

Matlab II: Computing and Programming



Updated: August 2012

Table of Contents

Section 5: Computational Procedures.....	4
Special Built-in Constants	4
Special Built-in Functions	6
Compound Expressions and Operator Precedence	8
Commutivity of Operations and Finite Decimal Expansion Approximations.....	10
Computing with matrices and vectors	10
Simultaneous linear equations	11
Eigenvectors and Eigenvalues	12
Roots of polynomials and zeros of functions.....	14
Poles, residues, and partial fraction expansion	15
Convolution and deconvolution.....	17
Fast Fourier and Inverse Fourier Transforms	17
Numerical Differentiation and Integration	18
Numerical solution of differential equations	20
Section 6: Programming.....	22
Using the Editor	22
Types and Structures of M-files.....	23
The Shortcut Utility	24
Publishing to HTML.....	25
Internal Documentation	26
Passing variables by name and value.....	27
Function evaluation and function handles	29
Function recursion	30
Flow control	30
String evaluation and manipulation	33
Keyboard input	34
Multidimensional arrays and indexing	35
Debugging.....	37
Using Matlab with External Code	41

Exchanging and viewing text information..... 41
Compiling and calling external files from Matlab..... 42
Calling Matlab objects from external programs 44
Using Java Classes in Matlab 45

Note: See Matlab I: Getting Started for more information about this tutorial series including its organization and for more information about the Matlab software. Before proceeding with this tutorial, download [Part 2 Computing and Programming.zip](#) . This zip file contains the example files needed for this portion of the tutorial.

Section 5: Computational Procedures

The fundamental concept to remember when doing computations with Matlab is that everything is actually a matrix, however it may be displayed. Indeed, the name *Matlab* itself is a contraction of the words “MATrix LABoratory”. Thus the input

```
>> a = 1
```

does not directly assign the integer **1** to the variable **a**, but rather assigns a 1×1 matrix whose first row, first column element is the integer **1**. Similarly, the input

```
>> b = 'matlab'
```

does not directly assign that character string to the variable, but rather assigns a 1×6 matrix, i.e., a six element row vector of type *text*. The elements are displayed as characters because of the *text* data type, but the underlying structure is a vector of integers representing the ASCII character code values for the letters, which becomes apparent if it is multiplied by the integer **1**, e.g.,

```
>> c = 1.*'matlab'
```

```
c =
```

```
109    97    116    108    97    98
```

Logical expressions can be assigned to a variable, with a binary value specified by their truth or falsity, but the actual variable is again a 1×1 matrix whose element is **1** or **0** rather than the binary digit itself. For example,

```
>> d = (1+1 == 2)
```

creates a logical 1×1 matrix [**1**] with the property that $(d + d)$ will produce the numeric 1×1 matrix [**2**], while $(d \& d)$ will produce the logical 1×1 matrix [**1**].

Special Built-in Constants

In addition to the default assignment for **i** and **j** as the complex-valued $\sqrt{-1}$, Matlab has several other built-in constants, some of which can be overwritten by assignment, but which are then restorable to the original default with a clearing command. Some commonly used ones with universal meaning are

pi for a rational approximation of π

inf for generic infinity (without regard to cardinality)

NaN for a numerically derived expression that is not a number, e.g., $\frac{0}{0}$

One particular built-in constant is a *de facto* variable that is constantly being overwritten:

ans for the evaluation of the most recent input expression with no explicit assignment

Care needs to be taken that no variable is intentionally assigned the name *ans* because that variable name can be reassigned a value without explicit instruction during the execution of a program. Thus, though *years* is a commonly used variable name in computational problems, its French equivalent *ans* should be avoided. Similar to this are random number “constants”, also *de facto* variables because the values are overwritten every time they are called:

rand for a random number drawn from a uniform distribution in [0,1]

randn for a random number from a normal distribution with zero mean and unit variance

Each of these functions can generate a matrix of random numbers if given two integer arguments, for example

```
>> x = rand(2,3)
x =
    0.9706    0.4854    0.1419
    0.9572    0.8003    0.4218
```

These random number "constants" are similar to the **i** and **j** used with complex numbers in terms of retaining their meaning. They represent pseudo random numbers which are revalued sequentially from a starting seed, but once assigned specific values the built-in definitions are lost. However, the built-in functionality can be restored with the commands *clear rand* and *clear randn*.

There are also some built-in constants that are characteristic of the particular computer on which calculations are being performed, again susceptible to being overwritten, such as

realmin for the smallest positive real number that can be used
realmax for the largest positive real number that can be used
eps for the smallest real positive number such that $(1 + eps)$ differs from (1)

Some built-in "constants" are matrices constant by structure but variable in dimension:

zeros(m,n) for an $m \times n$ matrix whose elements are all 0
ones(m,n) for an $m \times n$ matrix whose elements are all 1
eye(m,n) for an $m \times n$ matrix whose main diagonal elements are 1 with 0 elsewhere

For each of these functions a single argument n will produce an $n \times n$ matrix.

Special Built-in Functions

Matlab has very many named functions that come with the application. As with variable names, function names are also case sensitive. The pre-defined built-in functions usually have names whose alphabetic characters are all lower case. There are too many to enumerate in a tutorial, but some that have substantial utility in computation and mathematical manipulation and which use standard nomenclature are the usual trigonometric functions and their inverses (angle values in radians)

sin, cos, tan, csc, sec, cot, asin, acos, atan, acsc, asec, acot ;

and the hyperbolic trigonometric functions and their inverses (angle values in radians)

sinh, cosh, tanh, csch, sech, coth, asinh, acosh, atanh, acsch, asech, acoth ;

There are equivalent elementary functions which use angle arguments in degrees:

sind, cosd, tand, cscd, sec, cotd, asind, acosd, atand, acscd, asec, acotd

but the hyperbolic trigonometric functions do not have such an equivalent.

Also there are built-in exponential and various base logarithm functions

exp exponential of the argument
log natural logarithm of the argument
log2 base 2 logarithm of the argument
log10 base 10 logarithm of the argument

Matlab also has several less commonly used built-in mathematical functions which can be very useful in particular circumstances, such as various Bessel functions, gamma functions, beta functions, error functions, elliptic functions, Legendre function, Airy function, etc. All of these built-in mathematical functions are, of course, algorithms to compute rational numerical representation within word memory size, and thus do not produce exact values either for irrational numbers or for numbers whose decimal expansion does not terminate within machine limits.

In addition to these functions that do a numerical mapping, there are also built-in functions for characterizing an argument. Functions for characterizing complex numbers, some of which also are applicable to ordinary real numbers, include

abs for the absolute value
conj for the complex conjugate
real for the real part
imag for the imaginary part

Similarly, there are built-in functions that characterize ordinary real numbers

factor for producing a vector of prime number components
gcd for greatest common divisor
lcm for least common multiple
round for closest integer
ceil for closest integer of greater than or equal value
floor for closest integer of lesser than or equal value

Matlab also has built-in functions to characterize vectors of elements

length for the number of elements
min for the minimum value among the elements
max for the maximum value of among elements
mean for the mean value of the elements
median for the median value of the elements
std for the standard deviation from the mean of element values

There are some built-in logical functions as well, valued as **1** when true and **0** when false:

isprime for status of a number as prime
isreal for status of a number as real
isfinite for status of a number being finite

isinf for status of a number being infinite

Matlab also has several built-in functions for vectors and matrices in addition to those which are extensions of built-in functions for individual numerical matrix elements

size for number of rows and columns
det for determinant
rank for matrix rank
inv for inverse
trace for sum of diagonal elements
transpose for transposing rows and columns
ctranspose for matrix complex conjugate transpose (adjoint)
eig for column vector of eigenvalues
fft for fast Fourier transform
ifft for inverse fast Fourier transform
sqrtm for matrix square root
expm for matrix exponentiation
logm for matrix logarithm

Compound Expressions and Operator Precedence

Users can construct their own functions by defining algorithms or expressions that combine simpler or built-in functions and constants: by producing composites from nesting of simpler functions; by scaling or mapping to a different domain, etc.; and by installing computational code as function m-files in the Matlab search path. Some simple representative examples, presented here by definition with generic arguments **x** and **y** are:

```
>> e = x + log(x)
>> f = cos(sin(tan(x + sqrt(y))))
>> g = atan(y/x)
>> h = sqrt(x^2 + y^2)
```

Similarly, manipulations can be done to create compound expressions involving matrices and vectors. For simple examples, suppose **M** and **N** are generic square matrices of the same dimension. Some sample user-created composite expressions are:

```
>> P = M^2 - 2.*(M*N) + N^2      % example of concatenation
>> Q = 1./(M./(N + 1./M))      % example of nesting
>> R = inv(sqrtm(cosh(M)))      % example of composition
```

When the construction of composite expressions gets complicated the order and precedence of operations becomes important. In Matlab, expressions are evaluated first with grouping of parentheses, innermost first sequentially to outermost whenever there is nesting. Evaluation then takes place from left to right sequentially at each level of operator precedence. This is of importance when binary operations are not commutative. The precedence order is $\{ .^{\wedge}, \wedge \} > \{ \text{unary } +- \} > \{ .*, *, ./, /, \backslash \} > \{ +, - \}$ but the sequence of operations can be specified explicitly by parenthesis grouping. As an example consider the following value assignment with no grouping

```
>> k = 1-2^3/4+5*6
```

Matlab would first go from left to right in the right hand side expression looking for a right (i.e., closing) parenthesis. Finding none, it would then search from left to right looking for exponentiation operators. There is one present, linking the digit **2** with the digit **3**, so the first step in constructing a value would be to assign the segment 2^3 with its mathematical value of 8. That exhausts the top level of precedence, so the procedure is recursively repeated left to right at each level of lower precedence giving a sequence

```
k = 1-8/4+5*6
```

```
k = 1-2+5*6
```

```
k = 1-2+30
```

```
k = -1+30
```

```
k = 29
```

But what if there were some grouping parameters in the expression? Using the same sequence of elements in **k** suppose we have the value assignment

```
>> m = (1-2)^(3/(4+5)*6)
```

Now when Matlab initially goes from left to right looking for a right parenthesis it will find one immediately following the element **2**. So it then immediately backtracks to find the closest left parenthesis and does a sub-evaluation of the elements within that set of grouping parentheses. In this case it finds only one operation, i.e., $(1-2)$, and evaluates it to (-1) . Then the search continues to the right until the next right parenthesis is found. This continues until all groupings have been collapsed to single elements, after which the recursive left to right processing based on operator precedence continues. In this particular example, the result would be

```
m = (-1)^(3/(4+5)*6)
```

```
m = (-1)^(3/9*6)
```

```
m = (-1)^(0.33333...*6)
```

```
m = (-1)^(2)
m = 1
```

Logical operators also have a precedence order: { ~ } > { & } > { | }.

