C.
Matlab Tutorial

C.1 Introduction to Matlab

This is a brief tutorial on Matlab. The purpose is not to teach you to program, but to indicate features that Matlab has in common with other languages you may already know – C, Java, Python – and also to highlight those special features that make Matlab especially suited to the processing of arrays, which is at the heart of DSP. If you know any programming, Matlab is easy to pick up on your own and there are many resources available to help you.

- Matlab's online `help` and `doc` functions are always there for you. Typing '`doc matlab`' at the command prompt will get you to the top level of the Matlab's excellent help system.
- An extensive selection of tutorials is available online, at the MathWorks website and on YouTube.

C.1.1 What is Matlab?

Matlab is a program and development environment for scientific applications. It is a calculator, a programming language, an integrated development environment, and a sophisticated system with hundreds of specialized functions for analyzing and displaying data. It is well suited to DSP operations and is used extensively in academe and industry to develop and prototype sophisticated algorithms and applications. There is extensive built-in documentation as well as a large web-based user community to help with solutions to problems.

Matlab's roots are in the 1970's. It was initially developed as an interactive front-end to the LINPACK and EISPACK libraries for matrix and eigenvalue computations that were written in FORTRAN (hence the name: Matlab = Matrix Laboratory). Matlab's core strengths remain its speedy and robust algorithms for matrix manipulation and numerical analysis, many of which are now based on the open-source libraries designed for modern computer architectures, such as LAPACK and FFTW. However, some of Matlab's core deficiencies also derive from its FORTRAN roots. Specifically, addressing of arrays in Matlab still follows the ones-based indexing convention of FORTRAN, meaning the first element of an array, `s`, is `s(1)`, not `s(0)`, as it would be in C, C++, Java or other modern languages with zero-based indexing. This can be a source of much programming sadness, and is therefore something all Matlab programmers need to keep in mind as they code. Other issues that frequently raise the ire of Matlab users are its awkward handling of strings, multidimensional matrices and matrices with mixed data types (e.g. `cell`). MathWorks, the makers of Matlab, updates the software biannually. This introduces new features, but also frequently redefines or breaks old features. However, despite its flaws, Matlab continues to dominate the market and is well worth your effort to learn.

C.1.1 What is Matlab *not*?

Matlab is not cheap. It a relatively expensive, proprietary, commercial product that comprises the core Matlab product plus a number of application-specific toolboxes, such the Signal Processing Toolbox and the Image Processing Toolbox. There is a reasonably priced Student Edition of Matlab available for those who qualify. The basic price gets you the core Matlab product, and you can add a number of the most important toolboxes for a nominal cost.

There also exist a number of free, open-source Matlab-like clones; chief among which are Scilab and GNU Octave. These have a syntax similar to that of Matlab and can perform many of Matlab's basic functions. Octave, in particular, contains many functions that are important for signal processing, including `fft`, `ifft` and `filter`. There is also a growing community of programmers, particularly in academia, who use Python in combination with additional packages (e.g., Numpy, Scipy and Matplotlib) to do scientific computing. Because the basic syntax of Matlab and Python is quite similar, should be relatively easy for an experienced programmer to translate code in this book from Matlab to Python. Depending on your application, one of these open-source alternatives might suffice for your needs. However, all the laboratory exercises in this book are based on basic Matlab.

C.2 The elements of Matlab

Matlab features an impressive collection of basic and advanced functions. Here are a few highlights.

C.2.1 Calculator functions

At its simplest, Matlab is a calculator. After the prompt (>>), you type the expression you wish to evaluate:

```
>> 1+1
ans =
    2
```

More complex calculations can be done:

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

The usual rules of precedence apply in Matlab to expressions such as this: exponentiation is done first, multiplication and division are next (and equal) in precedence and are done left to right. Addition and subtraction are lowest in precedence. Parenthesis can be used to override these usual rules of precedence.

C.2.2 Variables

In addition to using Matlab as a calculator, you can set and manipulate variables

Variable basics

At its simplest, you can set variables from the command line.

```
>> x = 3
x =
    3

>> y = 2
y =
    2

>> z = x + y
z =
    5
```

Note that Matlab echoes your work each time you type return. You can suppress this tedious echoing by terminating each line with a semicolon:

```
>> x = 3;
>> y = 2;
>> z = x + y;
>> z
z =
    5
```

The last line shows that just typing the variable name, Matlab gives us the value of the requested variable. You can also get Matlab to display a variable's value less verbosely:

```
>> disp(z)
    5
```

Matlab allows multiple commands on a single line both on the command line and in the editor, for example,

```
>> x = 3; y = 2; z = x + y;
```

However, I recommend that you don't do this. Putting multiple commands on a single line makes code harder for you (and others) to read and harder to debug.

