10.34 Numerical Methods Applied to Chemical Engineering

MATLAB Tutorial

Kenneth Beers
Department of Chemical Engineering
Massachusetts Institute of Technology
August 1, 2001

The Nature of Scientific Computing

This course focuses on the use of computers to solve problems in chemical engineering. We will learn how to solve the partial differential equations that describe momentum, energy, and mass transfer, integrate the ordinary differential equations that model a chemical reactor, and simulate the dynamics and predict the minimum-energy structures of molecules. These problems are expressed in terms of mathematical operations such as partial differentiation and integration that computers do not understand. All that they know how to do is store numbers at locations in their memory and perform simple operations on them like addition, subtraction, multiplication, division, and exponentiation. Somehow, we need to translate our higher-level mathematical description of these problems into a sequence of these basic operations.

It is logical to develop simulation algorithms that decompose each problem into sets of linear equations of the following form.

$$a_{11}* x_1+ a_{12}*x_2 + \ldots + a_{1n}*x_n = b_1$$

$$a_{21}*x_1 + a_{22}*x_2 + \ldots + a_{2n}*x_n = b_2$$

$$.$$

$$.$$

$$.$$

$$a_{n1}*x_1 + a_{n2}*x_2 + \ldots + a_{nn}*x_n = b_n$$

A computer understands how to do the operations found in this system (multiplication and addition), and we can represent this set of equations very generally by the matrix equation Ax = b, where A={a_{ij}} is the matrix of coefficients on the left hand side, x is the solution vector, and b is the vector of the coefficients on the right hand side. This general representation allows us to pass along, in a consistent language, our system-specific linear equation sets to pre written algorithms that have been optimized to solve them very efficiently. This saves us the effort of coding a linear solver every time we write a new program. This method of relegating repetitive tasks to re-usable, pre written subroutines makes the idea of using a computer to solve complex technical problems feasible. It also allows us to take advantage of the decades of applied mathematics research that have gone into developing efficient numerical algorithms. Scientific programs typically involve problem-specific sections that perform the parameter input and results output, phrase the problem into a series of linear algebraic systems, and then the program spends most of its execution time solving these linear systems. This course focuses primarily on understanding the theory and concepts fundamental to scientific computing, but we also need to know how to translate these concepts into working programs and to combine our problem-specific code with pre written routines that efficiently perform the desired numerical operations.
So, how do we instruct the computer to solve our specific problem? At a basic level, all a computer does is follow instructions that tell it to retrieve numbers from specified memory

locations, perform some simple algebraic operations on them, and store them in some (possibly new) places in memory. Rather than force computer users to deal with details like memory addresses or the passing of data from memory to the CPU, computer scientists develop for each type of computer a program called a *compiler* that translates ãhuman-levelä code into the set of detailed machine-level instructions (contained in an *executable* file) that the computer will perform to accomplish the task. Using a compiler, it is easy to write code that tells a computer to do the following :

1. Find a space in memory to store a real number x
2. Find a space in memory to store a real number y
3. Find a space in memory to store a real number z
4. Set the value of x to 2
5. Set the value of y to 4
6. Set the value stored at the location z to equal 2*x + 3*y, where the symbol * denotes multiplication

In FORTRAN, the first modern scientific programming language that, in modified form - commonly FORTRAN 77, is still in wide use today, you can accomplish these tasks by writing the code :

```
REAL x, y, z
x = 2
y = 4
z = 2*x + 3*y
```

By itself, however, this code performs the desired task, but does not provide any means for the user to view the results. A full FORTRAN program to perform the task and write the result to the screen is :

```
IMPLICIT NONE
REAL x, y, z
x = 2
y = 4
z = 2*x + 3*y
PRINT *, 'z = ',z
END
```

When this code is compiled with a FORTRAN 77 compiler, the output to the screen from running the *executable* is : z = 16.0000. Compiled programming languages allow only the simple output of text, numbers, and binary data, so any graphing of results must be performed by a separate program. In practice, this requirement of writing the code, storing the output in a file with the appropriate format, and reading this file into a separate graphing or analysis program leads one to use for small projects "canned" software such as EXCEL that are ill-suited for technical computing; after all, EXCEL is intended for business spreadsheets!

Other compiled programming languages exist, most being more powerful than FORTRAN 77, a legacy of the past that is retained mostly due to the existence of highly efficient numerical routines written in the language. While FORTRAN 77 lacks the functionality of more modern languages, in terms of execution speed it usually has the advantage. In the 80's and 90's, C and C++ became highly popular within the broader computer science community because they allow one to organize and structure data more conveniently and to write highly-modular code for large programs. C and C++ have never gained the same level of popularity within the scientific computing community, mainly because their implementation has been focused more towards robustness and generality with less regard for execution speed. Many scientific programs have comparatively simple structures so that execution speed is the primary concern. This situation is changing somewhat today; however, the introduction of FORTRAN 90 and its update FORTRAN 95 have given the FORTRAN language a new lease on life.

FORTRAN 90/95 includes many of the data structuring capabilities of C/C++, but was written with a technical audience in mind. It is the language of choice for parallel scientific computing, in which tasks are parceled during execution to one or more CPU's. With the growing popularity of dual processor workstations and BEOWOLF-type clusters, FORTRAN 90/95 and variants such as High Performance Fortran remain my personal *compiled* language of choice for heavy-duty scientific computing.

Then why does this course use MATLAB instead of FORTRAN 90? FORTRAN 90 is my choice among *compiled* languages; however, for ease of use, MATLAB, an *interpreted* language, is better for small to medium jobs. In *compiled* languages, the "human-level" commands are converted directly to machine instructions that are stored in an executable file. Run-time execution of the commands does not take place until all of the compilation process has been completed (run-time debugging not excepted). In a *compiled* language, one needs to learn the commands for the input/output of data (from the keyboard, to the screen, to/from files) and for naming variables and allocating space for them in memory (like the command real in FORTRAN). *Compiled* languages are developed with the principle that the language should have a minimum amount of commands and syntax, so that any task that may be accomplished by a sequence of more basic instructions is not incorporated into the language definition but is rather left to a subroutine. Subroutine libraries have been written by the applied mathematics community to perform common numerical operations (e.g. BLAS and LAPACK), but to access them you need to link your code to them through operating system-specific commands. While not conceptually difficult, the overhead is not insignificant for small projects.

In an *interpreted* language, the developers of the language have already written and compiled a master program, in our case the program MATLAB, that will interpret our commands to the computer ãon-the-flyä. When we run MATLAB, we are offered a window in which we can type commands to perform mathematical calculations. This code is then interpreted line-by-line (by machine-level instructions) into other machine-level instructions that actually carry out the computations that we have requested. Because MATLAB has to interpret each command one-by-one, we will require more machine-level instructions to perform a certain job that we would with a *compiled* language. For demanding numerical simulations, where we need to use the resources of a computer as efficiently as possible, *compiled* languages are therefore superior.

Using an *interpreted* language has the benefit; however, that we do not need to compile the code before-hand. We can therefore type in our commands one-by-one and watch them be performed (this is very helpful for finding errors). We do not need to link our code to subroutine libraries, since MATLAB, being pre compiled, has all the machine-level instructions it needs readily at-hand. FORTRAN 77/90/95, C, and C++ cannot make graphs, so if we want to plot the results from our program, we need to write data to an output file that we use as input to yet another graphics program. By contrast, the MATLAB programmers have already provided graphics routines and compiled them along with the MATLAB code interpreter, so we do not need this additional data transfer step. An *interpreted* language can provide efficient and complex memory management utilities that, by operating behind a curtain, shield the programmer from having to learn their complicated syntax of usage. New variables can therefore be created with dynamic memory allocation without requiring the user to understand pointers (variables that point to memory locations), as is required in most *compiled* languages. Finally, since MATLAB was not developed with the principle of minimum command syntax, it contains a rich collection of integrated numerical operations. Some of these routines are designed to solve linear problems very efficiently. Others operate at a higher level, for example taking as input a function f(x) and returning the point x_0 that has $f(x_0)=0$, or integrating the ordinary differential equation dx/dt = f(x) starting from a value of x at t=0.

For these reasons, one can code more efficiently in *interpreted* languages than in *compiled* languages (McConnell, Steve, *Code Complete*, Microsoft Press, 1993 and Jones, Capers, *Programming Productivity*, McGraw-Hill, 1986), at the cost of slower execution due to the extra interpreting step for each command. But, we have noted before that execution speed is an important consideration in scientific computing, so is this acceptable? MATLAB has several features to alleviate this situation. Whenever MATLAB first runs a subroutine, it saves the results of the interpreting process so that successive calls do not have to repeat this work. Additionally, one can reduce the interpretation overhead by minimizing the number of command lines, a practice which incidentally leads to good programming style for FORTRAN 90/95. As an example, let us take the operation of multiplying a M by N matrix A with an N by P matrix B to form a M by P matrix C. In FORTRAN 77 we would first have to declare and allocate memory to store the A, B, and C matrices (as well as the counter integers i_row, i_col, and i_mid), and then, perhaps in a subroutine, execute the code :

```
DO i_row = 1, M
  DO i_col = 1, N
    C(i_row,i_col) = 0.0
    DO i_mid = 1, P
      C(i_row,i_col) = C(i_row,i_col) + A(i_row,i_mid)*B(i_mid,i_col)
    ENDDO
  ENDDO
ENDDO
```

If we simply translated each line, one-by-one, from FORTRAN 77 to MATLAB, we would have the code segment :

```
for i_row = 1:M
  for i_col = 1:N
    C(i_row,i_col) = 0;
    for i_mid = 1:P
      C(i_row,i_col) = C(i_row,i_col) + A(i_row,i_mid)*B(i_mid,i_col);
    end
  end
end
```

This code performs the task in exactly the same manner as FORTRAN 77, but now each line must be interpreted one-by-one, adding a considerable overhead. It would seem that we would be better off with FORTRAN 77; however, in MATLAB the language is extended to allow matrix operations so that we could accomplish the same task with the single command : C = A*B. We would not even have to pre allocate memory to store C, this would be automatically handled by MATLAB. The MATLAB approach is greatly to be preferred, and not only because it accomplishes the same task with less typing (and chance for error!). The FORTRAN 77 code, relying on basic scalar addition and multiplication operations, is not very easy to parallelize. It instructs the computer to perform the matrix multiplication with an exact order of events that the computer is constrained to follow. The single command C = A*B requests the same task, but leaves the computer free to decide how to accomplish it in the most efficient manner, for example, by splitting the problem across multiple processors. One of the main advantages of FORTRAN 90/95 over FORTRAN 77 is that it also allows these whole array operations (the corresponding FORTRAN 90/95 code is C = MATMUL(A,B)), so that writing fast MATLAB code rewards the same programming style as does FORTRAN 90/95 for producing code that is easy to parallelize.

MATLAB also comes with an optional compiler that converts MATLAB code to C or C++ and that can compile this code to produce a stand-alone *executable*. We therefore can enjoy the ease of programming in an *interpreted* language, and then once the program development is complete, we can take advantage of the efficient execution and portability offered by *compiled* languages. Alternatively, given the tools of the compiler, we can combine MATLAB code and numerical routines with FORTRAN or C/C++ code. Given these advantages, MATLAB seems a strong choice of language for an introductory course in scientific computing.

MATLAB Tutorial Table of Contents

This tutorial is presented with a separate webpage for each chapter. The commands listed in the tutorial are explained with comment lines starting with the percentage sign %. These commands may either be typed or pasted one-by-one into an interactive MATLAB window. Further information about a specific command can be obtained by typing help followed by the name of the command. Typing helpwin brings up a general help utility, and helpdesk provides links to extensive on-line documentation. For further details, consult the texts found in the Recommended Reading section of the 10.34 homepage.

