MATLAB TUTORIAL

Mark Austin
Department of Civil and Environmental Engineering
University of Maryland at College Park

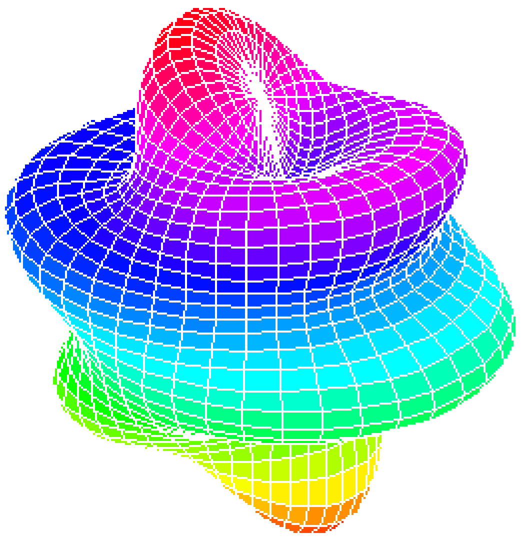
The original version of this tutorial is available on the Internet at http://www.isr.umd.edu/~austin/ence202.d/matlab-tutorial.html. It has been modified slightly by Tony Richardson for use at the University of Evansville.

GETTING STARTED

WHAT IS MATLAB ?

MATLAB, the "MATrix LABoratory" program, was initially written with the objective of providing scientists and engineers with interactive access to the numerical computation libraries Linpack and Eispack. These libraries are carefully tested, high-quality programming packages for solving linear equations and eigenvalue problems. MATLAB enables scientists and engineers to use matrix-based techniques to solve problems, without having to write programs in traditional languages like C and Fortran.

Nowadays MATLAB is a commercial "Matrix Laboratory" package which operates as an interactive programming environment with graphical output. The MATLAB programming language is exceptionally straightforward since almost every data object is assumed to be an array. Hence, for some areas of engineering MATLAB is displacing the Fortran and C programming languages, due to its interactive interface, reliable algorithmic foundation, fully extensible environment, and computational speed. MATLAB is available for many different computer systems, including Macs, PCs and UNIX platforms.



THIS TUTORIAL

The purposes of this tutorial are two-fold. In addition to helping you get started with MATLAB, we want you to see how MATLAB can be used in the solution of engineering problems. The latter objective is achieved through the presentation of a series of engineering application problems.

Throughout this tutorial we assume that you:

- Will read a couple of sections and then go to a PC to experiment with MATLAB.
- Have access to supplementary material on matrices, matrix arithmetic/operations, and linear algebra.

ENTERING AND RUNNING MATLAB

After starting MATLAB, you should see the following prompt in the "Command Window":

```
>>
```

You are now in MATLAB. From this point on, individual MATLAB commands may be given at the program prompt. They will be processed when you hit the "Enter" key. Many command examples are given below.

LEAVING MATLAB

A MATLAB session may be terminated by simply typing

```
>> quit
```

or by typing

```
>> exit
```

at the MATLAB prompt or selecting "Exit MATLAB" from the **File** menu.

ONLINE HELP

Online help is available from the MATLAB prompt (a double arrow), both generally (listing all available commands):

```
>> help
[a long list of help topics follows]
```

and for specific commands:

```
>> help demo
[a help message on MATLAB's demo program follows].
```

The "**doc**" command

```
>> doc
```

will start the Help browser. Try it. You can also use "**doc demo**" for help on a specific topic.

VARIABLES

MATLAB has built-in variables like **pi**, **eps**, and **ans**. You can learn their values from the MATLAB interpreter.

```
>> eps
eps =

      2.2204e-16
>> pi
ans =

      3.1416

>> help pi
```

```
PI      3.1415926535897....
        PI = 4*atan(1) = imag(log(-1)) = 3.1415926535897....
```

In MATLAB all numerical computing occurs with limited precision, with all decimal expansions being truncated at the sixteenth place [roughly speaking]. The machine's round-off, the smallest distinguishable difference between two numbers as represented in MATLAB, is denoted **eps**.

The variable **ans** will keep track of the last output which was not assigned to another variable.

Variable Assignment: The equality sign is used to assign values to variables:

```
>> x = 3

x =

      3

>> y = x^2

y =

      9
```

Variables in MATLAB are case sensitive. Hence, the variables "x" and "y" are distinct from "X" and "Y" (at this point, the latter are in fact, undefined).

Output can be suppressed by appending a "semicolon" to the command lines.

```
>> x = 3;
>> y = x^2;
>> y

y =

      9
```

Like the C and Fortran programming languages, MATLAB variables must have a value [which might be numerical, or a string of characters, for example]. Complex numbers are automatically available [by default, both **i** and **j** are initially aliased to sqrt(-1)].

Active Variables: At any time you want to know the active variables you can use who:

```
>> who

Your variables are:

ans       x        y
```

Removing a Variable: To remove a variable, try this:

```
>> clear x
```

Saving and Restoring Variables: To save the value of the variable "x" to a plain text file named "x.value" use

```
>> save x.value x -ascii
```

To save all variables in a file named "mysession.mat", in reloadable format, use

```
>> save mysession
```

To restore the session, use

```
>> load mysession
```

VARIABLE ARITHMETIC

MATLAB uses some fairly standard notation. More than one command may be entered on a single line, if they are separated by commas.

```
>> 2+3;
>> 3*4, 4^2;
```

Powers are performed before division and multiplication, which are done before subtraction and addition. For example

```
>> 2+3*4^2;
```

generates **ans = 50**. That is:

```
2+3*4^2  ==> 2 + 3*4^2    <== exponent has the highest precedence
         ==> 2 + 3*16     <== then multiplication operator
         ==> 2 + 48       <== then addition operator
         ==> 50
```

Double Precision Arithmetic: All arithmetic is done to double precision, which for 32-bit machines means to about 16 decimal digits of accuracy. Normally the results will be displayed in a shorter form.

```
>> a = sqrt(2)

a =

     1.4142
>> format long, b=sqrt(2)

b =

      1.41421356237310
>> format short
```

Command-Line Editing: The arrow keys allow "command-line editing" which cuts down on the amount of typing required, and allows easy error correction. Press the "up" arrow, and add "/2." What will this produce?

```
>> 2+3*4^2/2
```

Parentheses may be used to group terms, or to make them more readable. For example:

```
>> (2 + 3*4^2)/2
```

generates `ans = 25`.

BUILT-IN MATHEMATICAL FUNCTIONS

MATLAB has a platter of builtin functions for mathematical and scientific computations. Here is a summary of relevant functions, followed by some examples:

Function	Meaning	Example
sin	sine	sin(pi) = 0.0
cos	cosine	cos(pi) = 1.0

tan	tangent	tan(pi/4) = 1.0
asin	arcsine	asin(pi/2)= 1.0
acos	arccosine	acos(pi/2)= 0.0
atan	arctangent	atan(pi/4)= 1.0
exp	exponential	exp(1.0) = 2.7183
log	natural logarithm	log(2.7183) = 1.0
log10	logarithm base 10	log10(100.0) = 2.0

```
======================================================
```

The arguments to trigonometric functions are given in radians.

Example: Let's verify that

```
sin(x)^2 + cos(x)^2 = 1.0
```

for arbitrary x. The MATLAB code is:

```
>> format compact
>> x = pi/3;
>> sin(x)^2 + cos(x)^2 - 1.0
ans =
     0
>>
```

The "format compact" command eliminates spaces between the lines of output.

MATRICES

A matrix is a rectangular array of numbers. For example:

```
[ 1 2 3  4 ]
[ 4 5 6  7 ]
[ 7 8 9 10 ]
```

defines a matrix with 3 rows, 4 columns, and 12 elements.

MATLAB works with essentially only one kind of object, a rectangular numerical matrix possibly, with complex entries. Every MATLAB variable refers to a matrix [a number is a 1 by 1 matrix]. In some situations, 1-by-1 matrices are interpreted as scalars, and matrices with only one row or one column are interpreted as vectors.

ENGINEERING APPLICATION OF MATRICES

In the analysis of a wide range of engineering applications, matrices are a convenient means of representing transformations between coordinate systems, and for writing (moderate to large) families of equations representing the state of a system (e.g., equations of equilibrium, energy and momentum conservation, and so forth).

Example: Suppose that the following three equations describe the equilibrium of a simple structural system as a function of external loads and computed displacements.

```
 3 * x1 - 1 * x2 +  0 * x3 = 1
-1 * x1 + 4 * x2 -  2 * x3 = 5
 0 * x1 - 2 * x2 + 10 * x3 = 26
```

This family of equations can be written in the form A.X = B, where

```
    [  3 -1  0 ]       [ x1 ]           [  1 ]
A = [ -1  6 -2 ],  X = [ x2 ], and B = [  5 ].
    [  0 -2 10 ]       [ x3 ]           [ 26 ]
```

In a typical application, matrices A and B will be defined by the parameters of the engineering problem, and the solution matrix X will need to be computed.

Depending on the specific values of coefficients in matrices A and B, there may be: (a) no solutions to A.X = B, (b) a unique solution to A.X = B, or (c) an infinite number of solutions to A.X = B.

In this particular case, however, the solution matrix

```
         [ 1 ]
     X = [ 2 ]
         [ 3 ]
```

makes the right-hand side of the matrix equations (i.e., A.X) equal the left-hand side of the matrix equations (i.e., matrix B). We will soon see how to setup and solve this family of matrix equations in MATLAB (see examples below).

DEFINING MATRICES IN MATLAB

MATLAB is designed to make definition of matrices and matrix manipulation as simple as possible.

Matrices can be introduced into MATLAB in several different ways:

- Entered by an explicit list of elements,
- Generated by built-in statements and functions,
- Created in M-files (see section below),
- Loaded from external data files (see details below).

For example, either of the statements

```
>> A = [1 2 3; 4 5 6; 7 8 9];
```

and

```
>> A = [ 1  2  3
         4  5  6
         7  8  9 ]
```

creates the obvious 3-by-3 matrix and assigns it to a variable A.

Note that:

- The elements within a row of a matrix may be separated by commas as well as a blank.
- The elements of a matrix being entered are enclosed by brackets;
- A matrix is entered in "row-major order" [ie all of the first row, then all of the second row, etc];
- Rows are separated by a semicolon [or a newline], and the elements of the row may be separated by either a comma or a space. [Caution: Watch out for extra spaces!]

When listing a number in exponential form (e.g. 2.34e-9), blank spaces must be avoided. Listing entries of a large matrix is best done in an M-file (see section below), where errors can be easily edited away.

The matrix element located in the i^{th} row and j^{th} column of a is referred to in the usual way:

```
>> A(1,2), A(2,3)
ans =
      2
ans =
      6
>>
```

It's very easy to modify matrices:

```
>> A(2,3) = 10;
```

Building Matrices from Blocks: Large matrices can be assembled from smaller matrix blocks. For example, with matrix A in hand, we can enter the following commands;

```
>> C = [A; 10 11 12];      <== generates a (4x3) matrix
>> [A; A; A];              <== generates a (9x3) matrix
>> [A, A, A];              <== generates a (3x9) matrix
```

As with variables, use of a semicolon with matrices suppresses output. This feature can be especially useful when large matrices are being generated.

If A is a 3-by-3 matrix, then

```
>> B = [ A, zeros(3,2); zeros(2,3), eye(2) ];
```

will build a certain 5-by-5 matrix. Try it.

BUILT-IN MATRIX FUNCTIONS

MATLAB has many types of matrices which are built into the system e.g.,

Function	Description
diag	returns diagonal M.E. as vector
eye	identity matrix
hilb	Hilbert matrix
magic	magic square
ones	matrix of ones
rand	randomly generated matrix
triu	upper triangular part of a matrix
tril	lower triangular part of a matrix
zeros	matrix of zeros

Here are some examples:

Matrices of Random Entries: A 3 by 3 matrix with random entries is produced by typing

```
>> rand(3)
ans =
    0.0470    0.9347    0.8310
    0.6789    0.3835    0.0346
    0.6793    0.5194    0.0535
>>
```

General m by n matrices of random entries are generated with

```
>> rand(m,n);
```

Magic Squares: A magic square is a square matrix which has equal sums along all its rows and columns. For example:

```
>> magic(4)
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
>>
```

The elements of each row and column sum to 34.

Matrices of Ones: The functions

```
eye (m,n)   produces an m-by-n matrix of ones.
eye (n)     produces an n-by-n matrix of ones.
```

Matrices of Zeros: The commands

```
zeros (m,n)   produces an m-by-n matrix of zeros.
zeros (n)     produces an n-by-n one;
```

If A is a matrix, then **zeros(A)** produces a matrix of zeros of the same size as A.

Diagonal Matrices: If x is a vector, **diag(x)** is the diagonal matrix with x down the diagonal.

if A is a square matrix, then **diag(A)** is a vector consisting of the diagonal of A.

What is **diag(diag(A))**? Try it.

MATRIX OPERATIONS

The following matrix operations are available in MATLAB:

Operator	Description	Operator	Description
+	addition	'	transpose
-	subtraction	\	left division
*	multiplication	/	right division
^	power		

These matrix operations apply, of course, to scalars (1-by-1 matrices) as well. If the sizes of the matrices are incompatible for the matrix operation, an error message will result, except in the case of scalar-matrix operations (for addition, subtraction, and division as well as for multiplication) in which case each entry of the matrix is operated on by the scalar.

Matrix Transpose: The transpose of a matrix is the result of interchanging rows and columns. MATLAB denotes the [conjugate] transpose by following the matrix with the single-quote [apostrophe]. For example:

```
>> A'
ans =

    1     4     7
    2     5     8
    3     6     9
>> B = [1+i 2 + 2*i 3 - 3*i]'
B =

    1.0000 - 1.0000i
    2.0000 - 2.0000i
    3.0000 + 3.0000i
>>
```

Matrix Addition/Subtraction: Let matrix “A” have m rows and n columns, and matrix “B” have p rows and q columns. The matrix sum “A + B” is defined only when m equals p and n = q, The result is a n by m matrix having the element-by-element sum of components in A and B.