Matlab keeps all your variables around until you clear them with the `clear` command. You can clear all variables at once or just specific variables. Here's how you can query all your variables in the current Matlab workspace:

```
>> who
Your variables are:
ans    x    y    z
```

You can also get a more involved display:

```
>> whos
  Name      Size          Bytes  Class
  ans       1x1               8  double array
  x         1x1               8  double array
  y         1x1               8  double array
  z         1x1               8  double array
Grand total is 3 elements using 24 bytes
```

`ans` is a built-in Matlab variable that gives the result of the last computation. You can use `ans` just like any other variable, and it is replaced with each successive computation

```
>> 1+1
ans =
    2

>> ans + 2
ans =
    4
```

There are other "sort-of" reserved Matlab variables, such as `i` and `j` (see below) and `pi`. We'll have more to say about the "sort-of" qualification, below.

```
>> pi
ans =
    3.1416
```

By default, all numeric variables and calculations upon them are done in double precision floating point and are displayed in a format specified by the Matlab `format` command. The default format is short floating point. That's why the value of `pi`, above, was a bit truncated. That's easy to fix:

```
>> format long
>> pi
ans =
    3.14159265358979
```

Matlab is inherently capable of working with complex numbers. The variables `i` and `j` are used by Matlab to denote the imaginary part (mathematicians and physicists tend to use `i` while engineers use `j`). All operations (i.e. multiplication, division, addition and subtraction) work transparently with complex numbers

```
>> x = 1 + 2 * j
x =
    1.0000+ 2.0000i

>> y = 2 - j
y =
    2.0000- 1.0000i

>> z = x + y
z =
    3.0000+ 1.0000i
```

All Matlab's arithmetic functions work with complex numbers:

```
>> exp(z)
ans =
  10.8523 +16.9014i
```

However, watch out! You can use `j` (or any other built-in Matlab variable) as a variable in your own program by reassigning its value, but then it will no longer denote the imaginary operator:

```
>> j = 2;
>> x = 1 + 2 * j
x =
    5
```

Ooops! This is really pretty terrible. Matlab shouldn't allow this, but it does. You can reset a built-in Matlab variable such as `j` to its default value by clearing it:

```
>> clear j
>> j
ans =
    0 + 1.0000i
```

Better yet, use `1j` instead of using `j`, which will make your syntax unambiguous:

```
>> x = 1 + 2 * 1j
x =
    1.0000+ 2.0000i
```

The previous example shows that you can use `clear` to delete some or all variables and functions that you have defined:

```
>> who
Your variables are:
ans       x         y         z
>> clear y z
>> who
Your variables are:
ans       x
```

"Reserved" variables

Well, as we've just seen, Matlab doesn't prevent you from redefining important predefined variables, so it's incumbent upon *you* to know what they are. Here are the most important ones:

`ans`	Default variable name returned by a function with no specified output argument
`pi`	π
`i` or `j`	j
`inf` or `Inf`	∞. This occurs naturally when attempting to do a division such as $1/0$. You can also use it as an argument; for example, `atan(Inf)`
`nan` or `NaN`	Not a number. Returned by a calculation such as $0/0$ or `sin(inf)`.
`eps`	The smallest distinguishable positive real number.

Vectors and matrices

Matlab's most powerful feature is the ability to express and vectors and matrices compactly and operate upon them efficiently. Vectors (or arrays) in Matlab are declared using square brackets:

```
>> x1 = [1 2 3 4 5]
x1 =
     1     2     3     4     5
```

`x1` is a row vector. Putting a semicolon between elements creates a column vector:

```
>> x2 = [1; 2; 3; 4; 5]
x2 =
     1
     2
     3
     4
     5
```

The transpose operator converts row vectors to column vectors

```
>> x3 = x2'
x3 =
     1     2     3     4     5
```

Be careful when taking the transpose of complex quantities, since the transpose operator creates the Hermitian (the conjugate transpose):

```
>> xc = [1+j 2+j 3+j]
xc =
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i

>> xc'
ans =
    1.0000 - 1.0000i
    2.0000 - 1.0000i
    3.0000 - 1.0000i
```

To get a non-conjugate transpose, use the dot-transpose operator:

```
>> xc.'
ans =
    1.0000 + 1.0000i
    2.0000 + 1.0000i
    3.0000 + 1.0000i
```

Another source of error is white space. Some programmers put a lot of white space between characters to aid in readability (e.g, `x = 1 + 2 * j`), whereas other folks like a more compact style (e.g, `x=1+2*j`). In Matlab, adding white space doesn't matter… until it does. Look at this:

```
>> xc = [1 +j 2+j 3+j]
xc =
   1.0000 + 0.0000i   0.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i
```

Oops! The white space between the `1` and the `+j` led to an array of four numbers, not three.

