces at A

(b) Find the three cable tensions supporting the block of 163kg at A

---

\(^9\) When the sum is not nearly 1, it is sufficient to issue a warning and then return. e.g.

```c
if(abs(Sum-1) > 0.001) {
    notice("Direction cosines are incorrect. Sum = ", Sum);
    return;       // Refuse to go on further
}
```
Practice Drills 5.3

1. A load $P$ is applied to the cables CA and CB. Given that $P = 400 \text{ N}$, $\alpha = 35^\circ$, $\beta = 75^\circ$, determine the cable tensions for the range $\beta - 90^\circ \leq \theta \leq 90^\circ - \alpha$ with the increment $\Delta \theta = 5^\circ$. Determine the angle $\theta$ (accurate to 2 decimal places) for which the tension in cable AC is minimum. The program should plot the computed cable tensions.

2. A 1400kg-mass at A is supported by three cables AB, AC and AD. Determine the cable tensions for $x$ varying from 0 to 15 m at 0.5m increment. Plot the results versus $x$, and discuss on the optimum position $x$.

5.7 Moment of forces

When the relevant forces acting on the body are not concurrent, the body cannot be considered a particle, and problem solution often involves the moment of forces.

Sample Program 5.5 - Moment of a Force about a Point

Given:

- The coordinates of three points:

<table>
<thead>
<tr>
<th>Point</th>
<th>$x$ (m)</th>
<th>$y$ (m)</th>
<th>$z$ (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0.24+0.08</td>
</tr>
<tr>
<td>C</td>
<td>0.3</td>
<td>0</td>
<td>0.24+2*0.08</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0.24</td>
<td>0.08</td>
</tr>
</tbody>
</table>

- A force $F = 200 \text{ N}$ directed from C to D
Determine: The moment of $\mathbf{F}$ about point A.

Note: This is the Sample Problem 3.4, p. 81, Ref. 2: "Vector Mechanics for Engineers - Statics" by F. P. Beer and E. R. Johnston Jr, Third SI-Metric Edition, McGraw-Hill.

Solution:

The moment of $\mathbf{F}$ about A may be found:

$$\mathbf{M}_A = \overrightarrow{AC} \times \mathbf{F}$$

or $$\mathbf{M}_A = \overrightarrow{AD} \times \mathbf{F}$$

```c
main() {
    defmat(A[3], 0, 0, 0.24+0.08);  // Point A
    defmat(C[3], 0.3, 0, 0.24+2*0.08);  // Point C
    defmat(D[3], 0, 0.24, 0.08);    // Point D
    !uCD = (CD = D-C)/hypot(!CD);    // Unit vector along CD
    !F = 200 * !uCD;     // Components of F
    !MAc = (C-A)^F;    // Moment about A using AC
    !MAd = (D-A)^F;    // Moment about A using AD
    print(F, MAc, MAd);
}
```

Sample Program 5.6 - Moment of a Force about an axis

In addition to the data given in Sample Program 5.5, we have point B (0.1, -0.25, 0.5). Determine the moment of $\mathbf{F}$ about the axis AB.

Solution: The moment of $\mathbf{F}$ about the axis AB can be found as:

$$\mathbf{M}_{AB} = \overrightarrow{uAB} \cdot \mathbf{M}_A$$

```c
main() {
    defmat(A[3], 0, 0, 0.24+0.08);  // Point A
    defmat(C[3], 0.3, 0, 0.24+2*0.08);  // Point C
```
Sample Program 5.7 - Solution by Moment Equilibrium

A uniform plate of 12 kN is supported by a ball-and-socket joint at A and by the two cables EC, BD. Determine the cable tensions and the reactions at A.

Note: This is Sample Problem 4.8 (of Ref. 2, p.18). The solution approach in Ref. 2 does not make use of moments about a line.

Solution: We will find the tension in cable EC by taking moment about the line AD, and the tension in cable BD by taking moment about the line AC. Finally, the reactions at A are found as the negative of the resultant of the cable tensions and the weight.

```c
main()
{
    defmat(C[3], 0, 3, 2);    // point C
    defmat(D[3], 0, 4, -8);   // point D
    defmat(E[3], 6, 0, 0);    // point E
    // remaining code...
}
```
defmat(B[3], 8, 0, 0);       // point F
defmat(G[3], 4, 0, 0);       // position of weight
defmat(W[3], 0, -12, 0);     // weight of plate

!uBD = (BD=D-B)/hypot(!BD);  // unit vector in BD
!uEC = (EC=C-E)/hypot(!EC);  // unit vector in EC
!uAC = C/hypot(!C);          // unit vector in AC
!uAD = D/hypot(!D);          // unit vector in AD

// Tension in cable EC by taking moment about line AD
Tec = -(!uAD*(G^W))/(!uAD*(C^uEC));

// Tension in cable BD by taking moment about line AC
Tbd = -(!uAC*(G^W))/(!uAC*(D^uBD));

// Reactions at A to balance all other forces
!Fa = -(W+Tec*uEC+Tbd*uBD);

print(^, Tec, Tbd, Fa);

5.8 A General Approach for Ad Hoc Solution

Most elementary engineering problems involve the solution of a set of simultaneous equations. All linear and non-linear simultaneous equations can be written in the following general form, with \( x = \langle x_1, x_2, \ldots, x_n \rangle \) as the unknowns:

\[
\begin{align*}
f_1(x_1, x_2, \ldots, x_n) &= 0 \\
f_2(x_1, x_2, \ldots, x_n) &= 0 \\
& \vdots \\
f_n(x_1, x_2, \ldots, x_n) &= 0
\end{align*}
\]

where \( f_i \) represents the left-hand-side of equation \( i \). Provided that \( f_i \) can be computed for any given \( x \) (i.e. they need be not be formed analytically), the solution vector \( x \) can be found with the built-in root1() or roots() function. This technique is particularly easy to apply because the unknowns need not be separated.
Wasting no time on number crunching, students can spend more time on the engineering aspects such as the modeling (e.g. formulation of the equations) and the impacts of design factors on the solutions.

**Sample Program 5.8 - General Equilibrium of a rigid body**

The rectangular plate of weight \( W \) is supported by the cable \( CE \) and the hinges \( A, B \). With the geometry and weight known, we can determine the 6 unknown forces (e.g. three reactions at \( A \), two at \( B \), and the cable tension \( T \)) from 6 equilibrium equations written for the plate (e.g. Resultant force \( \sum F = 0 \) and resultant moment \( \sum M_y = 0 \) about point \( O \)):

\[
\begin{align*}
\{ f_{x,y,z} \} &= \left\{ \begin{array}{l}
\sum F = A + B + T + W \\
\sum M_y = r_x \times A + r_y \times B + r_z \times T + r_{ov} \times W
\end{array} \right\} = \{ 0 \}
\end{align*}
\]

where \( A, B, T, W \) are the forces at points \( A, B, C \) and \( G \), respectively, and symbol \( r \) designates the position vectors of the points of application of the forces.

Let the unknowns \( X \) be selected as follows:

\[
A = \begin{bmatrix} X_1 = A_x \\ X_2 = A_y \\ X_3 = A_z \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ X_4 = B_y \\ X_5 = B_z \end{bmatrix}, \quad T = (X_6 = T) u_{CE}
\]
where $\mathbf{u}_{CE}$ is the unit vector along the cable $CE$. The known force is the weight of the plate:

$$
\mathbf{W} = \begin{bmatrix}
0 \\
-W \\
0
\end{bmatrix}
$$

The following program forms the equilibrium equations and solves for the unknown forces in $\mathbf{X}$. 

```c
main()
{

// ----- Sample data -------------------
defmat(rA[3], 0.3,0,0);  // Position vectors
defmat(rB[3], 1.5,0,0);
defmat(rC[3], 1.8, 0, 1.2);
defmat(rE[3], 0, 1.4, 0);
defmat(rG[3], 0.9, 0, 0.6);
defmat(W[3], 0, -1000, 0); // Applied force

// -------------------------------------
N = 6; // Number of equations
zero(f[N]); // Residual vector
defmat(X[N], N:1);    // Estimates for solutions
roots(X, f, FX(X, f)); // Solves for unknown {X}
print(^, X, FX(X,f));
}

FX(mat X, mat f)
{ // Form 6 equilibrium equations \{f\} = 0

defmat(A[3], X[1], X[2], X[3]); // Force vector at A
defmat(B[3], 0, X[4], X[5]);     // Force vector at B

!uCE = (CE = rE-rC)/hypot(!CE); // Unit vector CE
! T = X[6]*uCE;                // Cable force vector

!RF = A + B + T + W;          // Resultant force
!RM = rA^A + rB^B + rC^T + rG^W; // Resultant moment

defmat(f, RF, RM); // Residual vector
}
```

© 2015 by K. Ha - BCEE 231 Chapter 5 - v4.3
Sample Program 5.9 - Moment equilibrium of rigid body

The window plate $ABC$ (of mass $M = 25$ kg) is kept open at the angle $\theta$ corresponding to a mass $m$ attached to the end of the cable $CDE$ which passes over the frictionless pulley $D$.

It is desired to plot the variation of \( m \) vs the angle \( \theta \).

Consider the free-body diagram of the window plate: while there are 6 unknown force components at the two hinges \( A, B \), we can eliminate them by taking moment of the forces about the line \( AB \):

\[
M_{AB} = \mathbf{u}_{AB} \cdot \mathbf{r}_G \times \mathbf{W} + \mathbf{u}_{AB} \cdot \mathbf{r}_C \times \mathbf{T} = 0
\]

where \( \mathbf{u}_{AB} \) is the unit vector along \( AB \):

\[
\mathbf{u}_{AB} = \{ 0, 0, 1 \}
\]

\( \mathbf{W} \) is the window weight and \( \mathbf{T} \) is the cable tension:

\[
\mathbf{W} = \begin{bmatrix} 0 \\ -Mg \\ 0 \end{bmatrix} \quad \mathbf{T} = mg \mathbf{u}_{CD}
\]

\( \mathbf{u}_{CD} \) is the unit vector along the cable \( CD \).

The position vectors of \( G, C \) and \( D \) are:

\[
\mathbf{r}_G = \begin{bmatrix} a \sin \theta \\ \frac{-a \cos \theta}{2} \\ b \frac{1}{2} \end{bmatrix} \quad \mathbf{r}_C = \begin{bmatrix} a \sin \theta \\ -a \cos \theta \\ \frac{b}{2} \frac{1}{2} \end{bmatrix} \quad \mathbf{r}_D = \begin{bmatrix} d \\ h \\ 0 \end{bmatrix}
\]

In the following program implementation, function \( \text{Mab}(\text{float} \ m, \ \text{float} \ \Theta) \) returns the value of \( M_{AB} \) for any given \( m \) and \( \Theta \). The built-in function \( \text{root1()} \) may be used to solve for \( m \) or \( \Theta \) which makes \( M_{AB} \) equal to zero. For example, to solve for \( m \) given \( \Theta \):

\[
\text{root1}(m, 0, M, \text{Mab}(m, \Theta))
\]

and the plot of \( m \) vs \( \Theta \) is done by varying \( \Theta \):

\[
\text{plot}(\Theta, 0, 90, \text{root1}(m, 0, M, \text{Mab}(m, \Theta)))
\]

The complete program is shown below.
main()
{
// ----- Sample data ---------------
    a = 0.2; b = 0.25; d = 0.2; h = 0.1;
    M = 25; g = 9.81;
    useoption("DEGREES");
    clearplot();
    plot(Theta, 0, 90, root1(m, 0, M, Mab(m, Theta)));
}

Mab(float m, float Theta)
{
    // Return moment of forces about line AB
    float c = cos(Theta), s = sin(Theta), MAB;

    // Position vectors
    defmat(rG[3], a/2*s, -a/2*c, b/2);
    defmat(rC[3], a*s, -a*c, b/2);
    defmat(rD[3], d, h, 0);

    // Unit vectors
    defmat(uAB[3], 0, 0, 1);
    !uCD = (CD = rD - rC)/hypot(!CD);

    // External forces
    defmat(W[3], 0, -M*g, 0);
    !T = m*g*uCD;

    // Return moment of forces as scalar value
    return MAB = ! uAB * rG ^ W + uAB * rC ^ T;
}
Chapter 7

Systems of Linear Equations

While one person hesitates because he feels inferior, the other is busy making mistakes and becoming superior -- Henry C. Link
7.1 Linear Equations

Most problems in engineering analysis and design are formulated in the form of a set of linear simultaneous equations. The equations are linear in the unknowns $X_i$ when $X_i$ appears only as first degree terms in the equations. Here is an example of 3 equations linear in the unknown $X$'s:

$$
\begin{align*}
A_{11}X_1 + A_{12}X_2 + A_{13}X_3 &= B_1 \\
A_{21}X_1 + A_{22}X_2 + A_{23}X_3 &= B_2 \\
A_{31}X_1 + A_{32}X_2 + A_{33}X_3 &= B_3
\end{align*}
$$

(a1)

where $X_i$ are unknown while all other quantities are known. Once we have found $X_i$, we may compute quantities that depend on $X_i$ such as (see Sample Program 7.1):

$$
C_1X_1 + C_2X_2 + C_3X_3 \Rightarrow D
$$

(b1)

Consistency of Units of Input Data

In numerical computations, the units of data must be consistent.