MATLAB Tutorial

Chapter 1. Basic MATLAB commands

1.1 Basic scalar operations

First, let's talk about how we add comments (such as this line) to a program. Comments are lines of text that we want to add to explain what we are doing, so that if we or others read this code later, it will be easier to figure out what the code is doing. In a MATLAB file, if a percentage sign, , appears in a row of text, all of the text following the sign is a comment that MATLAB does not try to interpret as a command. First, let us write a message to the screen to say that we are beginning to run section 1.1.

The command disp('string') displays the text string to the screen.
disp('Beginning section 1.1 ...')

Next, we set a variable equal to one.
x=1

This command both allocates a space in memory for the variable x, if x has not already been declared, and then stores the value of 1 in the memory location associated with this variable. It also writes to the screen "x = 1". Usually, we do not want to clutter the screen with output such as this, so we can make the command "invisible" by ending it with a semi-colon. As an example, let us use the following commands to "invisibly" change the value of x to 2 and then to write out the results to the screen. **x=2;** this changes the value of x but does not write to the screen disp('We have changed the value of x.');

Then, we display the value of x by typing "x" without a semi-colon.
x

Now, let's see how to declare other variables.
y = 2*x; This initializes the value of y to twice that of x
x = x + 1; This increases the value of x by 1.
z = 2*x; This declares another variable z.

z does not equal y because the value of x changed between the
times when we declared each variable.
difference = z - y

Next, we want to see the list of variables that are stored in memory. To do this, we use the command "who".
who;

We can get more information by using "whos".
whos;

These commands can be used also to get information about only certain variables.
whos z difference;

Let us say we want to get rid of the variable "difference".
We do this using the command "clear".
clear difference;
who;

Next, we want to get rid of the variables x and y.
Again, we use the command "clear".

```
clear x y;
who;
```

It is generally good programming style to write only one command per line; however, MATLAB does let you put multiple commands on a line.

```
x = 5; y = 13; w = 2*x + y; who;
```

More commonly one wishes to continue a single command across multiple lines due to the length of the syntax. This can be accomplished by using three dots.

```
z = 2*x + ...
y
```

Finally, when using clear we can get rid of all of the variables at once with the command "clear all".

clear all;
who; It does not print out anything because there are no variables.

1.2. Basic vector operations

The simplest, but NOT RECOMMENDED, way to declare a variable is by entering the components one-by-one.

```
x(1) = 1;
x(2) = 4;
x(3) = 6;
```

x display contents of x

It is generally better to declare a vector all at once, because then MATLAB knows how much memory it needs to allocate from the start. For large vectors, this is much more efficient.
y = [1 4 6] does same job as code above

Note that this declares a row vector. To get a column vector, we can either use the transpose (adjoint for complex x) operator **xT = x';** takes the transpose of the real row vector x or, we can make it a column vector right from the beginning

```
yT = [1; 4; 6];
```

To see the difference in the dimensions of a row vs. a column vector, use the command "size" that returns the dimensions of a vector or matrix.

```
size(xT)
size(y)
size(yT)
```

The command length works on both row and column vectors.

```
length(x), length(xT)
```

Adding or subtracting two vectors is similar to scalars.

```
z = x + y
w = xT - yT
```

Multiplying a vector by a scalar is equally straight-forward.

```
v = 2*x
c = 4;
v2 = c*x
```

We can also use the . operator to tell MATLAB to perform a given operation on an element-by-element basis. Let us say we want to set each value of y such that y(i) = 2*x(i) + z(i)^2 + 1. We can do this using the code

```
y = 2.*x + z.^2 + 1
```

The dot and cross products of two vectors are calculated by
dot(x,y)
z=cross(x,y)

We can define a vector also using the notation [a : d : b]. This produces a vector a, a + d, a + 2*d, a + 3*d, ... until we get to an integer n where a + n*d > b. Look at the two examples.
v = [0 : 0.1: 0.5];
v2 = [0 : 0.1: 0.49];

If we want a vector with N evenly spaced points from a to b, we use the command "linspace(a,b,N)".
v2 = linspace(0,1,5)

Sometimes, we will use a vector later in the program, but want to initialize it at the beginning to zero and by so doing allocate a block of memory to store it. This is done by
v = linspace(0,0,100)'; allocate memory for column vectors of zero

Finally, we can use integer counting variables to access one or more elements of a matrix.
v2 = [0 : 0.01 : 100];
c=v2(49)
w = v2(65:70)

clear all

1.3. Basic matrix operations

We can declare a matrix and give it a value directly.
A = [1 2 3; 4 5 6; 7 8 9]
We can use commas to separate the elements on a line as well.
B = [1,2,3; 4,5,6; 7,8,9]

We can build a matrix from row vectors
row1 = [1 2 3]; row2 = [4 5 6]; row3 = [7 8 9];
C = [row1; row2; row3]

or from column vectors.
column1 = [1; 4; 7];
column2 = [2; 5; 8];
column3 = [3; 6; 9];
D = [column1 column2 column3]

Several matrices can be joined to create a larger one.
M = [A B; C D]

We can extract row or column vectors from a matrix.
row1 = C(1,:)
column2 = D(:,2)

Or, we make a vector or another matrix by extracting a subset of the elements.
v = M(1:4,1)
w = M(2,2:4)
C = M(1:4,2:5)

The transpose of a real matrix is obtained using the ' operator

```
D = A'
C, C'
```

For a complex matrix, ' returns the adjoint (transpose and conjugate. The conjugation operation is removed by using the "transpose only" command .'

```
E = D;
E(1,2) = E(1,2) + 3*i;
E(2,1) = E(2,1) - 2*i;
E', E.'
```

The "who" command lists the matrices in addition to scalar and vector variables.

```
who
```

If in addition we want to see the dimensions of each variable, we use the "whos" command. This tells use the size of each variable and the amount of memory storage that each requires.

```
whos
```

The command "size" tells us the size of a matrix.

```
M = [1 2 3 4; 5 6 7 8; 9 10 11 12];
size(M)
num_rows = size(M,1)
num_columns = size(M,2)
```

Adding, subtracting, and multiplying matrices is straight-forward.

```
D = A + B
D = A - B
D = A*B
```

We can declare matrices in a number of ways.

We can create a matrix with m rows and n columns, all containing zeros by

```
m=3; n=4;
C = zeros(m,n)
```

If we want to make an N by N square matrix, we only need to use one index.

```
C = zeros(n)
```

We create an Identity matrix, where all elements are zero except for those on the principle diagonal, which are one.

```
D = eye(5)
```

Finally, we can use the . operator to perform element-by-element operations just as we did for vectors. The following command creates a matrix C, such that C(i,j) = 2*A(i,j) + (B(i,j))^2.

```
C = 2.*A + B.^2
```

Matrices are cleared from memory along with all other variables.

```
clear A B
whos
clear all
who
```

1.4. Using character strings

In MATLAB, when we print out the results, we often want to explain the output with text. For this, character strings are useful. In MATLAB, a character string is written with single quotation marks on each end.

```
course_name = 'Numerical Methods Applied to Chemical Engineering'
```

To put an apostrophe inside a string, we repeat it twice to avoid confusing it with the ' operator ending the string.

```
phrase2 = 'Course''s name is : ';
disp(phrase2), disp(course_name)
```

We can also combine strings in a similar manner to working with vectors and matrices of numbers.

```
word1 = 'Numerical'; word2 = 'Methods'; word3='Course';
phrase3 = [word1, word2, word3]
```

We see that this does not include spaces, so we use instead

```
phrase4 = [word1, ' ', word2, ' ', word3]
```

We can convert an integer to a string using the command "int2str".

```
icount = 1234;
phrase5 = ['Value of icount = ', int2str(icount)]
```

Likewise, we can convert a floating point number of a string of k digits using "num2str(number,k)".

```
Temp = 29.34372820092983674;
phrase6 = ['Temperature = ',num2str(Temp,5)]
phrase7 = ['Temperature = ',num2str(Temp,10)]
```

```
clear all
```

1.5. Basic mathematical operations

EXPONENTIATION COMMANDS

We have already seen how to add, subtract, and multiply numbers. We have also used on occasion the ^ operator where x^y raises x to the power y.

```
2^3, 2^3.3, 2.3^3.3, 2.3^(1/3.3), 2.3^(-1/3.3)
```

The square root operation is given its own name.

```
sqrt(27), sqrt(37.4)
```

Operators for use in analyzing the signs of numbers include

abs(2.3), abs(-2.3) returns absolute value of a number
sign(2.3), sign(-2.3), sign(0) returns sign of a number

The commands for taking exponents and logs are

a=exp(2.3) computes e^x
log(a) computer the natural log
log10(a) computes the base 10 log

TRIGONOMERTRY COMMANDS

The numerical value of pi can be invoked directly

```
pi, 2*pi
```

NOTE THAT MATLAB CALCULATES ANGLES IN RADIANS

The standard trigonometric functions are
sin(0), sin(pi/2), sin(pi), sin(3*pi/2)
cos(0), cos(pi/2), cos(pi), cos(3*pi/2)
tan(pi/4), cot(pi/4), sec(pi/4), csc(pi/4)

Their inverses are
asin(1),acos(1),atan(1),acot(1),asec(1),acsc(1)

The hyperbolic functions are
sinh(pi/4), cosh(pi/4), tanh(pi/4), coth(pi/4)
sech(pi/4), csch(pi/4)
with inverses
asinh(0.5), acosh(0.5), atanh(0.5), acoth(0.5)
asech(0.5), acsch(0.5)

These operators can be used with vectors in the following manner.
x=linspace(0,pi,6) create vector of x values between 0 and pi
y=sin(x) each y(i) = sin(x(i))

ROUNDING OPERATIONS

round(x) : returns integer closest to real number x
round(1.1), round(1.8)

fix(x) : returns integer closest to x in direction towards 0
fix(-3.1), fix(-2.9), fix(2.9), fix(3.1)

floor(x) : returns closest integer less than or equal to x
floor(-3.1), floor(-2.9), floor(2.9), floor(3.1)

ceil(x) : returns closest integer greater than or equal to x
ceil(-3.1), ceil(-2.9), ceil(2.9), ceil(3.1)

rem(x,y) : returns the remainder of the integer division x/y
rem(3,2), rem(898,37), rem(27,3)

mod(x,y) : calculates the modulus, the remainder from real division
mod(28.36,2.3)

COMPLEX NUMBERS

A complex number is declared using i (or j) for the square root of -1.
z = 3.1-4.3*i
conj(z) returns conjugate, conj(a+ib) = a - ib
real(z) returns real part of z, real(a+ib) = a
imag(z) returns imaginary part of z, imag(a+ib) = b
abs(z) returns absolute value (modulus), a^2+b^2
angle(z) returns phase angle theta with z = r*exp(i*theta)
abs(z)*exp(i*angle(z)) returns z

For complex matrices, the operator ' calculates the adjoint matrix, i.e. it transposes the matrix and takes the conjugate of each element
A = [1+i, 2+2*i; 3+3*i, 4+4*i]
A' takes conjugate transpose (adjoint operation)
A.' takes transpose without conjugating elements

COORDINATE TRANSFORMATIONS

2-D polar coordinates (theta,r) are related to Cartesian coordinates by

```
x=1; y=1;
[theta,r] = cart2pol(x,y)
[x,y] = pol2cart(theta,r)
```

3-D spherical coordinates (alpha,theta,r) are obtained from Cartesian coordinates by

```
x=1; y=1; z=1;
[alpha,theta,r] = cart2sph(x,y,z)
[x,y,z] = sph2cart(alpha,theta,r)
```

```
clear all
```

MATLAB Tutorial

Chapter 2. Programming Structures

2.1. for loops

Programs for numerical simulation often involve repeating a set of commands many times. In MATLAB, we instruct the computer to repeat a block of code by using a for loop. A simple example of a for loop is **for i=1:10** repeats code for i=1,2,...,10
i print out the value of the loop counter **end** This ends the section of code that is repeated.