For example:

```
    >> E = [ 2  3; 4 5.0; 6    7];
    >> F = [ 1 -2; 3 6.5; 10 -45];
    >> E+F
     ans =
        3.0000    1.0000
        7.0000   11.5000
       16.0000  -38.0000
    >>
```

Matrix Multiplication: Matrix multiplication requires that the sizes match. If they don't, an error message is generated.

```
    >> A*B, B*A;
    >> B'*A;
    >> A*A', A'*A;
    >> B'*B, B*B';
```

Scalars multiply matrices as expected, and matrices may be added in the usual way (both are done "element by element):

```
    >> 2*A, A/4;
    >> A + [b,b,b];
```

Example: We can use matrix multiplication to check the “magic” property of magic squares.

```
    >> A = magic(5);
    >> b = ones(5,1);
    >> A*b;              <== (5x1) matrix containing row sums.
    >> v = ones(1,5);
    >> v*A;              <== (1x5) matrix containing column sums.
```

Matrix Functions "any" and "all": There is a function to determine if a matrix has at least one nonzero entry, **any**, as well as a function to determine if all the entries are nonzero, **all**.

```
    >> A = zeros(1,4)
    >> any(A)
    >> D = ones(1,4)
    >> any(D)
    >> all(A)
```

Returning more than One Value: Some MATLAB functions can return more than one value. In the case of **max** the interpreter returns the maximum value and also the column index where the maximum value occurs.

```
    >> [m, i] = max(B)
    >> min(A)
    >> b = 2*ones(A)
    >> A*B
    >> A
```

Round Floating Point numbers to Integers: MATLAB has functions to round floating point numbers to integers. MATLAB has functions to round floating point numbers to integers. These are round, **fix**, **ceil**, and **floor**. The next few examples work through this set of commands and a couple more arithmetic operations.

```
    >> f = [-.5 .1 .5];
    >> round(f)
    >> fix(f)
    >> ceil(f)
    >> floor(f)
```

```
>> sum(f)
>> prod(f)
```

MATRIX-ELEMENT LEVEL OPERATIONS

The matrix operations of addition and subtraction already operate on an element-by-element basis, but the other matrix operations given above do not -- they are matrix operations.

MATLAB has a convention in which a dot in front of the operations,

`*`, `^`, `\`, and `/`,

will force them to perform entry-by-entry multiplication instead of the usual matrix operation. For example

```
[1,2,3,4].*[1,2,3,4]

ans =

     1     4     9    16
```

The same result can be obtained with

```
[1,2,3,4].^2
```

Element-by-element operations are particularly useful for MATLAB graphics.

SYSTEMS OF LINEAR EQUATIONS

One of the main uses of matrices is in representing systems of linear equations. Let:

"A"	A matrix containing the coefficients of a system of linear equations,
"X"	A column vector containing the "unknowns."
"B"	A column vector of "right-hand side constant terms."

The linear system of equations is represented by the matrix equation

A X = B

In MATLAB, solutions to the matrix equations are computed with “matrix division”' operations. More precisely:

```
>> X=A\B
```

is the solution of A*X=B (this can be read “matrix X equals the inverse of matrix A, multiplied by B”) and,

```
>> X=B/A
```

is the solution of x*A=b . For the case of left division, if A is square, then it is factored using Gaussian elimination and these factors are used to solve A*x=b.

Comment for Numerical Gurus: If A is not square, then it is factored using Householder orthogonalization with column pivoting and the factors are used to solve the under- or over- determined system in the least squares sense. Right division is defined in terms of left division by

```
>> b/A=(A' \ b')'
```

NUMERICAL EXAMPLE

In the opening sections of this tutorial we discussed the family of equations given by:

```
>> A = [ 3 -1 0; -1  6 -2; 0 -2 10 ];
>> B = [ 1; 5; 26 ];
```

The solution to these equations is given by:

```
>> X = A \ B
X =
    1.0000
    2.0000
    3.0000
>>
```

and X agrees with our initial observation. We can check the solution by verifying that

```
     A*X      ==> B
and  A*X - B  ==> 0
```

The required MATLAB commands are:

```
>> A*X
ans =
    1.0000
    5.0000
   26.0000
>> A*X - B
ans =
   1.0e-14 *
    0.0444
   -0.2665
   -0.3553
>>
```

Note : If there is no solution, a "least-squares" solution is provided (i.e., A*X - B is as small as possible).

CONTROL STRUCTURES

Control-of-flow in MATLAB programs is achieved with logical/relational constructs, branching constructs, and a variety of looping constructs.

Colon Notation: A central part of the MATLAB language syntax is the "colon operator," which produces a list. For example:

```
>> format compact
>> -3:3
ans =
    -3    -2    -1     0     1     2     3
```

The default increment is by 1, but that can be changed. For example:

```
>> x = -3 : .3 : 3
x =
  Columns 1 through 7
   -3.0000   -2.7000   -2.4000   -2.1000   -1.8000   -1.5000   -1.2000
  Columns 8 through 14
   -0.9000   -0.6000   -0.3000         0    0.3000    0.6000    0.9000
  Columns 15 through 21
    1.2000    1.5000    1.8000    2.1000    2.4000    2.7000    3.0000
>>
```

This can be read: "x is the name of the list, which begins at -3, and whose entries increase by .3, until 3 is surpassed." You may think of x as a list, a vector, or a matrix, whichever you like.

In our third example, the following statements generate a table of sines.

```
>> x = [0.0:0.1:2.0]' ;
>> y = sin(x);
>> [x y]
```

Try it. Note that since sin operates entry-wise, it produces a vector y from the vector x.

The colon notation can also be combined with the earlier method of constructing matrices.

```
>> a = [1:6 ; 2:7 ; 4:9]
```

RELATIONAL AND LOGICAL CONSTRUCTS

The relational operators in MATLAB are

```
Operator      Description
===================================
       <      less than
       >      greater than
      <=      less than or equal
      >=      greater than or equal
      ==      equal
      ~=      not equal.
===================================
```

Note that ``=" is used in an assignment statement while ``==" is used in a relation.

Relations may be connected or quantified by the logical operators

```
Operator      Description
===================================
       &      and
       |      or
       ~      not.
===================================
```

When applied to scalars, a relation is actually the scalar 1 or 0 depending on whether the relation is true or false (indeed, throughout this section you should think of 1 as true and 0 as false). For example

```
>> 3 < 5
ans =
     1
>> a = 3 == 5
a =
     0
>>
```

When logical operands are applied to matrices of the same size, a relation is a matrix of 0's and 1's giving the value of the relation between corresponding entries. For example:

```
>> A = [ 1 2; 3 4 ];
>> B = [ 6 7; 8 9 ];
>> A == B
ans =
   0    0
   0    0
```

```
>> A < B
ans =

     1     1
     1     1
>>
```

To see how the other logical operators work, you should also try

```
>> ~A
>> A&B
>> A & ~B
>> A | B
>> A | ~A
```

The for statement permits any matrix to be used instead of 1:n.

BRANCHING CONSTRUCTS

MATLAB provides a number of language constructs for branching a program's control of flow.

If-end Construct: The most basic construct is:

```
if <condition>,
   <program>
end
```

Here the condition is a logical expression that will evaluate to either true or false (i.e., with values 1 or 0). When the logical expression evaluates to 0, the program control moves on to the next program construction. You should keep in mind that MATLAB regards A==B and A<=B as functions with values 0 or 1.

Example :

```
>> a = 1;
>> b = 2;
>> if a < b,
      c = 3;
   end;
>> c

c =

     3

>>
```

If-else-end Construct: Frequently, this construction is elaborated with

```
if <condition1>,
   <program1>
else
   <program2>
end
```

In this case if condition is 0, then program2 is executed.

If-elseif-end Construct: Another variation is

```
if <condition1>,
      <program1>
   elseif
      <condition2>,
      <program2>
   end
```

Now if condition1 is not 0, then program1 is executed, if condition1 is 0 and if condition2 is not 0, then program2 is executed, and otherwise control is passed on to the next construction.

LOOPING CONSTRUCTS

For Loops: A for loop is a construction of the form

```
   for i= 1 : n,
      <program>,
   end
```

The program will repeat <program> once for each index value i = 1, 2 .... n. Here are some examples of MATLAB's for loop capabilities:

Example : The basic for loop

```
   >> for i = 1 : 5,
        c = 2*i
    end
   c =
        2

     ..... lines of output removed ...

   c =
        10
   >>
```

computes and prints "c = 2*i" for i = 1, 2, ... 5.

Example : For looping constructs may be nested.

Here is an example of creating a matrices contents inside a nested for loop:

```
   >> for i=1:10,
         for j=1:10,
             A(i,j) = i/j;
         end
      end
   >> A
```

There are actually two loops here, with one nested inside the other; they define

```
   A(1,1),  A(1,2), A(1,3) ... A(1,10), A(2,1), ... A(10,10)
```

in that order.

Example : MATLAB will allow you to put any vector in place of the vector 1:n in this construction. Thus the construction

```
   >> for i = [2,4,5,6,10],
      <program>,
   end
```

is perfectly legitimate.

In this case program will execute 5 times and the values for the variable i during execution are successively, 2,4,5,6,10.

Example : The MATLAB developers went one step further. If you can put a vector in, why not put a matrix in? So, for example,

```
>> for i=magic(7),
    <program>,
end
```

is also legal. Now the program will execute 7 (=number of columns) times, and the values of i used in program will be successively the columns of magic(7).

While Loops: A while loop is a construction of the form

```
while <condition>,
     <program>,
end
```

where condition is a MATLAB function, as with the branching construction. The program program will execute successively as long as the value of condition is not 0. While loops carry an implicit danger in that there is no guarantee in general that you will exit a while loop. Here is a sample program using a while loop.

```
function l=twolog(n)

% l=twolog(n). l is the floor of the base 2
% logarithm of n.

l=0;
m=2;
while m<=n
  l=l+1;
  m=2*m;
end
```

SOME NOTES

A relation between matrices is interpreted by while and if to be true if each entry of the relation matrix is nonzero. Hence, if you wish to execute statement when matrices A and B are equal you could type

```
if A == B
  statement
end
```

but if you wish to execute statement when A and B are not equal, you would type

```
if any(any(A ~ B))
  ( statement )
end
```

or, more simply,

```
if A == B else
  { statement }
end
```

SUB-MATRICES

We have already seen how colon notation can be used to generate vectors. A very common use of the colon notation is to extract rows, or columns, as a sort of "wild-card" operator which produces a default list. For example,

```
A(1:4,3)
```

is the column vector consisting of the first four entries of the third column of A . A colon by itself denotes an entire row or column. So, for example:

```
A(:,3)
```

is the third column of A , and

```
A(1:4,:)
```

is the first four rows of A. Arbitrary integral vectors can be used as subscripts. The statement

```
A(:,[2 4])
```

contains as columns, columns 2 and 4 of matrix A. This subscripting scheme can be used on both sides of an assignment statement:

```
A(:,[2 4 5]) = B(:,1:3)
```

replaces columns 2,4,5 of matrix A with the first three columns of matrix B. Note that the "entire" altered matrix A is printed and assigned. Try it.

Columns 2 and 4 of A can be multiplied on the right by the 2-by-2 matrix [1 2;3 4]:

```
A(:,[2,4]) = A(:,[2,4])*[1 2;3 4]
```

Once again, the entire altered matrix is printed and assigned.

When you a insert a 0-1 vector into the column position then the columns which correspond to 1's are displayed.

```
>> V=[0 1 0 1 1]
>> A(:,V)
>> A(V,:)
```

MATLAB SCRIPTING FILES

PROGRAM DEVELOPMENT WITH M-FILES

MATLAB statements can be prepared with any editor, and stored in a file for later use. Such a file is referred to as a script, or an "m-file" (since they must have a name extension of the form filename.m). Writing m-files will enhance your problem solving productivity.

Suppose that we create a program file

```
myfile.m
```

in the MATLAB language. The commands in this file can be exectued by simply giving the command

```
>> myfile
```

from MATLAB. The MATLAB statements will run like any other MATLAB function. You do not need to compile the program since MATLAB is an interpretative (not compiled) language.

An M-file can reference other M-files, including referencing itself recursively.

Example : Using your favorite editor, create the following file, named sketch.m:

```
[x y] = meshgrid(-3:.1:3, -3:.1:3);
z = x.^2 - y.^2;
mesh(x,y,z);
```

Then start MATLAB from the directory containing this file, and enter

```
>> sketch
```

The result is the same as if you had entered the three lines of the file, at the prompt. See Figure 1.

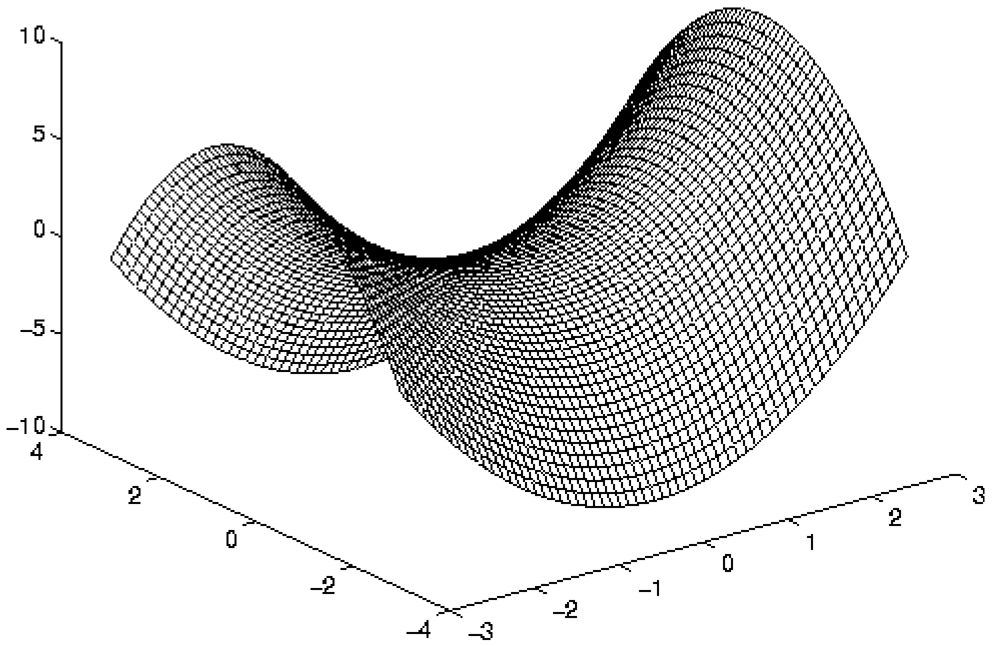


Figure 1 : Mesh generated by commands in M-file sketch.m

Example : You can also enter data this way: if a file named mymatrix.m in the current working directory contains the lines

```
A = [2 3 4; 5 6 7; 8 9 0]
inv(A)
quit
```

then the command

```
>> mymatrix
```

reads that file, generates A and the inverse of A, and quits matlab [quitting is optional]. You may prefer to do this, if you use the same data repeatedly, or have an editor that you like to use. You can use Control-Z to suspend MATLAB, then edit the file, and then use "fg" to bring matlab back to the foreground, to run it.