Commutivity of Operations and Finite Decimal Expansion Approximations

By definition matrix multiplication is non-commutative, e.g., $M*N$ is different in general from $N*M$, so many expressions involving matrix operations will also be dependent on positioning relative to operators. Addition and multiplication operations on scalars should be commutative, but computational procedures involving round off or irrational number representation by finite decimal expansion can occasionally lead to unexpected results, for example:

```
>> t = 0.4 + 0.1 - 0.5 % assigns a value 0 to the variable t
>> u = 0.4 - 0.5 + 0.1 % assigns a value 2.7756e-017 to the variable u
```

The same problem can also lead to occasional surprises in the value assignment for logical variables, for example

```
>> v = (sin(2*pi) == sin(4*pi))
```

assigns a value **0** to the variable **v** (logical false), giving the impression that the sine function does not have a period of 2π .

Computing with matrices and vectors

The matrix is the fundamental object in Matlab. Generically a matrix is an **n** row by **m** column array of numbers or objects corresponding to numbers. When **n** is 1 the matrix is a row vector, when **m** is 1 the matrix is a column vector, and when both **n** and **m** are 1 the 1×1 matrix corresponds to a scalar. Development of an end user interface for easy access of subroutines used in numerical computations with matrices, particularly in the context of linear algebra, was the original concept behind the development of Matlab. Thus, calculations involving matrices are really well-suited for the Matlab application. Matrix calculations also can represent an efficiency in doing simultaneous manipulations, i.e., vectorized calculations. This can eliminate the need to cycle through nested loops of serial commands. As an example, let's compare a matrix computation with an equivalent calculation using scalars. Suppose we have a matrix

$$\text{power1} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The vectorized matrix procedure

```
>> power2 = power1.^2
```

is more efficient than is the scalar procedure:

```
>> for nrow=1:3
    for mcolumn = 1:3
        power2(nrow, mcolumn) = (power1(nrow, mcolumn)).^2
    end
end
```

Simultaneous linear equations

One of the most powerful and commonly used matrix calculation procedures involves the solution of a system of simultaneous linear equations. As an illustrative example we will consider a very elementary case: the grocery bill for customers Abe, Ben, and Cal who purchase various numbers of apples, bananas, and cherries:

Customer	Apples	Bananas	Cherries	Cost
Abe	1	3	2	0.41
Ben	3	2	1	0.37
Cal	2	1	3	0.36

From this data we want to know the costs of an individual apple, an individual banana, and an individual cherry. Using Matlab we can create a count matrix **A** that has the quantities purchased arranged with the rows representing the customers and the columns representing the types of fruit, i.e.,

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 3 \end{pmatrix}$$

and a column vector **b** representing the total cost for each customer, i.e.,

$$\mathbf{b} = \begin{pmatrix} 0.41 \\ 0.37 \\ 0.36 \end{pmatrix}$$

If we then create a column vector \mathbf{x} representing the individual cost of an apple, a banana, and a cherry in descending order, i.e.,

$$\mathbf{x} = \begin{pmatrix} x(1) \\ x(2) \\ x(3) \end{pmatrix}$$

then the system can be stated as the matrix equation

$$\mathbf{A} * \mathbf{x} = \mathbf{b}$$

Multiplying each side from the left by the inverse of the matrix \mathbf{A} will give

$$((\text{inv}(\mathbf{A})) * \mathbf{A}) * \mathbf{x} = (\text{inv}(\mathbf{A})) * \mathbf{b}$$

But a matrix multiplied by its inverse from the right or left gives an identity matrix \mathbf{I} with ones on the diagonal and zeros elsewhere, so that

$$((\text{inv}(\mathbf{A})) * \mathbf{A}) * \mathbf{x} = \mathbf{I} * \mathbf{x} = \mathbf{x}$$

Thus, the simple solution using Matlab is the command

```
>> x = (inv(A)) * b
```

which will produce the vector with the unit costs of each individual fruit:

$$\mathbf{x} = \begin{pmatrix} 0.0500 \\ 0.0800 \\ 0.0600 \end{pmatrix}$$

Thus the grocer charges 5 cents for an apple, 8 cents for a banana, and 6 cents for a cherry.

Eigenvectors and Eigenvalues

Because Matlab was originally developed primarily for linear algebra computations, it is particularly suited for obtaining eigenvectors and eigenvalues of square matrices with non-zero determinants. The determinant of a square $n \times n$ matrix \mathbf{A} from which a constant λ has been subtracted from the diagonal elements will be a polynomial of degree n in λ and the roots of that polynomial are the eigenvalues of the original square matrix \mathbf{A} . The eigenvectors of such a matrix will be represented by a matrix of n column vectors, with n elements each, assembled as an n by n eigenvector matrix \mathbf{V} . If the n eigenvalues are inserted along the diagonal of an n by n matrix of zeros to form the diagonal eigenvalue matrix \mathbf{D} , then original matrix \mathbf{A} corresponds to

```
>> A = V*D*(inv(V))
```

After left multiplication by $\text{inv}(\mathbf{V})$ and right multiplication by \mathbf{V} on each side of the equation it can be seen that the diagonal matrix \mathbf{D} is equivalent to $\text{inv}(\mathbf{V}) * \mathbf{A} * \mathbf{V}$. The Matlab syntax for obtaining these entities is

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

where eigenvectors are assembled as column vectors in the matrix \mathbf{V} and eigenvalues are placed on a diagonal of matrix \mathbf{D} with zeros elsewhere. When there is only a single variable assignment, the *eig* function returns a column vector of the eigenvalues. As an example, let

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 5 & 4 \end{pmatrix}$$

Then the command line instruction $d = \text{eig}(A)$ returns

```
d =
    -1
     6
```

whereas the command line instruction $[V,D] = \text{eig}(A)$ returns

```
V =

   -0.7071   -0.3714
    0.7071   -0.9285
```

```
D =
```

```
-1  0
 0  6
```

Roots of polynomials and zeros of functions

Another computational procedure that is commonly used in Matlab is that of finding the roots of polynomials, or in the case of non-polynomial functions, argument values which produce functional values equal to zero. For example the trivial polynomial $x^2 - 4$ has zeros at $x = -2$ and at $x = +2$, which are also called roots since the function is a polynomial. Similarly the function $\sin(2\pi x)$ has zeroes at $x = 0, x = \pm 0.5, x = \pm 1$, etc.

Although the roots of the trivial polynomial can be evaluated by inspection, it can also be done using Matlab if we form a row vector **p** with coefficients of powers of **x** in descending order: $p = [1 \ 0 \ -4]$. The command line instruction will produce a column vector with the roots:

```
>> roots = roots(p)
```

```
roots =

    2.0000
   -2.0000
```

The polynomials can be much more elaborate, and roots are not always real. For example consider the polynomial $3x^5 - 10x^4 + 2x^3 - 7x^2 + 4x - 8$, which can be represented in Matlab as

```
q = [ 3  -10  2  -7  4  -8 ]
```

The roots are easily found and displayed as a column vector with the command line instruction

```
>> qroots = roots(q)
```

```
qroots =

    3.3292
   -0.5335 + 0.8285i
   -0.5335 - 0.8285i
    0.5356 + 0.7335i
    0.5356 - 0.7335i
```

The inverse of the *roots* function is the *poly* function, which will produce a row vector of polynomial coefficients in descending power order that describe a polynomial function whose

roots are given by the input data vector. This is arbitrary up to a multiplicative factor, so the answer given is with scaling such that the coefficient of the highest power is set equal to 1. Thus, using the variables defined here,

```
>> qcoeffs = poly(qroots)
```

will regenerate the values in the vector \mathbf{q} , scaled by $\frac{1}{3}$ since the lead coefficient was 3.

Argument values that produce functional values equal to zero in non-polynomial functions, built-in or user-defined, can be obtained in Matlab by using the `fzero` command. This is a search procedure however, and a starting point or an interval must be specified. Whenever a zero functional value is detected the corresponding argument is returned as the answer with no further searching. Thus, for multiple solutions only the first found from the search start value is returned, in the form of a scalar. The target function name is enclosed in single quotes or is preceded by an `@` handle symbol, and a start point or interval has to be specified. Thus, using the trivial example of $\sin(x)$, where we know the zeros by inspection, a particular solution near π can be obtained with Matlab by giving the command line instruction

```
>> sinzero = fzero(@sin, [0.9*pi 1.1*pi])
```

```
sinzero =
```

```
3.1416
```

which calculates the closest answer as 3.1416, approximately π , as expected.