For example, once the unit of force is chosen, it dictates the units to be used for length, mass and time:

<table>
<thead>
<tr>
<th>If forces are in kN ($= 1000$ kg.m/s$^2$), then</th>
<th>The unit of</th>
<th>must be in</th>
<th>because</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>m</td>
<td>m is used in N</td>
<td></td>
</tr>
<tr>
<td>Area</td>
<td>m$^2$</td>
<td>m is used in N</td>
<td></td>
</tr>
<tr>
<td>Moments of inertia</td>
<td>m$^4$</td>
<td>m is used in N</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>s</td>
<td>s is used in N</td>
<td></td>
</tr>
<tr>
<td>Mass</td>
<td>kg</td>
<td>kg is used in N</td>
<td></td>
</tr>
<tr>
<td>Elasticity modulus, Pressure, Distributed force over area</td>
<td>kPa</td>
<td>1Pa = 1N/m$^2$</td>
<td></td>
</tr>
<tr>
<td>Moments, torques</td>
<td>kN.m</td>
<td>m is used in N</td>
<td></td>
</tr>
<tr>
<td>Distributed force over a line</td>
<td>kN/m</td>
<td>m is used in N</td>
<td></td>
</tr>
</tbody>
</table>

$^1$ Terms containing $1/X$, $X^2$, $X^3$, sin($X$), cos($X$), $e^X$, etc.. are non-linear in $X$. 
Consistency of Units in Equations

- Addition and subtraction of terms is possible only if the terms are all in the same unit\(^2\); e.g. \(A_{11}X_1, A_{12}X_2, A_{13}X_3, B_1\) all have the same units.

- In an equality, the units of the two sides must be the same.

Unless consistency of units is ensured, mathematical operations performed on the equations will only lead to meaningless numbers. Expression or formula that results in a unit inconsistent with the physical quantity cannot be correct.

Example: In the following equations:

\[
\begin{align*}
A_{11}X_1 + A_{12}X_2 + A_{13}X_3 &= B_1 \\
A_{21}X_1 + A_{22}X_2 + A_{23}X_3 &= B_2 \\
A_{31}X_1 + A_{32}X_2 + A_{33}X_3 &= B_3 \\
C_1X_1 + C_2X_2 + C_3X_3 &= D
\end{align*}
\]

if the units of \(X, B,\) and \(D\) are

<table>
<thead>
<tr>
<th>Variables</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_1)</td>
<td>[kN]</td>
</tr>
<tr>
<td>(X_2)</td>
<td>[m]</td>
</tr>
<tr>
<td>(X_3)</td>
<td>[kN.m]</td>
</tr>
<tr>
<td>(B_1)</td>
<td>[kN.m]</td>
</tr>
<tr>
<td>(B_2)</td>
<td>[m]</td>
</tr>
<tr>
<td>(B_3)</td>
<td>[Radians]</td>
</tr>
<tr>
<td>(D)</td>
<td>[$]</td>
</tr>
</tbody>
</table>

What must be the units of the coefficients \(A_{ij}\) and \(C_j\)?

Answer: The following table shows the units of \(A_{ij}\) and \(C_j\) as required for the consistency of units among those quantities. For example, since the product \(A_{11}X_1\) must give the unit of \(B_1\), the unit of \(A_{11}\) must be meter.

\(^2\) Since all terms in a power series such as

\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots
\]

have the same unit, \(x\) must be in radians (which is dimensionless).
Standard Form for Computer Solution of Linear Equations

Expressing Equations (a1, b1) in matrix form:

\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= \begin{bmatrix}
B_1 \\
B_2 \\
B_3
\end{bmatrix}
\Rightarrow A_{3\times3}X_{3\times1} = B_{3\times1}
\] (a2)

\[
< C_1, C_2, C_3 >
\begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= D
\Rightarrow C_{1\times3}X_{3\times1} = D
\] (b2)

where the known coefficients are grouped into arrays \(A\), \(B\) and \(C\), while the unknowns into \(X\). The storage for the unknown vector \(X\) is reserved as a 1-D array ‘X’.

Note that the rule of matrix product ensures that Eqs.(a2, b2) are mathematically equivalent to Eqs.(a1, b1).

Eq. (a2) is the standard form for computer solution of linear equations. Here are some useful terms:

- Matrix \(A\) is the \textit{coefficient matrix}. It is a square matrix.
- Vector \(B\) is the \textit{right-hand side vector}. Many such vectors may be arranged as columns of matrix \(B\).
- Vector \(X\) is the \textit{solution vector}. If \(B\) is a matrix of many columns, then \(X\) is the corresponding matrix of solutions to be found.

For computer solution, matrices \(A\) and \(B\) must be given as data, and the solution may be returned either in a separate vector \(X\) or in \(B\) itself.
Sample Program 7.1 - Solution of Well-Conditioned Systems

Consider now a problem in the engineering context: Find the parameters $X_i$ and the total cost $D$ from the following relations among the variables:

\[
\begin{bmatrix}
  3 \text{ m} & 2 \text{ kN} & 0 \\
-1 \text{ m/kN} & -2 & 4 \text{ m/kN} \\
2 \text{ kN}^{-1} & -1 \text{ m}^{-1} & -3 \text{ kN}^{-1} \text{m}^{-1}
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= \begin{bmatrix}
20 \text{ kN} \text{m} \\
2 \text{ m} \\
0.01 \text{ Radian}
\end{bmatrix}
\] (a4)

\[
< 1240 \$/\text{kN}, \quad 546 \$/\text{m}, \quad 450 \$/\text{kN} \cdot \text{m} > \begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= D
\] (b4)

**Matrix format:**

For numerical computation, we remove the units, assuming that they are consistent:

\[
\begin{bmatrix}
3 & 2 & 0 \\
-1 & -2 & 4 \\
2 & -1 & -3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= \begin{bmatrix}
20 \\
2 \\
0.01
\end{bmatrix}
\Rightarrow
A_{3 \times 3} X_{3 \times 1} = B_{3 \times 1}
\] (a5)

\[
< 1240, \quad 546, \quad 450 > \begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
= D \Rightarrow C_{1 \times 3} X_{3 \times 1} = D_{1 \times 1}
\] (b5)

**Computation:**

```c
main()
{
    defmat(A[3,3], 3, 2, 0, -1, -2, 4, 2, -1, -3);
    defmat(B[3], 20, 2, 0.01 );
    defmat(C[3], 1240, 546, 450);
    !X = (Ai = A^-1) * B; // Note 1
}```
// solve(S, !A2 = A, !X = B); // Note 2
D = !C*X; // Scalar D
! R = A*X-B; // Residual vector -- Original A, B
print(Ai, X, ^^, "  D = ", D, ^^, R, ^^,
"   Length of R = ", hypot(!R));
}

Notes

1. Ai is the inverse of A, obtained by A^-1. Matrix A remains unchanged.

2. solve() is a better way for finding X. Here, [A] is copied into [A2] which will be altered by solve(). Similarly, {B} is copied into {X} which will contain the solution. Such copying is needed only if the original [A] and {B} are to be preserved for later computation of the residual vector.

Interpretation of results

The program prints numerical solution, and it is up to us to attach the units:

\[
X = \begin{bmatrix}
5.302 \text{kN} \\
2.047 \text{m} \\
2.849 \text{kN.m}
\end{bmatrix}
\]

Total cost \( D = \$8974.19 \)

Since the residual vector \( R = AX - B \) is practically a null vector, \( X \) is the "exact" solution. The solution found by matrix inverse or solve() is both "exact" and unique. This is true for well-conditioned systems.

Why Bother with Matrix Operations?

Matrix provides a convenient means and a compact form for the arrangement and storage of data as well as for doing operations on them. The results in compact form facilitate further deployment.

Equivalent scalar presentation/operations are possible; however, they are so tedious and clumsy that they tend to:

(a) Conceal the meaning in a big mess of details;
(b) Prohibit computerized computation;
(c) Inhibit further deployment of the results.

7.2 The Inverse of a Square Matrix

Given the linear system:
\[
\begin{bmatrix}
3 & 2 & 0 \\
-1 & -2 & 4 \\
2 & -1 & -3
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
=
\begin{bmatrix}
20 \\
2 \\
0.01
\end{bmatrix}
\Rightarrow
A_{3\times3}X_{3\times1} = B_{3\times1}
\]

where \( A \) is a square matrix. The inverse of a square matrix \( A \) is a square matrix (symbolically denoted by \( A^{-1} \)) such that:
\[
A_{3\times3}A^{-1}_{3\times3} = A^{-1}_{3\times3}A_{3\times3} = I_{3\times3}
\]

where the identity matrix \( I \) contains only unit values along its main diagonal, and zero elsewhere.

i.e.
\[
\begin{bmatrix}
3 & 2 & 0 \\
-1 & -2 & 4 \\
2 & -1 & -3
\end{bmatrix}
\begin{bmatrix}
0.25 & 0.15 & 0.2 \\
0.125 & -0.225 & -0.3 \\
0.125 & 0.175 & -0.1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Thus:
\[
A^{-1} = \begin{bmatrix}
0.25 & 0.15 & 0.2 \\
0.125 & -0.225 & -0.3 \\
0.125 & 0.175 & -0.1
\end{bmatrix}
\]

The matrix inverse \( A^{-1} \) exists only when the matrix \( A \) is non-singular (or well-conditioned).

The identity matrix \( I \)

The square matrix \( I \), which contains 1 along its main diagonal and 0 elsewhere, is called the identity matrix or unit matrix because its product with any matrix \( Q \) of compatible order gives the same matrix \( Q \):
This marvellous property of the matrix inverse allows us to convert

\[ A X = B \]

into (by multiplying both sides with \( A^{-1} \)):

\[
A^{-1}_{3\times3} (A_{3\times3} X_{3\times1}) = A^{-1}_{3\times3} B_{3\times1}
\]

The left-hand side may be reduced to

\[(A^{-1}_{3\times3} A_{3\times3}) X_{3\times1} = I_{3\times3} X_{3\times1} = X_{3\times1}\]

giving the desired solution as:

\[ X = A^{-1} B \]

**Units of the elements of the matrix inverse**

The units of the elements in \( A^{-1} \) are deduced from the fact that the product \( A^{-1} B \) must yield \( X \) as shown below:

<table>
<thead>
<tr>
<th>( B_1 ) = 20 kN.m</th>
<th>( B_2 ) = 2 m</th>
<th>( B_3 ) = 0.01 Rad</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25 m(^{-1})</td>
<td>0.15 kN.m(^{-1})</td>
<td>0.2 kN</td>
</tr>
<tr>
<td>0.125 kN(^{-1})</td>
<td>-0.225</td>
<td>-0.3 m</td>
</tr>
<tr>
<td>0.125</td>
<td>0.175 kN</td>
<td>-0.1 kN.m</td>
</tr>
</tbody>
</table>

**7.3 Solving Linear Equations**

We will use simple problems to illustrate a variety of interesting possibilities in solving linear equations. The issues are:

- **Does the solution exist?**
  Ans: There may be one (unique) solution, no solution, or infinite solutions. In the case of no solution, we may still find the best fit solution.
- **Is the solution unique?**
  Ans: Solution of a well-conditioned system is unique
- **Is the computed solution the desired one?**
  Ans: That depends on the range of solutions acceptable to the physical problem.
- **How can the equation system be ill-conditioned?**
  Ans: It may happen when the number of unknowns is different from the number of independent equations.
Could the formulation be responsible for bad solution?  
Ans: It's possible, but rare.

How well does the mathematical solution fit reality?  
Ans: Pretty well if formulated and solved correctly.

These issues will be illustrated using the following simple problem.

**Example problem**

Objective: Find the equation of the straight line that “passes” through a number of given points (see Fig. 1).

![Figure 1](https://via.placeholder.com/150)

**Formulation for solution**

The equation for a straight line in the $xy$-plane has the form

$$y = \alpha x + \beta$$

where the vector of unknowns is $Z = \{ \alpha, \beta \}$. We wish to determine the values of $\alpha, \beta$ from the given points.

For each given point at the coordinates $\{ x_i, y_i \}$, we may write one equation:

$$y_i = \alpha x_i + \beta \quad i = 1, 2, \ldots N$$

where $N$ is the number of given points. Thus, there will be $N$ such linear equations which are to be solved for $\{ \alpha, \beta \}$.
As demonstrated subsequently, we may have the following possibilities depending on the number of points and their locations:

- One unique solution
- An infinite set of solutions
- No solution
- Best solution in the least-squares sense

### 1. Unique Solution

Given any two distinct points (such as C, D in Fig. 1), there is only one line that passes through them. Thus, the solution for \( \mathbf{Z} = \{ \alpha, \beta \} \) must be unique for such case.

Formulation: Given 2 points C and D, substitute into \( y = \alpha x + \beta \) gives

Point C(1,2):
\[
2 = \alpha(1) + \beta
\]
Point D(2.5, 4):
\[
4 = \alpha(2.5) + \beta
\]

In standard form:

\[
\begin{bmatrix}
1 & 1 \\
2.5 & 1
\end{bmatrix}
\begin{bmatrix}
\alpha \\
\beta
\end{bmatrix}
= 
\begin{bmatrix}
2 \\
4
\end{bmatrix}
\text{ or } \mathbf{A} \mathbf{Z} = \mathbf{B}
\]

The exact and unique solution can be verified to be:

\[
\mathbf{Z} = \begin{bmatrix}
\alpha \\
\beta
\end{bmatrix} = \begin{bmatrix}
4/3 \\
2/3
\end{bmatrix}
\]

Thus, the straight line CD has the equation:

\[
y = \frac{4}{3} x + \frac{2}{3}
\]

Program for solution:
```c
main()
{
   // Solution of well-conditioned systems
   defmat(A[2,2], 1, 1, 2.5, 1);
   defmat(B[2], 2, 4);