M-File Examples : MATLAB comes with a lot of "m-file" examples! To find their location on your computer, use

```
>> path
```

This will also lead you to some really nifty demos.

FUNCTIONS AND FUNCTION FILES

Function files provide extensibility to MATLAB by allowing you to create new problem-specific functions having the same status as other built-in MATLAB functions.

Functions are like scripts, but for the purposes of computational speed, are compiled the first time.

Example 1 : Here we create a file named "sqroot.m" containing the following lines.

```
function sqroot(x)
% SQROOT Compute square root by Newton's method

% Initial guess

xstart = 1;

% Iteration loop to compute square root

for i = 1:100
    xnew = ( xstart + x/xstart)/2;
    disp(xnew);
    if abs(xnew - xstart)/xnew < eps, break, end;
    xstart = xnew;
end
```

The first line declares the function name, input arguments, and output arguments; without this line the file would be a script file. Notice that the file name corresponds to the function name.

Variables in a function file are local by default.

The first few lines of a function should be comment statements so that they can be used in online-help e.g.,

```
>> help sqroot

    SQROOT Compute square root by Newton's method
>>
```

With the function m-file in place, we can give the MATLAB commands

```
>> format long
>> sqroot(20)
```

and the square root of 20 will be computed, as in

```
>> sqroot(20)
     10.50000000000000

     6.20238095238095

     4.71347454528837

     4.47831444547438

     4.47214021706570

     4.47213595500161

     4.47213595499958
     4.47213595499958
  >>
```

Example 2 : A function may have multiple output arguments. For example:

```
  function  [mean, stdev] = stat(x)

  % STAT  Mean and standard deviation
  %      For a vector x, stat(x) returns the
  %      mean and standard deviation of  x.
  %      For a matrix x, stat(x) returns two row vectors containing,
  %      respectively, the mean and standard deviation of each column.

  [m  n] = size(x);
  if m == 1
     m = n;     % handle case of a row vector
  end
  mean  = sum(x)/m;
  stdev = sqrt(sum(x.^ 2)/m - mean.^2);
```

Notice that mean and stdev are both computed inside the function body.

Once this is placed in the m-file stat.m, a MATLAB command

```
  >> [xm, xd] = stat(x)
```

will assign the mean and standard deviation of the entries in the vector x to xm and xd, respectively. For example, the script of code:

```
  >> y = [1:10];
  >> [ym, yd] = stat(y)

  ym =
     5.5000

  yd =
     2.8723

  >>
```

shows computation of the mean and standard deviation of the integers one through ten.

Single assignments can also be made with a function having multiple output arguments. For example,

```
  >> xm = stat(x)
```

(no brackets needed around xm) will assign the mean of x to xm.
Example 3 : The following function, which gives the greatest common divisor of two integers via the Euclidean algorithm, illustrates the use of an error message (see the next section).

```
  function  a = gcd(a,b)

  % GCD  Greatest common divisor
  %      gcd(a,b) is the greatest common divisor of
  %      the integers a and b, not both zero.

  a = round(abs(a));
  b = round(abs(b));
```

```
if  a == 0 & b == 0
   error('The gcd is not defined when both numbers are zero')
else
   while b  ~= 0
      r = rem(a,b);
      a = b;  b = r;
end
end
```

For example,

```
>> a = gcd (25,45)

a =
     5

>>
```

The prime factorizations of 25 and 45 are 5.5 and 5.3.3, and so the greatest common divisor is 5.

BATCH JOBS

MATLAB is most often used interactively, but "batch" or "background" jobs can be performed as well. Debug your commands interactively and store them in a file (`script.m', for example). To start a background session from your input file and to put the output and error messages into another file (`script.out', for example), enter this line at the system prompt:

```
nice matlab < script.m >& script.out  &
```

You can then do other work at the machine or logout while MATLAB grinds out your program. Here's an explanation of the sequence of commands above.

The "nice" command lowers matlab's priority so that interactive users have first crack at the CPU. You must do this for noninteractive MATLAB sessions because of the load that number--crunching puts on the CPU.

The "< script.m" means that input is to be read from the file script.m. Details on scripting-files may be found below.

The ">& script.out" is an instruction to send program output and error output to the file script.out.
It is important to include the first ampersand (&) so that error messages are sent to your file rather than to the screen -- if you omit the ampersand then your error messages may turn up on other people's screens and your popularity will plummet.)

Finally, the concluding ampersand (&) puts the whole job into background.
Of course, the file names used above are not important -- these are just examples to illustrate the format of the command string.

GRAPHICS

Two- and three-dimensional MATLAB graphs can be given titles, have their axes labeled, and have text placed within the graph. The basic functions are:

```
Function                    Description
============================================================================
plot(x,y)                   plots y vs x
plot(x,y1,x,y2,x,y3)        plots y1, y2 and y3 vs x on the same graph
```

```
xlabel('x axis label')   labels x axis
ylabel('y axis label')   labels y axis
title ('title of plot')  puts a title on the plot
gtext('text')            activates the use of the mouse to position a
                         crosshair on the graph, at which point the
                         'text' will be placed when any key is pressed.

print filename.ps        saves the plot as a black and white postscript file
============================================================================
```

When in the graphics screen, pressing any key will return you to the command screen while the command (show graph)

```
>> shg
```

will return you to the current graphics screen. If your machine supports multiple windows with a separate graphics window, you will want to keep the graphics window exposed-but moved to the side-and the command window active.

MATLAB graphics windows will contain one plot by default. The command "subplot" can be used to partition the screen so that up to four plots can be viewed simultaneously. For more information, type

```
>> help subplot
```

TWO-DIMENSIONAL PLOTS

The plot command creates linear x-y plots; if x and y are vectors of the same length, the command plot(x,y) opens a graphics window and draws an x-y plot of the elements of x versus the elements of y.

Example : Let's draw the graph of the sine function over the interval -4 to 4 with the following commands:

```
>> x = -4:.01:4; y = sin(x); plot(x,y)
>> grid
>> xlabel('x')
>> ylabel('sin(x)')
```

The vector x is a partition of the domain with meshsize 0.01 while y is a vector giving the values of sine at the nodes of this partition (recall that sin operates entrywise). The result is shown in Figure 2.

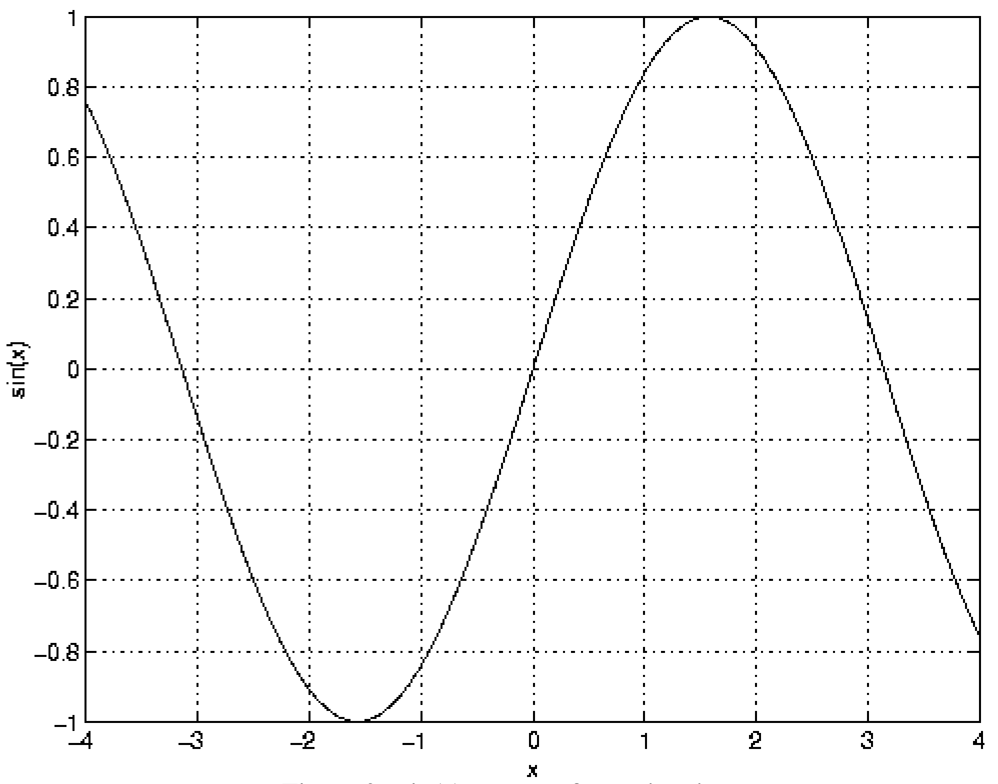


Figure 2 : sin(x) versus x from -4 to 4

Example : Plots of parametrically defined curves can also be made. For example,

```
>> t=0:.001:2*pi; x=cos(3*t); y=sin(2*t); plot(x,y)
```

Example : Two ways to make multiple plots on a single graph are illustrated by

```
>> x  = 0:.01:2*pi;y1=sin(x);y2=sin(2*x);
>> y3 = sin(4*x);plot(x,y1,x,y2,x,y3)
```

and by forming a matrix Y containing the functional values as columns

```
>> x=0:.01:2*pi; Y=[sin(x)', sin(2*x)', sin(4*x)']; plot(x,Y)
```

Another way is with hold. The command hold freezes the current graphics screen so that subsequent plots are superimposed on it. Entering hold again releases the ``hold.'' The commands hold on and hold off are also available in version 4.0.

Example : One can override the default linetypes and pointtypes. For example, the command sequence

```
>> x = 0:.01:2*pi; y1=sin(x); y2=sin(2*x); y3=sin(4*x);
>> plot(x,y1,'--',x,y2,':',x,y3,'+')
>> grid
>> title ('Dashed line and dotted line graph')
```

renders a dashed line and dotted line for the first two graphs while for the third the symbol is placed at each node. See Figure 3.

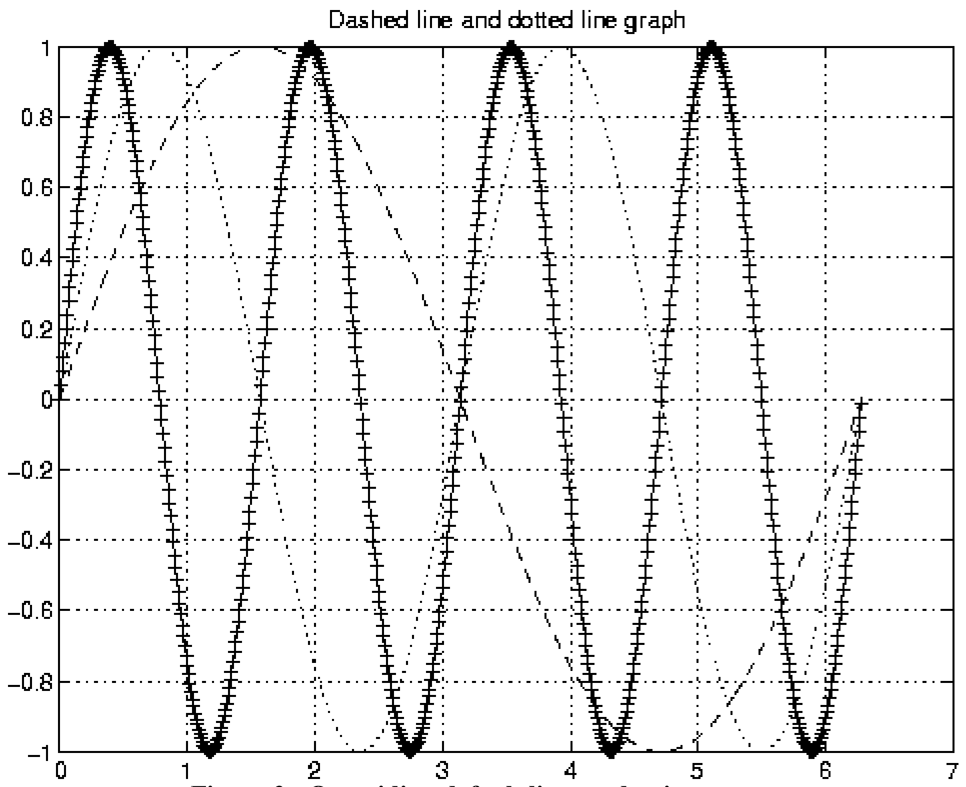


Figure 3 : Overriding default line- and point- types

The line- and mark-types are

```
=============================================================
Linetypes : solid  (-), dashed (-). dotted (:), dashdot (-.)
Marktypes : point  (.),   plus (+),   star (*,  circle   (o),
            x-mark (x)
=============================================================
```

See help plot for line and mark colors.

TWO-DIMENSIONAL CONTOUR PLOTS

Functions of two variables may be plotted, as well, but some "setup" is required!

```
>> [x y] = meshgrid(-3:.1:3, -3:.1:3);
>> z = x.^2 - y.^2;
>> mesh(x,y,z)
>> surf(x,y,z)
>> contour(z)
```

Notice how in the second command, the dot operator is inserted to computer the difference of

z(x,y) = x^2 - y^2

at the matrix element level.

The result of these commands is shown in Figure 4.