The counter can be incremented by values other than +1.
for i=1:2:10
disp(i);
end

This example shows that the counter variables takes on the values 1, 3, 5, 7, 9. After 9, the code next tries i=11, but as 11 is greater than 10 (is not less than or equal to 10) it does not perform the code for this iteration, and instead exits the for loop.
for i=10:-1:1
disp(i);
end

As the value of the counter integer is changed from one iteration to the next, a common use of for blocks is to perform a given set of operations on different elements of a vector or a matrix. This use of for loops is demonstrated in the example below.

Complex structures can be made by nesting for loops within one another. The nested for loop structure below multiplies an (m x p) matrix with a (p x n) matrix.
A = [1 2 3 4; 11 12 13 14; 21 22 23 24]; A is 3 x 4 matrix
B = [1 2 3; 11 12 13; 21 22 23; 31 32 33]; B is 4 x 3 matrix
im = size(A,1); m is number of rows of A
ip = size(A,2); p is number of columns of A
in = size(B,2); n is number of columns of B
C = zeros(im,in); allocate memory for m x n matrix containing 0's

now we multiply the matrices
for i=1:im iterate over each row of C
for j=1:in iterate over each element in row
for k=1:ip sum over elements to calculate C(i,j)
C(i,j) = C(i,j) + A(i,k)*B(k,j);
end
end
end
C print out results of code
A*B MATLAB's routine does the same thing

clear all

2.2. if, case structures and relational operators

In writing programs, we often need to make decisions based on the values of variables in memory. This requires logical operators, for example to discern when two numbers are equal. Common relational operators in MATLAB are : eq(a,b) returns 1 if a is equal to b, otherwise it returns 0

```
eq(1,2), eq(1,1)
eq(8.7,8.7), eq(8.7,8.71)
```

When used with vectors or matrices, eq(a,b) returns an array of the same size as a and b with elements of zero where a is not equal b and ones where a equals b. This usage is demonstrated for the examples below.

```
u = [1 2 3]; w = [4 5 6]; v = [1 2 3]; z = [1 4 3];
eq(u,w), eq(u,v), eq(u,z)
A = [1 2 3; 4 5 6; 7 8 9]; B = [1 4 3; 5 5 6; 7 9 9];
eq(A,B)
```

this operation can also be called using ==

```
(1 == 2), (1 == 1), (8.7 == 8.7), (8.7 == 8.71)
```

ne(a,b) returns 1 if a is not equal to b, otherwise it returns 0

```
ne(1,2), ne(1,1)
ne(8.7,8.7), ne(8.7,8.71)
ne(u,w), ne(u,v), ne(u,z)
ne(A,B)
```

another way of calling this operation is to use ~=

```
(1 ~= 2), (1 ~= 1), (8.7 ~= 8.7), (8.7 ~= 8.71)
```

lt(a,b) returns 1 if a is less than b, otherwise it returns 0

```
lt(1,2), lt(2,1), lt(1,1)
lt(8.7,8.71), lt(8.71,8.7), lt(8.7,8.7)
```

another way of performing this operation is to use <

```
(1 < 2), (1 < 1), (2 < 1)
```

le(a,b) returns 1 if a is less than or equal to b, otherwise 0

```
le(1,2), le(2,1), le(1,1)
le(8.7,8.71), le(8.71,8.7), le(8.7,8.7)
```

this operation is also performed using <=

```
(1 <= 1), (1 <= 2), (2 <= 1)
```

gt(a,b) returns 1 if a is greater than b, otherwise 0

```
gt(1,2), gt(2,1), gt(1,1)
gt(8.7,8.71), gt(8.71,8.7), gt(8.7,8.7)
```

this operation is also performed using >

```
(1 > 2), (1 > 1), (2 > 1)
```

ge(a,b) returns 1 if a is greater than or equal to b, otherwise 0

```
ge(1,2), ge(2,1), ge(1,1)
ge(8.7,8.71), ge(8.71,8.7), ge(8.7,8.7)
```

this operation is also performed using >=

```
(1 >= 1), (1 >= 2), (2 >= 1)
```

These operations can be combined to perform more complex logical tests.

(logic1)&(logic2) returns 0 unless both logic1 and logic2 are not equal to zero

```
((1==1)&(8.7==8.7))
((1==2)&(8.7==8.7))
((1>2)&(8.71>8.7))
((1<2)&(8.7<8.71))
((1>2)&(8.7>8.71))
i1 = 1; i2 = 0; i3=-1;
(i1 & i1), (i1 & i2), (i2 & i1), (i2 & i2), (i1 & i3)
((1==1)&(8.7==8.7)&(1<2))
((1==1)&(8.7==8.7)&(1>2))
```

This operation can be extended to multiple operations more easily by using the command all(vector1), that returns 1 if all of the elements of vector1 are nonzero, otherwise it returns 0

```
all([i1 i2 i3]), all([i1 i1 i3])
```

or(logic1,logic2) returns 1 if one of either logic1 or logic2 is not equal to zero or if they are both unequal to zero.

```
or(i1,i2), or(i1,i3), or(i2,i2)
```

This operation can be extended to more than two logical variables using the command any(vector1), that returns 1 if any of the elements of vector1 are nonzero, otherwise it returns 0.

```
any([i1 i2 i3]), any([i2 i2 i2]), any([i1 i2 i2 i2]),
```

Used less often in scientific computing is the exclusive or construction xor(logic1,logic2) that returns 1 only if one of logic1 or logic2 is nonzero, but not both.

```
xor(i1,i1), xor(i2,i2), xor(i1,i2)
```

We use these relational operations to decide whether to perform a block of code using an if structure that has the general form.

```
logictest1 = 0; logictest2 = 1; logictest3 = 0;
if(logictest1)
disp('Executing block 1');
elseif(logictest2)
disp('Executing block 2');
elseif(logictest3)
disp('Executing block 3');
else
disp('Execute end block');
end
```

The last block of code is executed if none of the ones before it has been performed.

```
logictest1 = 0; logictest2 = 0; logictest3 = 0;
if(logictest1)
disp('Executing block 1');
elseif(logictest2)
disp('Executing block 2');
elseif(logictest3)
disp('Executing block 3');
else
disp('Execute end block');
end
```

An if loop will not execute more than one block of code. If more than one logictest variable is not equal to zero, then the first one it encounters is the one it performs.

```
logictest1 = 0; logictest2 = 1; logictest3 = 1;
if(logictest1)
disp('Executing block 1');
elseif(logictest2)
disp('Executing block 2');
elseif(logictest3)
disp('Executing block 3');
else
disp('Execute end block');
end
```

If structures are often used in conjunction with for loops. For example, the following routine adds the components of a vector to the principal diagonal of a matrix that is the sum of two matrices A and B.

```
A = [1 2 3; 4 5 6; 7 8 9];
B = [11 12 13; 14 15 16; 17 18 19];
u = [10 10 10];
C=zeros(3);
for i=1:3
for j=1:3
if(i==j)
C(i,j) = A(i,j) + B(i,j) + u(i);
else
C(i,j) = A(i,j) + B(i,j);
end
end
end
```

As an alternative to if blocks, case structures can be used to chose among various alternatives.

```
for i=1:4
switch i;
case {1}
disp('i is one');
case {2}
disp('i is two');
case {3}
disp('i is three');
otherwise
disp('i is not one, two, or three');
end
end

clear all
```

2.3. while loops and control statements

A WHILE loops performs a block of code as long as the logical test expression returns a non-zero value.

```
error = 283.4;
tol = 1;
factor = 0.9;
while (error > tol)
error = factor*error;
disp(error)
end
```

If factor >= 1, then the value of error will increase and the while loop will not terminate. A better way, in general, to accomplish the job above is to use a for loop to place an upper limit to the number of iterations that will be performed. A "break" command stops the iteration of the most deeply nested for loop and is called when the condition (error < tol) is reached.

```
error = 283.4;
tol = 1;
factor = 0.9;
iter_max = 10000;
iflag = 0; signifies goal not reached
for iter=1:iter_max
if(error <= tol)
iflag = 1; signifies goal reached
break;
end
```

error = factor*error;
disp(error)
end
if(iflag==0) write message saying that goal not reached.
disp('Goal not reached');
disp(['error = ' num2str(error)]);
disp(['tol = ',num2str(tol)]);
end

clear all

2.4. screen input/output

In MATLAB, the basic command to write output to the screen is "disp".
disp('The disp command writes a character string to the screen.');

When writing integer or real numbers to the screen, the "int2str" and "num2str" commands should be used (for more details see chapter 1 of the tutorial.
i = 2934;
x = 83.3847;
disp(['i = ' int2str(i)]);
disp(['x = ' num2str(i)]);

The standard command for allowing the user to input data from the keyboard is "input".
i = input('Input integer i : ');
x = input('Input real x : ');
v = input('Input vector v : '); try typing [1 2 3]
i, x, v

clear all

Chapter 3. Basic graphing routines

3.1. 2-D plots

The basic command for making a 2-D plot is "plot". The following code makes a plot of the function sin(x).
x = linspace(0,2*pi,200);
f1 = sin(x);
plot(x,f1)

we now add a title and labels for the x and y axes
title('Plot of f_1 = sin(x)');
xlabel('x');
ylabel('f_1');

Let us change the axes so that they only plot x from 0 to 2*pi.
axis([0 2*pi -1.1 1.1]); [xmin xmax ymin ymax]

Next, we make a new figure with cos(x)
f2 = cos(x);
figure; makes a new figure window
plot(x,f2);
title('Plot of f_2 = cos(x)');
xlabel('x');
ylabel('f_2');
axis([0 2*pi -1.1 1.1]);

Now, we make a single graph with both plots
figure; creates a new graph
plot(x,f1);
hold on; tells MATLAB not to overwrite current plot

What happens if you forget to type hold on? "hold off" removes the hold.
plot(x,f2,'r'); plots with red curve
title('Plots of f_1 = sin(x), f_2 = cos(x)');
xlabel('x');
ylabel('f_1, f_2');
axis([0 2*pi -1.1 1.1]);

Now we add a legend.
legend('f_1', 'f_2');

If we want to move the legend, we can go to the "Tools" menu of the figure window and turn on "enable plot editing" and then drag the legend to where we want it.

Finally, we use the command "gtext" to add a line of text that we then position on the graph using our cursor.
gtext('f_1=f_2 at two places');

The command "help plot" tells how to make a graph using various types of points instead of lines and how to select different colors.

clear all

3.2. 3-D plots

First, we generate a grid containing the x and y values of
each point.
x = 0:0.2:2*pi; create vector of points on x-axis
y = 0:0.2:2*pi; create vector of points on y-axis

Now if n=length(x) and m=length(y), the grid will contain N=n*m grid points. XX and YY are n by m matrices containing the x and y values for each grid point respectively.
[XX,YY] = meshgrid(x,y);

The convention in numbering the points is apparent from the following lines.
x2 = 1:5; y2 = 11:15;
[XX2,YY2] = meshgrid(x2,y2);
XX2, YY2

This shows that XX2(i,j) contains the jth component of the x vector and YY2(i,j) contains the ith component of the y vector.