Matlab provides an easy way of generating sequences like `x1`, above:

```
>> x1 = 1:5
x1 =
     1     2     3     4     5
```

You can increment or decrement by integers or fractions,

```
>> 1:2:5
ans =
     1     3     5
```

or backwards:

```
>> 9:-2:3
```

```
ans =
    9     7     5     3
```

or in non-integer increments or decrements:

```
>> 5:-0.5:3
ans =
    5.0000    4.5000    4.0000    3.5000    3.0000
```

Matrix arithmetic with Matlab is easy. For example, to add two vectors of the same dimension, you simply use the `+` sign:

```
>> y = x1 + x3
y =
    2     4     6     8    10
```

Vector multiplication is also simple. Given `x1` and `x2` from above,

```
>> z = x1 * x2
z =   55
```

which is of course different from

```
>> z = x2 * x1
ans =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25
```

Matrices (and arrays) of compatible dimensions can be concatenated. Matrices with the same number of rows can be horizontally concatenated by using `[m1  m2 …]`; matrices with the same number of columns can be vertically concatenated using `[m1; m2; …]`, for example,

```
>> [x1 x3]
ans =
     1     2     3     4     5     1     2     3     4     5

>> [x1; x3]
ans =
     1     2     3     4     5
     1     2     3     4     5
```

Adding or subtracting vectors or matrices with differing dimensions in Matlab will generally produce an error (try it). However, the newest versions of Matlab allow what is called “implicit arithmetic expansion”. Prior to version R2016b, if you had a matrix like `z`, above, and you wanted to add a numbers `1:5` to each column respectively, you had to add a matrix that was exactly the same size as the original one, something like this:

```
>> z + ones(5, 1)*(1:5)
```

In newer versions of Matlab, you can just do this

```
>> z * (1:5);
```

This new syntax could be useful, but it can also lead to unexpected errors if used improperly.

C.2.3 Matlab functions

All Matlab's arithmetic functions are fully vectorized, meaning that they accept vectors and matrices as arguments, for example:

```
>> cos(2*pi*(0:7)/8)
ans =
    1.0000  0.7071  0.0000  -0.7071  -1.0000  -0.7071  -0.0000  0.7071
```

In a language such as C, you'd have to use a `for` loop to obtain the same result.

All of Matlab's elementary functions also accept complex arguments, which means that complicated expressions can be implemented very economically. This is something we frequently want to do in DSP programs. For example, consider the N -pt discrete Fourier transform (DFT) of an N -pt sequence, $x[n]$, defined by

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-jk2\pi n/N}, \qquad 0 \le k < N$$

All N values of $X[k]$ can be efficiently computed with a single line of code:

```
>> N = 8;
>> x = [1 1 1 1 1 0 0 0];
>> k = 0:N-1;
>> n = 0:N-1;
>> X = exp(-j*2*pi*k'*n/N)*x' % Compute DFT of x[n]
X =
   5.0000 + 0.0000i
   0.0000 - 2.4142i
   1.0000 + 0.0000i
   0.0000 - 0.4142i
   1.0000 + 0.0000i
  -0.0000 + 0.4142i
   1.0000 + 0.0000i
  -0.0000 + 2.4142i
```

It's worth understanding what happened here. A matrix multiplication such as `k'*n` creates a $k \times n$ matrix, where each row corresponds to a value of index, k. This entire matrix is then exponentiated in one fell swoop to form the $k \times n$ matrix, $e^{-jk2\pi n/N}$. When this matrix is multiplied by the $n \times 1$ column vector corresponding to $x[n]$, the result is the $1 \times k$ row vector that corresponds to $X[k]$. Writing vectorized code like this is extremely efficient. Computing this DFT without vectorization would require a pair of nested `for` loops and would be almost an order of magnitude slower. While Matlab's vectorized operators are powerful, they also allow you to write code that's as dense as a neutron star, and so obtuse as to be almost unintelligible to others and possibly even to yourself when you come back to it after a while. Accordingly, it's a good idea to comment byzantine code if you use it.

Accessing array values

You can access any value of an array or vector with subscripts of the general form `z(row, column)`:

```
>> z(2, 3)
ans =
    6
```

It is important to remember that Matlab uses ones-based indexing. Trying to declare or access array values with non-positive integers produces an error:

```
>> y(0)
??? Index into matrix is negative or zero.
```

This is really unfortunate for DSP, since we would like to be able to manipulate arrays with zero or negative indices. Fortunately, we will find a way around this restriction in the laboratory for Chapter 2, by creating our own `sequence` class, which will allow us to index with both positive and negative integers.

In Matlab, you can use integer arrays as subscripts as well:

```
>> y(1:3)
ans =
    2     4     6

>> z(2:4, :)
ans =
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
```

In the preceding example, the first argument to `z` says that you want rows 2 through 4. The second argument (the colon) is shorthand that tells Matlab you want all the columns. You can do very sophisticated things with the array indices, such as replacing or manipulating portions of vectors and matrices. Here we replace rows 2 to 4 in columns 3 and 5 with the scalar value 99:

```
>> z(2:4, [3 5]) = 99
z =
     1     2     3     4     5
     2     4    99     8    99
     3     6    99    12    99
     4     8    99    16    99
     5    10    15    20    25
```

Like its forbear, FORTRAN, Matlab stores data uses column-wise ordering or indices. You can actually access a multidimensional array with a single index, which demonstrates the column ordering:

```
>> z(3:8)
ans =
     3     4     5     2     4     6
```

Particularly with large arrays, indexing data by columns first is faster than indexing by rows.