Poles, residues, and partial fraction expansion

The *residue* function in Matlab is a bit unusual in that it is its own inverse, with the computational direction determined by the number of input and output parameters. Its utility is to determine the residues, poles, and direct truncated whole polynomial obtained with the ratio of two standard polynomials when that ratio does not reduce to a third standard polynomial, or vice versa. When there are three output variables and two argument vectors, the arguments are taken to be the coefficients of a numerator polynomial and the coefficients of a denominator polynomial. The output vectors are the residue coefficients for the poles, the pole locations in the complex plane, and descending coefficients of the direct whole polynomial part of the expansion. As an example suppose we define numerator and denominator coefficient vectors by

```
>> pn = [1 -8 26 -50 48];
```

```
>> pd = [1 -9 26 -24];
```

Then the residue coefficient vector and location of the poles can be obtained with the *residue* function

```
>> [r,p,k] = residue(pn,pd)
```

```
r =
    4.0000
    3.0000
    2.0000
p =
    4.0000
    3.0000
    2.0000
k =
     1     1
```

This is equivalent to the expression

$$\frac{x^4 - 8x^3 + 26x^2 - 50x + 48}{x^3 - 9x^2 + 26x - 24} = 1x^1 + 1x^0 + 2\frac{1}{x-2} + 3\frac{1}{x-3} + 4\frac{1}{x-4}$$

Using the *residue* function in reverse, if we know the partial fraction expansion vectors, we can then reconstruct the polynomials whose ratio generated it. In this case the residue coefficient vector, the pole location vector, and the direct whole polynomial vector are given as arguments with the output being the corresponding numerator and denominator polynomials which form an equivalent ratio:

```
>> [qn,qd] = residue(r,p,k)
```

```
qn =
    1.0000   -8.0000   26.0000  -50.0000   48.0000
qd =
    1.0000   -9.0000   26.0000  -24.0000
```

The residue function can also be used when the polynomial ratio has poles with multiplicity or order greater than one. Typing `help residue` at a command line prompt will give details concerning those special circumstances.

Convolution and deconvolution

Many engineering applications involving input and output functions to and from a system use numerical convolution and deconvolution to describe each in terms of a system transfer function. These numerical processes correspond to a process of polynomial multiplication and its inverse. If an input function and a transfer function can be represented as vectors of lengths n and m then the output can be expressed as a convolution vector whose length is $(n + m - 1)$. This is similar to polynomial multiplication. Consider for example the polynomials $y = (ax + b)$ and $z = (cx + d)$, both of which have a coefficient vector length of 2, i.e., $[a \ b]$ and $[c \ d]$. The multiplication of these two polynomials yields $(acx^2 + (ad + bc)x + bd)$, which has a vector length of $(2+2-1) = 3$, i.e., $[(ac) \ (ad+bc) \ (bd)]$. In Matlab, such convolution can be accomplished by using the *conv* function on the two vectors. The command line instruction in Matlab to find the convolution would require numerical values be assigned to the coefficients a , b , c , and d . Using these input vectors

```
>> x = conv(y, z)
```

would generate the convolution vector x . The *deconv* function is merely the inverse. For example, if the output vector x and the transfer function vector z were known but the input vector y unknown, then the deconvolution

```
>> y = deconv(x, z)
```

would reconstruct the values of y as a deconvolution vector having the same sequence of elements that produced the known output vector.

Fast Fourier and Inverse Fourier Transforms

The process of convolution in one domain corresponds to a process of multiplication in that domain's Fourier transform space. This can simplify calculations of complicated convolution problems and is commonly used in engineering and signal processing. Once the calculations have been performed, the results can be recast in the original domain using an inverse Fourier transform. Matlab provides functions for doing such transforms.

The function *fft*, and its 2-D equivalent *fft2*, transform a vector or the columns of a matrix to a Fourier transform domain. Similarly, the inverse functions *ifft* and *ifft2*, transform a vector or the columns of a matrix to the inverse Fourier transform domain. As a trivial example consider the vector $f = [1 \ 2 \ 3]$ and its transformation to a vector g in Fourier transform space

```
>> g = fft(f)
```

```
g =
    6.0000    -1.5000 + 0.8660i    -1.5000 - 0.8660i
```

then reversion to a vector h in the original space which reconstitutes the original f :

```
h = ifft(g)
```

```
h =
```

```
    1    2    3
```

Numerical Differentiation and Integration

Matlab has a few functions that can be used in obtaining finite approximations for differentiation and for numerical integration by quadrature. Numerical differentiation methods are difficult because vector or matrix elements cannot always be expanded in resolution to approximate limiting ratio values. However, if a function or data vector can be fit well by an approximating polynomial, then the *polyder* function can be used to get a derivative of that approximate function. This function considers an n element argument vector to represent the n coefficients of a polynomial of order $(n-1)$ in descending order, i.e., the last element is the zero order coefficient. For example, suppose we have an independent variable vector $x = [1 2 3 4 5]$ and suppose we have a dependent functional vector $y = [0 -3 -4 -3 0]$. We can fit this data exactly to a fourth degree polynomial with 5 coefficients. The Matlab function *polyfit* is used and we choose to store the coefficients in a vector z :

```
>> z = polyfit(x,y,4)
```

```
z =
```

```
   -0.0000    0.0000    1.0000   -6.0000    5.0000
```

The coefficients of the derivative of the approximating (and in this case exact) polynomial representation are then obtained with the *polyder* function, which we will store in a vector designated w . Because the zero order polynomial term is a constant whose derivative is identically zero, that final element is omitted in the vector of derivative coefficients generated by

polyder. Note that this vector is a vector of coefficients and can be used to obtain numerical values for the “derivative” $\frac{dy}{dx}$ at any particular value of the independent variable x .

```
>> w = polyder(z)

w =

    -0.0000    0.0000    2.0000   -6.0000
```

The *diff* function in Matlab uses the limiting approximation of the ratio of function change to argument change to obtain a numerical derivative. This can produce very crude results, but can be somewhat useful if the independent variable values are regularly spaced with high resolution and the functional value changes at that resolution are relatively smooth. The syntax for this procedure, using the same example data as were used illustrating the *polyder* method but this time storing in a vector v , is:

```
>> v = diff(y) ./ diff(x)

v =

    -3    -1     1     3
```

With this method, the “derivative” values themselves are the elements of the vector, rather than coefficients. Note that, because of the difference method used, the vector of derivative values is one element shorter than the original data vectors.

Numerical integration can be done with analogous methods. As an illustration, let’s consider the integration of the same dependent function variable vector y from the numerical differentiation example above over the range of the independent vector x , also from that prior example. We can construct a formula-based integration vector using the *polyint* function on the polynomial approximation vector z , and we shall store it in a vector u :

```
>> u = polyint(z, 0)

u =

    -0.0000    0.0000    0.3333   -3.0000    5.0000     0
```

The second argument of the *polyint* function is a constant of integration and will appear as the final element in the quadrature vector if it is specified. With no second argument a value of zero

is assumed, so it was unnecessary here but was used anyway to illustrate the two-argument syntax. The elements computed in the vector u are coefficients of a polynomial, in descending order, from which polynomial numerical values can be computed for any particular independent variable value.

Another method for integration is numerical quadrature with Matlab's *quad* function, which uses an adaptive Simpson's Rule algorithm. The arguments for this function are a function name or definition in terms of an independent variable, the lower limit of integration, and the upper limit of integration. Thus, to use the same example, we would need to create a function m-file, say *yvalues.f*, to define the function element by element from data vectors. Alternatively if an explicit relationship can be given, such as the exact polynomial found in this case with *polyfit*, then the functional expression can be given in single quotes as the argument. This latter method will be shown here since creating function m-files are not covered until the following section. The range of x in the example is 1 to 5. The polynomial coefficient vector for the curve fitting had third and fourth order coefficients of zero so that the function value at each value of x is given by $x.^2 - 6*x + 5$. Therefore, if we want to store the value of the numerical integration in the variable t , we can use the command

```
>> t = quad('x.^2 - 6*x + 5', 1, 5)
```

```
t =
```

```
-10.6667
```

Compare that result with the result derived from the u vector, i.e.,

$$0.3333*(5^3 - 1^3) - 3*(5^2 - 1^2) + 5*(5^1 - 1^1) = -10.6708$$

Numerical solution of differential equations

Matlab has several solver functions for handling ordinary differential equations (ODEs), and a few functions for handling initial value problems (IVPs) and boundary value problems (BVPs) are also available. Additionally there is a specialized PDE Toolbox with functions for use with partial differential equations, but this Toolbox is not included in the Matlab licenses issued to ITS for its time sharing servers. However, there is one function included in the standard distribution of Matlab 7 for use in cases of PDEs that are parabolic or elliptic with two dependent variables. In general the syntax for using solvers is

```
>> [x,y] = solver(@func,[xinitial xfinal], y0, {options})
```

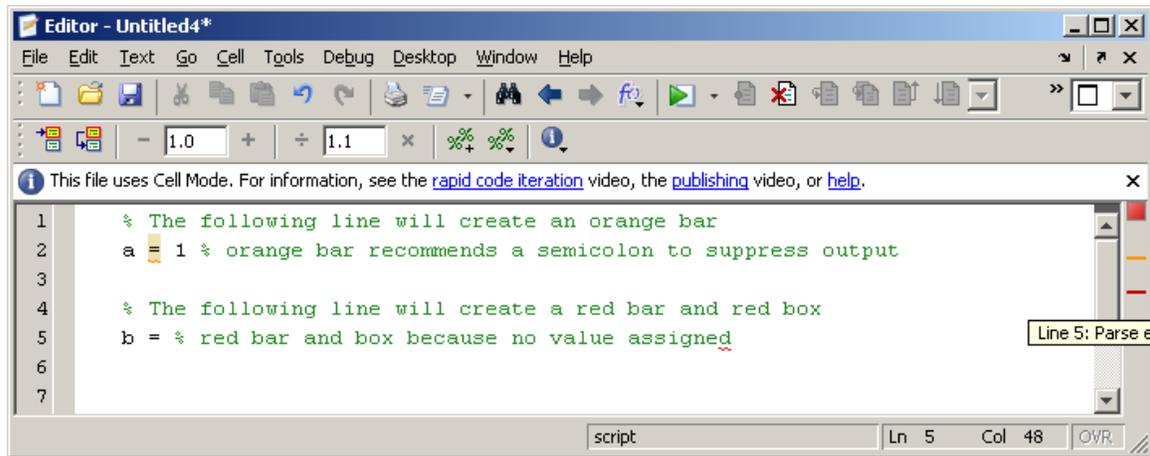
This assumes that a differential equation of order n is recast as a system of n first order differential equations of n variables and that it is described in a function m-file *func.m* where the independent variable vector x is to span the range from *xinitial* to *xfinal* and where dependent variable y to be solved has a known initial value vector y_0 . The independent variable vector x is uniformly spaced and is generated in response to solution iterations. Elements of the vector y are the solutions calculated for each value of the generated values of the independent variable x . The most commonly used ODE solvers for non-stiff systems are *ode45* and *ode23*, both of which use a Runge-Kutta type method. The solvers *ode15s* and *ode23s* are commonly used for stiff equations. For boundary value problems, the general release of Matlab 7 has a solver *bvp4c* and for parabolic and elliptic PDEs the solver *pdepe*.