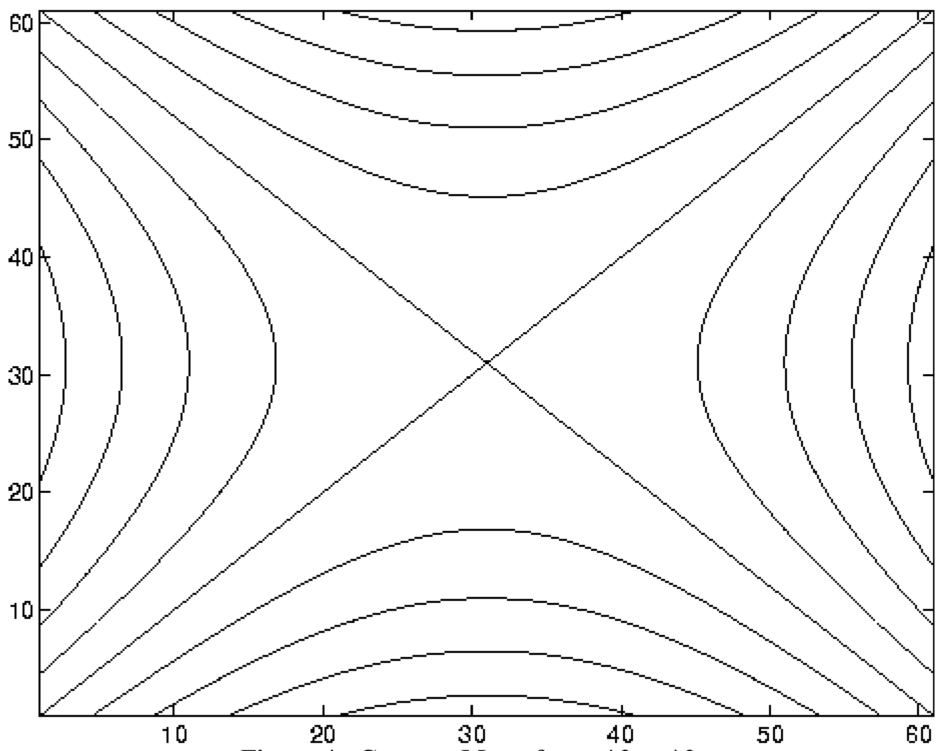


Figure 4 : Contour Map of z = x^2 - y^2

THREE-DIMENSIONAL GRAPHICS

Three dimensional mesh surface plots are drawn with the functions

```
Function      Description
==============================================================================
mesh(z)       Creates a three-dimensional perspective plot of the elements of
              the matrix "z". The mesh surface is defined by the z-coordinates
              of points above a rectangular grid in the x-y plane.
==============================================================================
```

To draw the graph of a function

z = f(x,y)

over a rectangle, one first defines vectors xx and yy, which give partitions of the sides of the rectangle.

With the function meshdom (mesh domain; called meshgrid in version 4.0) one then creates a matrix x, each row of which equals xx and whose column length is the length of yy, and similarly a matrix y, each column of which equals yy, as follows:

```
>> [x,y] = meshdom(xx,yy);
```

One then computes a matrix z, obtained by evaluating f entrywise over the matrices x and y, to which mesh can be applied.

Example : Suppose that we want to draw a graph of over the square [-2,2] x [-2,2] as follows:

```
>> xx = -2:.1:2;
>> yy = xx;
>> [x,y] = meshdom(xx,yy);
>> z = exp(-x.^2 - y.^2);
>> mesh(z)
```

The result is shown in Figure 5.

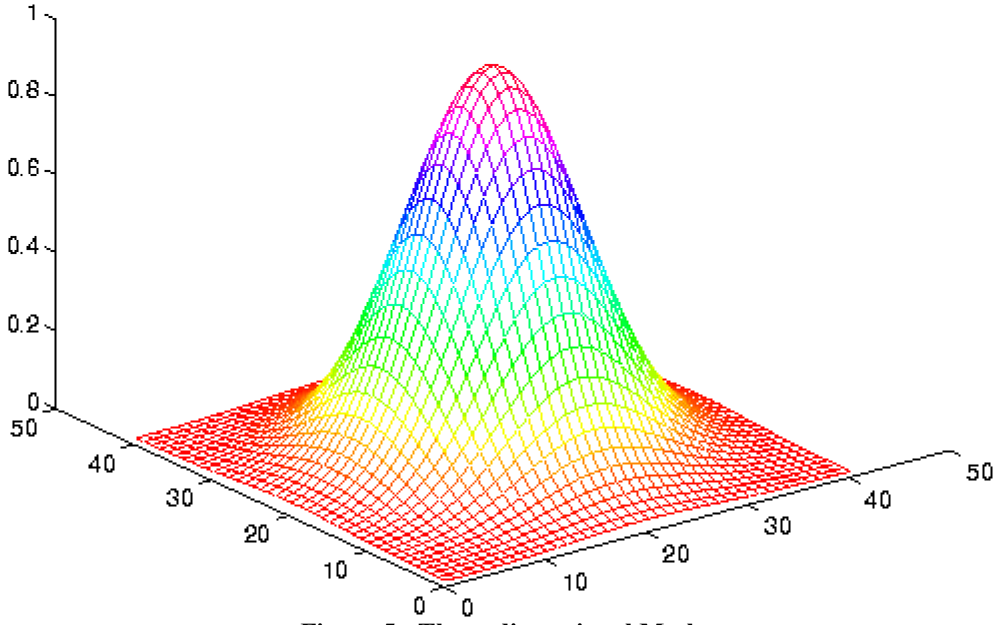


Figure 5 : Three-dimensional Mesh

Of course one could replace the first three lines of the preceding with

```
>> [x,y] = meshdom(-2:.1:2, -2:.1:2);
```

See the User's Guide for further details regarding mesh. Also see information on functions plot3, and surf.

CREATING POSTCRIPT HARDCOPIES

To create a black-and-white postcript file of a MATLAB figure just type

```
>> print name-of-figure.ps
```

Color postscript files can be generated with

```
>> print -dpsc name-of-figure.ps
```

Similarly, try the command

```
>> print -dgif8 name-of-figure.gif
```

to create a color "gif" file (i.e., graphics image format file). Online information can be obtained by typing "help print"

The UNIX tool "xv" can then be used to convert the postscript file into a "gif" file format, suitable for reading by WWW browsers.

ENGINEERING APPLICATIONS

EXAMPLE 1: TEMPERATURE CONVERSION PROGRAM

Problem Statement : The relationship between temperature measured in Fahrenheit (T_f) and temperature Celcius (T_c) is given by the equation:

9.0

```
T_f = --- T_c + 32
      5.0
```

Write a m-file that computes and plots the temperature conversion relationship over the range -50 through 100 degrees Celcius.

You should make sure the vertical and horizontal axes of your graph are properly labeled, and the plot has a suitable title. Your m-file should contain comment statements identifying the author, the purpose of the m-file, and a description of the variables and arrays used in the problem solution.

Problem Solution: Here is a m-file containing a first cut solution:

```
% ===================================================================
% Temperature Conversion Program for Celcius to Fahrenheit
% covering the range 0 through 100 degrees Celcius.
%
% Written By : Mark Austin                                March 1997
% ===================================================================

nopoints = 151;
temp = zeros(nopoints,2);

for i = 1 : nopoints
    temp(i,1) = i - 51;
    temp(i,2) = 9*temp(i,1)/5 + 32;
end

plot(temp(:,1), temp(:,2));
grid;
xlabel('Temperature (Celcius)');
ylabel('Temperature (Fahrenheit)');
title('Fahrenheit versus Celcius Temperature Conversion');
```

and the figure generated by this m-file.

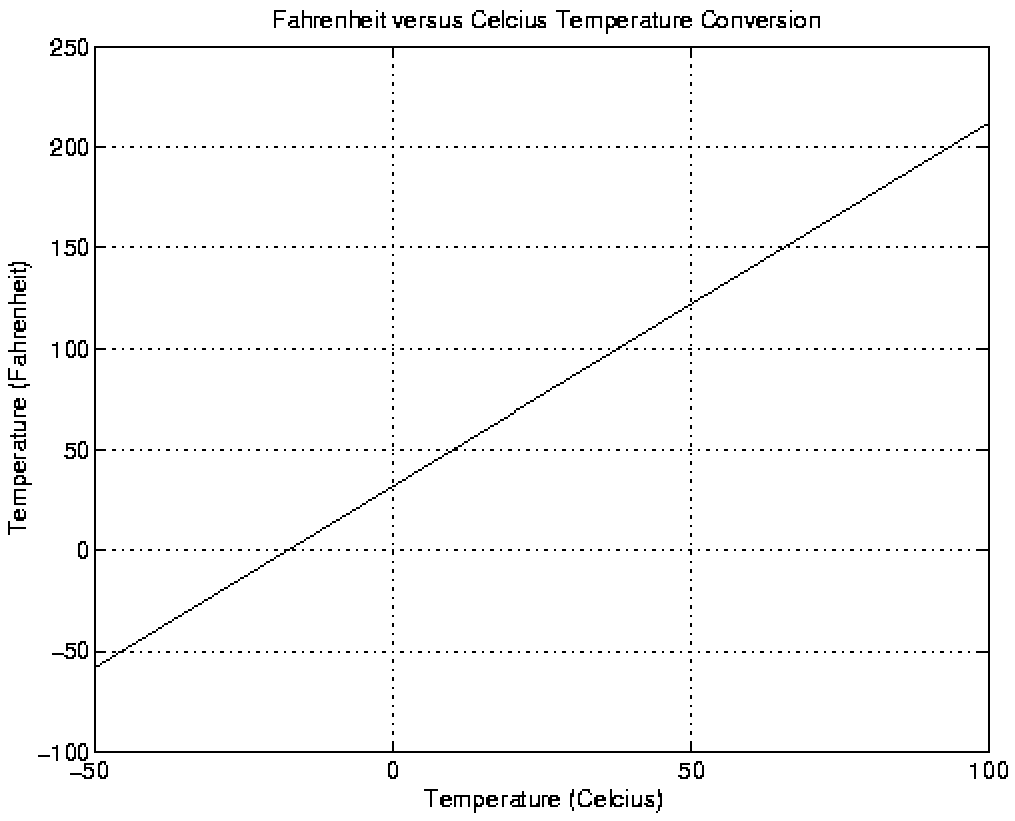


Figure 6 : Fahrenheit versus Celcius Temperature Conversion

Comments: Points to note are:

The first and second columns of matrix "temp" store the temperature in Celcius and Fahrenheit, respectively.

In the plot command:

```
plot(temp(:,1), temp(:,2));
```

the data points in column one of matrix "temp" are plotted against the data points in column two of "temp."

The temperature arrays can also be constructed by using colon notation and matrix-element operations. For example, the command

```
TempC = [ -50:100 ];
```

will define a one-dimensional array of 151 elements initialized to -50 through 100. A corresponding array of temperatures in Fahrenheit can be assembled by simply writing

```
TempF = TempC*(5/9) + 32; % Arithmetic operations with vectors
```

The slightly modified plot command is:

```
plot(TempC, TempF)      % Simple plot
```

EXAMPLE 2 : FREE VIBRATION RESPONSE OF UNDAMPED SDOF SYSTEM

Problem Statement: The time-history free vibration response of an undamped single-degree of freedom oscillator is given by:

$$x(t) = x(0) \cos (wt) + \frac{v(o)}{w} \sin (wt)$$

where

t = time.
x(t) = displacement at time t.
v(t) = velocity at time t.
w = sqrt(k/m)
= circular natural frequency of the system.
m = the system mass.
k = the system stiffness.

The natural period of this system is

T = 2*pi*sqrt(m/k).

Now let's suppose that the system has mass = 1, stiffness = 10, and an initial displacement and velocity x(0) = 10 and v(0) = 10.

Write a MATLAB m-file that will compute and plot the "displacement versus time" (i.e., x(t) versus t) and "velocity versus time" (i.e., v(t) versus t) for the time interval 0 through 10 seconds. To ensure that your plot will be reasonable smooth, choose an increment in your displacement and velocity calculations that is no larger than 1/10 of the system period T. (An elegant solution to this problem would do this automatically).

Solution : Here is a m-file containing a first cut solution:

```
% ==================================================================
% Compute dynamic response of sdof system.
%
% Written By : Mark Austin                              March 1997
% ==================================================================

% Setup array for storing and plotting system response

nopoints = 501;
response = zeros(nopoints,3);

% Problem parameters and initial conditions

mass  = 1;
stiff = 10;
w     = sqrt(stiff/mass);
dt    = 0.02;

displ0    =  1;
velocity0 = 10;

% Compute displacement and velocity time history response

for i = 1 : nopoints + 1
    time = (i-1)*dt;
    response(i,1) = time;
    response(i,2) =  displ0*cos(w*time)   + velocity0/w*sin(w*time);
    response(i,3) = -displ0*w*sin(w*time) + velocity0*cos(w*time);
end
```

```
% Plot displacement versus time

plot(response(:,1), response(:,2));
hold;

% Plot velocity versus time

plot(response(:,1), response(:,3));

grid;
xlabel('Time (seconds)');
ylabel('Displacement (m) and Velocity (m/sec)');
title('Time-History Response for SDOF Oscillator');
```

The graphical output is:

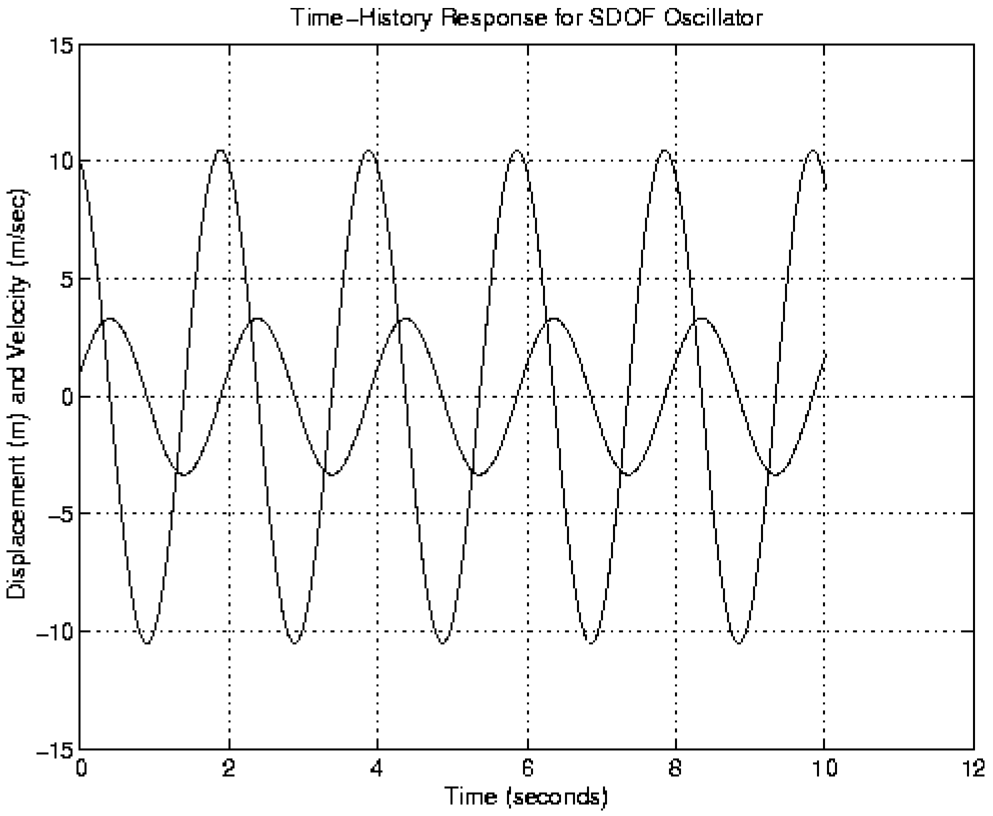


Figure 7 : Time-History Response of SDOF Oscillator

Comments: Points to note are:

The velociy of the SDOF system versus time is given by the derivative of the displacement with respect to time. In mathematical terms:

```
         d
v(t) = -- x(t) = -displ0*w*sin(w*t) + velocity0*cos(w*t);
        dt
```

where displ0 and velocity0 are the displacement and velocity of the system at t = 0.