Now, we generate a function to save as a separate z-axis value for each (x,y) 2-D grid point.
Z1 = sin(XX).*sin(YY); calculate value of function to be plotted

create a colored mesh plot
figure; mesh(XX,YY,Z1);
xlabel('x'); ylabel('y'); zlabel('z'); title('sin(x)*sin(y)');

create a colored surface plot
figure; surf(XX,YY,Z1);
xlabel('x'); ylabel('y'); zlabel('z'); title('sin(x)*sin(y)');

create a contour plot
figure; contour(XX,YY,Z1);
xlabel('x'); ylabel('y'); zlabel('z'); title('sin(x)*sin(y)');

create a filled contour plot with bar to show function values
figure; contourf(XX,YY,Z1); colorbar;
xlabel('x'); ylabel('y'); zlabel('z'); title('sin(x)*sin(y)');

create a 3-D contour plot
figure; contour3(XX,YY,Z1);
xlabel('x'); ylabel('y'); zlabel('z'); title('sin(x)*sin(y)');

clear all

3.3. Making complex figures

Using the subplot command, one can combine multiple plots into a single figure. We want to make a master figure that contains nrow # of rows of figures and ncol # of figures per row. subplot(nrow,ncolumn,i) makes a new figure window within the master plot, where i is a number denoting the position within the master plot according to the following order :
1 2 3 ... ncol
ncol+1 ncol+2 ncol+3 ... 2*ncol

First, generate the data to be plotted.
x = 0:0.2:2*pi;

y = 0:0.2:2*pi;
f1 = sin(x);
f2 = cos(y);
[XX,YY] = meshgrid(x,y);
Z1=sin(XX).*cos(YY);

The following code creates a figure with four subplots.
figure; create a new figure

subplot(2,2,1); create 1st subplot window
plot(x,f1); title('sin(x)');
xlabel('x'); ylabel('sin(x)'); axis([0 2*pi -1.1 1.1]);

subplot(2,2,2); create 2nd subplot window
plot(y,f2); title('cos(y)');
xlabel('y'); ylabel('cos(y)'); axis([0 2*pi -1.1 1.1]);

subplot(2,2,3); create 3rd subplot window
surf(XX,YY,Z1); title('sin(x)*cos(y)');
xlabel('x'); ylabel('y'); zlabel('z');

subplot(2,2,4); create 4th subplot window
contourf(XX,YY,Z1); colorbar; title('sin(x)*cos(y)');
zlabel('x'); ylabel('y');

clear all

Chapter 4. Advanced matrix operations

4.1. Sparse matrices

SPARSE MATRICES
To show the efficiency gained by using sparse matrices, we will solve a PDE using finite differences twice. First, we will use the matrix commands that use the full matrix that we have learned so far. Second, we will use new commands that take advantage of the fact that most of the elements are zero to greatly reduce both the memory requirements and the number of floating point operations required to solve the PDE.

clear all; remove all existing variables from memory

num_pts = 100; number of grid points in simulation

CALCULATION WITH FULL MATRIX FORMAT

The following matrix is obtained from using central finite differences to discretize the Laplacian operator in 1-D.
x = 1:num_pts; grid of x-values

Set the matrix from discretizing the PDE with a 1-D grid containing num_pts points with a spacing of 1 between points.
Afull=zeros(100,100);
Afull(1,1) = 1;
Afull(num_pts,num_pts) = 1;
for i=2:(num_pts-1) sum over interior points
Afull(i,i) = 2;
Afull(i,i-1) = -1;
Afull(i,i+1) = -1;
end

Dirichlet boundary conditions at x=1 and x=num_pts are set.
BC1 = -10; value of f at x(1);
BC2 = 10; value of f at x(num_pts);

For the interior points, we have a source term.
b_RHS = linspace(0,0,num_pts)'; create column vector of zeros
b_RHS(1) = BC1;
b_RHS(num_pts) = BC2;
b_RHS(2:(num_pts-1)) = 0.05; for interior, b_RHS is source term

We now use the standard MATLAB solver to obtain the solution of the PDE at the grid points.
f = Afull\b_RHS;
figure; plot(x,f);
title('PDE solution from FD-CDS method (full matrix)');
xlabel('x'); ylabel('f(x)');

Let us now take a closer look at Afull. The command spy(A) makes a plot of the matrix A by writing a point wherever an element of A has a non-zero value.
figure;
spy(Afull); title('Structure of Afull');

The number nz at the bottom is the number of non-zero elements. We see that only a small fraction of the matrix elements are non-zero. Since we numbered the grid points in a regular manner with the neighbors of each grid point stored in adjacent locations, the non-zero elements in this matrix are on the principal diagonal and the two diagonals immediately above and below. Even if we numbered the grid points irregularly, we would still have this small number of non-zero points. It is often the case, as it is here, that the matrices we encounter in the numerical simulation of PDE's are sparse; that is, only a small fraction of their points are non-zero. For this matrix, the total number of elements is
num_elements = num_pts*num_pts;
nzA = nnz(Afull); returns # of non-zero elements in Afull
fraction_filled = nzA/num_elements

This means that Afull is mostly empty space and we are wasting a lot of memory to store values we know are zero.

Remove all variables from memory except Afull.
clear x f b_RHS BC1 BC2 i num_elements nzA fraction_filled;

SPARSE MATRIX

We can convert a matrix to sparse format using the command "sparse".
Asparse = sparse(Afull)

MATLAB stores a sparse matrix as an NZ by 3 array where NZ is the number of non-zero elements. The first column is the row number and the second the column number of the non-zero element. The third column is the actual value of the non-zero element. The total memory usage is far smaller than with the full matrix format.
whos Afull Asparse;
clear Asparse; get rid of sparse matrix

NOW WE WILL SOLVE USING SPARSE MATRIX FORMAT

Next, we set the grid point values
x = 1:num_pts; grid of x-values

Now we declare the matrix A to have sparse matrix structure from the start. First, we calculate the number of non-zero elements (or an upper bound to this number). We see that for each row corresponding to an interior point, we have 3 values, whereas for the first and last row we only have one value. Therefore, the number of non-zero elements is
nzA = 3*(num_pts-2) + 2;

We now use "spalloc(m,n,nz)" that allocates memory for a m by n dimensioned sparse matrix with no more than nz non-zero elements.
A = spalloc(num_pts,num_pts,nzA);

We now set the values of the A matrix.
```
A(1,1) = 1;
A(num_pts,num_pts) = 1;
for i=2:(num_pts-1)
A(i,i) = 2;
A(i,i-1) = -1;
A(i,i+1) = -1;
end
```

Dirichlet boundary conditions at x=1 and x=num_pts are set.

```
BC1 = -10; value of f at x(1);
BC2 = 10; value of f at x(num_pts);
```

For the interior points, we have a source term.

```
b_RHS = linspace(0,0,num_pts)'; create column vector of zeros
b_RHS(1) = BC1;
b_RHS(num_pts) = BC2;
b_RHS(2:(num_pts-1)) = 0.05; for interior, b_RHS is source term
```

Now, when we call the MATLAB standard solver, it automatically identifies that A is a sparse matrix, and uses solver algorithms that take advantage of this fact.

```
f = A\b_RHS;

figure; plot(x,f);
title('PDE solution from FD-CDS method (sparse matrix)');
xlabel('x'); ylabel('f(x)');
whos A Afull;
```

From the lines for A and Afull, we can see that the sparse matrix format requires far less memory that the full matrix format. Also, if N is the number of grid points, we see that the size of the full matrix is N^2; whereas, the size in memory of the sparse matrix is only approximately 3*N. Therefore, as N increases, the sparse matrix format becomes far more efficient than the full matrix format. For complex simulations with thousands of grid points, one cannot hope to solve these problems without taking advantage of sparsity. To see the increase in execution speed that can be obtained by using sparse matrices, examine the following two algorithms for multiplying two matrices.

FULL MATRIX ALGORITHM FOR MATRIX MULTIPLICATION

```
Bfull = 2*Afull;
Cfull = 0*Afull; declare memory for C=A*B
num_flops = 0;
for i=1:num_pts
for j=1:num_pts
for k=1:num_pts
Cfull(i,j) = Cfull(i,j) + Afull(i,k)*Bfull(k,j);
num_flops = num_flops + 1;
end
end
end
disp(['# FLOPS with full matrix format = ', int2str(num_flops)]);
```

SPARSE MATRIX ALGORITHM FOR MATRIX MULTIPLICATION

```
B = 2*A;
nzB = nnz(B); # of non-zero elements of B
nzC_max = round(1.2*(nzA+nzB)); guess how much memory we'll need for C
C = spalloc(num_pts,num_pts,nzC_max);
[iA,jA] = find(A); find (i,j) elements that are non-zero in A
[iB,jB] = find(B); find (i,j) elements that are non-zero in B
num_flops = 0;
for ielA = 1:nzA iterate over A non-zero elements
for ielB = 1:nzB iterate over B non-zero elements
if(iB(ielB)==jA(ielA)) the pair contributes to C
i = iA(ielA);
k = jA(ielA);
```

```
j = jB(ielB);
C(i,j) = C(i,j) + A(i,k)*B(k,j);
num_flops = num_flops + 1;
end
end
end
disp(['# FLOPS for sparse matrix format = ', int2str(num_flops)]);
D = Cfull - C; check to see both algorithms give same result
disp(['# of elements where Cfull ~= C : ' int2str(nnz(D))]);
```

Finally, we note that taking the inverse of a sparse matrix
usually destroys much of the sparsity.

```
figure;
subplot(1,2,1); spy(A); title('Structure of A');
subplot(1,2,2); spy(inv(A)); title('Structure of inv(A)');
```

Therefore, if we have the values of A and of C = A*B and want to calculate the matrix B, we do NOT use inv(A)*C. Rather, we use the "left matrix division" operator A\C. This returns a matrix equivalent to inv(A)*C, but uses the MATLAB solver that takes advantage of the sparsity.

```
B2 = A\C;
figure; spy(B2); title('Structure of B2');
```

We see that the error from the elimination method has introduced very small non-zero values into elements off of the central three diagonals. We can remove these by retaining only the elements that are greater than a tolerance value.

```
tol = 1e-10;
Nel = nnz(B2);
[iB2,jB2] = find(B2); return positions of non-zero elements
for iel=1:Nel
if(abs(B2(iB2(iel),jB2(iel))) < tol) set to zero
B2(iB2(iel),jB2(iel)) = 0;
end
end
B2 = sparse(B2); reduce memory storage
figure; spy(B2); title('Structure of "cleaned" B2');
```

Since we do not want to go through intermediate steps where we have to store a matrix with many non-zero elements, we usually do not calculate matrices in this manner. Rather we limit ourselves to solving linear systems of the form A*x = b, where x and b are vectors and A is a sparse matrix whose value we input directly. We therefore avoid the memory problems associated with generating many non-zero elements from round-off errors.

```
clear all
```

4.2. Common matrix operations/eigenvalues

The determinant of a aquare matrix is calculated using "det".

```
A = rand(4); creates a random 4x4 matrix
det(A) calculate determinant of A
```

Other common functions of matrices are

```
rank(A) rank of A
trace(A) trace of A
norm(A) matrix norm of A
cond(A) condition number of A
```

A_inv=inv(A) calculates inverse of A

```
A*A_inv
```

The eigenvalues of a matrix are computed with the command "eig"

```
eig(A)
```

If the eigenvectors are also required, the syntax is

```
[V,D] = eig(A)
```

Here V is a matrix containing the eigenvectors as column vectors, and D is a diagonal matrix containing the eigenvalues.

```
for i=1:4
eig_val = D(i,i);
eig_vect = V(:,i);
A*eig_vect - eig_val*eig_vect
end
```

The command "eigs(A,k)" calculates the k leading eigenvalues of A; that is, the k eigenvalues with the largest moduli.

eigs(A,1) estimate leading eigenvalue of A

Similarly, the eigenvectors of the leading eigenvalues can also be calculated with eigs.

```
[V2,D2] = eigs(A,1);
eig_vect = V2; eig_val = D2;
A*eig_vect - eig_val*eig_vect
```

With sparse matrices, only the command "eigs" can be used.

```
clear all
```

4.3. LU decomposition

The linear system Ax=b can be solved with multiple b vectors using LU decomposition. Here, we perform the decomposition P*A = L*U, where P is a permutation matrix (hence inv(P)=P'), L is a lower triangular matrix, and U is an upper triangular matrix. P is an identity matrix when no pivoting is done during the factorization (which is essentially Gaussian elimination). Once the LU factorization is complete, a problem Ax=b is solved using the following linear algebra steps.