Section 6: Programming

Matlab has its own programming language, also known by the name Matlab, which is structured very much like most programming languages. Because its origin involved interfacing with many of the original linear algebra subroutines written in FORTRAN, Matlab has roots there. But there is also much similarity with C due to later development of graphics functionality and the desktop GUI. The Matlab programming language is thus not particularly difficult or obscure, and the primary task in becoming adept at using it involves learning the various notations and syntax rules.

Using the Editor

Matlab has the functionality of incorporating user-created files external to the Command Window. These files need to be placed in a folder or directory defined in the pathway and should consist of a series of valid Matlab commands. These are thus equivalent to macros or subroutines in other programming languages, and are called *m-files*. They should have an extension of a period and a lower case *m* so that they can be recognized as Matlab text files. The Matlab 7 GUI has a text editor interface that allows users to create m-files within the Matlab application. As was mentioned in Section 3 above, it is launched from the navigation toolbar with *File > New -> Blank M-file* which opens up a new floating window ready for code text input. This can be used as a start for creating regular scripts or for function files. However, if a function file is to be created, the Editor Window can be opened with a convenient template with *File > New > Function M-file*. The editing window has its own toolbar with conventional Windows type pull down menus and button icons. Lines are numbered outside the text area for convenient referencing when debugging or inserting break, pause or flagging points. This editor tool is very convenient, with color coding and indenting features that aid in construction, but m-files can also be created as plain ASCII text in other text editors such as MS Word or Note Pad. A useful color coding outside the text area is a strip on the right margin of the window. There is a small box at the top which is colored green if the code contained within the text area has no syntactical errors and no suspicious structure. That box will be colored orange if the syntax is legitimate but suspicious structure makes review advisable. In this case there will also be orange bars along the strip indicating the structure that should be reviewed. Similarly, if there are any syntactical errors that would prevent the script from being executed, the box will be colored red and there will be red bars in the strip to indicate such errors. Placing a cursor over a bar will give a small diagnostic with the code line number and an associated suggestion or identification of error type.



Types and Structures of M-files

There are two main types of m-files which are very similar to each other: script files that act as command sequence macros with no parameter passing; and function files that also act as command sequence macros but which accept parameters by value and return a computed number, vector, or matrix assigned by the calling program. M-file scripts have no formal structure other than consisting of a sequence of valid Matlab command line input. After saving an m-file to a location in the Matlab path, which can be done from the *File* pull down menu of the Editor Window, any subsequent command line reference to it will result in the execution of its contents. This applies both to input from a Command Window prompt and to any call from within another m-file.

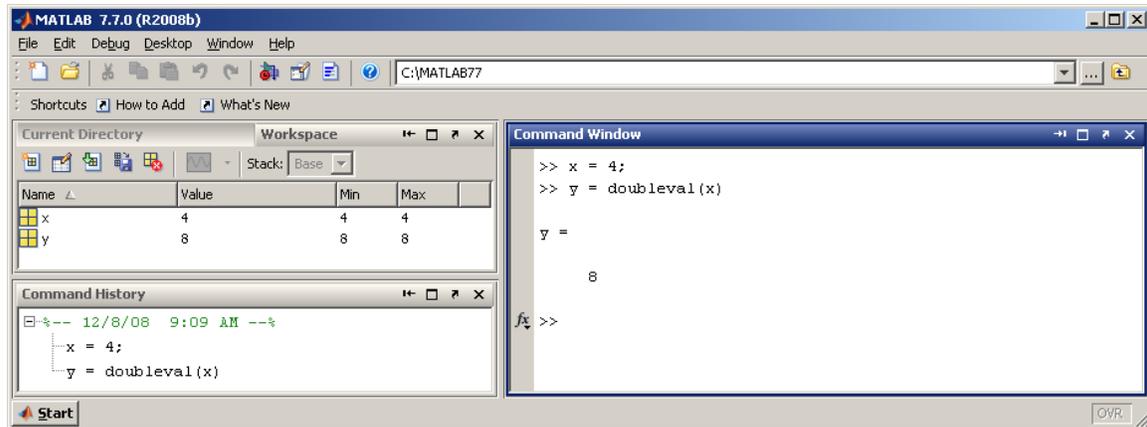
A function m-file is similar except that the first command line should have the form

```
function [ var1 var2 {...} ] = functionname(arg1, arg2, {...})
```

and saved under the same name as the function itself, i.e., *functionname.m* in the above example. The returned information should assign the computed functional values to **var1**, **var2**, etc., obtained with the function computational algorithm. A trivial example:

```
function f = doubleval(x)
f = 2*x;
```

which creates a function that doubles the value of the input as the return value. If you save this code with the name *doubleval.m*, then any subsequent call to *doubleval* from the Command Window prompt or from another script will return a value twice as large as the input argument value.



The Shortcut Utility

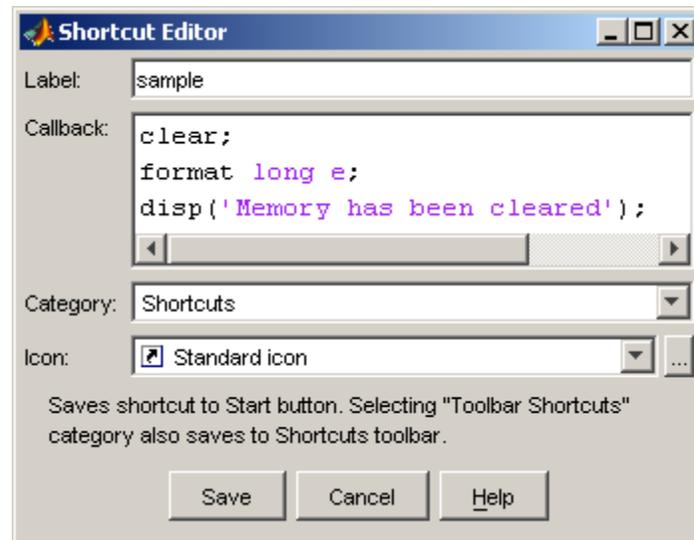
A Matlab shortcut is an easy way to run a group of Matlab commands that are used regularly, similar to a macro, but without the formality of having them incorporated as a script within the Matlab path. Shortcuts can be created, run, and organized from the *Start > Shortcuts* menu or from the desktop **Shortcuts** toolbar. From the *Start* button, select *Shortcuts > New Shortcut*. The Shortcut Editor dialog box appears. Create the shortcut by completing the dialog box using these steps:

1. Provide a shortcut name in the Label field to identify it
2. Put the group of command statements in the Callback field. They can be entered directly from keyboard typing, copied and pasted, or dragged via a desktop tool. If imported, edit the statements as needed. The field uses the Editor/Debugger preferences for key bindings, colors, and fonts, so the Callback field will have an appearance similar to an m-file. Note that if you copy the statements from the Command Window, the prompts at the beginning of a line appear in the shortcut, but Matlab removes these when the shortcut is saved.
3. Assign a category, which is like a directory, to be used for organizing shortcuts. To add the shortcut to the shortcuts toolbar, select the Toolbar Shortcuts category.
4. The default shortcuts icon is , but you may also select your own.
5. Click Save.

Suppose that the following group of commands is desired for a shortcut

```
clear;
format long e;
disp('Memory has been cleared');
```

The shortcut window would then be constructed something like the following



The new shortcut will be added to the Shortcuts entry in the Matlab *Start* button, and to the Shortcuts toolbar, if that category has been selected. Once created, a shortcut can then be run by selecting it from its category in the Matlab *Start* button.

An alternative to creating and running shortcuts via the *Start* button is to use the **Shortcuts** toolbar. To show or hide the shortcuts toolbar, toggle the *Desktop > Toolbars > Shortcuts* menu item. To create and run shortcuts via the desktop **Shortcuts** toolbar, go through a similar procedure:

1. Select statements from the Command History window, the Command Window, or an m-file.
2. Drag the selected group of commands to the desktop **Shortcuts** toolbar and the **Shortcut Editor** dialog box will appear. Here the Category field needs to be retained as *Toolbar Shortcuts* so that the shortcut will appear on the toolbar.
3. Choose a Label, select an Icon, and click the *Save* button. The shortcut icon and label will then appear on the toolbar.

Publishing to HTML

When an m-file script is written, Matlab allows you to use its cell features to publish them in a presentation format, such as HTML. Not only can the code itself be presented, but also commentary on the code and results from running the file can be displayed. This works only for script m-files, not for function m-files.

