The Freemat 4.0 Primer

By
Gary Schafer
Timothy Cyders

First Edition
August 2011

Table of Contents

About the Authors

I originally put this book together for my own use, but realized others might find it useful, too. Since the original version came out, Timothy Cyders has joined in. The original version was geared towards the Windows version of Freemat 3.6. This version will (a) be geared towards Freemat 4.0 and (b) will have information for both Windows and Linux.

Any suggestions for improvements, corrections, or the like can be sent to garystar1 at gmail dot com, or (even better) posted to the Freemat group on Google.

This book is not exhaustive. We don't know how to use every single button, icon, command, switch, dial, gauge, and lever within Freemat. Further, this book was designed primarily for the beginner. If you are looking for a book with details of all Freemat commands, you will want the *Freemat v4.0 Documentation* by Samit Basu. It is available for free online at http://freemat.sourceforge.net/FreeMat-4.0.pdf.

Acknowledgements

We'd like to thank the Freemat team (Samit Basu, Eugene Ingerman, Chuong Nguyen) and the other secondary players who have brought Freemat to life. We'd also like to thank the American Physical Society Public Outreach Program for providing the data from the roller coaster rides at Six Flags. Nothing like real-world data to demonstrate many of the basic concepts of Freemat (importing data, creating graphs, performing numerical integration and differentiation).
Finally, I would like to thank my wife, Debbie, for putting up with me while writing this. You make it worthwhile.

User Assumptions

We assume that you have Freemat properly installed and working. If you have any issues, direct them to the online Freemat group, http://groups.google.com/group/freemat.

This book was originally written for the Windows version. The book now covers more of the Linux and Mac versions, as well. In those cases where there are differences, we'll point them out.

How This Book Was Put Together

The images used Freemat Ver 4.0 (primarily) running within Microsoft Windows 7 (desktop system), MS Vista Basic Home Edition (laptop) and Ubuntu Linux (Maverick Meerkat, ver 10.10, on a desktop system). The book was written with OpenOffice Writer (various versions, starting with 3.0.5), while I used Gimp for the graphics.

Note that we assumed that most users would use already-built versions. Therefore, we used the same. We used the Windows executable (.exe) from the Freemat download web site for all Windows versions. For Linux, we used Ubuntu's "Software Center" to download the latest version available from the repositories.

In case you're wondering why it took so long to get this one out, well, I actually had a version ready

over a year ago. Lots of great examples, pictures, the whole nine yards (as we say in America). Then the hard drive upon which it rested died. Horribly. It's taken me this long to put everything back together.

Licensing

This book is published under the GNU Free Documentation License to conform with the GNU Public License (GPL) of the Freemat software. As such, I've made this document available in both Adobe PDF and OpenOffice open document text format (.odt). It may be reproduced for free by anyone, so long as the authors are given credit where due.

Using with Freemat v4.0 Documentation

Samit Basu, the man who wrote Freemat, created a document, the *Freemat v4.0 Documentation*, to explain all of the commands and functions available in Freemat. This document is available online and can be used along with this book to help better understand Freemat. Next to many of the commands and functions in this book, you will find a statement such as (*FD*, p. 121). That means that you can find more (and probably better) information on that function on page 121 of the *Freemat Documentation*.

Topic 1: Working with Freemat

Topic 1.1: The Main Screen - Ver 4.0

This is how Freemat looks in Windows 7 when you first start it up (Figure 1).

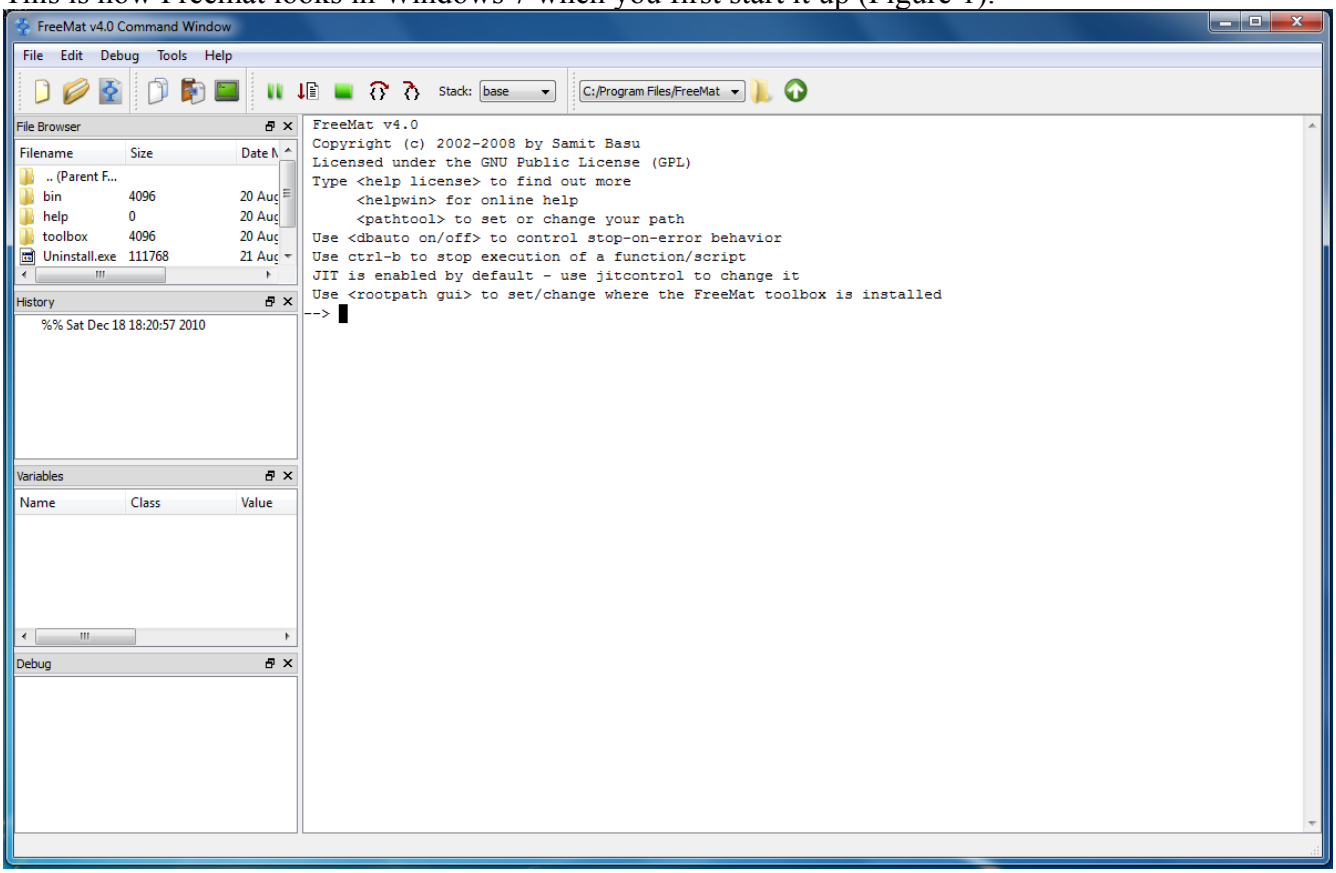


Figure 1: Main Freemat window as it appears in Microsoft Windows 7. This appearance is similar regardless of the underlying operating system.

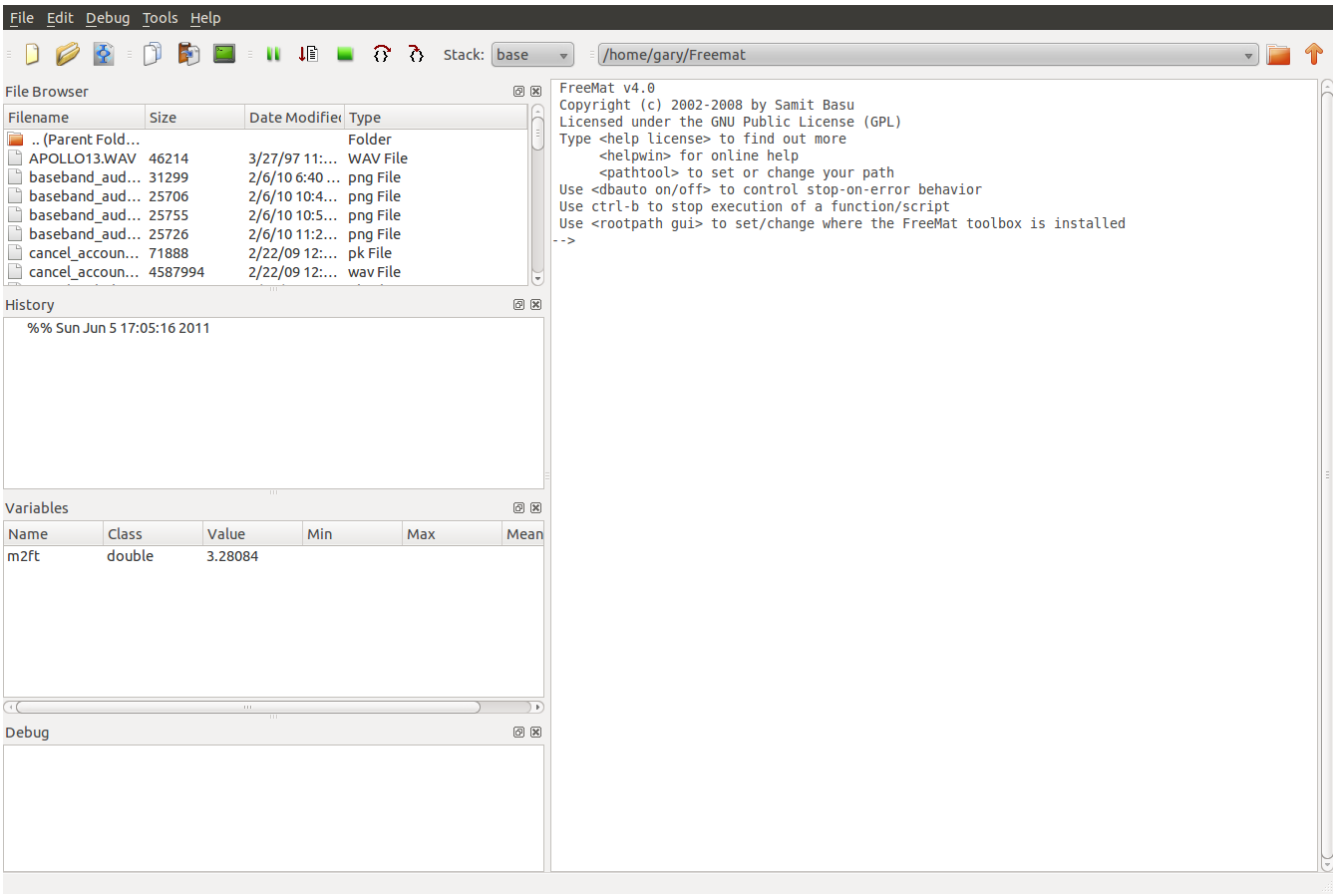


Figure 2: Main Freemat 4.0 window as seen in Linux (ver. 11.04).

Using the Command Window, its possible to enter variables, functions, programs and commands. The History Window shows, in chronological order, all of the commands that have been entered and the result. The newest commands are listed at the bottom; the older ones at the top.
When you first bring up the Freemat program, the first few lines of the Command Window will provide some basic information on the program, such as the version, copyright, and how to tie into the directories. The information for version 4.0 is shown below.

Example - Beginning lines of Freemat 4.0

```
FreeMat v4.0
Copyright (c) 2002-2008 by Samit Basu
Licensed under the GNU Public License (GPL)
Type <help license> to find out more
     <helpwin> for online help
     <pathtool> to set or change your path
Use <dbauto on/off> to control stop-on-error behavior
Use ctrl-b to stop execution of a function/script
JIT is enabled by default - use jitcontrol to change it
Use <rootpath gui> to set/change where the FreeMat toolbox is installed
-->
```

This window is divided into five different areas. These are the Command section, the File Browser, the History section, the Variables section and the Debug section. There's also a small area showing the working directory. This is shown in Figure 3.

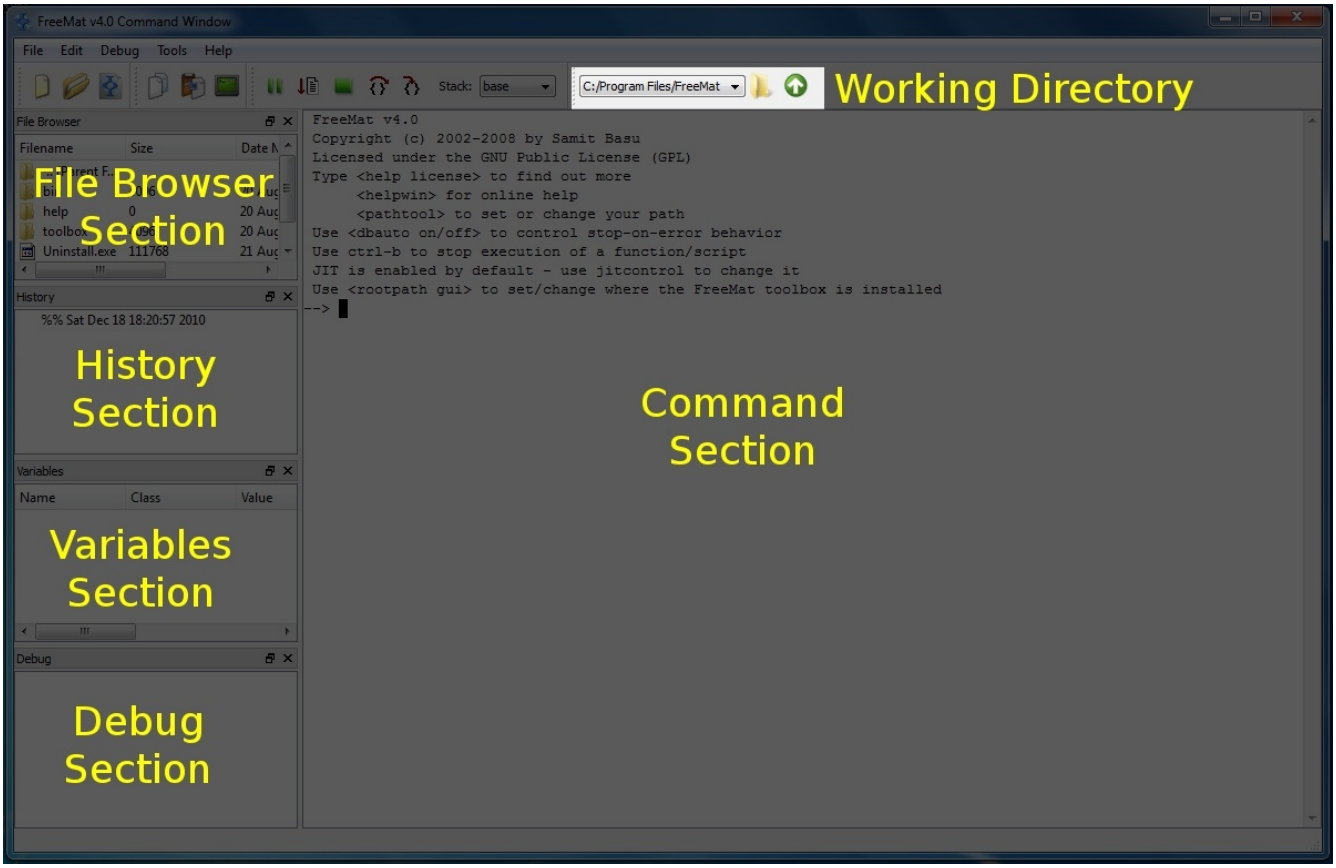


Figure 3: The Freemat 4.0 window with the different sections annotated.

We'll be spending most of our time covering the Command section. This is where the action happens. However, let's look briefly at the other sections and the working directory.

Topic 1.1.1: The File Browser Section

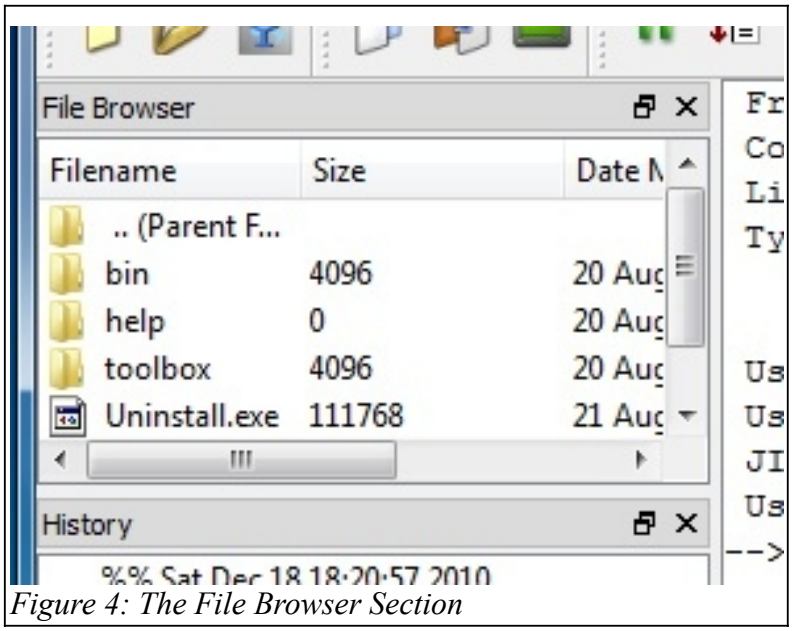


Figure 4: The File Browser Section

This section shows the standard directory tree of the working directory. We'll cover the working directory and its relationship to the files that Freemat deals with in Topic 1.1.6: The Working Directory.
In Freemat 3.6, this window only updated when the program was exited and re-entered. That has changed to where this window will now update each time the directory is changed.

Topic 1.1.2: The History Section

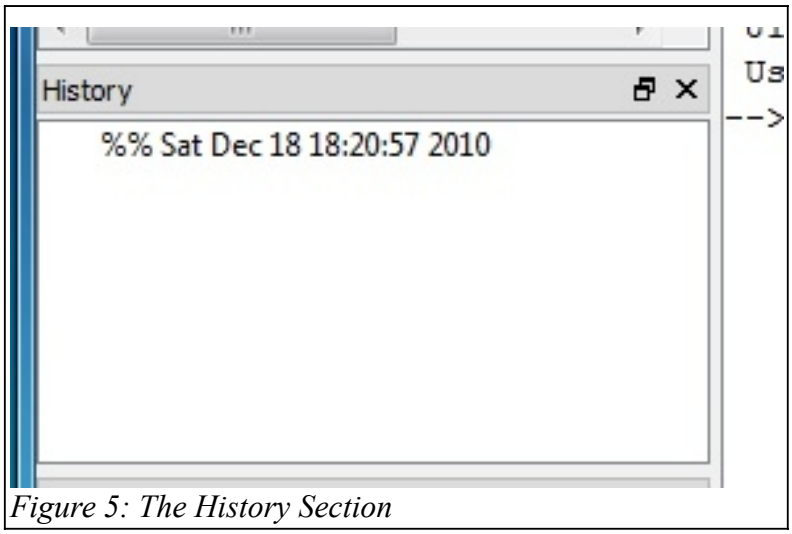


Figure 5: The History Section

The History section shows a list of every entered command. The most recent is at the bottom and the oldest is at the top. From the command prompt within the Command Window, you can select between previous commands in the history window using the up and down arrow keys. The history file can even be used from previous openings of the Freemat program. In other words, you can close the program, re-open it, and the history file will remain.

From the command prompt within the Command Window, you can select between previous commands using the up and down arrow keys.

You can re-enter commands from the History Window by using the up and down arrow keys within the Command Window, or by double-clicking on the command within the History Window.

You can remove everything from this window by selecting *Tools -> Clear History Tool.*

Topic 1.1.3: The Variables Section

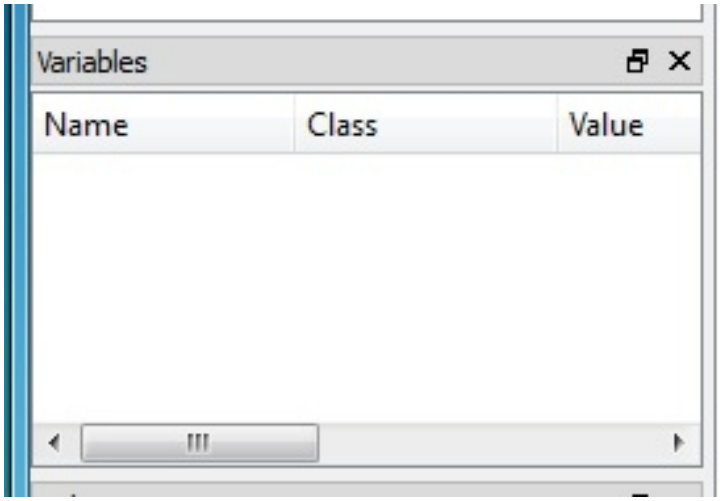


Figure 6: The Variables Section

The variables section shows all of the current global variables. If you worked with Freemat Ver. 3.6, this was found under the "Workspace" tab. Now it's under its own section. It will also show the variables used in a function so long as the function is operating. Frankly, I use this window perhaps more than any of the others. It gives me a quick preview of the variables being used, especially if I'm troubleshooting a script or function.

Topic 1.1.4: The Debug Section

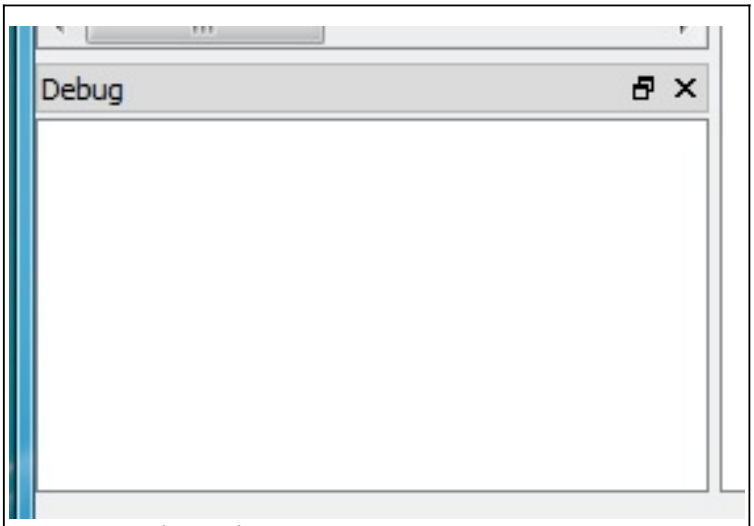


Figure 7: The Debug Section.

The "Debug" section shows any errors or warnings that Freemat provides. Typically, the first part of the line will be the time at which it occurred, followed by the text of the error or warning itself. See Figure 7. I mainly use this to identify when something goes wrong with a function or script I'm running.

Topic 1.1.5: The Working Directory

Figure 8: The Working Directory.

The working directory is nothing more than the default folder into which Freemat reads data files from and saves files to, unless a different folder is specified in the specific input or output operation. For example, if you use the print function to store an image, it will go into this directory unless you explicitly specify a different directory within the function. As well as from the small window at the top of the Freemat

window, you can see the working directory using the **pwd** (print working directory) command (FD, p. 420).
Here's an example from a Windows 7 system.

```
--> pwd
ans =
C:/Program Files/FreeMat
-->
```

Here's an example from a system running Linux.

```
--> pwd
ans =
/home/gary/Freemat
-->
```

Topic 1.2: Setting up Freemat

Before we go any further, let's set up a few things. When Freemat is running, it has certain folders on your hard drive that it uses. It will write data to your working directory, it will look for certain files that you have defined in your path (more on that in a minute), and it will read data from certain folders. When it's looking for scripts and functions, it will search every folder in your path list in order to find what it needs. The folders Freemat searches are, in order, the *working directory*, those listed under the *path* list, and the *rootpath*.

Search order for Freemat: Working Directory -> Path List -> Rootpath

Freemat will only look as far as it necessary to find the first instance of a file. As you work with Freemat, you may find that you've created multiple versions of a file with the same name. Except they're in different folders. When this occurs, if you call for a file with that one name, it will search until it finds the first instance of that file. "First instance" does not mean the first file date-wise; it means the first time it comes across that filename as it looks through the directory structure. For example, say you have multiple versions of a file "test.dat", with one copy in the path list and another version in the rootpath. Since it searches the path list first, it will use that version. In this instance, it will not use the file that exists within the rootpath.
Certain functions, such as *wavread()*, will only work if the requested file is in the working directory or if the absolute path is specified.
When Freemat writes a file, it writes that file to the working directory unless the absolute path is specified.

Caution - File Outputs for Windows Vista and Windows 7 Users

When Freemat is installed in either Windows Vista or Windows 7, the default working directory is made the Freemat folder within the Program Files directory (**C:/Program Files/Freemat**). Since WinVista and Win7 protect all files within the Program Files directory, as well as all sub-folders, Freemat will not be able to save any data to this directory or any of its sub-folders. And you thought that that funny "Cancel or Allow" Mac commercial was just being sarcastic? No. They were serious. This is due to the enhanced security Microsoft has built into these operating systems. It restricts which programs are allowed to manipulate the core operating files. **Unfortunately, neither Freemat nor Vista / Win7 will provide any indication that the save / store operation did not work.** I found this out the hard way while running some scripts on my Vista-based laptop. After running the scripts in which I expected to see saved data, I checked in the Program Files/Freemat folder and found... my data was not there. Please learn from my lesson. If you're using Freemat within Vista or Windows 7, change your working directory to some folder within your "User" folder. You'll be very glad you did!

Topic 1.2.1: Setting the Working Directory Using the cd Command

The working directory is nothing more than the default folder into which Freemat reads data files from and saves files to, unless a different folder is specified in the specific input or output operation.

The working directory is nothing more than the default folder from which Freemat reads data files and to which it saves files.

Example - Changing the Working Directory using the *cd* Command

The cd command changes the working directory to which directory is specified in the command. Here's an example from a Linux (Ubuntu) system.

```
--> pwd
ans =
/home/gary
--> cd '/home/gary/Freemat/'
--> pwd
ans =
/home/gary/Freemat
```

Here's an example from a Windows XP system.

```
--> pwd
ans =
C:/Program Files/FreeMat
--> cd 'C:/Documents and Settings/Owner/My Documents'
--> pwd
ans =
C:/Documents and Settings/Owner/My Documents
```

One thing to note. Typically, a Linux system uses a forward slash (/), whereas Windows systems sometimes use a forward slash (/) and sometimes use a back slash (\). In a Windows version of

Freemat, whether XP, Vista or Win7, it doesn't matter whether you use a forward slash (/) or a back slash (\) to separate the different directories. Freemat will change them to whatever it wants. In the Linux version, however, you must use a forward slash; otherwise, Freemat will return an error.
We'll cover scripts and functions in a later section. A script is nothing more than a set of commands listed in a text file (with the difference that it has a ".m" suffix). Freemat will run them one at a time. going from top to bottom, until it reaches the end. If there is a script called startup.m anywhere in the path for Freemat, it will run it each time that Freemat is started. Therefore, one way to set the working directory is to create a script called startup.m and use it to change the working directory using the cd command.

By default (for Windows users), the working directory is *C:/Program Files/Freemat x*, where *x* is replaced by whatever version of Freemat you first installed. And *x* may be different than what you are currently using. That's because Freemat may not change the directory name if you've upgraded. (It did not change for me when I upgraded Freemat on my WinVista machine from 3.5 to 3.6. It *did* change on my WinXP system.) By default, Freemat will attempt to save all of your data to your working directory. Okay, this is not *that* big a deal when saving scripts because Windows uses a standard "Save File" window that explicitly makes you choose a folder. But for your data reads (such as the fread, wavread or dlmread commands) and writes (such as the wavwrite or print commands), you should take heed and change the working directory to something *other* than the *Program Files* directory.

Example - Setting the Default Working Directory in a Windows System

After installing Freemat in a Windows system, you'll really want to change the default working directory. Changing the working directory such that it's set automatically when Freemat starts up is a three-step process. The steps are:

1. Create a new folder for your Freemat files in your "Users" directory (if you're running Win7 or WinVista), your "My Documents" directory (if you're running WinXP), or your home directory (if you're running Linux).
2. Add this new folder to your path list in Freemat.
3. Create a script called startup.m containing a **cd** command pointing to this new Freemat file.

The first step should be familar to anyone using a PC. Go into your personal directory, which is:

- "C:\Documents and Settings\<user name>" in a Windows XP system.
- "C:\Users\<user name>" in a Windows Vista or Windows 7 system.

Some examples of these are shown in Figure 9 and Figure 10.

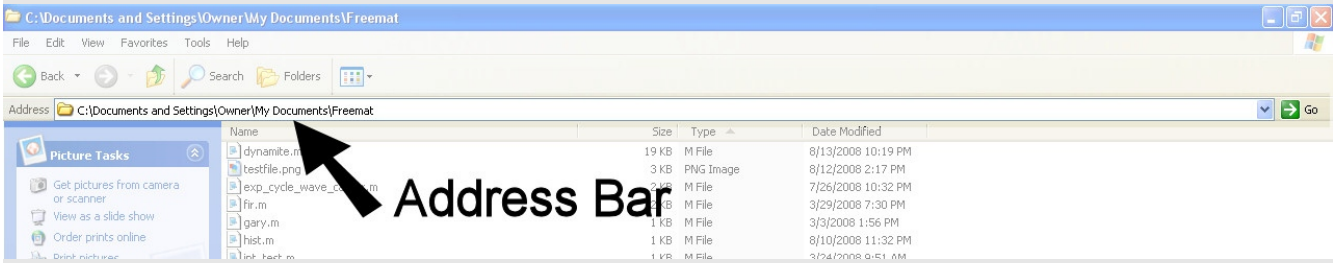


Figure 9: Windows Explorer showing Address Bar in a Windows XP system.

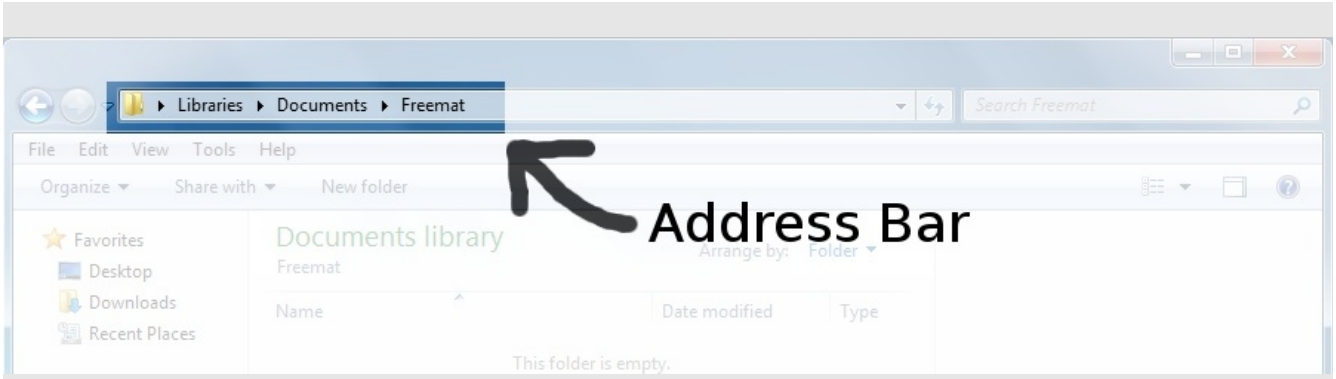


Figure 10: Windows Explorer showing Address Bar in a Windows 7 system.

The second step consists of opening the Freemat path tool, adding the folder just created to the path, saving the path, and closing the path tool. You can open the path tool in one of two ways. This is either by clicking on **Tools** -> **Path Tool** under the Freemat menu, or by typing **pathtool** and pressing <Enter> in the Command Section of Freemat. After opening the path tool, use the left pane to navigate to the folder you just created, click "Add" on the righthand side, then click "Save" and "Done", also on the righthand side. This is shown in Figure 11.

Caution - Changes Using the Path Tool

This applies to all operating systems. As of this writing, there is a bug in the Path Tool. If it is used to change the path, whether it be to add or remove elements from the path, it will not take affect until you close and restart Freemat. If you use the **path** or **setpath** commands directly from the Command Window to change the path list, they will take affect immediately without having to close and restart Freemat. If you're like me, you only need to change the path list occasionally. Hopefully, this bug will not be a major inconvenience.

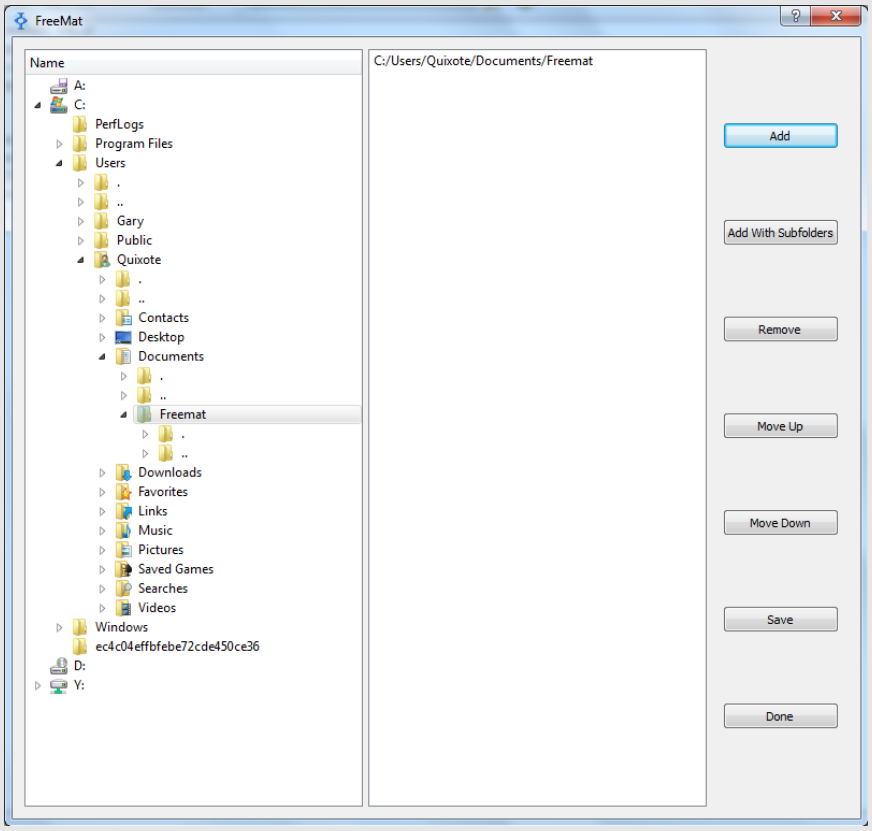


Figure 11: Path Tool window showing selection of newly-created folder under user directory.

The final step is to create a script called **startup.m** and place it in this folder. Here's a quick overview of the steps involved:

1. Open the Freemat Editor. To do so, type "edit" in the command section followed by pressing the <Enter> key or select Tools -> Editor from the Freemat menu.
2. In the editor window, type a **cd** (change directory) command (FD, p. 415) followed by the full path name of the folder you just created. Here are examples of the commands based on some different operating systems. You can also see an example, for a Windows 7 system, in Figure 12.
   (a) Windows XP: `cd 'C:\Documents and Settings\Owner\My Documents\Freemat';`
   (b) Windows Vista \ Windows 7: `cd 'C:\Users\Owner\Documents\Freemat';`
3. In the Editor window, click on File -> Save, navigate to the folder listed in the cd command, give it the name startup.m (This particular filename is important! Don't call it anything else!), and press "Save".
4. Exit the Editor window.

Now test your new script. Exit out of Freemat and re-start it. If the working directory is now listed as this new folder under your personal directory, then success! For now, anything you save from Freemat will be stored in this directory by default (again, unless you specifically say otherwise in whatever command is used to save the file).

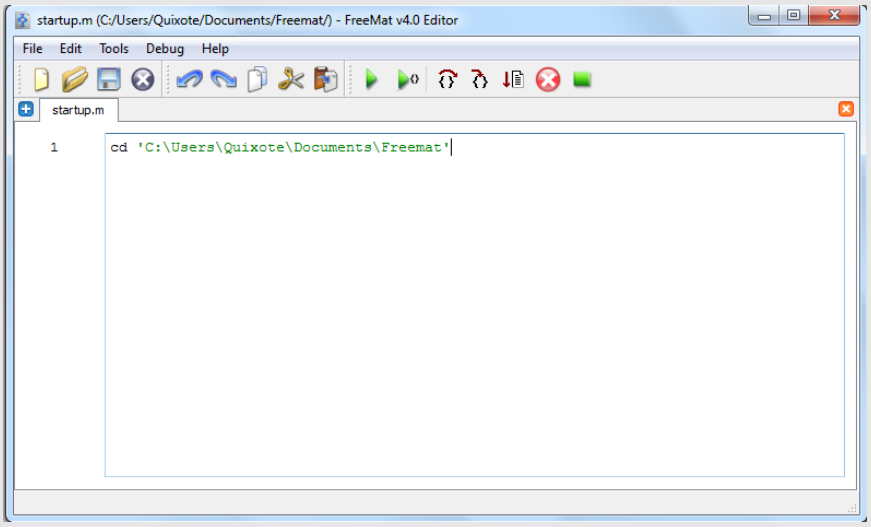


Figure 12: Freemat Editor window with cd command that changes the working directory to a folder under the user's personal directory. This is a Windows 7 system.

Example - Saving a File to the Working Directory

This is from the Freemat v4.0 Documentation. This example is being run on a Linux-version of Freemat. First, here's the current working directory:

```
--> pwd
/home/gary/Freemat
```

Here's a short set of commands that generates a set of random numbers, then writes them to a file called "test.dat". Since the path to the file is not explicitly stated, it's written to the working directory. In this case, it's "/home/gary/Freemat".

```
--> A=float(randn(512));
--> fp=fopen('test.dat','w');
--> fwrite(fp,A,'single');
--> fclose(fp);
```

This creates and opens a file called "test.dat" and makes it writable (the 'w' property in the **fopen** command). Since the path to the file is not specified, it's placed into the working directory.

Here's an example from a Windows 7 system showing how the operating system denies Freemat the ability to write to the default working directory (the **C:Program Files (x86)/Freemat** directory, to be specific).

```
--> pwd
ans =
C:/Program Files (x86)/Freemat
--> A=float(randn(512));
--> fp=fopen('test.dat','w');
Error: Access is denied. for fopen argument test.dat
-->
```

This error occurs because Win7 (and WinVista) prevent a user from making changes to the core files (such as are found in the Program Files directory). Both Win7 and WinVista require going through its

security features, such as prompting for the admin password or pressing the "Continue" button, in order to make changes to the core files.. Again, to keep this from happening, change the working directory as outlined below.

Topic 1.2.2: Changing the Working Directory in Windows

This option works under Windows, not in Freemat. To change the working directory under Windows, open a Windows Explorer window and navigate to your working directory. On my Windows XP system, the working directory is *C:\Documents and Settings\Owner\My Documents\Freemat* (See Figure 9). Note that this is simply a folder called *Freemat* under the *My Documents* directory. On my Windows 7 system, the working directory is *C:\Users\Quixote\Documents\Freemat* (See Figure 10.) The following steps will help you set the working directory under Windows.

1. Within Windows Explorer open to your working directory, as shown in Figure 13.
2. Highlight the file path name and right-click on the highlighted path name and select *Copy* from the menu, as shown in Figure 15.
3. Click on **Start** -> **All Programs** -> **Freemat3.6**.
4. Right-click on the icon for Freemat.
5. In the menu that comes up, select *Properties* at the bottom. A window will open showing the various properties for the program and icon. See Figure 15.
6. Highlight the text in the area titled "Start in:".
7. Right-click on the highlighted text and select "Paste".
8. When you are finished pasting in the new directory, click "Apply". You should have an image similar to that shown in Figure 16.
9. Click "OK" to close the window.
10. Open Freemat with this icon and you should have the working directory set to your actual working directory.

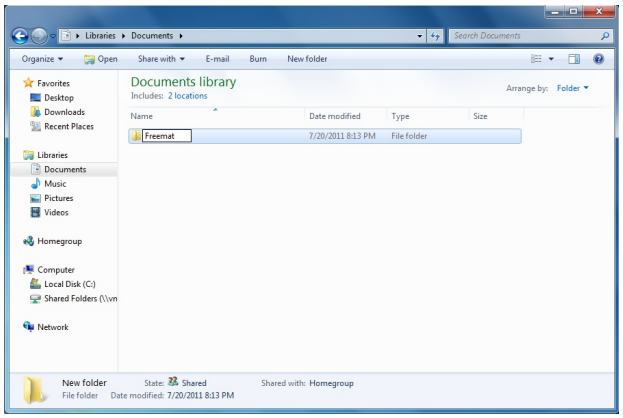


Figure 13: Adding a folder called "Freemat" under the "Documents" folder in a Windows 7 system.

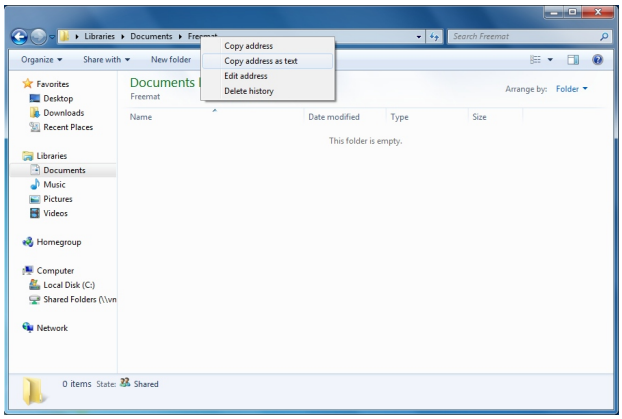


Figure 14: Copying the working directory filename in Windows 7.

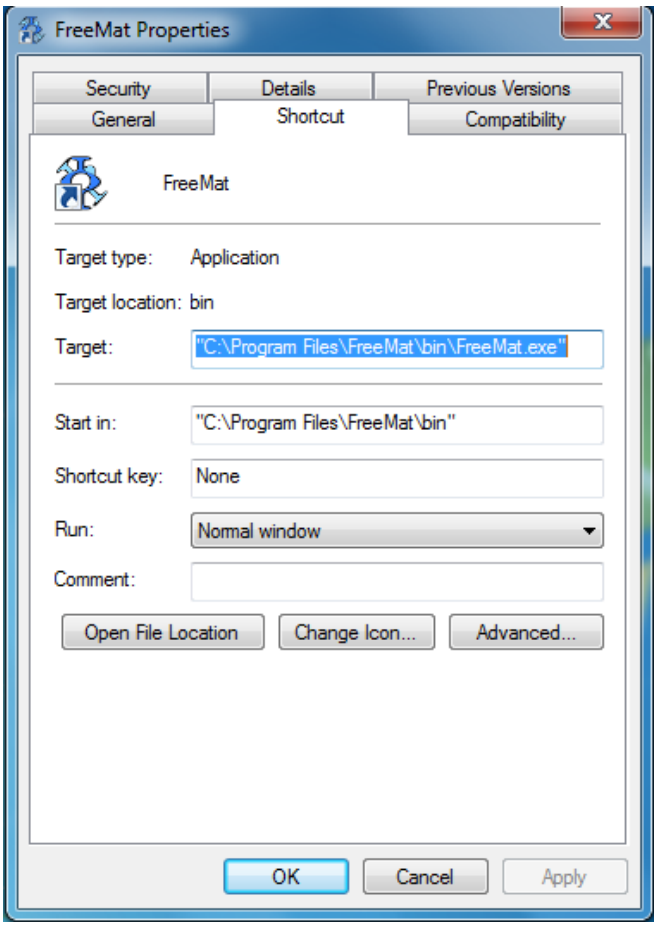


Figure 15: This is the "Properties" window for the Freemat icon.

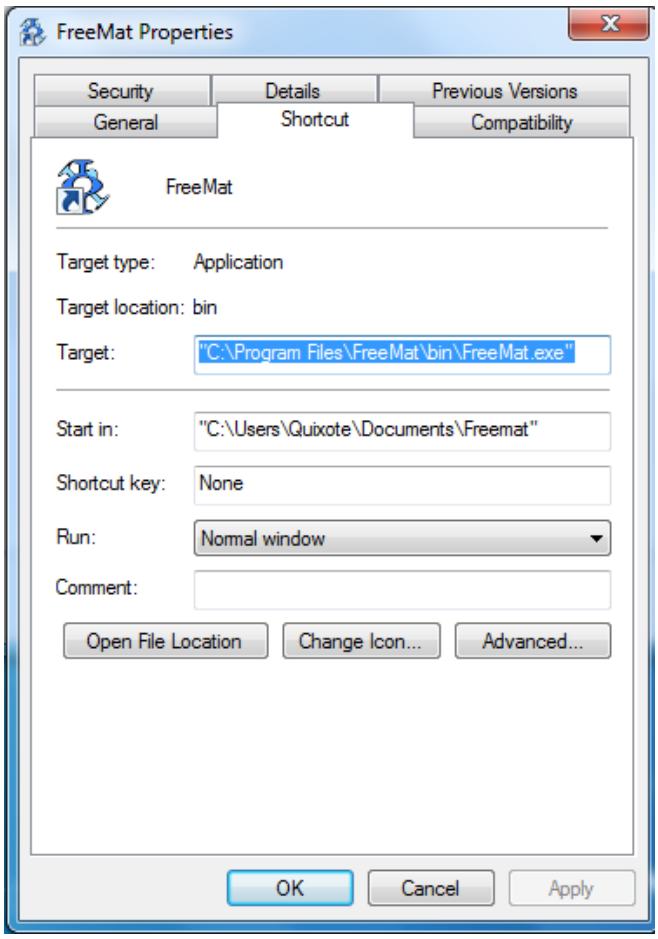


Figure 16: This is the "Properties" window for the Freemat icon with the "Start in" directory changed to the Freemat folder under the user's "Document" directory.

Topic 1.3: Setting the Path List

In order for Freemat to operate, it has certain files it needs to use. These are setup under a variable called *path*. The path is nothing more than a list of directories into which you can store items related to Freemat, such as scripts and functions, and Freemat will look through all of them, in the order you set, to find whatever scripts or functions you wish to use.
You can view the path list at any time simply by typing *path* on the command line.

Example - Viewing and Setting the Path

At the command prompt, type path and press <Enter>.

```
--> path
/home/gary/Freemat
/home/gary/Freemat/Data
/home/gary/Freemat/Freemat Training
```

```
/home/gary/Freemat/Freemat_images
/home/gary/Freemat/gary
/home/gary/Freemat/myFunctions
/home/gary/Freemat/myScripts
```
This is how the path looks on one of my computers running Linux. Here's another example, but on a system running Windows 7.
```
--> path
C:/Users/Quixote/Documents/Freemat
Y:/Brain_Public/myScripts
Y:/Brain_Public/myScripts/525.414_Probability
Y:/Brain_Public/myScripts/525.416_Communication_Systems_Engineering
Y:/Brain_Public/myScripts/525.441 Data and Computer Comms
Y:/Brain_Public/myScripts/525.722_Wireless_and_Mobile_Comms
Y:/Brain_Public/myScripts/605.471_Principles_of_Datacomms
Y:/Brain_Public/myScripts/Analog_Modulation
Y:/Brain_Public/myScripts/Digital_Modulation
Y:/Brain_Public/myScripts/Digital_Signal_Processing
Y:/Brain_Public/myScripts/Line_Encoding
Y:/Brain_Public/myScripts/Networking
Y:/Brain_Public/myScripts/Numeric_integration_and_differentiation
```
The only difference is that each lists a different directory tree.

Where is the path string stored? You may have noticed that Freemat remembers the path even if you exit the program and restart it. You might be wondering where the string is stored. In a Windows system, it's kept as a string value in the Windows registry. Specifically, it's kept in an entry under **HKEY_CURRENT_USER/Software/Freemat/Freemat v4.0/interpreter**. In Linux, it's kept as part of a configuration text file under the user's home directory. Specifically, it's kept as an entry in a file called "FreeMat v4.0.conf" in the folder **/home/<username>/.config/Freemat.**

The path is nothing more than a variable storing a string. The string is the list of all the different folders. You can see this by storing the variable *path* into a different variable, as shown below.

```
--> y=path
y =
/home/gary/Freemat:/home/gary/Freemat/Data:/home/gary/Freemat/Freemat
Primer:/home/gary/Freemat/Freemat
Training:/home/gary/Freemat/Freemat_images:/home/gary/Freemat/gary:/hom
e/gary/Freemat/myFunctions:/home/gary/Freemat/myScripts
--> length(y)
ans =
 230
```

Caution - The Number of Entries on the Path List

This applies to all operating systems. As of this writing, if there is only one entry in the path, Freemat will either return an error for the path list (Linux) or simply a blank line (Windows, all versions) when entering the *path* command. However, Freemat will still access this one folder properly when searching for any scripts or functions you may have stored in that one directory.

Topic 1.4: The Command Window

This is it. This is where it happens. The Command Window provides a prompt which is typically two dashes followed by a right-pointing carrot. It's the "-->" you see in the Command Window.
You will use the Command Window to perform, well, everything. It can be basic operations, such as simple addition or subtraction, or it can be lengthy, one-of-a-kind operations that uses many functions. I use Freemat as a calculator when I don't have my trusty Hewlett-Packard HP50G handy. Execution of any command in the Command Window simply requires entering the operation and hitting the <Enter> key.

Example - Entering Commands

Add 3 + 2

```
--> 3+2 <Enter>
ans =
 5
```

Multiply 5 x 3.1415

```
--> 5*3.1415
ans =
   15.7075
```

During a recent Bowie Baysox baseball game, newcomer Bryce Harper of the Harrisburg Senators (a farm team for the Washington Nationals) came up to hit. His hitting average at that time was 0.202. He hit a double out to center field. The next time he came up to hit, his average was 0.218. Using this information, we can run a few basic calculations with Freemat to figure out how many at-bats he's had, and how many hits he's had.

For the first at bat: $\frac{H}{AB}=0.202$

For the second at-bat: $\frac{H+1}{AB+1}=0.218$

Solving for AB, we wind up with this: $0.218AB-0.202AB=1-0.218$, $AB=\frac{1-0.218}{0.218-0.202}$

Using Freemat:

```
--> (1-0.218)/(0.218-0.202)
ans =
   48.8750
```

Considering rounding, I'm going to assume that he's had 49 at-bats. Using the first equation, we can now solve for his hits:

```
--> 49*.202
ans =
    9.8980
```

This means that he's had 10 hits at 49 at-bats.

When you press <Enter>, Freemat performs the calculation, displays the result (if that is what you want... more on that below), and provides a new command prompt ready for another operation. One thing to bear in mind is that, for the most part, the way you enter commands in the Command Section is exactly the same way you'll enter them when creating scripts and functions.

For the most part, the way you enter commands in the Command Window is exactly the same way you'll enter them when creating scripts and functions.

Topic 1.4.1: Seeing the Results (or Not)

Freemat allows you to use the semicolon (;) as a line end. When you do so, you're telling Freemat that that is the end of the line. You can (if you so desire) put a bunch of commands on one line and just put semicolons between them.

Example - Performing Multiple Commands on One Line

```
--> x=5.21; y=6.7; z=x*y
z =
34.9070
```

I do not recommend this, although there is absolutely nothing wrong with it. Look at the example above. Having all of the commands on one line makes it more difficult to understand. I recommend one line per command just to keep things neat and tidy. This is especially true if you are writing code (namely scripts and functions) that others will have to read and understand.
You can also choose whether you want to see the results of an operation using the semicolon (;). If you want to see the result immediately, merely enter the operation with nothing at the end. If you don't want to see the result, add a semicolon at the end of the operation.