The natural period of this system is

```
T = 2*pi*sqrt(m/k)
  = 6.282/sqrt(10)
```

= 2.0 seconds.

The timestep increment dt = 0.02 seconds easily satisfies the stated criteria for a smooth graph.

The first column of response stores the time, and the second and third columns of response the displacement and velocity, respectively.

A second way of computing the time, displacement, and velocity vectors is

```
time     = 0.0:0.02:10;
displ    = displ0*cos(w*time)    + velocity0/w*sin(w*time);
velocity = -displ0*w*sin(w*time) + velocity0*cos(w*time);
```

The first statement generates a (1x501) matrix called time having the element values 0, 0.02, 0.04 .... 10.0. The dimensions of matrices "displ" and "velocity" are inferred from the dimensions of "temp" with the values of the matrix elements given by the evaluation of formulae on the right-hand side of the assignment statements.

Now the plots can be generated with:

```
plot(time, displ);
hold;
plot(time, velocity);
```

So how do we know that these graphs might be correct? First, notice that at t = 0 seconds, the displacement and velocity graphs both match the stated initial conditions. Second, note that because the initial velocity is greater than zero, we expect the displacement curve to initially increase. It does. A final point to note is the relationship between the displacement and velocity. When the oscillator displacemnt is at either its maximum or minimum value, the mass will be at rest for a short time. In mathematical terms, peak values in the displacement curve correspond to zero values in the velocity curve.

EXAMPLE 3 : STATISTICAL ANALYSIS OF EXPERIMENTAL DATA

Problem Statement : Suppose that the concentration of spores of pollen per sq cm are measured over a 15 day period, and stored in a datafile expt.dat.

```
 1  12
 2  35
 3  80
 4  120
 5  280
 6  290
 7  360
 8  290
 9  315
10  280
11  270
12  190
13  90
14  85
15  66
```

The first and second columns of expt.dat store the "day of the experiment" and the "measured pollen count," respectively.

Write a MATLAB program that will:

Read the contents of the data file into an array.
Create a two-dimensional bar plot showing the "pollen count" versus "day"
Compute the mean and standard deviation of the pollen count for the duration of the experiment.

Plot and label dashed-lines of the mean pollen count plus and minus one standard deviation.
Solution : The m-file

```
%  ==============================
%  Analysis of Experimental Data.
%  ==============================

% Store experimental results in array

load expt.dat

% Generate bar plot of experimental results

bar(expt(:,1), expt(:,2),'b')
xlabel('Day of Expt');
ylabel('Pollen Count');

% Compute terms from experimental results.

[xm, xd] = stat(expt(:,2))

% Create and display mean value of pollen count

mean_minus = xm(1,1) - xd(1,1);
mean_plus  = xm(1,1) + xd(1,1);

data = [  1, xm(1,1), mean_minus, mean_plus;
         15, xm(1,1), mean_minus, mean_plus ];

hold;
plot (data(:,1), data(:,2), 'b');
plot (data(:,1), data(:,3), 'b:');
plot (data(:,1), data(:,4), 'b:');

text(1,   xm(1,1) + 10,'Mean Pollen Count');
text(1,mean_minus + 10,'Mean - Std');
text(1, mean_plus + 10,'Mean + Std');
```

generates the textual output:

```
    >> expt

    xm =

       184.2000

    xd =

       114.2309
    >>
```

and the graphical output:

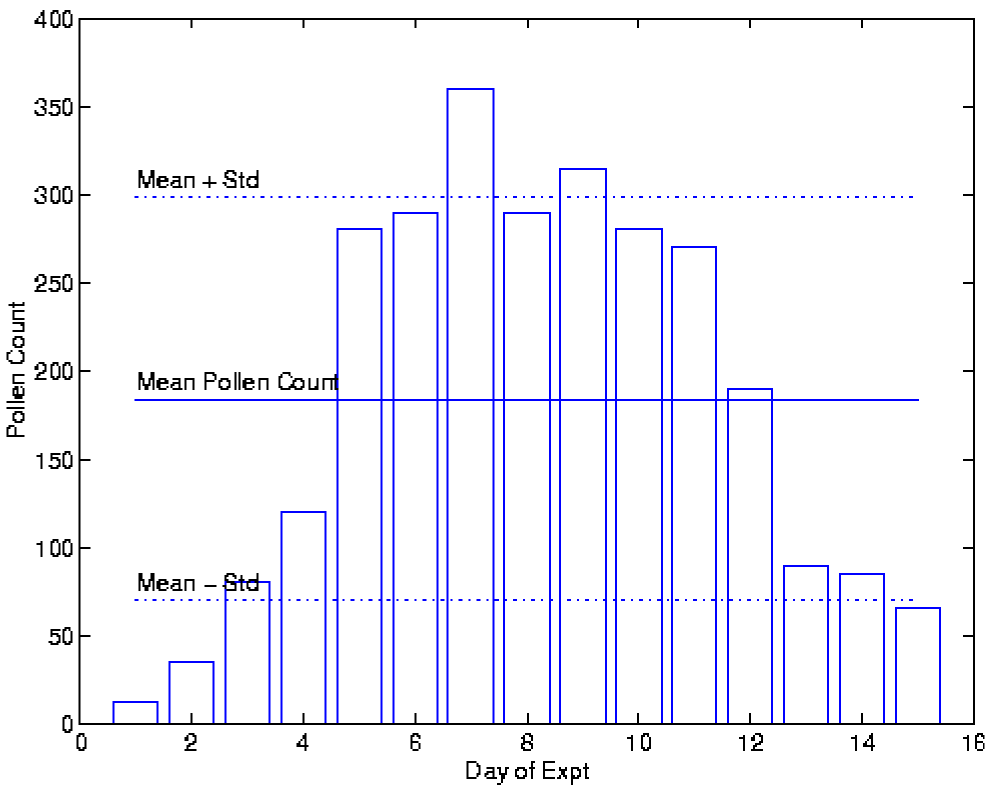


Figure 8 : Pollen Count versus Day of Experiment

Comments:

Our solution to this problem takes advantage of data files, and other user-defined functions located in the same working directory.
More precisely, the command:

```
load expt.dat;
```

loads the content of datafile "expt.dat" into the array "expt". Then the command:

```
[xm, xd] = stat(expt(:,2))
```

calls the user-defined function "stat" defined in stat.m to compute the mean and standard deviation of data stored in the second column of expt.

EXAMPLE 4 : LEAST SQUARES ANALYSIS OF GEOLOGICAL BORINGS

Problem Statement : Figure 1 is a three-dimensional view of a 2 km by 2 km site that is believed to overlay a thick layer of mineral deposits.

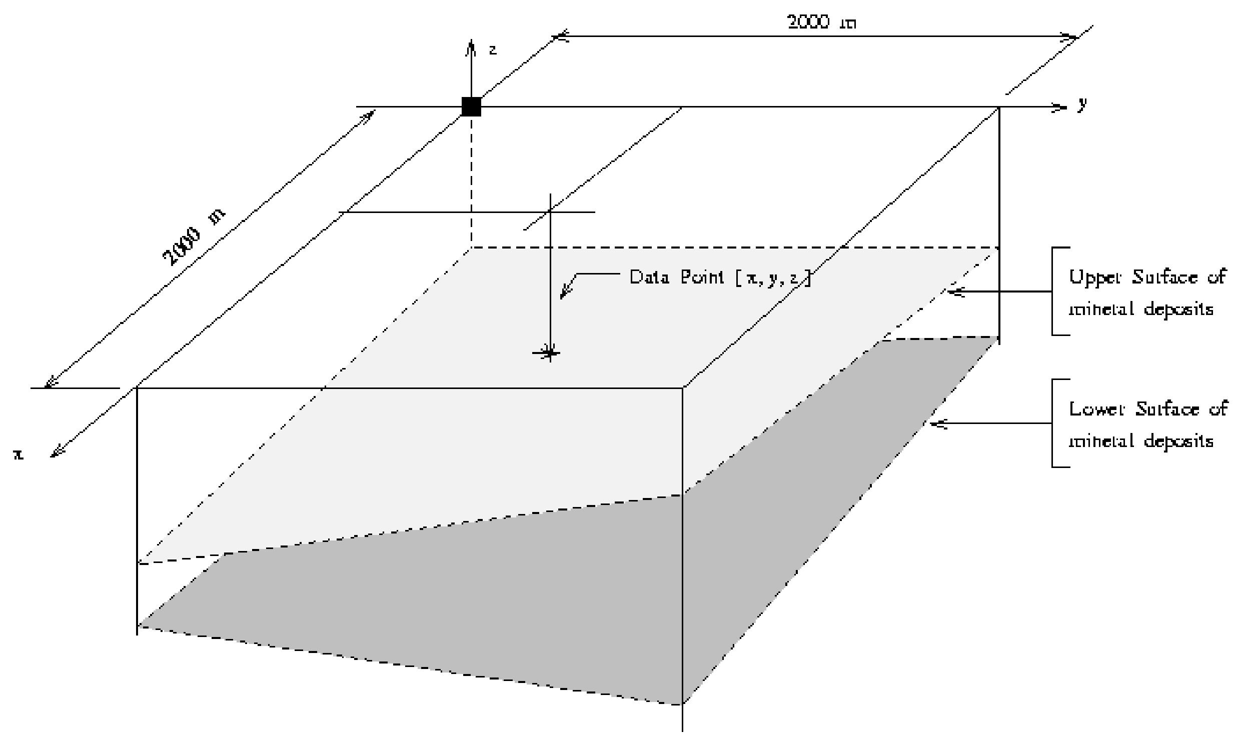

2000 m
z
y
2000 m
x
Data Point [x, y, z]
Upper Surface of mineral deposits
Lower Surface of mineral deposits

Figure 9: Three-Dimensional View of Mineral Deposits

To create a model of the mineral deposit profile and establish the economic viability of mining the site, a preliminary subsurface exploration consisting of 16 bore holes is conducted. Each bore hole is drilled to approximately 45 m, with the upper and lower boundaries of mineral deposits being recorded. The bore hole data is as follows:

```
Borehole  [ x, y ] coordinate      [ upper, lower ] mineral surfaces
====================================================================
       1  [   10.0 m,   10.0 m ]   [ -30.5 m,  -40.5 m ]
       2  [  750.0 m,   10.0 m ]   [ -29.0 m,  -39.8 m ]
       3  [ 1250.0 m,   10.0 m ]   [ -28.0 m,  -39.3 m ]
       4  [ 1990.0 m,   10.0 m ]   [ -26.6 m,  -38.5 m ]
       5  [   10.0 m,  750.0 m ]   [ -34.2 m,  -41.4 m ]
       6  [  750.0 m,  750.0 m ]   [ -32.8 m,  -40.6 m ]
       7  [ 1250.0 m,  750.0 m ]   [ -31.8 m,  -40.1 m ]
       8  [ 1990.0 m,  750.0 m ]   [ -30.3 m,  -39.4 m ]
       9  [   10.0 m, 1250.0 m ]   [ -36.7 m , -42.0 m ]
      10  [  750.0 m, 1250.0 m ]   [ -35.2 m,  -41.2 m ]
      11  [ 1250.0 m, 1250.0 m ]   [ -34.2 m,  -40.7 m ]
      12  [ 1990.0 m, 1250.0 m ]   [ -32.8 m,  -40.0 m ]
      13  [   10.0 m, 1990.0 m ]   [ -40.4 m,  -42.8 m ]
      14  [  750.0 m, 1990.0 m ]   [ -39.0 m,  -42.1 m ]
      15  [ 1250.0 m, 1990.0 m ]   [ -38.0 m,  -41.6 m ]
      16  [ 1990.0 m, 1990.0 m ]   [ -36.5 m,  -40.9 m ]
```

With the borehole data collected, the next step is to create a simplified three-dimensional computer model of the site and subsurface mineral deposits. The mineral deposits will be modeled as a single six-sided object. The four vertical sides are simply defined by the boundaries of the site. The upper and lower sides are to be defined by a three-dimensional plane

$$z(x,y) = a_o + a_1 x + a_2 y$$

where coefficients a_o, a_1, and a_2 correspond to minimum values of

$$S (a_o, a_1, a_2) = \sum_{i=1}^{i=N} [z_i - z(x_i, y_i)]^2$$

Things to do:

Show that minimum value of $S(a_o, a_1, a_2)$ corresponds to the solution of a family of (3x3) matrix equations

Write a matlab m-file that will create three-dimensional plots of the borehole data at the lower and upper surfaces. See the matlab function griddata().

Write a matlab m-file that will setup and solve the matrix equations derived in part 1 for the upper and lower mineral planes.

Compute and print the average depth and volume of mineral deposits enclosed within the site.
Note : You should find that the equation of the upper surface is close to

$$z(x,y) = -30.5 + x/500 - y/200$$

and lower surface close to

$$z(x,y) = -40.5 + x/1000 - y/850.$$

Solution : The least squares solution corresponds to the minimum value of function $S(a_o, a_1, a_2)$. At the minimum function value, we will have

$$\frac{dS}{da_o} = \frac{dS}{da_1} = \frac{dS}{da_2} = 0.0$$

In matrix form, the three three equations with three unknowns are:

$$\begin{bmatrix} N & sum\_X & sum\_Y \\ sum\_X & sum\_X.X & sum\_X.Y \\ sum\_Y & sum\_X.Y & sum\_Y.Y \end{bmatrix} . \begin{bmatrix} a\_o \\ a\_1 \\ a\_2 \end{bmatrix} = \begin{bmatrix} sum\_Z \\ sum\_X.Z \\ sum\_Y.Z \end{bmatrix}$$

where sum_X is the sum of the X coordinate values, sum_XY is the sum of the product of X.Y coordinate values, and so forth.