Matlab gives you the ability to produce arrays and matrices with all zeros or ones using the `zeros` or `ones` commands. For example, here we produce an array of four columns:

```
>> zeros(1, 4)
ans =
     0     0     0     0

>> ones(1, 4)
ans =
     1     1     1     1
```

One of the uses of the `zeros` command is to preallocate a data array when you are performing a calculation in a `for` loop that adds an element to an array, particularly a large array. Consider the following fragment:

```
y = 0;
for n = 1:10000
    y = [y n];
end
```

Here, Matlab has to resize, move and copy the array for `y` repeatedly as new elements are added on each pass through the loop. This leads to slow execution. Not good! Do this instead:

```
y = zeros(1, 10000)
for n = 1:10000
    y(n) = n;
end
```

Here, we first preallocate the entire array. Then, each pass through the loop just replaces one of the zero values with the desired value.

Size of matrices and arrays

You can determine the size of a matrix with the size command, which has several forms

```
>> sz = size(ones(4, 5))
sz =
     4     5

>> [nrows, ncols] = size(ones(4, 5))
nrows =
     4
ncols =
     5

>> nrows = size(ones(4, 5), 1)
nrows =
     4

>> ncols = size(ones(4, 5), 2)
ncols =
     4
```

For arrays (i.e., column or row vectors) you can use `length`, but watch out: the function will return the larger of the two dimensions.

```
>> len = length(zeros(2, 4))
len =
     4

>> len = length(zeros(4, 2))
len =
     4
```

The reserved keyword `end` can be used in the index of arrays. For example

```
>> x = 1:5;
>> x(3:end)
ans =
     3     4     5
```

`end` can be used to flip arrays:

```
>> x(end:-1:1)
ans =
     5     4     3     2     1
```

You can also use the Matlab commands `fliplr` and `flipud` commands to flip arrays and matrices. However, this brings up an important point. In Matlab, there are literally hundreds and hundreds of functions. You can spend a good chunk of your life looking for *just* the right function to perform your particular task. In some cases (e.g., `fft`), finding the right function is essential, because `fft` is a complicated, highly-optimized function you really don't want to program yourself. In other simpler cases (e.g. `fliplr`), it may be easier just to use basic Matlab language functions. The fewer special functions you call, the easier your code will be for others to understand and port to other languages.

Data types

Matlab is not a strongly typed language, in the sense that C is. That is, you don't have to declare variables as floating point or integer or character. However, Matlab does have a number of data types available in order to express floating-point and integer numbers, characters, and logical quantities. By default, all numerical expressions are double-precision floating point (eight bytes), but they can be cast (i.e. converted) to single precision (four bytes) with `single` and back to double precision by `double`. Operators `int` and `uint` with appropriate suffixes cast numerical quantities to integers. This is useful to implement fixed-point algorithms. For example, `uint8(x)` casts `x` to an eight-bit (one byte) unsigned integer; `int16(x)` casts `x` to a signed 16-bit (two-byte) integer. When dealing with integer arithmetic, watch out for overflows and underflows; for example,

```
>> uint8(-1)
ans =
    0

>> int8(257)
ans =
  127
```

Strings in Matlab are arrays of alphabetic characters enclosed in single quotes, for example,

```
>> str = 'hello';
```

Individual characters of the string can be addressed and concatenated just like elements of numerical arrays:

```
>> ['j' str(2:end)]
ans =
jello
```

Strings are of type `char`. String arrays can be cast to integers (or double) and integer arrays representing ASCII codes can be cast to strings:

```
>> x = int8(str)
x =
  104  101  108  108  111

>> char(x)
ans =
hello
```

Logical data arise from Matlab's logical operations and have value of either 0 or 1. A number of Matlab's functions and operations yield logical output – either 0 or 1. These include comparison operators on numbers (e.g., `<`, `==`, `>`, `all`, `any` and `find`), logical operations on logical data (e.g., `||` and `&&`) and a number of string operators (e.g. `strcmp`, `strfind` and `strmatch`). Judiciously used, these logical operators can save a lot of programming. For example,

```
>> x = 1:10;
>> indx = (x>3)&(x<7)
indx =
     0     0     0     1     1     1     0     0     0     0
```

Here that `indx` is logical data, can be used to select the parts of the numerical data array corresponding to logical 1:

```
>> x(indx)
ans =
     4     5     6
```

Notice that the only time you can use zeros indices into an array is if they are of the logical data type. You can also do the entire operation in one line:

```
>> x((x>3)&(x<7))
ans =
     4     5     6
```

You can cast logical data to double. For example, here's a trivial way to make a step function, $u[n]$, in Matlab:

```
function y = u(n)
    y = double(n >= 0);
end
```

Structures

Structures in Matlab are just like structures in C. They are containers that allow one to group together a variety of data under the umbrella of one variable name. The form of a structure is `variableName.field`. For example:

```
>> x.data = [1 2 3 4 5];
>> x.offset = -1;
>> x
x =
      data: [1 2 3 4]
    offset: -1
```

We have declared a variable `x`, with two fields. The `data` field contains an array, the `offset` field contains another number. Structures of this type can be useful for us in expressing sequences in Matlab. As we mentioned previously, Matlab always assumes that the index of the first element of an array is 1. So, when we just say

```
>> x = [1 2 3 4 5];
```

this means that `x[1]=1, x[2]=2`, and so on. But, what if we want to express a sequence `x[-1]=1, x[0]=2, ...`? We obviously need to provide the user with some additional information in addition to the array data, namely an offset. An offset of -1 tells the user that "we want you to interpret the first point in the Matlab array as corresponding to `x[-1]`." We can use structures for this purpose. We'll have a bit more to say about structures when we discuss classes, below.