A*x = b
P*A*x = P*b
L*U*x = P*b

This gives the following two linear problems invloving triangular matrices that may be solved by substitution.
L*y = P*b
U*x = y
The MATLAB command for performing an LU factorization is "lu" We use a random, non-singular matrix to demonstrate the algorithm. Non-singularity is ensured by adding a factor of an identity matrix.

```
A = rand(10) + 5*eye(10);
```

Perform LU factorization.

```
[L,U,P] = lu(A);
```

max(P*P'-eye(10)) demonstrates that P is orthogonal matrix
max(P*A - L*U) shows largest result of round-off error

Compare the structures of the matrices involved

```
figure;
subplot(2,2,1); spy(A); title('Structure of A');
subplot(2,2,2); spy(P); title('Structure of P');
subplot(2,2,3); spy(L); title('Structure of L');
subplot(2,2,4); spy(U); title('Structure of U');
```

LU factorization can be called in exactly the same way for sparse matrices; however, in general the factored matrices L and U are not as sparse as is A, so by using LU factorization, some efficiency is lost. This becomes more of a problem the the greater the bandwidth of the matrix, i.e. the farther away from the principal diagonal that non-zero values are found.

Sometimes we only want an approximate factorization B=L*U where B is close enough to A such that C = inv(B)*A is not too much different from an identity matrix, i.e. the ratio between the largest and smallest eigenvalues of C is less than that for A. In this case, B is called a preconditioner, and is used in methods for optimization and solving certain classes of linear systems. When we perform an incomplete LU factorization, we only calculate the elements of L and U that correspond to non-zero elements in A, or with different options, we neglect elements whose absolute values are less than a specified value.

The following code demonstrates the use of incomplete LU factorization.

make B=A, except set certain elements equal to zero.

```
B=A;
```

set some elements far-away from diagonal equal to zero.

```
for i=1:10
B(i+5:10,i) = 0;
B(1:i-5,i) = 0;
end

B=sparse(B);
[Linc,Uinc,Pinc] = luinc(B,'0');

figure;
subplot(2,2,1); spy(B); title('Structure of B');
subplot(2,2,2); spy(Pinc); title('Structure of Pinc');
subplot(2,2,3); spy(Linc); title('Structure of Linc');
subplot(2,2,4); spy(Uinc); title('Structure of Uinc');

D1 = P*A - L*U;
D2 = Pinc*B - Linc*Uinc;
tol = 1e-10; set tolerance for saying element is zero
for i=1:10
for j=1:10
if(D1(i,j)<tol)
D1(i,j) = 0;
end
if(D2(i,j)<tol)
D2(i,j) = 0;
end
end
end
```

figure;
subplot(1,2,1); spy(D1); title('(P*A - L*U)');
subplot(1,2,2); spy(D2); title('(Pinc*B - Linc*Uinc)');

But, look at the eigenvalues of the B and of the approximate factorization.
Bapprox = Pinc'*Linc*Uinc;
eigs(B) eigenvalues of B matrix
C = Bapprox\B; inv(Bapprox)*B (don't use "inv" for sparse matrices)
eigs(C)

clear all

4.4. QR decomposition

The factorization A*P = Q*R, where P is a permutation matrix, Q is a orthogonal matrix, and R is upper triangular is performed by invoking the command "qr".
A = rand(6);
[Q,R,P] = qr(A);

Q*Q' shows Q is orthogonal
A*P - Q*R

figure;
subplot(2,2,1); spy(A); title('Structure of A');
subplot(2,2,2); spy(P); title('Structure of P');
subplot(2,2,3); spy(Q); title('Structure of Q');
subplot(2,2,4); spy(R); title('Structure of R');

If the decomposition A=QR is desired (i.e. with P=1), the
command is :
[Q,R] = qr(A);

figure;
subplot(2,2,1); spy(A); title('Structure of A');
subplot(2,2,2); spy(Q); title('Structure of Q');
subplot(2,2,3); spy(R); title('Structure of R');

A - Q*R

clear all

4.5. Cholesky decomposition

If A is a Hermetian matrix (i.e. A=A') then we know that all eigenvalues are real. If in addition, all the eigenvalues are greater than zero, then x'*A*x > 0 for all vectors x and we say that A is positive-definite. In this case, it is possible to perform a Cholesky decomposition, i.e. A = R'*R, where R is upper triangular. This is equivalent to writing A = L*L', where L is lower triangular.

First, we use the following positive-definite matrix.
Ndim=10;
Afull=zeros(Ndim,Ndim);
for i=1:Ndim sum over interior points
Afull(i,i) = 2;
if(i>1)

```
Afull(i,i-1) = -1;
end
if(i<Ndim)
Afull(i,i+1) = -1;
end
end

Rfull = chol(Afull);
D = Afull - Rfull'*Rfull; eig(D)

figure;
subplot(1,2,1); spy(Afull); title('Structure of Afull');
subplot(1,2,2); spy(Rfull); title('Structure of Rfull');
```

For sparse matrices, we can perform an incomplete Cholesky decomposition that gives an approximate factorization with no loss of sparsity that can be used as a preconditioner. In this particular case, with a highly structured matrix, the incomplete factorization is the same as the complete one.

```
Asparse = sparse(Afull);
Rsparse = cholinc(Asparse,'0');
D2 = Asparse - Rsparse'*Rsparse; eig(D2)

figure;
subplot(1,2,1); spy(Asparse); title('Structure of Asparse');
subplot(1,2,2); spy(Rsparse); title('Structure of Rsparse');

clear all
```

4.6. Singular value decomposition

Eigenvalues and eigenvectors are only defined for square matrices. The generalization of the concept of eigenvalues to non-square matrices is often useful. A singular value decomposition (SVD) of the (m x n) matrix A is defined as A = U*D*V', where D is a (m x n) diagonal matrix containing the singular values, U is a (m x m) matrix containing the right eigenvectors and V' is the adjoint (transpose and conjugate) of the (n x n) matrix of left eigenvectors.

In MATLAB, a singular value decomposition is peformed using "svd"

A = [1 2 3 4; 11 12 13 14; 21 22 23 24];
[U,D,V] = svd(A);
D, U, V
U*D*V' show that decomposition works

clear all

MATLAB Tutorial

Chapter 5. File input/output

5.1. Saving/reading binary files and making calls to the operating system

When using MATLAB, either when running a m-file or performing calculations interactively, there is a master memory structure that MATLAB uses to keep track of the values of all of the variables. This memory space can be written in a binary format to a file for storing the results of your calculations for later use. This is often useful when you have to interrupt a MATLAB session. The following commands demonstrate how to use this storage option to make binary .mat files.

First, let us define some variables that we want to save.

```
num_pts =10;
Afull=zeros(num_pts,num_pts);
Afull(1,1) = 1;
Afull(num_pts,num_pts) = 1;
for i=2:(num_pts-1) sum over interior points
Afull(i,i) = 2;
Afull(i,i-1) = -1;
Afull(i,i+1) = -1;
end
b = linspace(0,1,num_pts)';
x = Afull\b;
```

whos; display contents of memory

The "save" command saves the data in the memory space to the named binary file.

```
save mem_store1.mat;
```

clear all;
whos; no variables are stored in memory

ls *.mat display all .mat files in directory

The "load" command loads the data stored in the named binary file into memory.

load mem_store1.mat;
whos; we see that the data has been loaded again

If we want to get rid of this file, we can use the "delete" command.

```
delete mem_store1.mat;
ls *.mat
```

In the commands above, I have used path names to specify the directory. We can view our current default directory using the command "pwd".

pwd displays the current directory

We can then change to another directory using the "cd" command.

cd .. move up one directory
pwd
ls list files in directory
cd MATLAB_tutorial; directory name may differ for you
pwd; ls

We can also use the "save" command to save only selected variables to a binary file.

```
save mem_store2.mat Afull;

clear all
whos

load mem_store2.mat
whos

delete mem_store2.mat

clear all
```

5.2. Input/output of data to/from an ASCII file

First, let use define some variables that we want to save.

```
num_pts =10;

Afull=zeros(num_pts,num_pts);
Afull(1,1) = 1;
Afull(num_pts,num_pts) = 1;
for i=2:(num_pts-1) sum over interior points
Afull(i,i) = 2;
Afull(i,i-1) = -1;
Afull(i,i+1) = -1;
end

b = linspace(0,1,num_pts)';
x = Afull\b;
```

whos; display contents of memory

Now, let us write out the contents of Afull into a file that we can read.

One option is to use the "save" command with the option -ascii, that writes to a file using the ASCII format.

```
save store1.dat Afull -ascii;
type store1.dat view contents of file
```

We can also load a file in this manner. The contents of the ASCII file filename.dat are stored in the MATLAB variable filename. This is a good way to import data from experiments or other programs into MATLAB.

```
load store1.dat;
```

If we add the option -double, the data is printed out with double the amount of digits for higher precision.

```
delete store1.dat;
save store1.dat Afull -ascii -double;
type store1.dat
```

We can use this command with multiple variables, but we see that no spaces are added.

```
delete store1.dat;
save store1.dat Afull b x -ascii;
type store1.dat view contents of file
delete store1.dat get rid of file
```

MATLAB also allows more complex formatted file input/output of data using commands that are similar to those in C.

First, we list all of the files in the directory.

```
ls
```

Next, we see create the output file and assign a label to it
with the "fopen" command that has the syntax
FID = fopen(FILENAME,PERMISSION)
where PERMISSION is usually one of :
'r' = read only
'w' = write (create if needed)
'a' = append (create if needed)
'r+' = read and write (do not create)
'w+' = create for read and write
'a+' = read and append (create if needed)

```
FID_out = fopen('test_io.dat','w');
ls
```

Now, we print the b vector to the output file as a column vector using the "fprintf" command.
In the FORMAT string '\n' signifies a carriage return, and 10.5f specifies a floating point decimal output with 5 numbers after the decimal point and a total field width of 10.

```
for i=1:length(b)
fprintf(FID_out,'10.5f \n',b(i));
end
```

We now close the file and show the results.

```
fclose(FID_out);
disp('Contents of test_io.dat : ');
type test_io.dat;
```

MATLAB's "fprintf" can also be loaded to avoid the need of
using a for loop

```
FID_out = fopen('test_io.dat','a');
fprintf(FID_out,'\n');
fprintf(FID_out,'10.5f \n',x);
fclose(FID_out);

disp('Contents of test_io.dat : ');
type test_io.dat;
```

We can also use "fprintf" to print out a matrix.

```
C = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
FID_out = fopen('test_io.dat','a');
fprintf(FID_out,'\n');
for i = 1:size(C,1)
fprintf(FID_out,'5.0f 5.0f 5.0f \n',C(i,:));
end
fclose(FID_out);

disp('Contents of test_io.dat : ');
type test_io.dat;
```

We can read in the data from the formatted file using
"fscanf", which works similarly to "fprintf".