The m-file:

```
% ============================================
% Least Squares Analysis of Geological Borings
% ============================================

% Define matrix A for geological borings information

A = [   10.0   10.0 -30.5 -40.5
       750.0   10.0 -29.0 -39.8
      1250.0   10.0 -28.0 -39.3
      1990.0   10.0 -26.6 -38.5
        10.0  750.0 -34.2 -41.4
       750.0  750.0 -32.8 -40.6
      1250.0  750.0 -31.8 -40.1
      1990.0  750.0 -30.3 -39.4
        10.0 1250.0 -36.7 -42.0
```

```
         750.0 1250.0 -35.2 -41.2
        1250.0 1250.0 -34.2 -40.7
        1990.0 1250.0 -32.8 -40.0
          10.0 1990.0 -40.4 -42.8
         750.0 1990.0 -39.0 -42.1
        1250.0 1990.0 -38.0 -41.6
        1990.0 1990.0 -36.5 -40.9 ];

% Compute terms in matrix BB and right-hand vectors

N=16;
sumx   = sum(A(:,1));
sumy   = sum(A(:,2));
sumx2  = sum(A(:,1).*A(:,1));
sumy2  = sum(A(:,2).*A(:,2));
sumxy  = sum(A(:,1).*A(:,2));

BB=[     N  sumx   sumy
      sumx sumx2  sumxy
      sumy sumxy  sumy2 ]

sumzt  = sum(A(:,3));
sumzb  = sum(A(:,4));
sumzty = sum(A(:,3).*A(:,2));
sumzbx = sum(A(:,4).*A(:,1));
sumzby = sum(A(:,4).*A(:,2));
sumztx = sum(A(:,3).*A(:,1));

DT = [sumzt; sumztx; sumzty];
DB = [sumzb; sumzbx; sumzby];

% Compute constants a,b,c

B=inv(BB);

CT =B*DT
CB =B*DB

% Compute average depth and volume

x=1000;
y=1000;

zt=CT(1,1)+(x*CT(2,1))+(y*CT(3,1));
zb=CB(1,1)+(x*CB(2,1))+(y*CB(3,1));

avedepth = abs(zb-zt)
volume=avedepth*(2000)^2

% Plot top and bottom mineral surfaces

[x y]=meshgrid(0:10:2000,0:10:2000);
zb=CB(1,1)+(x*CB(2,1))+(y*CB(3,1));
mesh(x,y,zb)
hold;
[x y]=meshgrid(0:10:2000,0:10:2000);
zt=CT(1,1)+(x*CT(2,1))+(y*CT(3,1));
mesh(x,y,zt)
grid
```

```
title('Mineral Deposit')
xlabel('Width')
ylabel('Length')
zlabel('Depth')
text(500,1750,-20,'Average Depth = 7.1812  Volume = 2.8725e+07')

% ====================================================================
% the end!
```

generates the output:

```
BB =

          16       16000       16000
       16000    24340800    16000000
       16000    16000000    24340800

CT =

  -30.4538
    0.0020
   -0.0050

CB =

  -40.5031
    0.0010
   -0.0012

avedepth =

    7.1812

volume =

   2.8725e+07
```

and a three dimensional graphic of the mineral deposit boundaries.

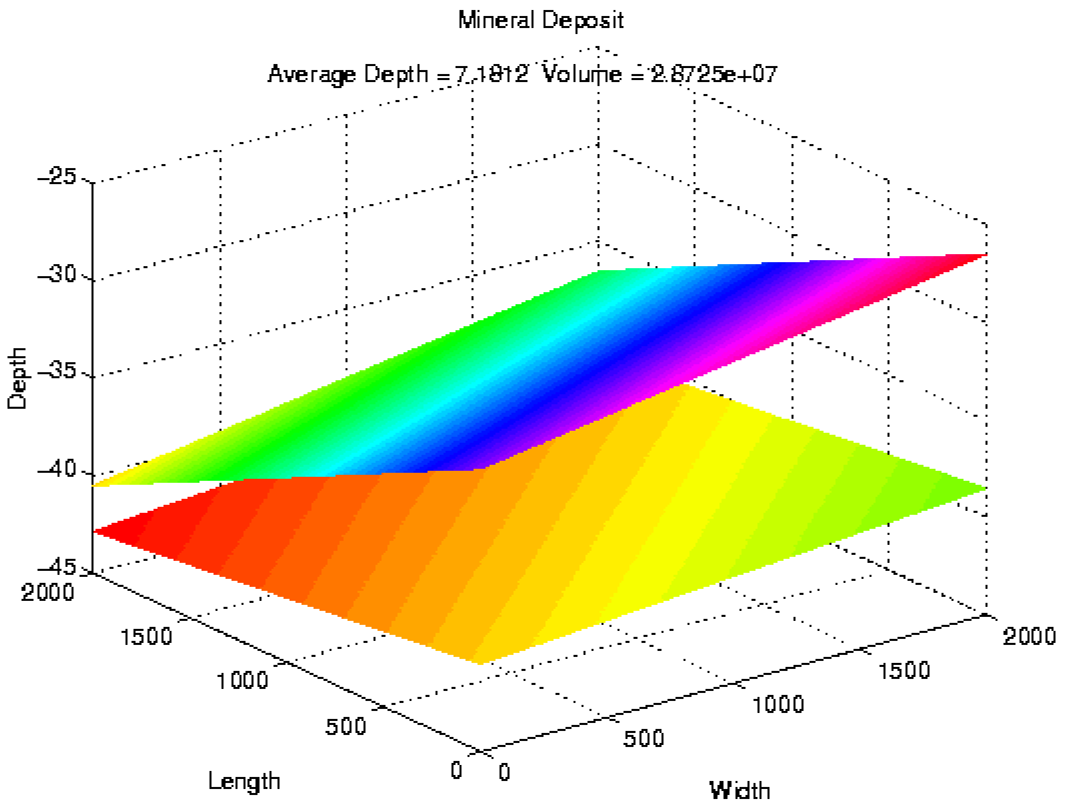


Figure 10: Upper- and Lower- Layers of Mineral Deposits

Comments:

Matrices CB and CT contain the coefficients for the least squares equations representing the lower and upper mineral plane surfaces. You should observe that the coefficients are very close to the equations mentioned in the problem statement.

The most straight forward way of computing the sum of product terms in matrix A, e.g.,

```
A(1,4)*A(1,2) + A(2,4)*A(2,2) + .... + A(16,4)*A(16,2)
```

is with blocks of matlab statements:

```
sumzby = 0.0;
for i=1:16
   sumzby = sumzby + A(i,4)*A(i,2);
end
```

The same numerical result can be obtained via matlab's sum() function combined with the multiply matrix-element operator.

```
sumzby = sum(A(:,4).*A(:,2));
```

In fact, it is possible to completely eliminate the variable "sumzby" from the calculation, and form the (3x3) least squares matrix in one block of statements:

```
BB = [        16          sum(A(:,1))         sum(A(:,2));
      sum(A(:,1))  sum(A(:,1).*A(:,1)) sum(A(:,1).*A(:,2));
      sum(A(:,2))  sum(A(:,1).*A(:,2)) sum(A(:,2).*A(:,2)) ]
```

Similar expressions can be written for matrices "DB" and "DT".

The average depth of mineral deposits is simply the difference in "z" coordinates at (x,y) = (1000,1000). And the volume of mineral deposits is simply the cross section area -- 2000x2000 m^2 -- times the average depth.

Acknowledgement: A draft of this problem solution was created by Amanda Cody.

EXAMPLE 5 : STEADY STATE TEMPERATURE IN A CHIMNEY CROSS SECTION

Problem Statement: Figure 2 shows the front elevation and square-shaped cross section A-A for the tall chimney.

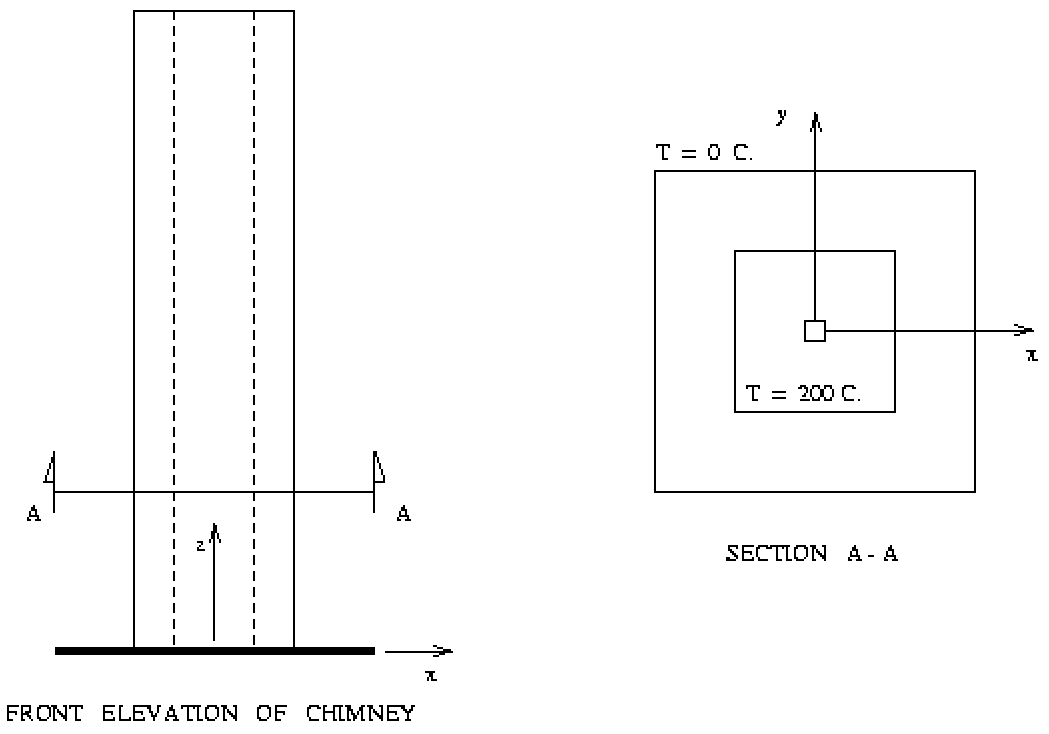


Figure 11: Front Elevation and Cross Section of Tall Chimney

The chimney is constructed from a material that is homogeneous and isotropic (i.e., the material has the same material properties in all directions). At the cross section, the inside and outside temperatures are 200 degrees Centigrade and 0 degrees Centigrade, respectively, and there is neither a net flow of heat to, or from, the chimney (i.e., the chimney is in thermal equilibrium). Finally, we will assume that at the chimney cross section, the distribution of temperature is constant along the z axis.

Because the chimney is constructed from a material that is homogeneous, thermal conductivity will not vary with position, and because the material is isotropic, thermal conductivity will not vary with direction. The steady state distribution of temperature

$$T = T(x,y)$$

throughout the chimney cross section is given by solutions to Laplace's equation

$$\frac{d^2 T(x,y)}{dx^2} + \frac{d^2 T(x,y)}{dy^2} = 0.0$$

with boundary conditions T = 0 degrees Centigrade along the exterior of the chimney, and T = 200 degrees Centigrade along at the chimney interior

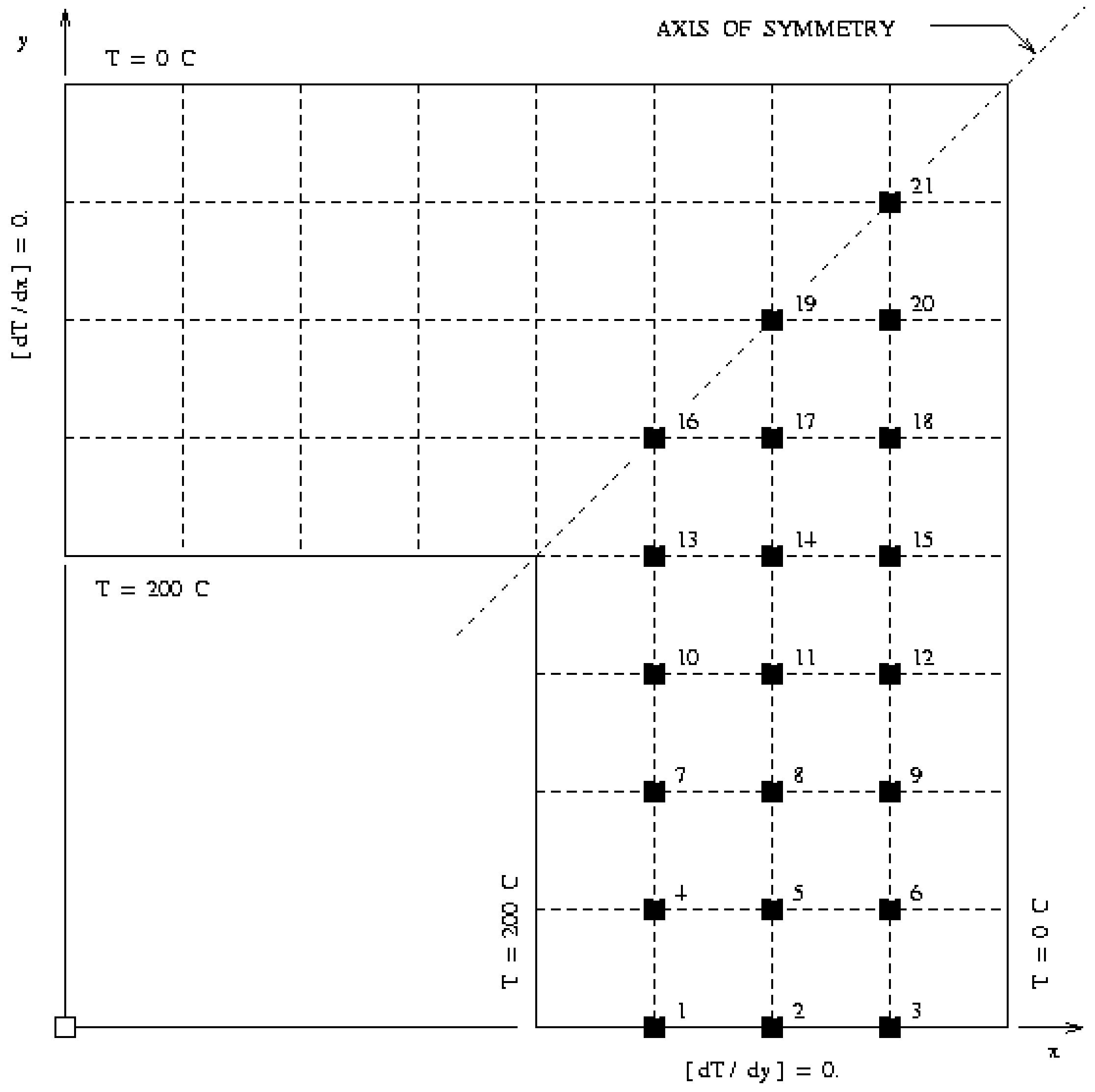


Figure 12 : Finite Difference Mesh

The chimney cross section is symmetric about the x- and y- axes, and the two diagonal axes. In our computational model, we will take advantage of symmetries about the x- and y-axes by modeling only one quarter of the chimney cross section, as shown in Figure 3.