This is the overall process of publishing to HTML:

1. In the MATLAB Editor/Debugger, enable cell mode. Select *Cell > Enable Cell Mode*. If Cell Mode is already enabled, the menu item will be to toggle to *Disable Cell Mode*. Items in the **Cell** menu become selectable and the cell toolbar appears.
2. Define the boundaries of the cells in an m-file script using cell features. A *cell* is just a defined contiguous section of code. Many m-file scripts consist of multiple sections and

often it is necessary to focus on just one section at a time. M-file *cells* facilitate this process. A cell can be created by positioning the cursor just above the line where the cell is to begin and then selecting *Cell > Insert Cell Break*. A line consisting of two percent signs (%%) will be inserted, indicating the start of the cell. An identifier, called the *cell title*, can be appended by entering a blank space followed by a text string on the same line. All subsequent lines of code will belong to the cell until another cell start (line beginning with %%) occurs.

3. Select *File > Publish <filename>.m*. A floating window with the code displayed in HTML format will appear and the html source code itself will be saved in an *html* folder, generated if not already present, within the *Current Directory as <filename>.html*.

An elaborate example can be seen within the Matlab Help utility by selecting *Help > Product Help* and then navigating to *MATLAB > Desktop Tools and Development Environment > Publishing M-Files > Overview of Publishing M-Files > Example of a Published M-File*. This example is a square wave formed from a fourier expansion using sine waves.

Internal Documentation

Matlab m-files can be annotated with internal documentation that is ignored by the command interpreter. Any part of a command line, at a command window prompt or within m-file code, following a percent sign, i.e. %, that is not enclosed in quotes is considered as documentation and does not get processed. The % delimiter can be anywhere on a line. For example, compare the following command line instructions:

```
>> u = 5% + 1%
```

```
u =
```

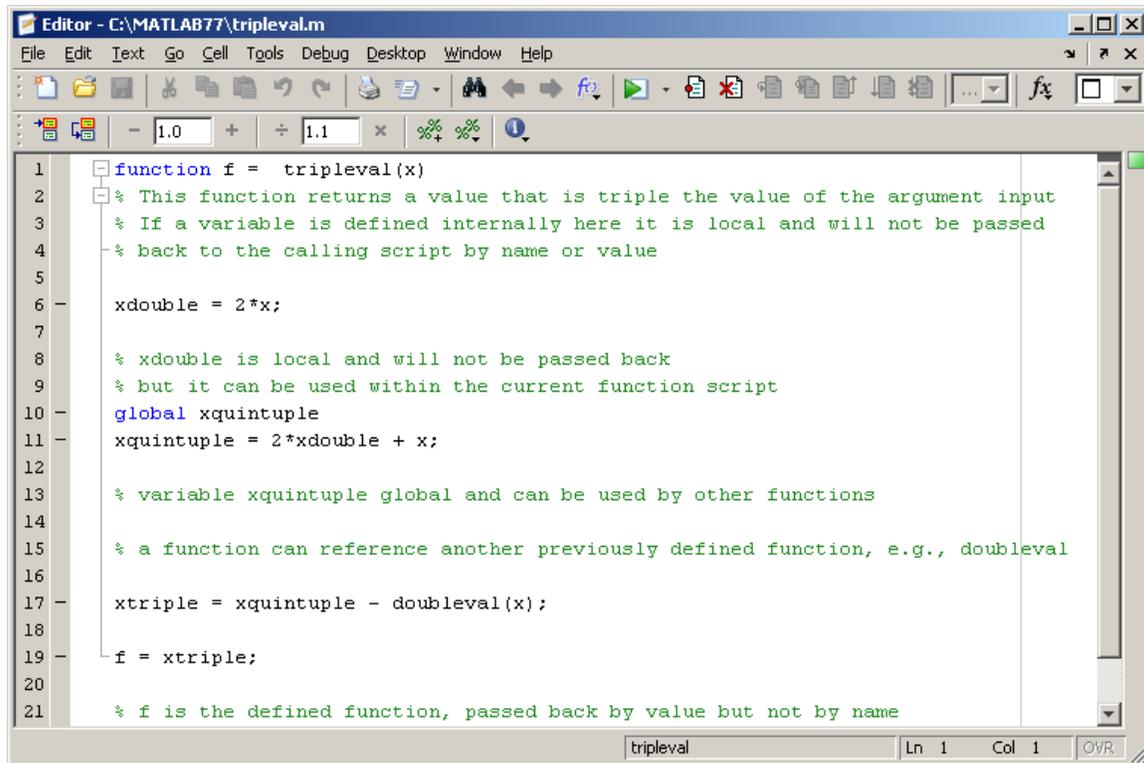
```
5
```

```
>> v = '5% + 1%'
```

```
v =
```

```
5% + 1%
```

For an illustration of internal documentation, here is the display of an annotated function m-file, *tripleval.m*, which returns a value three times that of the input:



```

1 function f = tripleval(x)
2 % This function returns a value that is triple the value of the argument input
3 % If a variable is defined internally here it is local and will not be passed
4 % back to the calling script by name or value
5
6 xdouble = 2*x;
7
8 % xdouble is local and will not be passed back
9 % but it can be used within the current function script
10 global xquintuple
11 xquintuple = 2*xdouble + x;
12
13 % variable xquintuple global and can be used by other functions
14
15 % a function can reference another previously defined function, e.g., doubleval
16
17 xtriple = xquintuple - doubleval(x);
18
19 f = xtriple;
20
21 % f is the defined function, passed back by value but not by name

```

Passing variables by name and value

If a script m-file is called from a Command Window prompt or by another script, then it is processed as a sequence of commands just as if each line had executed individually or had been incorporated as individual lines of the calling m-file. Thus, all variables created and values assigned to them become a part of the workspace and are recognized by any subsequent reference. On the other hand, function m-files have local variables by default. Only the values specified as function components are returned. However, a variable created within a function m-file can be declared global before assigning a value. If this is done then the next subsequent use of that variable name in another function m-file in which it has also been declared as global will start with the value already established. Every function that is invoked will have its own workspace, but global variables can be shared.

Also, function m-files may themselves contain functions, appended as subfunctions, which are constructed with the same structure as a regular function but with a `return` line indicating the end. Subfunctions within each level can also share global variables and can include their own subfunctions. As an illustration consider the following function m-file

```

1 function f = xseven(w)
2     % obtains a value seven times the input
3     f = xfour(w) + xthree(w);
4
5     % append subfunction
6     function f = xfour(x)
7         % obtains a value four times the input
8         global xone %defines a global variable
9         xone = x; % assigns a value to global variable
10        f = xone + xthree(x);
11        return
12
13    % append another subfunction
14    function f = xthree(y)
15        % obtains a value three times the input
16        global xone % allows use as defined in subfunction xfour
17        f = xone + xt看(y);
18        % append a subfunction of the subfunction
19        function f = xt看(z)
20            % obtains a value two times the input
21            f = 2*z;
22        return

```

The function `xseven` has two subfunctions, `xfour` and `xthree`. The subfunctions `xfour` and `xthree` share a global variable `xone`, the subfunction `xfour` calls the same level subfunction `xthree`, and the subfunction `xthree` has its own local subfunction `xt看`.

Using the function `xseven` is a very contorted way of multiplying a number by seven, but it illustrates function and subfunction architecture. In the Workspace Window notice that a variable generated within a function, e.g., `xone`, does not get carried over.

```

>> x = 2;
>> y = xseven(x)

y =

    14

>> z = xone
??? Undefined function or variable 'xone'.
fx >>

```

Name	Value	Min	Max
x	2	2	2
y	14	14	14

```

-- 12/8/08 9:46 AM --
x = 2;
y = xseven(x)
z = xone

```

Function evaluation and function handles

Once a Matlab function m-file has been constructed, named, and placed in the search path, it can be used in several ways. Let's consider the example that we have already constructed and named *xseven.m*. The most direct usage is to assign some variable the value of the function at a given argument, e.g.,

```
y(1) = xseven(5)
```

would assign a value of 35 to the variable *y*, whether interactively from a command line prompt or as a line of code in a script m-file. Alternatively, there is a command *eval* in Matlab which will evaluate a character string as a line of code. Thus, we get the same result if we use the command

```
y(2) = eval('xseven(5)')
```

However, this is not too efficient, as the entire Matlab interpreter is loaded for processing *eval*. For function evaluations the command *feval*, which is able to load only what is needed, can be used for greater efficiency. This command takes as arguments a function name, followed by that function's arguments. The function's arguments can be by value or by variable name if a value has already been assigned. As an example, suppose that we already have a variable *z* assigned the value 5 in the workspace. Then an equivalent assignment for *y* can be obtained with

```
y(3) = feval('xseven', z)
```

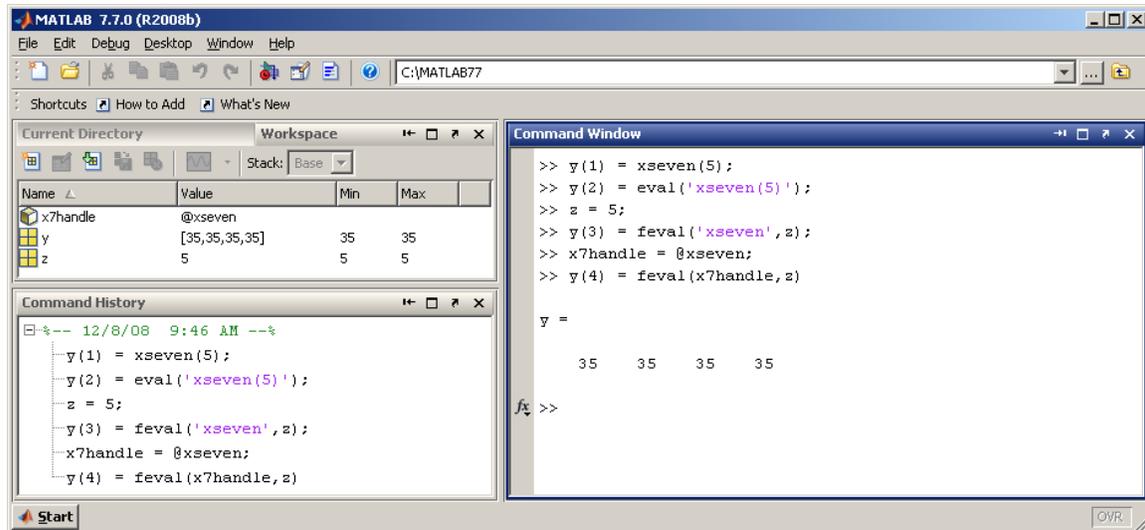
Another feature of Matlab is the *function handle* data type. This provides a means of speeding up execution by passing essential information about the function through a *function handle*, referenced with the syntax *@function_name*. The handle can be assigned to a variable name, which in turn can replace the character string in the arguments of *feval*. If we assign *@xseven* to a function handle variable, say

```
x7handle = @xseven
```

then yet another, faster method of assigning the variable *y* above would be

```
y(4) = feval(x7handle, z)
```

The results of all these methods, which have been stored in the vector *y*, are identical and have the value 35, i.e., the argument 5 operated on by the function *xseven* -- which in a roundabout way calculates a multiplication by seven.