C.3 Programming in Matlab

With Matlab, you can create applications using both procedural and object-oriented programming techniques. As a procedural language, you can create your own scripts and functions.

C.3.1 Scripts

A *script* is just a sequence of operations that Matlab executes one after the other, exactly as if you had entered the same operations in the command window one after another. Create a script using Matlab's editor by typing `edit` on the command line. When you finish entering your script, save it as a .m file. Make sure you save the file in your home directory, or in a directory that is on Matlab's path (see documentation for the `path` command). For example, here is a file `myscript.m`:

```
% MYSCRIPT A little script to add two numbers
%
% T. Holton
x = 3;
y = 5;
```

```
z = x + y
```

Now, when I type `myscript` following the Matlab prompt, I get the answer of this very complex calculation:

```
>> myscript
z =
    8
```

Note that the first few lines of the script start with the % character, and by default Matlab colors them green in the editor. This denotes a comment. It is good practice to comment your programs, giving the name of the program, the author and perhaps the revision history. If you put comments like this in the first few lines of the program, then your script is automatically accessible from the Matlab's `help` function:

```
>> help myscript
MYSCRIPT A little script to add two numbers

T. Holton
```

C.3.2 Functions

All the variables set in a script are available in the base Matlab workspace, that is to say, they are effectively global. This may be an advantage in some situations, but most often you'd like the script to work as an independent unit, perhaps one that takes one or more inputs and produces one or more outputs and doesn't litter your workspace with variables. This is precisely what Matlab's functions are for. Matlab's user-defined functions are like scripts except that all variables declared in the function and all calculations take place in a separate workspace (like those in other languages such as C). Functions look just like scripts except for the first and last lines:

```
function out = myfunction(in1, in2)
% MYFUNCTION A little function to add two numbers
%
% T. Holton
   temp = 2 * in1;
   out = temp + in2; % this is a comment
end
```

The first line of the function specifies the input and output variables. In this case there is one output variable, `out`, and two input dummy variables, `in1` and `in2`. The last line of the function is the `end` statement. By default, the Matlab editor colors this and other recognized commands in blue. When I call this function

```
>> clear
>> y = myfunction(3, 4)
y =
    10
>> who
Your variables are:
y
```

Because all lines in `myfunction` are terminated with semi-colons, the printing of all calculations from within the function is suppressed. Matlab did all the calculations required by the function in a separate workspace that is normally not accessible to you. None of the variables that we created in `myfunction` will show up in the calling (base) workspace. Also, since the input variables are dummy variables, if you ask for them in the function caller's workspace, you will be out of luck:

```
>> in1
??? Undefined function or variable 'in1'.
```

If you call a function from within another function, each function operates in its own workspace. There are ways of sharing variables between workspaces in Matlab if absolutely necessary; specifically, variables can be declared `global`

or one can use Matlab's builtin `assignin` function. However, extensive use of global variables is generally considered a hallmark of bad program design. It defeats the purpose of using functions, which is to isolate variables in different workspaces so that they can't interact with variables in the main workspace or other workspaces in unforeseen ways.

Function handles

The '@' sign is a Matlab operator that indicates to function handle, which is equivalent to a function pointer in C. Matlab functions have function handles that are just the function name preceded with the '@' sign; e.g., `@sin` and `@cos`. One can evaluate a function in a couple of ways, for example. by using the `feval` function with appropriate arguments:

```
feval(@functionhandle, arguments)
```

So for example, `feval(@sin, pi)` is equivalent to `sin(pi)`. You can also get the handle to any function that you create. The syntax is

```
f_handle = @function_name
```

So,

```
f_handle = @sin
f_handle(pi)
```

Use of function handles isn't restricted to Matlab's existing functions. Any function you have written is fair game. For example, in the previous subsection, we defined `myfunction(in1, in2)`, a function of two input variables and one output variable. Using the function handle syntax, we could call the function this way,

```
f_handle = @my_function
fhandle(in1, in2)
```

or just

```
feval(@myfunction, in1, in2)
```

Anonymous functions

One useful application of function handles is to define an *anonymous function*. An anonymous function is a function of a single output variable and one or more input variables that can be defined on a single line. Anonymous functions can be defined on the command line, in scripts or inside other functions. The syntax of the anonymous function is

```
output_variable = @(input_variables) function_of_input_variables;
```

For example, a single-line function that calculates the DFT could be defined as follows:

```
y = @(x) exp(-j*2*pi*(0:length(x))'*(0:length(x))/length(x))*x';
```

The output variable is `y`. In this example, there is a single input variable, `x`, prefaced by the '@' sign and enclosed in parenthesis. Following this is the definition of the function. As is the case with regular functions, the input variable is a dummy variable. Also, any variable used in the function definition must either appear in the list of imput variables or have been previously defined.