First, we open the file for read-only.

```
FID_in = fopen('test_io.dat');
```

We now read the b vector into the variable b_new. First, we allocate space for the vector, and then we read in the values one by one.

```
b_new = linspace(0,0,num_pts)';
for i=1:num_pts
b_new(i) = fscanf(FID_in,'f',1);
end
b_new
```

Now read in x to x_new, using the overloading possible in MATLAB.

```
x_new = linspace(0,0,num_pts)';
x_new = fscanf(FID_in,'f',num_pts);
x_new
```

Finally, we read in the matrix C to C_new.

```
C_new = zeros(4,3);
for i=1:size(C,1)
for j=1:size(C,2)
C_new(i,j) = fscanf(FID_in,'f',1);
end
end
C_new

fclose(FID_in);

clear all
```

MATLAB Tutorial

Chapter 6. Writing and calling functions

In this chapter we discuss how to structure a program with multiple source code files. First, an explanation of how code files work in MATLAB is presented. In compiled languages such as FORTRAN, C, or C++, code can be stored in one or more source files that are linked together to form a single executable at the time of compilation. MATLAB, being an interpreted language, deals with multiple source files in a more open-ended manner. MATLAB code is organized into ASCII files carrying the extension .m (also known as m-files). MATLAB 6 has an integrated word processing and debugging utility that is the preferred mode of editing m-files, although other ASCII editors such as vi or emacs may also be used.
There are two different kinds of m-files. The simplest, a script file, is merely a collection of MATLAB commands. When the script file is executed by typing its name at the interactive prompt, MATLAB reads and executes the commands within the m-file just as if one were entering them manually. It is as if one were cutting and pasting the m-file contents into the MATLAB command window. The use of this type of m-file is outlined in section 6.1.

The second kind of m-file, discussed in section 6.2, contains a single function that has the same name as that of the m-file. This m-file contains an independent section of code with a clearly defined input/output interface; that is, it can be invoked by passing to it a list of dummy arguments arg1, arg2, ... and it returns as output the values out1, out2, ... The first non-commented line of a function m-file contains the function header, which is of the form :
function [out1,out2,...] = filename(arg1,arg2,...);
The m-file ends with the command return, which returns the program execution to the place where the function was called. The function code is executed whenever, either at the interactive command prompt or within another m-file, it is invoked with the command :
[outvar1,outvar2,...] = filename(var1,var2,...)
with the mapping of input to dummy arguments : arg1 = var1, arg2 = var2, etc. Within the function body, output values are assigned to the variables out1, out2, etc. When return is encountered, the current values of out1, out2, ... are mapped to the variables outvar1, outvar2, ... at the point where the function was called. MATLAB allows much latitude in writing functions with variable length argument and output variable lists. For example, the function could also be invoked by the command :
outvar1 = filename(var1,var2,...)
in which case only a single output variable is returned, containing on exit the value of the function variable out1. The input and output arguments may be strings, scalar numbers, vectors, matrices, or more advanced data structures.

Why use functions? As is well known from every computer science course, splitting a large program into multiple procedures that perform each a single well defined and commented task, results in programs that are easier to read, easier to modify, and that are more resistant to error. In MATLAB, one writes first a master file for the program, either a script file or better yet a function m-file that returns a single integer (that might return 1 for program success, 0 for incomplete program execution, or a negative value to indicate a run-time error), that is the point of entry to the program. This program file then calls upon code in other m-files by invoking them as functions. But if there is no compilation process to link all of the source code files together, how does MATLAB know where to look for a function when it is called?

MATLAB's program memory contains a search path list, the contents of which can be viewed with the command path, that stores the names of the directories it has been told contain function m-files. Initially, the path lists only the directories that hold the built-in MATLAB functions such as **sin(), exp(),** etc.. As demonstrated in section 6.2, one uses the command addpath to add to this list the name of each directory that contains a m-file for the present project. Then, when the MATLAB code interpreter encounters a function, say with the name **filename**, it starts at the top of the path list and works its way down searching in each directory for a file **filename.m**. When it finds it, it executes the file's code in the manner described above. For this reason, it is imperative that the names of the m-file and of the function agree; in fact it is only the filename that counts.

6.1. Writing and running m-files

While MATLAB can be run interactively from the command line, you can write a MATLAB program by composing a text file that contains the commands you want MATLAB to perform in the order in which they appear in the file. The standard file suffix for a text file containing a MATLAB program is .m. In the MATLAB command window, selecting the pull-down menu File -> New -> M-file opens the integrated MATLAB text editor for writing a m-file. This utility is very similar to word processors, so the use of writing and saving m-files is not explained in detail here.

As an example, use this secion as a file "MATLAB_tutorial_c6s1.m" that has only the following executable commands.

```
file_name = 'MATLAB_tutorial_c6s1.m';
disp(['Starting ' file_name ]);
j = 5;
for i=1:5
j = j - 1;
disp([int2str(i) ' ' int2str(j)]);
end
disp(['Finished ' file_name]);
```

We can run this m-file from the prompt by typing its name

```
>> MATLAB_tutorial_c6s1
```

If we type "whos" now, we see that the variables that are in the memory at the end of the program also remain in memory after the m-file is done running. This is because we have written the m-file as a script file where we have simply collected together several commands in a file, and then the code executes them one-by-one when the script is run, as if we were merely typing them into the interactive session window. A more common use for m-files is to isolate a series of commands in an independent function, as explained in the following section.

6.2. Structured programming with functions
Unstructured programming approach

File unstructured.m

In this section, let us demonstrate the use of subroutines to write structured, well-organized programs. We do so for a particularly simple and familiar case, the simple 1-D PDE problem that we encounted in section 4.1. First, in this m-file, we solve the problem with a program the combines all of the commands into a single file. This "unstructured" approach is fine for very small programs, but rapidly becomes confusion as the size of the program grows.

```
num_pts = 100; # of grid points
x = 1:num_pts; grid of x-values
```

We now set the values for the matrix discretizing the PDE with Dirichlet boundary conditions.

```
nzA = 3*(num_pts-2) + 2; # of non-zero elements
A = spalloc(num_pts,num_pts,nzA); allocate memory
```

set values

```
A(1,1) = 1;
A(num_pts,num_pts) = 1;
```

for i=2:(num_pts-1)
A(i,i) = 2;
A(i,i-1) = -1;
A(i,i+1) = -1;
end

Next, we set the values of the function at each boundary.
BC1 = -10; value of f at x(1);
BC2 = 10; value of f at x(num_pts);

We now create the vector for the right hand side of the problem.
b_RHS = linspace(0,0,num_pts)'; create column vector of zeros
b_RHS(1) = BC1;
b_RHS(num_pts) = BC2;
b_RHS(2:(num_pts-1)) = 0.05; for interior, b_RHS is source term

Now, we call the standard MATLAB solver.
f = A\b_RHS;

Then, we make a plot of the results.
figure; plot(x,f);
title('PDE solution from FD-CDS method (sparse matrix)');
xlabel('x'); ylabel('f(x)');

While this approach of putting all of the commands together works for this small program, it becomes very unwieldy for large programs.

clear all

Structured programming approach

File structured.m

NOTE: BEFORE RUNNING THIS FILE, THE OTHER M-FILES CONTAINING THE SUBROUTINES MUST ALREADY EXIST.
First, we define the number of points
num_pts = 100;

We now create a vector containing the grid points.
x = 1:num_pts;

In MATLAB, each function is stored in a separate m-file of the same name. When you call the function at the interactive session prompt or in another script or funtcion m-file, MATLAB searches through a list of directories that it has been told contain functions until it finds an m-file with the appropriate name. Then, it executes the MATLAB code contained within that m-file. When we write m-files that contain a functions, before we can use them we have to tell MATLAB where they are; that is, we have to add the name of their directory to the search **path.**

We can check the current contents of the search path with the command "path".
path

The command "pwd" returns the current directory.
pwd

We use the command "addpath" to add the directory with our subroutines to this search list. We can remove a directory from the path using "rmpath".

```
addpath(pwd);
path
```

The following function calculates the A matrix. A function call has the following syntax : [out1,out2,...] = func_name(in1,in2,...), where the input arguments are the variables in1,in2,... and the output from the function is stored in out1,out2,... In our case, the input is the dimension of the matrix A, num_pts, and the output variables are A and iflag, an integer that tells us if the code was performed sucessfully.

```
[A,iflag] = c6s2_get_A(num_pts);
if(iflag ~= 1) then error
disp(['c6s2_get_A returned error flag : ' int2str(iflag)]);
end
```

We also see from the code below that the existence of a local variable in the function named i does nothing to alter the value of i at the point of calling.

```
i = 1234;
[A,iflag] = c6s2_get_A(num_pts);
i
```

Next, we ask the user to input the function values at the boundaries.

```
BC1 = input('Input the function value at x = 1 : ');
BC2 = input('Input the function value at x = num_pts : ');
source = input('Input the value of the source term : ');
```

We now call upon another subroutine that calculates the vector for the RHS.

```
[b_RHS,iflag] = c6s2_get_b_RHS(num_pts,BC1,BC2,source);
```

We now solve the system.

```
f = A\b_RHS;
```

Then, we make plots of the output.

```
figure; plot(x,f);
phrase1 = ['PDE solution with source = ' num2str(source)];
phrase1 = [phrase1 ', BC1 = ' num2str(BC1)];
phrase1 = [phrase1 ', BC2 = ' num2str(BC2)];
title(phrase1); xlabel('x'); ylabel('f(x)');
```

Then, to clean up, we clear the memory

```
clear all
```

File c6s2_get_A.m

The first executable line of the m-file declares the name and input/output structure of the subroutine using the "function" command.

```
function [A,iflag] = c6s2_get_A(Ndim);
```

iflag = 0; signifies job not complete

If Ndim < 1, then we have an error, since we can't have a matrix with a dimension less than 1.
if(Ndim<1) signify error

A=-1;
iflag = -1;
we return control to the m-file that called this subroutine without executing the rest of the code.
return;
end

First, we declare A using sparse matrix format.
nzA = 3*(Ndim-2) + 2; # of non-zero elements
A = spalloc(Ndim,Ndim,nzA); allocate memory
A(1,1) = 1;
A(Ndim,Ndim) = 1;
for i=2:(Ndim-1)
A(i,i) = 2;
A(i,i-1) = -1;
A(i,i+1) = -1;
end

iflag = 1; signify job complete and successful

return; return control to the m-file that called this routine

File c6s2_get_b_RHS.m
function [b_RHS,iflag] = c6s2_get_b_RHS(num_pts,BC1,BC2,source);
iflag = 0; declares job not completed

if(num_pts < 3) not enough points
iflag = -1;
b_RHS = -1;
return;
end

We allocate space for b_RHS and initialize to zeros.
b_RHS = linspace(0,0,num_pts)';

Now, we specify the first and last components from the boundary conditions.
b_RHS(1) = BC1;
b_RHS(num_pts) = BC2;

Next, we specify the interior points.
for i=2:(num_pts-1)
b_RHS(i) = source;
end

iflag=1; signifies successful completion

return;

6.3. Inline functions

Sometimes, we do not want to go through the bother of writing a separate m-file to define a function. For these times, we can define an inline function. Let us say that we want to define the function
f1(x) = 2*x + 3*x^2

We can define this function using
f1 = inline('2*x + 3*x^2');

Then, we can call this function directly
f1(1), f1(23)

We can also define functions using vectors and matrices as input.
invariant2 = inline('(trace(A)*trace(A) - trace(A*A))/2');
A = rand(3);
invariant2(A)

We can check the definition of the function by typing its name
invariant2

While this is convenient, the execution of inline functions is rather slow.

clear all;
try
invariant2
catch
disp('We see that inline functions are cleared also');
end

6.4. Functions as function arguments

The function, trig_func_1, listed below,
returns the value of
f(x) = a*sin(x) + b*cos(x)
for given values of a, b, and x.
The function, plot_trig_1, listed below, plots a function on the domain 0 to 2*pi.