Example - Using the Semicolon (;)

If you wish to see the results immediately, do NOT use the semicolon:

```
--> x=7*3
x =
21
```

If you don't want to see the results, end the statement with a semicolon:

```
--> x=7*3;
-->
```

In either case, the result is the same. Freemat performs the calculation, enters the result into the variable "x", and stores it in memory. The only difference is whether you want to see the result of the

calculation right now. The semicolon really becomes handy when you start creating your own scripts (essentially short programs) so that you won't be bombarded with intermediate calculation results. Otherwise you could easily see hundreds or thousands of calculations, perhaps just in order to see the final calculation of one number.
Even if you use the semicolon at the end of the statement, you can still see the result. There are several ways to see the result of a calculation already performed. Here are a few:

- Enter the variable onto the command line and press <Enter>.
- Use the "Workspace" window. Simply click on the "Workspace" tab at the top of the narrow window on the left of the screen.

Example - Viewing a Variable Value using the Command Line

To view the value of a variable, enter the name of the variable on the command line and press <Enter>.

```
--> x=7*3;
--> x
ans =
21
```

Topic 1.4.2: How Many Decimal Points Do You Want?

Freemat uses the **format** command (*FD*, p. 343) to show you a different number of decimal points within the Command Window for numeric variables. If you want to see a lot of decimal places (14, to be precise), use the command **format long**. If you don't, use **format short** (for 4 places).

By default, Freemat uses the *short* format.

Example - Setting the Number of Decimal Places with the *Format* Command

```
--> format short
--> pi
ans =
3.1416
--> format long
--> pi
ans =
3.14159265358979
```

Topic 1.4.3: Understanding Variables

A *variable* is simply a single character or a set of characters used to store some data. A variable name can contain letters, numbers, and/or an underscore. Note: The variable name must start with a letter. Also, variable names are case-sensitive. The variable *X* is different from the variable *x*.
In general, a variable can store:

- a number.
- a string.

- a matrix of numbers, strings and/or other matrices. Matrices will be covered in Topic 3: Matrices & Arrays on page 42.
- a pointer to a function, which is the anonymous function, explained in Topic 4.7: The Anonymous Function on page 67.

Numbers can be one of many different types (see *Topic 1.4.3.1: Variable Types*). There are both signed and unsigned integers, single and double precision floating-point numbers, and single and double precision complex numbers. There is only one type of string. Or, as they say, "A string is a string is a string."

A quick note on the allowable length of a variable name. In Freemat, I've made a variable name up to 73 characters long. Frankly, I don't know how long variable names can be in Freemat, but the length of a variable will almost certainly not be a problem. If you're using variable names longer than 73 characters, you need a day job.

To assign a value to a variable, use the equals (=) sign. A quick example is x = 5. The **x** is the variable and **5** is the number being stored.

A way to view all of the variables currently in use is the **who** (FD, p. 241) function. Simply type *who* at the command prompt and hit <Enter>.

```
--> x=1;
--> who
Variable Name      Type   Flags             Size
              x     int32                    [1 1]
```

In this case, only one variable has been defined. If there had been more than one, it would be listed here as well.

Example - Assigning a Value to a Variable

To assign a value to a variable, type in the variable name, followed by the equals (=) sign, then the value (a number or a string). The following are some examples of assigning numeric values and strings to variables, followed by the **who** command to show their variable types.

```
--> x=5;
--> y=491.2768;
--> s='This is a string';
--> s2=['Part of a string ' num2str(2) ' part of another string']
s2 =
Part of a string 2 part of another string
--> who
  Variable Name       Type   Flags             Size
            ans    double                   [1 1]
              s      char                   [1 16]
             s2      char                   [1 41]
              x    double                   [1 1]
              y    double                   [1 1]
-->
```

You'll learn more about the different types, both numeric and strings, later in this book.

Topic 1.4.3.1: Variable Types

This topic is covered in the Freemat Documentation starting on page 269. Each variable has a type, which is based on the type of number or alphanumeric string it stores and how many bits it uses to store

that number or string. The variable types, which are also functions to create those types, are:

- **int8** - a signed, 8 bit integer. Allowable values are -128 to 127. This function, like all of the integer functions (int16, int32, int64), is periodic. This means that, if a value outside of its limits (-128 to 127) is entered, it wraps around to the other end of the range. For example, int8(128) = -128 and int8(-129) = 127. This is shown in Figure 17 below.

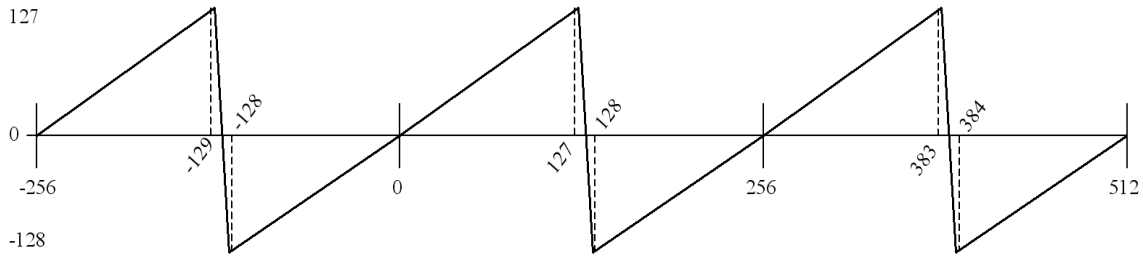


Figure 17: Graph of the 8-bit integer (int8) function

Example - The int8 Variable Type

```
--> int8(53)
ans =
53
--> int8(130)
ans =
-126
```

Note how the value of 130 wraps around to a value of -126 when entered in the **int8** function.

```
--> int8(33.98)
ans =
33
-->
```

Any non-integer will be rounded down, meaning any part after the decimal point will be removed.

- **int16** - a signed, 16 bit integer. Allowable values are -32,768 to 32,767. The int16 function is also periodic in a similar fashion as the int8 function. For example, int16(32,768) = -32,768 and int16(-32,769) = 32,767.
- **int32** - a signed, 32 bit integer. Allowable values are -2,147,483,648 to 2,147,483,647. *This is the default Freemat type for integers. This means that, if you create a variable that is only an integer without specifying it, it will be created as a 32-bit integer (int32).*

Example - Creating an Unspecified Integer

This will show how a variable holding an integer value, but without being specified as to what type of integer it is, will be created as an int32 type.

```
--> x=5;
--> who
Variable Name      Type   Flags             Size
            x     int32                   [1 1]
```

```
-->
```

- **int64** - a signed, 64 bit integer. Allowable values are -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- **uint8** - an unsigned, 8 bit integer. Allowable values are 0 to 255.
- **uint16** - an unsigned, 16 bit integer. Allowable values are 0 to 65,535.
- **uint32** - an unsigned, 32 bit integer. Allowable values are 0 to 4,294,967,295.
- **uint64** - an unsigned, 64 bit integer. Allowable values are 0 to 9,223,372,036,854,775,808.
- **float** - a signed, 32 bit floating point number. Allowable values are -3.4×10^{38} to 3.4×10^{38}. This is also called a *single precision floating point* number.
- **double** - a signed, 64 bit floating point number. Allowable values are slightly less than -1.79×10^{308} and slightly more than 1.79×10^{308}. This is also called a *double precision floating point number*. *This is the default Freemat type for floating point numbers.*
- **complex** - a signed, 32 bit complex floating point number. I *believe* that the values for both the real and imaginary parts are single precision floating point numbers.
- **dcomplex** - a signed, 64 bit complex floating point number. Just as with the single precision complex numbers (*complex*), I *believe* that each part of the complex number, real and imaginary, is represented by a double precision floating point number.
- **string** - any combination of letters, numbers, and/or special characters. A string variable can be a single character long, or up to 65535 characters long. A string variable is entered using single quotes on each end. For example, `a = 'hello'` enters the string *hello* into the variable *a*. See *Topic 1.8: Strings* below.

Topic 1.4.3.2: Binary Types

It is possible to manipulate binary numbers, but it cannot be done directly. Instead, it must be done using the decimal and integer representations of the binary number, for the actual calculations, then displayed using string versions of the binary numbers.

Topic 1.4.3.3: Displaying Binary Numbers

Binary numbers can either be displayed as a vector, meaning its still a number just in binary form, or as a string. Freemat provides two functions, **int2bin** and **dec2bin**, that convert either integer numbers or decimal numbers, respectively, into a binary representation.
int2bin(x,n): This function actually creates a binary number. This calculates the binary value of the integer number x with n bits used to represent the binary version of x. The result is a vector that has a length of n. The most signicant bit is index 1, and goes to the least significant bit (LSB) at index n.
dec2bin(x): This function creates a string that is the binary value of the input number. The output is not a binary number, but a string that is the binary representation of the original number.

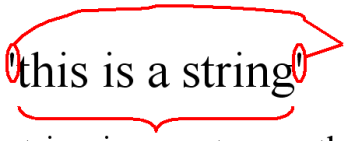
Topic 1.4.3.4: Calculating with Binary Numbers

There are three binary operations. These are:

- bitor
- bitand
- bitxor

Topic 1.5: Strings

In Freemat, a string is nothing more than a matrix consisting of letters, numbers and special characters. To create a string, put any combination of alphanumeric (A, B, C, a, b, c, 1, 2, 3, etc) or special characters (!, @, # , $, %, etc) between single quotes. Note that a space is considered a string character. Freemat treats a string as a vector of size [1 n], where n is the number of characters (including spaces) in the string. Note that this does not include the quote marks on each end when making strings.



A string is enclosed by single quote marks.

A string is a vector, so this string will have a size of [1 16]. Remember: A space is considered a character.

Figure 18: Graphic showing string structure in Freemat

Example - Strings in Freemat

Here are how several strings appear in Freemat, as well as a **who** command to show their vector length, which is a count of each character in each string.

```
--> clear all
--> a='here is a string';
--> b='yet another string to ponder';
--> c='hello';
--> who
Variable Name       Type   Flags             Size
            a     string                  [1 16]
            b     string                  [1 28]
            c     string                  [1 5]
-->
```

Topic 1.5.1: Creating a String

You can create strings using the following methods:

- place alphanumeric and/or special characters between single quotes, as shown in Figure 18.
- the **string** function (FD, p. 259) or the **char** function (FD, p. 248) to convert from ASCII codes into strings. If you want to understand ASCII codes more, jumpt to Topic 4.8.4: Printing Special Characters on page 71.
- the **num2str** function, which converts numbers into strings.
- create a one-row matrix that consists of one or more strings, whether using the single quote method, the **string** function, or the **num2str** function.

Example - Creating Strings

This first example just uses single quotes around text to create a string.

```
--> x='It was a dark and stormy night.'
```

```
x =
It was a dark and stormy night.
--> y='To be or not to be.  That is the question.'
y =
To be or not to be.  That is the question.
```
We can also use the **string** function to create a string based on ASCII codes.
```
--> x=[70 114 101 101 109 97 116];
--> string(x)
ans =
Freemat
```
The advantage of the **string** function is that it does not require that you actually have the particular key on your keyboard in order to use that character. For example, my keyboard is a standard US keyboard. It does not provide for the degree (°) symbol, for example. But so long as I know what the ASCII code is, I can add it to a string. The ASCII code for the degree symbol is 176. This example also shows the concatenation process (putting two or more strings together).
```
--> x=['The temperature yesterday was 105' string(176) 'F.']
x =
The temperature yesterday was 105°F.
```
We can also create a string using the values stored in variables. This uses the **num2str** function.
```
--> temp=105;
--> x=['The temperature yesterday was ' num2str(temp) string(176) 'F.']
x =
The temperature yesterday was 105°F.
```

Topic 1.5.2: Concatenating Strings

To concatenate two or more strings together, put them together by putting them within a matrix.

Example - Concatenating Strings

We can start with two basic strings created using the single quotes, them concatenate them.
```
--> s1='To be or not to be.';
--> s2='That is the question.';
--> s=[s1 s2]
s =
To be or not to be.That is the question.
```
Note that we do not have a space between the two sentences. We can fix this by either putting a space at the end of the first string, the beginning of the second string, or by placing a space in the concatenation itself, as shown here:
```
--> s1='To be or not to be.';
--> s2='That is the question.';
--> s=[s1 ' ' s2]
s =
To be or not to be. That is the question.
```
All we did was to put a single space between two quotes in between the two strings.
As noted in the previous section, we can also create strings using characters not available on your keyboard. For example, the math "division" symbol (ASCII code 247) is typically not available directly from the keyboard. But we can still create it either with the **string** function or the **char** function.

```
--> s1='20';
--> s2='4';
--> s3='5';
--> s=[s1 string(247) s2 '=' s3]
s =
20÷4=5
```

Or we could do this slightly differently using the actual math involved and the **num2str** function.

```
--> s1=20;
--> s2=4;
--> s3=s1/s2;
--> s=[num2str(s1) string(247) num2str(s2) '=' num2str(s3)]
s =
20÷4=5
```

Topic 1.6: Built-In Variables

Freemat provides several different constants that are built in (*FD*, p. 201). The more useful ones (for me, at least) are:

- **e** - the base of the natural logarithm. Value is approximately 2.71828.
- **i** - the imaginary operator.
- **inf** - infinity. Note that this is only used for *float* and *double* variable types. If you try to use this on an integer-type variable, you may get strange results. You've been warned.
- **pi** - the ratio of the circumference of a circle to its diameter. The ever-wonderful 3.1415926..., ad infinitum.

- **ans** - You may have already noticed the variable "ans". This variable is used by Freemat to display results. However, if you specify a variable, such as "x = 1+1", in which to direct the result of your calculation, then the result of "1+1" will be stored in that variable (in this case "x") rather than "ans".

The variable "ans" is used to store the results of a math operation any time you do not specify the variable yourself or to display calculation results.

Example - The Built-In Variables

Here are some examples of different built-in variables.

```
--> format long
--> e
ans =
2.71828182845905
--> i
ans =
0.0000000 + 1.0000000i
--> inf
ans =
1.#INF0000000000
--> pi
ans =
```

```
3.14159265358979
```
Here are some examples of the "ans" variable.
```
--> 5*pi
ans =
   15.7080
--> 1:10
ans =
  1  2  3  4  5  6  7  8  9 10
--> 1:3:13
ans =
  1  4  7 10 13
--> 9/11
ans =
    0.8182
-->
```
You can also use the "ans" variable for further calculations. Note, however, that unless you assign the result to another variable, the new result will be placed in "ans".
```
--> 9/11
ans =
    0.8182
--> ans*4.5
ans =
    3.6818
-->
```

Topic 1.7: Using Built-In Functions

Freemat provides a plethora / potpourri / myriad / bunch of functions. But what is a function? A function is nothing more than an algorithm or process to produce a result. For example, look at the **sin()** function (*FD*, p. 193). It takes an input, in this case ***x***, and calculates the sine of that input. It then provides a return for the function. The return can be a number, just as with the already-mentioned **sin()**, or it can be a display, as with the **plot()** function (*FD*, p. 514). In order to use functions properly, you need to know:

- What arguments does the function need and in what order? For example, take the function **cumsum(x,d)** (*FD*, p. 219). The ***x*** refers to an array, and the ***d*** refers to a dimension within that array. It's important that when you invoke the function, such as typing it onto the command line, that you put the proper arguments into the proper place. In this case, if you write it as **cumsum(d,x)**, you'll probably get an error, or at the very least you won't get the proper result. Check the Freemat documentation. From the Command Window, click **Help -> Online Manual**, press the F1 key to bring up the documentation, or download the documentation from http://freemat.sourceforge.net/FreeMat-4.0.pdf.
- Do the arguments have to be a certain type? Some functions want integers, others want floating point variables, and still others want strings. For example, if you try to perform the function **sin('string')**, you will definitely get an error. The **sin** function wants a floating point argument, not a string. Make sure that you give the function what it wants, not what you *think* it wants. How to be sure? Check the Freemat documentation.
- Do the arguments have to be a certain units? For example, the trignometric functions **sin**, **cos**, and **tan** need their arguments to be in units of radians. If you put in 30^{o} as the variable into the

sin function and expect to get a value of 0.5, you will get the wrong answer.

- What type of output does the function provide? For example, if you type into the command line:

```
t=1:128;
x=sin(2*pi*t/32);
y=plot(x)
```

What you will get is a plot of the function *y* as well as the following within the command window:

```
y =
100002
```

What does *y = 100002* mean? Who knows. But the **plot** function provides a graphical output. It's not meant to return a number.

- Does the function already exist? Maybe. An easy check is to use the **which** command (FD, p. 241). To use this command, type it into the command line followed by a space and then the command that you would like to check. The output is either the path to the function, or a statement that it does not exist. The following are some examples. The first one is a Freemat function, the second one is one that does not exist, and the third one is one of my own functions.

```
--> which diff
Function diff, M-File function in file
'/usr/share/freemat/toolbox/general/diff.m'
--> which myFunc
Function myFunc is unknown!
--> which labelSet
Function labelSet, M-File function in file
'/home/gary/Freemat/myScripts/labelSet.m'
```

You can also check the Freemat Documentation, either the actual PDF manual or the online documentation.

Topic 2: Working with Math

Topic 2.1: Basic Math Operations

Basic math operations consist of add (+), subtract (-), multiply (* ; this is an asterisk, if you cannot tell), divide (/), and exponent (^). The exponent is discussed in more detail in *Topic 2.4: Exponentials and Logarithms*.

Example - Basic Math Operations

Here are examples of each of the five basic math operations.

```
--> 5.4+3.8
ans =
9.2000
--> 5.4-3.8
ans =
1.6000
--> 5.4*3.8
ans =
20.5200
--> 5.4/3.8
ans =
1.4211
--> 5.4^3.8
ans =
606.8709
```

Note that in each of these examples, I did not use the equals ("=") sign. **The only time you use the equals sign is when you want to assign a value to a variable.**

Topic 2.2: Precedence

When performing math operations, Freemat uses the standard mathematical system of precedence. This means that Freemat will perform some operations before others. The system of precedence[1] is:

- Calculations in parentheses
- Exponential calculations
- Multiplication and division
- Addition and subtraction

This means that, given how a calculation is written, a calculation written using the same variables or numbers, may have two, different outcomes. Here are two examples:

```
--> 4+5*3
ans =
 19
```

1 An acryonym used to remember the order is PEMDAS, or "Parenthese, Exponents, Multiplication, Division, Addition, Subtraction". A mnemonic used is "Please Excuse My Dear Aunt Sally". From Wikipedia, "Order of operations", http://en.wikipedia.org/wiki/Order_of_operations, 31 July 2008.

```
--> (4+5)*3
ans =
 27
```

In the first calculation, Freemat will, according to the rules of precedence, perform the multiplication (5*3) before the addition. Thus, it has an intermediate step of, first, the multiplication, which results in 15, followed by the addition of 4. This results in a total of 19.
In the second calculation, Freemat performs the addition first. This is due to the parentheses. Thus, it first adds 4+5, giving an intermediate value of 9, which is then multiplied by 3. This results in a final value of 27.

Example - Looking at Precedence and Negative Number Math

At one point in the movie "Stand By Me", the character Jaime Escalante (played by Edward James Olmos, one of my favorite actors) walks around the classroom telling them over and over "A negative times a negative equals a positive." He even has the class repeat it multiple times. The same holds true with Freemat; a negative times a negative equals a positive. It's basic math. But you have to understand what Freemat considers to be a negative number. Here are some examples.

```
--> -1*-5
ans =
 5
```

In this first multiplication, we multiplied -1 times -5 and wound up with 5. That's what we'd expect. In this next one, we take the square of -5.

```
--> -5^2
ans =
 -25
```

We wind up with -25, not 25 as we would expect. That's because Freemat first does the exponential function (5^2) then does the subtraction. Again, the exponent has higher precedence than subtraction. To correct this, we'll use parentheses (the highest precedence) to ensure that the number is seen as "negative five".

```
--> (-5)^2
ans =
 25
```

To help prevent this in the future, I recommend putting parentheses around any negative numbers. This will ensure that Freemat sees them as negative numbers, and treats them that way.

Topic 2.3: Sum, Products, Cumulative Sums & Products, and Factorials

Freemat provides several functions that allow you to efficiently calculate the sum, product, cumulative sum, and cumulative product of an array of numbers, as well as the factorial of a number. These are:

- **sum(x)** - This is the sum of the array x. This creates a scalar number that is the sum of all of the numbers in the array. (FD, p. 216)
- **prod(x)** - This is the product of all of the numbers in the array x. This creates a scalar number. (FD, p. 212)
- **cumsum(x)** - This is the cumulative sum of the array x. This creates an array, with each point of the array the sum of each number in the array that came before it. (FD, p. 199)
- **cumprod(x)** - This is the cumulative product of the array x. This creates an array, with each

point of the array the product of each number in the array that came before it. (FD, p. 198)

- **gamma(x)** - This is an integral function. We'll discuss its primary purpose later. However, for integer values of x this function can be used to calculate the factorial of a number. With this function, x! = gamma(x+1). (FD, p. 326)

Example - Cumulative Sum, Products and Factorials

```
--> x=rand(1,10)
x =
Columns 1 to 5
0.9386    0.9957    0.1178    0.9420    0.3271
Columns 6 to 10
0.5635    0.2212    0.6741    0.9849    0.2905
--> y=cumsum(x)
y =
Columns 1 to 5
0.9386    1.9342    2.0521    2.9941    3.3212
Columns 6 to 10
3.8847    4.1059    4.7800    5.7649    6.0554

--> y=cumprod(x)
y =
Columns 1 to 5
0.9386    0.9345    0.1101    0.1037    0.0339
Columns 6 to 10
0.0191    0.0042    0.0029    0.0028    0.0008
```

Here's an example of calculating 5! (5 factorial):

```
--> gamma(5+1)
ans =
120
```

Topic 2.4: Exponentials and Logarithms

When numbers are raised to a power, the power is the exponent. For example, for x^2 the 2 is an exponent; x is the base of the number. There are two ways to use exponents. The first is with the power symbol (^). The second is with the power and exponential functions. The power symbol is typically easiest. Simply type a number or variable, the ^ symbol, and then the power to which you want to raise it. If you're raising each number in an array of values, then use the form ".^" (put a period in front of the carat). The power function has the following syntax:

```
power(x,y)
```

where: x = the number to be raised.
y = the exponent.

Example - Raising a Number or Variable to a Power

```
--> x=5.6;
--> x^2.5
ans =
```

```
74.2113
--> 8^3
ans =
512
--> power(8,3)
ans =
512
--> power(2,5)
ans =
 32
--> power(1:10,2)
ans =
   1   4   9  16  25  36  49  64  81 100
```

We can also do this with the exponential operator (^). But when using math functions on any array, we have to precede it with a period (.). We'll cover that in more detail when we cover arrays and matrices in Topic 3: Matrices & Arrays.

```
--> (1:10).^2
ans =
   1   4   9  16  25  36  49  64  81 100
```

Common bases for exponential functions are 10 and *e*. Freemat provides two functions for the natural exponent. These are:

- **exp(x)** - This is exactly the same as if you had typed **e^x**. What's the difference? At one point, on an older computer running Freemat 3.6, there was a difference in the 14th digit. Now, there's no difference. Use whichever of the two forms you prefer. I used to prefer the **e^x**. Now, I mainly use the **exp()** form. (FD, p. 166)
- **expm1(x)** - This is useful for calculating the value of exp(x)-1 when x is a small value. This is one of those times when it's better to do it with the function than do-it-yourself. (FD, p. 167)

Example - Exponential Functions

The first example shows the use of the exponent function exp(). It compares it to using the power function with the base of the natural logarithm, e.

```
--> exp(4)
ans =
54.5982
--> e^4
ans =
54.5982
--> expm1(0.0001)
ans =
1.0001e-004
```

Logarithms: The logarithm function is a way to calculate the exponent of a number of a certain base. Common bases for logarithm functions are 10 and the natural base *e*. Freemat provides four logarithmic bases. These are:

- **log10(x)** - calculates the log, base 10, of the variable x. (FD, p. 171)
- **log2(x)** - calculates the log, base 2, of the variable x. (FD, p. 172)

- **log(x)** - calculates the log, base *e*, of the variable x. This is oft-times called the natural logarithm function and has the form ln(x). (FD, p. 170)
- **log1p(x)** - calculates the logarithm, base e, for the argument plus one. In other words, calculate the natural logarithm for x+1. (FD, p. 172)

Example - Logarithm Functions

```
--> log10(10)
ans =
1
--> log10(25)
ans =
1.3979
--> log(10)
ans =
2.3026
--> log(e)
ans =
1
--> log2(2)
ans =
1
--> log2(16)
ans =
4
--> log2(10)
ans =
3.3219
```

Calculating the Logarithm to an Arbitrary Base: There's a common way, mathematically, to calculate the logarithm of a variable for any base. Simply calculate the log of the number to any base, then divide that by the log, to the same base, of the number whose base you want to calculate. Clear as mud? Let's try an example.

Example - Calculating the Logarithm of a Number to an Arbitrary Base

Calculate the log, base 3, of the number 5. In mathematical parlance, this would be written as $\log_3(5)$.

```
--> format long
--> log(5)/log(3)
ans =
1.46497352071793
```

Calculate the log, base 6.2, of the number 16.

```
--> log(16)/log(6.2)
ans =
1.51960198297685
--> log10(16)/log10(6.2)
ans =
1.51960198297685
```

The last example shows that any base logarithm can be used to calculate the log of a number to an arbitrary base. The next example shows that this method of calculating the log using this method

works the same as if you had calculated using the actual base logarithm.

```
--> log(5)
ans =
1.60943791243410
--> log10(5)/log10(e)
ans =
1.60943791243410
```

Topic 2.5: Trignometric Functions

Freemat provides every trignometric function you will ever need. Which means all of them. This includes standard and hyperbolic, inverse standard and inverse hypebolic functions, as well as ones that will accept degrees instead of radians. As a matter of fact, almost every single function mentioned in the Freemat Documentation between pages 143 and 163 deals with trignometric functions of one form or another.

Table of Trignometric Functions

Standard (use radians)	Inverse Standard (return radians)	Hyperbolic	Inverse Hyperbolic	Standard Degree (use degrees)	Inverse Standard Degree (return degrees)
sin()	asin()	sinh()	asinh()	sind()	asind()
cos()	acos()	cosh()	acosh()	cosd()	acosd()
tan()	atan() or atan2()	tanh()	atanh()	tand()	atand()
sec()	asec()	sech()	asech()	secd()	asecd()
csc()	acsc()	csch()	acsch()	cscd()	acscd()
cot()	acot()	coth()	acoth()	cotd()	acotd()

In order, the standard functions require the use of angles in radians. The inverse standard functions return radians. The hyperbolic functions use straight numbers and return the same. The standard degree functions use angles in degrees (as opposed to radians).

Example - Calculating a Driveway Slope

My driveway is on a slope. Out of curiousity, I decided to measure the actual slope of the driveway. I used a 24" level along with a tape measure. With the level providing the horizontal distance and the tape measure measuring the vertical, I found that my driveway rose 2.75" over the 24" distance of the level. Using Freemat's trignometric capabilities, this calculates as:

```
--> atand(2.75/24)
ans =
    6.5366
```

This means my driveway rises at an angle of 6.5 degrees. Most people who work on roads use units of "gradians", which runs from 0% (flat) to 100% (straight up and down). There are 100 gradians in 90 degrees, which shortens to a conversion factor of 10/9.

```
--> atand(2.75/24)*10/9
```

```
ans =
    7.2629
```
My driveway is thus on a 7.3% slope.

Example - Calculating the Great Circle Distance

As you're probably aware, the Earth is not flat; it's spherical. Well, okay, it's mostly spherical. The best shape that it can technically be called is a geoid, meaning it's shaped closer to an M&M candy than it is a truly round sphere.
When calculating the distance between two points on the Earth, you can either store lots of data showing its true shape, or you can assume its a sphere and make the calculation much smaller. We're going to use this second method to find a rough approximation for the distance between two points. Before we do that, let's make sure you understand what is meant by "Great Circle". This is the distance over the globe. Let's start by looking at a "straight" route between two points using a normal, flat map, as shown in Figure 19. This is how the route between Los Angeles, USA, and Ankara, Turkey, would appear on a flat map.

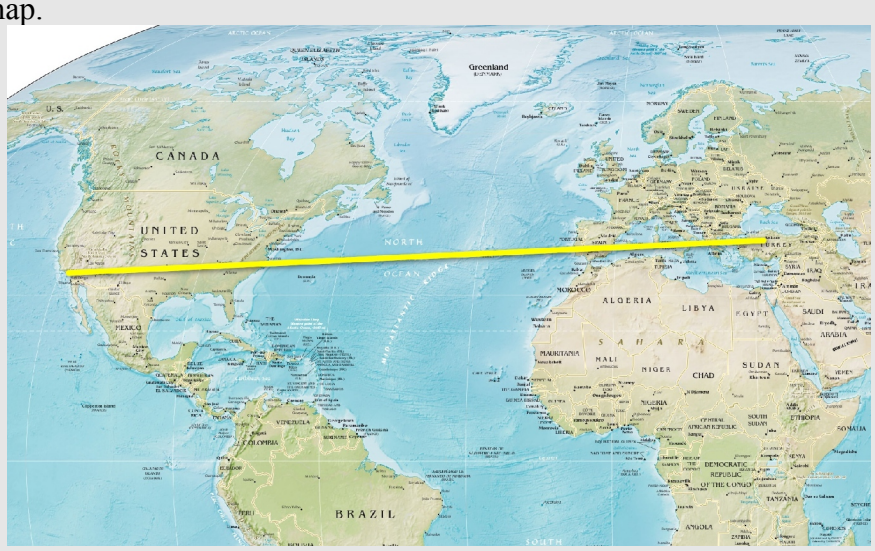


Figure 19: How the "shortest" route between Los Angeles, USA, and Ankara, Turkey, appears when looking at a flat map. (Map from CIA World Factbook.)

Now look at it on a actual globe image with the route between the same two points.

Figure 20: How the shortest route between Los Angeles, California, USA, and Ankara, Turkey, appears on an actual globe. This is using a short piece of string stretched taut between the two cities. Note the difference between this route and the "straight" route shown above. Instead of the route going across the widest part of the Atlantic Ocean, the shortest route passes through the Arctic Circle. The distance along this route is known as the "Great Circle" distance.

This "Great Circle" distance can be calculated using basic trignometry. Look at Figure 21.

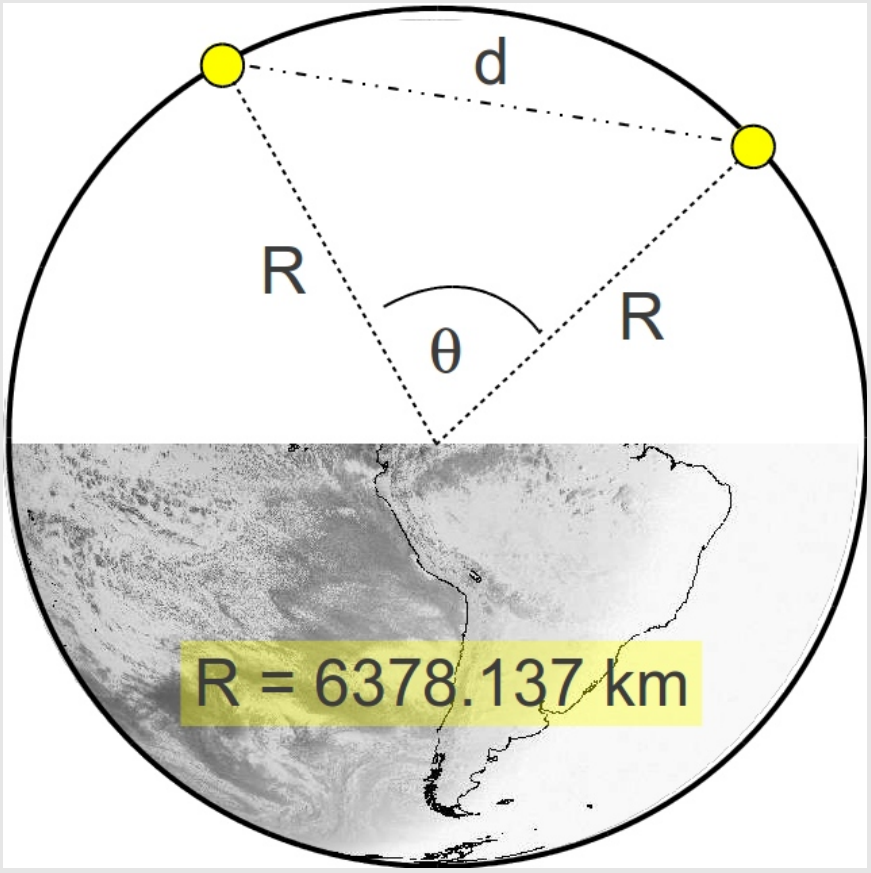


Figure 21: The Great Circle distance between two points can be calculated using basic trignometry. Starting with the latitude / longitude, then calculating the three-dimensional Cartesian coordinates, you can calculate the Cartesian distance between the two cities. Once you know this distance "d", you can calculate the angle θ. If the angle θ is in radians, then the Great Circle distance is Rθ.

We start by calculating the three-dimensional coordinates of each city using their respective latitude (the angle from the equator) and longitude (the angle from the Prime Meridian). From these Cartesian coordinates, we can calculate the Cartesian distance "d". The combination of the Cartesian distance "d" and the two lines formed between the respective cities and the center of the earth form a triangle. We can calculate the angle "θ" at the vertex of these two lines. If the angle is in radians, then the Great Circle distance is Rθ. We won't go through all of the math, but this calculation can be boiled down to one (fairly long) equation. This equation can combine radian and degree trignometric functions. We'll use the degree functions for the latitude and longitude, since those are the most common units. We'll use the radian version of the inverse cosine since that's part of the calculation for calculating the distance. The equation for calculating the Great Circle distance is as follows:

$$d_{GC} = R_E \cos^{-1}\left(\cos\left(Lat_{P1}\right)\cos\left(Lat_{P2}\right)\cos\left(Lon_{P1} - Lon_{P2}\right) + \sin\left(Lat_{P1}\right)\sin\left(Lat_{P2}\right)\right)$$

where:

d_{GC} = the Great Circle distance

R_E = average radius of the Earth, which is 6378.137 km

Lat_{P1} = latitude of point 1. Points to the south of the equator are -.
Lon_{P1} = longitude of point 1. Points to the west of the Prime Meridian are -.
Lat_{P2} = latitude of point 2.
Lon_{P2} = longitude of point 2.
For these calculations, both the azimuth and elevation are assumed to be in DDD.FFFF, where DDD is the degrees and the FFFF is the fraction of a degree. This is different than DDD.MMSS, where DDD is the degrees, MM is the minutes, and SS is the seconds.
For this example, we'll assume the following latitudes and longitudes for the two cities:
Los Angeles, USA: latitude=34.0522342, longitude=-118.2436849
Ankara, Turkey: latitude=39.92077, longitude=32.85411

```
--> latp1=34.0522342;
--> lonp1=-118.2436849;
--> latp2=39.92077;
--> lonp2=32.85411;
```

Since the equation is a long one, we're going to use an anonymous function, as discussed in Topic 4.7: The Anonymous Function on page 67. That way, we only have to enter this long equation once, but we can use it over and over again.

```
--> d=@(latp1,lonp1,latp2,lonp2) (6378.137*acos(cosd(latp1)*cosd(latp2)*cosd(lonp1-
lonp2)+sind(latp1)*sind(latp2)))
d =
 @(latp1,lonp1,latp2,lonp2)   (6378.137*acos(cosd(latp1)*cosd(latp2)*cosd(lonp1-
lonp2)+sind(latp1)*sind(latp2)))
--> d(latp1,lonp1,latp2,lonp2)
ans =
 1.1283e+04
```

The distance we've calculated with our function is 11,283 km. Using Google Earth, the distance is calculated to be 11,280 km. Not too shabby for a one-line function!

Topic 3: Matrices & Arrays

A matrix is a rectangular array of numbers, strings, To Freemat, every number is a matrix. The matrix is the most basic element to Freemat. (See FD, p. 49). Single numbers, such as 5.342, are simply a matrix with a size of [1 1], where the first number is the number of rows and the second number is the number of columns. That is why if you store a single variable, it will be listed as a size of [1 1]. This is known as a scalar. Here's an example using the **who** command (FD, p. 241):

```
--> y=5.342;
--> who
  Variable Name        Type    Flags              Size
              y     double                      [1 1]
```

Or you can use the **size** command (FD, p. 237):

```
--> y=5.342;
--> size(y)
ans =
 1 1
```

Here's another way to look at it. We just assigned a value of 5.342 to the variable "y". If we want to look at y's value, we just type "y" and hit enter, as follows:

```
--> y
ans =
  5.3420
```

You use an index to organize each element in a matrix. Indices are placed between parentheses () at the end of the variable holding the matrix. With our current value of "y", we can use an array index to look at its value, as follows:

```
--> y(1)
ans =
  5.3420
```

As stated before, a single value variable is a matrix of size [1 1]. Therefore, we can also use a double-valued index operator on it, as follows:

```
--> y=5.342;
--> y(1,1)
ans =
  5.3420
```

Note that when you want to look at the element of a multi-dimensional matrix, you put a comma between each dimension.

A two-dimensional matrix is listed in the *[row column]* format. Such a matrix is an array, meaning it has its elements sorted in rows and columns. Thus, a matrix listed as a size of [2 3] means that it has 2 rows and 3 columns. Just to be clear, a row means "elements going horizontally" and columns means "elements going vertically", as shown in Figure 22. Here's a quick overview of the difference between a matrix, an array and a vector.

Type	Description
Matrix	A rectangular array of numbers.
Array	A grouping of elements (numbers, string, other arrays, or pointers to a function) that can have one or more dimensions. A two-dimensional array containing numbers is a matrix.

Vector	A vector consists of either one row or one column, and two or more elements in the other dimension. For example, a row of 10 numbers would be considered a vector of length 10.
Scalar	This is a single number with just one row and one column.

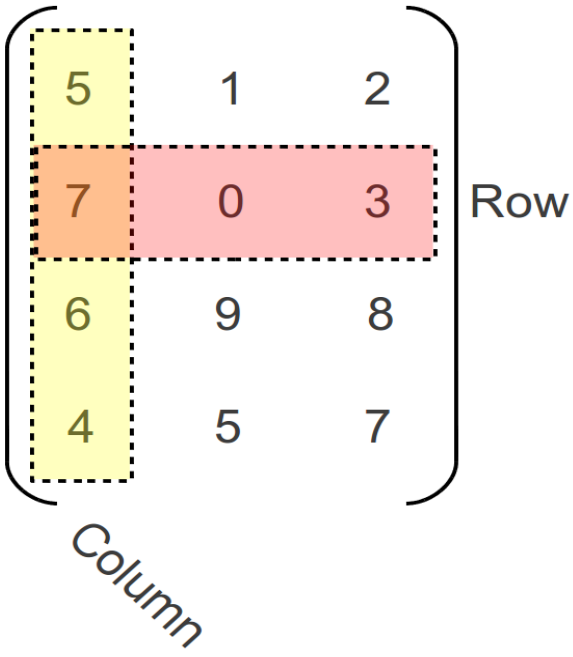


Figure 22: Overview of a matrix and how it differentiates between a row and a column. In this particular example, this is a matrix of size [4 3], meaning it has 4 rows and 3 columns. The point at which the highlighted row and column intersect is (2,1), meaning row 2 and column 1.

Topic 3.1: Understanding Cells, Matrices, Vectors and Indexing

In many of the mathematical programming languages such as Freemat, there are different ways to hold values within variables. As we discussed earlier, a variable can hold a number, a string, or a pointer to a function. Within Freemat, there are two different ways to look at how these variables are organized to store values. The first is the matrix. It's used to hold either a letter or a group of letters (meaning a string) -OR- a number or a group of numbers. It cannot hold both at the same time. The second is a cell. A cell can hold either strings, numbers or both at the same time. Now is the time when we need to discuss the various types of brackets.

<table>
<tr><th colspan="2">Understanding Brackets</th></tr>
<tr><td>[ ] (square brackets)</td><td>A set of square brackets is used to denote a matrix or vector. An example is:
<pre>--> x=[4 5 2 1 6 0]
x =
 4 5 2 1 6 0</pre>Here's another example, but with multiple rows. We use the semicolon (;) to differentiate between rows.
<pre>--> x=[5 1 8 3 4 7 6;3 6 1 9 7 5 0]
x =
 5 1 8 3 4 7 6
 3 6 1 9 7 5 0</pre></td></tr>
<tr><td>( ) (parentheses)</td><td>A set of parentheses is used for indexing a matrix or vector. An example is:
<pre>--> x=[4 5 2 1 6 0]
x =
 4 5 2 1 6 0
--> x(2)
ans =
 5</pre>Index operators do not require a variable be used. While you cannot put the index directly with the array itself, you can put the index directly on a function used to create the array.
<pre>--> x=linspace(0,2*pi,10);
--> sin(x)(4)
ans =
    0.8660</pre>Again, you cannot use an index operator directly on an array, as shown here:
<pre>--> [7 1 8 5 2 3 9](3)
Error: Unexpected input

     [7 1 8 5 2 3 9](3)
                    ^</pre></td></tr>
<tr><td>{ } (curly braces)</td><td>The curly braces denote a cell array, or can be used to index a cell array. For example:
<pre>--> y={'this is a test' [4 5 2 1 6 0] rand(4,4)}
y =
 [this is a test] [1 6 double array] [4 4 double
array]
--> y{2}
ans =
 4 5 2 1 6 0</pre></td></tr>
<tr><td>; (semicolon)</td><td>Within either a cell array or a matrix array, a semicolon is used to denote different rows. Here's an example:
<pre>--> y={'this is a test';[4 5 2 1 6 0];rand(4,4)}
y =
 [this is a test]
 [1 6 double array]
 [4 4 double array]
--> y{2}
ans =
 4 5 2 1 6 0</pre></td></tr>
</table>

, (comma) or space	There are two ways to denote columns within an array. You can either put a space between the elements, or a comma. Your preference. Freemat sees them as the same.
: (colon)	The colon has a special purpose. It can be used to denote an entire dimension of either a matrix or cell array.

end	No, that doesn't mean it's the end of this section. The word "end" has a special purpose. It can be used to denote the last element of an array. While this may seem superfluous, it's not. It can make certain tasks much simpler, such as using the cumsum (cumulative sum) function, as shown below:
	`--> x=1:50;`
	`--> y=cumsum(x);`
	`--> y(end)`
	`ans =`
	`1275`
	This can be shortened even further, as follows:
	`--> x=1:100;`
	`--> y=cumsum(x)(end)`
	`y =`
	`5050`
	Or shortened further still:
	`--> cumsum(1:100)(end)`
	`ans =`
	`5050`

Topic 3.2: Creating a Sequential Array

A sequential matrix or array can be very handy. You can use it as a counter or to store variables sequentially in an single-dimension array.

To create a simple array, you use the colon (:). By default, the colon operator uses a step size of 1 between the beginning and ending of the array. For example, to create a sequential array that goes from 1 to 10, with a step size of 1, use the following command:

```
t=1:10;
```

Note: It's a good idea of getting in the habit of using the semi-colon at the end of the line when creating an array. If you create a large array and don't use it, Freemat will try to display the variable. You'll wind up with a screenful of unneeded numbers.

Freemat uses a step size of 1 by default. You can also make the array have a user-specified step size. For example, if you want the sequence to go from 1 to 10 with a step size of 3, use the following format:

```
t=1:3:10;
```

There are two functions you can use to create sequential arrays. These are the **linspace** function (FD, p. 287) and the **logspace** function (FD, p. 288). The linspace function creates an array between two numbers such that the numbers are linearly spaced between the minimum and maximum values.

Example - Creating a Sequential Array

This first example uses the colon operator to create an array from 1 to 10.

```
t=1:10
t =
```

```
   1  2  3  4  5  6  7  8  9 10
```
The next example creates an array from 1 to 10, but with a step size of 3.
```
t=1:3:10
t =
   1  4  7 10
```
Now look at a similar matrix, but created using the linspace command.
```
--> t=linspace(1,10,7)
t =
    1.0000    2.5000    4.0000    5.5000    7.0000    8.5000   10.0000
```
What this means is that we're making a linear array that runs from 1 to 10 in 7 steps. If you do not specify the number of steps, linspace uses the default value of 100 steps.

The **logspace** function is different. It creates a linear array between the minimum and maximum points defined in the function, then it creates an array from 10^(minimum) to 10^(maximum) using this array. This is probably better explained with an example.

Example - Creating a Logarithmically-Spaced Array

Here's the logspace function with a straight beginning and endpoint.
```
--> logspace(1,10,10)
ans =
          10         100        1000       10000      100000     1000000
10000000   100000000  1000000000 10000000000
```
Note that it's calculating an array starting at 10^1 and ending with 10^{10}. If we want to create an array with logarithmically spaced values between two specific numbers, we have to combine the **logspace** with the **log10** function (FD, p. 171), as shown below:
```
--> logspace(log10(1),log10(10),10)
ans =
    1.0000    1.2915    1.6681    2.1544    2.7826    3.5938    4.6416    5.9948
7.7426   10.000
```

Topic 3.3: Creating a Random Array

Freemat provides a whole lot of functions to create a matrix filled with random values. These are:

- **rand** - creates a random variable uniformly spaced between 0 - 1. (FD, p. 319)
- **randbeta(alpha,beta)** - creates a random variable with a beta distribution. The required parameters, alpha and beta, set the shape of the probability distribution function (PDF) of the samples. (FD, p. 321)
- **randi(low,high)** - creates an array of random integers between the values of *low* and *high* (inclusive). (FD, p. 327)
- **randn** - creates an array of random values with a normal (Gaussian) distribution. (FD, p. 328)

These are just a few of the many ways to create a random array. The functions can be found in the Freemat Documentation from 319 - 334.
When creating a random array, you should be explicit with the number of rows and columns. The reason is that, by default, Freemat will use a single value entered into the function for both the rows and colums, as demonstrated here:
```
--> x=rand(10000);
```

```
--> size(x)
ans =
 10000 10000
```

Whereas if you want to create an array of one row with 10000 columns, use this format:

```
--> x=rand(1,10000);
--> size(x)
ans =
     1 10000
```

We can also use the hist (histogram) function to see the distribution of values.

Caution - A Bug in the Hist Function

As of this writing, there is a problem with the **hist** function. It does not appear to operate correctly. Until it is corrected, you can use the function provided in Appendix B, saving it as **hist.m** in one of the folders in your path. It should provide the proper functionality as described in the Freemat Documentation.

Using the hist function, we can see the distribution of values generated using the **rand** and **randn** functions:

```
--> x=rand(1,10000);
--> y=hist(x)
y =
 1042 1020 1019 1001  977 1022  914 1005 1041  957
--> y=hist(x,10,1)
y =
    0.1042    0.1020    0.1019    0.1001    0.0977    0.1022    0.0914
0.1005    0.1041    0.0957
--> plot(y)
--> ylim([0,0.2])
```

The result is shown in Figure 23.

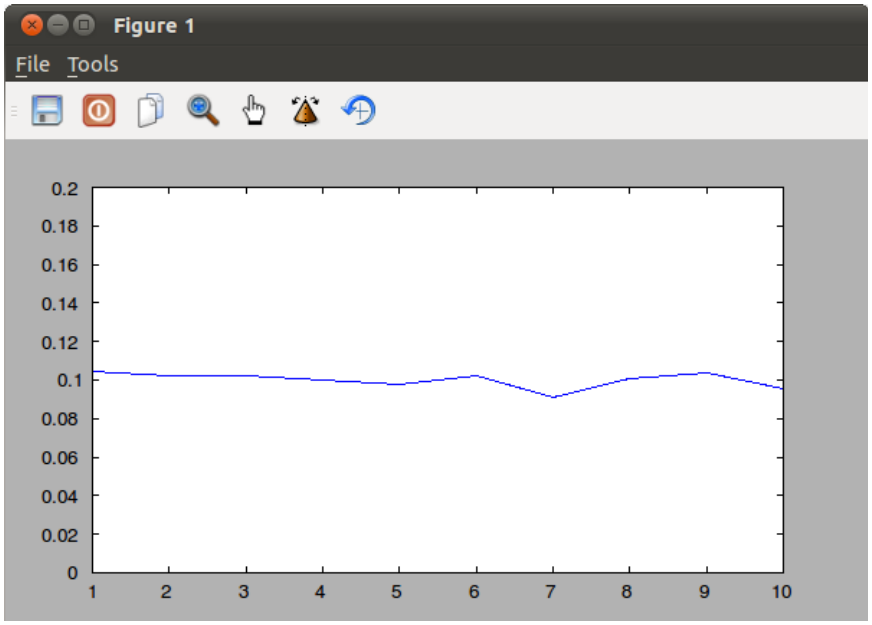


*Figure 23: Histogram plot of values using the **rand** (uniform random) function.*

Topic 3.4: Viewing a Matrix Value

To see the value of a single element of a matrix, use the following format:

```
x(d1,d2,...,dn)
```

where: x = the name of the variable
d1 = first dimension within the matrix
d2 = second dimension within the matrix
dn = n-th dimension within the matrix

Example - Viewing a Single Element of a Matrix

```
x=rand(3,3)
x =
       0.4421    0.9413    0.5842
       0.3209    0.1061    0.7719
       0.2505    0.3724    0.7086

x(1,2)
ans =
      0.9413

x(2,1)
ans =
      0.3209
```

To see a complete dimension of a matrix, use the colon (:). The format is:

```
x(:,:)
```

where x = the name of the variable
: = the dimension along which you wish to see

Example - Viewing a Complete Dimension of a Matrix

```
x=rand(3,3)
x =
       0.4421    0.9413    0.5842
       0.3209    0.1061    0.7719
       0.2505    0.3724    0.7086
x(:,1)
ans =
      0.4421
      0.3209
      0.2505
x(1,:)
ans =
      0.4421    0.9413    0.5842
```

Topic 3.5: Matrix Math

This will not be a tutorial on matrix math. There are plenty of books already available on that, such as

Elementary Linear Algebra by Howard Anton (Drexel University, John Wiley & Sons, 1987). Instead, this will be a "Here's how Freemat performs various matrix math calculations."

Topic 3.5.1: Matrix Addition

When you add two matrices together, the result is a matrix that is the sum of the elements of the summing matrices on an element-wise basis. Again, clear as mud? This means that the first matrix will have its elements summed with the same elements of the second matrix. This is shown graphically in Figure 24. Note: Matrix addition requires that the matrices being summed be the same size. This means that each must have the same number of dimensions and the same number of elements in each dimension.

Example - How Two Matrices are Summed

```
x=rand(2,2)
x =
0.3759    0.9134
0.0183    0.3580
y=rand(2,2)
y =
0.7604    0.0990
0.8077    0.4972
x+y
ans =
1.1363    1.0124
0.8260    0.8552
```

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} \\ a_{21}+b_{21} & a_{22}+b_{22} \end{pmatrix}$$

Figure 24: Matrix Addition in Freemat

Topic 3.5.2: Matrix Subtraction

This is the same as matrix addition. Each matrix must have the same number of dimensions and the same number of elements in each dimension. The each element of the second matrix is subtracted from the same element in the first matrix.

Example - Matrix Subtraction

```
--> x=rand(3,3)
x =
0.7457    0.9862    0.9445
0.3774    0.0484    0.4942
0.7623    0.0395    0.0089
```

```
--> y=rand(3,3)
y =
0.4399    0.3236    0.9954
0.9112    0.5196    0.5407
0.6565    0.1273    0.0840
--> x-y
ans =
0.3058    0.6626   -0.0509
-0.5338   -0.4713   -0.0465
0.1059   -0.0878   -0.0751
```

Topic 3.5.3: Matrix Multiplication

There are two ways to multiply matrices in Freemat. The first is the standard, linear algebra way. In this method, a two-dimensional matrix(x,y) multiplied by another two-dimensional matrix(y,z) will create a matrix(x,z). For example, if the first matrix is a (10,3) and the second a (3,7), the resulting matrix will be a (10,7) matrix.