   !Z = A^-1 * B;    // By matrix inversion
   // solve(S, !AC = A, !Z=B);    // By solve()
   !R = A*Z - B;    // Residual vector
   print(A, B, Z, R, ^^, " Length R:", hypot(!R));
}
```

The residual vector

The residual vector \( \mathbf{R} = \mathbf{A}\mathbf{Z} - \mathbf{B} \) is a measure of the accuracy of the solution. Exact solution gives residuals of zero. The length of the residual vector given by \( \text{hypot}(\mathbf{R}) \) is a more convenient measure of the error.

**General conclusions:**

- When the coefficient matrix \( \mathbf{A} \) is non-singular (i.e. \( \mathbf{A}^{-1} \) exists), or when function `solve()` succeeds, the computed solution is unique. The residual vector of near zero length indicates good accuracy.

- In addition, if the right-hand side vector \( \mathbf{B} \) is zero\(^3\), the unique solution is also 0, which we call the trivial solution\(^4\).

**2. Infinite Set of Solutions**

Given only one point, we have only one equation for finding 2 unknowns. There are an infinite set of solutions representing infinite number of lines passing through the point. Thus, the solution for \( \mathbf{Z} = \{\alpha, \beta\} \) must be an infinite set.

\(^3\) When the right-hand side is a null vector, the resulting system (i.e. \( \mathbf{A}\mathbf{Z} = 0 \)) is called homogeneous equations.

\(^4\) This hints at the possibility of a homogeneous system (\( \mathbf{A}\mathbf{Z} = 0 \)) that may have non-trivial solutions (\( \mathbf{Z} \neq 0 \)) when \( \mathbf{A} \) is singular.
When there are more unknowns than equations, the equation system is called *underdetermined*.

**Formulation:** Given only one point, say C, substitute into $y = \alpha x + \beta$ gives

Point C(1,2): $2 = \alpha(1) + \beta$

Rewriting the equation in the standard form for two unknowns:

$$
\begin{bmatrix}
1 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
\alpha \\
\beta
\end{bmatrix}
= 
\begin{bmatrix}
2 \\
0
\end{bmatrix}
$$
or $A Z = B$

Note that we add the second equation ($0 \alpha + 0 \beta = 0$ ) in order to make $A$ a square matrix. The vector $Z$ that satisfies this equation can be verified to be:

$$Z = \begin{bmatrix}
\alpha \\
\beta
\end{bmatrix} = \begin{bmatrix} 1+\gamma \\
1-\gamma
\end{bmatrix}$$

where $\gamma$ is arbitrary, and hence this solution is an infinite set. The standard way to express this solution is:

$$Z = \begin{bmatrix} 1 \\
1
\end{bmatrix} + \gamma \begin{bmatrix} 1 \\
-1
\end{bmatrix} = Z_p + \gamma Z_h$$

$Z_p = \{ 1, 1 \}$ is the vector of *particular solution* of $AZ_p = B$

$Z_h = \{ 1, -1 \}$ is the solution of the *homogeneous equation* $A Z = 0$ and hence called the *homogeneous solution*.

Since $\gamma$ is arbitrary, the vector of homogeneous solution $Z_h$ is often “normalized” so that its length is 1. The complete solution is then:

---

5 If $Z = Z_p + \gamma Z_h$ is the solution, its substitution into $AZ = B$ gives

$$A Z_p + \gamma A Z_h = B$$

Since $\gamma$ is arbitrary while the right-hand-side is constant, we must have $A Z_h = 0$ and $A Z_p = B$. Thus $Z_p$ is the particular solution while $Z_h$ is the homogeneous solution.
\[ Z = \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix} + \gamma \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} \]

Program for solution:

For this data set of \( A \) and \( B \), the standard methods for solving well-conditioned equations fail because matrix \( A \) is now singular (i.e. \( A^{-1} \) does not exist).

In such case, the best tool for getting both solutions \( Z_p \) and the normalized homogeneous solution \( Z_h \) is by the method of singular-value decomposition via the built-in functions \texttt{svd()} and \texttt{svsolve()} as shown below.

```c
main()
{
    // Solution by singular value decomposition
    // This technique works for any matrix A regardless of
    // its condition.
    defmat(A[N = 2,N], 1, 1, 0, 0);
    defmat(B[N], 2, 0);
    SolveSVD();
}
SolveSVD()
{
    !U = A; // U is a copy of A
    Cond = \texttt{svd}(U,W,V); // Note 1
    if(Cond == i#)
    {
        print("Matrix A is singular");
    }
    \texttt{svsolve}(U,W,V,B,Z); // Note 2
    !R = A*Z - B; // Residual vector
    print("Particular solution vector:", Z, "
          Length of residual vector:", hypot(!R));
    print("Homogeneous solution is in column k of V ",
          "where W[k] is zero", W, V);
}
```
Notes

1. Function svd() accepts $A$, stored as $U$, and computes 3 matrices $U$, $W$, $V$ such that:

$$A = U W V^T$$

where $W$ is a diagonal matrix. Using the output matrices, we can verify that:

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1.414 \end{bmatrix} \begin{bmatrix} 0.7071 & -0.7071 \\ 0.7071 & -0.7071 \end{bmatrix}^T$$

The values in the diagonal of $W$ are called the singular values of matrix $A$. This decomposition is called singular value decomposition.

Function svd() returns the condition number of matrix $A$. The larger the condition number, the more ill-conditioned the matrix $A$ is. When $A$ is singular, svd() returns infinity ($i#$).

2. Function svsolve() solves for $Z$ from the linear system $A Z = B$ where $A$ is represented by its three decomposed components $U$, $V$, $W$ previously computed by svd().

Output

Matrix $A$ is singular

Particular solution vector:

Matrix ..Z

$$\begin{bmatrix} 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

Length of residual vector: $4.44089e-016$

Homogeneous solution is in column $k$ of $V$ where $W[k]$ is zero

Matrix ..W

$$\begin{bmatrix} 1 & 2 \\ 1 & 0 & 1.414 \end{bmatrix}$$

Matrix ..V

$$\begin{bmatrix} 1 & 2 \\ 1 & 0.7071 & -0.7071 \\ 2 & -0.7071 & -0.7071 \end{bmatrix}$$
Complete solution

Since the singular value $W[1]$ is zero, column 1 of $[V]$ is the homogeneous solution $Z_h$.

Thus, the complete solution set is $Z = Z_p + \gamma Z_h$

$$Z = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \gamma \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}$$

General conclusions:

- When the coefficient matrix $A$ is singular (i.e. $A^{-1}$ does not exist), or when function `solve()` fails, there is no unique solution.\(^6\)

- In the case of a underdetermined (consistent) system, the set of solutions is infinite, and the technique of singular value decomposition is a very effective tool for getting the complete solution as

$$Z = Z_p + \sum_k \gamma_k Z_{hk}$$

where $\gamma_k$'s are arbitrary factors. For every $W[k]$ that is zero, the homogeneous solution $Z_{hk}$ is the column $k$ of $V$. The residual vector for $Z_{hk}$ should be nearly zero since the solution is “exact”.

3. A Case of No Solution

Given 3 or more distinct, general points (such as C, D, E in Fig. 1), there is no straight line that could pass through them all.\(^8\) Thus, the solution for $Z = \{\alpha, \beta\}$ does not exist.

---

\(^6\) The equilibrium equation of linear structural systems has the form $A_{m \times n}X_{n \times 1} = B_{m \times 1}$ where $X$ is the vector of unknown internal forces, and $B$ is the vector of external forces. If matrix $A$ is rectangular with $m < n$, the number of equations is less than the number of unknowns, and the system is said to be statically indeterminate. For such systems, there is an infinite set of internal member forces that satisfy equilibrium equations. And hence, equilibrium is not sufficient for getting the unique solution of indeterminate systems.

\(^7\) If the system is inconsistent, there is no solution.

\(^8\) If we are given colinear points (e.g. C, D and G), the unique solution does exist (See Practice Drill 7.1.1) because the equations are consistent, albeit overdetermined.
Formulation: Given 3 points C, D, E, substitute into $y = \alpha x + \beta$ gives

Point C(1, 2): \[2 = \alpha(1) + \beta\]
Point D(2.5, 4): \[4 = \alpha(2.5) + \beta\]
Point E(3, 3.2): \[3.2 = \alpha(3) + \beta\]

In standard form:
\[
\begin{bmatrix}
1 & 1 & 0 \\
2.5 & 1 & 0 \\
3 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
\alpha \\
\beta \\
\phi \\
\end{bmatrix} =
\begin{bmatrix}
2 \\
4 \\
3.2 \\
\end{bmatrix}
\]

The third column of zeroes is added in order to make $A$ square. Matrix $A$ is singular (i.e. $A^{-1}$ does not exist) and the function `solve()` also fails because it encounters a division by zero. Thus, it is impossible to find 2 unknowns {$\alpha, \beta$} that will satisfy the three independent equations. The system is called overdetermined\(^9\); it has no solution if it is also inconsistent as in the present case.

However, the straight line that best fits the three points will be found as:

\[y = 0.7692x + 1.4\]

In other words, the solution that minimizes the residual error\(^{10}\) is

\[Z = \begin{cases} 
\alpha = 0.7692 \\
\beta = 1.4 
\end{cases}\]

Program for least-squares-fit\(^{11}\) solution:

When the coefficient matrix $A$ is singular, we resort to `svd()` and `svsolve()` for solution. The solution will be the least-squares-fit solution, and in general, it does not satisfy the equation $A \cdot Z = B$ exactly.