The following code asks the user to input values of a and b, and then uses plot_trig to plot trig_func_1 by
including the function name as an argument in the list.
disp('Plotting a*sin(x) + b*cos(x) ...');
a = input('Input a : ');
b = input('Input b : ');
func_name = 'trig_func_1';

make sure current direction is in the path
addpath(pwd)

plot_trig_1(func_name,a,b);

clear all

File trig_func_1.m
function f_val = trig_func_1(x,a,b);
f_val = a*sin(x) + b*cos(x);

return;

File plot_trig_1.m
function iflag = plot_trig_1(func_name,a,b);
iflag = 0; signifies no completion

First, create an x vector from 0 to 2*pi

```
num_pts = 100;
x = linspace(0,2*pi,num_pts);
```

Next, make a vector of the function values. We evaluate the argument function indirectly using the "feval" command.

```
f = linspace(0,0,num_pts);
for i=1:num_pts
f(i) = feval(func_name,x(i),a,b);
end
```

Then, we make the plot.

```
figure;
plot(x,f);
xlabel('Angle (radians)');
ylabel('Function value');

return;
```

Chapter 7. Data structures and input assertion

7.1. User-defined data structures

Vectors and matrices are not the only means that MATLAB offers for grouping data into a single entity. User defined data structures are also available that enable the programmer to create variable types that mix numbers, strings, and arrays. As an example, let us create a data structure that contains the information for a single student.

We will store the name, status (year and department), the homework and exam grades, and the final class grade.

First, we can define a NameData structure to contain the name. Here, the "." operator, used in the case of Structure.Field tells MATLAB to access the field named "Field" in the structure "Structure".

```
NameData.First = 'John';
NameData.MI = 'J';
NameData.Last = 'Doe';
```

We now create a StudentData structure with a name field.

```
StudentData.Name = NameData;
```

We now initialize the rest of the structure.

```
StudentData.Status = 'ChE grad 1';
StudentData.HW = 10;
StudentData.Exam = linspace(100,100,3);
```

We can now view the contents of the structure

```
StudentData
StudentData.Name
StudentData.Exam
```

We can operate on the elements of a structure.

```
StudentData.Exam(3) = 0;
StudentData.Exam
StudentData.Name.First = 'Jane';
StudentData.Name
```

We can also create arrays of structures

```
num_students = 5;
for i=1:num_students
ClassData(i) = StudentData;
end
ClassData
ClassData(2)
```

Structures can be passed as arguments to functions in the same manner as scalars, vectors, and matrices. In this case, we use the function pass_or_fail listed below.

```
message = pass_or_fail(ClassData(2));
message
```

File pass_or_fail.m

```
function message = pass_or_fail(StudentData)
Exam_avg = mean(StudentData.Exam);

if(Exam_avg >= 70)
message = 'You pass!';
else
message = 'You fail!';
end

return;
```

7.2. Input assertion routines

Good programming style dictates the practice of defensive programming, that is, anticipating and detecting possible errors before they cause a run-time error that results in a halt to the program execution or a crash. This allows one to save the current data to the disk or take corrective action to avoid a catastrophic failure. One common source of errors can be avoided by having each subroutine make sure that the data that it has been fed through its argument list is of the approriate type, e.g. argument 1 should be a real, positive, scalar integer and argument 2 should be a real, non-negative column vector of length N. The following m-files are useful for automating this checking process, and a scalar input function is provided to allow the robust entry of data from the keyboard.

assert_scalar.m

```
function [iflag_assert,message] = assert_scalar( ...
i_error,value,name,func_name, ...
check_real,check_sign,check_int,i_error);
```

This m-file contains logical checks to assert than an input value is a type of scalar number. This function is passed the value and name of the variable, the name of the function making the assertion, and four integer flags that have the following usage :

i_error : controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise just return the appropriate negative number.
if i_error > 1, then dump current state to dump_error.mat before calling error().

check_real : check to examine whether input number is real or not. See table after function header for set values of these case flags
check_real = i_real (make sure that input is real)
check_real = i_imag (make sure that input is purely imaginary)
any other value of check_real (esp. 0) results in no check

check_real
i_real = 1;
i_imag = -1;

check_sign : check to examine sign of input value see table after function header for set values
of these case flags
check_sign = i_pos (make sure input is positive)
check_sign = i_nonneg (make sure input is non-negative)
check_sign = i_neg (make sure input is negative)
check_sign = i_nonpos (make sure input is non-positive)
check_sign = i_nonzero (make sure input is non-zero)
check_sign = i_zero (make sure input is zero)

any other value of check_sign (esp. 0) results in no check

check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

check_int : check to see if input is an integer
if = 1, then check to make sure input is an integer
any other value, perform no check

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/21/2001

```
function [iflag_assert,message] = assert_scalar( ...
i_error,value,name,func_name, ...
check_real,check_sign,check_int);
```

```
iflag_assert = 0;
message = 'false';
```

First, set case values of check integer flags.
check_real
```
i_real = 1;
i_imag = -1;
```
check_sign
```
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;
```

Check to make sure input is numerical and not a string.
```
if(~isnumeric(value))
message = [ func_name, ': ', ...
name, ' is not numeric'];
iflag_assert = -1;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Check to see if it is a scalar.

```
if(max(size(value)) ~= 1)
message = [ func_name, ': ', ...
name, ' is not scalar'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Then, check to see if it is real.

```
switch check_real;

case {i_real}
if(~isreal(value))
message = [ func_name, ': ', ...
name, ' is not real'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_imag}
if(real(value))
message = [ func_name, ': ', ...
name, ' is not imaginary'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Next, check sign.

```
switch check_sign;

case {i_pos}
if(value <= 0)
message = [ func_name, ': ', ...
name, ' is not positive'];
iflag_assert = -4;
```

```
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonneg}
if(value < 0)
message = [ func_name, ': ', ...
name, ' is not non-negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_neg}
if(value >= 0)
message = [ func_name, ': ', ...
name, ' is not negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonpos}
if(value > 0)
message = [ func_name, ': ', ...
name, ' is not non-positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonzero}
if(value == 0)
message = [ func_name, ': ', ...
name, ' is not non-zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_zero}
if(value ~= 0)
message = [ func_name, ': ', ...
name, ' is not zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Finally, check to make sure it is an integer.

```
if(check_int == 1)
if(round(value) ~= value)
message = [ func_name, ': ', ...
name, ' is not an integer'];
iflag_assert = -5;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

set flag for succesful passing of all checks

```
iflag_assert = 1;
message = 'true';

return;
```

assert_vector.m

```
function [iflag_assert, message] = ...
assert_vector( ...
i_error,value,name,func_name,num_dim, ...
```

check_real,check_sign,check_int,check_column);

This m-file contains logical checks to assert than an input value is a vector of a given type.
This function is passed the value and name of the variable, the name of the function making the
assertion, the dimension that the vector is supposed to be, and five integer flags that have the following usage :

i_error : controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise
just return the appropriate negative number.
if i_error > 1, create file dump_error.mat
before calling error()

check_real : check to examine whether input is real
see table after function header for set
values of these case flags
check_real = i_real (make sure that input is real)
check_real = i_imag (make sure that input
is purely imaginary)
any other value of check_real (esp. 0)
results in no check

check_real
i_real = 1;
i_imag = -1;

check_sign : check to examine sign of input see table after function header for set
values of these case flags
check_sign = i_pos (make sure input is positive)
check_sign = i_nonneg (make sure input is non-negative)
check_sign = i_neg (make sure input is negative)
check_sign = i_nonpos (make sure input is non-positive)
check_sign = i_nonzero (make sure input is non-zero)
check_sign = i_zero (make sure input is zero)
any other value of check_sign (esp. 0)
results in no check

check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

check_int : check to see if input is an integer
if = 1, then check to make sure input is an integer
any other value, perform no check

check_column : check to see if input is a column or row vector
check_column = i_column (make sure input is
column vector)
check_column = i_row (make sure input is row vector)
any other value, perform no check

check_column
i_column = 1;

i_row = -1;

if the dimension num_dim is set to zero, no check as to the dimension of the vector is made.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/21/2001

function [iflag_assert,message] = ...
assert_vector(...
i_error,value,name,func_name,num_dim, ...
check_real,check_sign,check_int,check_column);

First, set case values of check integer flags.
check_real
i_real = 1;
i_imag = -1;
check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;
check_column
i_column = 1;
i_row = -1;

iflag_assert = 0;
message = 'false';

Check to make sure input is numerical and not a string.
if(~isnumeric(value))
message = [func_name, ': ', ...
name, 'is not numeric'];
iflag_assert = -1;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

Check to see if it is a vector of the proper length.
num_rows = size(value,1);
num_columns = size(value,2);
if it is a multidimensional array
if(length(size(value)) > 2)
message = [func_name, ': ', ...
name, 'has too many subscripts'];

```
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

if both the number of rows and number of columns are not equal to 1, then value is a matrix instead of a vector.

```
if(and((num_rows ~= 1),(num_columns ~= 1)))
message = [ func_name, ': ', ...
name, 'is not a vector'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

if the dimension of the vector is incorrect

```
if(num_dim ~= 0)
if(length(value) ~= num_dim)
message = [ func_name, ': ', ...
name, 'is not of the proper length'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

check to make sure that the vector is of the correct type (e.g. column)

```
switch check_column;

case {i_column}
```

check to make sure that it is a column vector

```
if(num_columns > 1)
message = [ func_name, ': ', ...
name, 'is not a column vector'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
```

```
else
return;
end
end

case {i_row}
if(num_rows > 1)
message = [ func_name, ': ', ...
name, 'is not a row vector'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Then, check to see if all elements are of the proper complex type.

```
switch check_real;

case {i_real}
```

if any element of value is not real

```
if(any(~isreal(value)))
message = [ func_name, ': ', ...
name, ' is not real'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_imag}
```

if any element of value is not purely imaginary

```
if(any(real(value)))
message = [ func_name, ': ', ...
name, ' is not imaginary'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Next, check sign.

```
switch check_sign;

case {i_pos}
```

if any element of value is not positive

```
if(any(value <= 0))
message = [ func_name, ': ', ...
name, ' is not positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonneg}
```

if any element of value is negative

```
if(any(value < 0))
message = [ func_name, ': ', ...
name, ' is not non-negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_neg}
```

if any element of value is not negative

```
if(any(value >= 0))
message = [ func_name, ': ', ...
name, ' is not negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonpos}
```

if any element of value is positive

```
if(any(value > 0))
message = [ func_name, ': ', ...
name, ' is not non-positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

```
case {i_nonzero}
```
if any element of value is zero
```
if(any(value == 0))
message = [ func_name, ': ', ...
name, 'is not non-zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

```
case {i_zero}
```
if any element of value is non-zero
```
if(any(value ~= 0))
message = [ func_name, ': ', ...
name, ' is not zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Finally, check to make sure it is an integer.
```
if(check_int == 1)
if(any(round(value) ~= value))
message = [ func_name, ': ', ...
name, ' is not an integer'];
iflag_assert = -5;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

set flag for succesful passing of all checks

iflag_assert = 1;
message = 'true';

return;

assert_matrix.m

function [iflag_assert,message] = assert_matrix(...
i_error,value,name,func_name, ...
num_rows,num_columns, ...
check_real,check_sign,check_int);