To perform the standard multiplication, just multiply the two matrix variables as you would normally multiply them. **Remember that the second dimension of the first matrix must have the same number of elements as the first dimension of the second matrix.**

Example - Standard Matrix Multiplication

```
--> x=rand(3,5);
--> y=rand(5,6);
--> z=x*y;
--> who
Variable    Name        Type        Flags       Size
            x           double                  [3 5]
            y           double                  [5 6]
            z           double                  [3 6]
```

The second method multiplies each matrix on an element-wise basis, in the same way as addition and subtraction. This requires an extra little bit on the command line. Instead of *****, you need to use **.*** as the operator. Thus, the operation would look like:

```
x .* y
```

This operation will multiply each element in ***x*** by the same element in ***y***. Note that you can perform this operation on matrices with any number of dimensions, so long as each matrix is equally sized, meaning they have the same number of dimensions and the same number of elements in each dimension.

Example - Element-wise Multiplication

```
--> x=rand(1,5)
x =
0.7202    0.1487    0.7576    0.6396    0.2645
```

```
--> y=rand(1,5)
y =
0.5725    0.5640    0.4085    0.8969    0.4665
--> x .* y
ans =
0.4123    0.0839    0.3095    0.5737    0.1234
```

Personally, this is one of my most-used operations. The reason is that, if the two variables represent signals, then this type of element-wise multiplication is the same as mixing the two signals.

Topic 3.5.4: Matrix Division

Topic 3.5.4.1: Dividing a Matrix by a Single Number

You can divide a matrix by a single number, either real or imaginary. This simply divides each element of the matrix by the single number. This is an example of element-wise math since each element is divided individually.

Example - Dividing a Matrix by a Single Number

```
--> y
ans =
0.9132    0.0756    0.6215
0.0204    0.3774    0.4662
0.2899    0.4398    0.1450

--> y/5
ans =
0.1826    0.0151    0.1243
0.0041    0.0755    0.0932
0.0580    0.0880    0.0290
```

Topic 3.5.4.2: Dividing a Matrix by Another Matrix

Dare I say it? There is no such thing as matrix division. This means that you cannot, technically, divide one matrix by another. You might say, "But in Freemat, I can create two matrices, then divide one by the other." Yes. That is correct. But this is not a case of dividing one matrix by another. This is a case of one matrix being multiplied by the inverse of the other. In order for this process to work, the dividing matrix must have certain properties. These are:

1. The matrices must be square. This means they have to have the same number of rows as they do columns (and vice-versa). This means only 2x2, 3x3, 4x4, etc, matrices are allowed. A 2x3 matrix or a 3x2 matrix would not be allowed.
2. The matrix must be invertible. The method to "divide" one matrix by another is to first take its inverse, then multiply the inverse. But if the matrix is not invertible, the multiplication operation will not be possible. In order for it to be invertible, the determinant of the matrix cannot be zero.

If it meets these two criteria, then Freemat performs the inverse operation and multiplies the inverse by the first matrix. Clear as mud? Let's look at an example.

Example - Multiplying a Matrix by the Inverse of Another Matrix

```
--> x=rand(3,3)
x =
0.0603    0.4750    0.1794
0.5541    0.6298    0.6902
0.9326    0.4791    0.8172

--> y=rand(3,3)
y =
0.3334    0.7200    0.3153
0.5322    0.0361    0.1913
0.2064    0.1283    0.9270

--> x/y
ans =
0.6701   -0.3186    0.0313
0.7843    0.3961    0.3961
0.5233    1.2519    0.4452
```

Yes, it seems as if I'm dividing one matrix, ***x***, by another matrix, ***y***. But that's not the case. Instead, what Freemat has done is to take the inverse of ***y***, then multiplied that by ***x***. I'll show this by showing the intermediate step of calculating the inverse of the matrix ***y***.

```
--> z=y^-1
z =
-0.0294    2.0584   -0.4148
1.4898   -0.8009   -0.3415
-0.1996   -0.3476    1.2184

--> x*z
ans =
0.6701   -0.3186    0.0313
0.7843    0.3961    0.3961
0.5233    1.2519    0.4452
```

Note that this result is the same as above.

Topic 3.5.4.3: Calculating the Inverse of a Matrix

To calculate the inverse of a matrix, simply use the exponent (^) function and take the matrix to the power of -1.

Example - Calculating the Inverse of a Matrix

This example will create a square matrix (the only kind that can be inverted), then invert it.

```
y=rand(3,3)
y =
0.9132    0.0756    0.6215
0.0204    0.3774    0.4662
0.2899    0.4398    0.1450

y^-1
ans =
0.7923   -1.3832    1.0506
-0.6969   0.2520    2.1773
0.5295    2.0016   -1.8086
```

For more (and better) information, I recommend a book on linear algebra and matrix math, such as the already-mentioned *Elementary Linear Algebra* by Howard Anton (Drexel University, John Wiley & Sons, 1987).

Topic 3.5.5: Element-wise Matrix Math

It's possible to perform multiplication, division and exponential functions on matrices on an element-wise basis. For multiplication and division, this means that I can multiply or divide each element of an MxN matrix by the corresponding element of another matrix. In order to perform such element-wise operations, each matrix must be the exact same size. For example, a 2x3 matrix requires another 2x3 (not a 3x2 or any other size!) matrix in order to perform this type of operation.
To check whether the matrices are the same dimensions, you can:

a) use the **who** command (FD, p. 240) or the **size** command (FD, p. 237) to see the variable dimensions
b) use the "Workspace" tab next to the Command Window to see the variable dimensions.

Example - Problems with Element-wise Matrix Math

This is an example showing what happens when you try to perform element-wise math on two matrices of either the wrong dimensions and/or the wrong sizes.

```
--> x=rand(100,1);
--> t=1:100;
--> who
Variable Name      Type       Flags          Size
           t      int32                     [1 100]
           x      double                    [100 1]
--> x .* t
Error: Size mismatch on arguments to arithmetic operator .*
```

In this case, the matrices have the same number of elements, but the variable *t* is a size [1 100] while the variable *x* is a size [100 1]. To make them the same size, you can use the **transpose** command (FD, p. 91) to transpose one of the two matrices and align them properly.

```
--> x=transpose(x);
--> who
Variable Name      Type       Flags         Size
          ans      double                   [0 0]
           t      int32                     [1 100]
           x      double                    [1 100]
```

Note that the dimensions and sizes are the same. Thus, the math can now proceed.

```
--> z=x .* t;
--> who
Variable Name     Type        Flags       Size
           ans     double                  [0 0]
             t     int32                   [1 100]
             x     double                  [1 100]
             z     double                  [1 100]
```

Note that that the following operations are always performed element-wise:

- Multiplication, division, addition or subtraction of a matrix with a single number.
- Addition or subtraction of two matrices

When performing element-wise multiplication (Freemat name = DOTTIMES, *FD* p. 83) or division (Freemat name = DOTRIGHTDIVIDE, *FD* p. 80) using two matrices, precede the particular math operator with a period. The particular math operators are:

- Multiplication: **.***
- Division: **./**
- Exponential: **.^**

Example - Element-wise Multiplication of Two Matrices

```
--> x
ans =
0.0603    0.4750    0.1794
0.5541    0.6298    0.6902
0.9326    0.4791    0.8172

--> y
ans =
0.3334    0.7200    0.3153
0.5322    0.0361    0.1913
0.2064    0.1283    0.9270

--> x .* y
ans =
0.0201    0.3420    0.0566
0.2949    0.0228    0.1320
0.1925    0.0615    0.7576
```

Example - Element-wise Division of Two Matrices

```
--> x
ans =
0.0603    0.4750    0.1794
0.5541    0.6298    0.6902
0.9326    0.4791    0.8172

--> y
ans =
0.3334    0.7200    0.3153
0.5322    0.0361    0.1913
```

```
0.2064    0.1283    0.9270

--> x ./ y
ans =
0.1809    0.6597    0.5688
1.0411   17.4284    3.6083
4.5176    3.7355    0.8816
```

Example - Element-Wise Exponentials

When using an exponent on an array on an element-by-element basis, use the .^ operator, as follows:

```
--> x=rand(1,10)
x =
    0.7314    0.9302    0.5387    0.0270    0.5100    0.8202    0.7037    0.7831
0.1695    0.3414
--> x.^2
ans =
    0.5349    0.8653    0.2902    0.0007    0.2601    0.6727    0.4952    0.6132
0.0287    0.1165
```

Topic 4: Scripts & Functions

Scripts and functions in Freemat are nothing more than a bunch of commands chained together. You create a file that is nothing more than a series of commands running sequentially, meaning one after the other. A script or function starts at the top and goes through each command in order. When it reaches the end, it stops. These are computer programs. The difference between a script and a function is that a script is the same as typing it into the command window; a function has its own variables that it can use.

The difference between a script and a function is that a script is the same as typing it into the command window; a function has its own variables that it can use.

You can use the same variables in a function as you've used in either the Command Window, in a script, or even in another function. Variables used in a function are localized to that function. Variables used in a script are global variables, meaning that they are available to other scripts or if you want to run statements in the Command Window. When you put them into a script file (which ends with **.m**), you create a file that Freemat will execute as if you entered them directly into the Command Window one at a time. When working with lots of commands, you may find it easier to work with scripts than from the Command Window. That way, you can execute all of the commands with just one command, that of the file name of your script.

To create a script, you need a plain text editor. Within Windows, *Notepad* works well. You *can* use word processors such as MS Word or OpenOffice Writer *so long as* you save the file as a plain text (ASCII) format file. If you save it as a standard MS Word .doc file or anything that adds its own formatting to the document, Freemat will return an error. (Yes. I tried it. I like to try things to see how they will cause various processes to go kaflooey. But this one didn't go kaflooey. It just didn't go.)

Example - Running a Script Saved as a MS Word Document

Here's a quick example of what happens if you try to use a word processor to make Freemat scripts. In this case, I used Microsoft Word to create a short script. The script I made is irrelevant. When I saved the script, Word absolutely insisted on adding the extension **.doc** to the file. The filename was now **dynamite.doc**. I couldn't stop it. So, after closing Word, I went into Windows Explorer and changed it to **.m**. When I attempted to run the file, this is what I got:

```
--> dynamite
Error: Unexpected input at line number: 1 of file
C:Program Files/Freemat/dynamite.m
ÐÏ#à¡±
^
```

Like I said, a whole lot of nothing. You're much better off with a plain, Jane text editor or, even better, the Freemat Editor (See below).

Topic 4.1: The Freemat Editor

One of the great things about Freemat is that it comes with a built-in editor. This is the one I highly recommend that you use. To access it, use one of the following options:

- just type "edit" in the Command Window and hit <Enter>
- use the keyboard combination of *Ctrl + E*
- go to the menu **Tools -> Editor**.

This editor provides line numbering (useful when you get error messages for scripts and functions, since the error message tells you which line number created the error), color coding and automatic indentations for different functions (**for** and **while** loops, for example), and automatic file name extension when saving and retrieving scripts and functions.

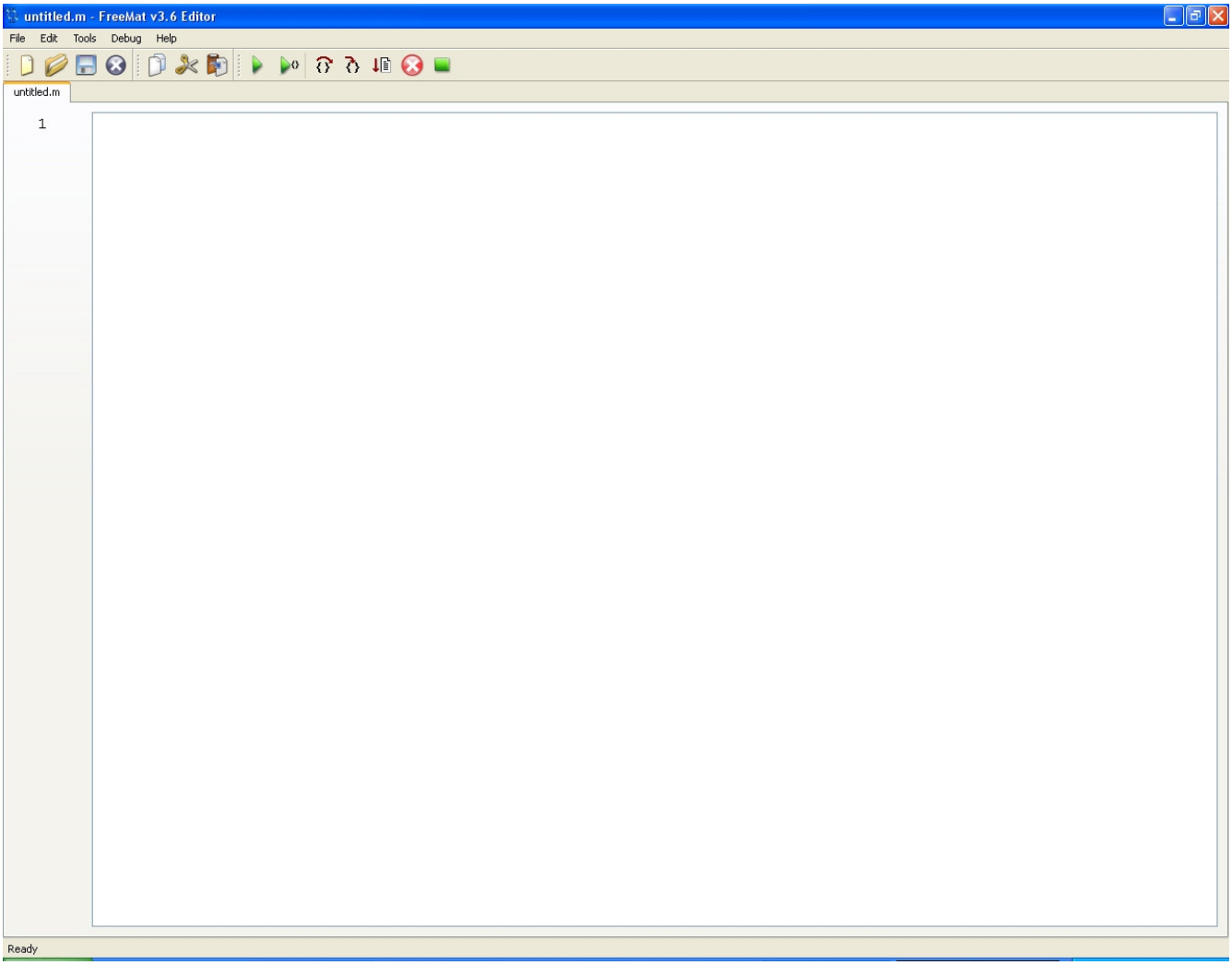


Figure 25: Editor Window

When you do, you'll get the window shown in Figure 25.

Topic 4.2: Creating a Script

Time to create a script. To create a script, type the commands, functions, numbers and variables into the Editor Window as you would if you were typing them into the Command Window. Enter the

commands in the order in which you want them executed. The difference between the Editor and Command Window is that, in the Editor Window, nothing will be executed. Yet.

Once you've entered the commands, save the script as a file with **.m** as the extension. To do so, from the Freemat Editor, click **File -> Save** or **Save As**. Or click on the **Save** icon (). This opens a window that should be familiar to anyone who has used Windows before. It's the standard "Save File" window. It will ask you both where you want to put the file and a file name. The where needs to be somewhere within your file path as defined in the *Path Tool* menu. The file name needs to be a standard name that windows allows. And the file name convention for Freemat scripts is:

- A script file name must start with a letter.
- A script file name can contain letters, numbers and the underscore.
- A script file name cannot have any spaces.

NOTE: If you're using the Freemat editor, it will automatically put the **.m** extension on the end if you don't. But I recommend getting into the habit of doing it yourself. That way, if you're making a file with another editor (Notepad, vi in Linux, or whatever), you'll always have the correct extension and there will be less chance of problems later on.
The following screen shots show some of the important parts of the Editor Window.

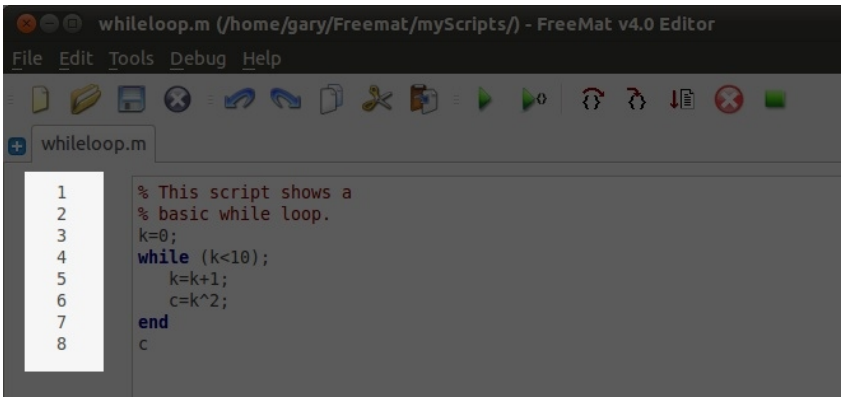


Figure 26: Editor Window showing line numbering

Line Numbering: Figure 26 shows a short, (essentially) useless script that shows the line numbering in the Editor Window. The line numbers are automatically generated. If an error occurs in the script or function, Freemat provides an error statement listing the line number where the error occured.

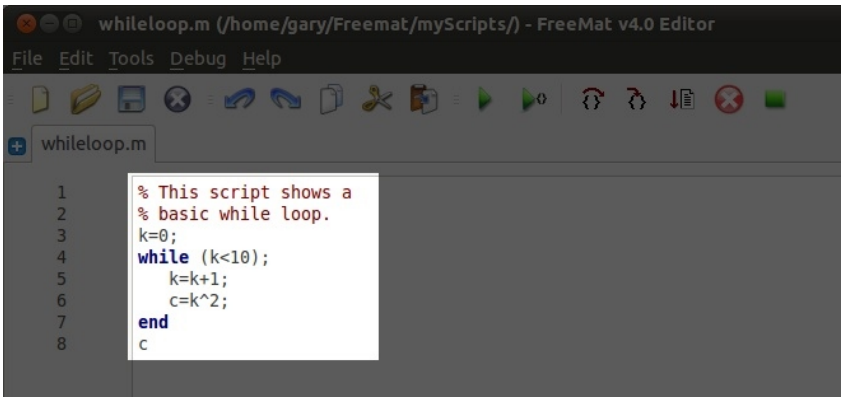
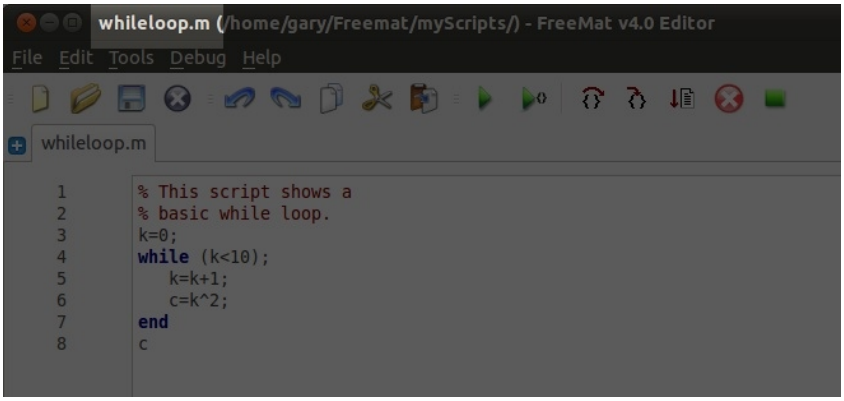


Figure 27: Editor Window showing color coding and indentation.

Color Coding & Indentation: Figure 27 shows the Editor Window with a **for** loop. (**For** loops will be covered in *Topic 10: Flow Control*.) Note the color coding used in the **for** loop. Also note the indentation of the statements within the **for** loop.



Filename: Figure 28 shows the filename at the top of the Editor Window. The filename shows up either (a) after you've saved a file or (b) if you loaded the file.

Figure 28: Editor Window showing the filename

New Tab Icon: Figure 29 shows the New Tab icon within the Editor Window. When you click this icon, it creates a new tab (essentially, a new Editor Window) that you can use to create a new script or function, or to edit a script or function saved on your system.

Figure 29: Editor Window showing the icon used to create a new tab.

Example - Creating a Script

Try this example script. Enter the following commands into the editor window. (Note: You can use copy & paste if you don't want to type them in yourself.)

```
% Create a damped sine wave.
% This was adapted from the Freemat Documentation
% on the hold command (p. 466).
x=linspace(-3,3,1024);
y=(exp(-(x.^2))).*cos(8*pi*x);
plot(x,y);
```

Figure 30 shows the Editor Window with the program.

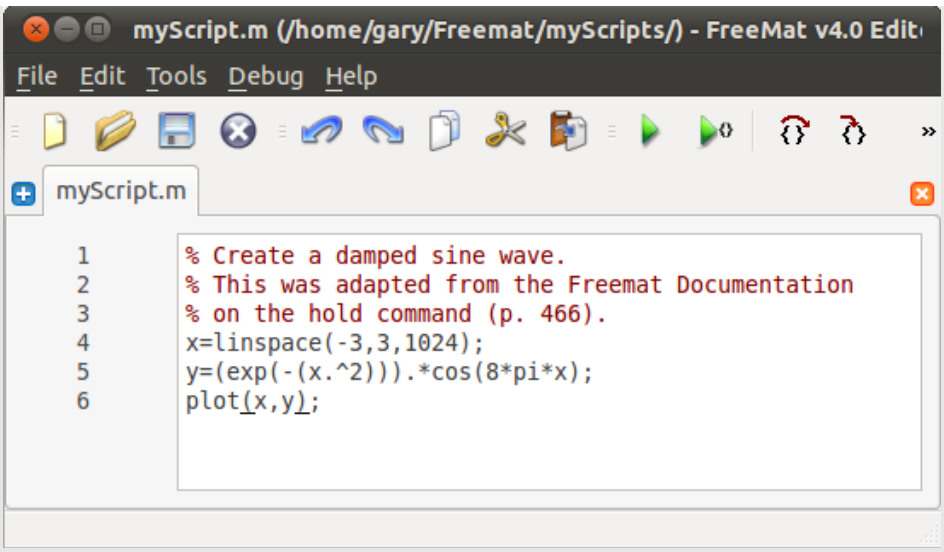


Figure 30: Editor window with test script

When you are finished, save the script file. From the "Save File" window, enter a file name. I used **myScript.m** as my file name. Now go back to the Command Window. Type in the name minus the extension:

```
myScript
```

Freemat executes each command in the script in the sequence you entered them.
For this simple script, it will create a variable array from 1 to 128, calculate eight cycles of a sine wave, then plot the sine wave. The plot is shown in Figure 31.

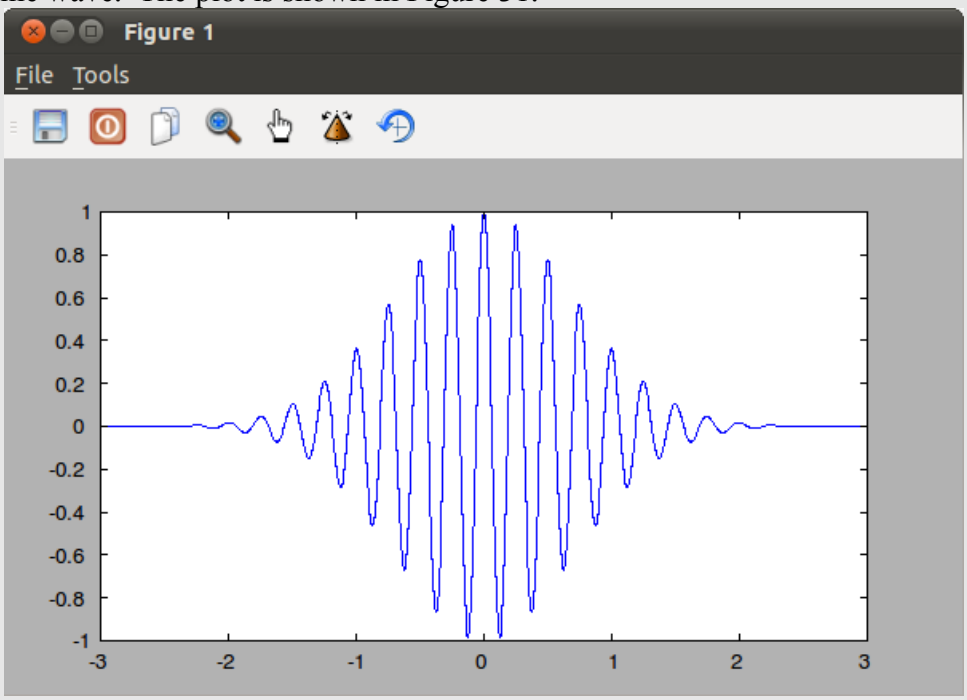


Figure 31: Basic plot, in this case a damped sine wave, generated from a script.

The great thing about scripts is that Freemat makes it very easy to repeat them. If you run a script, the

command to run the script (which is simply the file name minus the .m extension) is stored in the history file. From the Command Window, you can use the up and down arrows on your keyboard to cycle through these old commands. To redo a script, from the Command Window, use the up arrow to put the previous command onto the command line. Then press <Enter>. This repeats the script. Thus, you can run a script, make any changes within the script you want, re-save the script, then repeat running it from the Command Window. This is a very easy way to run long scripts repeatedly.

Remember: Do NOT type the ".m" after the name when you want to run the script from the command line!

Topic 4.3: Running a Script

Once you've created a script and saved it as a **.m** file, you can run it simply by typing the name of the file *without typing the .m extension.* Then press <Enter>. For example, if you created a file called "myscript.m", in the Command Window, you'd type the following to execute the script:

```
myscript
```

It's that simple.

Topic 4.4: Creating & Using a Function

A function is a short script that performs an operation (FD, p. 56). The difference between a script and a function is that the variables used in a script will be global; the variables used within a function will be local. This means that you can use the same variable names in a function as you've used outside of it. The operation can be either to perform a series of calculations, provide an output, create a display, or combinations of these. You typically create a function for some action you routinely perform.
A function is a script that starts with the word function and accepts one (or more) input values. It may also return one or more values. It has the following syntax:

```
function [returnValue1, returnValue2, ... , returnValueM] =
function_name (parameter1, parameter2, ... , parameterN)
      statements
```

The function starts, literally enough, with the word function. If it will return a value (or values), it should have a variable that will be used to return the value (or values). This return value is set equal to the name of the function and any input variables it will accept. Here's a general syntax for function that accepts one input variable.

```
function rv=funcName(x)
   (various commands)
   rv=some final value;
```

Notice that a function does not finish with the **end** statement. Once your function is complete, just stop. If the function is called, Freemat goes through the function until it reaches the last line. Once it's completed, it returns any values needed and returns to wherever the function was called.
When you create a function, you need to give the filename for that function the same name as the function. For example, if you create a function called **myFunc**, it should be saved with the filename **myFunc.m** somewhere in your path.
To use a function, enter the function name and any parameters it requires. Here are some examples to help you understand.

Example - A Function to Calculate the Factorial of a Number

Freemat does not have a built-in factorial function, typically denoted by the exclamation point (!). However, it's straightforward to make your own. This function calculates the factorial value of the input variable n.

```
function return_value=fact(n)
n=floor(n);
return_value=gamma(n+1);
```

Here is how it would appear in the Freemat Editor window:

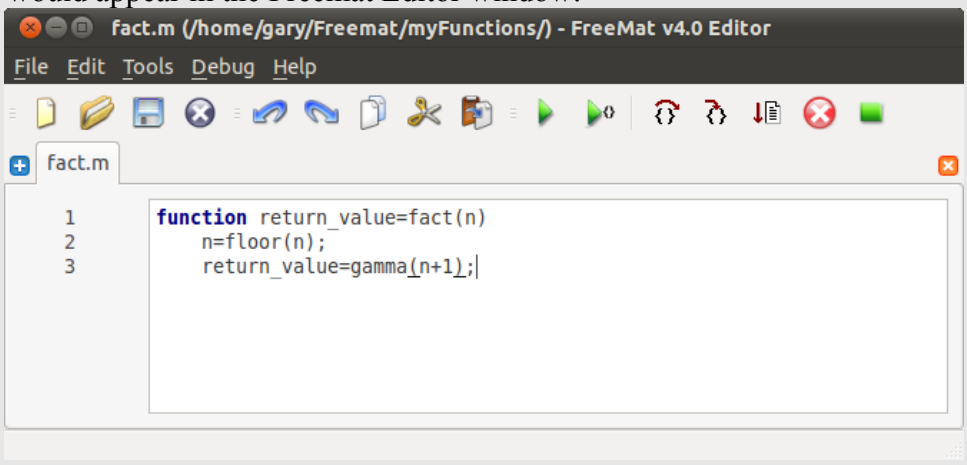


Figure 32: The editor window with a short function written

Because the function has to be saved with the same filename as the function itself, the filename is **fact.m**. Below shows the result of calculating 5!. (And that is 5 factorial, not that I'm really excited about calculating the value of 5.)

```
--> fact(5)
ans =
  120.0000
```

We can also assign the output of a function to a variable.

```
--> y=fact(5)
y =
  120.0000
```

It's also possible to create a function that requires more than a single parameter. To create such a function, simply provide variables for all of the required parameters. Thus, for a function that requires three input parameters, the function would appear as follows:

```
exampleFunction(x,y,z)
```

where: x,y,z = input parameters

Example - A Function with Multiple Inputs

In probability, a combinatorial is the number of combinations (with replacement) of "r" objects in a set of "n" objects. The equation for calculating the number of combinations is:

$$C_{n,r} = \frac{n!}{(n-r)!\,r!} = \binom{n}{r}$$

Bernoulli trials are one way that allows you to calculate the probability of having "k" successes in "n" trials, with each trial having a probability of success of "p". The overall equation uses the

combinatorial function from above and is as follows:

$$P_n(k)=\binom{n}{k}p^k(1-p)^{n-k}$$

All of these equations take multiple inputs. We can create functions for each of these. We'll start with the combinatorial function.

```
function rv=comb(n,r)
   rv=gamma(n+1)./(gamma(n-r+1).*gamma(r+1));
```

Saving this under the filename of **comb.m**, we can then try this function. We'll look at the number of combinations of 3 objects within a set of 5.

```
--> comb(5,3)
ans =
   10.0000
```

This says that 3 objects in a set of 5 can be combined in 10 different ways. Here's how the 3 X's might be arranged in a set of 5 objects.

```
0XXX0
0XX0X
0X0XX
00XXX
X0XX0
X0X0X
X00XX
XX0X0
XX00X
XXX00
```

Next, we'll look at making a function to calculate the probability for Bernoulli trials. This function requires three inputs, which are the number of trials (n), the number of successes (k), and the probability of success (p).

```
function return_value = bern(n,k,p)
   x1 = comb(n,k);
   x2 = p.^k;
   x3 = (1-p).^(n-k);
   return_value = x1.*x2.*x3;
```

Let's say that a baseball player has a batting average (hits to times at bat) of 0.24. (We'll ignore the differences due to being walked, etc.) We'll use this as the probability of "success". What's the probability that he will get 8 hits in the next 10 at bats? In this case, n=10, k=8, and p=0.24.

```
--> bern(10,8,0.24)
ans =
 2.8611e-04
```

That probability that this player will get 8 hits in the next 10 at-bats is pretty low. But what is the probability that he'll get no hits at all?

```
--> bern(10,0,0.24)
ans =
    0.0643
```

This means that he has a 6.4% chance that he won't get any hits at all during his next 10 at-bats. In other words, there's a better probability that he'll get no hits than he will get 8 hits.

Topic 4.5: Improving a Function's Utility

There are a couple of things you can do to improve the overall functionality of your home-brewed function. These are checking the inputs and using element-wise multiplication and division whenever possible.

Topic 4.5.1: Checking Function Inputs

There is a command called **nargin** (FD, p. 61), which stands for "number of arguments input". This command, when invoked in a function, counts the number of input arguments passed to the function. You can use this to determine if all of the necessary parameters have been passed to your function, or to set up defaults for parameters that may use specific default numbers. The example on page 62 of the Freemat Documentation does a pretty good job of demonstrating this.

Example - Counting Arguments for a Function Input

In this first example, we'll look at the number of inputs to a particular function. If all of the arguments are not present, we'll display an error message, then quit out of the argument. We're going to use the Bernoulli trials function from earlier, but with some added code to check the input arguments.

```
function return_value = bern(n,k,p)
if(nargin<3);
printf('This function requires three inputs, which are (in order):\n');
printf('the number of trials, the number of successes and the probability\n');
printf('of success of each trial.\n');
return
end
x1 = comb(n,k);
x2 = p.^k;
x3 = (1-p).^(n-k);
return_value = x1.*x2.*x3;
```

Running this function with only two inputs, we get this:

```
--> bern(10,0)
ERROR: This function requires three inputs, which are (in order):
the number of trials, the number of successes and the probability
of success of each trial.
```

We can also make it so that if all of the arguments are not present, we can set up defaults. For example, we can set the probability of success for our Bernoulli trials function to 0.5 if it's not entered.

```
function return_value = bern(n,k,p)
if(nargin<2);
printf('ERROR: This function requires at least two inputs, which are (in
order):\n');
printf('the number of trials and the number of successes.\n');
return
end
if(nargin<3);
p=0.5;
end
x1 = comb(n,k);
x2 = p.^k;
x3 = (1-p).^(n-k);
return_value = x1.*x2.*x3;
```

The probability of getting heads when a coin if flipped is 0.5, which is our default if we don't set the

probability for our **bern** function. Let's calculate the probability that we'll get heads 5 times in 10 flips of a coin.

```
--> bern(10,5)
ans =
    0.2461
```

In this case, since we only entered the total number of trials (10) and the number of successes (5), the function defaults to setting p=0.5. It then calculates the rest of the probability with these three numbers.

Topic 4.5.2: Using Element-Wise Math for Functions

Look at our original fact function.

```
function rv=comb(n,r)
   rv=gamma(n+1)./(gamma(n-r+1).*gamma(r+1));
```

Note that for the multiplication we're using ".*" and for the division we're using "./". This is the element-wise multiplication and division discussed in Topic 3.5.5: Element-wise Matrix Math. If we were to use just the normal multiplication or division operations, we'd be limited to only entering scalar numbers as inputs. By using element-wise math for functions, we can use them for arrays as well as scalars.

Example - The Difference of Element-Wise Math in Functions

Let's get back to our combinatorial function comb and set it for normal, not element-wise, division and multiplication.

```
function rv=comb(n,r)
   rv=gamma(n+1)/(gamma(n-r+1)*gamma(r+1));
```

If we try this with an array as an input, here's how it turns out:

```
--> comb(10,0:3)
In /home/gary/Freemat/myFunctions/comb.m(comb) at line 21
    In docli(builtin) at line 1
    In base(base)
    In base()
    In global()
Error: Requested matrix multiplication requires arguments to be conformant.
```

By using element-wise math for all multiplication and divisions in the function, we won't have to worry about entering arrays as inputs. Back to our original function.

```
function returnValue=comb(n,r)
returnValue=gamma(n+1)./(gamma(n-r+1).*gamma(r+1));
```

Let's look at this by calculating the number of combinations of 1 to 5 objects in a set of 10.

```
--> comb(10,1:5)
ans =
   10.0000   45.0000  120.0000  210.0000  252.0000
```

The point here is that I recommend that any multiplications or divisions in a function be set for element-wise, and not just for scalars. It will make your functions much more useful.

Topic 4.6: What Was I Thinking? -or- Comment Your Functions and Scripts

Here's a piece of trivia. You will most likely forget. That's it. You will forget, so don't forget that. The script or function that you're writing today may seem "intuitively obvious to the casual observer". But tomorrow, or the next day, or perhaps even the day after, you'll probably look at it and go, "What was I thinking when I wrote this?"

My friends, that's where comments come into play. A comment is nothing more than text put into the code of your function or script which provides some explanation and context. It merely says, "This is what is going on with this code." I'm not a programmer, but all of the good ones I know say that, on average, your program, whether a function or a script, will have more comments than code. I've taken this advice to heart. In one case (see the example below), I wrote a script in which the code to run only required 13 lines. The comments to explain the code required another 113 lines. Yes, 126 lines in a simple function of which ~90% are just comments. I can almost guarantee you, however, that when I look at the code five years from now, I will be able to remember what I was thinking when I wrote it and I will have no difficulty understanding how it operates.

To put a comment into your code, use the percentage (%) mark, followed by your comment. If you're using the Freemat Editor, you'll note that the editor automatically uses a light brown font for comments to make them easily stand out from the code itself (yet another reason I really prefer using the Freemat Editor). A comment can either be on a line by itself, or it can come after some other code. Just understand that whatever comes after the percent (%) mark is considered a comment and will have no effect on the script or function execution.

```
% This is an example of a comment on its own line.
n=5;  % This is a comment on a line with code.
```

What you will see throughout this book are various scripts and functions that will have comments in them. This is not only to allow you, the reader, to better understand what they are doing, but also to help up, the writers, to remember how we structured them.

Example - Adding Comments to a Function

Looking at our combinatorial function **comb**, I've added several comments to help ensure that I'll remember not only what it's doing, but how it's doing it.

```
% Combinatorial function
% filename = comb.m
% This function returns the number of combinations
% of size r from a set of size n. This is the number
% of ways that items can be selected without regard
% to the order in which they are selected. The math
% form is sometimes written as:
%
% n (n) n!
% Ck = (k) = --------
% k!(n-k)!
% Ex: How many ways is it possible to select 2
% objects from a set of 5, without regard to
% order?
%
% (5)
% (2) = (5!)/(2!(5-2)!) = 120/(2*6) = 10
```

```
%
function returnValue=comb(n,r)
returnValue=gamma(n+1)./(gamma(n-r+1).*gamma(r+1));
```

We can also add comments to the Bernoulli trials function **bern**. This includes comments on the lines of the math operations.

```
% Bernoulli trial function
% filename = bern.m
% This function returns the probability of k successes
% in n trials, with each trial having a probability
% of success of p.
%
%         (n)  k     (n-k)
% Pn(k) = (k)*p *(1-p)
function return_value = bern(n,k,p)
if(nargin<2);
printf('ERROR: This function requires at least two inputs, which are (in
order):\n');
printf('the number of trials and the number of successes.\n');
return
end
if(nargin<3);
p=0.5;
end
x1 = comb(n,k); % Calculate the combinatorial function.
x2 = p.^k; % Calculate the probability of k successes.
x3 = (1-p).^(n-k); % Calculate the probability of n-k failures.
return_value = x1.*x2.*x3;
```

Topic 4.7: The Anonymous Function

Did you know that it's possible to not only have "normal" values in a variable, such as numbers and strings, but it's also possible to have a function be represented by a variable? Yes, that's something that really gives a lot of flexibility to Freemat. An anonymous function consists of a variable name, the "=@" operator, a list of the independent variables of the function, and the function math. This is shown in . In essence, an anonymous function is a function stored in a variable.

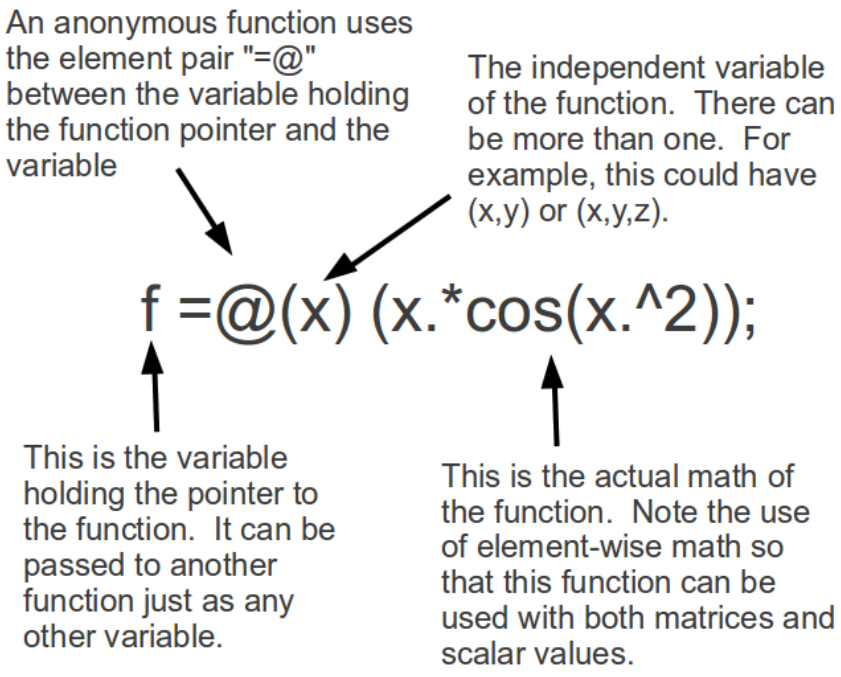


Figure 33: Overview of the anonymous function.

Here's a quick example demonstrating an anonymous function. Just like a standard function, it's a good idea to use element-wise multiplication and division to ensure that your anonymous functions work on arrays as well as scalars.

```
--> f=@(x) (exp(-x.^2));
--> x=linspace(-1,3,10);
--> f(x)
ans =
    0.3679    0.7344    0.9877    0.8948    0.5461    0.2245    0.0622
0.0116    0.0015    0.0001
```

I'll take a quick aside and explain what I just did – the name of our function (f) can be passed around just like any other variable. When we say f = @(x), the @ symbol denotes that it's a function (not a vector or cell or other entity), and the (x) denotes that it's a function of x. If it were a function of x,y and z, we would have f = @(x,y,z). The rest of the expression is simply the function to be evaluated, in this case:

$$f(x)=e^{-x^2}$$

When we call f(3), for example, it will return exp(-3^2). Anonymous functions will be used quite extensively in Topic 9: Basic Numerical Methods starting on page 160.

Topic 4.8: Program Inputs and Printing Text

There is a function to input data into scripts, called input. The printf function uses certain character strings, called switches, to format the data for display. This can be used to input either numbers or alphabetic characters. Another function, printf, allows for the display of text and numbers. The printf function allows for the formatting of output data.

Topic 4.8.1: The printf Function

The **printf** function is used to print strings. The format for the **printf** function is:

```
printf(string, n1, n2...)
```

where

- *string* is the string to be printed, including the \n switch which should end every line.
- n1, n2... : either numbers or strings which will be printed using the format set-up in *string*.

Example - The printf Function

```
--> printf('This is a string with the newline character at the end.\n');
This is a string with the newline character at the end.
--> printf('This is a test.\n');
This is a test.
--> printf('How now brown cow?\n');
How now brown cow?
-->
```

For the next example, I'll print multiple lines using one printf function. I do this using the *newline* switch (\n).

```
--> printf('This is line 1.\nThis is line 2.\n');
This is line 1.
This is line 2.
-->
```

A string is any set of characters, including letters, numbers and special characters, that is placed within single quote marks. The following are examples of strings:

```
'hello'
'Hello'
'my name is Joe.'
'This is yet another example of a string.'
'100 bottles of beer on the wall / 100 bottles of beer...'
'I well & truly feel that $100 in 10 years will get you a nice cup of
coffee.'
```

The **printf** function also uses special character sets, called *switches*, to format the data for printing. This includes formatting for the display of numbers and for a new line to be added. This latter switch is called the newline switch, and is \n. It must be included within the quotes of the string. Therefore, for each of the strings listed above, they should appear as follows if used in the **printf** function:

```
'hello\n'
'Hello\n'
'my name is Joe.\n'
'This is yet another example of a string.\n'
'100 bottles of beer on the wall / 100 bottles of beer...\n'
'I well & truly feel that $100 in 10 years will get you a nice cup of
coffee.\n'
```

If you do not put the \n at the end of the string, Freemat will overwrite the line with its next command prompt or next **printf** function.

Example - Not Using the Newline (\n) Switch

This example shows what happens if you do not put a newline switch at the end of the string to be printed.

```
--> printf('intvalue is %d, floatvalue is %f',3,1.53);
-->
```

What happened was that *the string in the printf function was actually printed, but then the command prompt overwrote the line you just printed*. Here's the same example, but with the \n added at the end of the string:

```
--> printf('intvalue is %d, floatvalue is %f\n',3,1.53);
intvalue is 3, floatvalue is 1.530000
-->
```

With the \n added at the end of the string, the line is properly shown.

You can also put a variable representing a string into the **printf** function.

Example - Printing a String Variable

```
--> s1='This is a my first string.\n';
--> printf(s1);
This is a my first string.
-->
```

Note that the string above has the newline switch (\n) at the end. However, in the case of printing using a variable, it's possible to add the newline switch as part of the **printf** function. This can be done by concatenating the switch onto the string inside the function.

```
--> s1='This is a my first string.';
--> printf([s1 '\n']);
This is a my first string.
-->
```

Topic 4.8.2: Printing Numbers

You cannot simply put a number into the **printf** function. If you attempt to simply print a number, you will get an error.

Example - Putting a Number into the printf Function

```
--> printf(6);
Error: printf format argument must be a string
-->
```

In order to print numbers, you have to setup a string to print the number based on its type. The numbers can be formatted based on type, such as integer or floating point. The switch, or flag as stated in both the Freemat online documentation and in the Freemat v4.0 Documentation, is the percent (%)

character followed by another character, or set of characters, that formats the printing of the numbers. For example, "%f" is the formatting for a floating point number.
The following are the allowed formatting strings for the **printf** function:

- **%f**: floating point number, with a default of 6 digits of precision after the decimal point.
- **%+f**: floating point number, but with a "+" sign placed in front of all positive numbers and a "-" sign in front of all negative numbers.
- **%e**: This is a floating point number, but with the exponential notation for all numbers, regardless of the position of the decimal point.
- **%n.mf**: floating point number, but with n total characters (where the decimal point takes up one character position) and m points of precision after the decimal point.
- **%0n.mf**: floating point number, preceded with a 0. The number will only be preceded by zeros if n>(m+3).
- **%d**: integer number

Example - Printing Numbers with Different Formats

```
--> printf('The number is %f.\n',pi/10);
The number is 0.314159.
--> printf('The number is %f\n',pi);
The number is 3.141593
-->
```

Topic 4.8.3: Printing Special Characters

You can print characters based on their ASCII (American Standard Code for Information Interchange) code. The ASCII code is an integer between 1 - 255. For example, the capital letter "A" is ASCII code 65, the number "7" is ASCII code 55, and the question mark (?) is 63. ASCII codes are divided into two groups, standard and special. The standard are those from 0 - 127 and are standardized, meaning pretty much everyone uses the same codes to mean the same things. The codes from 128 to 255, however, may mean different things to different people.
Freemat provides a format for characters, represented by "%c", that allows you to add special characters to your strings. For many special characters, such as $, #, @, &, and !, are already available on your keyboard. You don't need to use the character format. However, some characters are not. And, depending where you are in the world, you may have different characters available on your keyboard than the keyboard I'm using here in the United States. This is where the character format comes in handy. It allows for the printing of special characters using the character format (%c) followed by the appropriate ASCII code (an integer between 1 - 255). The character format has the following syntax:

```
printf('%c', code)
```

where: %c is the formatting for the character to be printed
If you would like to find out what characters go with which ASCII codes, use the following script.

Example - Finding Your ASCII Codes

If you would like to create a list of all ASCII characters, you can use the following one-line script. Note that I start at 32 (the space character), since characters below 32 are used for things that are irrelevant to Freemat.

```
--> for(i=32:254);printf('%i %c\n',i,i);end;
32
33 !
34 "
35 #
36 $
37
38 &
39 '
40 (
41 )
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
49 1
50 2
51 3
52 4
...  NOTE:  I cut the lines between 52 - 230.
230 æ
231 ç
232 è
233 é
234 ê
235 ë
236 ì
237 í
238 î
239 ï
240 ð
241 ñ
242 ò
243 ó
244 ô
245 õ
246 ö
247 ÷
248 ø
249 ù
250 ú
251 û
252 ü
253 ý
254 þ
```

These are the results I achieved with both the Freemat running on XP and on the MS Vista system. I imagine it is what is created when Freemat is compiled. For those of you running on Linux systems in which you compiled the kernel yourself, you may wind up with different results.

I've found the following codes useful for scripts in which I wanted to print certain characters.

ASCII Code	Character	Comments
163	£	Symbol representing the British pound
165	¥	Symbol representing the Japanese yen
169	©	copyright symbol
176	°	degree symbol
177	±	plus / minus symbol
181	µ	Greek letter "mu", which represents "micro" or 10^{-6} in math notation.
197	Å	Symbol for the angstrom, or 10^{-10} meters.
215	×	Multiplication sign (more subtle than just using an "x")
247	÷	Division sign

Example - Printing with Special Characters

This first, short example shows how to display a currency exchange rate between US dollars and British pounds. This uses the %c format with a code of 163, which will print the British pounds symbol (£).

```
--> pounds=0.619;
--> printf('The current exchange rate is %c%4.2f to every $1.\n',163,pounds);
The current exchange rate is £0.62 to every $1.
```

This next example uses the division sign.

```
--> printf('7 %c 3 = %f.\n',247,7/3);
7 ÷ 3 = 2.333333.
-->
```

%c: character requiring an integer number between 0 - 255. The number corresponds to the ASCII code for the desired character.

Example - Printing the % Symbol

The **printf** function uses the percent (%) symbol as a means to identify special formatting. How, then, do you print the actual % symbol itself? For example, what if you want to print a percentage and denote it using the % symbol? To print the percent symbol in Freemat, add "%%" to the string, as shown below:

The first example just prints the % symbol.

```
--> printf('%%\n')
%
```

Here's another method to print the "%" (percent symbol). This uses the ASCII code along with the char formatting symbol:

```
--> printf('%c\n',37);
%
```

```
-->
```
This next example uses the percent symbol in a string with a floating point number.
```
--> x=65;
--> y=33;
--> p=y/x;
--> printf('The percentage is %4.2f%%.\n',p*100);
The percentage is 50.77%.
-->
```
Using the second method, the final printf function would appear as follows:
```
--> printf('The percentage is %0.3f%c.\n',p*100,37);
The percentage is 50.769%.
-->
```
The string uses the floating point format (%4.2f) followed by a char format. After the string, two variables are listed. The first is for the numeric value of the percentage (p*100). The last one(,37) is the ASCII code for the "%" symbol. This is similar to printing "%" within the string.

The **printf** function does not word wrap. This means that, if you put a string that is longer than a single line, the sentence may be split mid-word, as shown in the example below. This means that, if you are writing scripts or functions that will use the **printf** function, you will have to choose how wide you think your users will set their Freemat screens.

Example - Lack of Word Wrapping

The following is an example of how Freemat does not word wrap. Therefore, it falls on you (the programmer) to understand the form in which the printf statement will be displayed.
```
--> printf('The quick brown fox jumped over the lazy dog.  The quick brown fox jump
ed over the lazy dog.  The quick brown fox jumped over the lazy dog.\n');
The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the l
azy dog.  The quick brown fox jumped over the lazy dog.
-->
```

Topic 4.9: The input Function

Freemat provides a means to input data from the keyboard. This is the input function (FD, p. 354). The input function can be used to enter either numbers or text. The general syntax for entering numbers is:
```
r=input('prompt text')
```
The general syntax for entering a string is:
```
r=input('prompt text','s')
```

Caution - Using the Input Function for Numbers

Depending on the particular version of Freemat 4.0 that you are running, you may find that trying to use the input function to enter numbers. The typical behavior is as follows:

```
In /usr/share/freemat/toolbox/io/input.m(input) at line 37
      In docli(builtin) at line 1
      In base(base)
      In base()
      In global()
Error:  Too many inputs to function evalin
```

If this is the error you are seeing, there are two remedies; one is a work-around and the other is a more permanent fix (though perhaps not the fix that Freemat really wants).