Two new boundary conditions are needed for this model -- along the y axis the temperature gradient

dT/dx = 0,

and along x axis

dT/dy = 0.

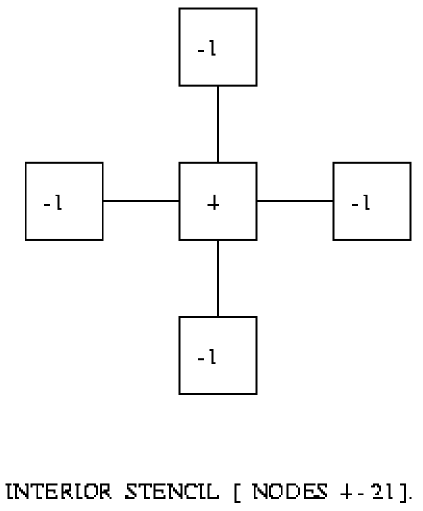
If dx and dy are the mesh distance in the x and y axis directions, a suitable finite difference approximation to Laplace's equation is

T(x+dx,y) - 2T(x,y) + T(x-dx,y)   T(x,y+dy) - 2T(x,y) + T(x,y-dy)

============================== + ============================== = 0.0
dx^2   dy^2

When dx = dy our finite difference equation can be rearranged to give

4 T{(x,y)} - T(x-dx,y) - T(x+dx,y) - T(x,y+dy) - T(x,y-dy) = 0.



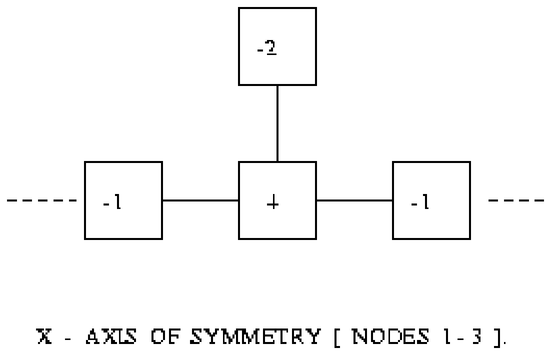


Figure 13: Finite Difference Stencils

The left-most schematic of Figure 4 shows the weighting of discrete temperatures in the finite difference approximation. The nodes along the x axis (i.e., y = 0) satisfy the finite difference equation

4 T(x,0) - T(x-dx,0) - T(x + dx,0) - 2 T(x,dy) = 0

and along the y axis (i.e., x = 0)

4 T(0,y) - 2 T(dx,y) - T(0,y+dy) - T(x,y-dy) = 0.

The finite difference mesh has 65 nodes, 26 of them being on the interior (i.e., T = O degrees Centigrade) and exterior (i.e., T = 200 degrees Centigrade) boundaries. This leaves 39 nodes on the chimney interior for evaluation. Instead of evaluating the temperature stencils at all 39 interior nodes, we compute temperature only at the 21 nodes labeled with small filled-black boxes, and fill in the remaining unknowns by noting the symmetry in temperature along the line

x = y

The four node stencil is used at nodes 1 to 3, and the five node stencil nodes 4 to 21.

Things to do:

Write an m-file that uses the ``method of iteration" to compute the temperature at the internal nodes.

Create a 2-dimensional color contour plot of the temperature distribution inside the chimney wall.

Write an m-file that sets up the finite difference equations in matrix form, and then computes a solution by solving A.T = B, where T is the temperature at the internal nodes of the chimney.

Create a 3-dimensional color plot of the temperature distribution inside the chimney wall.
Note : You can mimic the hole in the chimney by creating a matrix for 1/4 of the chimney cross section and then filling relevant matrix items with NaNs.

Solution : Our solution to this problem has two parts, first by computing the temperature profile by iteration, and then by computing the temperature profile by writing and solving the finite difference equations in matrix form.

Iterative Solution: The m-file for iterative solution is:

```
%  ====================================================================
%  Compute and displace profiles of temperature in chimney cross section
%  ====================================================================

%  Setup working array and boundary conditions along internal/external walls.

 T = zeros(9,9);

 for i = 5:9;
     T(i,5) = 200;
 end;
 for i = 1:5;
     T(5,i) = 200;
 end;
 for i = 6:9;
 for j = 1:4;
     T(i,j) = NaN;
 end;
 end;

% Loop over internal nodes and compute new temperatures

 counter = 0;
 maxchange = 200;
    while (maxchange > 1)
        counter = counter+1;
        maxchange = 0;
        k=5;
        l=4;
        for c = 6:8;
            newtemp    = 0.25*(2*T(l+1,c)+2*T(l,c+1));
            tempchange = newtemp - T(l,c);
            maxchange  = max(maxchange,abs(tempchange));
            T(l,c)=newtemp;
           for r=k:8
               newtemp    = 0.25*(T(r,c-1)+T(r,c+1)+T(r-1,c)+T(r+1,c));
               tempchange = newtemp - T(r,c);
               maxchange  = max(maxchange,abs(tempchange));
               T(r,c)= newtemp;
           end
           newtemp    = 0.25*(T(9,c-1)+T(9,c+1)+2*T(8,c));
           tempchange = newtemp - T(9,c);
           maxchange  = max(maxchange,abs(tempchange));
           T(9,c)=newtemp;
           l=l-1;
           k=k-1;
        end
        counter;
        maxchange;              % to view counter or maxchange remove ;
   end

% Compute reflected temperature
```

```
  for i = 2:4
      for j = 1: 11-i
          T(i,j) = T(10-j,10-i);
      end
  end

% Print Temperature array.

T

% Plot temperature contours.

contour(T)
hold;

% Now overlay perimeter of chimney section on contours.

perim = [ 1 , 1;
          9 , 1;
          9 , 9;
          5 , 9;
          5 , 5;
          1 , 5;
          1 , 1 ];

plot(perim(:,1),perim(:,2),'w');
text(1.1,5.3,'Temp = 200.0');
text(7.0,1.3,'Temp = 0.0');
```

produces the output

```
T =

  Columns 1 through 7

         0         0         0         0         0         0         0
   47.9527   47.4495   45.9241   42.8111   37.2518   28.7560   19.0375
   96.9120   96.1208   93.6699   88.3435   77.7632   59.1105   38.4899
  147.6536  147.0166  144.9543  139.9098  127.2653   92.5014   59.1105
  200.0000  200.0000  200.0000  200.0000  200.0000  127.2653   77.7632
       NaN       NaN       NaN       NaN  200.0000  139.9098   88.3435
       NaN       NaN       NaN       NaN  200.0000  144.9543   93.6699
       NaN       NaN       NaN       NaN  200.0000  147.0166   96.1208
       NaN       NaN       NaN       NaN  200.0000  147.6536   96.9120

  Columns 8 through 9

         0         0
    9.3215         0
   19.0375         0
   28.7560         0
   37.2518         0
   42.8111         0
   45.9241         0
   47.4495         0
   47.9527         0
```

and the two-dimensional graphic

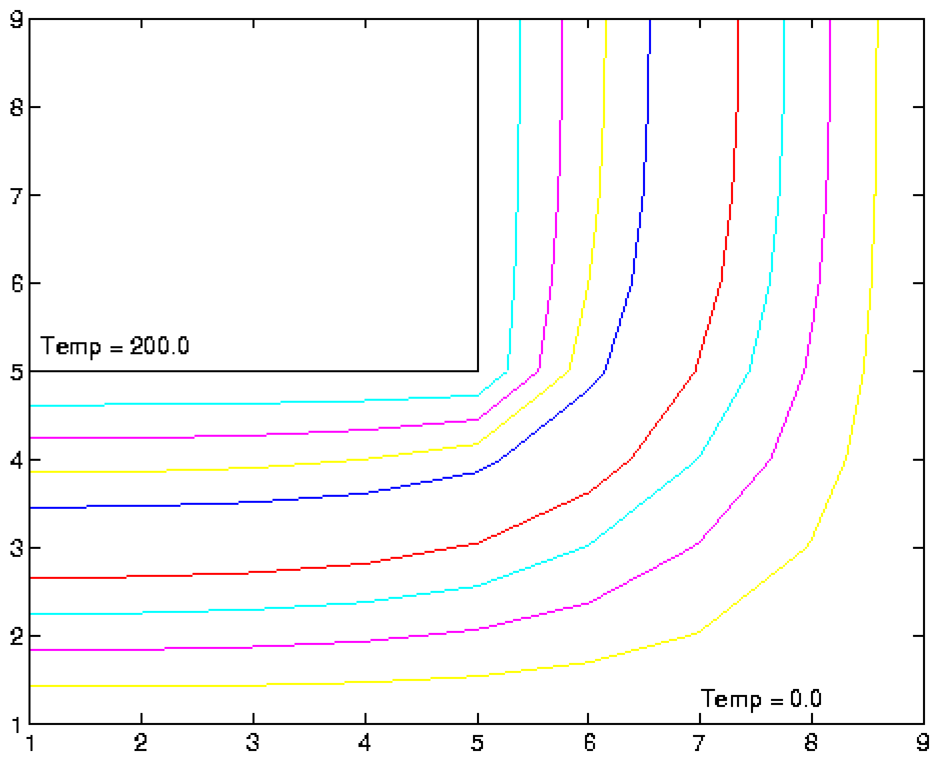


Figure 14: Temperature Profile in Chimney

Some Comments: Points to note are:

The first block of code sets up a (9x9) matrix for modeling 1/4 of the chimney cross section. The temperature along the interior and exterior walls is set to 200 and 0 degrees, respectively. In Matlab, the interior region of the chimney can be represented with NaNs -- that is a missing data item.

At this point in the program execution, the contents of matrix T are:

```
    T =

         0      0      0      0      0      0      0      0      0
         0      0      0      0      0      0      0      0      0
         0      0      0      0      0      0      0      0      0
         0      0      0      0      0      0      0      0      0
       200    200    200    200    200      0      0      0      0
       NaN    NaN    NaN    NaN    200      0      0      0      0
       NaN    NaN    NaN    NaN    200      0      0      0      0
       NaN    NaN    NaN    NaN    200      0      0      0      0
       NaN    NaN    NaN    NaN    200      0      0      0      0
```

The main block of code walks along columns 6 through 8 and evaluates the finite difference stencils for

```
     Column No             Row Nos
     ============================
             6         4 through 9
             7         3 through 9
             8         2 through 9
     ============================
```

For example, after the algorithm has walked along column 6 for the first time the contents of T are

```
T =
  Columns 1 through 7

         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
  200.0000  200.0000  200.0000  200.0000  200.0000   50.0000         0
       NaN       NaN       NaN       NaN  200.0000   62.5000         0
       NaN       NaN       NaN       NaN  200.0000   65.6250         0
       NaN       NaN       NaN       NaN  200.0000   66.4062         0
       NaN       NaN       NaN       NaN  200.0000   83.2031         0

  Columns 8 through 9

         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
```

Bear in mind that the temperature at stencil T(4,6) is still zero because all of the neighboring stencils are initially zero.

The three-line blocks of code

```
   newtemp    = 0.25*(T(9,c-1)+T(9,c+1)+2*T(8,c));
   tempchange = newtemp - T(9,c);
   maxchange  = max(maxchange,abs(tempchange));
```

compute the new temperature estimate at the node, the change in noday temperature from the previous iteration, and the maximum change in temperature occurring over rows 6 through 8 for the current iteration. This algorithm will iterate until the maximum change in temperature is less than 1 degree.

At the conclusion of the main block of code, the temperature profile is:

```
T =

   Columns 1 through 7

         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0   38.4899
         0         0         0         0         0   92.5014   59.1105
  200.0000  200.0000  200.0000  200.0000  200.0000  127.2653   77.7632
       NaN       NaN       NaN       NaN  200.0000  139.9098   88.3435
       NaN       NaN       NaN       NaN  200.0000  144.9543   93.6699
       NaN       NaN       NaN       NaN  200.0000  147.0166   96.1208
       NaN       NaN       NaN       NaN  200.0000  147.6536   96.9120

   Columns 8 through 9

         0         0
    9.3215         0
   19.0375         0
   28.7560         0
   37.2518         0
   42.8111         0
   45.9241         0
   47.4495         0
   47.9527         0
```

The final temperature profile is obtained by reflecting the temperatures along the line y = x.