Function recursion

A function value can also be passed back to itself through recursion. A simple example is a function that computes a triangular number (sum of all integers less than or equal to the argument). In this function script notice that the function *triangular* calls itself:

```
function f = triangular(n)

% finds the sum of all integers from 1 through n

t = n;
if n > 1
    t = t + triangular(n-1);
end;
f = t;
```

Flow control

In Matlab, flow control procedures, as well as syntax, are very similar to procedures in many other programming environments. The *for* and *while* commands are constructs for executing a block of commands multiple times until a specified incrementing index value is reached or a logical termination condition is met. The finish of a repeating block of commands must be specified with an *end* statement. The general format is thus

```
for {index} = {firstvalue}:{lastvalue}
    {command1; command2; ...}
end;
```

```

while {logical condition}
{command1; command2; ...}
end;

```

Just as in other programming environments, the *for* and *while* loops can have layers of nesting. As an example let's look at a script for finding the number of seconds in a week, with emphasis on illustrating loops and nesting rather than efficiency:

```

% find the number of seconds in a week
total = 0; % start with none yet counted
for m = 1:7 % consider every day in a week
    for n = 1:24 % consider every hour in a day
        minval = 1; % start minutes counter
        while (minval <= 60)
            secval = 1; % start seconds counter
            while (secval <= 60)
                total = total + 1; % increment total
                secval = secval + 1; % increment secs
            end;
            minval = minval + 1; % increment mins
        end;
    end;
end;
weeksecond = total

```

Matlab also has the capability of logical contingency flow control with the *if* and *switch* commands. The general construction of an *if* command is

```

if {logical condition}
{command1; command2; ...}
elseif {logical condition}
{command1; command2; ...}
.
.
.
elseif {logical condition}
{command1; command2; ...}
else
{command1; command2; ...}
end

```

The blocks of commands are executed if the specified logical condition is true and an optional block of commands following *else* is executed if none of the preceding logical conditions are true. The *elseif* segment can be expanded to include multiple logical conditions and command blocks, but only the block following the first logically true condition will be executed.

The *switch* command has a similar function but flow is controlled by the current value of a specified expression. The general structure is

```
switch {expression}
    case {value1}
        {command1; command2; ...}
    case {value2}
        {command1; command2; ...}
        .
        .
        .
    otherwise
        {command1; command2; ...}
end
```

The command block executed is that following the *case* value which corresponds to the value of the specified expression. An optional *otherwise* command block can be included to cover expression values which are not enumerated in the cases.

As an example of using *if* and *switch* flow control, consider a script that determines the number of non-primes, primes, and twin primes (incrementing by 2 gives another prime) in a particular range of integers, for example between 1001 and 2000:

```
% find prime status of integers in a range
nontwins = 0; % start with none yet counted
twins = 0; % start with none yet counted
nonprimes = 0; % start with none yet counted
for n = 1001:2000 % test all numbers in the range
    if isprime(n)
        switch isprime(n+2)
            case 1
                twins = twins + 1; % increment
            otherwise
                nontwins = nontwins + 1; % increment
        end
    else
        nonprimes = nonprimes + 1; % increment
    end
end
```

```

        end;
    else
        nonprimes = nonprimes + 1;    % increment
    end;
end;
notprime = nonprimes
prime = nontwins + twins
twinprime = twins

```

String evaluation and manipulation

Although character strings are stored as vectors of ASCII character code integers within Matlab, they can be evaluated and manipulated in certain circumstances. If a string contains text that represents a valid Matlab command or expression, then the *eval* command can process the string as if it were command line input. Similarly, if a string consists of the name of a defined or built-in function, then the *feval* command specifying the function name and its input arguments can be used to generate a value. As an illustration consider the following script that computes the integer within a given range that has the largest positive difference between its sine and cosine:

```

% maximum difference in sin vs cos in integer range
maxnum = 1; % initialize with first argument
maxdiff(1) = sin(1) - cos(1); % initialize difference
x = 'sin(n) - cos(n)'; % make a string function
for n = 2:1000          % select integer range
    y(n) = eval(x);    % evaluate the string
    maxdiff(n) = feval('max',y(1:n)); % update maximum
    if (y(n)>maxdiff(n-1))
        maxnum = n;    % update the argument if needed
    end;
end;
maxnum          % integer maximizing (sin - cos)
maxdiff = feval('max',y) % difference maximum

```

In this particular example the result will be the integer that most closely approximates a value of $\frac{3}{4}\pi + 2k\pi$, which, for the selected range, turns out to be 882, or about $\frac{3}{4}\pi + 280\pi$. There are also special built-in functions for evaluating and manipulating strings which are the text equivalent of integers, numbers, or even matrices. All these require that any arguments which are not decimal-based integers or numbers be in a character string format (enclosed within single quotes). This class of functions includes

`str2num` and `num2str` [converting from strings to numbers and vice versa]

int2str	[converting integers to character strings]
hex2dec and dec2hex	[converting hexadecimal strings to integers and vice versa]
bin2dec and dec2bin	[converting binary strings to integers and vice versa]
base2dec and dec2base	[converting non-decimal strings to integers and vice versa]

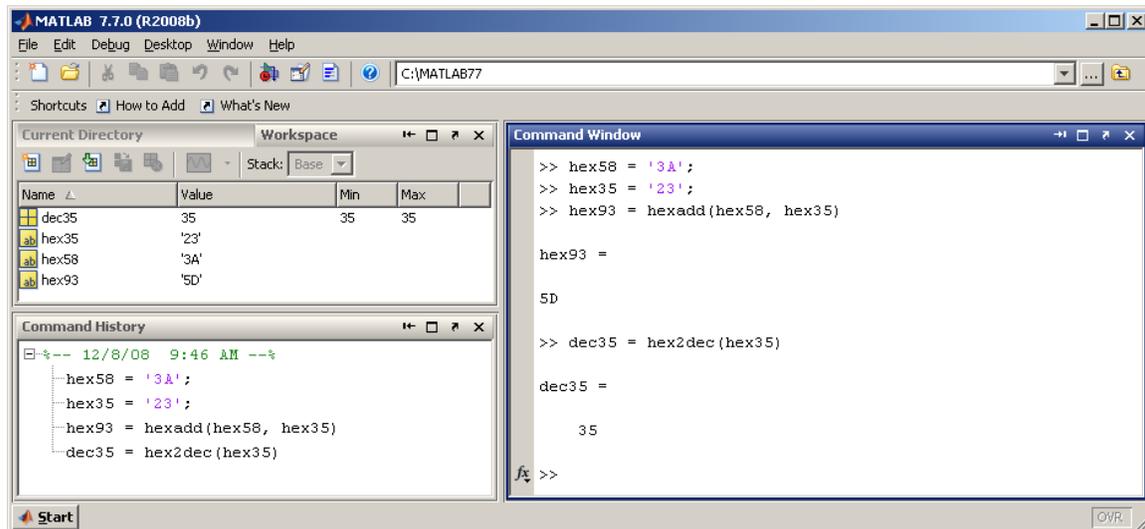
The general syntax used with these functions is

```
stringvalue = number2string(number)
numbervalue = string2number(string)
```

As an example of the use of these commands we can construct a function for adding hexadecimal numbers, a feature which is not a built-in Matlab utility. Here is one way to do this, where the input is two hexadecimal character strings:

```
function f = hexadd(hex1,hex2)
% Add 2 hexadecimals represented by text strings
newhex = dec2hex(hex2dec(hex1) + hex2dec(hex2));
f = newhex;
```

Note that the output is also a hexadecimal character string:



Keyboard input

Matlab allows the programmer to include interactive input from the keyboard with the *input* command. The command incorporates a prompt in the Command Window so that the user knows that an input is waiting. A variable is assigned the value of the user's input according to the syntax of the *input* command's argument. The syntax for the *input* command argument is a character string for prompting the user, enclosed in single quotes, followed by a lower case *s* in single quotes if the input is to be considered a string rather than a number or matrix. This is a common way to allow a user to specify a starting guess in an optimization script or to specify a

cutoff for an infinite series computation or iterative process. As an example, let's construct a script to approximate π recursively from an iterative algorithm in which the user can set a limit on the number of iterations:

```
% script to illustrate user keyboard input
disp('Iterative approximation to pi');
reply = input('Limit to iterations? [y/n]','s');
if (reply == 'y')
    itmax = input('Maximum iterations?');
else
    itmax = 100;
end;
iter = 0; % initialize iteration number
res = 0.0001; % resolution for detecting change
prevpi = 2; % initialize a previous value
piapprox = 2 + 2/sqrt(2); % initial guess
while((abs(piapprox - prevpi) >= res)& (iter < itmax))
    prevpi = piapprox; % reset the previous value
    piapprox = 2 + 2/sqrt(piapprox); % new iteration
    iter = iter + 1; %increment iteration number
end;
iterations = iter
pi = pi
picalc = piapprox
difference = pi - piapprox
```

If you test out this script you will find that a consistent value is reached after six iterations even if more than that number is specified as the maximum. Also you will also notice that the iterative convergence of the approximation is to a number very slightly less than the analytic value of π .