The first is a work-around. Instead of using the normal syntax of:

```
x=input('The number is ');
```

Use this syntax instead:

```
x=str2num(input('The number is ','s'));
```

This takes the input as a string (which appears to work fine) then converts it to a number.

The second method is a bit more work. Open the **input.m** file and change the problem line. You can find this file in the following locations depending on the operating system

Windows (7, Vista, XP): **C:\Program Files\FreeMat\toolbox\io\input.m**

Linux: **/usr/share/freemat/toolbox/io/input.m**

After opening this file, find the line that says:

```
y = evalin('caller',a,'0;printf(''%s\n'',lasterr);needval=1;');
```

Change it to this:

```
y = eval(a,'0;printf(''%s\n'',lasterr);needval=1;');
```

Save the file and re-start Freemat. Note that, depending on the operating system, you may need to use administrative privileges to change this file.

Example - Inputting Data

This example will make use of a script to display a graph based on a mathematical function, ask for a user to input two x-axis limits, then highlight that part of the function in the graph. Note that this will make use of plot techniques that we have not covered yet.

```
% Create a graph, then highlight section based
% on user input.
clear all;
close('all');
xMin=-2;
xMax=3;
x=linspace(xMin,xMax,1024);
y=x.*cos(x.^2);
plot(x,y);
hMin=input('Enter lower limit: ');
hMax=input('Enter upper limit: ');
x=linspace(hMin,hMax,1024);
y=x.*cos(x.^2);
line(x,y,zeros(1,length(x)),'linewidth',6,'color','k');
line(x,y,zeros(1,length(x)),'linewidth',2,'color',[1 1 0]);
```

Saving this script as plotInput.m, then running it, we get this:

```
--> plotInput
```

This creates the following graph:

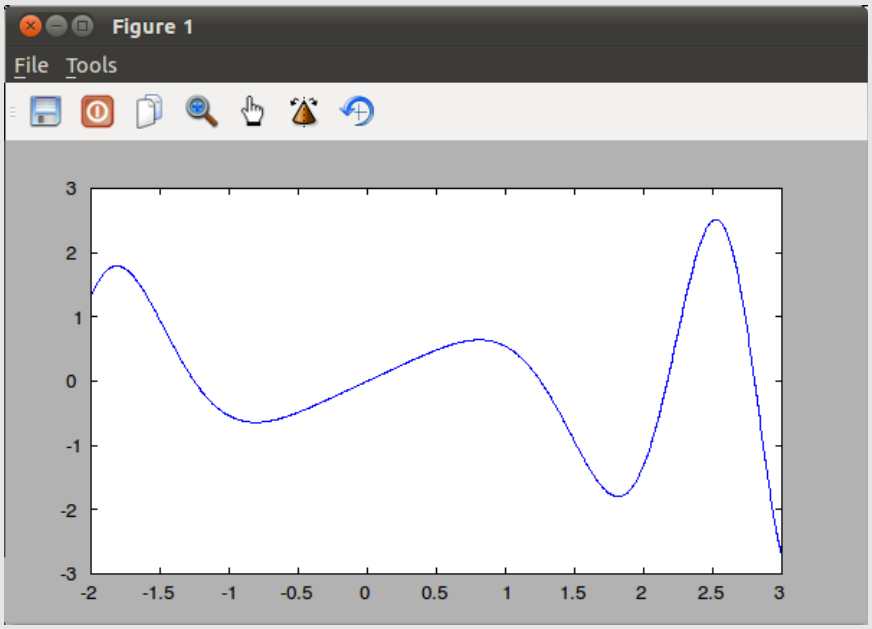


Figure 34: Basic graph, from which a portion will be highlighted.

Then we enter the lower and upper limits of the portion that will be highlighted:

```
Enter lower limit: -0.5
Enter upper limit: 1.5
-->
```

The result is a portion of the graph, from -0.5 to 1.5, being highlighted as shown in Figure 35.

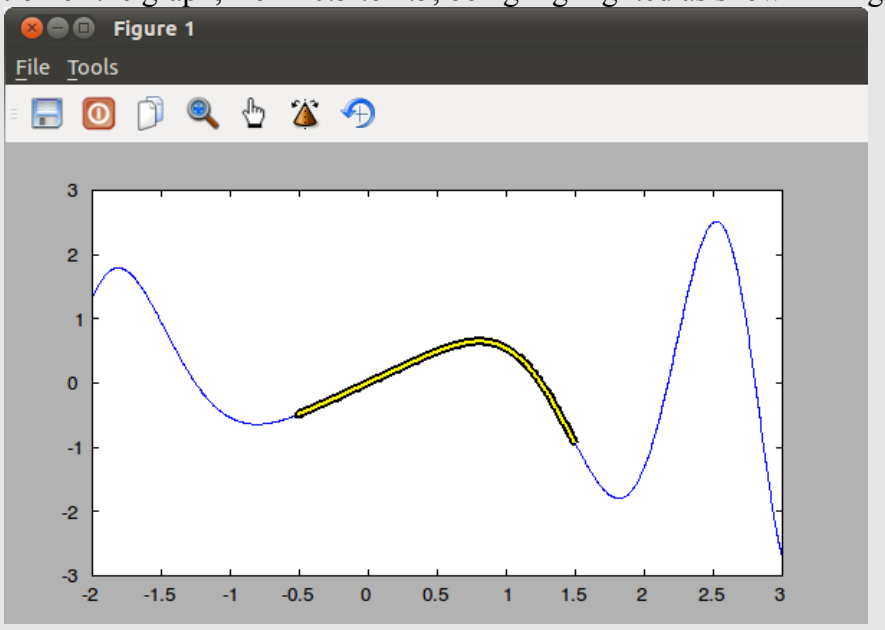


Figure 35: This shows a graph with a portion highlighted based on user input.

This could also have been done with a single **input** function. In this case, the input would require a

two element vector, with the first element being the lower limit and the second element as the upper limit.

```
% Create a graph, then highlight section based
% on user input.
clear all;
close('all');
xMin=-2;
xMax=3;
x=linspace(xMin,xMax,1024);
y=x.*cos(x.^2);
plot(x,y);
hLim=input('Enter the lower & upper limit: ');
x=linspace(hLim(1),hLim(2),1024);
y=x.*cos(x.^2);
line(x,y,zeros(1,length(x)),'linewidth',6,'color','k');
line(x,y,zeros(1,length(x)),'linewidth',2,'color',[1 1 0]);
```

Running this new script as plotLimit.m, we get this:

```
--> plotInput
Enter the lower & upper limit: [-0.5,1.5]
-->
```

The resulting graphs are the same as Figure 34 and Figure 35.

Example - Inputting Text into a Script

This example will use multiple input commands to enter text that will be used to create a short "Mad Lib".

```
% Create a "MadLib" using the input command.
clear all;
adj1=input('Enter a color: ','s');
noun1=input('Enter a noun: ','s');
printf('I decided to take my %s %s for a drive down the street.\n',adj1,noun1);
```

Running this script, we get the following:

```
--> madLib
Enter a color: blue
Enter a noun: tree
I decided to take my blue tree for a drive down the street.
-->
```

Topic 4.10: Inputting Data from ASCII Text Files

The **dlmread** function (FD, p. 338) provides the ability to read in numbers from an ASCII text file. **The following examples use data provided by the American Physical Society Public Outreach Program and we are indebted to them for providing the data which allowed these examples to be created.** During a meeting of a group of the members of the American Physical Society, they visited the Six Flags Theme Park located in Largo, Maryland (just east of Washington, DC). While there, one of their members wore a vest containing accelerometers that measured acceleration in three axes (x, y and z) and an instrument that recorded altitude. These data sets were converted into ASCII text files. Here are the first ten lines of one data set measuring the acceleration in the y-axis (the up & down

acceleration).

```
Acceleration, Y, acceleration Y
Time ( s )  Acceleration, Y ( m/s/s )
0.0000      8.1
0.0500      7.9
0.1000      8.1
0.1500      8.0
0.2000      8.1
0.2500      7.9
0.3000      8.2
0.3500      8.2
```

The first column is the relative time, in seconds. The second column is the acceleration in m/s^2 (meters per second squared or m/s/s). The normal acceleration due to gravity is roughly 9.8 m/s^2. Therefore, there are times when the wearer of the vest, none other than "Major Laser", was feeling less than his usual weight.

Back to our example. We have a text file that contains two columns (separated by a tab character) of numbers. We're going to use the **dlmread** function to read in this data.

The general syntax for this function is:

```
data=dlmread(filename,<separator>,<range>)
```

where: <separator> is one of the character strings shown in the table below.
<range> is the range of columns to read in.

There are four different separation characters that this function can use. These are:

Separator	Character
Comma	`','`
Semi-colon	`';'`
Colon	`':'`
Tab	`char(9)`

Caution - The Tab Delimiter on the dlmread Function

There appears to be a problem for the tab delimiter on the **dlmread** function. The Freemat Documentation states that its supposed to be a whitespace. This should be represented by two single quote marks with a space between them (e.g. ' '). However, this does not appear to work. Therefore, instead of using the whitespace, use the statement char(9), which creates the tab character using the **char** command and the decimal ASCII code for the tab character, which is 9.

Example - Reading in a Two-Column ASCII File

In this example, we'll read in a two-column ASCII file. The columns are separated by the tab character. As discussed above, we'll use the statement **`char(9)`** to represent the tab within the dlmread function.

```
clear all;
close('all');
data=real(dlmread('supermanAccelY1.txt',char(9)));
t=data(:1,);
accel=data(:,2);
plot(t,accel);
```

When reading in the data, the **dlmread** function defaults to setting the numeric type to a double complex. In this case, I used the **real** function to convert the numbers from complex to real. Also note the use of the **`char(9)`** statement, which denotes the tab character. This particular data set is tab-delimited.
Saving this script as **plotAccelData.m** and running it, we get the result shown in Figure 36.

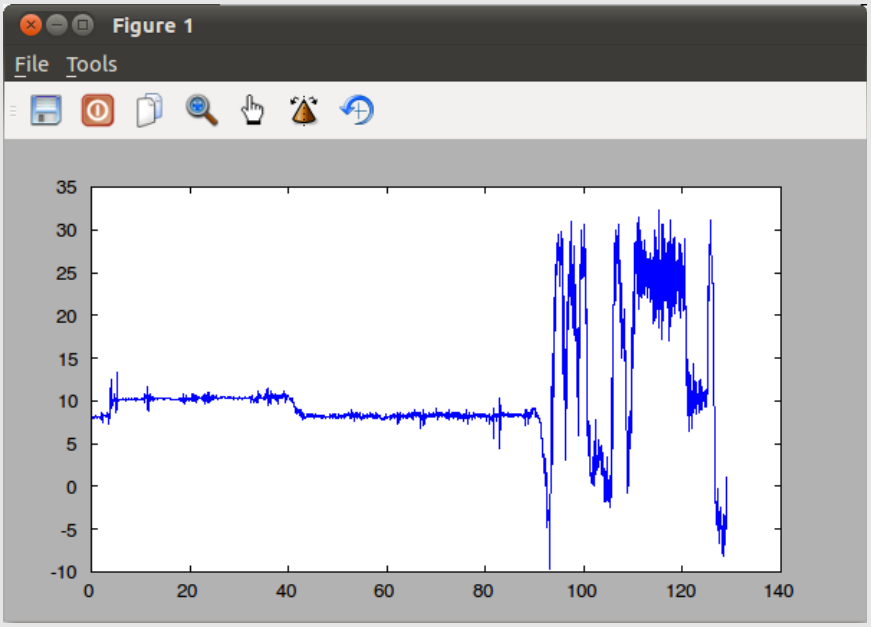


Figure 36: Plot of acceleration data from the "Superman" ride at the Six Flags theme park in Largo, Maryland. (With thanks to the American Physical Society Public Outreach Program.)

Topic 4.11: The File Read & Write Functions

Example - Reading in from a File

Here's an example showing how to use the **fread** function to read in an ASCII-text file. The text file contains the first sentence of the U.S. Declaration of Independence, which is as follows:
When in the Course of human events it becomes necessary for one people to dissolve the political bands which have connected them with another and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.
This leads to the following script, which will read in each character into the variable *charVals*. Each character will be an 8-bit binary value, which means it can cover all possible ASCII values. The whole variable will then be converted to a string using the string command, which converts from character values (0 - 255) to the ASCII equivalent.

```
% Read in ASCII text from standard text file.
clear all;
fp=fopen('DoI.txt','r');
```

```
charVals=fread(fp,[1,inf],'int8');
doiText=string(charVals);
printf([charVals '\n']);
```

Saving this file as readDoi.m and running it, we get:

```
--> readDoi
When in the Course of human events it becomes necessary for one people to dissolve
the political bands which have connected them with another and to assume among the
powers of the earth, the separate and equal station to which the Laws of Nature and
of Nature's God entitle them, a decent respect to the opinions of mankind requires
that they should declare the causes which impel them to the separation.
-->
```

Topic 5: Flow Control

This section will demonstrate different forms of flow control. This refers to the different ways in which the flow of a script, a function, or just a setof commands can be changed according to certain conditions that you specify. The specific flow control operators we'll look at are **for** and **while** loops as well as **if-elseif-else** statements.

Topic 5.1: For Loops

The **for** loop (FD, p. 96) is used when you need to run a loop a fixed number of times. The format for its use is:

```
for (variable=expression);
      statements
end
```

The part within the parentheses will be a counter. Freemat does not require the parentheses. Personally, I find the code more readable using the parentheses.

Example - Using a For Loop to Calculate Altitude of a Projectile

This short script (saved under the filename *projectile.m*) will calculate and display the vertical height (altitude) of a projectile launched straight up. The **for** loop will be used to generate a time counter in 1 second intervals. Then, each second, the new altitude will be calculated.

```
% Calculating vertical height (altitude) for a projectile
%
% This script will calculate and display the height at 1 second
% intervals until the projectile reaches zero altitude.

velocity=100; % vertical velocity in meters/second.
zeroTime=velocity/9.8; % the time needed to achieve maximum height
maxHeight=4.9*(zeroTime)^2; % the maximum height, in meters, of the projectile

printf('The projectile initial velocity is %.2f meters/second.\n',velocity);
printf('The maximum height is %.3f meters.\n',maxHeight);
printf('The time to maximum height is %.2f seconds\n\n',zeroTime);

totalTime=ceil(2*zeroTime);   % This is the total time the projectile will be
                              % airborne
for (k=1:totalTime); % This for loop uses 1 second increments
   distance=velocity*k-(4.9*k^2);
   printf('Height at %d seconds is %.4f meters.\n',k,distance);
end
```

The output from this script appears as follows:

```
--> projectile
The projectile initial velocity is 100.00 meters/second.
The maximum height is 510.204 meters.
The time to maximum height is 10.20 seconds

Height at 1 seconds is 95.1000 meters.
Height at 2 seconds is 180.4000 meters.
```

```
Height at 3 seconds is 255.9000 meters.
Height at 4 seconds is 321.6000 meters.
Height at 5 seconds is 377.5000 meters.
Height at 6 seconds is 423.6000 meters.
Height at 7 seconds is 459.9000 meters.
Height at 8 seconds is 486.4000 meters.
Height at 9 seconds is 503.1000 meters.
Height at 10 seconds is 510.0000 meters.
Height at 11 seconds is 507.1000 meters.
Height at 12 seconds is 494.4000 meters.
Height at 13 seconds is 471.9000 meters.
Height at 14 seconds is 439.6000 meters.
Height at 15 seconds is 397.5000 meters.
Height at 16 seconds is 345.6000 meters.
Height at 17 seconds is 283.9000 meters.
Height at 18 seconds is 212.4000 meters.
Height at 19 seconds is 131.1000 meters.
Height at 20 seconds is 40.0000 meters.
Height at 21 seconds is -60.9000 meters.
-->
```

Example - Calculating the Sine & Cosine Function Using a For Loop

If you're relatively new to the engineering, physics or math fields and are curious as to why we use radians (when most people are probably more comfortable with degrees for angles), it's partly based on how we actually calculate the value of the sine (sin), cosine (cos) and tangent (tan) functions. The calculations are as follows:

$\sin(x)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\frac{x^9}{9!}\cdots$, where x = angle in radians.

$\cos(x)=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\frac{x^8}{8!}\cdots$, where x = angle in radians.

$\tan(x)=\frac{\sin(x)}{\cos(x)}$, where x = angle in radians.

This example will show how to use a for loop to calculate the sine and cosine functions. Even though the series theoretically go to infinity, we don't need that many products to get a fairly accurate value. In the scripts shown below, we'll only use 8 products from each series to calculate an approximate answer.

```
% Calculate sin(x) using the Taylor series
% and a for loop.
clear all;
x=input('Angle = '); % Input the angle in radians
c=1; % Use this to alternate between adding and subtracting
totalSum=0; % Use this to keep a running tally.
for(ii=1:2:15);
totalSum=totalSum+(x^ii)/fact(ii)*c; % Calculate for each part of the series
c=c*(-1); % Alternate between adding and subtracting
end
```

```
totalSum % Display the final sum
```
Saving this as the script **mySin.m**, running it, and entering x=3, we get this:
```
--> mySin
Angle = 3
ans =
    0.1411
```
Now let's compare that with the value when using the sin() function as provided in Freemat.
```
--> sin(3)
ans =
    0.1411
-->
```
They're pretty close! We can use the **format** command to show more decimal places to get a better idea of how well they match.
```
--> format long
--> mySin
Angle = 3
ans =
   0.14111965434119
--> sin(3)
ans =
   0.14112000805987
--> ans-totalSum
ans =
 3.53718673018477e-07
-->
```
They match to within 3.5e-7. That's pretty close. By adding more products to the series, we can make it an even closer match.
The same concept holds for the cosine function. We can calculate it using the series.
```
% Calculate cos(x) using series
clear all;
x=input('Angle = '); % Enter the angle in radians.
c=1; % Use this to alternate between adding and subtracting series products.
totalSum=0; % Use this to track the total sum of the series
for(ii=0:2:14);
totalSum=totalSum+(x^ii)/fact(ii)*c; % Calculate each product in the series
c=c*(-1); % Alternate between adding and subtracting.
end
totalSum % Display the final sum.
```
Saving this as **myCos.m**, running it, and entering x=3, we get this:
```
--> myCos
Angle = 3
ans =
  -0.98999449490242
```
Now compare to the value from the actual cosine function:
```
--> cos(3)
ans =
  -0.98999249660045
-->
```
Again, they're quite close. Mind you, what Freemat uses is a bit more sophisticated than the simple scripts shown above. There are some tricks that help with large angles (What if the angle is 1000 radians? The series above would get out of control quickly.) as well as with the accuracy of small angles.
And what of the tangent function? Is there a special series for that? Nope. The tangent function

requires calculating the sine and cosine functions and dividing the former by the latter.

Topic 5.2: Comparison / Equality Operators

Before going into **while** loops (FD, p. 105) and **if-elseif-else** statements (FD, p. 97), I'll briefly cover the comparison & equality operators. Such operators and comparisons are used to test values for equality or (more likely) inequality. Freemat provides six (6) comparison operators. They are:

- **Exactly Equal (==)**
- **Not Equal (~=)**
- **Less Than (<)**
- **Greater Than (>)**
- **Less Than Or Equal To (<=)**
- **Greater Than Or Equal To (>=)**

While we humans think of a comparison as either *true* or *false*, computers think of them as 1 or 0, respectively. These comparison operators return a value of either 0 (if the comparison is false) or 1 (if the comparison is true).

```
--> 5>1
ans =
 1
--> 5<1
ans =
 0
--> 6*5==30
ans =
 1
--> x=1:10
x =
  1  2  3  4  5  6  7  8  9 10
--> x<=3
ans =
 1 1 1 0 0 0 0 0 0 0
```

We can also use the AND and OR operations (though not these specific commands). The AND is done by multiplying the comparisons together, and the OR by adding them together.

```
AND: <comparison #1>*<comparison #2>
OR: <comparison #1>+<comparison #2>
```

Here are a few examples:

```
--> (125^(1/3)<7)*(24/3==8)
ans =
 1
--> x=1:5;
--> y=1:5;
--> (x<2).*(y>4)
ans =
 0 0 0 0 0
--> (x<=2).*(y<4)
ans =
 1 1 0 0 0
```

Topic 5.3: While Loops

A **while** loop (FD, p. 105) is one which is run so long as the value in **while** command is a positive number. Despite the fact that true comparison operators return a value of 1, the **while** loop (as well as the **if** statement) only requires that the value in the loop be a positive number. The **for** loop uses an array; once the end of the array is reached, the loop stops. The while loop, on the other hand, will run until the value in the statement equals 0. The typical use of the while loop involves some form of comparison operation.

```
while (comparison operation);
      statements
end
```

Example - Calculating the Greatest Common Denominator

This example of the **while** loop will create a function to calculate the greatest common divisor of two integers. This was adapted from a C code function as provided in "Programming in C" by Stephen G. Kochen (Third Edition, 2005).

```
function gcd_value=gcd(n,m)
   while (m ~= 0);
      temp = mod(n,m);
      n = m;
      m = temp;
   end
   gcd_value=n;
```

It works by calculating the remainder of *n* modulo *m* using the **mod** command (FD, p. 173). It then uses the remainder to home in on the greatest common denominator. The while loop keeps running until the remainder of the modulo operation is 0. This algorithm is called a Euclidean version of the greatest common divisor algorithm. I saved it under the filename of **gcd.m**. Thus, if I want to calculate the greatest common divisor of 100 & 36, the calculation would appear as follows:

```
--> gcd(36,100)
ans =
4
```

This says that the greatest common divisor of 36 and 100 is 4.

Example - Calculating the Cube Root

Here's a short function that will demonstrate the **while** loop to calculate the cube root ($x^{1/3}$) of a number. This function uses an iterative loop to converge on the answer. Starting with a "guess", which will be s(1), this function calculates successive values using this calculation.

$$s(n+1)=\frac{1}{2}\left(s(n)+\frac{x}{s(n)^2}\right)$$

Here's how I coded this function.

```
% cubeRoot.m
function returnValue=cubeRoot(x)
n=1;
s(n)=1; % Initial guess
```

```
diffS=1; % Difference between successive values
epsilon=1e-8; % Desired precision of the answer.
while(diffS>abs(epsilon*s(n)));
s(n+1)=(1/2)*(s(n)+x/(s(n)^2));
n=n+1;
diffS=abs(s(n)-s(n-1));
end
returnValue=s;
```

This function returns an array of successive values starting with the initial guess, which is 1. The **while** command looks at whether the difference is less than the precision times the last value of the loop. Running it to calculate the cube root of 2, we get this:

```
--> format long
--> y=cubeRoot(2);
--> y'
ans =
   1.00000000000000
   1.50000000000000
   1.19444444444444
   1.29814163812271
   1.24248215661613
   1.26900936034694
   1.25547429372185
   1.26216808077850
   1.25880353145720
   1.26048129769035
   1.25964129946292
   1.26006101831131
   1.25985108900747
   1.25995603616619
   1.25990355821644
   1.25992979609835
   1.25991667688420
   1.25992323642298
   1.25991995663651
   1.25992159652548
   1.25992077657993
   1.25992118655243
   1.25992098156611
   1.25992108405926
   1.25992103281268
   1.25992105843597
   1.25992104562433
   1.25992105203015
```

While it takes several loops to get a fairly precise answer, we can adjust the script such that the last line only returns the last value. The last line now appears as follows:

```
returnValue=s(n);
```

Running this modified function with for a value of 5, we get this:

```
--> cubeRoot(5)
ans =
   1.70997595038430
```

Here's a short script that demonstrates that the while loop only requires a positive value to operate.

```
% testWhile.m
```

```
n=5;
while(n);
printf('The number is %d.\n',n);
n=n-1;
end
```

Running this script, we get this:

```
--> testWhile
The number is 5.
The number is 4.
The number is 3.
The number is 2.
The number is 1.
```

Topic 5.4: If-Elseif-Else Statements

This is similar to the **if-then-else** statements many programmers are familiar with. The **if** command (FD, p. 97) uses a comparison operator. So long as the comparison produces a positive value, the statements following the **if** command will be run. If so, it runs the commands until the **end** statement is reached. The format for its use is as follows:

```
if (comparison operation);
   statement
elseif (comparison operation)
   statement
else (comparison operation)
   statement
end
```

The most basic statement has just one decision using **if**, such as follows:

```
if (comparison operation)
   statement
end
```

Example - If Statement

This simple example is a script that compares two numbers and prints the larger of the two.

```
x=5;
y=3;
greaterNumber=x;
if (x<y);
   greaterNumber=y;
end
greaterNumber
```

In this example, the two variables are *x* and *y*. Before the statement, the program makes the assumption that *x* is larger. The *if* statement tests to see if *y* is greater. If it is, it executes the statement in which *y* is set as the larger number. I saved the script under the filename **greaterNumber.m**. To run it, simply type in the name (minus the ".m"):

```
--> greaterNumber
ans =
5
```

Topic 6: Graphs & Plots

There are two commands available with which to create 2D (two-dimensional) graphs. These are the **plot** command (FD, p. 478) and the **line** command (FD, p. 474). There are differences between these two commands, which we'll briefly cover.

- The **plot** command is solely for 2D graphs. (Note: There is a separate command, **plot3** (FD, p. 482), for graphing three-dimensional graphs.) The plot command will create a figure window (if one is not already opened) or overwrite the current figure window (if it's already opened and if the hold command is not used). The **plot** command allows one to use either one, two or three vectors, related to x, y and z, respectively. If only one vector is provided, it will draw a 2-dimensional graph using the vector values for the y-axis and the counter of the array for the x-axis. The plot command can graph multiple arrays at the same time and will color them differently according to the colororder property.
- The **line** command can do either 2D or 3D graphs. As a matter of fact, it always requires providing at least an x- and y-axis vector set. Further, if you wish to set the various properties (line color, linewidth, line style), you have to provide an x-, y- and z-axis vector set. The **line** command will create a figure window (if one is not already opened) or add another line to an already-existing graph. The **line** command can only plot one graph, but provides greater flexibility in setting non-standard line colors than does the **plot** command.

To me, the big differences between the two commands is that the **plot** command will overwrite any other existing graphs, whereas the **line** command simply adds another graph to an already-existing graph. The **line** command is a bit more kludgy to use because of the fact that you must use all three vectors in order to have any control over the line properties, such as linewidth or color. You can use the **line** command with just an x-array and y-array, but it only gives you a simple, thin, black line.

When first invoked, either the **plot** command or the **line** command creates a figure window that is 600 x 400 pixels. The area minus the toolbars is 600 x 335 (for the Linux version of Freemat) or 600 x 344 (for the Windows version). The actual plot itself, which I call the *plot area*, is roughly 480 x 268, or roughly 80% of the full plot dimensions.

Topic 6.1: Creating a Graph or a Plot

You can create several different types of graphs with Freemat. Depending on the specific command you use, you can graph one, two or three variables as y, XY, and XYZ graphs, respectively. Using a single variable gives you a plot of that variable along a straight line. Two variables provides an xy graph in which you can create all kinds of two-dimensional shapes. And an xyz graph allows you to create three-dimensional shapes. For this book, we're going to focus on the single variable and two-dimensional, xy graphs. We'll cover 3D graphs in the future. shows

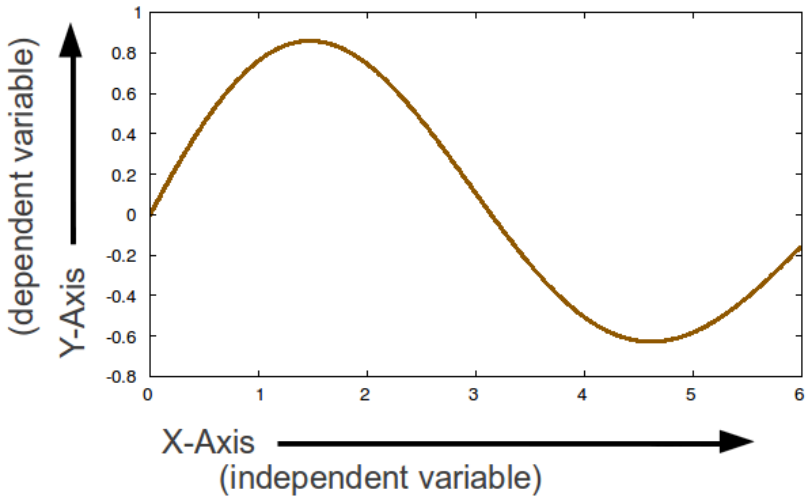


Figure 37: This is a basic overview of the parts of a XY graph or plot. The horizontal axis or ordinate is called the "x-axis" (since the variable most often used on this axis is the letter "x"). The vertical axis. or abscissa, is the "y-axis". In cases of three-dimensional plots, the third axis is called (appropriately enough) the "z-axis".

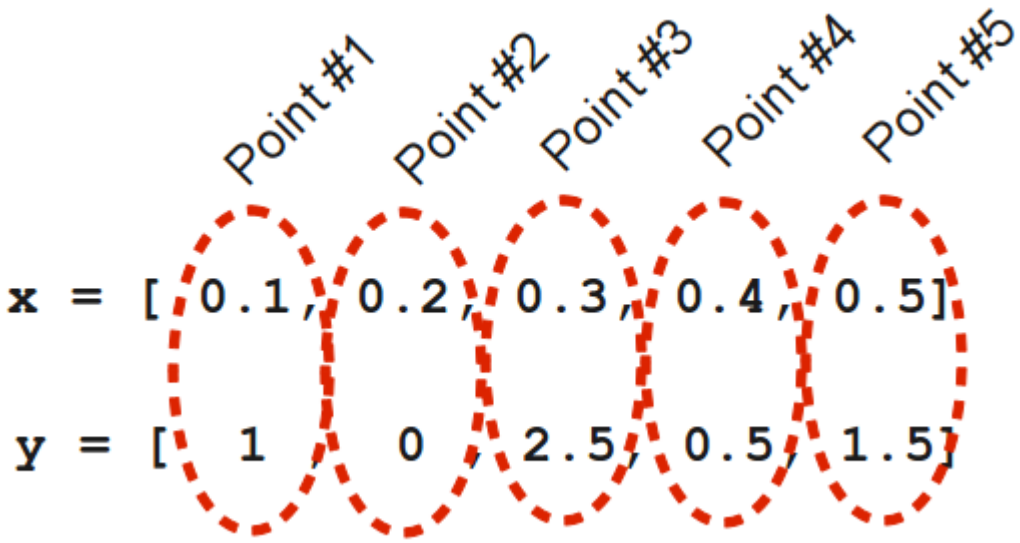
Regardless of the type of graph being created, the basic principle is the same. Freemat uses each point in the x-axis vector and each point in the y-axis vector and places them on the graph. If the linestyle property specified is a form of line (solid line, dotted or dashed line), it will then create that type of line between each point. If the linestyle is specified as a marker (dot, square, diamond, etc), it will simply put the specified marker at each point.

For example, let's look at two vectors of five points each. We'll use the following arrays:

```
x = [ 0.1, 0.2, 0.3, 0.4, 0.5]
```

```
y = [  1 ,  0 , 2.5, 0.5, 1.5]
```

If we look at each point as an XY pair, then we wind up with the following:



Note that we have the same number of elements in both arrays. If we try to make a plot with different size arrays, Freemat will simply return a blank figure window. You will not receive an error nor any other indication as to why the plot is blank.
Back to this concept, though. Compare this with the resulting plot, as shown in Figure 38.

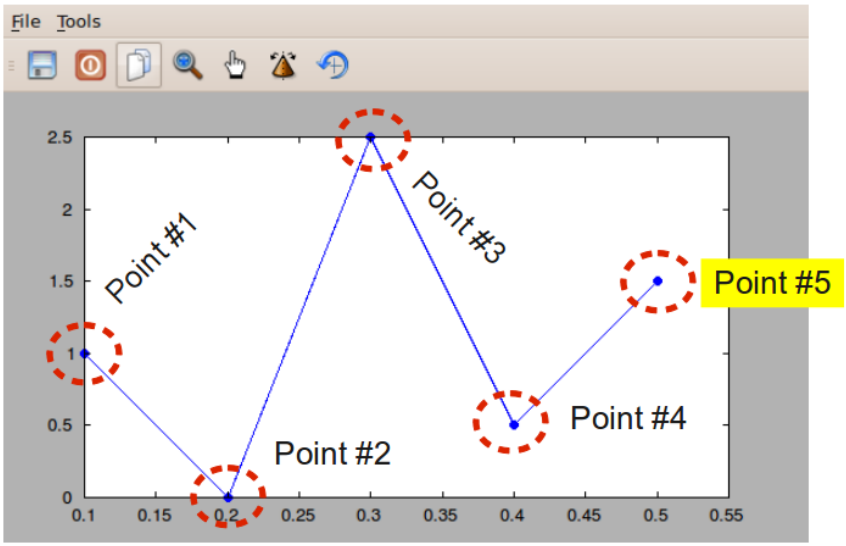


Figure 38: Five point graph showing how points are plotted and connected.

What Freemat does here is that it plots each XY point, then draws a connecting line between consecutive points. In the cases where the x-axis vector is not specified, Freemat will show the x-axis as the index counter used for the y-axis array.

Topic 6.1.1: Creating a Graph of a Single Variable

The most basic graph uses the **plot** command, as follows:

plot(y)

where: y = the name of the variable containing the array to be plotted.

You can also use the **line** command, although it requires a minimum of an x-axis and y-axis vector set, as follows:

line(x,y)

where: x = the variable containing the x-axis values.
y = the variable containing the y-axis values. (NOTE: The x-axis and y-axis vector sets must have the same dimensions. If not, the graph will not be visible.)

When you create a plot, it's put into a separate window that has a number. By default, the first figure will be Figure 1. Any further plots will be put into this same window unless you explicitly tell Freemat to put them into different windows. If you use the **hold** command (FD, p. 466), you can place multiple graphs in the same window. Otherwise, the original graph will be erased and the new one put in its place. More on that later.
Let's start by looking at the plot of a single variable, an array with one dimension.

Example - Creating a Graph of a Single Variable

The first example will use the **plot** command.

```
t=linspace(0,8*pi,256);
y=sin(t);
plot(y)
```

The resulting graph is shown in Figure 39. Note that the x-axis (also called the "independent variable axis") is the index counter used for the variable "y". What Freemat does when plotting single variables is that it creates a x-axis array using the index counter, hence the reason that the x-axis runs from 1 to 256.

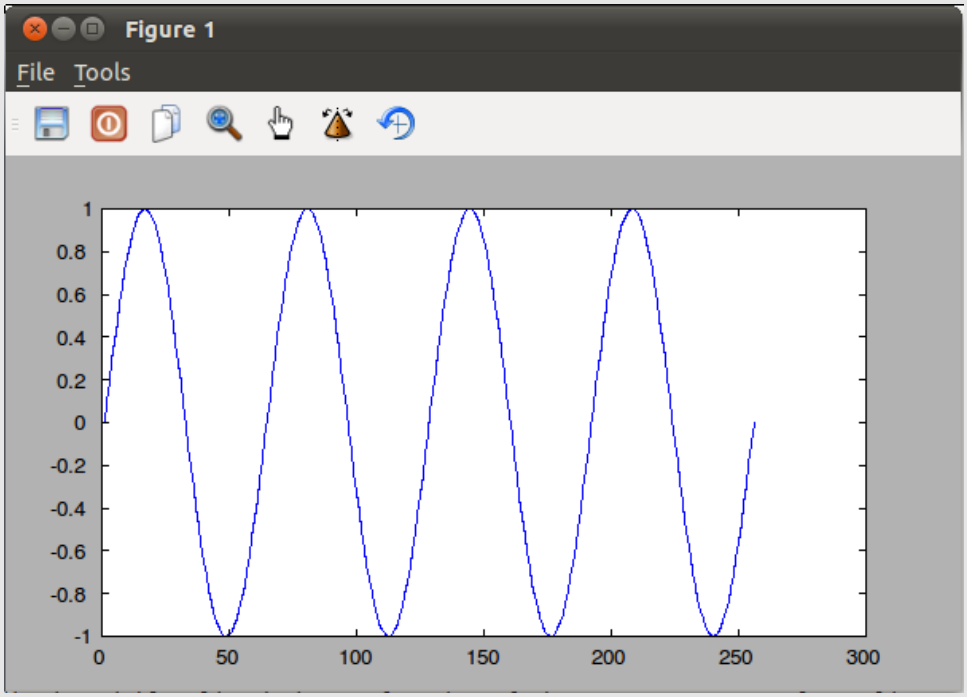


*Figure 39: Graph of a single variable using the **plot** command.*

The second example will use the **line** command. Note that the **line** command always requires at least using the x- and y-axis variables.

```
t=linspace(0,8*pi,256);
y=sin(t);
line(t,y);
```

The resulting graph is shown in Figure 40.

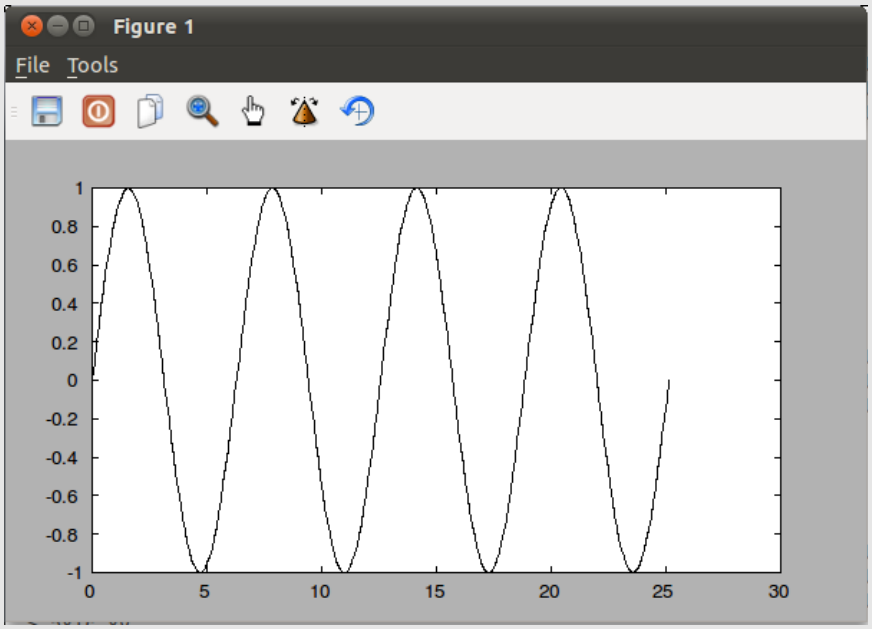


Figure 40: Basic graph of a single variable using the line command.

In order to create an XY graph, you need to have two arrays that have the same length and then use the following format for the **plot** command:

plot(x,y)

where: x = the variable containing the format for the x-axis (horizontal) direction
y = the variable containing the format for the y-axis (vertical) direction

The **line** command is similar:

line(x,y)

where: x = the variable containing the format for the x-axis (horizontal) direction
y = the variable containing the format for the y-axis (vertical) direction

Caution - The Size of the X & Y Vectors in the Plot or Line Commands

When making an XY plot, both arrays used to create the X and Y axes must have the same number of elements. Otherwise, the plot will be blank when you actually invoke it. It will not show any error; the plot display will simply be blank.

Example - Making an XY Plot

The first example uses the **plot** command to do nothing more than plot a line using the beginning and endpoints of the line.

```
--> clear all;
--> close('all')
--> line([-2,2.5],[-0.3,1])
```

The result is shown in Figure 41. The ability to make lines between points will allow for the creation of your own axes in any place you want

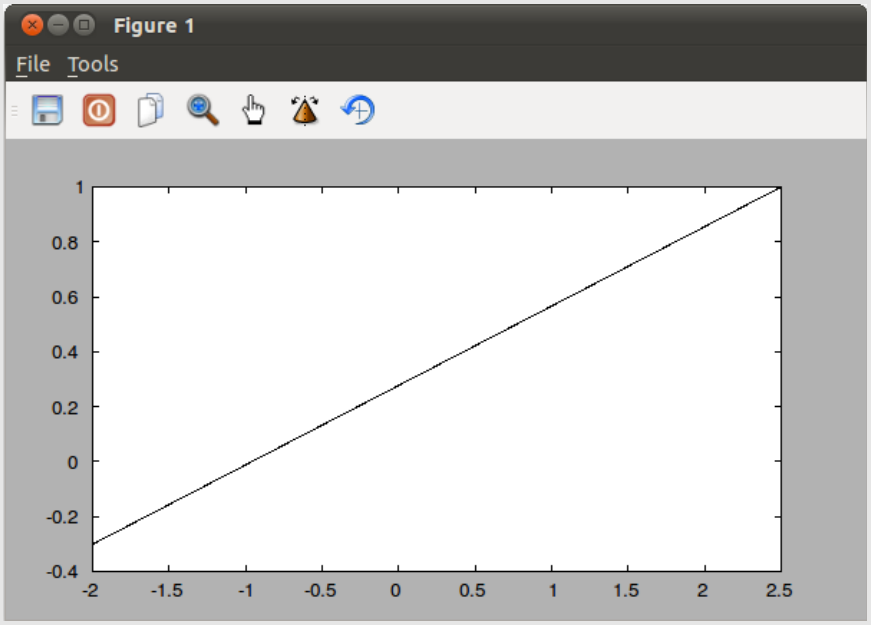


Figure 41: Plot of a line using nothing more than a beginning and end point. This plot goes from the XY point (-2,-0.3) to the point (2.5,1).

The second example shows a cosine wave plotted on the horizontal axis against a sine wave on the vertical axis. The result is a circle.

```
theta=linspace(0,2*pi,500);
x=cos(theta);
y=sin(theta);
plot(x,y)
```

The result is shown in Figure 42.

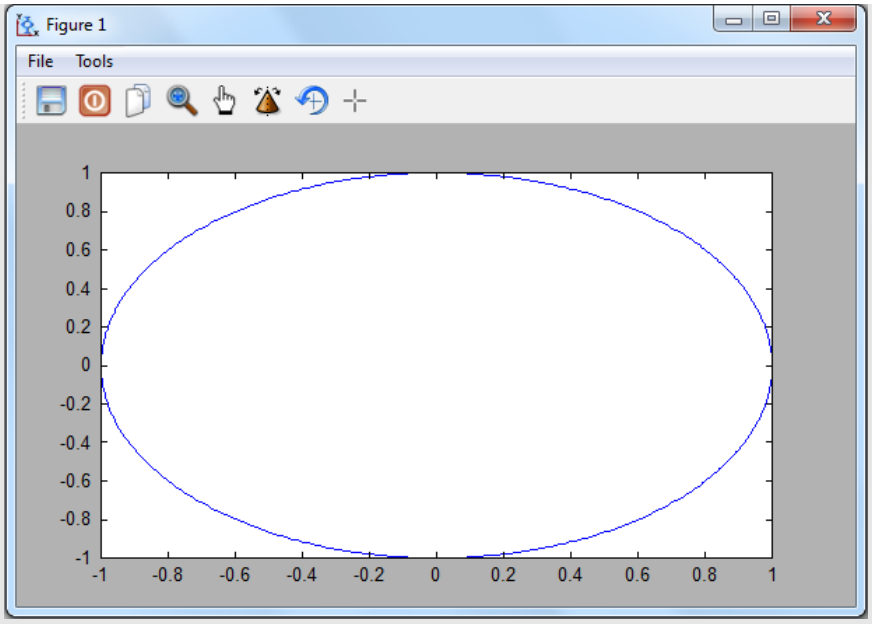


Figure 42: Plot of an X-Y Graph

Here's another example that uses the **line** command to create a five-point star.

```
theta=0:144:720; % The angle change, in degrees, between points.
x=sind(theta); % Use the degree version of the sin function
y=cosd(theta); % Use the degree version of the cos function
clf; % Clear the previous figure, if any.
line(x,y); % Plot the points to create the star.
```

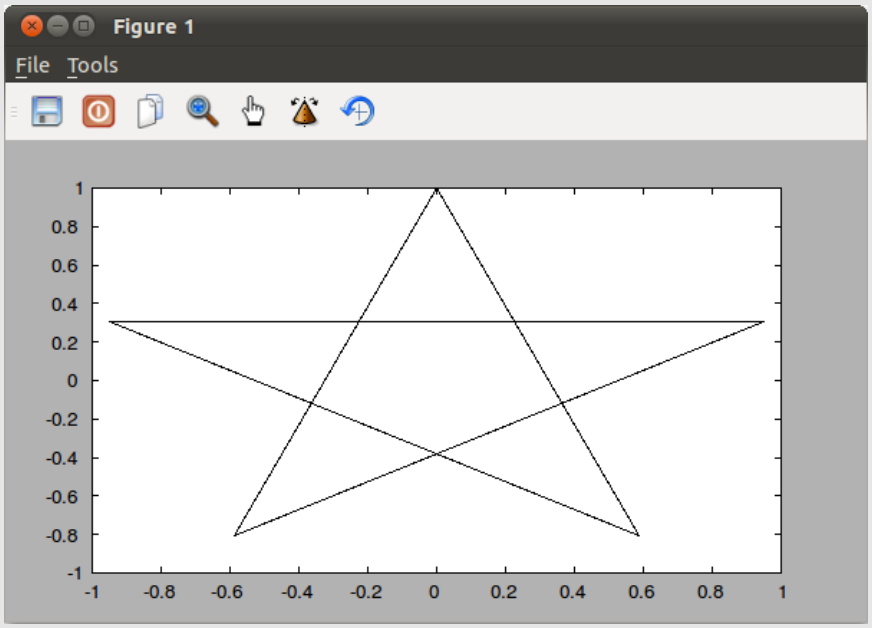


Figure 43: Scatter plot using black dot markers

There is something special you can do for your XY plots that may make them appear better. And that's using the **axis equal** command. This command makes the length of both the X and Y axes the same such that the dimensions of each will be properly proportioned. Simply put, this means Figure 42 will look more like a circle and less as an ellipse. It also means that the star in Figure 43 will appear more

proportionally correct. We'll use the axis equal command on the star to demonstrate.

```
theta=0:144:720; % The angle change, in degrees, between points.
x=sind(theta); % Use the degree version of the sin function
y=cosd(theta); % Use the degree version of the cos function
clf; % Clear the previous figure, if any.
line(x,y); % Plot the points to create the star.
axis equal;
```

Here's the result.

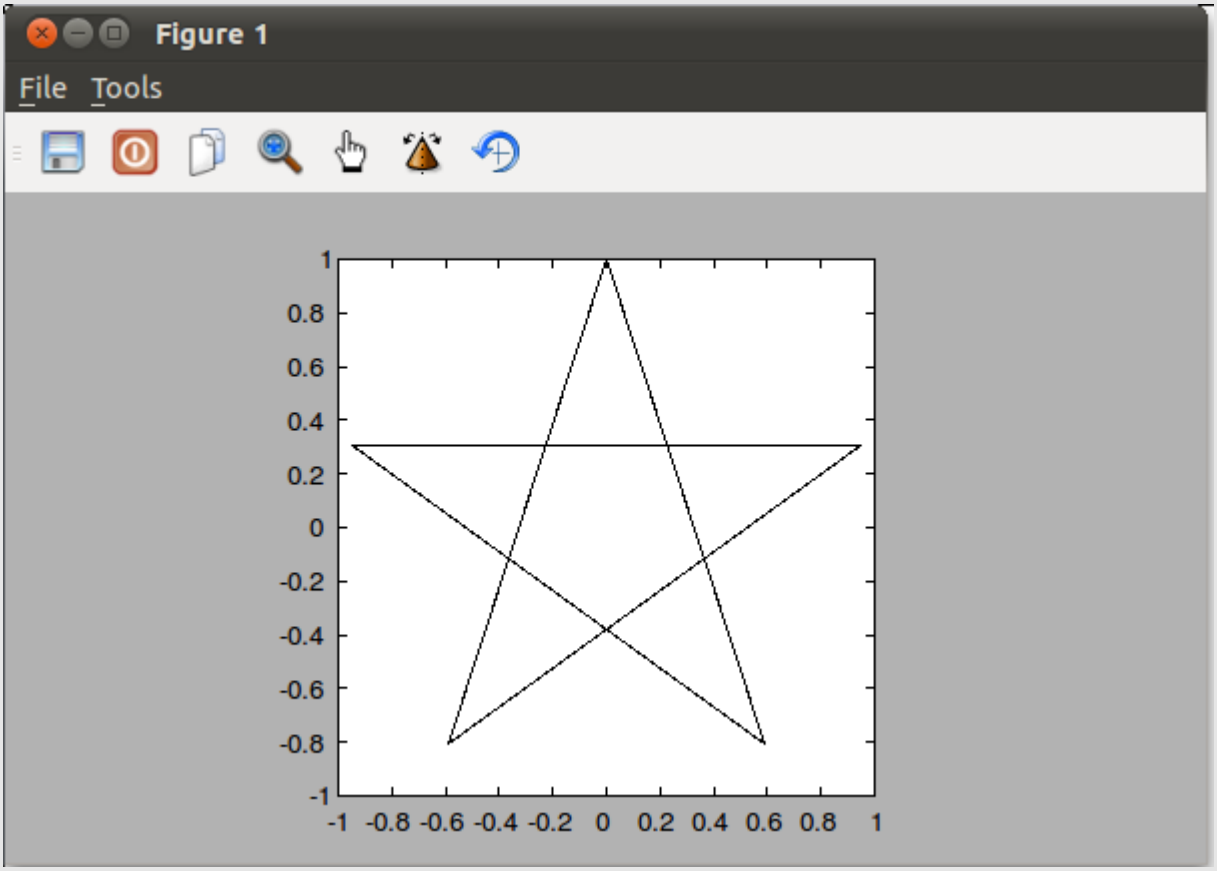


Figure 44: A five-point star created with a line command and re-sized using the axis equal command. Note that it appears more proportionally-correct than the original star (without the axis equal command).

Topic 6.1.2: Creating a Graph of Multiple Variables

There are three ways to create multiple traces in the same figure. These are:

1. Use a single **plot** command with multiple variables.
2. Use multiple **plot** commands along with the **hold** command.
3. Use multiple **line** commands.

Example - Creating a Graph with Multiple Variables

In this first example, we'll use a single plot command with multiple y-axis, or dependent, variables. In order to do so, we have to use XY pairs, hence the use of "x,y1,x,y2,x,y3" as opposed to simply "y1,y2,y3".

```
x=linspace(0,6*pi,600);
y1=sin(x);
y2=sin(x)-1;
y3=sin(x)-2;
plot(x,y1,x,y2,x,y3)
```

The result is shown in Figure 45. Note that we did not define the colors; by default, Freemat assigns colors from a list of seven colors for the plot command. The line command, on the other hand, will always be black unless the explicit color is stated in the line command itself. We'll cover that in the next section when we discuss graph colors.