```
main() { // Best-fit solution by singular value decomposition
    defmat(A[N = 3,N], 1, 1, 0, 2.5, 1, 0, 3, 1, 0);
defmat(B[N], 2, 4, 3.2 );
}
```

\(^9\) An overdetermined system has more independent equations than unknowns. It becomes generally impossible for any solution vector to satisfy all equations.

\(^{10}\) For a given vector $Z$, the residual error of $A Z - B = 0$ is the length of the vector $A Z - B$.

\(^{11}\) [https://en.wikipedia.org/wiki/Least_squares](https://en.wikipedia.org/wiki/Least_squares)
SolveSVD();
}

SolveSVD()
{
!U = A; // U is a copy of A
Cond = svd(U,W,V);      // Note 1
if(Cond == i#)
{
    print("  Matrix A is singular");
}
svsolve(U,W,V, B, Z);     // Note 2
!R = A*Z - B;   // Residual vector
print("  Particular solution vector:", Z, ^
      "  Length of residual vector:", hypot(!R));

print("  Homogeneous solution is in column k of V ",^
      "  where W[k] is zero",^, W, V);
}

Complete solution

The complete solution is $\mathbf{Z} = \mathbf{Z}_p + \gamma \mathbf{Z}_h$

\[
\mathbf{U} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0.7692 \\ 1.4 \end{bmatrix} + \gamma \begin{bmatrix} 0 \\ 0 \end{bmatrix}
= \begin{bmatrix} 0.7692 \\ 1.4 \end{bmatrix}
\]

Since the homogeneous solution, $\mathbf{Z}_h$, is zero, the above solution is unique, and it is the best fit solution that minimizes the residual error\(^{12}\). Note that the residual vector is not zero.

\(^{12}\) While the proof for this is beyond the scope of this course, we can do numerical experiments to confirm the conclusion.
General conclusions:

- When the coefficient matrix $A$ is singular (i.e. $A^{-1}$ does not exist), or when function solve() fails, the solution may not exist. In the case of overdetermined (inconsistent) system, there is no solution, and the technique of singular value decomposition will provide the least-squares-fit solution.

- The complete solution is, in general:
  $$Z = Z_p + \sum k \gamma_k Z_{hk}$$

- If the homogeneous solution $Z_h$ is zero, the particular solution is the unique least-squares-fit solution, otherwise, there is a family of least-squares-fit solutions.

- In rare cases, it might happen that the overdetermined system is well-conditioned and consistent; the solution will be both exact and unique (e.g. One unique line passes through 3 points C, D, G. See Practice Drill 7.1.1).

4. Bad Solution due to Poor Formulation

A formulation may fail to yield the desired solution. For example, the formulation based on $y = \alpha x + \beta$ through the two points C and F (Fig. 1) leads to the following set of equations:

$$\begin{align*}
\alpha + \beta &= 2 \\
\alpha + \beta &= 4
\end{align*}$$

or

$$\begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}\begin{bmatrix}
\alpha \\
\beta
\end{bmatrix} = \begin{bmatrix}
2 \\
4
\end{bmatrix}$$

Figure 7.2 The family of lines that best-fit points C and F

Here, the two equations are inconsistent or contradictory. Thus, the formulation based on $y = \alpha x + \beta$ turns out to be a very poor
formulation\(^\text{13}\) for this well-conditioned problem. Practice Drill 7.1.2 shows a better formulation for this case.

Even for such an inconsistent set of equations, the technique of singular-value decomposition still gives solutions which represent the best-fit solutions (Fig.7.2). This family of solutions, however, is far from the desired solution.

**General conclusions:**

- The powerful technique of singular-value decomposition always give some kind of solution which, however, may not be the desired solution.
- A poor formulation of a well-conditioned problem will still give bad solution regardless of the technique for solving linear equations. Thus, we should be aware of the limitations of the formulation.
- When in doubt, substitute the computed solution into the original equations in order to get the residual vector for inspection.
- Inconsistent equation system, be it under- or over-determined, generally has no exact solution. SVD will then give the best-fit solution.

**Practice Drills 7.1**

1. If we use the three points C, D and G in Fig.1, the exact and unique solution for the straight line \(y = \alpha x + \beta\) is:

\[
Z = \begin{bmatrix}
\alpha \\
\beta
\end{bmatrix} = \begin{bmatrix}
4/3 \\
2/3
\end{bmatrix}
\]

The resulting linear system \(A_{3x3} Z_{3x1} = B_{3x1}\) is overdetermined and consistent. Write the program to verify:

(a) The above solution \(Z\).
(b) The computed solution \(Z\) satisfies all three equations and unique.

\(^{13}\) Since the line CD is parallel to the y-axis, its slope with respect to the x-axis is infinity (i.e. \(\alpha\) is infinity). There is no set of \(\alpha, \beta\) that will lead to the correct solution of \(x = 1\).
2. Given the two points C and F in Fig.1, the line CF is unique and well defined. Its equation is \( x = 1 \).

   (a) If we pass the straight line \( y = \alpha x + \beta \) through these two points, show that the standard form of the resulting linear system is:

   \[
   \begin{bmatrix}
   1 & 1 \\
   1 & 1
   \end{bmatrix}
   \begin{bmatrix}
   \alpha \\
   \beta
   \end{bmatrix}
   =
   \begin{bmatrix}
   2 \\
   4
   \end{bmatrix}
   \]

   Write a program to obtain the “solution” for this system. What is the meaning of this “solution”?

   (b) Now if we pass the straight line \( x = \alpha y + \beta \) through these two points, formulate the linear equation system and write a program to compute the unique and exact solution.

3. Write the computer programs to determine the solution for the following underdetermined inconsistent system of equations:

   \[
   \begin{bmatrix}
   3 & 7 & -2 & 3 \\
   -1 & -2 & 4 & -3 \\
   -2 & -4 & 8 & -6 \\
   0 & 0 & 0 & 0
   \end{bmatrix}
   \begin{bmatrix}
   X_1 \\
   X_2 \\
   X_3 \\
   X_4
   \end{bmatrix}
   =
   \begin{bmatrix}
   4 \\
   5 \\
   9 \\
   0
   \end{bmatrix}
   \]

   Ans: Infinite set of least-squares-fit solutions:

   \[
   \begin{pmatrix}
   0.3764 \\
   1.037 \\
   1.332 \\
   -0.5735
   \end{pmatrix}
   +
   \begin{pmatrix}
   0.927 \\
   -0.3655 \\
   -0.01329 \\
   -0.08307
   \end{pmatrix}
   \gamma_1
   \]

   \[
   +
   \begin{pmatrix}
   0 \\
   0.2074 \\
   -0.01329 \\
   -0.8296
   \end{pmatrix}
   \gamma_2
   \]

4. Write one or more computer programs to determine the solution vector for the following systems of equations \( \mathbf{A}\mathbf{X} = \mathbf{B} \). The programs should also output the residual vector \( (\mathbf{A}\mathbf{X}-\mathbf{B}) \) using the computed solution vector \( \mathbf{X} \).

   (a) \[
   \begin{bmatrix}
   3 & 7 & -2 & 3 \\
   -1 & -2 & 4 & -3 \\
   2 & -5 & -2 & 2 \\
   -6 & 3 & -1 & 2
   \end{bmatrix}
   \begin{bmatrix}
   X_1 \\
   X_2 \\
   X_3 \\
   X_4
   \end{bmatrix}
   =
   \begin{bmatrix}
   4 \\
   5 \\
   9 \\
   -10
   \end{bmatrix}
   \]
Sample Program 7.2 - Least Squares Fit and Linearization

Given a set of 10 data points:

\[
x[N = 10]: \quad <0.5, 1.2, 1.8, 2.2, 3.4, 4.0, 5.0, 6.3, 8.0, 9.5>
\]
\[
y[N]: \quad <0.92, 1.6, 2.4, 2.8, 3.1, 3.7, 3.9, 4.1, 4.4, 4.5>
\]

our objective is to find the coefficients \( c_1, c_2 \) of the model equation

\[
y = c_1 \frac{x}{c_2 + x}
\]

to best fit the above given data.

Linearization:

Since the model equation is non-linear in the unknowns \( c \)'s, we first try to linearize the model equation:

\[
\frac{c_2}{x} + 1 = c_1 \frac{1}{y} \quad \Rightarrow \quad \frac{1}{y} = \left( \frac{c_2}{c_1} \right) \frac{1}{x} + \frac{1}{c_1}
\]

With the substitution \( Y = \frac{1}{y}, \ X = \frac{1}{x}, \ \alpha = \frac{c_2}{c_1}, \ \beta = \frac{1}{c_1} \) the model equation becomes linear (in the unknowns \( \alpha, \beta \)):

\[
Y = \alpha X + \beta
\]
With the above linear model equation, we apply standard least-squares-fit procedure for the transformed data set \( \{X\}, \{Y\} \) to find \( Z = \{\alpha, \beta\} \), and then the actual model coefficients can be computed next:

\[
c_1 = \frac{1}{\beta} = \frac{1}{Z[2]}, \quad c_2 = c_1\alpha = \frac{Z[1]}{Z[2]}
\]

The following program also plots the best-fitted curve together with the data points for a visual inspection of the solution.

```c
main()
{
    defmat(x[N = 10], 0.5,1.2,1.8,2.2,3.4,4,5,6.3,8,9.5);
    defmat(y[N], 0.92,1.6,2.4,2.8,3.1,3.7,3.9,4.1,4.4,4.5);

    zero(X[N], Y[N]); // Linear model \( Y = \alpha X + \alpha \)
    float I;
    // Linearization by data transformation
    for(I=1; I<=N; I=I+1)
    {
        X[I] = 1/x[I];
        Y[I] = 1/y[I];
    }

    SetUpEquation();
    SolveSVD();

    print("^", " c1 = ", c1=1/Z[2]," c2 = ",c2 = Z[1]/Z[2]);
    DrawData();
}

SetUpEquation()
{ // Set up \([A]\) and \([B]\) of system \([A] \{Z\} = \{B\}\)
    // \( Z = \{\alpha, \beta\} \)
zero(A[N,N], B[N]);
float I;
for(I=1; I<=N; I=I+1)
{
    A[I,1] = X[I];
    A[I,2] = 1;
    B[I] = Y[I];
}
}

SolveSVD()
{
    !U = A;
    Cond = svd(U,W,V);
    svsolve(U,W,V, B, Z);
    print(W, V);
    !R = A*Z - B; // Residual vector
    print(" Particular solution:", Z, " Length of residual vector:", hypot(!R));
}

f(float x)
{ // Model equation
    return c1 * x/(c2+x);
}

DrawData()
{
    float i, xi, yi;
    clearplot();
    plot(t, x[1], x[N], f(t)); // Fitted curve
    setop(C,7); // Change color
for(i=1; i<=N; i=i+1)
{
    // Put a square at each data point
    xi = x[i];
    yi = y[i];
    putxy(xi, yi, "SQUARE"); // Put square dot at data point
    // Draw vertical line from data point to fitted curve
    plotline(xi, yi, xi, f(xi));
}

Practice Drills 7.2

1. Develop and test a program that finds the coefficients $a$, $b$ and $c$ of the parabola $y = ax^2 + bx + c$ in order to best fit the following set of data.

   \[
   \begin{array}{|c|c|c|c|c|}
   \hline
   x & 1 & 2 & 3 & 4 \\
   \hline
   y & -12 & -6 & 15 & 20 \\
   \hline
   y & 612.1 & 219.9 & 114.4 & 399.9 \\
   \hline
   \end{array}
   \]

   N=5

2. The given truss is subject to the joint loads $R$ as shown.

   \[
   \begin{align*}
   R_1 &= 0.8F_1 - 0.8F_4 \\
   R_2 &= 0.6F_1 + F_3 + 0.6F_4 \\
   R_3 &= F_2 - F_3 \\
   R_4 &= -F_3 \\
   R_5 &= 0.8F_4 + F_5
   \end{align*}
   \]
where $F_i$ is the axial force in member $i$.

(b) Write the program to compute the member forces in the truss which is subject to the particular loading of $\mathbf{R} = \{10 \text{ kN}, -20 \text{ kN}, 0, -5 \text{ kN}, 0\}$. 
Chapter 9

Graphs, Derivatives, and Integrals of Functions of One Variable

In the fall of 1972 President Nixon announced that the rate of increase of inflation was decreasing. This was the first time a sitting president used the third derivative to advance his case for reelection -- Hugo Rossi
Calculus is the most important mathematical tool for the development and application of advanced engineering sciences. The treatment here is informal and less rigorous than normally seen in calculus courses. The emphasis is on basic understanding of the meaning and application of calculus.

9.1 Numerical Derivatives

Understanding the properties of derivatives is much more important than remembering all those formulas for differentiation. After all, even without formulas, we can still compute the value of the derivative (if it exists) at a given point of any function.

Definition:

Derivative at $x_0$

- Standard definition: $\frac{dy}{dx} \bigg|_{x=x_0} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$

- Numerical computation with suitably small $\Delta x$:

$$\frac{\Delta y}{\Delta x} \bigg|_{x=x_0} = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

$$D = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$  \hspace{1cm} (1)
where $Dx$ is a suitably small value, and $f(x)$ is an expression in terms of $x$.

A more accurate result is often given by the central derivative formula (if the function is continuous and smooth at $x_0$):

$$D = \frac{(f(x_0+Dx/2) - f(x_0-Dx/2))/Dx}; \quad (2)$$

If $Dx$ is excessively small, the value of the numerator will loose significant digits because of the subtraction of two very close values.

- By built-in function:
  - `deriv(x, xo, f(x));`  // 2-point central derivative
  - `deriv(x^3, xo, f(x));`  // 3-point central derivative
  - `deriv(x^5, xo, f(x));`  // 5-point central derivative

**Sample Program 9.1 - Computation of First Derivatives**

Given the function $f(x)$ and its exact first derivative:

$$f(x) = e^{\sin x}, \quad \frac{df}{dx} = 2\sin x \cos x \ e^{\sin x}$$

the following program investigates the accuracy of the first derivatives computed using the right derivative formula Eq.1, and central derivative formula Eq.2. Decreasing interval $Dx$ is used while the results are compared to the exact value.

```plaintext
clearplot();
useoption("RADIANS");
plot(x, 0, pi#, f(x));
float Dx = 0.01;
E = ED(xo); // Exact derivative
B = deriv(x, xo, f(x)); // 2-point built-in
print(^," xo =", xo, " Exact =", E,
      " deriv(): ", B, " %Error =", PE(B, E));
```

© 2015 K. H. Ha – BCEE 231 Chapter 9 - v4
\begin{verbatim}
print("        Dx         RD%        CD%",";
  do
  { 
    print("Dx",PE(RD(xo,Dx), E), PE(CD(xo,Dx), E));
    Dx = Dx/10;
  } while(Dx >= 1e-15);

f(float x)
{ // Return f(x) at x
    return exp(sin(x)^2);
}

ED(float x)
{ // Return exact derivative at x of f(x)
    return 2*sin(x)*cos(x)* exp(sin(x)^2);
}

RD(float x, float Dx)
{ // Return right derivative at x using interval Dx
    return (f(x+Dx)-f(x))/Dx;
}

CD(float x, float Dx)
{ // Return central derivative at x using interval Dx
    return (f(x+Dx/2)-f(x-Dx/2))/Dx;
}

PE(float a, float b)
{ // Return percentage difference: (a-b)*100/b
    return (a-b)*100/b;
}
\end{verbatim}
9.3 Second Derivatives

The derivative of the first derivative is the second derivative. Its symbolic notation is:

$$\frac{d^2y}{dx^2} \equiv \frac{d}{dx} \left( \frac{dy}{dx} \right)$$

The second derivative indicates how the first derivative varies with \(x\).