Multidimensional arrays and indexing

The fundamental unit in Matlab is a two dimensional matrix whose elements can be referenced by row position n and column position m . These element locations are specified in parentheses following the matrix name, with the row position given first. Thus, for example, $A(2,3)$ refers to the element of matrix A which occupies the second row and third column. However, the contents of the matrix are stored within Matlab memory, regardless of how they may be displayed or referenced, as a single column matrix, i.e., a column vector, with sequential values column by column. Therefore the n^{th} element of the m^{th} column in an $N \times M$ matrix A can also be referenced by a single index, $A(p)$ where

$$p = (m-1) * N + n$$

so that the last element in the first column has index $p = N$, the first element in the second column has index $p = N+1$, etc.

Matrices can be concatenated by defining a matrix of matrices, or by using the *cat* command, but of course the result of every concatenation must be a rectangular array. As an example, suppose we have four simple rectangular matrices

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 4 & 5 & 6 \\ 3 & 2 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad D = \begin{pmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{pmatrix}$$

These can be concatenated as

$E = [A \ B]$ giving a matrix of 2 rows and 6 columns

$F = [C; D]$ giving a matrix of 6 rows and 2 columns

As with simple matrices, row components can be separated with commas in lieu of blank spaces and column components can be separated with line feeds in lieu of semi-colons.

The *cat* command is equivalent, using a direction argument, *1* or *2*, to specify concatenation in a column or in a row respectively. Thus, we would get the same result as above by constructing the matrices *E* and *F* as

$E = \text{cat}(2, A, B)$

$F = \text{cat}(1, C, D)$

If an element position outside a matrix is **referenced**, there will be an error message displayed, for example when trying to display $A(3,2)$ from above. However, if an element outside the current range is **assigned** a value, Matlab will accept the assignment and fill in all other element positions necessary to form an expanded rectangular matrix with zeros. We cannot directly concatenate all four of the example matrices above as

$G = [A \ B; \ C \ D]$

because *A* and *C* have differing numbers of columns and rows, as do *B* and *D*. However, if we assign a value (0 for example) to $A(3,3)$, $B(3,3)$, $C(3,3)$, and $D(3,3)$, then all four of the component matrices will be restructured as square matrices of dimension 3, padded with zeroes where not explicitly defined., and the assignment for *G* above can be processed.

Arrays of dimension greater than two can be useful, for specifying three dimensional coordinates, assembling functional values determined by three or more independent variables, and so forth. Matlab was developed in the context of matrix algebra where most functions, matrix multiplication and so forth, are defined on the basis of two dimensional rectangular arrays, so two dimensional arrays are the primary objects. Nevertheless, Matlab can accommodate higher dimensional array notation. The positional index numbers can be expanded to three, four, or more. However, display is still limited to two dimensional “cross sections” or slices, and the actual storage is again as a single column vector with an internal index for a multidimensional array element assigned sequentially by row position, then by column position, then by third index, and so forth. Higher dimensions are not really amenable to visual conceptualization, but think of a four dimensional array being like a set of books along a shelf, each having the same number of pages and with each page having a same size rectangular matrix. If there are five books each with six pages of matrices with seven columns and eight rows, then a particular element in the array Z with all the aggregate matrix elements could be referenced in multidimensional form as $Z(\text{row}, \text{column}, \text{page}, \text{book})$, e.g. $Z(1,2,3,4)$ for the first row and second column of the matrix on page 3 of book 4. This would correspond to an alternative single column internal storage reference. Before book 4 there are three books having six pages with 7×8 matrices, thus containing $3*6*7*8 = 1008$ total elements. Then in book 4, before page 3, there are two pages with 7×8 matrices, thus containing $2*7*8 = 112$ total elements. Finally, there is one column before the second, in the matrix on page 3 of book 4, thus containing $1*8 = 8$ total elements. Hence the internal indexing would assign $1008 + 112 + 8 = 1128$ positions before getting to $Z(1,2,3,4)$. Therefore, the programmer could reference $Z(1,2,3,4)$ alternatively by $Z(1129)$ in this example.

Debugging

Matlab has extensive features to assist in debugging program errors, both compile time errors that primarily involve user mistakes in syntax and run time errors which either interrupt execution or produce unexpected or erroneous results. Help for both command line input and m-file source code debugging is available.

The Matlab interpreter will display error messages in the Command Window when it encounters some operational instruction that it cannot understand. Typically it will give some feedback about the problem, including a line number if the error is in an m-file. For example, suppose at the command line we type

```
>> x(0) = (1^0)/factorial(0)
```

The interpreter will respond with

??? Subscript indices must either be real positive integers or logicals.

because array indices are restricted to be positive integers. This could also happen if the problem occurred within an m-file script. Let an initial version of a script *ebase.m*, which is supposed to find the value of the natural logarithm base be constructed as

```
%script to find the natural logarithm base e
evaluate = 0;           % initialize the value
prev = -1;             % initialize a previous value
n = 0;                 % initialize index
while (evaluate ~= prev) % iterate if still converging
    nterm(n) = (1^n)/factorial(n); % next term
    prev = evaluate; % update previous value
    evaluate = prev + nterm(n); % update value
    n = n + 1; % increment index
end;
evaluate % show final value
```

There will be the same syntax error concerning a non-positive integer index shown, but this time there will also be an additional diagnostic comment of the form

```
Error in ==> ebase at 6
    nterm(n) = (1^n)/factorial(n); % next term
```

specifying the line in the function m-file where the syntax error occurred.

We, as the user, can then edit the script to make an adjustment. If we do not care about keeping the values of the individual terms in the series we can merely convert *nterm* to a transiently valued scalar by removing the index, or we could fix the problem by replacing *nterm(n)* with *nterm(n+1)* in the script.

Run time errors are usually more difficult to spot and correct, and there are sometimes several possibilities as to the origin. Operations that would terminate execution in other programming environments do not always do so in Matlab. For example value assignment involving division of a non-zero number by zero will give an assignment of *Inf* (infinity) without stopping program execution. Likewise, assignment involving division of zero by zero produces an assignment of *NaN* (not a number), without halting execution. Errors in logic from user programming can lead to false calculations without giving any diagnostic warning. In the *ebase.m* script for obtaining a value for the natural logarithm base, suppose the series term to be added had been written as

```
nterm(n+1) = (1^n)/gamma(n);
```

where we have made the logic mistake of using $\text{gamma}(n)$ rather than $\text{factorial}(n)$. Because the gamma function for a positive integer is actually the factorial function of that same integer argument less one, i.e.,

```
gamma(n) = factorial(n-1);
```

we have a logic error in user supplied instruction. The function $\text{gamma}(0)$ will evaluate to Inf so that the first series term computes to zero with the result that the base value for natural logarithms remains at the initialization value without any subsequent iteration being performed. This error in logic can be fixed by the user via changing $\text{gamma}(n)$ to $\text{gamma}(n+1)$, or its equivalent, $\text{factorial}(n)$.

Other common logic errors that will not cause termination of program execution are things such as infinite loops, mistakes in operator precedence or grouping, unintentionally overwriting a variable value from name duplication, and systematic propagation of round-off or truncation error. There are several methods to use in tracking down such errors. One straightforward way is to delete the semicolons from the end of assignment instruction lines so that variable values are displayed in the Command Window during program execution. Another method is to have some user-specified diagnostic string displayed at certain designated points in the coding or whenever some logical condition (such as the value of a variable exceeding a certain number) is satisfied. This can be done with the *disp* command, which displays the value of its argument in the Command Window. The programmer can also insert instances of the *keyboard* command in the code, which will cause execution to halt and give a *K>>* prompt in the Command Window. At this point instructions such as changing or testing a variable value can be given from the keyboard. Upon a keyboard command of *return*, the program execution continues. An example code to illustrate these techniques is the following for getting the value of a Riemann zeta function ζ from its infinite series definition, i.e., the sum of all reciprocals of positive integers raised to the power given as the argument. We assume it has been saved in a file *zeta.m*. This script should find an asymptotic value for zeta with an argument supplied within the code, in this instance 2. The theoretical asymptotic value for $\zeta(2)$ is $\frac{\pi^2}{6}$, but we don't know the rate of convergence, or the processor time needed. To periodically check the progress and confirm that the program is not stalled or executing an infinite loop, a line has been included within the *while* loop to display the series term number after every hundred thousand iterations. This is followed by a keyboard command to allow us to check the current sum from a command line prompt and terminate the loop if we think a sufficiently accurate value has been reached. The script can be run in debug mode with *Debug -> Save and Run* from the Editor toolbar

The screenshot shows the MATLAB Editor window with the following code:

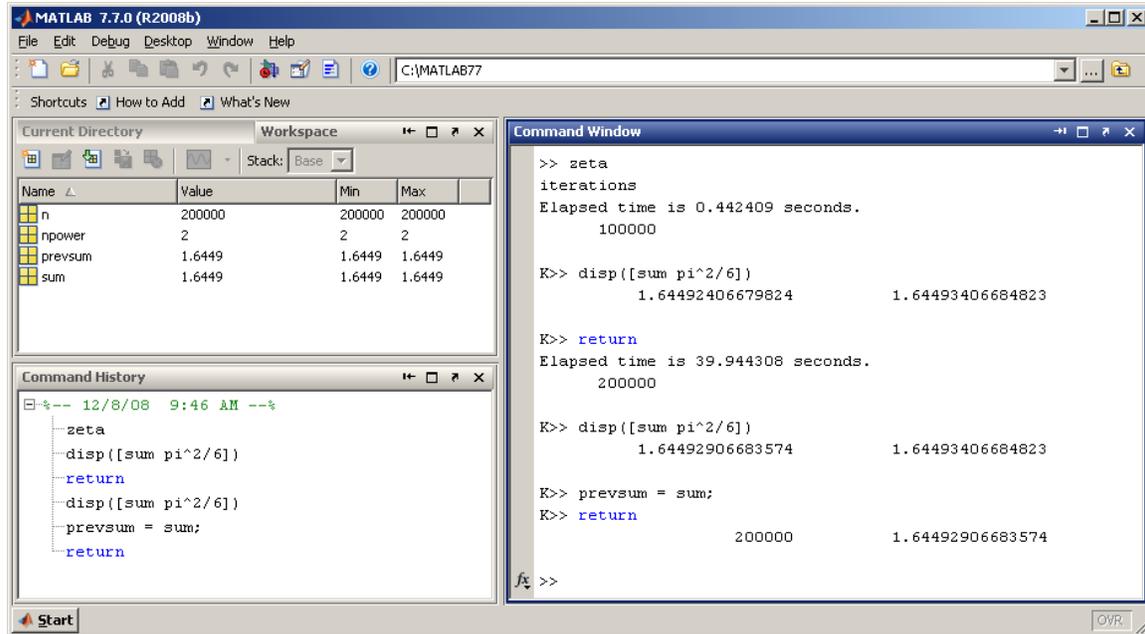
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14 % asymptotic evaluation of the Riemann zeta function
15 sum = 0; % initialize the asymptotic sum
16 prevsum = -1; % initialize a previous sum
17 npower = 2; % specify the zeta function parameter value
18 n = 1; % initialize the series term counter
19 disp('iterations')
20 tic; % start timer
21 while (sum ~= prevsum) % test for convergence
22     prevsum = sum; % reset the comparison value
23     sum = prevsum + n^(-npower); % add series term value
24     n = n + 1;
25     % include a monitoring/debugging instruction
26     if (n == 1e5*floor(n/1e5)) % identify each hundred thousandth term
27         toc % show cumulative calculation time
28         disp(n) % display the series term number
29         keyboard % start keyboard prompt input
30     end;

```

The debug menu is open, showing options such as Step (F10), Step In (F11), Step Out (Shift+F11), Run zeta.m (F5), and Exit Debug Mode (Shift+F5).

From the keyboard let's check the sum a couple of times at hundred thousandths term intervals to confirm that we are really in a long process that indeed is converging to an expected value rather than being stuck in some unending computational process, then terminate the iteration when we reach an acceptable level of accuracy.



Using Matlab with External Code

When developing programs to be run in Matlab or in another environment in which Matlab files can be called, there may be times when various interface tools will be needed. A wide variety of such tools exist, but naturally they will be platform-dependent or source code language-dependent. Thus, there are not really any generic methods. A developer will have to customize the tools for the particular circumstances.

Exchanging and viewing text information

Matlab programs can communicate with and use external code in many circumstances. The simplest examples are transferring data to and from external files in the form of ASCII numerical matrices with the *save* and *load* commands. Communication can also be on the level of examining and transferring ASCII text source code. For example, consider a simple piece of ASCII text

```

function f = conevolume(r,h)
% volume of a cone with base radius r and height h
f = pi*(r^2)*h/3;

```

saved in three formats in a folder within the Matlab path: *conevolume.doc* (MS Word document), *conevolume.txt* (plain ASCII text document), and *conevolume.m* (Matlab m-file). All can be opened from the Matlab Editor, but the MS Word document will contain binary code and not be useable. Similarly, all these files can be opened and viewed as text in MS Word. The MS Word

text file can be used by Matlab if it is opened in an editor and saved as ASCII or m-file with a “.m” extension.

Compiling and calling external files from Matlab

Matlab can compile external FORTRAN and C code into MEX (Matlab EXternal) files using the *mex* command. The executable is stored in a file with an extension that depends on the external platform. Examples are *filename.dll* for a Windows platform and *filename.mexsol* for a Sun Solaris OS platform. When compiled these external code files can be utilized just as if they were built-in Matlab scripts or functions. As an illustration, let’s consider some code written in C that accomplishes the same task as the *conevolume.m* code above, compiled on a Windows platform. In order to avoid the confusion of having programs of the same name with various extensions in the same folder or directory, we will name the source code for this C version *cvolume.c* to distinguish it from *conevolume.m*

```

/*=====
 * CVOLUME.C volume of cone from base radius and height
 *   Calling syntax:      v = cvolume(r,h)
 *=====*/

#include <math.h>
#include <mex.h>

/* Input Arguments */

#define   R_IN prhs[0]
#define   H_IN prhs[1]

/* Output Arguments */

#define   V_OUT      plhs[0]

#define PI 3.14159265

static void cvolume(
    double v[],
    double r[],
    double h[]
)
{

```

```

    v[0] = PI*r[0]*r[0]*h[0]/3;
    return;
}

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    double *v;
    double *r,*h;

    V_OUT = mxCreateDoubleMatrix(1,1, mxREAL);

    /* Assign pointers to the various parameters */

    v = mxGetPr(V_OUT);
    r = mxGetPr(R_IN);
    h = mxGetPr(H_IN);

    /* Do the actual computation in a subroutine */
    cvolume(v,r,h);
    return;
}

```

You can see that the C code is quite a bit longer and more complex, so normally you would just use the Matlab programming language to do such a simple calculation. However, there may be occasions when execution time can be improved by using compiled code of another language. This situation can occur, for example, when the Matlab code involves extensive looping or iterating. Also, some other languages may have libraries available for specialized purposes or functions.

To compile this code, you will need to setup for MEX files if you have not done so already. At the Command Window Matlab prompt, type

```
>> mex -setup
```

and you will subsequently be prompted for information about your computer's operating system and the compiler that you want to use. UNIX systems usually have FORTRAN and C compilers such as *f77* or *gcc* installed, but Windows platforms may not come configured with any. Thus, Matlab has a bundled compiler, *Lcc*, which is used as the default C compiler if no other is

specified. Unfortunately the current release of Matlab does not have a bundled FORTRAN compiler. After finishing the setup, you should be ready to compile programs, at least ones written in C. The syntax for compiling is to give the file name and extension as the target for the *mex* command, e.g.

```
>> mex cvolume.c
```

to compile the C code given in the illustrative example. If you do this on a Windows platform, you will generate an executable *cvolume.dll*. If you compile it on a Solaris platform, you will generate an executable *cvolume.mexsol*, etc.

Once you have done the compilation, you can use the command *cvolume* at a Matlab prompt and get the same answer as you get using the command *conevolume* whose source is a Matlab language m-file. The volume of a cone which has a base radius 1 and a perpendicular height 1 is computed by these two codes as

```
>> volume_Matlab = conevolume(1,1)
```

```
volume_Matlab =
```

```
1.0472
```

```
>> volume_C = cvolume(1,1)
```

```
volume_C =
```

```
1.0472
```

The current distribution of Matlab has a more elaborate example based on a three body orbital dynamics problem, which consists of an m-file *yprime.m* and an equivalent C code version *yprime.c* that can be compiled as a MEX-file. Those example files are located in the directory or folder $\$MATLAB > extern > examples > mex$ where $\$MATLAB$ is the Matlab installation directory or folder.

Calling Matlab objects from external programs

Matlab has an engine with functions that allow it to act as a callable external function or subroutine library in FORTRAN and C programs. The functions open the Matlab engine, close the engine, get and send matrix arrays, and evaluate Matlab strings. There is extensive information about these in the built-in Matlab help utility, with examples of FORTRAN and C programs that call the Matlab engine. External FORTRAN and C programs can also call binary

Matlab files such as those created by default with the *save* command. The *save <filename>* command with no format flag will save the current workspace information in a binary file format (MAT) with a default name *filename.mat*, or the default *matlab.mat* if no file name is supplied. MAT file information can be accessed by external programs with commands similar to those for the Matlab engine. There is extensive information about the commands for exchanging data with external programs using MAT files in the built-in Matlab help utility. For example, from the navigation bar see *Help > Product Help > MATLAB > External Interfaces > Importing and Exporting Data > Examples of MAT-Files > Reading a MAT-File in C*.

Using Java Classes in Matlab

The desktop interface that is used for Matlab was developed and written in Java. As a result, Matlab has an integrated Java Virtual Machine (JVM) and Java classes can be used directly in Matlab at the command line or within function m-files. Before usage, there must be a file with the name *classpath.txt* in the Matlab search path. Matlab has a default file located in the folder or directory *\$MATLAB > toolbox -> local*. This can be copied to a directory in the Matlab path and edited if the programmer has created classes other than those in the default list.

A class or a package of classes can be imported into the workspace with the *import* command. If this has not been done for a particular class, that class has to be referenced by its complete name. As an illustration, consider a simple example where the default *classpath.txt* file has already been copied to the Current Directory and the Java *awt* class package has been imported but the Java *lang* class package has not. In this case we can use a class name from the imported package directly on the command line but must give the complete full name for a class in the package which has not been imported

```

MATLAB 7.7.0 (R2008b)
File Edit Debug Desktop Window Help
Current Directory: C:\MATLAB77

Workspace
Name Value Min Max
promo <1x1 java.lang.String>
twolengths <2x1 cell>

Command History
12/8/08 9:46 AM --%
import java.awt.*
twolengths = fieldnames(Dimension)
promo = String('Matlab Rocks!')
promo = java.lang.String('Matlab Rocks!')

Command Window
>> import java.awt.*
>> twolengths = fieldnames(Dimension)

twolengths =

    'width'
    'height'

>> promo = String('Matlab Rocks!')
??? Undefined function or method 'String' for input arguments of type 'char'.

>> promo = java.lang.String('Matlab Rocks!')

promo =

Matlab Rocks!

fx >>

```

The *Dimension* class is recognized because its package has been imported, but the *String* class is not recognized as a Java class because its package has not been imported. However, if the complete name of the *String* class, i.e., *java.lang.String*, is used then the Matlab interpreter recognizes it as a Java class.