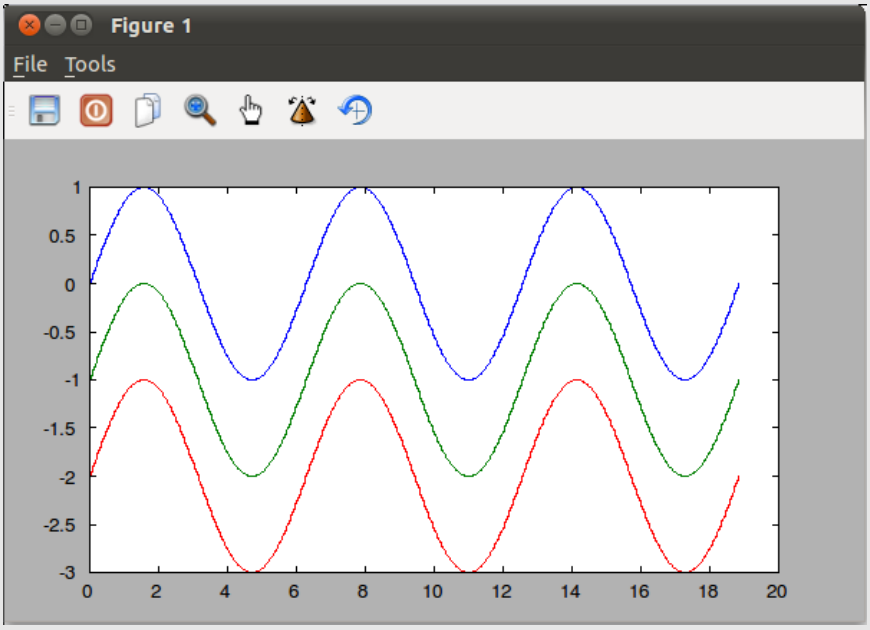


*Figure 45: Graph of three dependent variables using the **plot** command. Note the different colors used for each.*

For the second example, we'll use multiple plot commands. For this, we must use the **hold** command; otherwise, each plot command will simply erase and overwrite the one that came before it.

```
x=linspace(0,6*pi,600);
y1=sin(x);
y2=sin(x)-1;
y3=sin(x)-2;
plot(x,y1);
hold on;
plot(x,y2);
plot(x,y3);
hold off;
```

The result is shown in Figure 46. Note that it is identical to Figure 45. The advantage of this method is that each line can have individual linewidths, colors and types, which is not possible if they are all put into a single **plot** command.

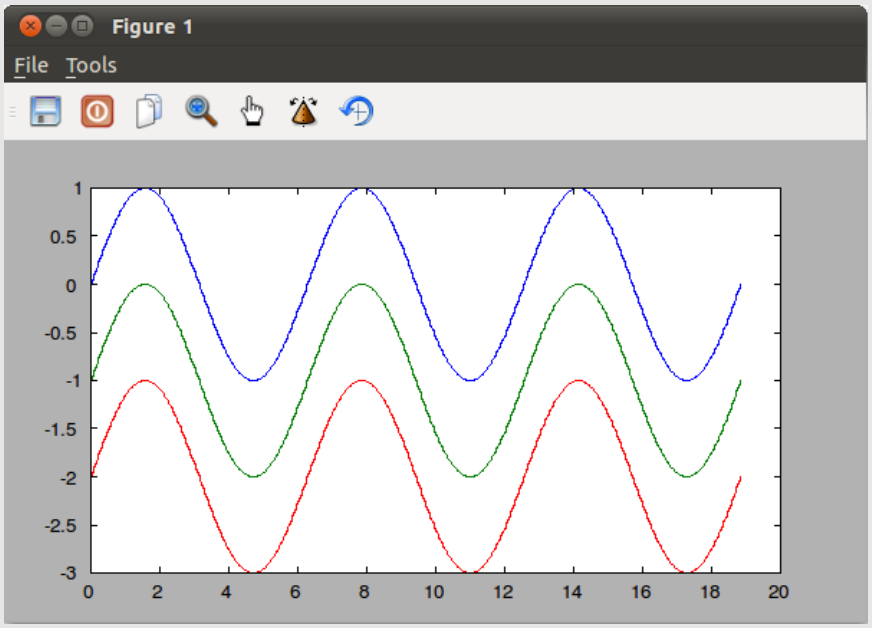


Figure 46: Plot of multiple traces created using multiple plot commands long with the hold command.

In this next example, we'll use the **line** command. Note that the **line** command only allows one trace per command; we have to use multiple commands in order to have multiple traces.

```
clf;
x=linspace(0,6*pi,600);
y1=sin(x);
y2=sin(x)-1;
y3=sin(x)-2;
line(x,y1);
line(x,y2);
line(x,y3);
```

The result is shown in Figure 47. Note that, this time, all of the lines are black. The line command defaults to a 1-pixel wide black line. Also note that the line command always requires at least an XY pair, whereas the plot command may only require a dependent (y-axis) variable.

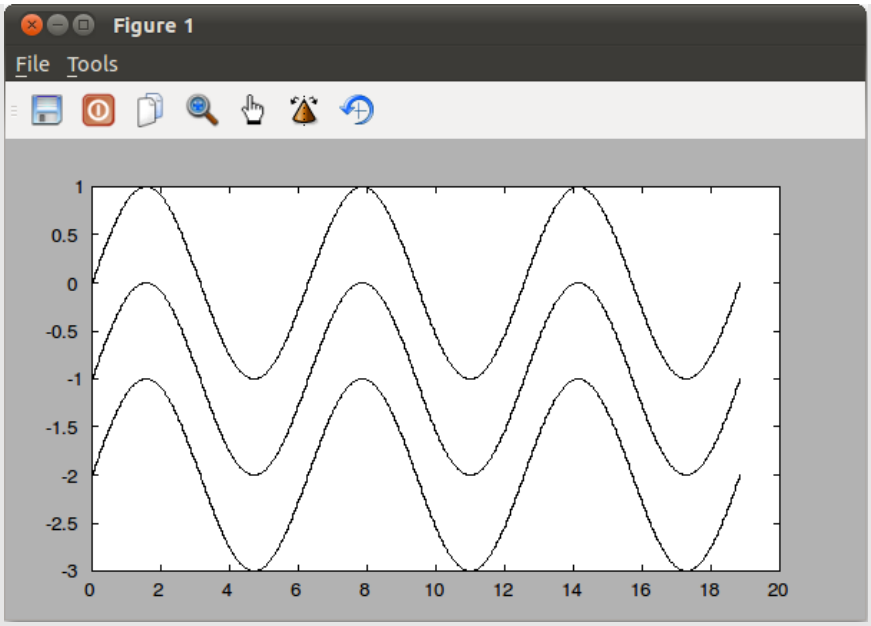


Figure 47: Plot of multiple dependent variables using the line command.

Topic 6.2: Graph Properties

Both the **plot** and **line** commands come with various properties that you can set when you invoke them. This includes such attributes as the color of the line, the width of the line, and the style of the line. Note that the only time possible to set the properties for plot color, line type, and point markers is when invoking the plot function. It is not possible to change these after the graph is created.

Topic 6.2.1: Graph Colors

Freemat provides several mechanisms to change the line color for a graph. First, it provides a set of seven basic colors that can be used as a single-character handle for either the **plot** or **line** commands. These colors are:

- `'b'` - Color Blue
- `'g'` - Color Green
- `'r'` - Color Red
- `'c'` - Color Cyan
- `'m'` - Color Magenta
- `'y'` - Color Yellow
- `'k'` - Color Black

Example - Graphing with Standard Colors

Here are a couple of examples showing use of the standard colors. The first example uses the **plot** command.

```
--> x=linspace(-3,3,1000);
```

```
--> y=exp(-x.^2).*sin(10*pi*x);
--> plot(x,y,'r')
```

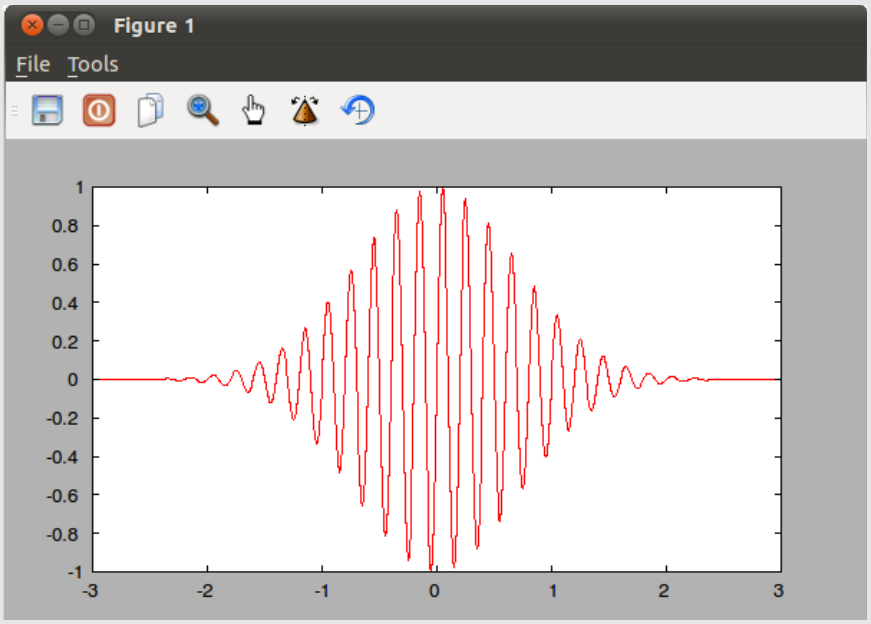


*Figure 48: Graph using a red trace created with the **plot** command.*

Here's another example using the **line** command.

```
--> close('all')
--> x=linspace(-3,3,1000);
--> line(x,y,zeros(1,length(x)),'color','y');
```

Note that, in this case, we added a z-axis array (essentially an array of zeros) since the line command requires an x, y and z array in order to add any line properties (width, color, etc).

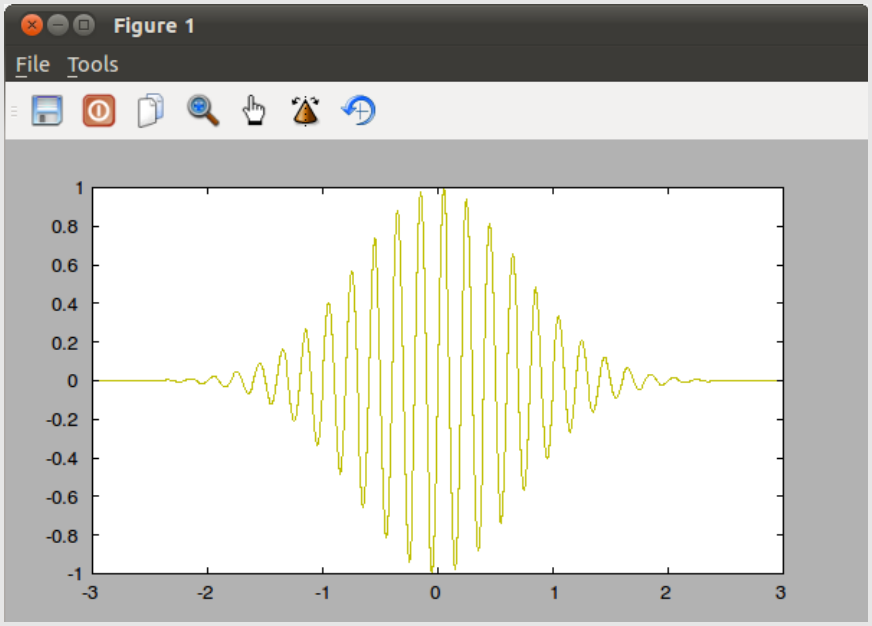


Figure 49: Graph of damped sinusoid created with a yellow trace using the line command.

Next, we'll combine multiple traces on one plot along with using different colors. This is a little trick I developed during the writing of this book. It allows you to create a graph, then highlight a section by "filling" it with a particular color. This works by creating the trace, then creating a zig-zag trace for the highlighed section, as shown in Figure 50. (The original method I developed for this used a **for** loop. That was very slow. This method works quickly and efficiently.)

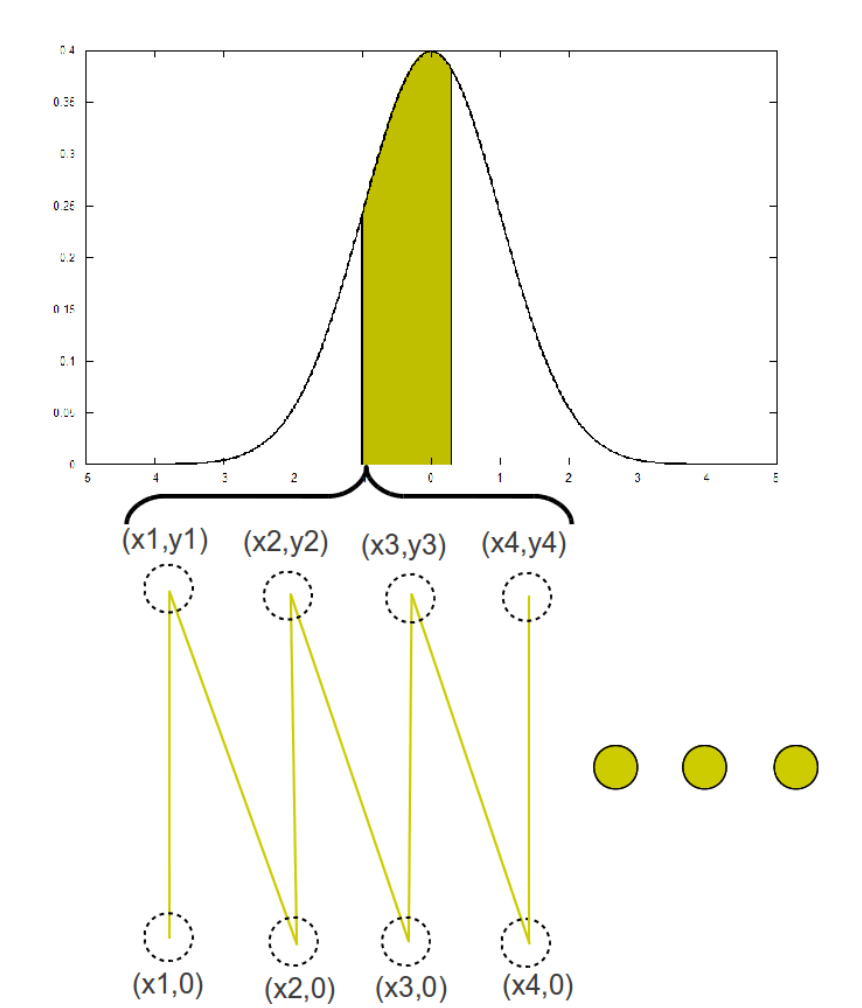


Figure 50: Overview of how to create a highlighted section of a particular graph. This is done by creating a zig-zag trace that starts with the first point on the x-axis at y=0, goes straight up til it reaches the graph, then drops at an angle to the next x-axis point. This is repeated for the entire highlighted section. If each line is right next to the line on each side, it will appear as if it were "filled".

The example we'll use will be similar to that shown above. We'll create a Gaussian curve, then highlight the section from -1 to 0.3, as is done above.

```
clear all;
close('all');
% Create the Gaussian curve as an anonymous function
fcn=@(x) ((1/sqrt(2*pi))*exp((-x.^2)/2));
% Set the various limits for the graph and the highlighted section
```

```
start_x=-3; % The lower limit of the graph
stop_x=3; % The upper limit of the graph
start_point=-1; % Set the lower limit for highlighting
stop_point=0.3; % Set the upper limit for highlighting.
steps=600;
% Create the x- and y-axis arrays for the curve itself.
xx=linspace(start_x,stop_x,steps);
yy=fcn(xx);
% Plot the Gaussian curve.
plot(xx,yy);
% Calculate the x-axis and y-axis values for the zig-zag line.
x=linspace(start_point,stop_point,steps);
y=fcn(x); % Calculate the top of the highlighted section.
c=1:(2*steps); % Counter to create the x-axis values.
c2=2:2:(2*steps); % Counter to create the y-axis values.
xx(c)=x(ceil(c/2)); % Repeat each x-axis value so that each one is doubled.
yy=zeros(1,length(xx)); % Set all y-axis values to zero.
yy(c2)=y(c2/2); % Set every other y-axis value to that of the graph.
line(xx,yy,zeros(1,length(xx)),'color','y'); % Plot the highlighted section
```

Running this script, we get this figure.

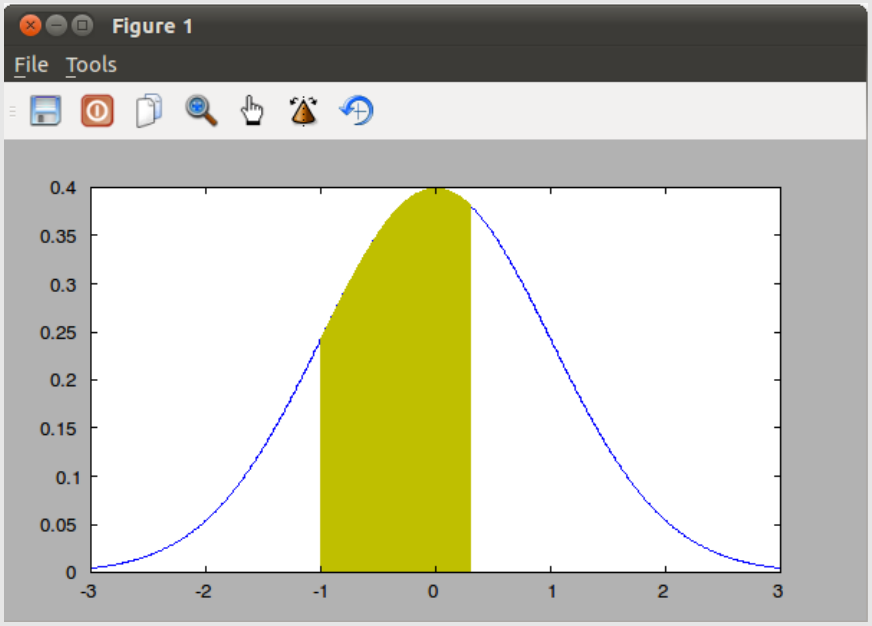


*Figure 51: Graph of a Gaussian curve with the section from -1 to 0.3 highlighted. The highlighted section was created using a single **line** command.*

This technique was used to create many of the images shown in the "Numerical Integration and Differentiation" section.

When graphing multiple traces into the same figure window with a single **plot** command, the **plot** command cycles through these seven colors in the following order: blue, green, red, cyan, magenta, yellow, black. An example of this for the first three colors (blue, green and red) is shown in Figure 45. These are defined using a 7x3 array, with each row having a color balance of RGB (red / green / blue). You can see this using the **get** command with the colororder property:

```
--> close('all')
--> figure(1)
--> get(gca,'colororder')
ans =
         0         0    1.0000
         0    0.5000         0
    1.0000         0         0
         0    0.7500    0.7500
    0.7500         0    0.7500
    0.7500    0.7500         0
    0.2500    0.2500    0.2500
```

Each row is a set of three variables, with each column defining the amount of red, green and blue, respectively, with a 0 meaning no color and a 1 meaning full color.

When you plot a single variable using the **plot** command, it will always use the first color in the array, which defaults to blue ([0 0 1.0]. It is possible to change this array with the **set** command, and we'll cover that later.

It's possible to use non-standard colors with either the **plot** or the **line** commands. Personally, I find the **line** command easier to set for non-standard colors. This is due to the fact that, in order to use non-standard colors with the **plot** command, you first have to open a figure window, turn on hold using the **hold** command (*hold on*), then use the **set** command along with the coloroder property to create the color you desire. *Then*, you can use the **plot** command to actually create the graph. With the line command, on the other hand, you just use the color property in the **line** command itself along with a 3-element array to set your desired color using the standard RGB (red / green / blue) scale.

With the plot command, if the **hold** command is not used before changing the color vector set, the colors will revert to the default set the next time the **plot** command is used. Note that the colororder vector set only applies to the **plot** command; it does not apply to the **line** command. Also, regardless of the color palette set using the colororder property, if one of the specific basic colors is specified in the plot command, it will use that basic color and not one of the colors in the vector set.

Example - Creating Graphs with Non-Standard Colors

The methods to use a non-standard are different depending on whether the **line** command or the **plot** command is used. The methods for both commands are demonstrated below. For each example, we'll create a basic waveform and use a brown-colored trace. The brown color we'll use will have a mix of 144/256 red, 88/256 green and 0 blue.

For the first example, we'll use the **plot** command.

```
% Create a graph trace using a brown-colored line
clear all;
close('all')
x=linspace(0,6,600);
y=x.*sin(10*x).*exp(-x);
figure(1);
hold on;
set(gca,'colororder',[(144/256) (88/256) 0]);
plot(x,y);
hold off;
```

The result of this script is shown in Figure 52.

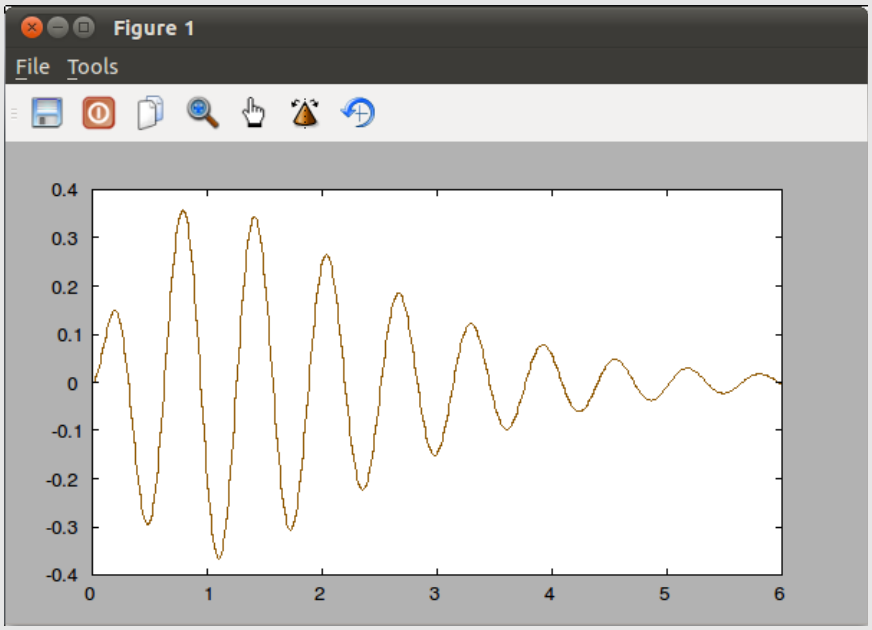


Figure 52: Example of a brown trace using the plot command.

Here's a similar example, but using the **line** command.

```
clear all;
close('all')
x=linspace(0,6,600);
y=x.*sin(10*x).*exp(-x);
z=zeros(1,length(x));
line(x,y,z,'color',[(144/256) (88/256) 0]);
```

The result of this script is identical with the **plot** command, as shown in Figure 53; the only differences are that we had to include a z-axis (set to all zeros) and we didn't have to bother with the **hold** or **set** commands.

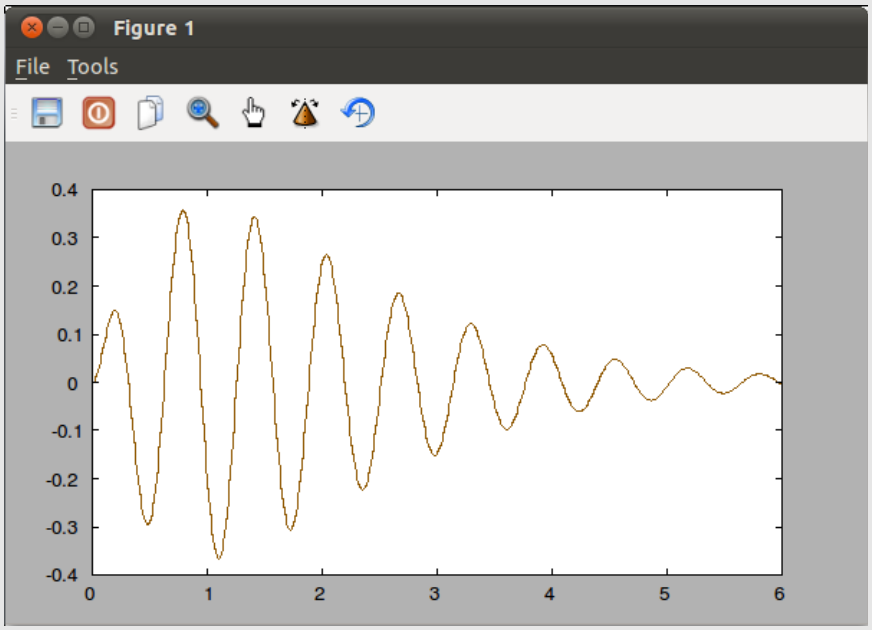


Figure 53: Graph using a brown trace line using the line command.

Topic 6.2.2: Setting the Line Width

The default line created in Freemat is a thin line. You can, however, set a thicker linewidth using the linewidth property. It's syntax for the **plot** command is as follows:

plot(x,'linewidth',n)

where: x = the variable to be graphed
n = an integer from 1 - 32 stating the thickness of the trace line. The higher the integer, the thicker the line.
The syntax for the line command is as follows:
line(x,y,z,'linewidth',n)
where: x, y, z = the respective axes arrays for the x-, y- and z-axes.
n = an integer representing the line width from 1 - 32.
Note that, in Freemat, each individual **plot** command, no matter how many independent variables it is plotting, will only allow one linewidth property. You can set different linewidths to different variables if they're using the same plot, but you have to use separate plot commands combined with the hold command, to do so (see below).

Example - Setting the Line Width

This example will show the differences in line widths for the normal display.

```
x=linspace(0,4*pi,600);
```

```
y=cos(x);
plot(x,y,'linewidth',1);
```

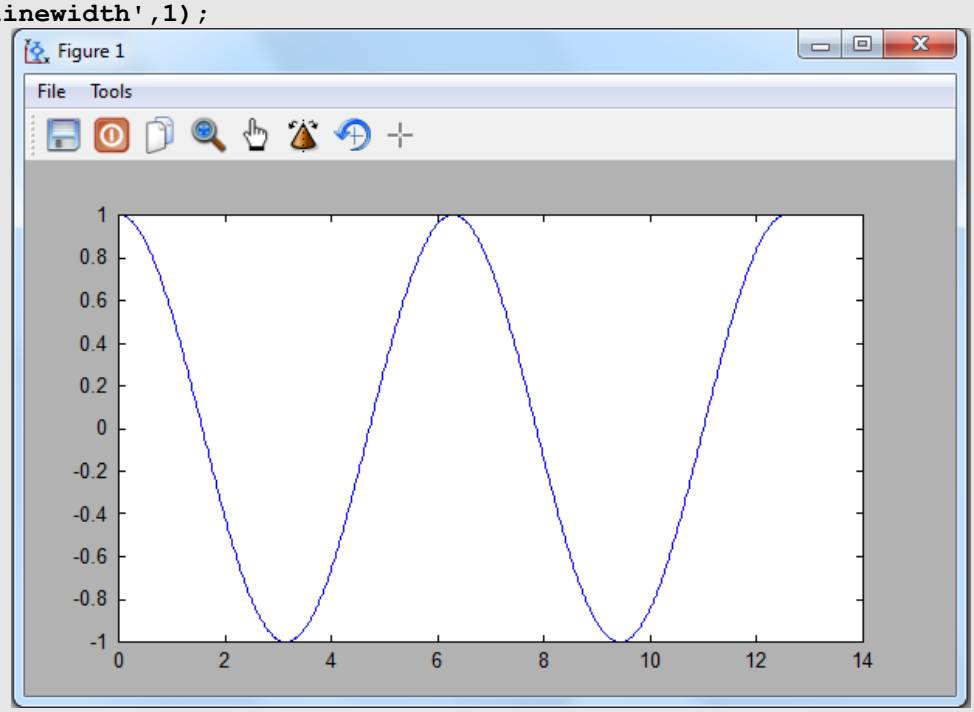


Figure 54: Plot of a sinewave with a linewidth of 1.

```
x=linspace(0,4*pi,600);
y=cos(x);
plot(x,y,'linewidth',2);
```

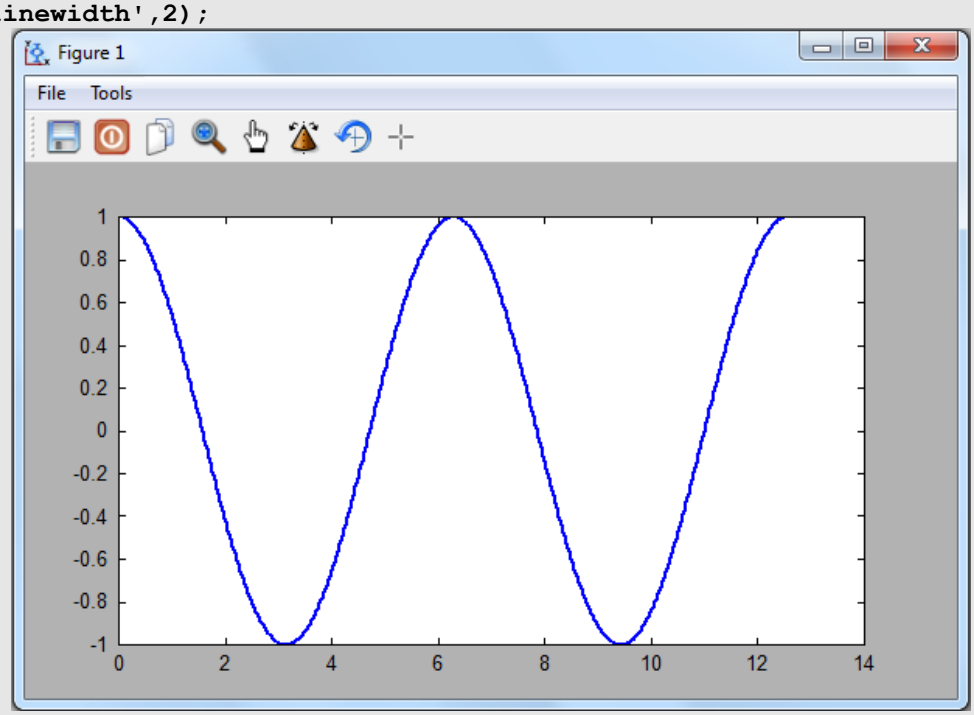


Figure 55: Plot of a sinewave with a linewidth of 2.

```
x=linspace(0,4*pi,600);
y=cos(x);
plot(x,y,'linewidth',6);
```

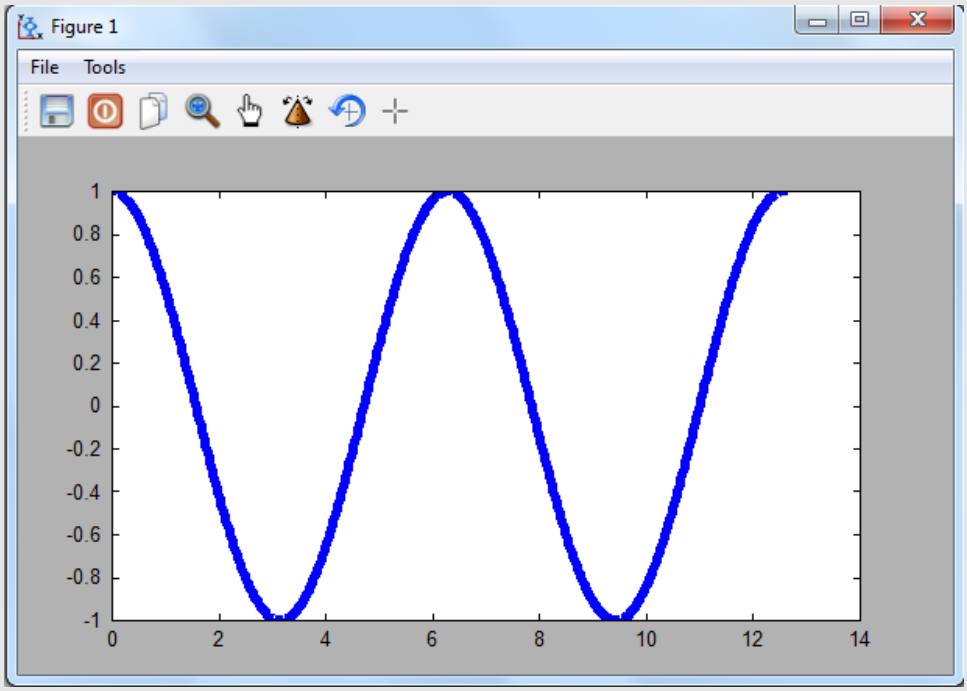


Figure 56: Plot of a sinewave with a linewidth of 6.

```
x=linspace(0,4*pi,600);
y=cos(x);
plot(x,y,'linewidth',32);
```

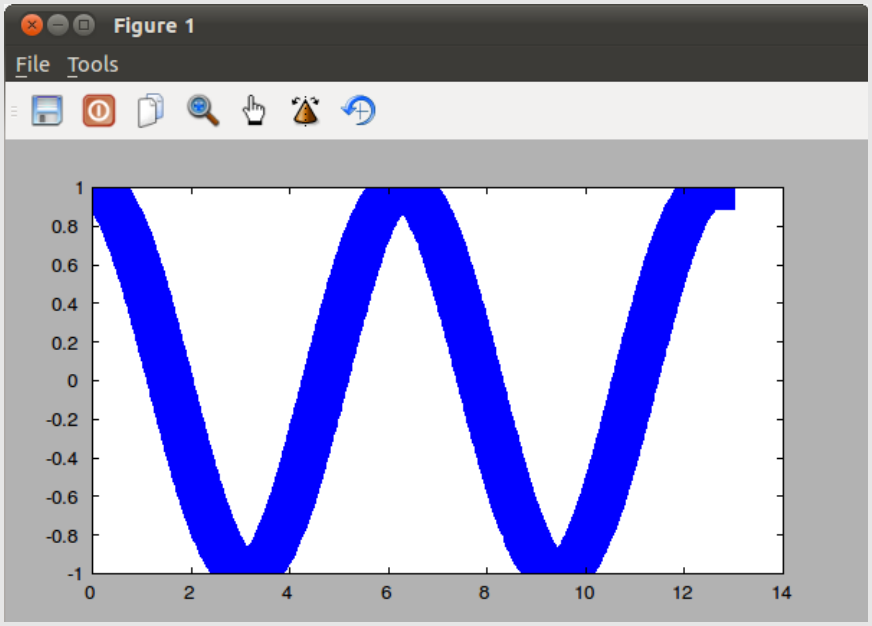


Figure 57: Plot of a sinewave with a linewidth of 32.

Finally, here's an example using the **line** command. Note that the **line** command requires a z-axis variable (in this case just an array of zeros) if you want to change the linewidth, color or any other **line** command property.

```
clear all;
close('all');
```

```
x=linspace(0,4*pi,600);
y=cos(x);
z=zeros(1,length(x));
line(x,y,z,'linewidth',10);
```

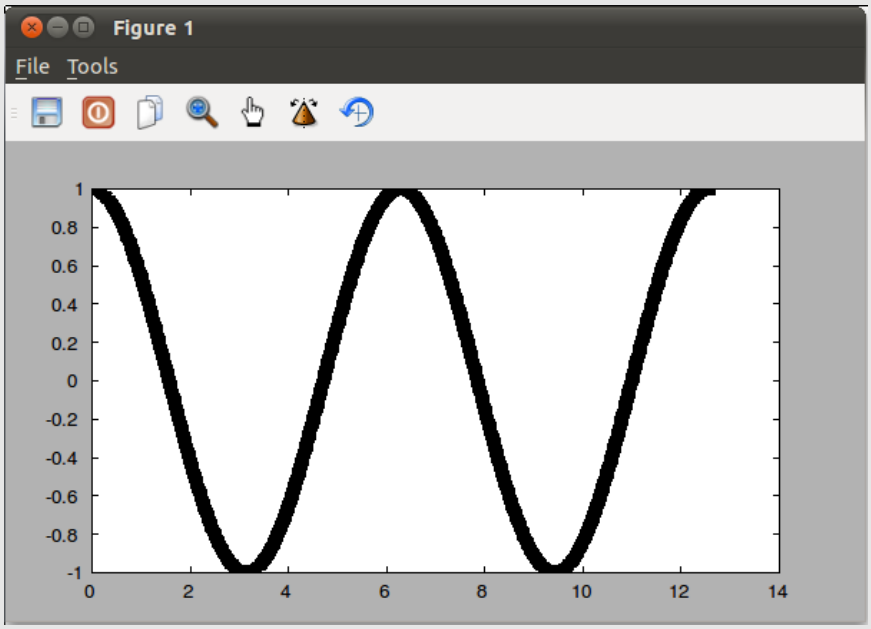


*Figure 58: Graph of a sinewave with a linewidth of 10 using the **line** command.*

This next example demonstrates that changing the linewidth in a plot command containing multiple dependent (y-axis) variables will change all of the lines.

```
x=linspace(0,6*pi,600);
y1=sin(x);
y2=sin(x)-1;
y3=sin(x)-2;
plot(x,y1,x,y2,x,y3,'linewidth',6)
```

The result is shown in Figure 59.

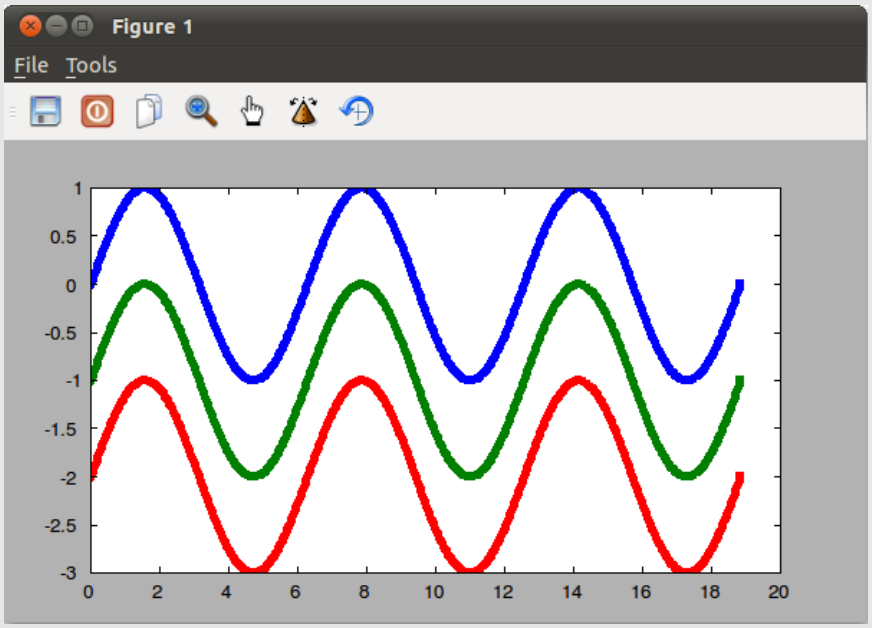


*Figure 59: Multiple traces in one graph using the **plot** command. Since it's one **plot** command, they all have to use the linewidth in that command.*

Here's an example that uses two **line** commands with different linewidths and different colors. The first creates a line that appears to "glow" while the second makes it appear as if the line is filled with yellow.

```
% Create a glowing line
clear all;
close('all');
x=linspace(0,6,600);
y=x.*sin(2*pi*x).*exp(-x);
z=zeros(1,length(x)); % This z-axis variable is created since the line command
                      % requires a z-axis in order to change the linewidth and/or
                      % the color.
line(x,y,z,'color','y','linewidth',6); % Create a thick, yellow line.
line(x,y,z,'linewidth',2); % Write a thinner, black line.  Now the black line
                           % appears to glow.
```

This results in the following graph.

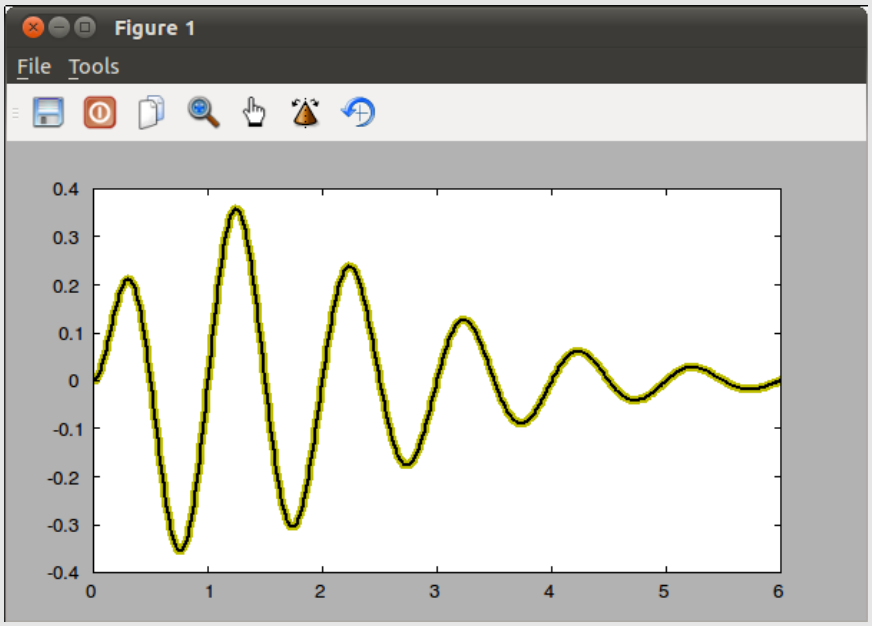


*Figure 60: A graph of a "glowing" line created using two different linewidth and colored **line** commands.*

The next script is similar; the changes are that the first line (the thick one) will be black and the second, thinner line will be yellow.

```
% Create a glowing line
clear all;
close('all');
x=linspace(0,6,600);
y=x.*sin(2*pi*x).*exp(-x);
z=zeros(1,length(x));
line(x,y,z,'linewidth',6);
line(x,y,z,'color','y','linewidth',2);
```

This results in the following graph.

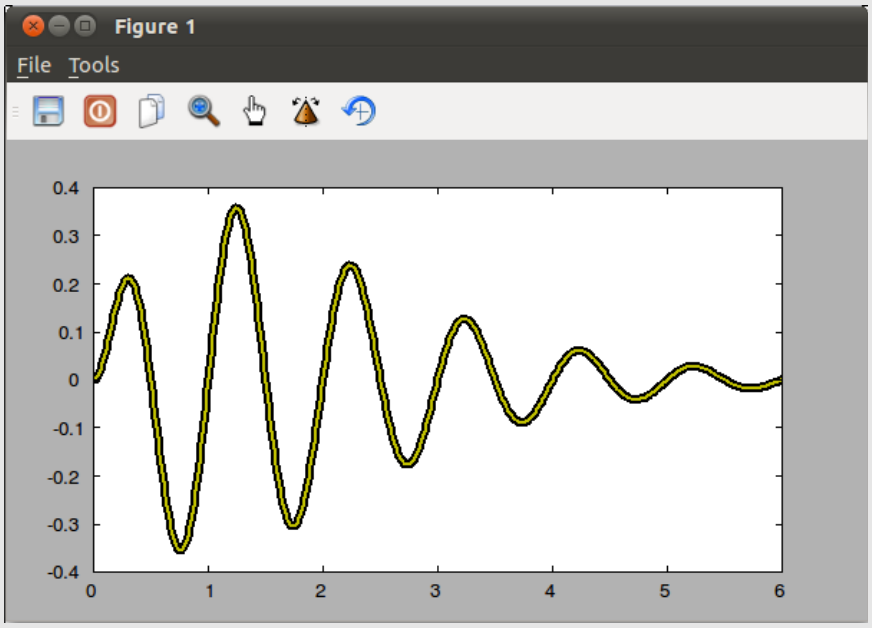


Figure 61: A graph of a "filled" line created using two line commands that are different widths and different colors. This graph was created by first plotting a thick, black line then overwriting that plot with a thinner, yellow line.

Topic 6.2.3: Graph Line Types

Freemat provides several different line types. Line types mean that these will actually create some type of line between the defined points. These are:

- `'-'` - (hyphen) Solid line style
- `':'` - (colon) Dotted line style
- `'-.'` - (hyphen followed by a period) Dot-Dash-Dot-Dash line style
- `'--'` - (two hyphens in a row) Dashed line style

Example - Graphs using Different Line Types

The first example uses the **line** command. Using the following commands, or as a script, you get the graph shown in Figure 62. This graph is a green, dotted line. The green color comes from the letter "g" after the color property; the dotted line comes from the colon after the linestyle property. Note that we used a vector using the zeros command in order to create a z-axis variable. In order to use the different properties of the line command, you must have a z-axis variable.

```
close('all')
t=linspace(0,8*pi,256);
y=sin(t)-(1/3)*sin(3*t)+(1/5)*sin(5*t);
line(t,y,zeros(1,length(t)),'color','g','linestyle',':');
```

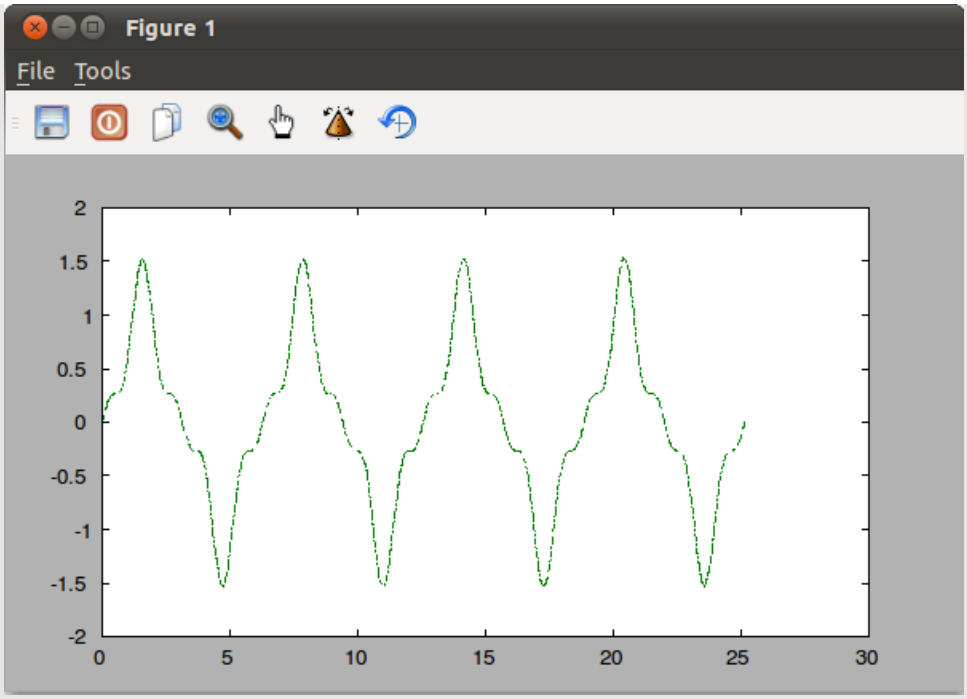


Figure 62: Plot using a dotted line

Or change the command to create a solid, black line, as shown in Figure 63.

```
close('all')
t=linspace(0,8*pi,256);
y=sin(t)-(1/3)*sin(3*t)+(1/5)*sin(5*t);
line(t,y,zeros(1,length(t)),'color','k','linestyle','-');
```

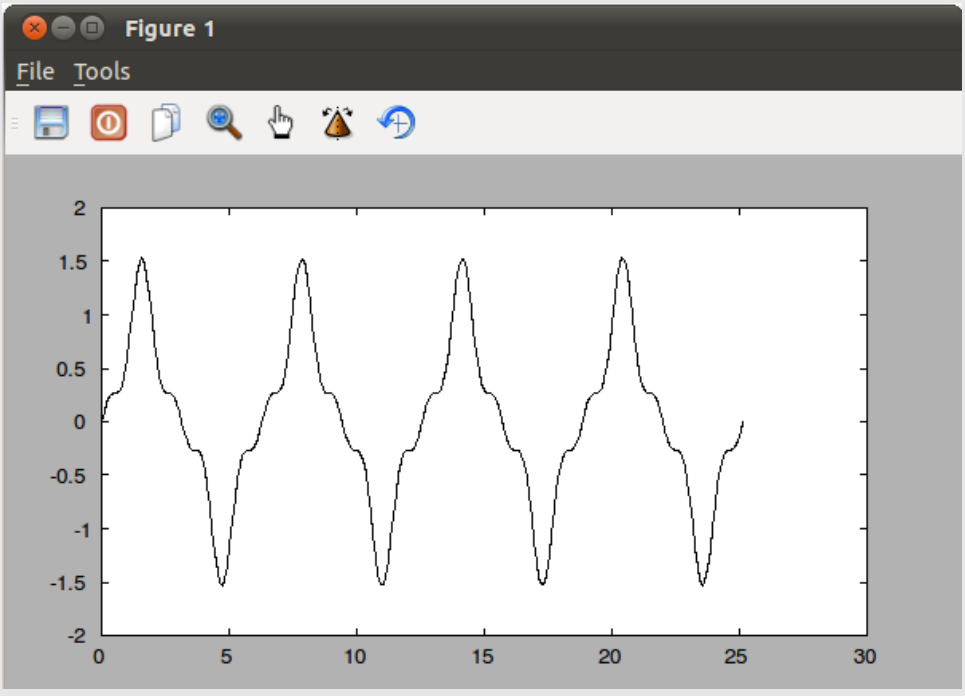


Figure 63: Plot using a black, solid line

Topic 6.2.4: Point Markers

Freemat allows you to show the actual point as a marker. The markers provided are:

- `'.'` - (period) Dot symbol
- `'o'` - (small letter "o") Circle symbol
- `'x'` - (small letter "x") Times symbol
- `'+'` - (self-explanatory) Plus symbol
- `'*'` - (self-explanatory) Asterisk symbol
- `'s'` - (small letter "s") Square symbol
- `'d'` - (small letter "d") Diamond symbol
- `'v'` - (small letter "v") Downward-pointing triangle symbol
- `'^'` - (self-explanatory) Upward-pointing triangle symbol
- `'<'` - (self-explanatory) Left-pointing triangle symbol
- `'>'` - (self-explanatory) Right-pointing triangle symbol

The marker symbol can either take the place of the line type, or it can be used simulataneously.

Example - Plots using Point Markers

Here's a short script that creates a plot. The plot is similar to Figure 60 and Figure 61, but with point markers (squares, in this case) rather than a continuous line.

```
clear all;
close('all');
x=linspace(0,6,57);
y=x.*sin(2*pi*x).*exp(-x);
plot(x,y,'ks');
```

The result is below.

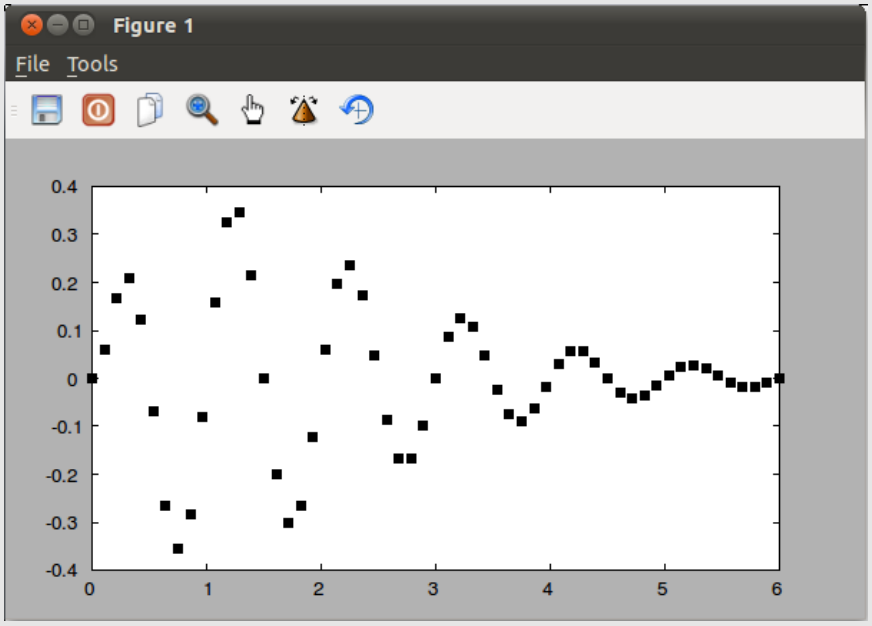


Figure 64: Graph of a curve using square point markers.