<table>
<thead>
<tr>
<th>Sign of second derivative at (x = \bar{x})</th>
<th>Nature of first derivative</th>
<th>Nature of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\frac{d^2y}{dx^2}\bigg</td>
<td>_{x=\bar{x}} &gt; 0)</td>
<td>(\frac{dy}{dx}) increasing</td>
</tr>
<tr>
<td>(\frac{d^2y}{dx^2}\bigg</td>
<td>_{x=\bar{x}} &lt; 0)</td>
<td>(\frac{dy}{dx}) decreasing</td>
</tr>
<tr>
<td>(\frac{d^2y}{dx^2}\bigg</td>
<td>_{x=\bar{x}} = 0)</td>
<td>(\frac{dy}{dx}) is stationary. Value of (\frac{dy}{dx}) is a local maximum or minimum</td>
</tr>
</tbody>
</table>

- Value of \(y\) is a local maximum at \(\bar{x}\) when \(\frac{dy}{dx}\bigg|_{x=\bar{x}} = 0\) \& \(\frac{d^2y}{dx^2}\bigg|_{x=\bar{x}} < 0\)

- Value of \(y\) is a local minimum at \(\bar{x}\) when \(\frac{dy}{dx}\bigg|_{x=\bar{x}} = 0\) \& \(\frac{d^2y}{dx^2}\bigg|_{x=\bar{x}} > 0\)
The second derivative $\frac{d^2}{dx^2} f(x) \bigg|_{x=x_0}$ can be written in the following equivalent form:

$$\frac{d^2 f(x)}{dx^2} \bigg|_{x=x_0} = \frac{d}{dx} \left( \frac{d(f(t))}{dt} \bigg|_{t=x} \right) \bigg|_{x=x_0}$$

We can evaluate it by nesting deriv() within deriv():

$$\text{deriv}(x, x_0, \text{deriv}(t, x, f(t)));$$

and plot it by varying $x_0$ of the preceding expression:

$$\text{plot}(x_0, a, b, \text{deriv}(x, x_0, \text{deriv}(t, x, f(t))));$$

Like in mathematics, dummy variable names such as $x_0$, $x$, and $t$ are arbitrary, and their scope is limited to within the expression. Thus, the following expression is essentially the same as the preceding one:

$$\text{plot}(z, a, b, \text{deriv}(v, z, \text{deriv}(u, v, f(u))));$$

### Sample Program 9.2 - Plotting Derivatives and Finding Extremum

The following program:

- Plots the function $f(x) = \frac{x \cos \frac{x}{2}}{x^2 + 2x + 3}$ and its first two derivatives for the range $0 \leq x \leq \pi$.
- Finds $x$ where $\frac{df(x)}{dx} = 0$ and determines $f(x)$ at this $x$, and find out if it is maximum or minimum.

```c
main()
{
    clearplot();
    useoption("RADIANS");
    a = 0; b = pi; // Plotting limits
```
plot(x, a, b, f(x));  // Function
setop(C, 9);        // Light blue
plot(xo, a, b, deriv(t, xo, f(t)));// First deriv
setop(C, 10);      // Light green for second derivative
plot(xo, a, b, deriv(x, xo, deriv(t, x, f(t))));
// Find x for derivative = 0
x = root1(x, a, b, deriv(t, x, f(t)));
// Second derivative at x
D2 = deriv(u, x, deriv(t, u, f(t)));
print(\^, " At x = ", x);
if(D2 > 0) {
    print(" f(x) is minimum: ", f(x));
} else {
    print(" f(x) is maximum: ", f(x));
}
}

f(float x)
{
    // Return f(x) at x
    return x*cos(x/2)/(x^2+2*x+3);
}

Practice Drills 9.1

The beam ABC is supported by the hinge support at A, and by the cable at point B, a distance x from A. Under the load P, the tension in the cable is

\[ T = \frac{Pld}{x\sqrt{d^2-x^2}} \]
Given \( L = 4 \text{ m}, d = 2 \text{ m} \) and \( P = 5 \text{ kN} \), write the program to:

(a) Plot the cable tension for the range \( 0.1d \leq x \leq 0.9d \)
(b) Find the best place to attach the cable so that its tension is minimum.
(c) Compare your solutions to the results given by the built-in function \texttt{minz1()} for minimization of \( T \):

\[
\begin{align*}
    x_0 &= \texttt{minz1}(x, a, b, T(x)); \\
    T_{\text{min}} &= T(x); \quad \text{// Minimum T}
\end{align*}
\]

### 9.5 Definite Integrals

Examples of definite integrals \( I = \int_a^b f(x)dx \)

- Distance traveled:

A moving particle has its velocity as a function of time: \( v(t) \).
At the time \( t_i \) its velocity is \( v(t_i) \), and the distance traveled during a small time interval \( \Delta t_i \) is \( v(t_i) \Delta t_i \). The total distance traveled between the time \( a \leq t \leq b \) can be approximately found as:

\[
d = \sum_{i=1}^{N} v(t_i) \Delta t_i \quad N = \text{number of sub-intervals within } a \leq t \leq b
\]

If \( N \) approaches infinitive (i.e. \( \Delta t_i \) approaches 0), the value \( d \) will be exact as given by the definite integral:

\[
d = \sum_{i=1}^{N} v(t_i) \Delta t_i = \int_{a}^{b} v(t)dt
\]

- Work done by a varying force:
A force $f$ varies from point to point. At the position $x_i$, the force is $f(x_i)$. The work done by this force during a small distance $\Delta x_i$ is $f(x_i) \Delta x_i$. The total work done during the traveled interval $a \leq x \leq b$ can be approximately found as:

$$ W = \sum_{i=1}^{N} f(x_i) \Delta x_i \quad N = \text{number of sub-intervals within } a \leq x \leq b $$

If $N$ approaches infinitive (i.e. $\Delta x_i$ approaches 0), the value $W$ will be exact as given by the definite integral:

$$ W = \sum_{i=1}^{\infty} f(x_i) \Delta x_i = \int_{a}^{b} f(x)dx $$

Geometrical meaning of the definite integral $I = \sum_{i=1}^{\infty} f(x_i) \Delta x_i = \int_{a}^{b} f(x)dx$

$f(x_i)\Delta x_i$ : Area of the hatched strip

$$ I = \sum_{i=1}^{\infty} f(x_i) \Delta x_i = \int_{a}^{b} f(x)dx = \text{Total shaded area under the curve} $$

In the above, $x_i$ is at the midpoint of the $i^{\text{th}}$-subinterval, and thus the formula is called the midpoint rule.
Sample Program 9.3 - Numerical Integration using the Midpoint Rule

The following program computes the approximate (net) area between \( a \) and \( b \) by summing the areas of \( N \) strips of constant width \( h \):

\[
I = \int_a^b f(x)\,dx \approx \sum_{i=1}^N f(x_i) \cdot h
\]

where \( f(x) = \frac{x}{1 + \sin 2x}, \; a = 0, \; b = \pi / 2 \)

The summation process is done by the user-defined function `Integral(float a, float b, float N)` for increasing \( N \). The results are compared to:

- The exact solution \( I = F(b) - F(a) \)
  
  where \( F(x) = -\frac{x}{2}\tan\left(\frac{\pi}{4} - x\right) + \frac{1}{2}\ln\left(\sin\left(\frac{\pi}{4} + x\right)\right) + \text{Constant} \)

- The value given by the built-in function `integ()`:

  \[
  I = \text{integ}(x, a, b, f(x));
  \]

The width of each of the \( N \) strips is \( h = \frac{b - a}{N} \)
The centre line of the first strip is at \( x_i = a + \frac{h}{2} \), and that of the next strip is at \( x_i \leftarrow x_i + h \), and so on. At a mid-point \( x_i \), the function value is \( f(x_i) \), and thus, the area of the strip is \( f(x_i)h \).

The accuracy increases as the number of strips increases.

```c
main()
{
  float n;
  clearplot();
  useoption("RADIANS");
  // Initial data that may be changed at runtime
  a=0; b=pi#/2;       // Limits of integral
  plot(x, a, b, f(x));
  E = F(b) - F(a);    // Exact value
  B = integ(x, a, b, f(x)); // Built-in function
  // Compute integral using N = 5,10,15, ...100 strips
  print(^^, "  Exact = ", E, "  integ() = ", B,
       "  %Error = ", PE(B, E));
  float N;        // Number of strips
  print(^^, "           N       I           PE", ^^);
  for(N = 5; N <= 100; N = N+5)
  {
    I = Integral(a, b, N);    // Compute integral
    print(^, N, I, PE(I, E));
  }
}

PE(float a, float b)
{
  // Percentage difference
  return (a - b)*100/b;
```
// Return value of function f(x) at x
f(float x)
{
    return x/(1+sin(2*x));
}

// Return value of function F(x) at x
F(float x)
{
    return -x/2*tan(pi#/4 - x) + 0.5*ln(sin(pi#/4 + x));
}

/////////////////////////////////////////////////////////////////////
// Compute area of f(x) between limits a and b
// Input: N = Number of strips
//        a, b: limits (global variables)
Integral(float a, float b, float N)
{
    float I, xi, Sum, h;    // Temporary variables
    h = (b-a)/N;               // Width of strip
    // Sum areas of rectangles
    Sum = 0;                   // Initialized
    xi = a+h/2;                // First strip midpoint
    for(I=1; I<= N; I=I+1) {
        Sum = Sum + f(xi)*h;    // Accumulate areas
        xi = xi+h;              // Next strip
    }
    return Sum;
}
Practice Drills 9.2

1. Under a differential force $wdz$ applied at the position $z$, the
differential tension in the cable $BD$ of the beam shown is:

$$dT = \frac{zL}{uh} \, w \, dz$$

where

$h = \text{distance } AD = 2 \, \text{m}$

$u = \text{distance } AB = 2.5 \, \text{m}$

$L = \text{length of the cable } = \text{hypot}(h, u)$

$wdz = \text{a differential force applied at the position } z$

If the actual load on the beam is a distributed load specified by the
function $w(z)$ over the interval $a \leq z \leq b$, the cable tension is obtained
by summing $dT$ over the interval from $a$ to $b$:

$$T = \int_{a}^{b} dT = \int_{a}^{b} \frac{zLw(z)}{uh} \, dz$$

Write a user-defined function `Integral(float a, float b, float N)` that
finds the tension $T$ for the following distributed load:

$$w(z) = 10(1 + \frac{z}{4}) \text{ over the interval } 0 \leq z \leq 4$$
The program should vary \( N \) from 5 to 20 in step of 5, and compare the computed solutions with the results obtained by the built-in function \( \text{integ}() \).

2. Repeat the above exercise for each of the following distributed loads:

(a) \( w(z) = 5 \left(1 + \frac{z^2}{16}\right) \) over the interval \( 0 \leq z \leq 4 \)

(b) \( w(z) = 10 \sin \frac{\pi z}{8} \) over the interval \( 0 \leq z \leq 4 \)
Chapter 10

Partial Derivatives and Multiple Integrals

He who has a why can endure any how -- Friedrich Nietzsche
10.1 Functions of Many Variables

Functions of many variables arise naturally in many areas of your study in Civil or Building Engineering. E.g.

(a) Temperature distribution:
   In the x-y plane: \( T(x, y) \): Steady state (independent of time)
                      \( T(x, y, t) \): Transient state (time is a factor)
   In 3-D space: \( T(x, y, z) \): Steady state
                  \( T(x, y, z, t) \): Transient state
(b) Stress distribution in soil, foundations, structures
(c) Pressure in fluid
(d) Particle concentration in air, water, soil.

10.2 Plotting Functions of Two Variables

Figure 10.1 is the plot of \( f(x, y) = \sin x \cos y \quad 0 \leq x, y \leq \pi / 2 \) effected by the following expression:

\[
\text{sp\_surface}(y, 0, \pi/2, x, 0, \pi/2, \sin(x) \ast \cos(y));
\]

![Figure 10.1 Surface plot](image)

The plotted surface can be rotated, resized and moved:

- Left mouse-button: drag to move
- Right mouse-button: drag to zoom in/out
• Both mouse buttons: drag to rotate about x, y-axes (i.e. varying Pitch & Yaw angles)

• Both mouse buttons + Shift-key: drag to rotate about z, x-axes (i.e. varying Roll & Pitch angles)

### 10.3 Partial Derivatives

The plot is a 3-D surface that shows grid lines at different values of constant x and y. For example, at the point \( P(x = \frac{\pi}{4}, y = \frac{\pi}{4}) \), the line of constant \( y = \frac{\pi}{4} \) is \( f(x, \frac{\pi}{4}) = \frac{\sqrt{2}}{2} \sin x \)

The slope at \( P \) of the above line (with \( y \) constant) is:

\[
\left. \frac{df(x, \frac{\pi}{4})}{dx} \right|_{x = \frac{\pi}{4}} = \left. \frac{\sqrt{2}}{2} \cos x \right|_{x = \frac{\pi}{4}} = \frac{1}{2}
\]

In general, the derivative of \( f(x, y) \) with respect to \( x \) at constant \( y \) is defined as the partial derivative \( f(x, y) \) with respect to \( x \):

\[
\frac{\partial f(x, y)}{\partial x} = \cos x \quad \cos y
\]

Thus, \( \frac{\partial (x, y)}{\partial x} \) is the rate of change in \( f(x, y) \) due to change in \( x \) alone. Similarly, \( \frac{\partial (x, y)}{\partial y} \) is the rate of change in \( f(x, y) \) due to change in \( y \) alone:

\[
\frac{\partial f(x, y)}{\partial y} = -\sin x \quad \sin y
\]