This m-file contains logical checks to assert than an input value is a matrix of a given type. This function is passed the value and name of the variable, the name of the function making the assertion, the dimension that the matrix is supposed to be, and four integer flags that have the following usage :

i_error : controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise just return the appropriate negative number.
if i_error > 1, create file dump_error.mat
before calling error()

check_real : check to examine whether input is real see table after function header for set values of these case flags
check_real = i_real (make sure that input is real)
check_real = i_imag (make sure that input is
purely imaginary)
any other value of check_real (esp. 0)
results in no check

check_real
i_real = 1;
i_imag = -1;

check_sign : check to examine sign of input
see table after function header for set
values of these case flags
check_sign = i_pos (make sure input is positive)
check_sign = i_nonneg (make sure input is non-negative)
check_sign = i_neg (make sure input is negative)
check_sign = i_nonpos (make sure input is non-positive)
check_sign = i_nonzero (make sure input is non-zero)
check_sign = i_zero (make sure input is zero)
any other value of check_sign (esp. 0)
results in no check

check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

check_int : check to see if input value is an integer

if = 1, then check to make sure input is an integer
any other value, perform no check

if the dimensions num_rows or num_columns
are set to zero, no check as to that dimension of the matrix is made.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/21/2001

```
function [iflag_assert,message] = assert_matrix( ...
i_error,value,name,func_name, ...
num_rows,num_columns, ...
check_real,check_sign,check_int);
```

First, set case values of check integer flags.
check_real

```
i_real = 1;
i_imag = -1;
check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

iflag_assert = 0;
message = 'false';
```

Check to make sure input is numerical and not a string.

```
if(~isnumeric(value))
message = [ func_name, ': ', ...
name, ' is not numeric'];
iflag_assert = -1;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Check to see if it is a matrix of the proper length.
if it is a multidimensional array

```
if(length(size(value)) > 2)
message = [ func_name, ': ', ...
name, ' has too many subscripts'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
```

```
save dump_error.mat;
end
error(message);
else
return;
end
end
```

check that value has the proper number of rows

```
if(num_rows ~= 0)
if(size(value,1) ~= num_rows)
message = [ func_name, ': ', ...
name, ' has the wrong number of rows'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

check that value has the proper number of columns

```
if(num_columns ~= 0)
if(size(value,2) ~= num_columns)
message = [ func_name, ': ', ...
name, ' has the wrong number of columns'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Then, check to see if all elements are of the proper complex type.

```
switch check_real;

case {i_real}
```

if any element of value is not real

```
if(any(~isreal(value)))
message = [ func_name, ': ', ...
name, ' is not real'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
```

```
end
end

case {i_imag}
if any element of value is not purely imaginary
if(any(real(value)))
message = [ func_name, ': ', ...
name, ' is not imaginary'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Next, check sign.

```
switch check_sign;

case {i_pos}
if any element of value is not positive
if(any(value <= 0))
message = [ func_name, ': ', ...
name, ' is not positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonneg}
if any element of value is negative
if(any(value < 0))
message = [ func_name, ': ', ...
name, ' is not non-negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_neg}
if any element of value is not negative
```

```
if(any(value >= 0))
message = [ func_name, ': ', ...
name, ' is not negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonpos}
if any element of value is positive
if(any(value > 0))
message = [ func_name, ': ', ...
name, ' is not non-positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonzero}
if any element of value is zero
if(any(value == 0))
message = [ func_name, ': ', ...
name, 'is not non-zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_zero}
if any element of value is non-zero
if(any(value ~= 0))
message = [ func_name, ': ', ...
name, ' is not zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
```

```
return;
end
end
end
```

Finally, check to make sure it is an integer.

```
if(check_int == 1)
if(any(round(value) ~= value))
message = [ func_name, ': ', ...
name, ' is not an integer'];
iflag_assert = -5;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

set flag for succesful passing of all checks

```
iflag_assert = 1;
message = 'true';

return;
```

assert_structure.m

function [iflag_assert,message] = assert_structure(...
i_error,Struct,struct_name,func_name,StructType);

This MATLAB m-file performs assertions on a data structure. It makes use of assert_scalar, assert_vector, and assert_matrix for the fields.

INPUT :
=======
i_error controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise just return the appropriate negative number.
if i_error > 1, then dump current state to dump_error.mat before calling error().
Struct This is the structure to be checked
struct_name the name of the structure
func_name the name of the function making the assertion
StructType this is a structure that contains the typing data for each field.
.num_fields is the total number of fields
Then, for i = 1,2, ..., StructType.num_fields, we have :
.field(i).name the name of the field
.field(i).is_numeric if non-zero, then field is numeric
.field(i).num_rows # of rows in field
.field(i).num_columns # of columns in field
.field(i).check_real value of check_real passed to assertion
.field(i).check_sign value of check_sign passed to assertion
.field(i).check_int value of check_int passed to assertion

OUTPUT :
=======
iflag_assert an integer flag telling of outcome message a message passed that describes the result of making the assertion

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/25/2001

```
function [iflag_assert,message] = assert_structure(...
i_error,Struct,struct_name,func_name,StructType);
```

```
iflag_assert = 0;
message = 'false';
```

first, check to make sure Struct is a structure

```
if(~isstruct(Struct))
iflag_assert = -1;
message = [func_name, ': ', struct_name, ...
' is not a structure'];
if(i_error ~= 0)
if(i_error > 1);
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Now, for each field, perform the required assertion.

```
for ifield = 1:StructType.num_fields
```

set shortcut to current field type

```
FieldType = StructType.field(ifield);
```

check if it exists in Struct

```
if(~isfield(Struct,FieldType.name))
iflag_assert = -2;
message = [func_name, ': ', struct_name, ...
' does not contain ', FieldType.name];
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

extract value of field

```
value = getfield(Struct,FieldType.name);
```

if the field is supposed to be numeric

```
if(FieldType.is_numeric ~= 0)
```

check to make sure field is numeric

```
if(~isnumeric(value))
iflag_assert = -3;
message = [func_name, ': ', ...
struct_name, '.', FieldType.name, ...
' is not numeric'];
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

decide which assertion statement to use based on array dimension of field value
If both num_rows and num_columns are set equal to zero, then no check of the dimension of this field is made.

```
if(and((FieldType.num_rows == 0), ...
(FieldType.num_columns == 0)))

message = [func_name, ': ', ...
struct_name,'.',FieldType.name, ...
' is not checked for dimension'];
if(i_error ~= 0)
disp(message);
end
```

else, peform check of dimension to make sure it is a scalar, vector, or matrix (i.e. a two dimensional array).

```
else
```

check that is is not a multidimensional array

```
if(length(size(value)) > 2)
iflag_assert = -4;
message = [func_name, ': ', ...
struct_name,'.',FieldType.name, ...
' is multidimensional array'];
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
```

else if scalar

```
elseif(and((FieldType.num_rows == 1), ...
(FieldType.num_columns == 1)))
assert_scalar(i_error,value, ...
[struct_name,'.',FieldType.name], ...
```

func_name,FieldType.check_real, ...
FieldType.check_sign,FieldType.check_int);

else if a column vector
elseif (and((FieldType.num_rows > 1), ...
(FieldType.num_columns == 1)))
dim = FieldType.num_rows;
check_column = 1;
assert_vector(i_error,value, ...
[struct_name,'.',FieldType.name], ...
func_name,dim,FieldType.check_real, ...
FieldType.check_sign,FieldType.check_int, ...
check_column);

else if a row vector
elseif (and((FieldType.num_rows == 1), ...
(FieldType.num_columns > 1)))
dim = FieldType.num_columns;
check_column = -1;
assert_vector(i_error,value, ...
[struct_name,'.',FieldType.name], ...
func_name,dim,FieldType.check_real, ...
FieldType.check_sign,FieldType.check_int, ...
check_column);

otherwise, a matrix
else
assert_matrix(i_error,value, ...
[struct_name,'.',FieldType.name], ...
func_name, ...
FieldType.num_rows,FieldType.num_columns, ...
FieldType.check_real,FieldType.check_sign, ...
FieldType.check_int);

end selection of assertion routine
end if perform check of dimension
end if (FieldType.is_numeric ~= 0)
end for loop over fields

set return results for succesful assertion

iflag_assert = 1;
message = 'true';

return;

get_input_scalar.m

function value = get_input_scalar(prompt, ...
check_real,check_sign,check_int);

This MATLAB m-file gets from the user an input scalar value of the appropriate type. It asks for input over and over again until a correctly typed input value is entered.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/25/2001

```
function value = get_input_scalar(prompt, ...
check_real,check_sign,check_int);

func_name = 'get_input_scalar';
name = 'trial_value';

input_OK = 0;

while (input_OK ~= 1)
trial_value = input(prompt);
[iflag_assert,message] = ...
assert_scalar(0,trial_value, ...
name,func_name, ...
check_real,check_sign,check_int);
if(iflag_assert == 1)
input_OK = 1;
value = trial_value;
else
disp(message);
end
end

return;
```

Chapter 8. MATLAB compiler

The previous chapters have discussed programming within the MATLAB environment. It has been noted that MATLAB is an interpreted language, meaning that each command is converted to machine-level instructions one-by-one during execution. While this allows one to program in both interactive and batch mode, the extra overhead required to convert the commands at run-time is not desired. Also, any programs written in MATLAB can only be run on computers that have a copy of MATLAB, so portability is limited. MATLAB includes an optional compiler to circumvent these problems by converting m-files to C or C++ code, and optionally linking this code with its mathematics and graphics libraries to produce a stand-alone executable that may be run, without the interpretation overhead, on any machine with a compatible operating system platform. In this section, we demonstrate the MATLAB compiler to produce a stand-alone executable from the simple example of section 6.4. Note that the program containing the main program has been rewritten from the previous script file version since the MATLAB compiler only works with function m-files. The first file, a script file called make_file.m, is executed from the interactive prompt to perform the compilation; alternatively, the command mcc ... can be entered manually.

make_file.m

This MATLAB script m-file calls the compiler to convert the MATLAB source code files for make_plot_trig to C, link the object files with the MATLAB graphics library, and then produce a stand-alone executable.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/31/2001

```
mcc -B sgl ...
make_plot_trig ...
plot_trig_1 ...
trig_func_1 ...
get_input_scalar ...
assert_scalar
```

make_plot_trig.m (main program file)

make_plot_trig.m

This MATLAB m-file makes a plot of the general function
f(x) = a*sin(x) + b*cos(x)
for user-selected values of a and b.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/31/2001

function iflag_main = make_plot_trig();

iflag_main = 0; signifies no completion

```
disp('RUNNING make_plot_trig ...');
disp(' ');
disp('This program produces a plot in [0,2*pi]');
disp('of the function : ');
disp('f(x) = a*sin(x) + b*cos(x)');
disp('for user-input values of the real scalars a and b');
disp(' ');
```

The following code asks the user to input values of a and b, and then uses plot_trig to plot trig_func_1 by including the function name as an argument in the list.

```
prompt = 'Input a : ';
check_real=1; check_sign=0; check_int=0;
a = get_input_scalar(prompt, ...
check_real,check_sign,check_int);

prompt = 'Input b : ';
check_real=1; check_sign=0; check_int=0;
b = get_input_scalar(prompt, ...
check_real,check_sign,check_int);
```

We now call the routine that produces the plot.

```
func_name = 'trig_func_1';
plot_trig_1(func_name,a,b);
```

We now require the user to strike a key before exiting the program.

```
pause

iflag_main = 1;

return;
```

plot_trig_1.m

```
function iflag = plot_trig_1(func_name,a,b);

iflag = 0; signifies no completion
```

First, create an x vector from 0 to 2*pi

```
num_pts = 100;
x = linspace(0,2*pi,num_pts);
```

Next, make a vector of the function values. We evaluate the argument function indirectly using the "feval" command.

```
f = linspace(0,0,num_pts);
for i=1:num_pts
f(i) = feval(func_name,x(i),a,b);
end
```

Then, we make the plot.

```
figure;
plot(x,f);
xlabel('Angle (radians)');
ylabel('Function value');
```

```
return;
```

trig_func_1.m

```
function f_val = trig_func_1(x,a,b);
f_val = a*sin(x) + b*cos(x);

return;
```