C.3.3 Conditionals and loops

Like all reasonable programming languages, Matlab has syntax that enables conditional and repeated execution of code. Conditionals are of the form

```
if (condition)
    statements
elseif (condition)
    statements
else (condition)
    statements
end
```

where `else` and `elseif` are optional. The parentheses around condition statements are also not necessary, but I recommend then to aid in the readability of the code.

Loops are either of the form of a `while` loop or the familiar `for` loop:

```
while (condition)
    statements
end

for index = values
    statements
end
```

`break` allows for the premature termination of a `for` or `while` loop, for example

```
for k = 1:10
    if (k > 4)
        break
    end
end
```

By default, the editor displays all recognized function names in blue.

C.3.4 Classes

Beyond simple scripts and functions, Matlab also offers a relatively fully featured object-oriented programming (OOP) language that permits you to create more complex applications with user-defined classes. Purists may complain that Matlab's OOP implementation lacks a number of features (e.g., templates, strong typing) found in other OOP languages. But at least since version R2008a, Matlab's OOP syntax and implementation has evolved to look similar enough to other "standard" OOP languages, that if you know object-oriented languages like C++, Java or Python, you'll feel right at home with Matlab. If you haven't run across object-oriented programming and plan to spend anytime professionally with Matlab, this approach to programming is well worth learning.

In brief, an *object* is essentially a software "container" that encapsulate data variables, termed *properties*, plus the functions, termed *methods*, that define the permissible ways the world can access or operate on the data. Objects are members of *classes* and are defined by a *class definition* file. For example, floating point variables in Matlab are members of the double class (actually, the double subclass of the numeric class); logical variables are members of the logical class, string variables are members of the char class and so on.

You can use Matlab's `class` and `isa` operators to interrogate the class of objects:

```
>> y = 6;
>> class(y)
```

```
ans =
    double

>> isa(y, 'double')
ans =
     1
```

There are a lot of methods in Matlab that allow us to operate on numeric data, for example `plus` and `minus`. Matlab's plus method returns the sum of two numeric variables, which is also of the numeric class:

```
>> plus(1, 2)
ans =
     3
```

Of course, ordinarily, we never call commonly used functions like `plus` and `minus` directly, since Matlab offers the familiar shortcut notation

```
>> 1 + 2
ans =
     3
```

What Matlab has done behind the scenes here is to note that the '+' operator has been called with two numeric arguments and then applied the `plus` function of the numeric class to arrive at the answer. (To see all the methods for this class, type `methods('double')`). The power of classes is that we can create our own classes and define how operators – even existing operators such as `plus` and `minus` will work on them. This reuse of operators in different classes is called *operator overloading*, and is one of the key features of object oriented programming.

As an example, we will use Matlab to create a *sequence class* with a data structure that describes the sequence's data and offset, plus a collection of methods that allow us to operate on one or more sequences, for example to add two sequences together using the '+' operator or plot a sequence.

To define a class in Matlab, you create an .m file with the name of the class, in this case, *sequence.m*. The class definition as well as all the methods that operate on sequence data, including those methods such as `plus` and `minus` that override (i.e. redefine) Matlab operations live in this file. Here's what the beginning of the file looks like:

```
classdef sequence
    properties
        data
        offset
    end

    methods
        function s = sequence(data, offset)
            % SEQUENCE   Sequence object
            %            S = SEQUENCE(DATA, OFFSET) creates sequence S
            %            using DATA and OFFSET
            %
            %            T. Holton  1 Jan 2017

            s.data = data;
            s.offset = offset;
        end

        function s = plus(s1, s2)
        %    Add two sequences, s1 and s2, and provide an output sequence, s
            data = s1.data + s2.data; % <-- bogus
            offset = s1.offset + s2.offset; % <-- bogus
```

```
            s = sequence(data, offset);
        end
    end
end
```

The code starts with `classdef sequence`, which tells Matlab that you are defining a class called `sequence`. Following this is a `properties` block terminated by an `end` statement, which enumerates the properties of the class. In this case, there are only two properties: `data` and `offset`. There follows a `methods` block, also terminated by an `end` statement, which includes all the functions that construct the class and allow us to operate on its data. The first function in the block is required to be the class *constructor*:

```
function s = sequence(data, offset)
```

The constructor creates and returns an instance of the sequence object, `s`. The object, `s`, is a structure that has the data of the sequence in the `data` member and the offset (i.e. the first index) in the `offset` element. For example, to create a sequence with data `[1 2 3 4]` and offset `-1`, we write

```
>> s = sequence([1 2 3 4], -1)

s =
  sequence handle

  Properties:
      data: [1 2 3 4]
    offset: -1

  Methods, Events, Superclasses
```

Matlab tells us, in a very verbose fashion, that we've just created a sequence. `s` isn't a member of the numeric class; it's a member of our newly defined class, the sequence class.

```
>> class(s)
ans =
    sequence
```

or.

```
>> isa(x, 'sequence')
ans =
     1
```

We can now define functions — such as addition, subtraction and plotting – that work on the sequence class. In the code fragment above, I've defined a `plus` method that adds two sequences, `s1` and `s2`, with different offsets and data, and generates a new output sequence, `s`. The particular algorithm I've used to compute the `data` and `offset` of the output sequence is bogus, but the last line of the method is relevant. Once you've properly computed the parameters of the output sequence (e.g., `data` and `offset`), you call the constructor, which returns a new sequence object.