Now we'll make one adjustment to the plot command to make it a line and square point markers.

```
clear all;
close('all');
x=linspace(0,6,57);
y=x.*sin(2*pi*x).*exp(-x);
plot(x,y,'ks');
```

The result is shown below.

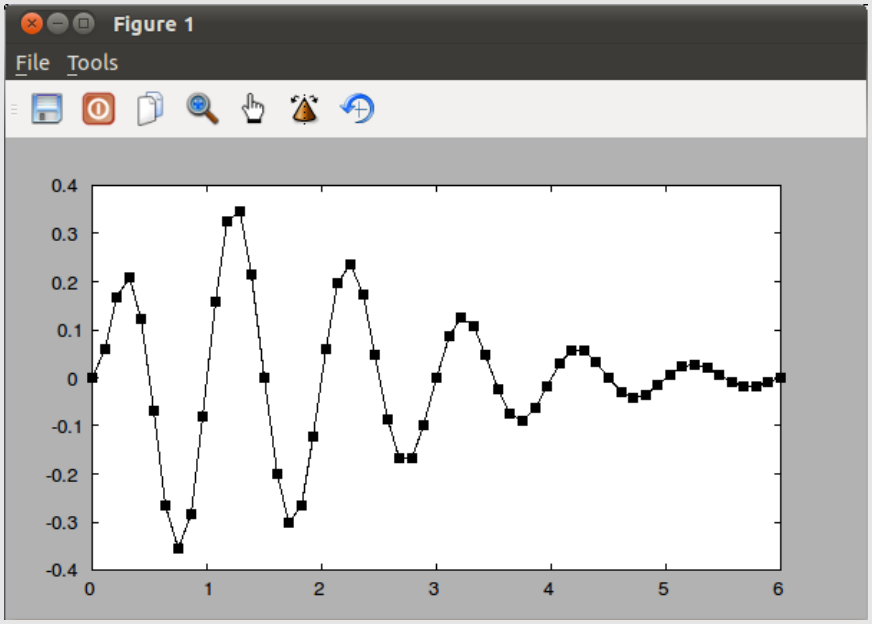


Figure 65: Graph of a curve drawn with both point square markers and a continuous line.

We can use other forms of point markers, as desired. For example, instead of squares, we can use circles (which is the lowercase letter "o" as opposed to the letter "s" for squares). We'll also use the default color.

```
clear all;
close('all');
x=linspace(0,6,57);
y=x.*sin(2*pi*x).*exp(-x);
plot(x,y,'o-');
```

The result is shown below.

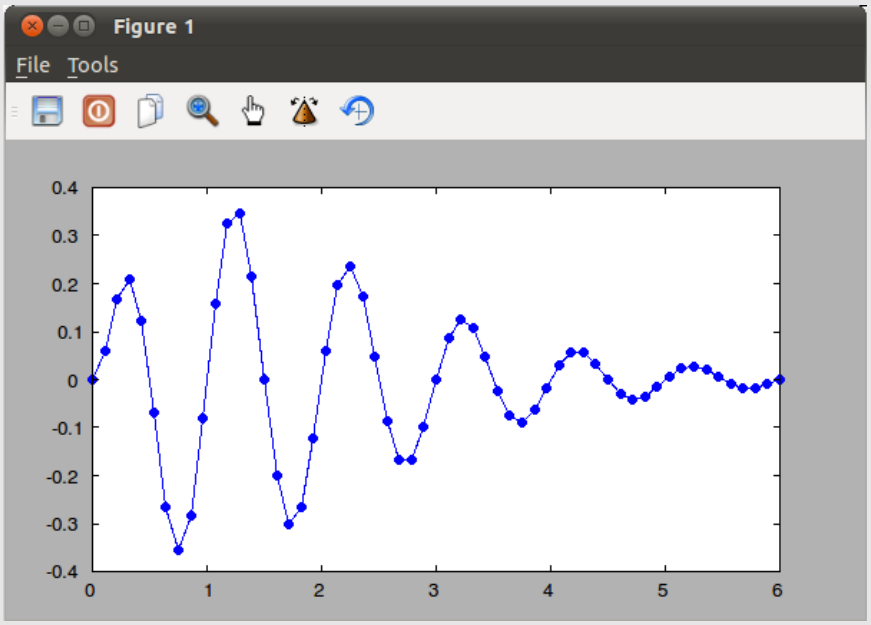


Figure 66: Graph of a curve using small circle point markers as well as a continuous line.

We'll next demonstrate using the **line** command as opposed to the **plot** command for creating point markers.

```
clear all;
close('all');
x=linspace(0,6,57);
y=x.*sin(2*pi*x).*exp(-x);
z=zeros(1,length(x));
line(x,y,z,'marker','o');
```

The result is shown below.

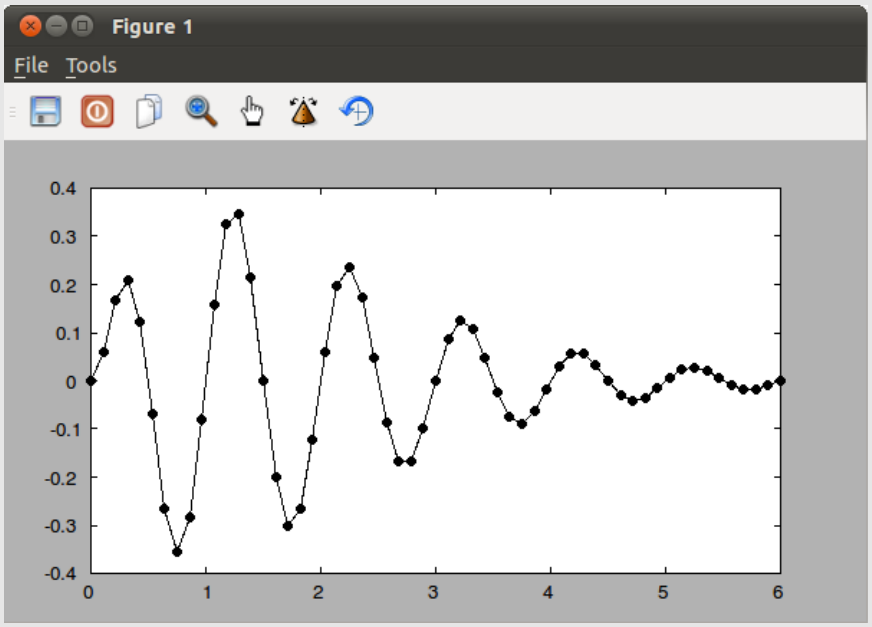


Figure 67: Graph of a curve using both a continuous line as well as small circle point markers. This was created using the line command.

Note that the **line** command defaults to drawing a continuous line unless otherwise specified. Therefore, when using the **line** command, if you don't want the line to be drawn, use the linestyle property to specify "none" as the line. Here's an example.

```
clear all;
close('all');
x=linspace(0,6,57);
y=x.*sin(2*pi*x).*exp(-x);
z=zeros(1,length(x));
line(x,y,z,'linestyle','none','marker','o');
```

The result is shown below.

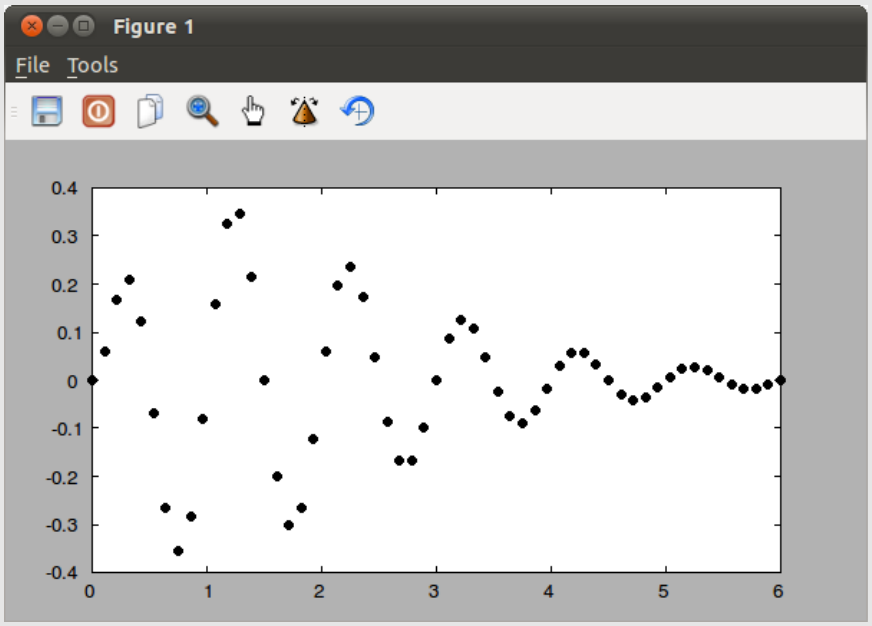


Figure 68: Graph of a curve using small circle point markers. This curve was created using the line command, and required that the linestyle property be set to "none" to keep the continuous line from being drawn.

Here's an example of a scatter plot, meaning an XY plot of points. This uses the **line** command.

```
clear all;
close('all');
x=randn(1,2000);
y=randn(1,length(x));
z=zeros(1,length(x));
line(x,y,z,'linestyle','none','marker','.');
```

The result is shown below.

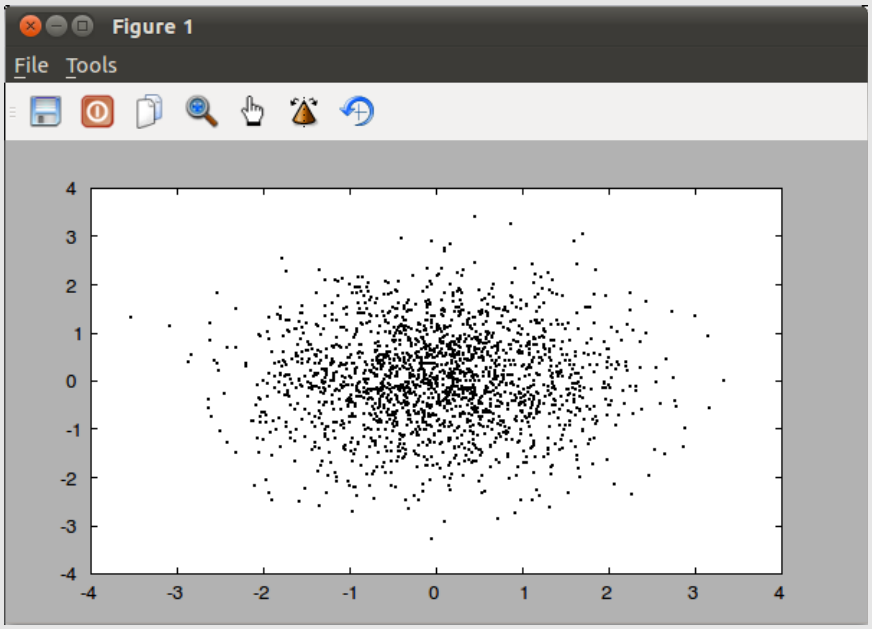


*Figure 69: Scatter plot created with the **line** command. The markers are periods (.).*

Topic 6.3: Improving the Presentation of Graphs

There are several ways you can improve the look and readability of your graph. These include adjusting the limits of the X and Y axes, adding labels (title, x-axis, y-axis), adding a legend and resizing the graph. An important point to make about several of these commands is that they must be invoked after the graph has been created (either with the **plot** or the **line** commands).

Topic 6.3.1: Setting Horizontal and Vertical Limits

You may have noticed that, in several of the plots, the graph does not completely fill the plot area. That's because Freemat sets limits that allow for nice, neat divisions. You can manually set the limits, however. This uses the **xlim** and **ylim** functions (*FD*, p. 508 & 510, respectively).
To start, after you've plotted your graph, if you want to see the limits of your data (on both the x-axis and y-axis), you can use the **get** command with the datalimits property. This gives you a short
These functions are separate from the **plot** and **line** commands, and the **xlim** and **ylim** commands must be invoked *after* the graph is created.
To set the horizontal limits, the syntax is **xlim([lo,hi])**, where **lo** and **hi** are the beginning (left-most) and the end (right-most) limits, respectively, for the graph.
To set the vertical limits, the syntax is **ylim([lo,hi])**, where **lo** and **hi** are the lowest (bottom) and highest (top) limits, respectively, for the graph.

Example - Setting the Horizontal and Vertical Limits of a Plot

We'll start with a graph that has multiple relative maxima and minima.

```
clear all;
```

```
close('all');
x=linspace(-2,3,600);
y=x.*cos(x.^2);
plot(x,y);
```

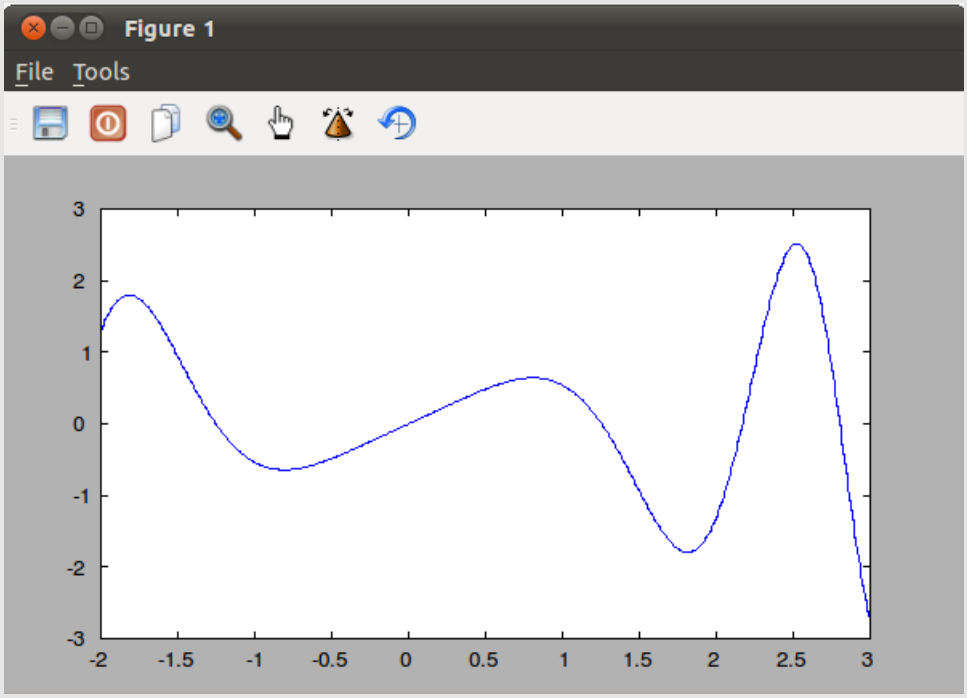


Figure 70: Graph of a curve that has multiple maxima and minima.

First, we'll look at the current limits of the data.

```
--> get(gca,'datalimits')
ans =
   -2.0000    3.0000   -2.7334    2.5143   -0.5000    0.5000
```

This is a six-element vector. The first two elements are the minimum and maximum of the x-axis, the second and third elements are the minimum and maximum of the y-axis, and the last two are the minimum and maximum of the z-axis. Note that the z-axis data is not used in this context, even though it's provided. While we won't get into it here, you can make any 2D graph into a 3D graph using the **view** command. We'll discuss that in future lessons.

We can adjust the limits so that we "zoom in" on the part between 0 - 2.

```
xlim([0,2])
```

The result is shown in Figure 71. Note that changing the limits of the graph does not change the limits of the data as provided by the datalimits property.

```
--> get(gca,'datalimits')
ans =
   -2.0000    3.0000   -2.7334    2.5143   -0.5000    0.5000
```

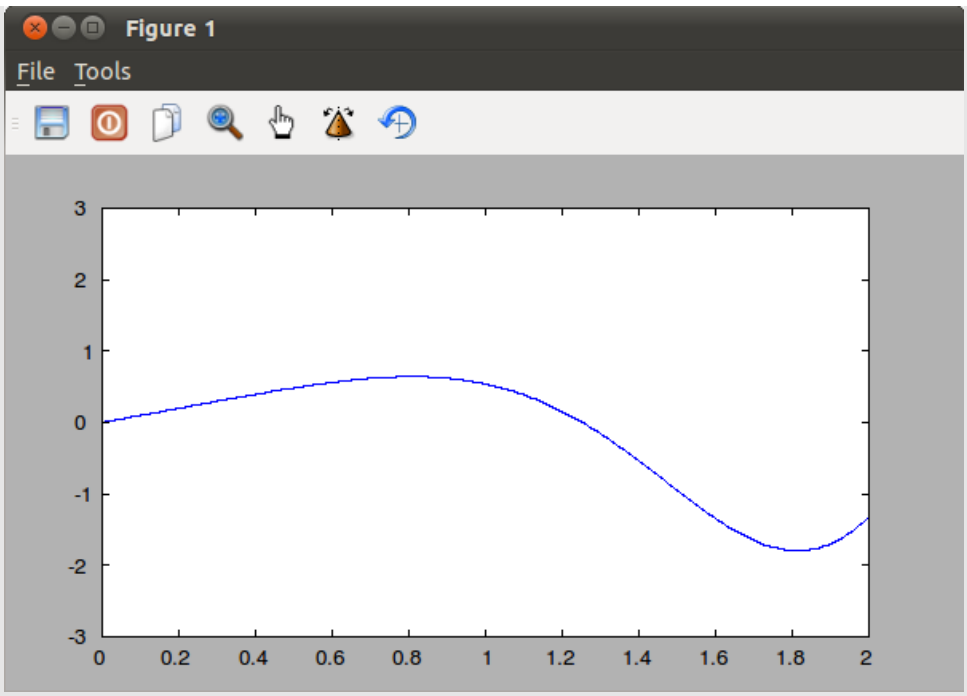


Figure 71: Plot with horizontal limits manually set

Note that we have a large area between the maximum point on the graph and the top of the plot area. We can adjust the vertical limits so that the graph fills more of the plot area.

```
ylim([-2,1])
```

The result is shown in Figure 72.

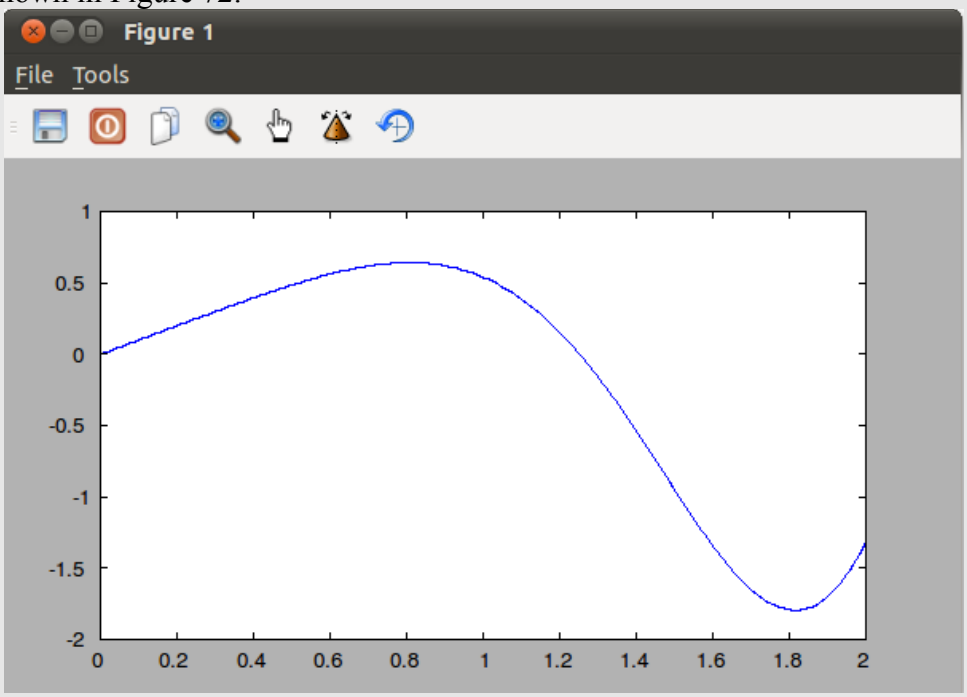


Figure 72: Plot after setting both the horizontal and vertical limits

Topic 6.3.2: Resizing a Plot

You can resize the entire plot window either by using the **sizefig** function (FD, p. 487) or by using the mouse to grab the sides or corners and clicking-&-dragging to resize the window. Within the window itself, you can adjust the relative position of the plot area within the window.
To understand the figure size, look at .

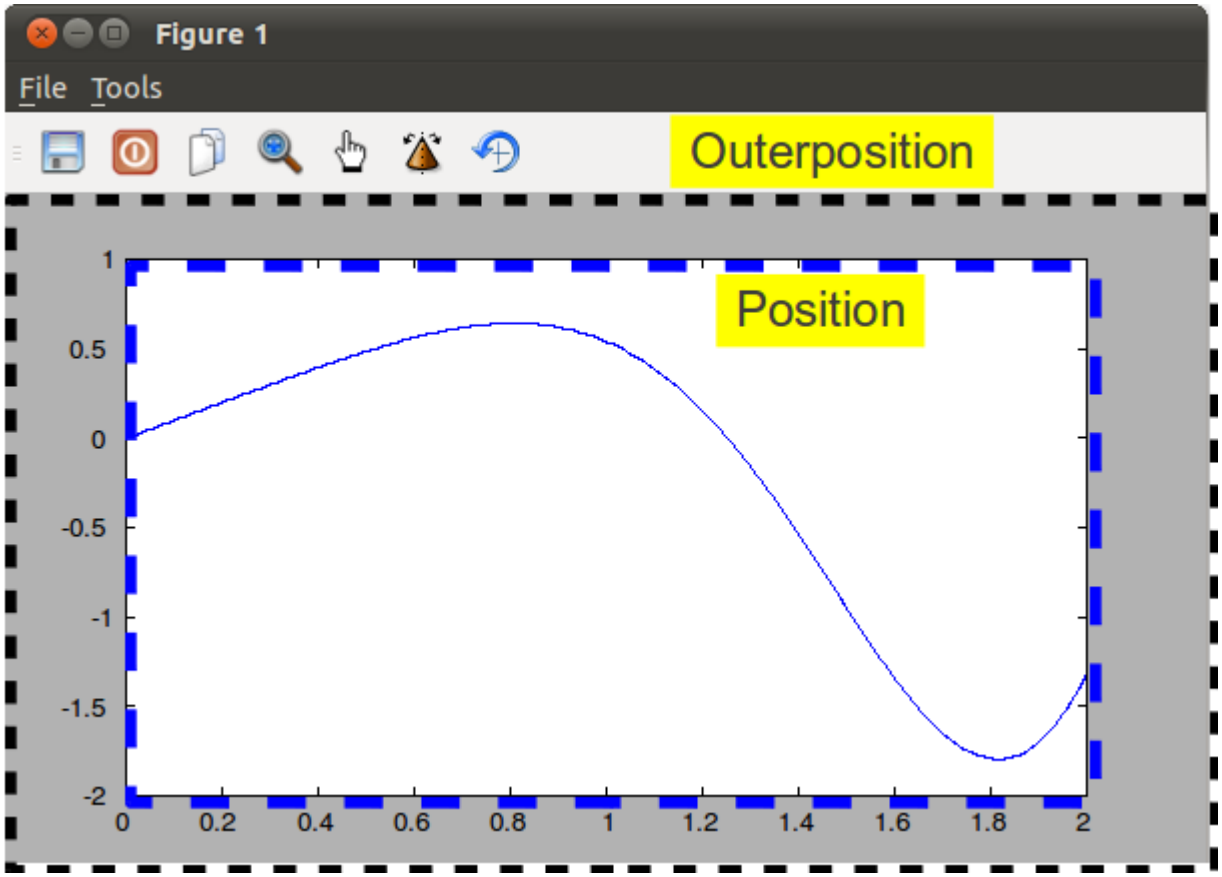


Figure 73: To understand the figure size, you need to understand how a figure window is put together. The window bar (at the very top) and the various tool menus (both text and icon) are not considered in the figure size. The size of the "figure" consists of the area of the box labeled as "outerposition". The part labeled as "position" is the plot area, meaning the area where the graph is actually plotted.

To see the size of the current figure:

```
get(gcf,'figsize')
```

This is one of the figure properties (FD, p. 461). Note that when you want to see a *figure* property, you use the gcf (get current figure) handle with the **get** command. When you want to see an *axis* property (FD, p. 441), you use the gca (get current axis) handle with the **get** command.
To change the size of the figure, you use the **sizefig** command, which uses the following syntax:

```
sizefig(x,y)
```

where: x = the size of the total window horizontally
y = the size of the gray area vertically. The vertical size of the window will be this number plus 56 pixels.

This function is also invoked separately from the **plot** function and must be invoked after the **plot** function.
The size of the actual plot area, meaning the area of the "position" box as shown in Figure 73, will be less than that stated in **sizefig**. The sizefig command states the size of the box labeled as "outerposition" in Figure 73. You can see the relative size of the plot area using the following command:

```
get(gca,'position')
```

You can adjust the relative size of the plot area within the window using the following command:

```
set(gca,'position',[x_start y_start x_size y_size])
```

Example - Viewing the Size of and Resizing a Plot

In this example, we'll look at the current size as well as resizing a plot.

```
clear all;
close('all');
x=linspace(-2,3,600);
y=x.*cos(x.^2);
plot(x,y);
```

These commands will create the plot shown in Figure 74.

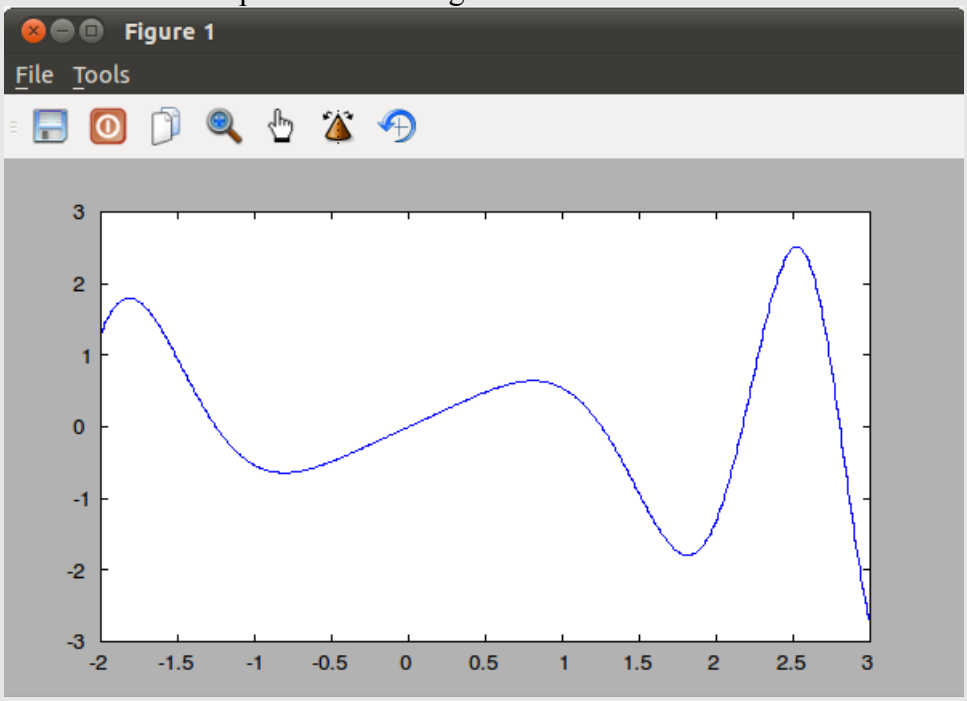


Figure 74: Plot shown in default size

To view the current size, we use the **get** command with the figsize property. Since this is a figure property, we need to use the figure number (as provided by the **gcf** command), not the axis number (as provided by the **gca** command).

```
--> get(gcf,'figsize')
ans =
 600 335
```

This means that the size of the "outerposition" box is 600 pixels wide by 335 pixels high.
We can also view the relative size of the plot area using the **get** command with the **position** property.

This property is one of the axis properties, so it has to be used with the axis handle (gca). Yes, the difference between an "axis" and a "figure" can be confusing.

```
--> get(gca,'position')
ans =
    0.1000    0.1000    0.8000    0.8000
```

The position property returns a four-element vector. The first two elements are the relative position of the lower, lefthand corner of the position box. The first element is the x-axis position and the second element is the y-axis position. The third and fourth elements are the relative width and height of the "position" box, respectively. In this example, the "outerposition" box is 600 pixels wide and 335 pixels high. The relative size of the "position" box is 0.8 x 0.8, or 600 x 0.8 = 480 pixels wide by 335 x 0.8 = 268 pixels high. We can combine the figsize and position properties to create a set of commands that will give us the absolute width and height of the current figure, as follows:

```
--> get(gca,'position')(3)*get(gcf,'figsize')(1)
ans =
 480
--> get(gca,'position')(4)*get(gcf,'figsize')(2)
ans =
 268
```

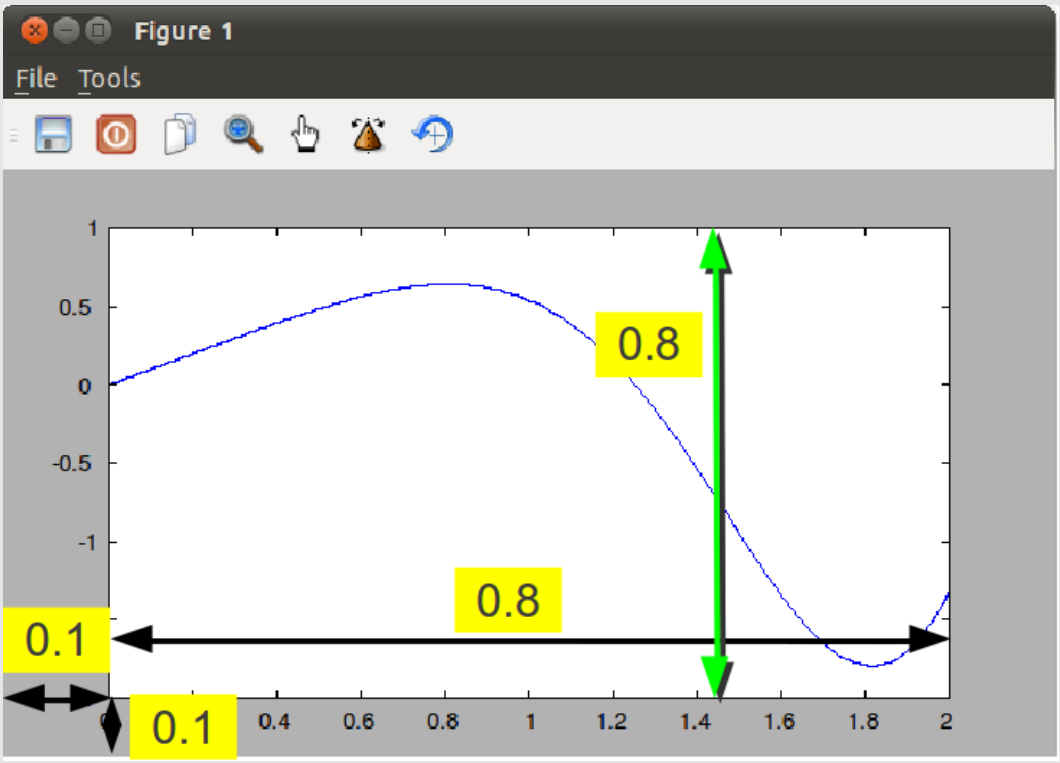


Figure 75: The position property returns the relative position of the lower, lefthand corner of the plot area, as well as the relative size (as stated by the figsize property) of the plot area.

Now we'll actually change the size of the figure window. We'll change it to a smaller size, though so far pretty much any size is possible.

```
sizefig(300,200)
```

The result is shown in Figure 76.

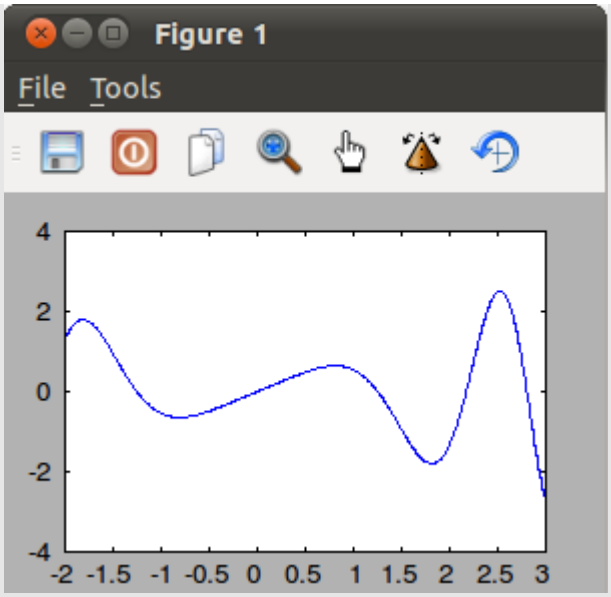


Figure 76: Resized plot at a smaller size

Topic 6.3.3: Adding Plot Labels

Now that you've created your plot, you can add labels. These include a title, x-axis and y-axis labels, and text labels within the graph itself.

Topic 6.3.3.1: Adding a Plot Title

You can add a title just above the graph using the **title** command (FD, p. 498). The general syntax is:

```
title('Put Your Title Here')
```

The **title** function comes with several properties, including position, fontsize, background color (around the title, not the whole graph), and other properties. The title properties include all properties that can be assigned to textproperties (FD, p. 497).

Example - Adding a Plot Title

In this first example, we'll add a plot title at the default size, then use a larger font (which requires re-doing the title) at a 20-point font. A basic title uses an 8-point font (the font is 8 pixels high).

```
clear all;
close('all');
x=linspace(-2,3,600);
y=x.*cos(x.^2);
plot(x,y);
title('Plot of Curves')
```

Here's the result.

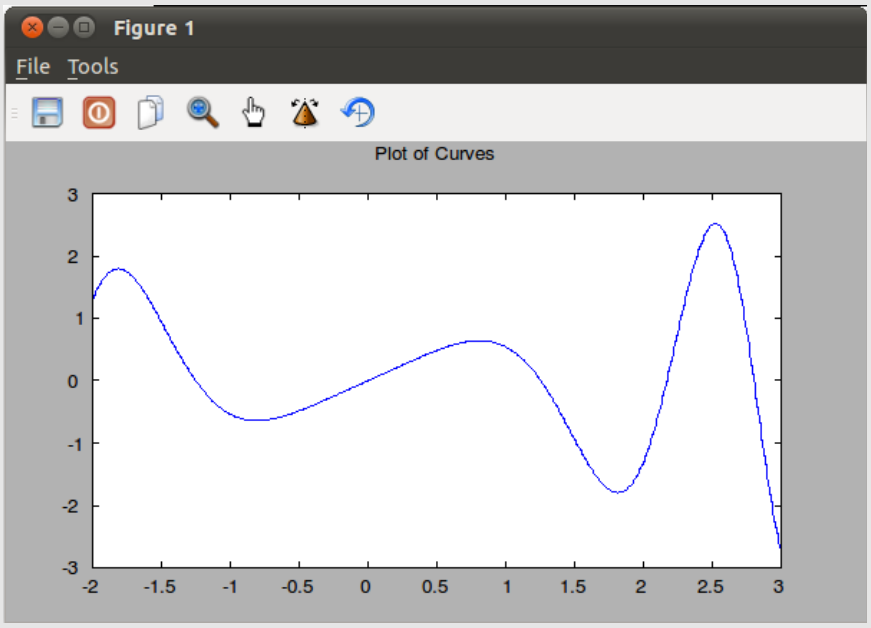


Figure 77: Plot with basic title using the default 8-point font.

Now we'll re-do the title with a 20-point font.

```
--> title('Plot of Curves','fontsize',20);
Warning: Newly defined variable nargin shadows a function of the same name.  Use
clear nargin to recover access to the function
```

Here's the result.

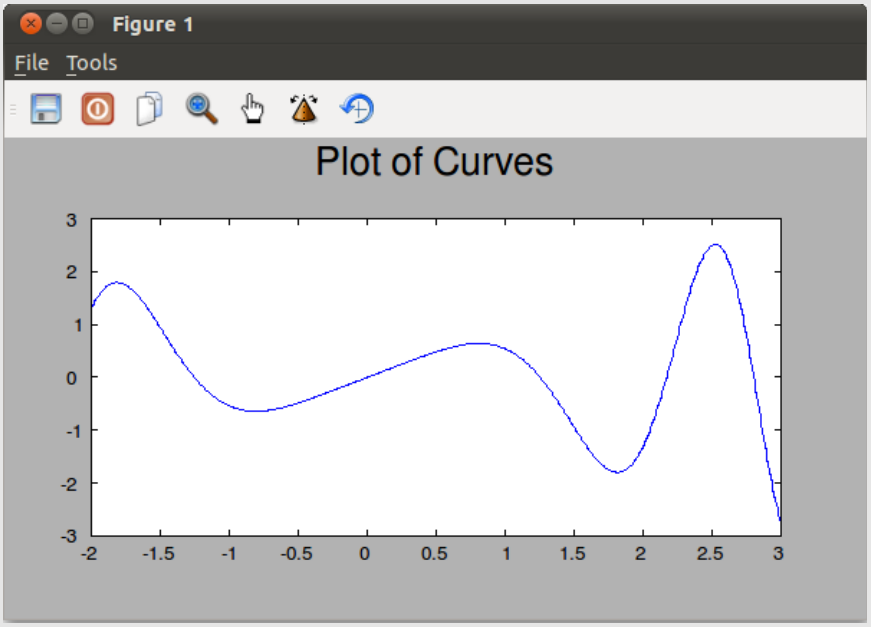


Figure 78: Plot with a title using a 20-point font.

Next, we'll add some more properties to the title, including a specific position, text alignment, and a different background color.

The title can be both positioned within the "outerposition" box as well as according to the alignment of

the text itself. The title string can be aligned along nine different points, as shown in Figure 79. By default, the alignment point of the title will be the top, center point. You can set the alignment point using two different text properties, horizontalalignment and verticalalignment.
The position property will define where the alignment point of the title string is placed within the outerposition box. The default position is [0.5 1], which is centered at the top of the display window.Clear as mud?

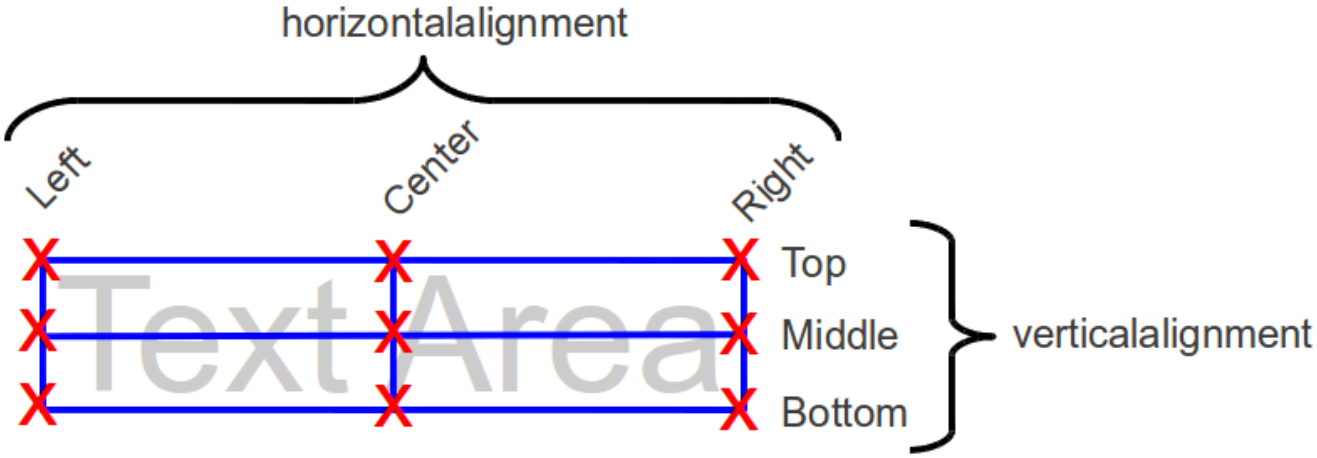


Figure 79: Text alignment points, as shown by the red X's. There are three along both the horizontal and vertical axes, making a total of nine points possible.

Example - Positioning a Plot Title & Setting the Alignment

We'll continue with the previous example, except we'll re-position and re-align the title text.

```
title('Plot of Curves','fontsize',20,'position',[0.25
0.9],'verticalalignment','middle')
```

The result is shown in Figure 80.

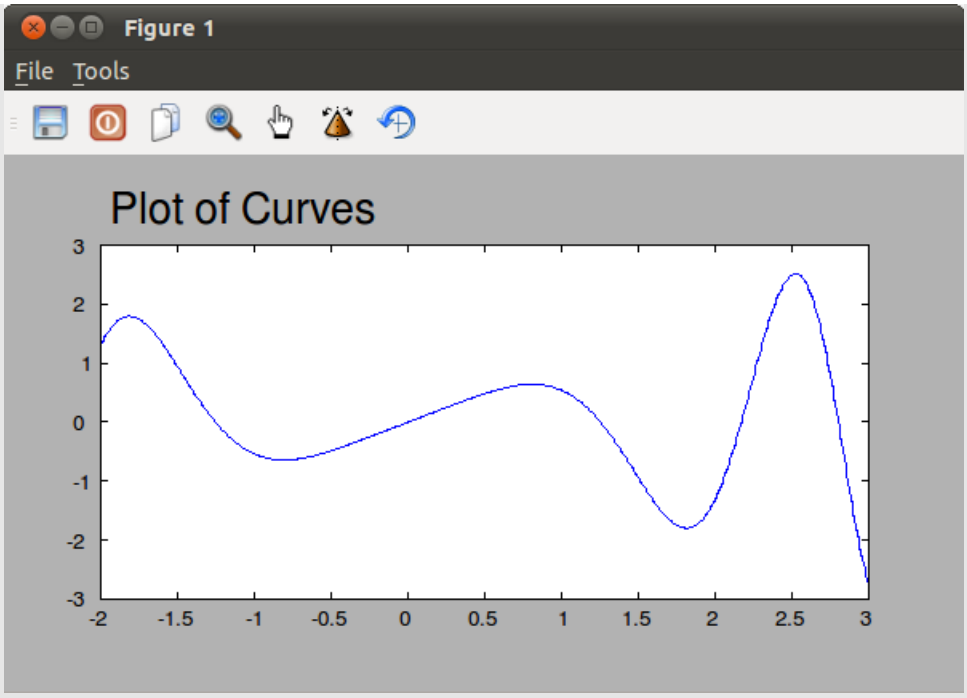


Figure 80: Positioning and aligning a plot title. In this case, the x-position has been set to 0.25 from the left and 0.9 from the bottom of the outerposition box. The alignment of the text has been set to a vertical alignment of "middle", which means the alignment point is the middle of the text vertically. This is a more subtle effect.

Topic 6.3.3.2 Setting X-Axis (Horizontal) & Y-Axis (Vertical) Labels

Next we'll add labels to the horizontal and vertical axes. This is done using the **xlabel** command (FD, p. 507) and the **ylabel** command (FD, p. 509), respectively. While some of the text properties do apply to the xlabel and ylabel, some do not. For example, the fontsize property can be applied to these two labels, but the alignment and position properties cannot be applied.

Caution - The Position of the Text When Using the Xlabel Command

As of this writing, there is a bug in Freemat 4.0 that, depending on the size of the figure window, causes the xlabel to be positioned such that the bottom-half is outside of the outerposition box. This is a problem for the smaller size figure windows, including the default size (600 x 335 pixels for Linux-based Freemat; 600 x 344 for Windows-based Freemat). A work-around for this is to use the position property with the **get** command to re-size the inner "position" box such that the text becomes fully visible. There is a special function, called labelSet, listed in Appendix A: Special Function to Resize Plots with Titles and Labels that will use this command and some special calculations to resize your plot such that all of the text around (title, xlabel, ylabel) are properly visible.

The syntax for the two commands are:

```
xlabel('Put the horizontal axis label here')
```

```
ylabel('Put the vertical axis label here.')
```

Example - Putting Horizontal & Vertical Axis Labels on a Plot

We'll again use the previous example (minus the title) and add horizontal and vertical labels.

```
clear all;
close('all');
x=linspace(-2,3,600);
y=x.*cos(x.^2);
plot(x,y);
```

This provides the basic plot (without the title). Next, we'll add an x-axis label.

```
xlabel('Here Is A Label for the X-Axis')
```

The result is shown in Figure 81.

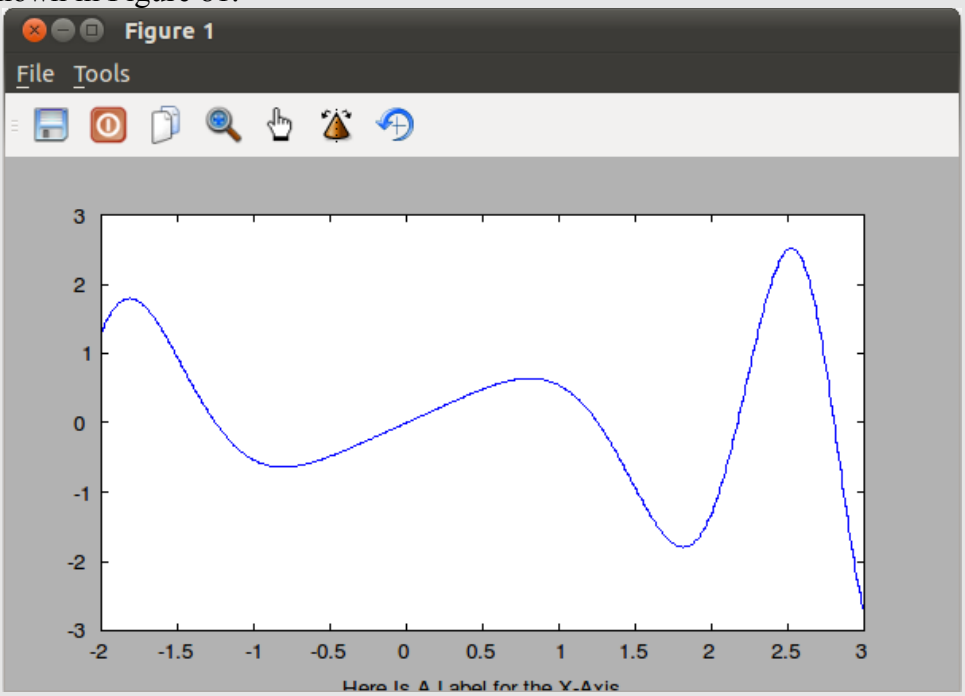


Figure 81: Adding a horizontal axis label. In this case, we didn't use any of the properties possible. In this case, the label is cut-off and only the top half shows. This is a bug in Freemat (which will be fixed in a future release).

In this example, a basic **xlabel** command creates a label on the horizontal axis that is half-way cut-off. This is a bug in Freemat that will be fixed in a future release. In the meantime, I've created a function called labelSet (listed in Appendix A: Special Function to Resize Plots with Titles and Labels on page 215) that you can use to fix this problem. This function looks for the presence of a title, xlabel or ylabel, looks at their sizes and the size of the overall graph, and then calculates the appropriate size of the plot area and position.

The output of this function is that the graph will be resized so that the horizontal label is visible.

```
labelSet
```

The result is this:

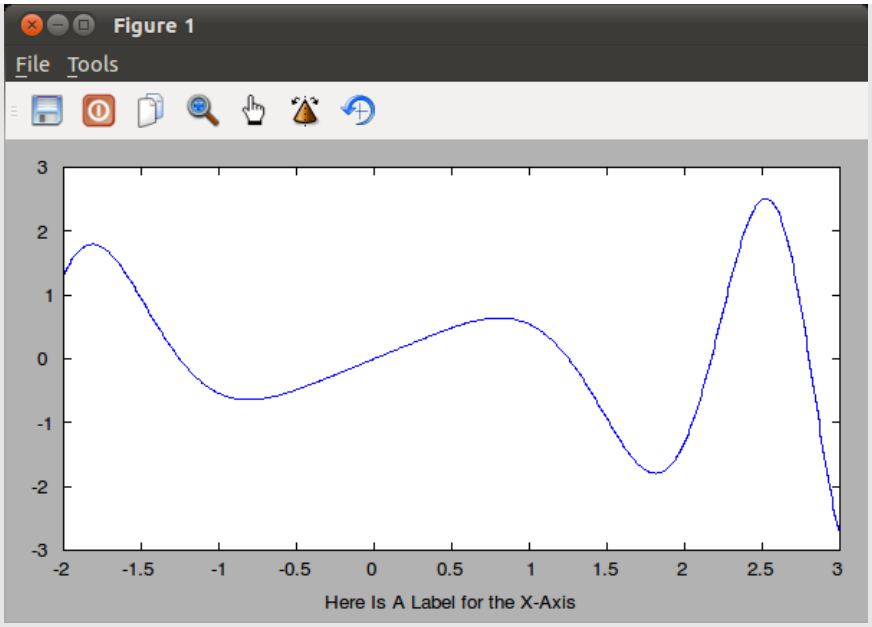


Figure 82: Here's the same graph, but after having run the labelSet function described above. This function will resize the graph so that the horizontal axis label is visible.

Let's re-run the **xlabel** command but with a larger font. We'll also use the labelSet function we created to position the label.

```
xlabel('Here Is A Label for the X-Axis','fontsize',16)
labelSet
```

The result is this.

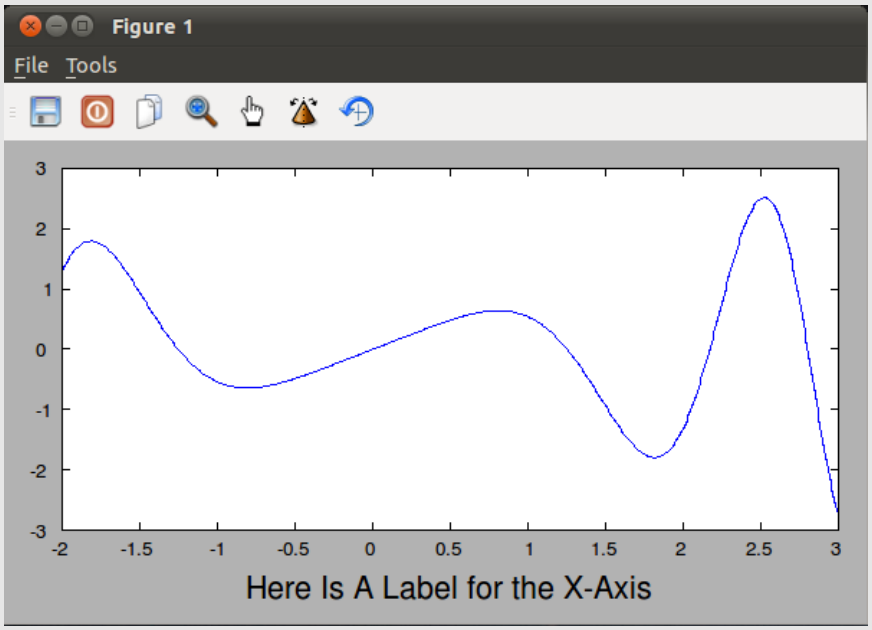


Figure 83: Another x-axis label, but with a 16 point font. Note that the labelSet function (listed in the appendix) was used to reposition the label.