Knowing the rates of change with respect to \( x \) and \( y \), the change in \( f(x, y) \) due to changes in both \( x, y \) can be expressed as, approximately:

\[
df = f(x + dx, y + dy) - f(x, y) = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy
\]

\( df \) is called the total differential of \( f \).
1. Computation

If \( f(x, y, z) \) is an existing function, its partial derivatives \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \) at the point \((x=x_0, y=y_0, z=z_0)\)

can be approximately evaluated by the following respective expressions:

\[
D_x = \text{deriv}(x, x_0, f(x, y_0, z_0)); \quad \text{// \( x \) varies}
\]
\[
D_y = \text{deriv}(y, y_0, f(x_0, y, z_0)); \quad \text{// \( y \) varies}
\]
\[
D_z = \text{deriv}(z, z_0, f(x_0, y_0, z)); \quad \text{// \( z \) varies}
\]

Second-order partial derivatives, by definition:

\[
f_{xx} = \frac{\partial^2 f(x, y, z)}{dx^2} = \frac{\partial}{\partial x} \left( \frac{\partial f(u, y, z)}{\partial u} \right)_{u=x}
\]
\[
f_{yy} = \frac{\partial^2 f(x, y, z)}{dy^2} = \frac{\partial}{\partial y} \left( \frac{\partial f(x, u, z)}{\partial u} \right)_{u=y}
\]
\[
f_{zz} = \frac{\partial^2 f(x, y, z)}{dz^2} = \frac{\partial}{\partial z} \left( \frac{\partial f(x, y, u)}{\partial u} \right)_{u=z}
\]

The second form of the above can be directly translated into CMAP expressions as follows (for the derivatives at the point \(x_0, y_0, z_0\)):

\[
D_{xx} = \text{deriv}(x, x_0, \text{deriv}(u, x, f(u, y_0, z_0))); \quad // \( x \) varies
\]
\[
D_{yy} = \text{deriv}(y, y_0, \text{deriv}(u, y, f(x_0, u, z_0))); \quad // \( y \) varies
\]
\[
D_{zz} = \text{deriv}(z, z_0, \text{deriv}(u, z, f(x_0, y_0, u))); \quad // \( z \) varies
\]

Mixed second-order partial derivatives:

\[
f_{xy} = \frac{\partial^2 f(x, y, z)}{dxdy} = \frac{\partial}{\partial x} \left( \frac{\partial f(x, y, z)}{\partial y} \right)
\]
\[
f_{xz} = \frac{\partial^2 f(x, y, z)}{dxdz} = \frac{\partial}{\partial x} \left( \frac{\partial f(x, y, z)}{\partial z} \right)
\]
\[ f_{yz} = \frac{\partial^2 f(x, y, z)}{dydz} = \frac{\partial}{\partial y} \left( \frac{\partial f(x, y, z)}{\partial z} \right) \]

Evaluated at the point \( x_0, y_0, z_0 \):

\[
\begin{align*}
D_{xy} &= \text{deriv}(x, x_0, \text{deriv}(y, y_0, f(x, y, z_0))) ; \\
D_{xz} &= \text{deriv}(x, x_0, \text{deriv}(z, z_0, f(x, y_0, z))) ; \\
D_{yz} &= \text{deriv}(y, y_0, \text{deriv}(z, z_0, f(x_0, y, z))) ;
\end{align*}
\]

**Sample Program 10.1 - Partial Derivatives**

Given \( f(x, y) = \ln(3x + 5y) \) and its partial derivatives:

\[
\begin{align*}
& f_x = \frac{\partial f(x, y)}{\partial x} = \frac{3}{3x + 5y}, \quad f_y = \frac{\partial f(x, y)}{\partial y} = \frac{5}{3x + 5y} \\
& f_{xx} = \frac{\partial^2 f(x, y)}{\partial x^2} = \frac{-9}{(3x + 5y)^2}, \quad f_{yy} = \frac{\partial^2 f(x, y)}{\partial y^2} = \frac{-25}{(3x + 5y)^2} \\
& f_{xy} = \frac{\partial^2 f(x, y)}{\partial x \partial y} = \frac{-15}{(3x + 5y)^2}
\end{align*}
\]

The following program computes the partial derivatives of \( f(x, y) \) at \((x_0=1, y_0=1.5)\) using \texttt{deriv()} and compare the results with the exact values given by the above expressions.

```c
main()
{
    xo = 1; yo = 1.5;
    Dx = deriv(x, xo, f(x, yo));
    Dy = deriv(y, yo, f(xo, y));
    Dxx = deriv(x, xo, deriv(u, x, f(u, yo)));
    Dyy = deriv(y, yo, deriv(u, y, f(xo, u)));
    Dxy = deriv(x, xo, deriv(y, yo, f(x, y)));

    print("  Accuracy of partial derivatives by deriv()");
    print("", ^
    print("  Exact % Error", ^
    print("  fx: ", E = fx(xo,yo), PE(Dx, E), ^
    print("  fy: ", E = fy(xo,yo), PE(Dy, E), ^
```
The Gradient Vector

Consider a multi-variable function \( f(x) = f(x_1, x_2, x_3, \cdots) \). The complete list of its first partial derivatives is
The vector \( \mathbf{G} \) is called the gradient vector. It is conveniently computed using the built-in function \( \text{grad()} \):

\[
\text{grad}(x, \ G, \ f(x)) ;
\]

where vector \( x \) contains the values of \( x_1, x_2, x_3, \ldots \) at which the gradient is computed, and \( G \) is a (new or existing) variable-name for storing the gradient vector.

2. Application to Error Estimation

Suppose that we compute the quantity \( f(a, b, c) \) using the measured values of \( a, b \) and \( c \). Since measured values are subject to errors, the computed quantity \( f \) is also subject to error, which we want to estimate.

The total differential of \( f(a, b, c) \) is:

\[
df = \frac{\partial f}{\partial a} \, da + \frac{\partial f}{\partial b} \, db + \frac{\partial f}{\partial c} \, dc
\]

It gives the change in \( f \) due to the changes (i.e. variations, errors) \( da, db, dc \).

Let \( \pm \Delta a, \pm \Delta b, \pm \Delta c \) be the probable errors of \( a, b, \) and \( c \); the probable maximum error in \( f \) is given by:

\[
\Delta f = \sqrt{\left(\frac{\partial f}{\partial a} \Delta a\right)^2 + \left(\frac{\partial f}{\partial b} \Delta b\right)^2 + \left(\frac{\partial f}{\partial c} \Delta c\right)^2}
\]
Sample Program 10.2 - Propagation of Errors

Determine the maximum error $\Delta f$ of

$$f(x) = 2x_1^2x_2x_3 + x_2^3x_3^2$$

where the values of $\{x\}$ and their estimated errors are

$$x = \{5.24 \pm 0.01, 2.66 \pm 0.03, 12.05 \pm 0.04\}$$

The error estimate $\Delta f$ is

$$\Delta f = \sqrt{\left(\frac{\partial f}{\partial x_1}\Delta x_1\right)^2 + \left(\frac{\partial f}{\partial x_2}\Delta x_2\right)^2 + \left(\frac{\partial f}{\partial x_3}\Delta x_3\right)^2}$$

```c
main() {
    defmat(x[3], 5.24, 2.66, 12.05);
    // zero(G[3]);
    grad(x,G,f(x));        // Note 1
    defmat(Ex[3], 0.01, 0.03, 0.04);  // Errors in \{x\}
    !A = G &* Ex;          // Note 2
    Ef = hypot(!A);        // Note 3
    print(x,G,A,^ " f = ", f(x)," Error Ef = ", Ef);
}

f(mat x) {
} // Ans: f =  4493.07   Ef =  115.046
```

Notes

1. `grad()` returns the gradient vector $G$ computed at the given $x$. Since `grad()` will create $G$, we don’t have to reserve the storage for $G$ in advance.

2. The `array operator` &* multiplies each element of $G$ to a corresponding element of $Ex$ to form the new vector $A$. I.e.
\[
A = \begin{bmatrix}
\frac{\partial f}{\partial x_1} \\
\frac{\partial f}{\partial x_2} \\
\frac{\partial f}{\partial x_3}
\end{bmatrix}
\text{ & } \Delta
= \begin{bmatrix}
\Delta x_1 \\
\Delta x_2 \\
\Delta x_3
\end{bmatrix}
\]

3. The function `hypot(A)` returns the square root of the sum of squares of the elements of `A`.

The following is a condensed version of the same program. This version makes use of the matrices returned by `grad()` and `defmat()`.

```c
main() {
    Ef = hypot(!A = (grad(!defmat(x[3], 5.24, 2.66, 12.05),
                    G,f(x)) &* defmat(Ex[3], 0.01, 0.03, 0.04)));
    print(x,G,A, ^ " f = ", f(x),"   Error Ef = ", Ef);
}

f(mat x) {
}
```

3. Reporting the Results of Computation

Computer-generated numerical results may contain many digits, but not all of them should be reported. The number of meaningful digits to be reported depends on the precision obtainable. In general, the last significant digit in a reported value must be the only digit that we are not sure of. For example, if we write `A = 467.263 m`, then we are not sure of the digit 3, implying that the maximum error in `A` is ±0.005m. And if we write `A = 467.2 m`, then we are not sure of the digit 2, implying that the maximum error in `A` is ±0.5 m. Finally, if we write `467.2000 m`, then this implies that the precision is ±0.0005 m.

Sample Program 10.2 gives the result

\[f = 4493.06709754 \text{ & its error is } Ef = 115.046433501662\]

The value for `f` should be reported as:
f = 4500. ± 120.

Since the error is 115, f is rounded to the nearest 100. Adding more significant digits would be senseless as that would give a false impression of precision. Technical writers avoid studiously such mistakes.

Practice Drills 10.1

1. Replace the built-in function grad() in the program of Example 10.1 by a user-defined function GRAD(mat x, mat G) that computes and returns the gradient vector G of an existing function f(mat x) at the point defined by vector x. Test the revised program.

2. Write a program to compute the following quantities as well as their error estimates. Inspect the computed results and report them with the appropriate number of digits that reflect the obtainable precision.

(a) The area of a triangle having three sides $a, b, c$:

$$A = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{where} \quad p = \frac{1}{2}(a + b + c)$$

$a = (41 \pm 1) \, \text{mm}$, $b = (24 \pm 1) \, \text{mm}$, $c = (35 \pm 1) \, \text{mm}$

(b) The volume of a room having three sides $a, b, c$:

$$V = abc$$

$a = (3 \pm 0.01) \, \text{m}$, $b = (6 \pm 0.01) \, \text{m}$, $c = (5 \pm 0.01) \, \text{m}$

(c) The volume of a cylinder having radius $a$ and height $b$:

$$V = \pi a^2 b$$

$a = (45 \pm 0.1) \, \text{mm}$, $b = (240 \pm 1) \, \text{mm}$

(d) The cable tension $T$ given by:

$$T = \frac{PLd}{x\sqrt{d^2 - x^2}}$$

$P = (50 \pm 5) \, \text{kN}$, $L = (4 \pm 0.005) \, \text{m}$, $d = (2 \pm 0.005) \, \text{m}$, $x = (1.4 \pm 0.1) \, \text{m}$
10.4 Double Integrals

Suppose that the shaded area \( A \) is a city bounded by \( a \leq x \leq b \) and \( f_1(x) \leq y \leq f_2(x) \). If \( C(x_i,y_i) \) is the rain density \((m^3/m^2)\) at the point \( p(x_i,y_i) \), what is the total amount of rain water in the city?