In the laboratory accompanying Chapter 2, we (or, more accurately, *you*) will write a non-bogus `plus` method that can do the following:

```
>> x = sequence([1 2 3 4], -1);
>> h = sequence([1 1 1], -2);
>> x + h
ans =
      data: [1 2 3 3 4]
    offset: -2
```

We are using the same operator ('+') for the sequences as we do for numeric data. When you call the `plus` operator either explicitly (e.g. `plus(x, h)`) or implicitly (e.g. `x+h`) with objects of the sequence class as arguments, Matlab will look in the *sequence.m* file to find the `plus` method, and then use it. You can also overload functions such as Matlab's `conv` function to deal with sequences.

```
>> conv(x, h)
ans =
    data: [1 3 6 9 7 4]
  offset: -3
```

Pretty handy! All the overloaded functions for your sequence class such as `plus`, `minus` and `conv` must live in the methods block of the *sequence.m* file, which is where Matlab expects to find them.

C.4 Matlab help

Matlab has an excellent help system. To get help on anything else in Matlab, use the `help` or `doc` functions. `help` lists information in the command window; `doc` opens Matlab's documentation browser a separate window and provides more comprehensive information and often examples of usage. For example, to get help on any given function, type

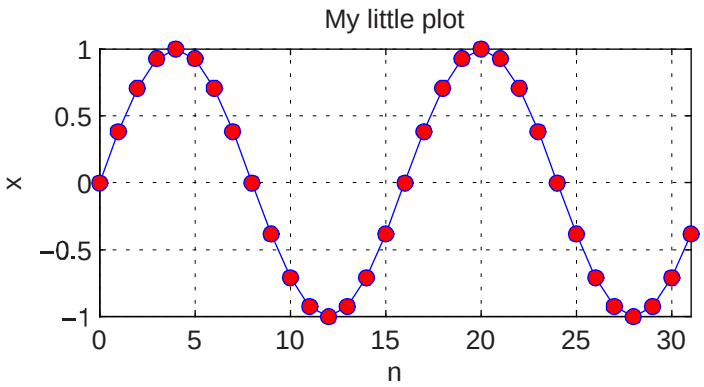
```
>> help sin
>> doc sin
```

C.5 Plotting

One of Matlab's best features is its comprehensive set of powerful and easy-to-use plotting functions. The most useful for us are the basic `plot` and `stem` functions. Here's a small example that demonstrates a few of the available features:

```
set(gcf, 'Unit', 'inch', ...
    'Position', [1 1 4 2], ...
    'PaperPositionMode', 'auto')
n= 0:31;
y = sin(2*pi*n/16);
plot(n, y, 'o-', ...
    'MarkerSize', 6, ...
    'MarkerFaceColor', 'r')
grid on
axis tight
xlabel('n')
ylabel('x')
title('My little plot')
print('MyLittlePlot', '-dpng') % output to a .png file
```

The editor displays strings in purple. Here's the result of running the code:



The `set` command specifies the units and position of current plot on the screen (`gcf` means 'get current figure'). Setting `PaperPositionMode` to `auto` is necessary if you wish to maintain the aspect ratio of the plot you see on the screen when you print your figures to an output device or to a file using Matlab's `print` command, as I've done here.

Relatively sophisticated control of plot attributes, such as colors, line styles, symbols, text and axes, as well as multiple and overlaid plots are possible with Matlab's plotting functions. Type `doc plot` for more info.

C.6 The Matlab environment

Matlab's integrated development environment (IDE) makes the work of writing, debugging and running code easy. The Matlab IDE has a number of components, the most important of which are the Command window and the Editor.