Now we'll add a vertical axis (y-axis) label, also with a 16-point font, again using our custom-made

labelSet function.

```
ylabel('Here is a Label for the Y-Axis','fontsize',16)
labelSet
```

The result is shown in Figure 84.

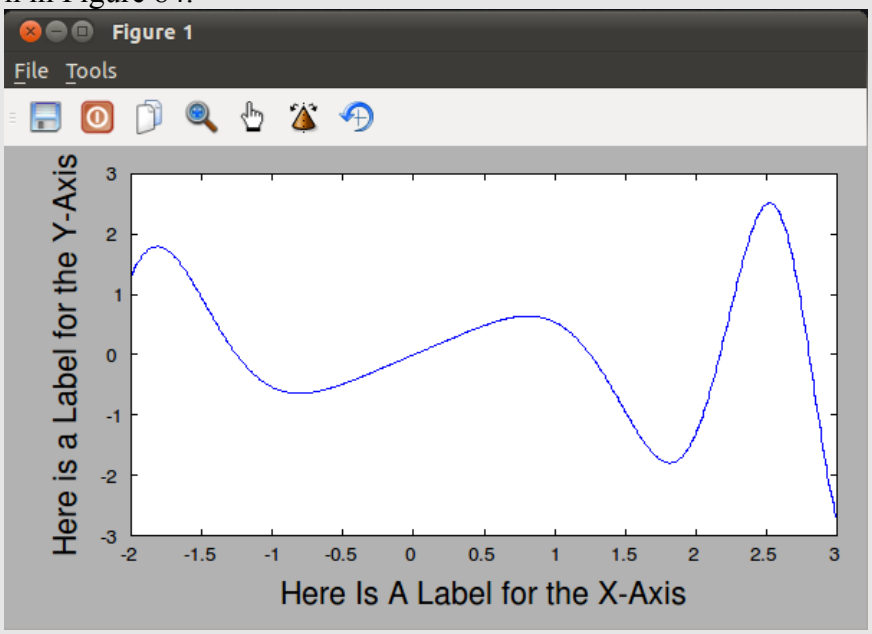


Figure 84: This is the result of adding a 16-point font y-axis (vertical) label.

Topic 6.3.3.3: Adding a Legend

You can add a legend, which shows what every line of a plot with multiple graphs means. This uses the **legend** command (FD, p. 473) and requires that one string be provided for each trace within the graph. The syntax is as follows:

```
legend('String for trace 1','String for trace 2')
```

Note that you only use one legend command for each plot. Do not try to use one legend command for each trace; you'll simply wind up overwriting the previous legend command.

Caution - Adding a Legend and the Hold Command

I've noticed some issues when using the **legend** function after using the **hold on** command. Therefore, if you are using the **hold on** command, I recommend that you not add the legend until the **hold off** command has been issued.

This command has one (and only one) property that can be adjusted. That's the location property. This property can be one of eight directions. These are north, south, east and west and the other four directions between these (northeast, southeast, southwest, northwest). By default, this property is set to "northeast" (meaning the upper, righthand corner of the graph).
Here are a few examples.

Example - Adding a Legend to Plots

We'll start the first plot with just one trace.

```
clear all;
close('all');
x=linspace(-2,3,600);
y=x.*cos(x.^2);
plot(x,y);
legend('This is a basic curve')
```

This generates the plot shown in Figure 85.

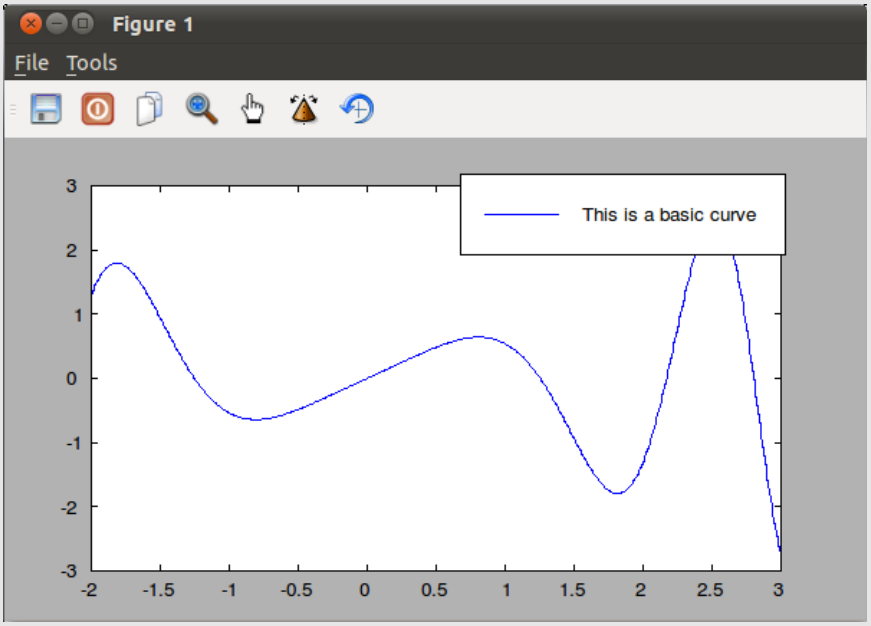


Figure 85: Basic graph with one trace and a legend.

You can adjust the location property to the bottom, lefthand corner ("southwest"), as follows:

```
legend('This is a basic curve','location','southwest')
```

This results in the following graph:

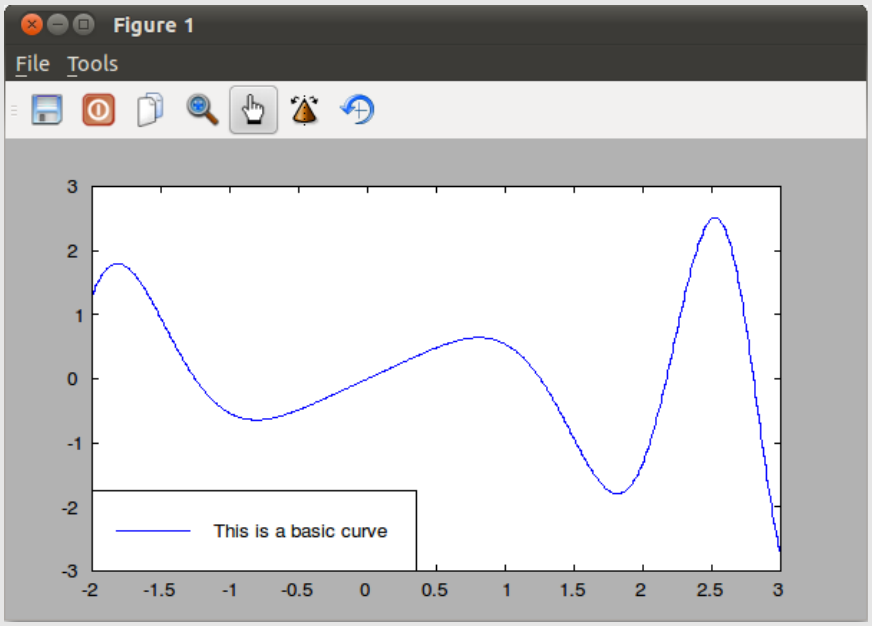


Figure 86: This is a graph with a legend moved to the lower, lefthand (southwest) corner.

We'll create another curve which will be the differential of the one shown above. Then we'll add a legend for each trace (the original and the differential). This uses the **diff** command (FD, p. 202).

```
clear all;
close('all');
x=linspace(-2,3,600);
y=x.*cos(x.^2);
plot(x,y);
dy=diff(y); % Calculate differential on y-axis.
dx=diff(x); % Calculate differential step size.
ds=dy./dx; % Calculate differential slope.
w=dx(1); % Calculate x-axis step size.
xx=linspace(-2,(3-w),length(dy)); % Calculate x-axis array.
line(xx,ds); % Plot differential curve.
legend('This is a basic curve','Differential of curve','location','southwest');
```

The result of this script is shown in Figure 87.

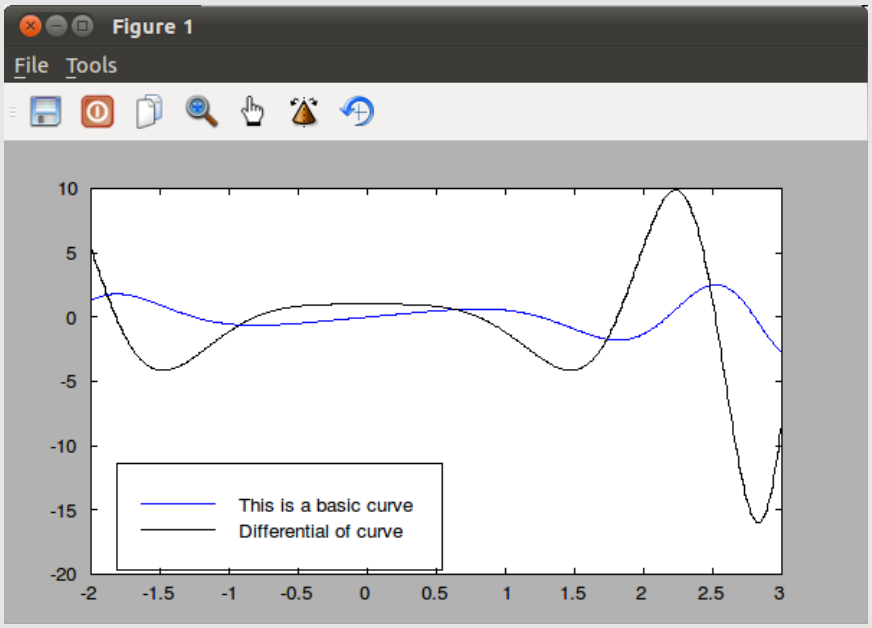


Figure 87: Plot showing two traces and a legend for each.

We can use one more example for a legend, and that's to show the difference between the curve of a hyperbolic sine function and a hyperbolic cosine function.

```
x=linspace(-5,5,600);
y1=sinh(x);
y2=cosh(x);
plot(x,y1);
line(x,y2,zeros(1,length(x)),'linewidth',2);
legend('Sinh(x)','Cosh(x)','location','southeast');
```

This results in the following image.

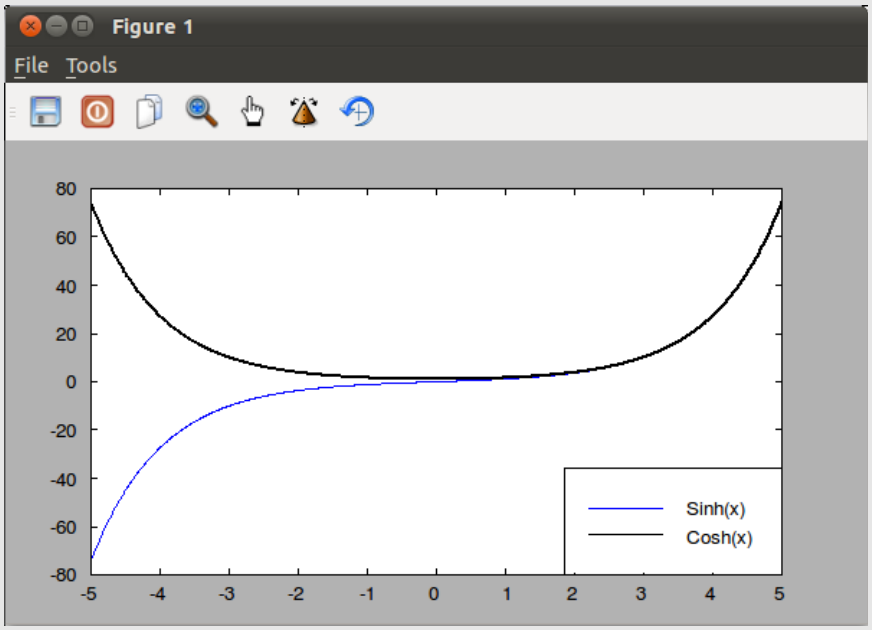


Figure 88: Comparison of the hyperbolic sine and cosine functions between -5 to 5.

Topic 6.3.3.4: Adding General Text

We can also add text directly to the graph itself using the **text** command (FD, p. 496). This command has the general syntax of:

```
text(x,y,<string>);
```

where: x = the point on the x-axis where the leftmost point of the text will reside.
y = the point on the y-axis where the middle of the text will reside.
<string> = this is the text, entered as a standard string, that will appear on the graph.

Example - Adding Text to a Plot

In this example, we'll use the plot from one of the previous examples. The difference is that instead of using the legend command to describe the different traces on the graph, we'll use text next to each trace.

```
x=linspace(-5,5,600);
y1=sinh(x);
y2=cosh(x);
plot(x,y1,'linewidth',2);
line(x,y2,zeros(1,length(x)),'linewidth',2);
text(-4,-40,'Hyperbolic sine');
text(-4,40,'Hyperbolic cosine');
```

This provides the graph shown in Figure 89.

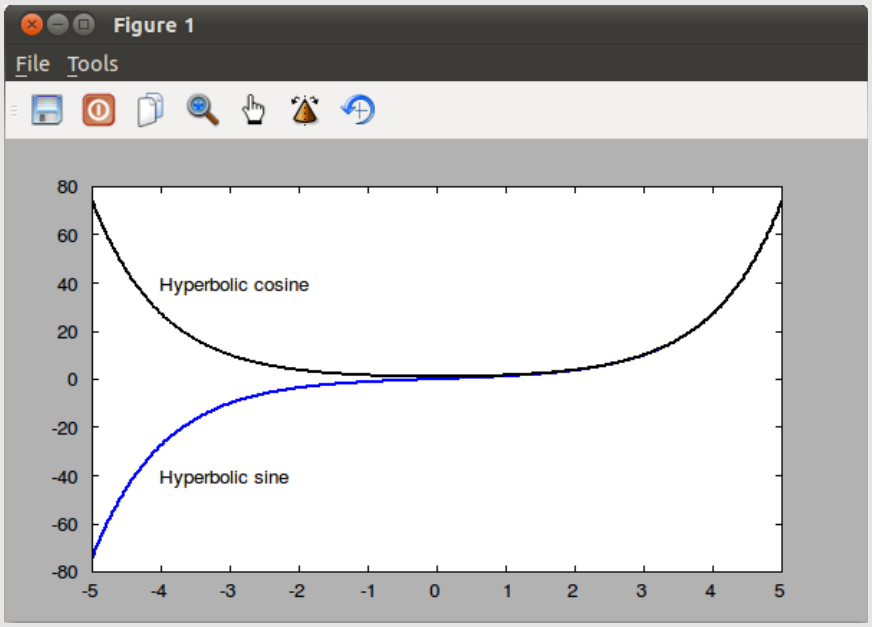


Figure 89: Using the text command, we can put text directly on the graph. In this case, we use it to describe the different traces, rather than using the legend command.

Topic 6.3.4: Adding a Grid

To add a grid to a plot is nothing more than:

```
grid on
```

To turn off the grid, using this command:

```
grid off
```

Example - Adding a Grid to a Graph

This example will look at plotting data retrieved from a text files. This will be similar to the example in Topic 4.10: Inputting Data from ASCII Text Files on page 77, except we'll add plot labels and a grid to assist in understanding the distribution of the data.

```
clear all;
close('all');
data=real(dlmread('todAccelY.txt',char(9)));
t=data(:1,);
accel=data(:,2);
plot(t,accel);
xMin=get(gca,'datalimits')(1);
xMax=get(gca,'datalimits')(2);
xlim([xMin,xMax]);
grid on;
title('Tower of Doom Y-Axis Acceleration Data');
xlabel('Time (sec)');
ylabel('Acceleration (m/s^2)');
labelSet;
```

Running this script, we get the following graph.

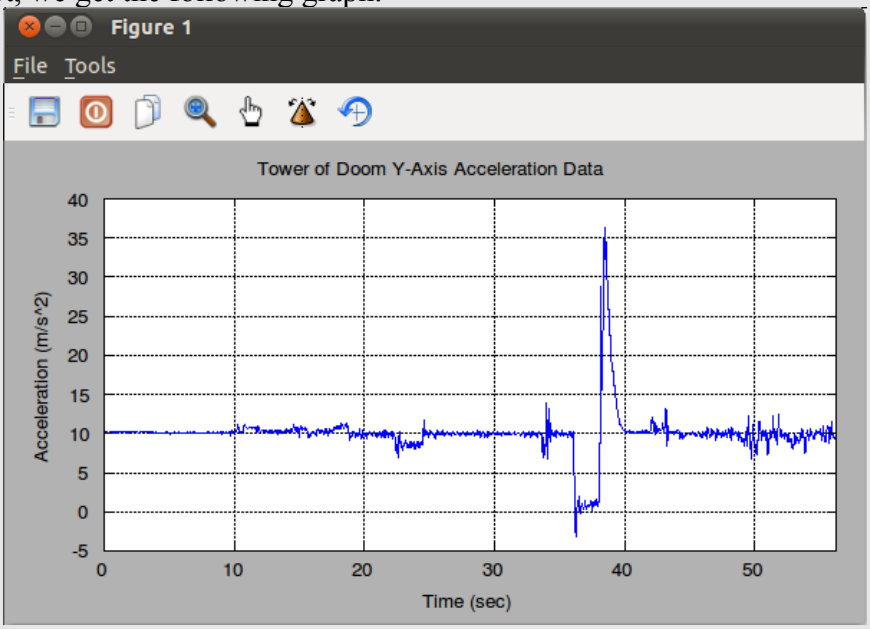


Figure 90: Graph shown with a grid. This is the y-axis acceleration data from the "Tower of Doom" ride at Six Flags theme park (Largo, MD), courtesy of the American Physical Society Public Outreach Program.

Topic 6.4: Working with Multiple Plots

When you create a plot, it has a number. Note how the plot window says "Figure X" at the top. X is its figure number. The first one, by default, is 1. Unless you change the figure number, any other plot commands will be directed towards this plot window. Thus, you can inadvertently overwrite a plot when you'd rather have two (or more) plots.

Fortunately, Freemat provides a simple way to manage multiple plots. This is with the **figure(x)** command (FD, p. 461). You use this command to make figure *x* the active plot. If figure *x* doesn't already exist, Freemat makes a blank plot window. This will remain the active plot unless and until you change it. There are three ways to change the active plot. The first is using the **figure** command. The second is if you close the plot windows. For example, if you have two plots open, say 1 and 2, and you close 1, then 2 will become the active plot. If you close both, plot 1 will become the active plot again. Essentially, you've started over. The third way is to use your mouse and click on the plot you wish to make active.

Unfortunately, there's no way to determine the total number of figures you have open at any one time; however, you can use the **gcf** (get current figure) command (FD, p. 462) to determine the current active figure.

Example - Creating Multiple Plots

```
t=1:128;
x=sin(2*pi*t/32);
plot(x) % This creates the first plot, Figure 1
y=cos(2*pi*t/16);
figure(2) % This creates the second plot, Figure 2
plot(y)
```

Topic 6.5: Saving Your Plots

Frankly, in my book, being able to save the plots as images is the single greatest thing about Freemat. After processing some data, making an image (while setting the colors *just right*), re-sizing it, setting the vertical and horizontal limits, and adding descriptive labels, the ability to quickly, painlessly, and easily save them as image files is the single best thing about Freemat.

To save the active Freemat plot, use the **print** function (FD, p. 483). It has the following syntax:

```
print('filename.png')
```

This saves it to the file *filename* as a PNG (Portable Network Graphics) image. Other allowable extensions are *.jpg*, *.pdf*, and *.svg*. Frankly, in my opinion, .png tends to look the best.

Example - Saving an Image

```
t=linspace(0,2,1000);
f=3; % Frequency in Hz.
y=cos(2*pi*f*t);
plot(t,y)
print('testfile.png')
```

The resulting image is shown in Figure 91. Note that, since the full directory was not specified, the file was saved into the working directory.

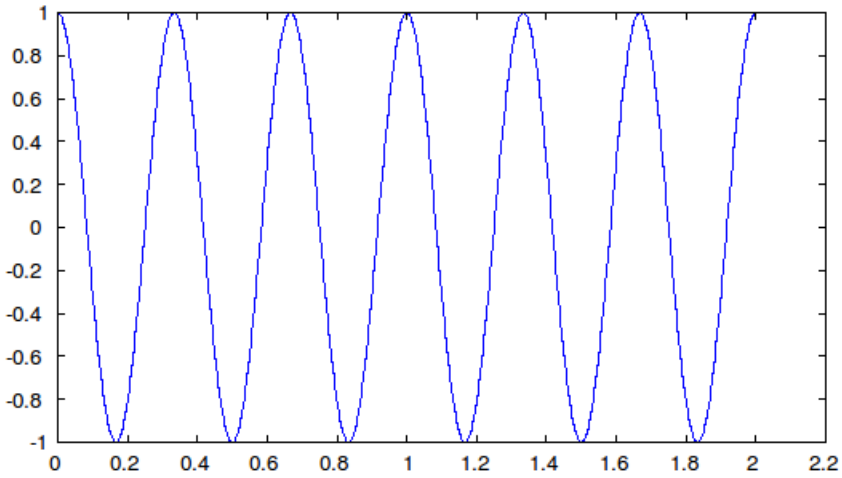


Figure 91: Plot as shown when saved as an image

Topic 6.6: Where Are My Saved Images?

If you just used the **print** function, you may be wondering, *Where did my saved image go?* Unless you spelled out the full path name in the **print** command, it went into your working directory. (See Topic 1.2.1: Setting the Working Directory Using the cd Command for more information.) To confirm that your file has been properly saved, you can either use the standard file tools (Windows Explorer, Linux Nautilus, etc) or you can print out the contents of the current directory using the **ls** command (FD, p. 419).

Topic 7: Working with WAV Files

A WAV (.wav) file is a standard, Microsoft audio file. Freemat provides a function to both read from and write to these types of file. It allows you to read the samples (mono and stereo), the sample rate, the number of bits per sample used to digitize the audio, only retrieve a portion of the file if you so desire, and to read the total size of the file.

Topic 7.1: Reading a WAV File Size

Caution - Input WAV Files Must be in the Working Directory

As of this writing, any .wav files to be read must be in the working directory. Files just within the path as defined by the Path Tool will not be found and Freemat will return an error.

We'll look at the reading first, which uses the **wavread** function (FD, p. 365). Let's start by determining the size of the file as well as whether it is a monophonic or stereo signal. The syntax is:

```
y=wavread('filename.wav','size')
```

where: y = variable used to store the two-dimensional matrix with the number of audio samples and the number of audio channels (1 for monophonic, 2 for stereo).

filename.wav = filename of WAV file.

This is the command to use when you don't know the size of the file, or whether it is mono or stereo.

Example - Checking a WAV File Size

Let's say I have a WAV audio file named **myaudio.wav**. To check the file size, I'll use the **wavread** function with the **'size'** handle, as follows:

```
y=wavread('myaudio.wav','size')
y =
    90061    2
```

This means that I have a WAV file that is stereo (that's the "2") and has 90061 audio samples in each channel (left and right).

Topic 7.2: Reading a WAV File

To read a .wav file, use the following syntax:

```
y=wavread('filename.wav')
```

where:

y = the variable used to store the audio samples. The **.wav** file can be either monophonic or stereo.

filename.wav = a string containing the audio filename as listed in the directory.

The matrix containing the audio samples will either be a 1 or 2 column matrix. The size of the matrix will be:

```
[x d]
```

where:

x = number of samples in the audio file

d = 1 for a monophonic signal or 2 for a stereo signal. In the case of a stereo signal, the left channel will be column 1 and the right channel will be column 2.

When a WAV file is stored, it's stored as either 8 or 16 bit samples, typically. An 8 bit sample will have acceptable values of 0 - 255. A 16 bit sample will have acceptable values of 0 - 65535. When Freemat imports the samples, it normalizes the amplitudes to fall within the range of -1 to 1. To me, this is a great thing. It makes it much easier to process the samples later on if they are already normalized. Otherwise, I would have to normalize them myself.

Example - Reading from a WAV File

I'll use a .WAV file, **myaudio.wav**. It's a stereo audio file of a human voice.

```
% Read in and display .WAV data
% This script will read in the stereo
% .WAV file called 'myaudio.wav'. It will
% then graph and display the left and right
% channel data.
% Script filename = audioPlot.m
[y sampleRate bitDepth]=wavread('myaudio.wav'); % Read in the data
yLeft=y(: 1); % Create array for left channel.
yRight=y(: 2); % Create array for right channel.
t=(1:length(yLeft))/sampleRate; % Create time array
plot(t,yLeft); % Plot the left channel.
xDataset; % Set the x-axis so that it fills up the plot.
grid on; % Turn on the grid on the plot.
```

Here's how the graph of the left channel of this .WAV file appears.

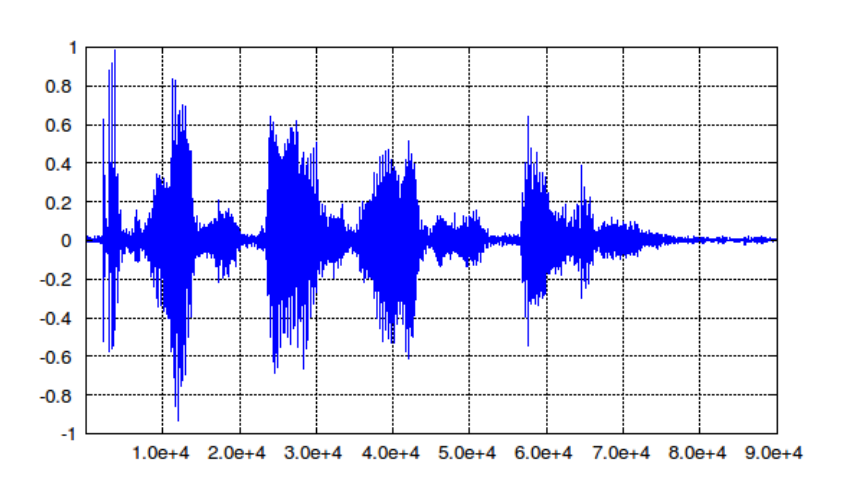


*Figure 92: Plot of left channel from stereo audio file, **myaudio.wav**. This is a WAV audio file of human voice. Note that the amplitudes are normalized to fall within the range of -1 to 1.*

Topic 7.3: Reading the Sample Rate and Bit Depth

To read the sample rate and the number of bits, you use the following syntax:

```
[y sample_rate bit_depth] = wavread('filename.wav')
```

where:

y = the variable used to store the audio samples.
sample_rate = the sample rate, in Hz, of the digitized audio.
bit_depth = the number of bits used to digitize each sample. **Note that, in order to read the number of bits, you must also read the sample rate. However, you do not need to read the bit depth in order to read in the sample rate.**
filename.ave = the audio file as listed in the directory.

By reading the sample rate, you can make plots that have time on the horizontal axis and/or you can make frequency domain plots that have the horizontal axis in Hz, rather than the default sample numbers.

Example - Reading the Sample Rate & Bit Depth

Here's an example showing how to read in the .WAV data, the sample rate, and the bit depth. Then we'll create a time array for the x-axis on the graph..

```
% Read in and display .WAV data
% This script will read in the stereo
% .WAV file called 'myaudio.wav'. It will
% then graph and display the left
% channel data.
% Script filename = audioPlot.m
[y sampleRate bitDepth]=wavread('myaudio.wav'); % Read in the data
yLeft=y(: 1); % Create array for left channel.
yRight=y(: 2); % Create array for right channel.
t=(1:length(yLeft))/sampleRate; % Create time array
plot(yLeft); % Plot the left channel.
xDataset; % Set the x-axis so that it fills up the plot.
grid on; % Turn on the grid on the plot.
printf('The bit depth is %d bits.\n',bitDepth);
```

Running this script, we get this in the Command Window:

```
--> audioPlot
Warning: Newly defined variable nargin shadows a function of the same name.  Use
clear nargin to recover access to the function
Warning: Newly defined variable format shadows a function of the same name.  Use
clear format to recover access to the function
The bit depth is 16 bits.
```

The graph appears as follows:

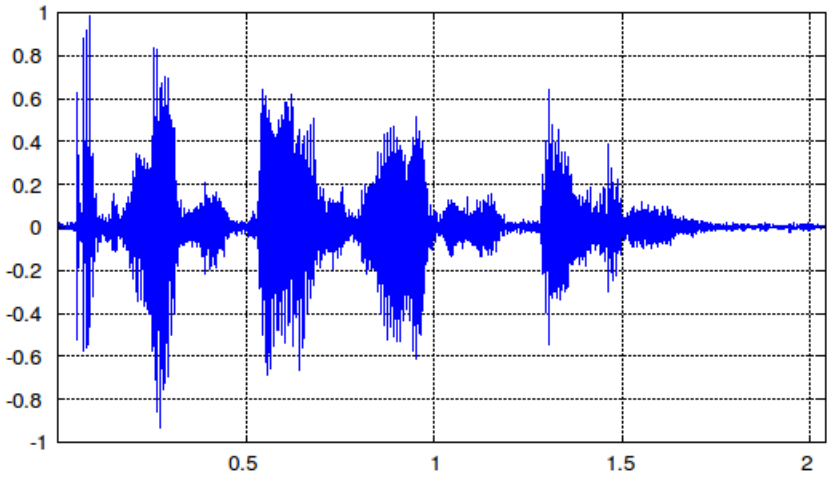


*Figure 93: Plot of left channel of **myaudio.wav** file with time (in seconds) on the horizontal axis.*

Topic 7.4: Reading Only a Portion of a WAV File

Caution - Reading on a Portion Does Not Work

As of this writing, there's a bug in the wavread function such that it is not possible to read a portion of a .WAV file. If you try to do so, whether using only a scalar (which is supposed to read from 1 to the scalar number of samples) or an array (which provides the beginning and ending sample to read in), you will receive an error similar to the following:

```
--> x=wavread('myaudio.wav', 8192);
Warning: Newly defined variable nargin shadows a function of the same name.
Use clear nargin to recover access to the function
Warning: Newly defined variable format shadows a function of the same name.
Use clear format to recover access to the function
In /usr/share/freemat/toolbox/io/wavread.m(wavread) at line 157
    In docli(builtin) at line 1
    In base(base)
    In base()
    In global()
Error: Cannot combine data of different integer data classes
```

This will be fixed in a future release of Freemat.

Topic 7.5: Writing a WAV File

When writing a WAV file, you start with a one- or two-dimensional matrix. These are the audio samples. The function to use is wavwrite (*FD*, p. 412), and its syntax is as follows:

```
wavwrite(x,'filename.wav')
```

where:

x = variable containing audio samples

filename.wav = name of file into which the WAV file will be stored.

Remember: Files written from Freemat will be stored into the working directory. You can change this by either changing the working directory of by specifying the full path name into the **wavwrite** function. If you're using Windows Vista, you need to write to a directory that allows writing. The Program Files folder does not allow writing from Freemat. Therefore, you need to either change the working directory or you need to specify the full pathname in the writing function (such as wavwrite or print) to something other than your Program Files folder. Otherwise, you won't be saving your files. They will simply not be stored. See Topic 1.2.1: Setting the Working Directory Using the cd Command.

Example - Creating a Monophonic WAV Audio File

This short script creates a tone at middle C (261.626 Hz). The script uses the default Freemat sample rate (8000 Hz), and it will last for 1 second. This will be a one-dimensional matrix, so the WAV file will be monophonic. You can play the resulting WAV file, "sinewave.wav", using any WAV player, such as Window's built-in audio player, Windows Media Player.

```
f=261.626;
t=1:8000;
tone_c=0.8*cos(2*pi*t*f/8000);
wavwrite(tone_c,'sinewave.wav');
```

Example - Creating a Stereo WAV Audio File

This script creates two tones, one at middle C (261.626 Hz) and the second harmonic of middle C (~523 Hz). The middle C tone is written into the left channel (column 1) of a matrix; the harmonic is written into the right channel (column 2).

```
f=261.626;
t=1:8000;
tone_c(:,1)=0.8*cos(2*pi*t*f/8000);
tone_c2(:,1)=0.8*cos(2*pi*t*2*f/8000);
```

The next line will create a two-column matrix. The middle C tone (tone_c) will be placed into column 1, which corresponds to the left channel. The harmonic (tone_c2) will be placed into column 2, which corresponds to the right channel.

```
y=[tone_c tone_c2];
wavwrite(y,sample_rate,bit_depth,'sinewave_stereo.wav');
```

By default, Freemat will assume that the samples have an 8 kHz (8000 Hz) sample rate and 16 bit depth. However, you can change this by adding the sample rate and the bit depth into the write operation. The syntax is as follows:

```
wavwrite(x,sample_rate,bit_depth,'filename.wav')
```

where:

x = one- or two-dimensional matrix containing audio samples.
sample_rate = sample rate, in Hz, of audio samples.
bit_depth = number of bits used to digitize each sample. **NOTE: This is optional.**
'filename.wav' = name of file used to store resulting WAV file.

Example - Writing a WAV File with a Different Sample Rate

In this example, I'll create a middle C tone at a sample rate of 16 kHz (16000 Hz) and 8 bit depth.

```
sample_rate=16000;
bit_depth=8;
f=261.626;
t=1:sample_rate;
y=0.8*cos(2*pi*t*f/sample_rate);
wavwrite(y,sample_rate,bit_depth,'my_tone.wav');
```

Topic 8: The Frequency Domain

Much of the data one collects is time-domain based. For example, if you make an audio recording, the data is recorded and stored as time-domain samples. But the frequency-domain can provide details that are not readily apparent in the time-domain. For digital data, you can see the frequency-domain using the discrete Fourier transform (DFT). Within Freemat, you do this using the fast Fourier transform (FFT) with the **fft** function (FD, p. 387).

Topic 8.1: Using the FFT

The **fft()** function is used as follows:

```
y=fft(x)
```

where: x = an array, preferably one whose length is a power of 2.
The **fft** function is an efficient means of computing a discrete Fourier transform or DFT. The FFT works most efficiently when its length is some power of 2, such as 2, 4, 8, 16, and so on. Don't worry. If the number of samples is not a power of 2 (for example, you pass a set of samples of length 101), the fft command will still work. It will just take a little bit longer. If the samples passed to the **fft** function are real samples (as opposed to complex samples), then the last half of the FFT will be a complex-conjugated, mirror image of the first half. This means that frequency bin 2 will be the complex conjugate of frequency bin N, where N is the total number of bins in the **fft** function output, frequency bin 3 will be the complex conjugate of frequency bin N-1, and so on. This means you only need the first half of the **fft** function values in order to have all of the information contained in the FFT.
The most common display of FFT data is the magnitude data. However, the output of an FFT is complex data. This makes it possible to also calculate the phase for each frequency bin.

Example - FFT of Real Samples

This example uses a short number of samples of a cosinusoid to show how the last half of the frequency bins of an FFT of real samples is the complex-conjugated, mirror image of the first half.

```
--> t=1:7;
--> x=cos(2*pi*t/4);
--> y=fft(x);
--> y'
ans =
  -1.0000 + -0.0000i
  -1.3019 +  0.6270i
   1.7470 -  2.1906i
   0.0550 -  0.2408i
   0.0550 +  0.2408i
   1.7470 +  2.1906i
  -1.3019 -  0.6270i
```

The first number (-1.0000 + 0.0000i) is the DC component, or DC bias, of the signal. The next number, -1.3019 + 0.6270i, is the complex conjugate of the last number, -1.3019 - 0.6270i. The second number, 1.7470 - 2.1906i, is the complex conjugate of the second-to-last number, 1.7470 + 2.1906i. And so on.

Complex numbers, such as those that are the output of the **fft** function, cannot be plotted on a two-

dimensional graph. The most common display of the FFT is the magnitude of each frequency bin. This is accomplished using the absolute value or **abs** function. The magnitude of a complex number is its absolute value.

Example - Creating a Basic FFT Display

We'll start with a basic sinewave, as shown in Figure 94. This sinewave has a frequency of 31.25 Hz.

```
% Basic FFT display
% filename = basicFftdB.m
close('all');
N=256;
sampleRate=1000;
t=(0:N-1)/sampleRate;
fc=(N/32)*sampleRate/N;
x=cos(2*pi*t*fc);
figure(1)
plot(t,x,'k-','linewidth',2);
grid on;
xlim([0 4/fc]);
figure(2);
f=(0:(N-1))*sampleRate/N;
y=abs(fft(x));
yMag=20*log10(y);
plot(f,y,'linewidth',2)
xDataset
grid on;
```

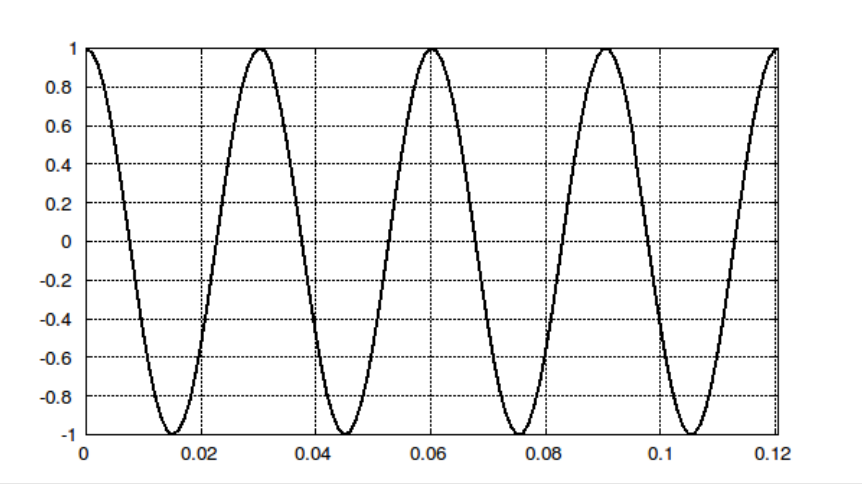


Figure 94: Basic sinewave with a frequency of 31.25 Hz.

The second display is the frequency domain created using the **fft** command:

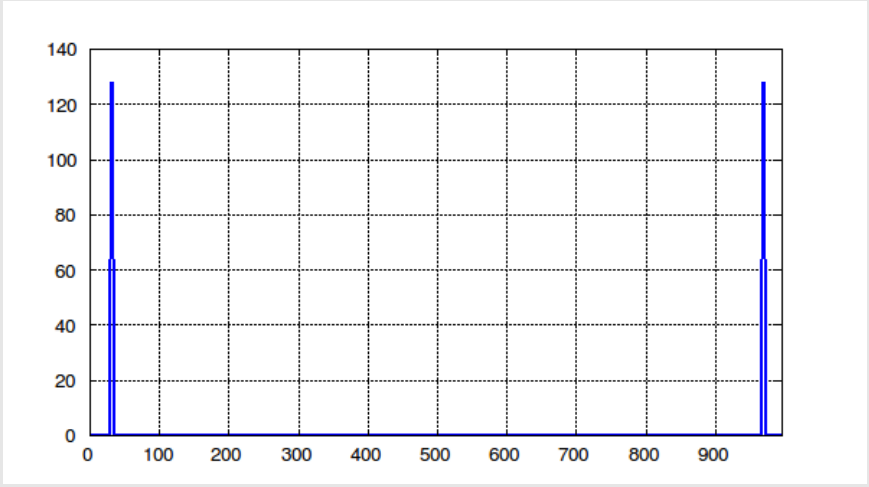


Figure 95: Spectrum of sinewave. The sinewave has a frequency of 31.25 Hz.

There are a couple of things to note about the spectral display. First, note that you see two "spikes" or "lines" in the spectral display. These correspond to the frequency of the sinewave. Second, note that there are two, not just one. This is due to the fact that the DFT occurs over the entire range of the sampling rate. In the case where the samples used in the FFT are real samples (as was done here), the upper-half of the spectral display is a mirror image of the lower-half. This is shown in Figure 96.

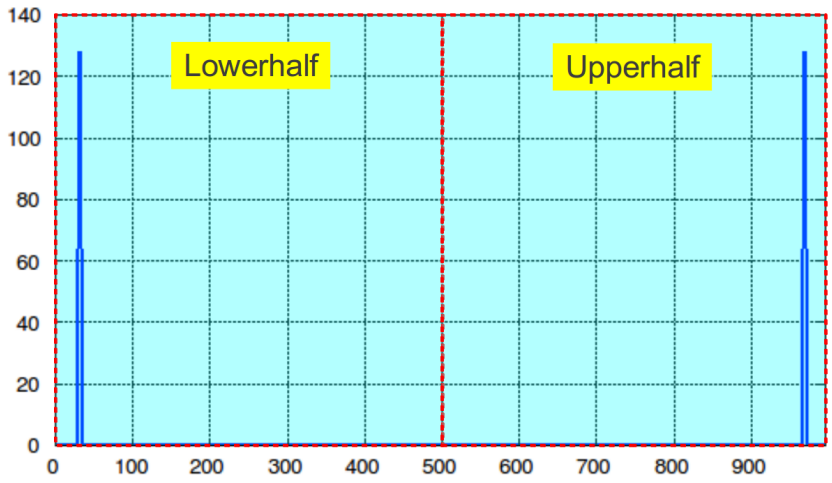


Figure 96: This shows the upper and lower half of an FFT. Note that the upperhalf is a mirror image of the lowerhalf.

A typical spectral display only shows the lowerhalf, which makes sense when the samples used to create the display are real. According to Nyquist theory, the maximum frequency that can be accurately reproduced is one-half of the sample rate.

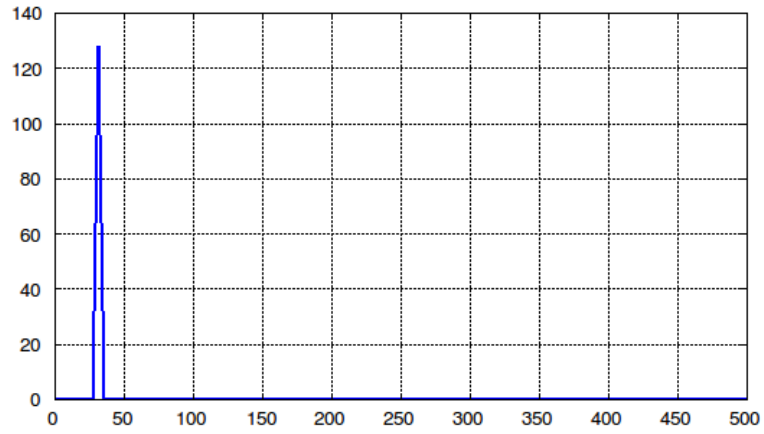


Figure 97: A typical spectral display created using an FFT only shows the lowerhalf of the full display. That's due to the fact that the information is contained fully within this section of the FFT.

Third, note that the "line" has a small amount of width. This width is based on the width of the bins in the FFT. Theoretically, a sinewave has zero bandwidth. However, this would require an infinitely long sinewave. Reality says that the minimum width that is discernable on an FFT display is called the "bin width". This is equal to (Sample Rate)/N, where N = number of samples in the FFT.

Fourth, note the amplitude in Figure 97. Note that the vertical scale is linear. The vertical scales for spectral displays are typically logarithmic, specifically "decibel" or "dB". The data used to feed most spectral displays is volts. The spectral displays themselves we like to use "watts". The conversion from volts to watts is $20\log_{10}(x)$, where x = FFT amplitude. This is shown in Figure 98.

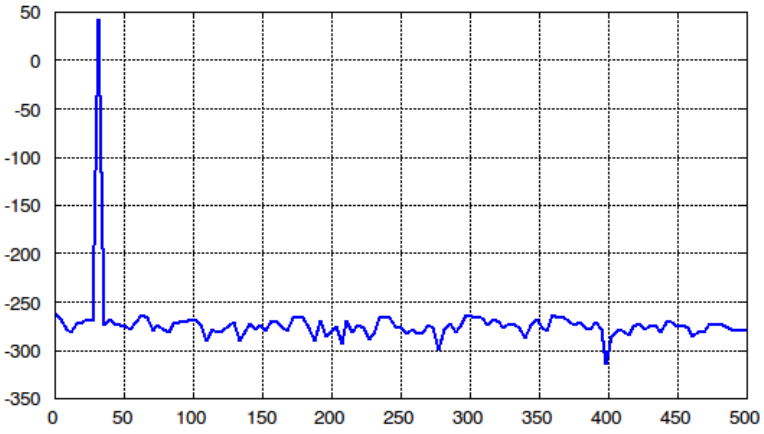


Figure 98: This is the spectral display of a sinewave created using a 267-point FFT. The vertical scale is in dB.

The amplitude of the sinewave that was put into the **fft** command was 1. Yet, the amplitude on the spectral display is listed as somewhere between 120 - 140. We'll go into how to calibrate the scale vertically in the next topic.

Topic 8.2: Calibrating the Vertical Scale

Note the vertical scale in Figure 98. The peak is near 50 dB. This is due to the fact that, as the number of samples goes up, the total sum also goes up. To correct for this, the Freemat version of the FFT can be normalized by 2/N, where N = the number of samples in the FFT.

Example - Calibrating the Vertical Scale of the FFT

This will use one of the previous examples to calibrate the vertical scale, in dB, for the Freemat version of the FFT.

```
close('all');
N=256;
sampleRate=1000;
t=(0:N-1)/sampleRate;
fc=(N/32)*sampleRate/N;
x=cos(2*pi*t*fc);
f=(0:(N-1))*sampleRate/N;
y=abs(fft(x))*2/N;
yMag=20*log10(y);
plot(f,yMag,'linewidth',2)
xDataset
xlim([0 sampleRate/2]);
%ylim([-110 10]);
grid on;
```

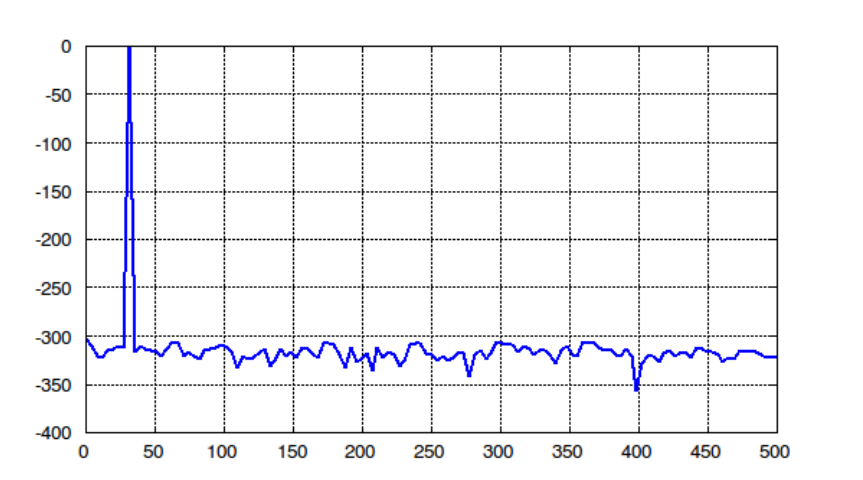


Figure 99: This is the FFT with a calibrated vertical scale. This is similar to Figure 98 except that the vertical scale is properly calibrated.

Topic 8.3: Windowing the Time-Domain Samples

We'll start by reading in samples from an audio file, then converting a portion of those samples into a spectral display.

```
% Read in and display .WAV data
% This script will read in the stereo
```

```
% .WAV file called 'myaudio.wav'. It will
% then graph and display the left
% channel data.
% Script filename = audioPlotFFT.m
[y sampleRate bitDepth]=wavread('myaudio.wav'); % Read in the data
yLeft=y(: 1); % Create array for left channel.
yRight=y(: 2); % Create array for right channel.
t=(1:length(yLeft))/sampleRate; % Create time array
tPortion=25000;
N=2048;
signal=yLeft(tPortion:(tPortion+N-1));
signal=signal';
yf=abs(fft(signal))*2/N;
yfMag=20*log10(yf);
yfw=abs(fft(h))*2/N;
yfMagw=20*log10(2*yfw);
f=(0:(N-1))*sampleRate/(N*1000);
plot(f,yfMag,'b-');
xlim([0 sampleRate/2000]);
ylim([-120 0]);
grid on;
```

The spectral snapshot is shown in Figure 100.

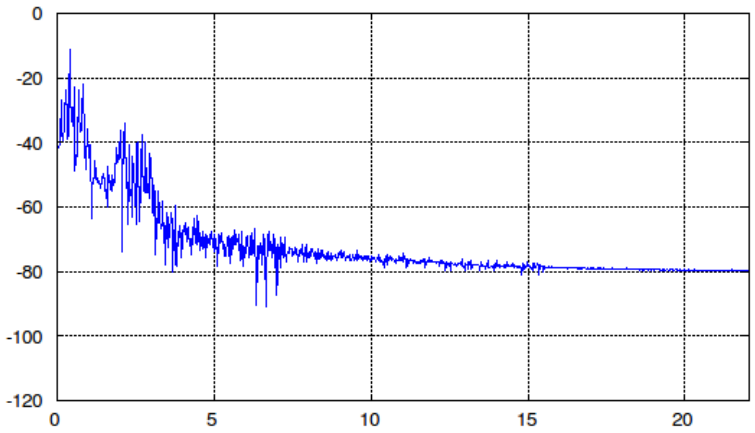


Figure 100: This is the spectral display of a portion of an audio file.

Note the flat section of the spectral display near the higher end frequencies. The flattened section of the FFT is due to "smearing" from higher amplitude components.

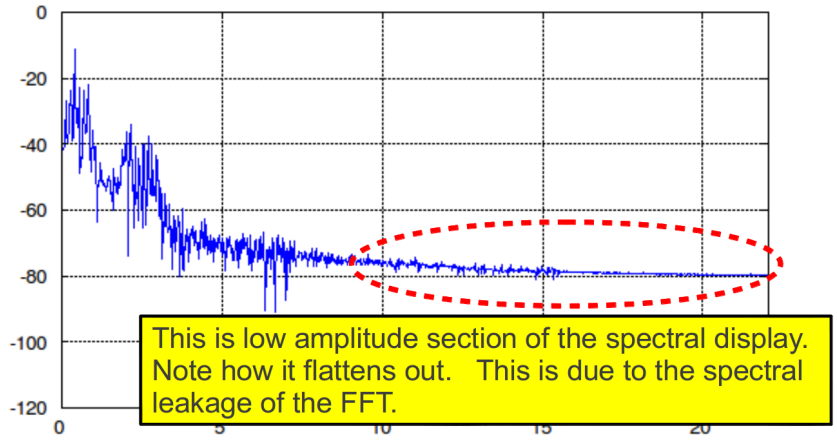


Figure 101: Spectral display of audio file with spectral leakage showing on higher frequencies.

Does this make sense to you? If you're new to the Fourier transform, especially the discrete Fourier transform, then it probably doesn't make sense. It's definitely not intuitive to the casual observer. One way to understand it is to look at what a FFT is doing. A FFT is a method for converting a finite time-domain signal into a discrete spectral display. And "discrete" means that the frequency display is divided up into a number of bins. The center of each bin is an integer value of (Sample Rate)/N, where N = the number of samples used to create the FFT. For example, if the sample rate is 1000 Hz and there are 8 samples used to create the FFT, then each bin will be centered at an integer value of 1000/8 = 125 Hz.