We decompose the area \( A \) into \( n \) elemental areas \( dA_i = dx_i dy_i \), \( i = 1,2,3,...,n \). The amount of rain water in the small area \( dA_i \) is

\[
C(x_i,y_i) dA_i = C(x_i,y_i) dx_i dy_i
\]

It is the volume of the vertical column shown in the figure.
The total amount of rain in the entire area $A$ is then the volume covered by the surface $C$ over the area $A$. This volume is computed by summing all the elemental volumes:

$$I = \sum_{i=1}^{n} C(x_i, y_i) dA_i = \sum_{i=1}^{n} C(x_i, y_i) dx_i dy_i$$

When $n$ approaches infinity, the elemental areas $dA$ decreases to zero. At the limit, we get the double integral:

$$I = \int_{A} C(x, y) dA = \int_{A} \int_{f(x)}^{g(x)} C(x, y) dy dx$$

1. Evaluation

CMAP function `integ()` evaluates the preceding double integral as:

```plaintext
integ(x, a, b, integ(y, f1(x), f2(x), C(x,y)));
```

Practice Drills 10.2

1. Write one self-contained expression for evaluating each of the following proper integrals - Use nested `integ()`:

<table>
<thead>
<tr>
<th>Integral</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $$I_1 = \int_{0}^{2} \int_{-1/(x^2-1)}^{1/(x^2-1)} (x^2 + y^2) \ dy \ dx$$</td>
<td>$3\pi/2$</td>
</tr>
<tr>
<td>(b) $$I_2 = \int_{-\sqrt{2}}^{\sqrt{2}} \int_{\sqrt{2-x^2}}^{\sqrt{2-x^2}} \left[ \sqrt{6-x^2-y^2} - (x^2 + y^2) \right] \ dy \ dx$$</td>
<td>7.74</td>
</tr>
<tr>
<td>(c) $$I_3 = \int_{-1}^{1} \int_{-1}^{1} \frac{1}{\sqrt{1+x^2+y^2}} \ dx \ dy$$</td>
<td>$4 \ln(2 + \sqrt{3}) - \frac{2}{3} \pi$</td>
</tr>
</tbody>
</table>

2. Write one self-contained expression for evaluating each of the following improper double integrals – Use nested `integ()`:
<table>
<thead>
<tr>
<th>Integral</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( I_4 = \int_{0}^{1} \int_{0}^{1} \frac{1}{1-x^2y^2} , dx , dy )</td>
<td>( \frac{\pi^2}{8} )</td>
</tr>
<tr>
<td>(b) ( I_5 = \int_{0}^{1} \int_{0}^{1} \frac{1}{1-xy} , dx , dy )</td>
<td>( \frac{\pi^2}{6} )</td>
</tr>
<tr>
<td>(c) ( I_6 = \int_{0}^{1} \int_{0}^{1} \frac{1}{(x+y)\sqrt{(1-x)(1-y)}} , dx , dy )</td>
<td>( 4K )</td>
</tr>
<tr>
<td>( K = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2} = 0.91596 )</td>
<td></td>
</tr>
<tr>
<td>(d) ( I_7 = \int_{0}^{1} \int_{0}^{1} \frac{x-1}{(1-xy)\ln(xy)} , dx , dy )</td>
<td>( \gamma = \lim_{n \to \infty} \left( \sum_{k=1}^{n} \frac{1}{k} - \ln n \right) = 0.5772156 )</td>
</tr>
</tbody>
</table>


2. Application: Centroids of Areas

When the domain \( A \) can be specified by the extreme points \( a \) and \( b \) (on \( x \)-axis) and two curves: the lower one \( y=f_1(x) \), and the upper one \( y=f_2(x) \), the double integrals in the formulas for locating the centroid can be simplified to single integrals as follows.

The area: \( A = \int_{x=a}^{x=b} \left( \int_{y=f_1(x)}^{y=f_2(x)} \, dy \right) \, dx = \int_{x=a}^{x=b} \left[ y \right]_{y=f_1(x)}^{y=f_2(x)} \, dx = \int_{x=a}^{x=b} \left[ f_2(x) - f_1(x) \right] \, dx \)

Note: The integrand \( \left[ f_2(x) - f_1(x) \right] \, dx \) is the area \( dA \) of the elemental strip shown in the figure.
The first moment of area about x-axis:

\[ M_x = \int_{x=a}^{x=b} \left( \int_{y=f_1(x)}^{y=f_2(x)} y \, dy \right) \, dx = \int_{x=a}^{x=b} \left( \frac{y^2}{2} \right)_{f_1(x)}^{f_2(x)} \, dx = \frac{1}{2} \int_{x=a}^{x=b} \left[ f_2^2(x) - f_1^2(x) \right] \, dx \]

Note: The last term can be written immediately by noting that \( \frac{1}{2} \left[ f_2^2(x) - f_1^2(x) \right] \, dx \) is the first moment (about the x-axis) of the elemental strip \( dA \).

The first moment of area about y-axis:

\[ M_y = \int_{x=a}^{x=b} \left( \int_{y=f_1(x)}^{y=f_2(x)} x \, dx \right) \, dy = \int_{x=a}^{x=b} \left[ \frac{x^2}{2} \right]_{f_1(x)}^{f_2(x)} \, dx = \int_{x=a}^{x=b} \left[ f_2(x) - f_1(x) \right] \, dx \]

Note: The integrand \( \left[ f_2(x) - f_1(x) \right] \, dx \) is the first moment (about the y-axis) of the elemental strip \( dA \).

Centroid’s position:

\[ \bar{y} = \frac{M_x}{A}, \quad \bar{x} = \frac{M_y}{A} \]

Sample Program 10.3 - Centroid of Plane Area


The following program evaluates the integrals using the preceding simplifications as well as by using double integrals:
main()
{
    A = integ(x, a=0, b=20, sqrt(x/0.05)-0.05*x^2);
    Mx = 0.5*integ(x, a, b, (x/0.05)-0.05^2*x^4);
    My = integ(x, a, b, x*(sqrt(x/0.05)-0.05*x^2));
    print(^, A, Mx/A, My/A); 
    DoubleSum();
}

DoubleSum()
{
    A = integ(x, a,b, integ(y, 0.05*x^2, sqrt(x/0.05), 1));
    Mx = integ(x, a,b, integ(y, 0.05*x^2, sqrt(x/0.05), y));
    My = integ(x, a,b, integ(y, 0.05*x^2, sqrt(x/0.05), x));
    print(^^, " Using double sum",^, A, Mx/A, My/A);
}

Output: 133.327 9.00 9.00

Using double sum
133.327 9.00045 9.00054

3. Application: Moments of Inertia
In order to find the moments of inertia of the shaded area about the $x$, $y$-axes, we find the moments of inertia of the elemental area $dA$ and integrate the results:

$$I_x = \int_a^b \left( \frac{\eta^3}{12} + \left[ \frac{f_2 + f_1}{2} \right]^2 \eta \right) dx \quad \text{where} \quad \eta = (f_2 - f_1)$$

$$I_y = \int_a^b x^2 \, dA = \int_a^b x^2 \left( f_2 - f_1 \right) dx$$

$$I_{xy} = \int_a^b x \left( \frac{f_2 + f_1}{2} \right) \, dA = \int_a^b \frac{1}{2} x \left( f_2^2 - f_1^2 \right) dx$$
Chapter 11

Solution of Non-linear Equations

I can calculate the movement of the stars, but not the madness of men – *Isaac Newton*

Two things are infinite: the universe and human stupidity; and I'm not sure about the universe. -- *Albert Einstein*
11.1 Equations with One Unknown

1. Problem Statement

Standard form

Solve for $x$ so that

$$F(x) = 0$$

where the right-hand-side is always 0. This is the standard form\(^1\) for computer solution.

If $x_o$ is a solution, then the value of $F(x_o)$ is zero or sufficiently close to zero. $x_o$ is also called the root of $F(x)$. If $x_o$ is not an accurate solution, the residual value $F(x_o)$ is too large to be acceptable.

Examples

Here are some examples of $F(x)$.

- $F(x) \equiv 2x - 4$
- $F(x) \equiv xe^{3x} - x^2 - 1$
- $F(x) \equiv \int_0^x \sin(t + \sqrt{x^2 + t}) \, dt - 0.025$

For each $F(x)$, any $x_o$ that makes $F(x_o)$ sufficiently small is a solution.

2. Accuracy Tolerance

To verify if $x_o$ is a solution, we evaluate the residual $F(x_o)$ to see if it is sufficiently small:

$$\left| F(x_o) \right| \leq tol\#$

where $tol\#$ is a small number such as 0.0001 depending on the accuracy required.

\(^1\) The standard form is convenient for computer solution because (i) It is completely general, and (ii) Only $F(x)$ need be specified (the right-hand side is always zero).
3. Lower and Upper Bounds of Solution

If the equation $F(x) = 0$ is non-linear in $x$, it may have many solutions. To isolate the desired solution we need to give an estimate of the root or a range within which it is found.

Built-in root1() function needs a range for the solution:

$$x_{\text{min}} \leq x_o \leq x_{\text{max}}$$

The solution limits should be based on our understanding of the physical problem. For example, if $x$ is the radius of a steel cable, its limits may be $0 \leq x \leq 1$m.

To search for the range of $x$ that encloses a root, we may plot the function $F(x)$. For example, the following expression:

```latex
plot(x, 0, 1, x*exp(3*x)-x^2-1);
```

plots the function $F(x) \equiv xe^{3x} - x^2 - 1$ for the range $0 \leq x \leq 1$ as shown in Fig. 1(a). Inspection of the plot shows the root $x_o$ close to 0.4.

![Figure 1(a): Plot of $F(x) \equiv xe^{3x} - x^2 - 1$](image)

Instead of bracketing the root within a range, the Newton-Raphson's method needs an estimate of the root to start the solution process as shown below.
4. Solution by Newton-Raphson's Method

Newton-Raphson method is an iterative technique for computing a better root based on an estimated value of the root. It makes use of the first derivative dF/dx as explained below.

Let $x_1$ be the estimated root as shown on Fig. 1(b). Being approximate, the residual $F(x_1)$ is non-zero as represented by point $A$ on the curve. We then draw a tangent to the curve at point $A$, intersecting the x-axis at the abscissa $x_2$ which is closer to the root $x_0$. Noting that the slope of the tangent at $A$ is the derivative of $F$ with respect to $x$ at the point $A$, an improved solution $x_2$ may be found by geometrical proportion (Fig. 1(b)) as:

$$x_2 = x_1 - \Delta x = x_1 - \frac{F(x_1)}{dF(x_1)/dx}$$

It is called the *iterative formula*. Once the value of $x_2$ is found, it is then used as the new initial estimate, leading to an even better solution. This process is repeated until the improvement becomes insignificant:

$$\left| \frac{x_2 - x_1}{x_2} \right| \leq \text{Small value}$$

where the left-hand side quantity is a measure of the *relative error* of the computed root. Depending on the situation, one may prefer to use the *absolute error* instead:

$$|x_2 - x_1| \leq \text{Small value}$$
Sample Program 11.1 is a simple implementation of the Newton-Raphson method. The flow-chart below shows the iterative process.

Sample Program 11.1 - Newton-Raphson's Method

In the following program, the function Root(float x1) returns the root of $F(x) = 0$ given the initial estimate $x_1$. The main() function invokes Root() using different estimates varying from -10 to 10 in step of 1 and outputs the results for $F(x) \equiv xe^{3x} - x^2 - 1$. The iteration stops when the relative error is less than 0.1%.

```c
float K; // Global var to count number of iterations
main()
{
    float x, R;
    print(" Estimate  Root  Number of Steps ");
    for(x=-10; x<=10; x=x+1) // Generate estimates
    {
        K = 0;  // Initialize counter
        R = Root(x);  // Find root
        print(", x, R, K);
    }
}

Root(float x1)
{
    float Err, x2;
```
do
{
    x2 = x1 - F(x1)/deriv(x, x1, F(x));
    Err = abs((x2-x1)/x2);  // Error estimate
    x1 = x2;     // Use improved value as new estimate
} while (Err > 0.001);
return x2;
}

F(float x)
{
    K = K+1;  // Increment counter
    return x*exp(3*x)-x^2-1;
}

Depending on the function F(x), the Newton Raphson method may not converge to the solution, particularly when the estimate is poor.

5. Solution with Function root1()

The built-in function root1(x, xmin, xmax, F(x)) returns the value of xo, if it exists, that makes F(xo) equal to zero within the tolerance specified by the pre-defined global constant tol# (= 10^-4 by default). The arguments xmin, xmax specify the range of the root. The root found should be saved by an assignment expression.

tol# = 1.e-4;  // Tolerance
plot(x, xmin, xmax, F(x));  // Plot F(x)
xo = root1(x, xmin, xmax, F(x));  // Save the root

In the above, F(x) may be an expression or a user-defined function. It must return a value for any given x. It may be defined explicitly by an expression or by lengthy computation within the user-defined function F(x).
Examples: In the following examples, F(x) is simple enough to be defined explicitly by an expression:

- $F(x) \equiv 2x - 4$
  \[
  \text{root1}(x, 0, 10, 2x-4); // Ans: 2
  \]

- $F(x) \equiv xe^{3x} - x^2 - 1$
  \[
  \text{root1}(x, 0, 1, x*\exp(3x)-x^2-1); //Ans: 0.3725
  \]

- $F(x) \equiv \int_0^1 \sin(t + \sqrt{x^2 + t}) \, dt - 0.025$
  \[
  \text{root1}(x, 0, 1, \text{integ}(t, 0, x, \sin(t+\sqrt{x^2+t}))-0.025); //Ans: 0.093
  \]

Sample Program 11.2 - Minimization by Root-Finding

Find $x$ to minimize the function $f(x)$

\[
  f(x) = \int_0^1 \frac{x^2 + tx^4 + \sin^2(\pi + t + x) + 1}{(1 + x + t)^2 + \sqrt{1 + tx^2}} \, dt
  \]

Note: This problem is taken from the text by John R. Rice where the stated solution is correct only when the square in the sine-term is removed.