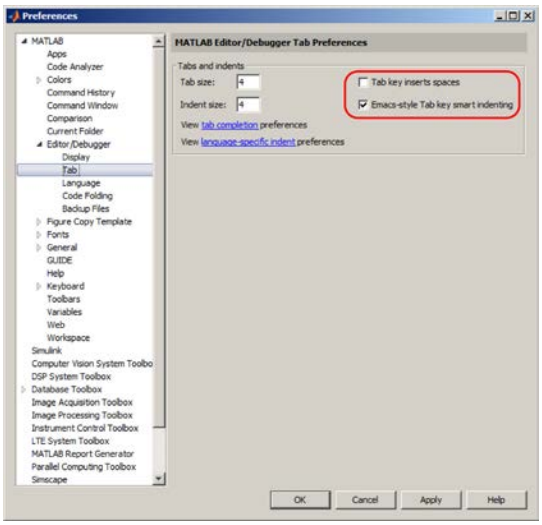
C.6.1 Command window and editor

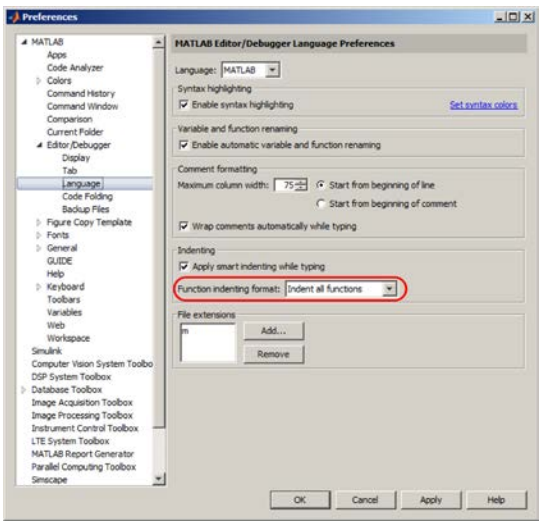
Command Window

When you launch Matlab for the first time, a large window opens that contains a number of panels. Depending on the version of Matlab you have installed, these can include the *Command Window*, *Current Directory*, *Workspace*, and *Command History*. My personal preference is to close all of these windows except the Command Window so that you'll have the whole window to enter commands and view results.

Editor

Typing '`edit`' at the command prompt ('`>>`') in the command window launches Matlab's *Editor*. While in the editor you can write, edit, run and debug code. While the editor may initially be docked inside the main Matlab window, you will most likely find it convenient to undock it to its own window on your desktop. The IDE has many parameters that are controlled via the *Preferences* panel, which is accessed through the File menu of the Main Window in older versions of Matlab, and through the Environment tab in more recent versions. In order to make your code easier to read, allow me to recommend that you set the editor to use 'Emacs-style tabs' and require it to indent all functions, as shown below.

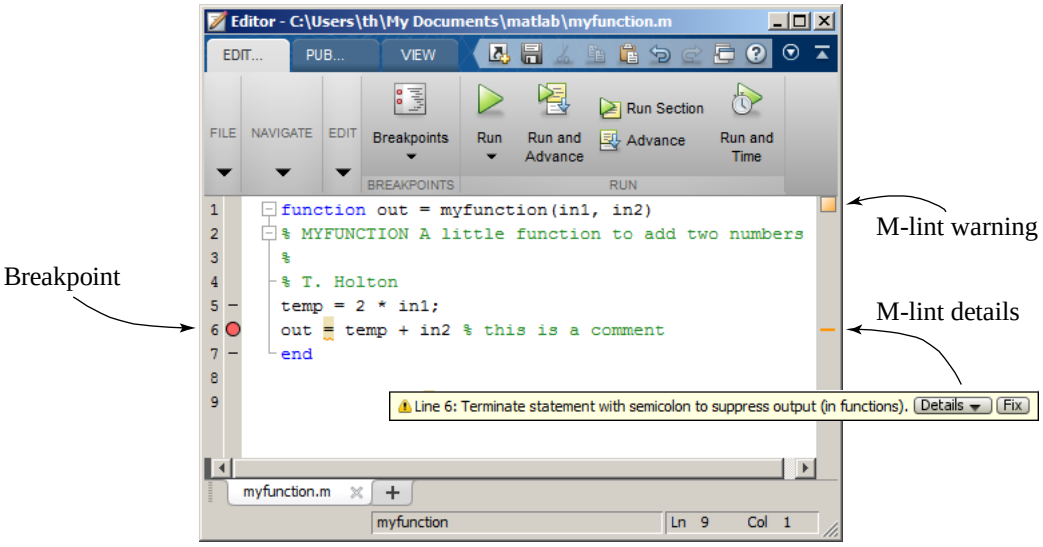




With these settings, when you select all your code (e.g. by typing 'Ctrl-a') and push the tab key, your code will be properly indented, which improves readability and helps in debugging.

C.6.2 Debugging and writing 'clean' code

With the Matlab IDE, you can set absolute or conditional breakpoints, step through code line-by-line or run it in sections. You can also interrogate workspace variables. In the example below, I've set a breakpoint in the editor at line 6 by clicking in the margin at the location of the red dot. When the program is run, it will stop at that line. By hovering the mouse over any variable in the editor, you will see its value.



Matlab acts like an interpreted language, in that you can enter commands directly in the command window and it will execute them immediately, but, it is actually a just-in-time (JIT) compiler. That means that while you are writing a piece of code in the editor, each time you add a character or terminate a line, Matlab is furiously recompiling your program in the background. Recent versions of Matlab include a valuable tool to check your code called *M-lint*. M-lint is the Matlab's version of the C-language 'lint' program. It is built into the editor and enabled by default. It continuously combs through your code looking for incorrect, ambiguous, inefficient or deprecated syntax and usage, unused or uninitialized variables and more. You may not find it necessary to use M-lint for short programs, but if/when you write longer programs, it can be valuable in identifying potential problems before they cause you headaches. In the screenshot

of the short program shown above, the M-lint notifications are indicated on the right window border. Here, M-lint has noticed something minor, namely, that one statement has not been terminated with a semicolon, which means that this line is going to produce unwanted output in the command window when the function is called. It has flagged the offending line with a thin yellow bar and has issued a warning that you can see by hovering your mouse over the line. You can choose to have M-lint fix the problem or ignore the warning, and you can disable M-lint entirely by unchecking a box in the preference panel of the editor if its punctiliousness annoys you.