Matrix Solution: The m-file for matrix solution is:

```
% ===========================================================================
% Compute profile of temperature in chimney via finite difference matrices
% ===========================================================================

% Define matrix of zeros

A = zeros(21,21);

% Fix non-zero values in matrix

for i=1:21
    A(i,i)=4;
end

A(1,2)  =-1; A(1,4)  =-2;
A(2,1)  =-1; A(2,3)  =-1; A(2,5)=-2;
A(3,2)  =-1; A(3,6)  =-2;
A(4,1)  =-1; A(4,5)  =-1; A(4,7)  =-1;
A(5,2)  =-1; A(5,4)  =-1; A(5,6)  =-1; A(5,8)=-1;
A(6,3)  =-1; A(6,5)  =-1; A(6,9)  =-1;
A(7,4)  =-1; A(7,8)  =-1; A(7,10) =-1;
A(8,5)  =-1; A(8,7)  =-1; A(8,9)  =-1; A(8,11)=-1
A(9,6)  =-1; A(9,8)  =-1; A(9,12) =-1;
A(10,7) =-1; A(10,11)=-1; A(10,13)=-1;
A(11,8) =-1; A(11,10)=-1; A(11,12)=-1; A(11,14)=-1;
A(12,9) =-1; A(12,11)=-1; A(12,15)=-1;
A(13,10)=-1; A(13,14)=-1; A(13,16)=-1;
A(14,11)=-1; A(14,13)=-1; A(14,15)=-1; A(14,17)=-1;
A(15,12)=-1; A(15,14)=-1; A(15,18)=-1;
A(16,13)=-2; A(16,17)=-2;
A(17,14)=-1; A(17,16)=-1; A(17,18)=-1; A(17,19)=-1;
A(18,15)=-1; A(18,17)=-1; A(18,20)=-1;
A(19,17)=-2; A(19,20)=-2;
A(20,18)=-1; A(20,19)=-1; A(20,21)=-1;
A(21,20)=-2;

% Define matrix for rhs

N=zeros(21,1);
N(1,1) =200;
N(4,1) =200;
N(7,1) =200;
N(10,1)=200;
N(13,1)=200;

% Solve for temps (compute inverse of A and move to other side).

AA =inv(A);
ANS=round(AA*N)
```

```
% Create matrix to mimic chimney

M=zeros(9,9);

M(6,4) =NaN;
M(7,3) =NaN; M(7,4)=NaN;
M(8,2) =NaN; M(8,3)=NaN; M(8,4)=NaN;
M(9,1) =NaN; M(9,2)=NaN; M(9,3)=NaN; M(9,4)=NaN;

M(5,5) = 200; M(6,5) = 200; M(7,5) = 200;
M(8,5) = 200; M(9,5) = 200; M(9,6) = ANS(1,1);

M(9,7) = ANS(2,1);  M(9,8) = ANS(3,1);
M(8,6) = ANS(4,1);  M(8,7) = ANS(5,1); M(8,8) = ANS(6,1);
M(7,6) = ANS(7,1);  M(7,7) = ANS(8,1); M(7,8) = ANS(9,1);
M(6,6) = ANS(10,1); M(6,7) = ANS(11,1); M(6,8) = ANS(12,1);
M(5,6) = ANS(13,1); M(5,7) = ANS(14,1); M(5,8) = ANS(15,1);
M(4,6) = ANS(16,1); M(4,7) = ANS(17,1); M(4,8) = ANS(18,1);
M(3,7) = ANS(19,1); M(3,8) = ANS(20,1);
M(2,8) = ANS(21,1);

M(2,1) = M(9,8); M(2,2) = M(8,8); M(2,3) = M(7,8);
M(2,4) = M(6,8); M(2,5) = M(5,8); M(2,6) = M(4,8); M(2,7) = M(3,8);
M(3,1) = M(9,7); M(3,2) = M(8,7); M(3,3) = M(7,7);
M(3,4) = M(6,7); M(3,5) = M(5,7); M(3,6) = M(4,7);
M(4,1) = M(9,6); M(4,2) = M(8,6); M(4,3) = M(7,6);
M(4,4) = M(6,6); M(4,5) = M(5,6);
M(5,1) = M(9,5); M(5,2) = M(8,5); M(5,3) = M(7,5); M(5,4) = M(6,5);
M(6,1) = M(9,4); M(6,2) = M(8,4); M(6,3) = M(7,4);
M(7,1) = M(9,3); M(7,2) = M(8,3);
M(8,1) = M(9,2);
M(1,1) = M(9,9); M(1,2) = M(8,9); M(1,3) = M(7,9); M(1,4) = M(6,9);
M(1,5) = M(5,9); M(1,6) = M(6,9); M(1,7) = M(5,9); M(1,8) = M(2,9);

% Create 3-D color plot of solution

xx =1:9;
yy =xx;
[x y]=meshdom(xx,yy);

z=M;
mesh(z)
grid
title('Temperature in Chimney')

xlabel('width')
ylabel('depth')
zlabel('temp')

% ====================================================
% The end!
```

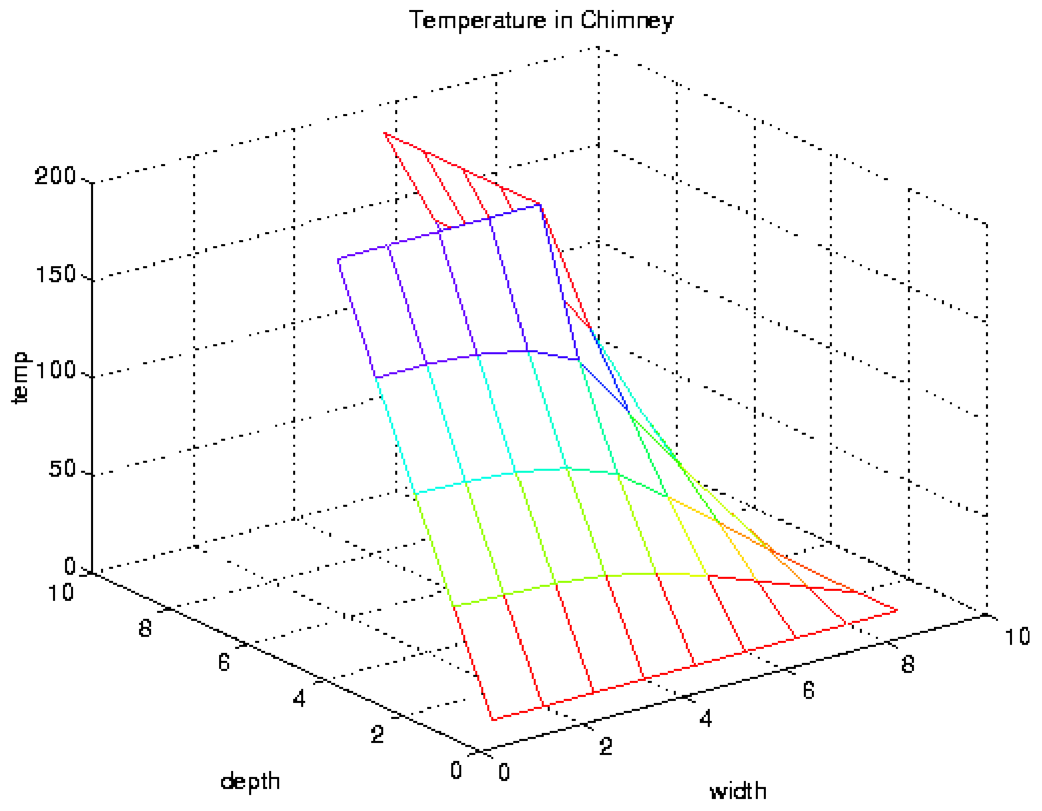


Figure 15: Temperature Profile in Chimney

Comments:

Acknowledgement: Drafts of the problem solutions were created by Amanda Cody and James Hozdic.

EXAMPLE 6 : MATRIX ARITHMETIC

Problem Statement:

Solution: The first is matrix addition.

```
function c=add(a,b)

% c=add(a,b). This is the function which adds
% the matrices a and b. It duplicates the MATLAB
% function a+b.

[m,n]=size(a);
[k,l]=size(b);
if m~=k | n~=l,
   r='ERROR using add: matrices are not the same size';
   return,
end
c=zeros(m,n);
for i=1:m,
   for j=1:n,
      c(i,j)=a(i,j)+b(i,j);
   end
end
```

The next program is matrix multiplication.

```
function c=mult(a,b)

% c=mult(a,b). This is the matrix product of
% the matrices a and b. It duplicates the MATLAB
% function c=a*b.

[m,n]=size(a);
[k,l]=size(b);
if n~=k,
   c='ERROR using mult: matrices are not compatible
      for multiplication',
   return,
end,
c=zeros(m,l);
for i=1:m,
   for j=1:l,
      for p=1:n,
         c(i,j)=c(i,j)+a(i,p)*b(p,j);
      end
   end
end
```

For both of these programs you should notice the branch construction which follows the size statements. This is included as an error message. In the case of add, an error is made if we attempt to add matrices of different sizes, and in the case of mult it is an error to multiply if the matrix on the left does not have the same number of columns as the number of rows of the the matrix on the right. Had these messages not been included and the error was made, MATLAB would have delivered another error message saying that the index exceeds the matrix dimensions. You will notice in the error message the use of single quotes. The words surrounded by the quotes will be treated as text and sent to the screen as the value of the variable c. Following the message is the command return, which is the directive to send the control back to the function which called add or return to the prompt. I usually only recommend using the return command in the context of an error message. Most MATLAB implementations have an error message function, either errmsg or error, which you might prefer to use.

EXAMPLE 7 : USE BISECTION METHOD TO COMPUTE ZEROS OF A FUNCTION

Some more advanced features are illustrated by the following function. As noted earlier, some of the input arguments of a function-such as tol in the example, may be made optional through use of nargin (``number of input arguments''). The variable nargout can be similarly used. Note that the fact that a relation is a number (1 when true; 0 when false) is used and that, when while or if evaluates a relation, ``nonzero'' means ``true'' and 0 means ``false''. Finally, the MATLAB function feval permits one to have as an input variable a string naming another function.

```
function  [b, steps] = bisect(fun, x, tol)
%BISECT Zero of a function of one variable via the bisection method.
%       bisect(fun,x) returns a zero of the function.  fun is a string
%       containing the name of a real-valued function of a single
%       real variable; ordinarily functions are defined in M-files.
%       x  is a starting guess.  The value returned is near a point
%       where  fun  changes sign.  For example,
%       bisect('sin',3) is pi.  Note the quotes around sin.
%
%       An optional third input argument sets a tolerence for the
%       relative accuracy of the result.  The default is eps.
%       An optional second output argument gives a matrix containing a
%       trace of the steps; the rows are of form  [c f(c)].

% Initialization
```

```
if nargin < 3, tol = eps; end
trace = (nargout == 2);
if x  ~= 0, dx = x/20; else, dx = 1/20; end
a = x - dx;  fa = feval(fun,a);
b = x + dx;  fb = feval(fun,b);

% Find change of sign.
while (fa > 0) == (fb > 0)
     dx = 2.0*dx;
     a = x - dx;  fa = feval(fun,a);
     if (fa > 0) ~= (fb > 0), break, end
     b = x + dx;  fb = feval(fun,b);
end
if trace, steps = [a fa; b fb]; end

% Main loop
while  abs(b - a) > 2.0*tol*max(abs(b),1.0)
     c = a + 0.5*(b - a);  fc = feval(fun,c);
     if trace, steps = [steps; [c fc]]; end
     if (fb > 0) == (fc > 0)
          b = c;  fb = fc;
       else
         a = c;  fa = fc;
     end
end
```

Some of MATLAB's functions are built-in while others are distributed as M-files. The actual listing of any M-file-MATLAB's or your own-can be viewed with the MATLAB command **type functionname**. Try entering **type eig**, **type vander**, and **type rank**.

MISCELLANEOUS FEATURES

You may have discovered by now that MATLAB is case sensitive, that is

"a" is not the same as "A."

If this proves to be an annoyance, the command

```
>> casesen
```

will toggle the case sensitivity off and on.

The MATLAB display only shows 5 digits in the default mode. The fact is that MATLAB always keeps and computes in a double precision 16 decimal places and rounds the display to 4 digits. The command

```
>> format long
```

will switch to display all 16 digits and

```
>> format short
```

will return to the shorter display. It is also possible to toggle back and forth in the scientific notation display with the commands

```
>> format short e
```

and

```
>> format long e
```

It is not always necessary for MATLAB to display the results of a command to the screen. If you do not want the matrix A displayed, put a semicolon after it, A;. When MATLAB is ready to proceed, the prompt >> will appear. Try this on a matrix right now.

Sometimes you will have spent much time creating matrices in the course of your MATLAB session and you would like to use these same matrices in your next session. You can save these values in a file by typing

```
>> save filename
```

This creates a file

```
filename.mat
```

which contains the values of the variables from your session. If you do not want to save all variables there are two options.

One is to clear the variables off with the command

```
>> clear a b c
```

which will remove the variables a,b,c. The other option is to use the command

```
>> save x y z
```

which will save the variables x,y,z in the file filename.mat. The variables can be reloaded in a future session by typing

```
>> load filename
```

When you are ready to print out the results of a session, you can store the results in a file and print the file from the operating system using the "print" command appropriate for your operating system. The file is created using the command

```
>> diary filename
```

Once a file name has been established you can toggle the diary with the commands

```
>> diary on
```

and

```
>> diary off
```

This will copy anything which goes to the screen (other than graphics) to the specified file. Since this is an ordinary ASCII file, you can edit it later. Discussion of print out for graphics is deferred to the project "Graphics" where MATLAB's graphics commands are presented.

Some of you may be fortunate enough to be using a Macintosh or a Sun computer with a window system that allows you to quickly move in and out of MATLAB for editing, printing, or other processes at the system level. For those of you who are not so fortunate, MATLAB has a feature which allows you to do some of these tasks directly from MATLAB. Let us suppose that you would like to edit a file named myfile.m and that your editor executes on the command ed. The MATLAB command

```
!ed myfile.m
```

will bring up your editor and you can now work in it as you usually would. Obviously the exclamation point is the critical feature here. When you are done editing, exit your editor as you usually would, and you will find that you are back in your MATLAB session. You can use the ! with many operating system commands.

I am going to describe the basic programming constructions. While there are other constructions available, if you master these you will be able to write clear programs.

PROGRAMMING SUGGESTIONS

Here are a few pointers for programming in MATLAB:

1. I urge you to use the indented style that you have seen in the above programs. It makes the programs easier to read, the program syntax is easier to check, and it forces you to think in terms of building your programs in blocks.

2. Put lots of comments in your program to tell the reader in plain English what is going on. Some day that reader will be you, and you will wonder what you did.

3. Put error messages in your programs like the ones above. As you go through this manual, your programs will build on each other. Error messages will help you debug future programs.

4. Always structure your output as if it will be the input of another function. For example, if your program has "yes-no" type output, do not have it return the words "yes" and "no," rather return 1 or 0, so that it can be used as a condition for a branch or while loop construction in the future.

5. In MATLAB, try to avoid loops in your programs. MATLAB is optimized to run the built-in functions. For a comparison, see how much faster A*B is over mult(A,B). You will be amazed at how much economy can be achieved with MATLAB functions.

6. If you are having trouble writing a program, get a small part of it running and try to build on that. With reference to 5), write the program first with loops, if necessary, then go back and improve it.

REFERENCES

1. Adrian Biran and Moshe Breiner, "MATLAB for Engineers," Addison-Wesley Publishing Company, 1995.
2. Kermit Sigmon, "MATLAB Primer (Second Edition)," Department of Mathematics, University of Florida, 1992.
3. The Student Edition of MATLAB, Version 4, Users Guide, The Math Works Inc., Prentice Hall, Englewood Cliff, NJ, 1995.