Method 1: Function $f(x)$ is minimum or maximum when its derivative with respect to $x$ is zero. Thus, the problem becomes: Find $x_o$ so that:

\[
  F(x) \equiv \frac{df(x)}{dx} \bigg|_{x=x_o} = 0
  \]

The second derivative of $f(x)$ at $x_o$ is positive if $f(x_o)$ is the minimum:

\[
  \frac{d^2 f(x)}{dx^2} \bigg|_{x=x_o} \equiv \frac{d}{dx} \left( \frac{df(u)}{du} \right) \bigg|_{u=x} > 0
  \]

---


© 2015 K. H. Ha – BCEE 231 Chapter 11 - v4
```c
main()
{
    xo = rootl(xo, -10, 10, deriv(x, xo, f(x)));  
D2 = deriv(x, xo, deriv(u, x, f(u)));  // Second derivative
print(`, xo, D2, f(xo));
}

f(float x)
{
    return integ(t, 0, 1, (x^2+t*x^4+sin(pi#+t+x)^2+1)/((1+x+t)^2+sqrt(1+t*x^2)));
}  
// Ans: xo = 0.225143   D2 = 0.470888  f(xo) = 0.375756  

Method 2:  Direct minimization of \( f(x) \) by using the built-in function \texttt{minz1}().

main()
{
    xo = minz1(x, -10, 10, f(x));
    print(`, xo, f(xo));
}

f(float x)
{
    return integ(t, 0, 1, (x^2+t*x^4+sin(pi#+t+x)^2+1)/((1+x+t)^2+sqrt(1+t*x^2)));
}  
// Ans: xo = 0.225133    f(xo) = 0.375756  

Sample Program 11.3 - Unknowns in Limits of Integration

Find \( x \) so that \( F(x) = f(x) - 12 = 0 \) where

\[
f(x) = \int_{-x}^{x} (t^3 - x) \ln \frac{t^2 + 1}{1 + x^2} \, dt
\]
```
**Note:** This problem is adapted from John R. Rice's text (p. 381) where the stated solution is incorrect unless \( f(x) \) is as shown above.

```c
main()
{
    xo = root1(x, -10, 10, f(x) - 12);
    print(\^, xo, f(xo));
}
```

```c
f(float x)
{
    return integ(t, -x, x, (t^3 - x)^2*ln((1+t^2)/(1+x^2)));
}
```

// Ans: xo = -1.76855     f(xo) = 11.9995

### Practice Drills 11.1

1. For each of the following functions: (i) Write an expression to plot the function for the given interval of \( x \). (ii) Write an expression to find the value of \( x \) (within the given interval) at which the function crosses the \( x \)-axis:

   (a) \[ f(x) = x^{1/19} - 19^{1/19} \quad 1 \leq x \leq 100 \]
   (b) \[ g(x) = x^2 - (1 - x)^20 \quad 0 \leq x \leq 1 \]
   (c) \[ h(x) = 2xe^{-5} - 2e^{-5x} + 1 \quad 0 \leq x \leq 1 \]
   (d) \[ u(x) = 2xe^{-20} - 2e^{-20x} + 1 \quad 0 \leq x \leq 1 \]

2. Write a program to find the root(s) \( 0 \leq x \leq 1 \) of each of the following 3 equations:

   \[ f(x) = 0.3 \]
   \[ \frac{d}{dx} f(x) = \frac{0.4}{\cos x} \]
   \[ \int_0^x f(t)dt = 0.1 \]

   where \[ f(x) = xe^{-x^2} \sin x \]
11.2 Systems of Equations

Sample Program 11.4 - Two-Equation System

Solve for $x_1, x_2$ so that $f_1(x_1, x_2) = 0$, $f_2(x_1, x_2) = 0$
where:

\[
\begin{align*}
    f_1 & \equiv x_1^2 x_2 + 2x_1 x_2^2 - x_1 x_2 - 3 \\
    f_2 & \equiv x_1 x_2^2 - 2x_1^2 x_2 + 4x_1 x_2 + 1
\end{align*}
\]

The solution may be obtained with the built-in function \texttt{roots()} as shown in the following program.

```
main()
{
    N = 2; // Number of equations
    zero(f[N]); // Storage for f1, f2
    defmat(X[N], N:1); // Solution estimates
    roots(X, f, FX(X, f)); // Store roots in X
    print(^, X, "Residual =", !VResd = FX(X, f));
}

FX(mat X, mat f)
{
    // Define local variables x1, x2 to facilitate writing
    // the expressions of f1, f2:
    float x1 = X[1], x2 = X[2];
    f[1] = x1^2*x2+2*x1*x2^2-x1*x2-3;
    f[2] = x1*x2^2-2*x1^2*x2+4*x1*x2+1;
    return !f; // Return a matrix
}
```
1. Standard Form

A system of \( N \) equations for \( N \) unknowns can always be put in the following standard form:

- Solve for \( \mathbf{X} \equiv \{x_1, x_2, \ldots, x_N\} \) so that \( \mathbf{F}_X(\mathbf{X}) = 0 \) where

\[
\mathbf{F}_X(\mathbf{X}) \equiv \begin{bmatrix}
f_1(x_1, x_2, \ldots, x_N) \\
\vdots \\
f_N(x_1, x_2, \ldots, x_N)
\end{bmatrix}
\]

The function \( \mathbf{F}_X(\mathbf{X}) \) returns the residual vector \( \mathbf{f} = \{f_1, f_2, \ldots, f_N\} \) for any given vector \( \mathbf{X} = \{x_1, x_2, \ldots, x_N\} \). If \( \mathbf{X} \) is the correct solution, the residuals in \( \mathbf{f} \) will be negligible.

The advantage of the standard form is that we need to specify only the expressions of \( \mathbf{F}_X(\mathbf{X}) \). And the objective of root finding is to find \( \mathbf{X} \) to make \( \mathbf{F}_X(\mathbf{X}) \) zero.

2. Solution Estimates and Convergence

Solution techniques for nonlinear equations often use an iterative process which needs the initial estimates:

\( \mathbf{X} = \{x_1, x_2, \ldots, x_N\} : \text{Initial estimates of solution} \)

- Good initial estimates will provide quick convergence to solution. Very poor estimates may fail to give solution.
- The initial solution estimates should be based on our understanding of the physical problem.
- Non-linear equations may have many solutions that can be found by changing the initial estimates.
- Furthermore, not all mathematical solutions will be valid for the physical problem.

3. Accuracy Tolerance

To verify if the vector \( \mathbf{X}_o \) is the solution, we compute the length of the residual vector to see if it is sufficiently small:

\[
\| \mathbf{F}(\mathbf{X}_o) \| \leq \text{tol}\#
\]
where tol# is a small number such as 0.0001 depending on the accuracy required. Much smaller tolerance may not lead to convergence because only 15 digits (of any number) are stored in computers.

Sample Program 11.5 - Two-Equation Statics Problem

Determine the angle $A$ and the magnitude of $R$ to ensure equilibrium of the particle shown.

Formulation: $R + P + F = 0$

Scalar equations:

\[ \sum F_x = 0 : \quad f_1 = P \cos A - F \cos 30^\circ = 0 \]
\[ \sum F_y = 0 : \quad f_2 = P \sin A + F \sin 30^\circ - R = 0 \]

Two equations with two unknowns: $A$ and $R$.


```c
main()
{
    // Solution using scalar equations
    P = 500; F = 425;
    N = 2;    // Number of equations
    useoption("DEGREES");
    zero(f[N]);   // Residual vector
    defmat(X[N],30, 1);  // Estimates for A and R
    roots(X, f, FX(X, f)); // Roots X = {A, R}
    print('^, X, "Residual =", !VResd = FX(X, f));
}

FX(mat X, mat f)
{
    float A = X[1], R = X[2];    // Local names
    f[1] = P*cos(A) - F*cos(30);
    f[2] = P*sin(A) + F*sin(30) - R;
}
```

© 2015 K. H. Ha – BCEE 231 Chapter 11 - v4
return !f;
}

Ans: \( A = 42.6 \), \( R = 550.9 \)

Notes: If we allow \( R \) to be negative, we'll get the solution \( \{ A = 677.4^\circ \text{ (i.e. } -42.59^\circ \), \( R = -125.9 \text{ N } \} \) with the initial estimates \( \{1, 1\} \). We shouldn't be surprised that this simple problem is non-linear. Even the simplest trigonometric equation such as \( \cos(\theta) = 0 \) is non-linear in \( \theta \). It has infinite solutions: \( \theta = \pm \frac{\pi}{2} \pm n\pi, n = 1,2,... \)

Sample Program 11.6 - Five-Equation Cable Tension System

The beam \( EFGH \) is supported by vertical cables \( BF, CH \) hanging from the cable \( ABCD \). Find the tension forces \( T_{1,2,3,4,5} \) in the 5 cables in terms of the force \( P \) applied at \( G \).

Use the following additional data:
\( d_1 = 0.6 \, L \), \( d_2 = 0.8 \, L \), \( d_3 = 0.6 \, L \),
\( d_4 = 1.5 \, L \), \( \alpha = 40^\circ \), \( \beta = 20^\circ \),
\( \gamma = 45^\circ \)

Formulation: We need 5 equations written in terms of the 5 unknowns \( T \)'s.

Equilibrium of \( B \):
\[
\sum F_x = 0: \quad f_1 = T_2 \cos \beta - T_1 \cos \alpha = 0
\]
\[
\sum F_y = 0: \quad f_2 = T_2 \sin \beta + T_1 \sin \alpha - T_4 = 0
\]

Equilibrium of \( C \):
\[
\sum F_x = 0: \quad f_3 = T_3 \cos \gamma - T_2 \cos \beta = 0
\]
\[
\sum F_y = 0: \quad f_4 = T_3 \sin \gamma - T_2 \sin \beta - T_5 = 0
\]

Moment about \( E \) of all the forces acting on the free-body-diagram of the beam \( EFGH \):
\[ \sum M_E = 0: \quad f_5 \equiv T_4d_1 + T_5(d_1 + d_2) - Pd_4 = 0 \]

In the following program for solution of the 5 equations, we use unit value for \( L \) and \( P = 1 \). The computed cable tensions should then be scaled by \( P \).

```plaintext
main()
{
    P = 1; L = 1;
    d1 = 0.6*L; d2 = 0.8*L; d3 = 0.6*L; d4 = 1.5*L;
    Alpha = 40; Beta = 20; Gamma = 45;
    N = 5; // Number of equations
    useoption("DEGREES");
    zero(f[N]); // Residual vector
    defmat(X[N], N:1); // Estimates for cable tensions
    roots(X, f, FX(X, f)); // Roots X = {T1, T2, T3, T4, T5}
    print(^, X, ^^, " Residuals ", !VResd = FX(X, f));
}

FX(mat X, mat f)
{
    float T1 =X[1], T2 =X[2], T3 =X[3], T4 =X[4], T5 =X[5];
    f[1] = T2*cos(Beta)-T1*cos(Alpha);
    f[2] = T2*sin(Beta)+T1*sin(Alpha)-T4;
    f[3] = T3*cos(Gamma)-T2*cos(Beta);
    f[4] = T3*sin(Gamma)-T2*sin(Beta)-T5;
    f[5] = T4*d1+T5*(d1+d2)-P*d4;
    return !f;
}
```

**Food for thought:**

1. Why does the value of \( L \) have no effect on the cable tensions?
2. Given that the horizontal projection of \( AD \) must be \( 2L \), what is the best arrangement of the cables to minimize the maximum tension in the cables?
3. How would you solve the 5 equations by hand?

Sample Program 11.7 - A Complex Nonlinear System

Find \( x, y \) so that:

\[
f_1 \equiv ax + b_1y - x^2 + y^2 - 0.5 = 0
\]
\[
f_2 \equiv cx + d_1y - x^2 - y^2 + 0.5 = 0
\]

where

\[
a = 17.6504
\]
\[
b_1 = \left. \frac{d^2}{dt^2} \left( te^{-t} \right) \right|_{t=x}
\]
\[
b_2 = \left. \frac{d^2}{dt^2} \left( te^{-t} \right) \right|_{t=y}
\]
\[
c = 14(a^2 - 2b_2)
\]
\[
d = \left. \frac{d}{dt} \left( t^2 e^{-\frac{t}{2}} \right) \right|_{t=y}
\]

\[
= \left. \frac{d}{dt} \ln(1+t) \right|_{t=y}
\]

**Notes:** This problem is taken from John Rice's text but his expressions for \( b_1 \) and \( b_2 \) contain typo errors. It has many solutions that can be found using different initial estimates. E.g. \( X = \{-1, -1\} \) gives solution \( \{-0.173, -1\} \). Positive solution \( \{0.0009119, 2.216\} \) are obtained with initial estimates \( \{0, 1\} \). It's up to us to find solutions meaningful to the physical problem.

```c
main()
{
  tol# = 1.e-4;       // Tolerance
  N = 2;              // Number of equations
  zero(f[N]);        // Residual vector
  defmat(X[N],1, 1);  // Solution estimates
  a = 17.6504;        // Constant
  roots(X, f, FX(X, f));  // Roots in X
  print(^, X, !VResd = FX(X, f));
}```
FX(mat X, mat f)
{
    float x = X[1], y = X[2];
    b1 = deriv(t, x, deriv(u,t, u*exp(-u)));
    b2 = deriv(t, y, deriv(u,t, u*exp(-u)));
    c = 14*(a^2-2*b2);
    d = deriv(t,y, t^2*exp(-sqrt(abs(t)))) * deriv(t,y, ln(abs(1+t)));
    f[1] = a*x+b1*y-x^2+y^2-0.5;
    f[2] = c*x+d*y-x^2-y^2+0.5;
    return !f;
}

Ans: Obtained with the initial estimates X = {1, 1}
    X      = { -0.0001173,   -0.2257 }
    VResd  = { 0.0003478,   0.000264 }