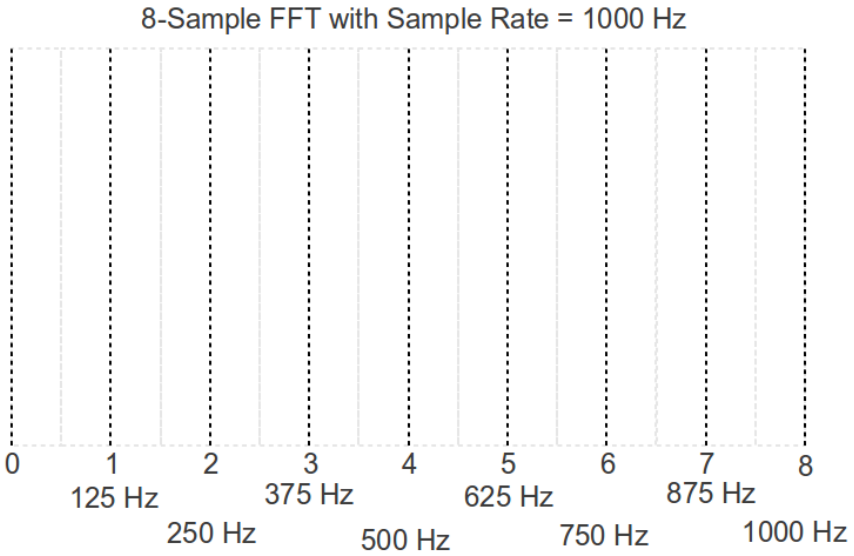


Figure 102: This is how the spectral display of an 8-point FFT in which the sample rate is 1000 Hz. The black, dotted lines are the center of each bin.

A FFT has only a finite number of cells, but a signal going into it may be one of an infinite number of frequencies. If the frequency of the signal lines up at (or very near) one of the center of the frequency bins, the FFT will show an accurate representation of the signal. The signal will correlate perfectly (or

almost perfectly) with the center of the bin. However, as the signal center frequency moves towards the edge of the bins, the signal peak amplitude will drop. This is due to the fact that it will no longer correlate perfectly with the bin on either side. Further, it will correlate somewhat with bins further away, with the amount of correlation decreasing as the change in frequency increases. This correlation of the signal with bins beyond the center frequency is called "spectral leakage" or "spectral smearing".

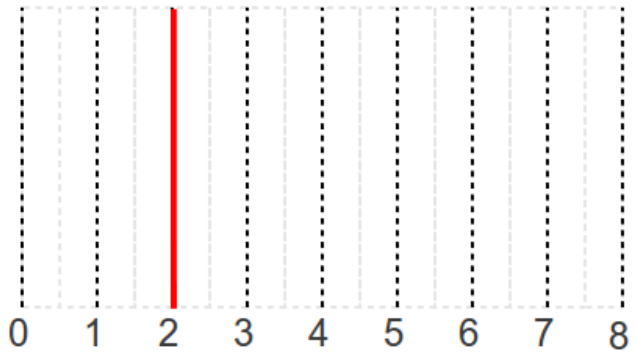


Figure 103: This is how a signal lines up with the center of the bin.

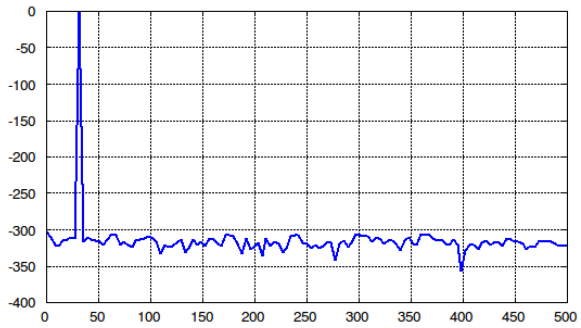


Figure 104: This is how the spectrum of a sinewave appears when the sinusoid frequency lines up with the center of one of the frequency bins.

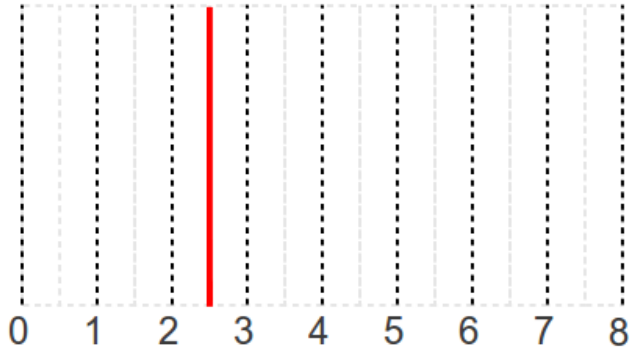


Figure 105: This is how the signal lines up with the edge of a bin.

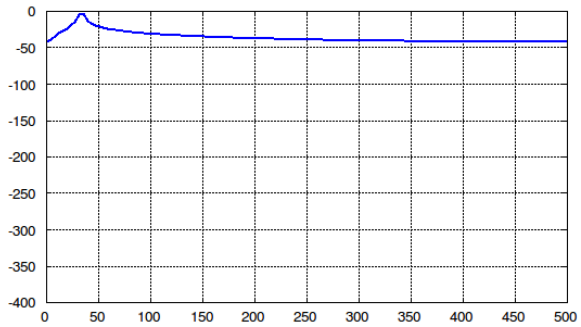


Figure 106: This is how the spectrum appears for a sinewave in which the frequency is at the edge of a frequency bin. Note the amount of energy present in the spectrum away from the center frequency. This is all due to "spectral leakage" or "spectral smearing".

As the table above shows, if the number of cycles of a periodic waveform is a non-integer number, there will be a discontinuity between the end of the sample set and the beginning. This is shown in Figure 107, which shows a shifted, non-integer number of cycles.

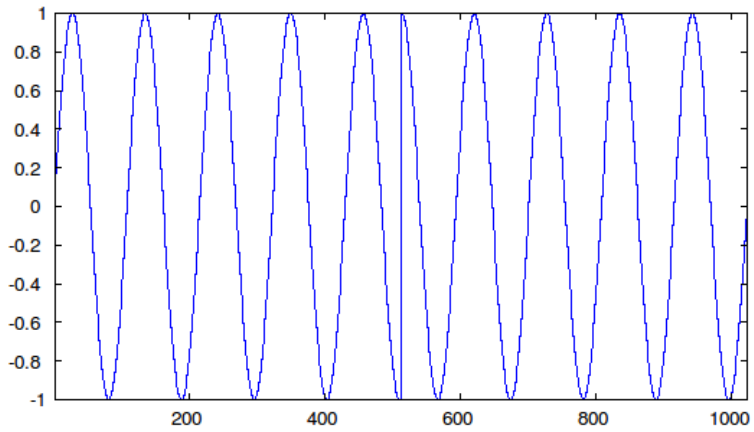


Figure 107: This is a non-integer number of cycles of a sinewave. In this example, there's a large discontinuity that has been shifted to the center of the display.

Any discontinuity will require higher frequency components in order to duplicate that discontinuity. Another way to understand spectral leakage is looking at what the FFT is doing. The fast Fourier transform is nothing more than an efficient method for calculating the discrete Fourier transform (DFT). Remember that. An FFT = DFT (although the reverse is not necessarily true, meaning DFT ≠ FFT). And what is a DFT? It's a correlation of a particular waveform against a set of sinusoids, called the basis set for the DFT.

The method used to alleviate this problem is called windowing. A window is a function that takes a vector and forces the beginning and ends to, or close to, zero. While Freemat does not have any built-in window functions (yet?), you can make your own. Below are a couple of simple ones.

Hanning window:

```
t=1:N;
w=0.5-0.5*cos(2*pi*t/N);
```

Raised cosine squared window:

```
t=1:N;
w=0.25*(1-cos(2*pi*t/N)).^2;
```

Blackman-Harris[2] window

```
t=1:N;
w=0.42-0.5*cos(2*pi*t/N)+0.08*cos(4*pi*t/N);
```

Here's a short set of commands to create a Hanning window, then display it.

```
--> N=1024;
--> t=0:N;
--> w=0.5-0.5*cos(2*pi*t/N);
--> plot(w)
```

The time-domain plot is shown below.

2 Interestingly, fred harris does not capitalize his names. It's all lowercase. He's a quirky fellow, but one of the greats of digital signal processing. So he's earned his quirks. But in his seminal paper on windowing, however, he called the "Blackman-Harris" window with his last name capitalized. I'm willing to go with that quirk, too.

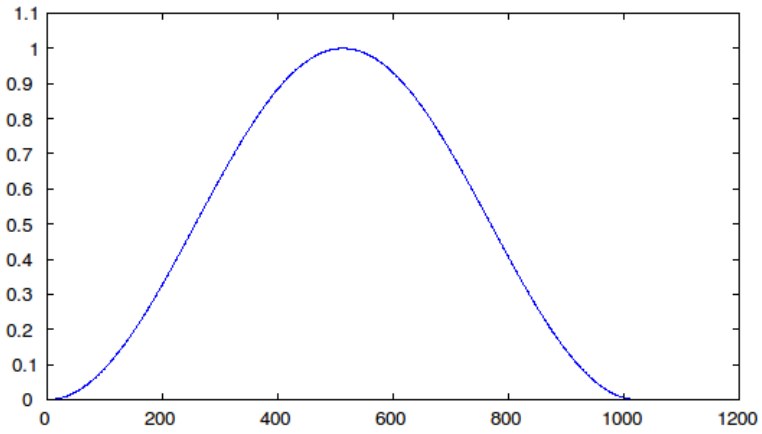


Figure 108: Display of a Hanning window. Note how it drops to zero on each end.

The figure below provides a general overview of how a window operates on a signal, in this case a sinewave.

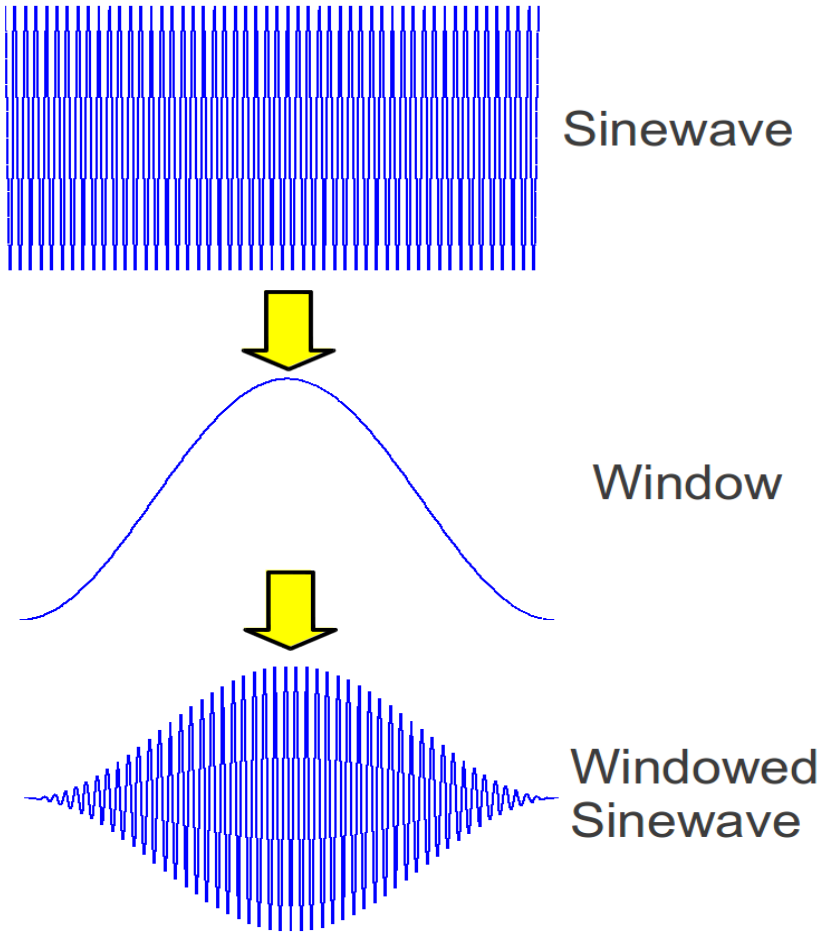


Figure 109: Overview of how a window works on a signal, in this case a sinewave.

Example - Adding a Window for the FFT Plot

We'll add a Hanning window to the spectral display of the audio plot shown previously.

```
[y sampleRate bitDepth]=wavread('myaudio.wav'); % Read in the data
yLeft=y(: 1); % Create array for left channel.
yRight=y(: 2); % Create array for right channel.
t=(1:length(yLeft))/sampleRate; % Create time array
tPortion=25000;
N=2048;
signal=yLeft(tPortion:(tPortion+N-1));
signal=signal';
% Window the data with a Hanning window
w=0.5-0.5*cos(2*pi*(1:N)/N);
h=w.*signal;
tPart=t(tPortion:(tPortion+N-1));
figure(1)
plot(tPart,signal);
xDataset;
grid on;
yf=abs(fft(signal))*2/N;
yfMag=20*log10(yf);
yfw=abs(fft(h))*2/N;
yfMagw=20*log10(2*yfw);
f=(0:(N-1))*sampleRate/(N*1000);
figure(2);
plot(f,yfMagw,'b-');
xlim([0 sampleRate/2000]);
ylim([-120 0]);
grid on;
```

This windowed audio plot appears as follows.

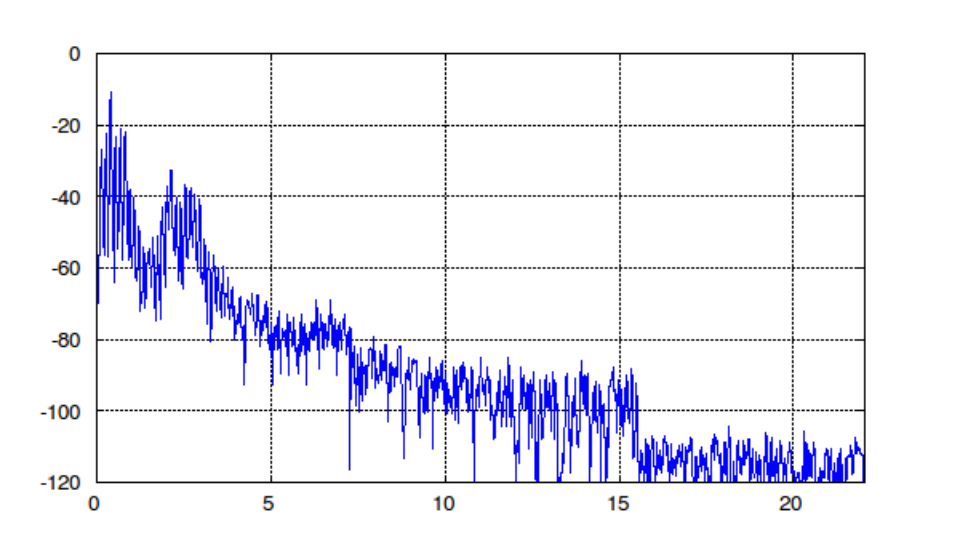


Figure 110: FFT plot of portion of audio plot with Hanning window

Now compare this with the original FFT plot of this audio file, as shown in Figure 100. The first thing you'll notice is that the higher frequency, lower amplitude portion of the signal is better represented.

The second thing, which is harder to discern, is that the peak amplitude has dropped. This is the effect of the window. Without going into the math behind it, the drop in the peak amplitude is 6.02 dB. To calibrate the plot, we'll add this back. We'll create a plot of the two graphs (windowed vs non-windowed) to better show the difference between the plots.

```
[y sampleRate bitDepth]=wavread('myaudio.wav'); % Read in the data
yLeft=y(: 1); % Create array for left channel.
yRight=y(: 2); % Create array for right channel.
t=(1:length(yLeft))/sampleRate; % Create time array
tPortion=25000;
N=2048;
signal=yLeft(tPortion:(tPortion+N-1));
signal=signal';
% Window the data with a Hanning window
w=0.5-0.5*cos(2*pi*(1:N)/N);
h=w.*signal;
tPart=t(tPortion:(tPortion+N-1));
figure(1)
plot(tPart,signal);
xDataset;
grid on;
yf=abs(fft(signal))*2/N;
yfMag=20*log10(yf);
yfw=abs(fft(h))*2/N;
yfMagw=20*log10(yfw)+6.02;
f=(0:(N-1))*sampleRate/(N*1000);
figure(2);
plot(f,yfMagw,'r-');
hold on;
plot(f,yfMag,'b-');
hold off;
xlim([0 sampleRate/2000]);
ylim([-120 0]);
grid on;
legend('Windowed FFT','Non-windowed FFT');
```

The result is shown below.

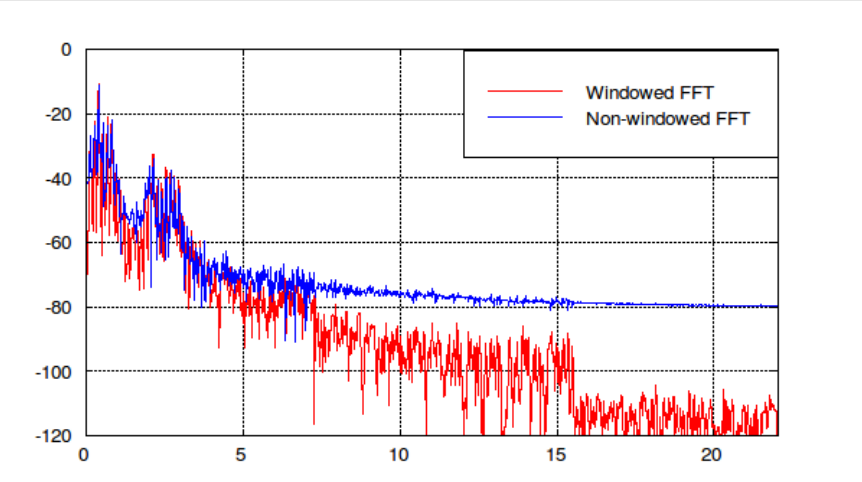


Figure 111: FFT of audio plot comparing windowed vs. non-windowed graphs. The windowed graph has been amplitude calibrated.

Topic 8.4: Calculating the Inverse FFT

Freemat provides a function, **ifft**, which will calculate the inverse FFT of an input array. However, the output array will be complex.

> The **ifft** function is not listed in the online reference nor in the *Freemat v4.0 Documentation*. Not to worry. It's there and it works.

If the original function was real samples, such as samples from a WAV file, then the output of the **ifft** function will also be real samples. In the case of Freemat, this is actually the case, though the samples don't know it. (Yes, that was a joke.) The output of the **ifft** function is complex samples, but if the input was real samples, then the output of the **ifft** function will have zero values for the imaginary components. It's a straightforward affair to convert the complex to real samples. This can be done with the **real** function.

Example - Calculating the Inverse FFT

This example will calculate a time-domain waveform, then the FFT of this waveform, then the inverse FFT. The **real** function will be used to recover the real waveform from the complex array output from the **ifft** function.

```
--> x=linspace(0,10*pi,1000);
--> y=cos(x);
--> yf=fft(y);
--> y2=real(ifft(yf));
--> plot(x,y,'k-',x,(y2+2),'b--','linewidth',2)
```

Here's the resulting plot.

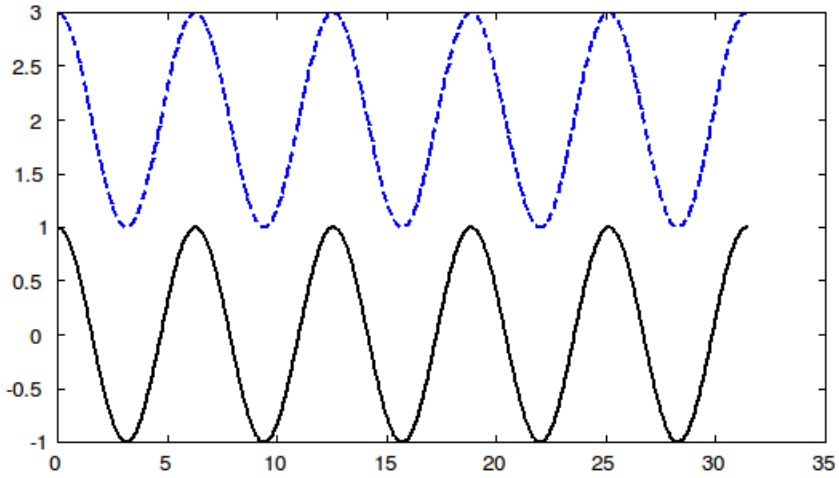


Figure 112: Comparison of original waveform (bottom) and the waveform resulting from the FFT followed by inverse FFT (top).

Topic 8.5: Setting the Frequency Units for the FFT Display

There are three ways to display the units for the horizontal axis of an FFT display. These are:

- samples
- fraction of the sampling frequency
- actual frequency

The simplest is to use the sample number. This requires no extra work on your part. By calculating the FFT and displaying the magnitude of the result, the default is to display the sample number on the horizontal axis. The second method is to display the frequency as a fraction between 0 - 1 of the sampling rate. The last method, which has been used up til now, is to actually calculate the proper frequency scale of the spectral display.

When talking about signal processing, most FFTs are displayed using the fraction of the sample frequency as the horizontal axis. This is shown using an example.

Example - FFT Display with Different Types of Horizontal Values

I'll create a 512-point sine wave, calculate the FFT of that sine wave, then create a basic plot with the sample number used for the horizontal axis.

```
--> N=512;
--> x=linspace(0,200*pi,N);
--> y=cos(x);
--> yf=abs(fft(y))/N;
--> plot(yf)
```

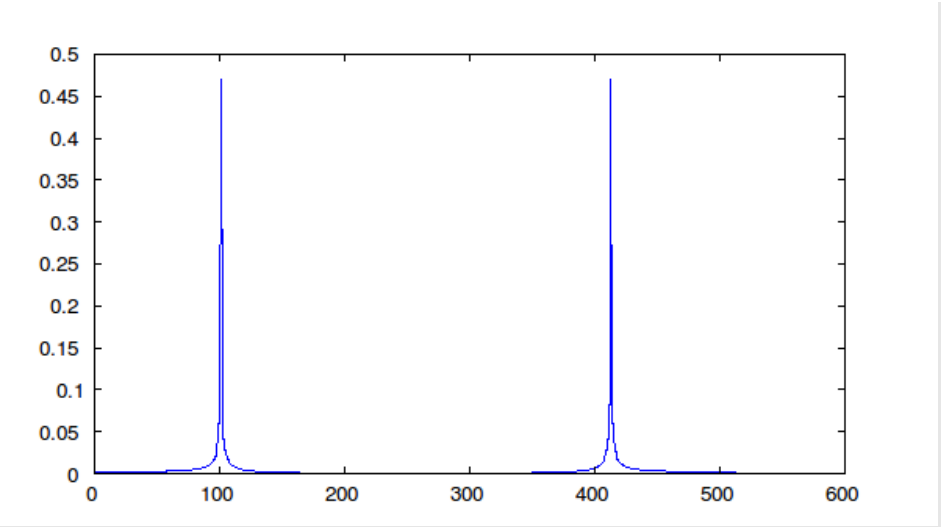


Figure 113: FFT display with sample numbers for the horizontal axis

Next, we'll change the horizontal axis so that it is the fraction of the sample rate. This uses the following commands:

```
--> N=512;
--> x=linspace(0,200*pi,N);
--> y=cos(x);
--> yf=abs(fft(y))/N;
--> f=(0:length(yf)-1)/N;
--> plot(f,yf);
```

The result is shown in Figure 114.

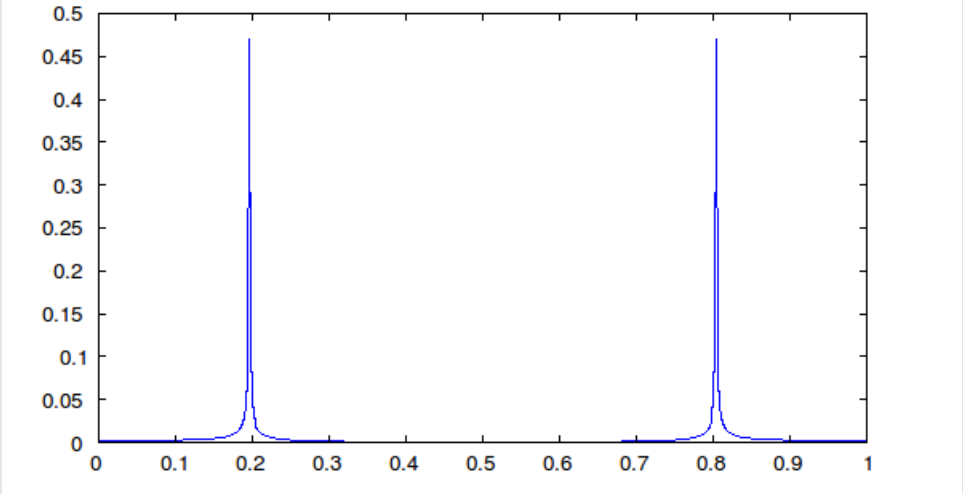


Figure 114: FFT display using the fraction of the sample rate for the horizontal axis

The last method for displaying the horizontal axis is the actual frequency. In order for this to work, you have to know the actual sample frequency used to sample the signal. If you don't know that information (or if its irrelevant), then you need to use one of the other two methods for displaying the horizontal axis. To calculate the actual frequency, use the following equation:

$$x_f = \frac{f}{\text{samples}} * (\text{sample rate})$$

where: x = vector used to store the frequency values
f = frequency sample
samples = total number of samples used to create the FFT.
sample rate = frequency used to sample the digital signal. This is the inverse of the sample period.

Example - Displaying Actual Frequency for an FFT Display

I'll create a sine wave using a known sample rate, calculate the FFT, then display the magnitude of the FFT values using frequency values on the horizontal axis.

```
--> N=512;
--> sampleRate=8000; % Sample rate, in Hz.
--> freq=2300; % This will be the frequency, in Hz, of the sine wave.
--> samples=0:N-1;
--> y=sin(2*pi*samples*freq/sampleRate);
--> yf=abs(fft(y))/N;
--> f=(0:length(yf)-1)*sampleRate/N;
--> plot(f,yf);
```

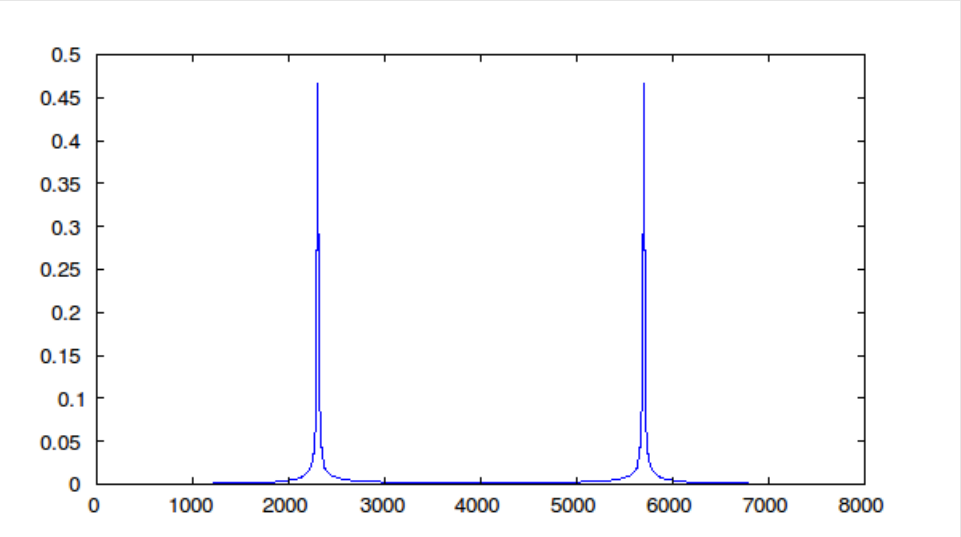


Figure 115: FFT display using actual frequencies of a 100 kHz sine wave sampled at 1 MHz.

Note that, in Figure 115, the horizontal axis uses a lot of zeros. One way around this is to use different suffixes, such as kHz (kilohertz, or 10^3 Hz) and MHz (megahertz, or 10^6 Hz), to make it easier to read. In this example, I'll divide the horizontal axis by 1000 to make the units kHz. This is shown in Figure 116.

```
--> N=512;
--> sampleRate=8000;
--> freq=2300; % This will be the frequency of the sine wave.
--> samples=0:N-1;
--> y=sin(2*pi*samples*freq/sampleRate);
--> yf=abs(fft(y))/N;
--> f=(0:length(yf)-1)*sampleRate/(N*1000);
--> plot(f,yf);
```

```
--> xlabel('Frequency (kHz)')
--> sizefig(900,600); % This command gets around the problem of the 'xlabel'
command.
```

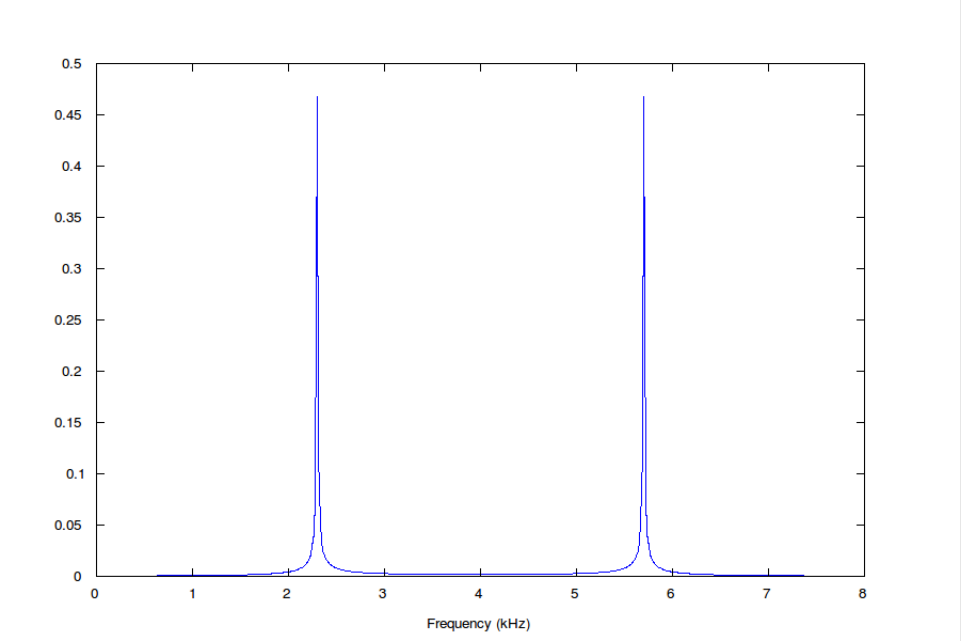


Figure 116: FFT display of 2.3 kHz sine wave. This is using an 8 kHz (8000 Hz) sample rate and the display has horizontal units of kHz.

Topic 9: Basic Numerical Methods

Numerical methods provide a way to solve problems quickly and easily compared to analytic solutions. Whether the goal is integration or solution of complex differential equations, there are many tools available to reduce the solution of what can be sometimes quite difficult analytical math to simple algebra and some basic loop programming. FreeMat has a big advantage in terms of simple looping thanks to the new JIT compiler, which can run loops and other simple programming much faster than even most commercial packages such as MATLAB.

Topic 9.1: Root Finding

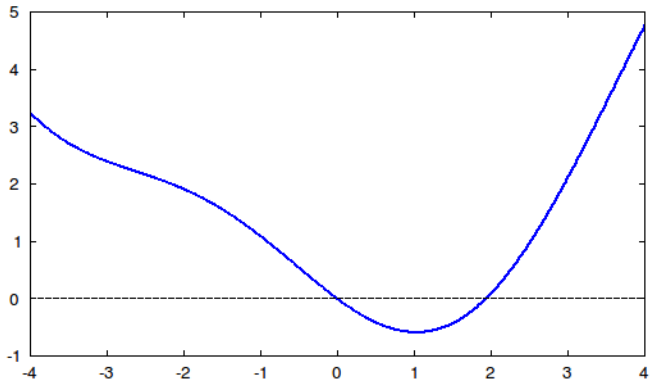
One of the most basic applications of numerical methods is to find the roots of a single equation. In cases where an equation's solution cannot be obtained by algebraic means (as is the case with many non-linear equations), there are several methods for finding the roots (solutions) with the power of a computer and some basic algorithms, which will be discussed here.

Topic 9.1.1: The Bisection Method

If you have ever searched in a phone book for a name, you've intuitively performed something like the bisection method. The bisection method is a simple way to find a single root of an equation, given that the root exists between two bounds (which must be defined at the outset of the script), and it is the only root between those bounds. The method then reduces these bounds by half, always selecting the half containing the root, until the bounds are reduced below an acceptable error limit.

Example: Let's examine the nonlinear equation

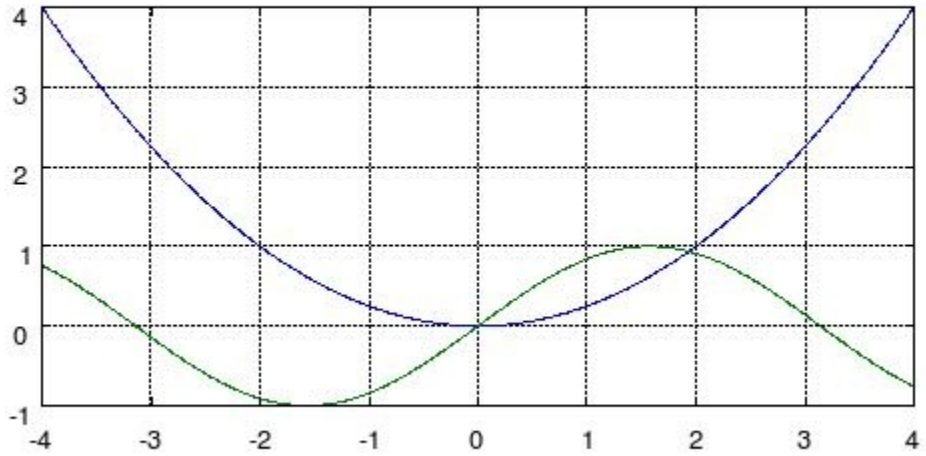
$$\left(\frac{x^2}{4}\right)-\sin(x)=0$$



We can quickly notice several things about this equation. First, it can be described as the intersection of two functions,

$$F1(x)=\frac{x^2}{4}$$

$$F2(x)=\sin(x)$$



Second, we see either by inspection of the equation or either graph that there are two solutions, one at zero (a trivial solution) and one near 2. Finally, we see that this equation cannot be solved with conventional algebra, so the only way to obtain a solution is either graphically or by numerical methods. The basic operating principle of the bisection method works as follows:

You are looking for the name Stevens in a phonebook. Pick up the phonebook, and open it to the middle, you will find names beginning with M. Which half of the book is Stevens in? As Stevens is in the latter half, we then take *that half* of the book, and split *it* in two. The resulting page has the name Reuben on it. Stevens comes after Reuben (i.e.: Stevens is in the second half), and so we split the second half again. Continue this process, always selecting the half containing the solution you are looking for, and you will find the page with Stevens on it in a matter of seconds, even with a large phonebook. You reduce your bounds by a factor of two each iteration, so the method converges *very* quickly!

We can now apply this method to root finding using a **while** loop with some simple boolean operators. First, we will determine which half of our bounds contains the solution. Then, we'll reduce our bounds by setting a new point for either the upper limit or lower limit to the midpoint of the bounds, and reiterate the selection process. After just a few iterations, we will narrow our bounds quickly to the point that we can estimate an answer. Let's start with a set of bounds that surround the root in question (and *only* the root in question!), [1 3]. An example code would look like so:

```
%% --- Define Bounds and Error --- %%
hi = 3; % Upper bound
low = 1; % Lower bound
```

```
epsilon = 2e-5; % Acceptable error for solution
counter = 0; % Counter (to avoid infinite loops)
limit = 1000; % Limit number of iterations to avoid infinite loops

%% --- Start loop --- %%
while (hi-low) > epsilon && counter < limit; % Cycle bound reduction
until solution or maxits reached
      mid = (hi + low) / 2; % Set midpoint halfway between lo and hi

      if (((hi*hi/4) - sin(hi))*((mid*mid/4) - sin(mid)) < 0) %
evaluate % F(x) at mid and hi, and multiply the results – a negative
result means % the function crosses an axis (a root!)
            low = mid;  % Eliminate lower half if it doesn't contain
the %root
      else
            hi = mid; % Eliminate upper half if it doesn't contain the
%root
      end

      counter = counter + 1; % Increase counter by 1 for each iteration
end

mid = (hi + low) / 2 % Set midpoint halfway between lo and hi one last
% time (to output solution)
```

Execute this script, and it will arrive at the answer 1.93375. If we plug this into the original equation, we get a very small error on the order of 10e-7, in only a millisecond and 17 iterations! To determine an appropriate number for maximum iterations, remember that your bound width is:

$$\frac{|(hi-low)|}{2^n}$$

where n is the number of iterations completed. To narrow Ayn Rand's book Atlas Shrugged (645,000 words, approximately 590 words per page – the third largest single-volume work in the English language) to the two words “to shrug” (page 422, Centennial Edition) would take only

$$2=\frac{645000}{2^n}\therefore n=\frac{\ln(322500)}{\ln(2)}\approx 19$$

iterations to narrow down to those words alone.

There are a few things to note about this method. First, there are several ways to select which half contains a solution, but perhaps the simplest is to multiply F(mid)*F(hi). If the solution lies in between mid and hi, the effect of the function crossing an axis will force the output to be negative (so we compare the product to 0). Second, if there are multiple roots inside the bounds, the sequence will converge on one of the original bounds (except in the case of an odd number of roots, in which case it will converge on only one root). For these reasons, you must know acceptable bounds before running the script to use this method, so it is somewhat limited.

Topic 9.1.2: The Newton-Raphson Method

The Newton-Raphson Method (or simply Newton's Method) is another so-called local method to determine the root of an equation. This method uses a single starting point (as opposed to the bounds required by the bisection method), and repeatedly uses a derivative to project a line to the axis of the root in question, as shown below. This results in a very fast convergence in many cases, but does require the solution of a derivative before the method can be used.

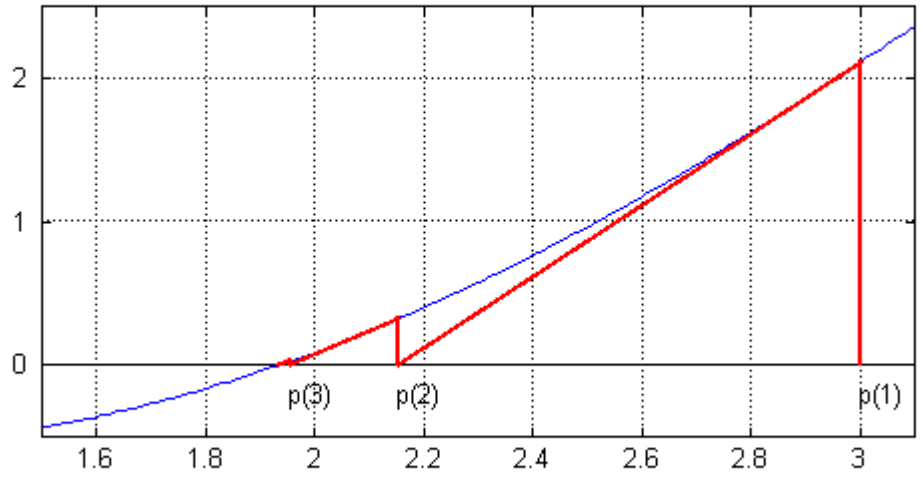


Figure 117: Image explaining the Newton-Raphson Method.

Here's how it works: First, you need to evaluate the function and its derivative. We'll start again with the same equations we used in the last section:

$$F(x) = \frac{x^2}{4} - \sin(x)$$

$$F'(x) = \frac{x}{2} - \cos(x)$$

Now, we take our initial guess, find the tangent line to our function F(x), and set our new guess to the x-intercept of that tangent line. The actual equation to do this is:

$$p_{i+1} = p_i - \frac{F(p_i)}{F'(p_i)}$$

where i is counting iterations, starting at 1. We continue this process until each new iteration of p

increases by an interval less than our acceptable error (i.e. $|p_{i+1} - p_i| < \text{error}$). Now, to do this in FreeMat, we'll use a while loop again, because we're not sure just how many iterations we will need. We'll also use a method called anonymous functions that allows you to pass a function as a variable. This will help condense our code, and allow for future modification to our program. First, we'll define our function f, and its derivative fp:

```
f = @(x) ((x.^2/4) - sin(x));
fp = @(x) (x/2 - cos(x));
```

Notice the decimal after x in the first relation – this ensures that the matrix elements are squared, instead of FreeMat trying to square the matrix, which is a different operation all together, and requires the matrix to have specific properties. Now, at the command window, if you type

```
--> f(0)
ans = 0
--> f(1)
ans = -.05915
```

We can see from the plot of our function on page 163 that these values are simply the evaluation of the function at those two points. So now, we can set our error and initial guess, and form a **while** statement to do our looping for us:

```
error = 2e-5; % Set error bound
p = 3; % Initial guess
i = 1; % Start counter at 1
p(i+1) = p(i) - f(p(i))/fp(p(i)); % Get initial value for p(2) for
while loop

while abs(p(i+1) - p(i)) > error
      i = i + 1;
      p(i+1) = p(i) - f(p(i))/fp(p(i));
end
```

Run this script, and it will complete after only 5 iterations, much faster than the bisection method! The way we did it here was by simply creating new elements in a p matrix at each iteration. In the end, we have:

```
--> p'
ans =
      3.00000000000000
      2.15305769201339
      1.95403864200580
      1.93397153275207
      1.93375378855763
      1.93375376282702
```

Notice that each iteration, we double the number of accurate digits after our first guess. The last iteration is accurate to all the decimal places shown in long format. If we plug p(6) into our original function, we should get something very close to 0:

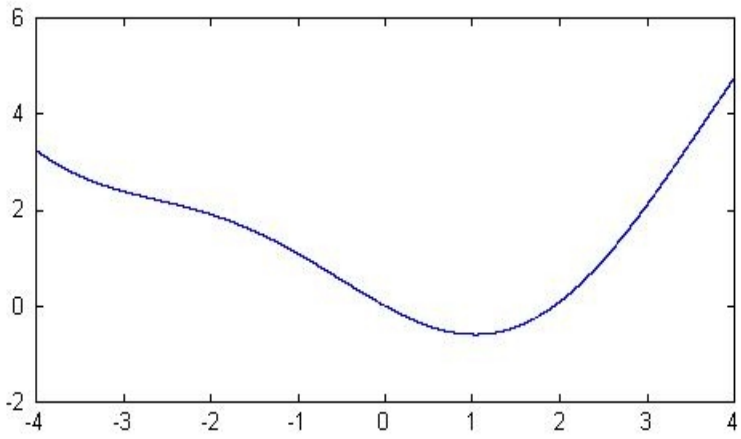
```
--> f(p(6))

ans =

   5.55111512312578e-016
```

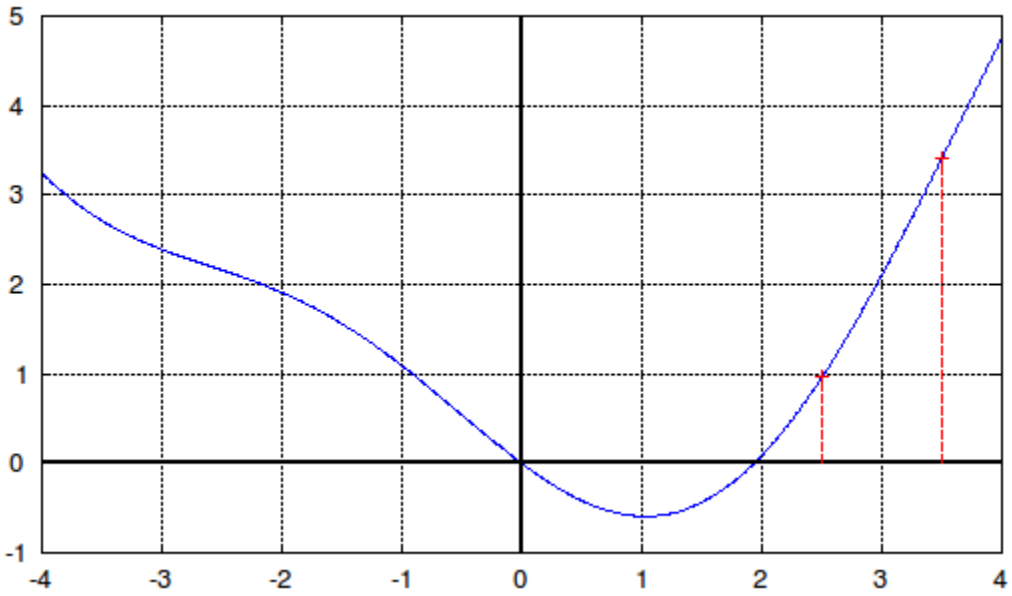
Our last guess was actually accurate on the level of 10e-16! Newton's method is very powerful, but you have to know the derivative beforehand. Also, if you pass a minimum or maximum in the function (or a spot where the derivative approaches 0), this method will quickly diverge without warning. To get around our derivative problems, we can estimate a derivative numerically and include some workarounds – convergence won't be as fast, but it will be easier in some cases, and a bit more reliable. This is called the Secant Method.

Topic 9.1.3: The Secant Method

The Secant Method performs the same task and works much in the same way as the Newton-Raphson Method, but without needing to know the derivative (and also without being subject to singularities with said analytical function). It differs on several key points: first, we start with two initial guesses at the root. It is important that these guesses be close to each other and near the root, otherwise we (as always) risk divergence. Second, instead of using the analytical derivative, we numerically calculate a secant line, saying that it's *close* to the derivative. By drawing this line between f(x) at our two guesses, the root of that line gives us our next guess for our root. As the method converges, our guess gets closer and closer to zero. Here are several images depicting what happens - given again our test function:

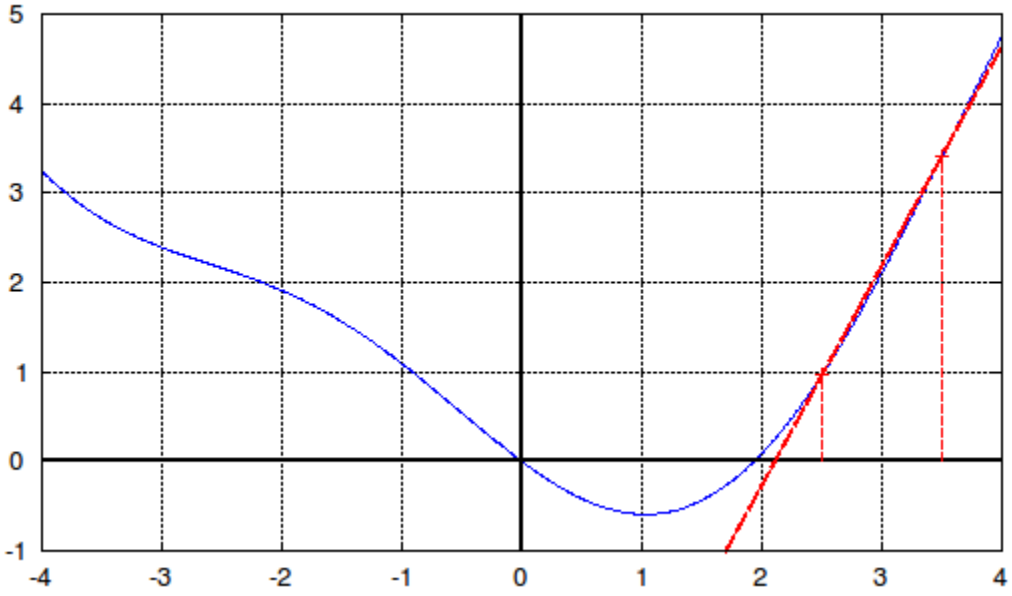
$$f(x)=\left(\frac{x^2}{4}\right)-\sin(x)$$





We pick two points, in this case we'll use 2.5 and 3.5.
Now, we'll draw a line between the two points and find its root using the following formula:

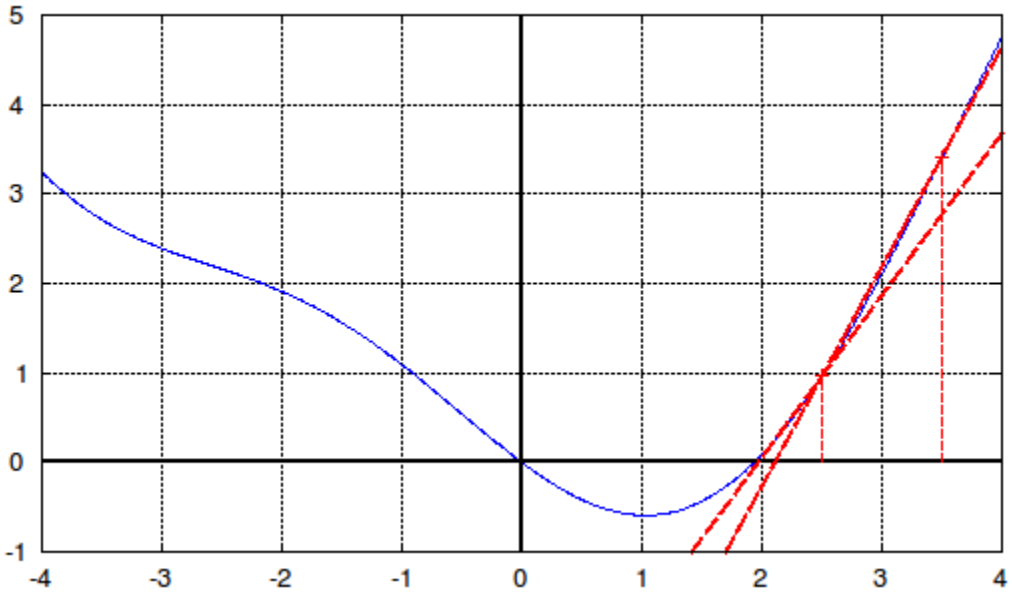
$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$



The x-intercept of our line turns out to be 2.1064. Let's check to see how close we are to zero:

$$\frac{2.1064^2}{4} - \sin(2.1064) = 0.2493$$

We're still a fair bit from the root, so now we'll use the points 2.5 and 2.1064 for our next iteration:



As we continue this process, we eventually find our result at 1.9337, our approximate root. So, how do we code this? Here's an example script:

```
%secantmethod.m script
x = [3.5; 2.5]; % x array of guesses
n = 2;

while( (x(n)^2)/4 - sin(x(n)) > 1e-5)
      x(n+1) = x(n) - ((x(n) - x(n-1))/(((x(n)^2)/4 - sin(x(n))) -
((x(n-1)^2)/4 - sin(x(n-1))))*((x(n)^2)/4 - sin(x(n)));
      n = n+1;
end
```

If we run this from the FreeMat command line, we have:

```
--> secantmethod
--> x
ans =
   3.50000000000000
   2.50000000000000
   2.10639961557374
   1.96913325845350
   1.93668028297996
   1.93380863593915
   1.93375384981807
```

Perhaps a more robust program would allow us to simply pass our function and two initial guesses as arguments:

```
function ret = secant(fx,x1,x2)

x = [x1; x2]; % initial guesses
n = 2; % initiate counter for loop

while(abs(fx(x(n)) ) > 1e-5) % precision of 1e-5
      x(n+1) = x(n) - ((x(n) - x(n-1))/(fx(x(n)) - fx(x(n-
1))))*fx(x(n));
      n = n+1;
end

ret = x(n); % return last value of x (our root)
```

Now, we can pass any function we like, with two initial guesses:

```
--> fx = @(x) (x.^5 - x - 1); % we can't solve this one analytically
--> secant(fx,0.9,1) % one of the roots is somewhere near 1

ans =
   1.16730389499850

--> fx(ans) % let's check our answer

ans =
 -6.89697854161508e-07
```

The secant method converges a bit slower than Newton's Method in most cases, but tends to be somewhat more stable, and is easier to embed into adaptive code that helps it avoid divergence.

Topic 9.2: Nonlinear Systems of Equations

Perhaps a more interesting (and more useful) application of root-finding is to solve *systems* of non-linear equations. In many areas of science, we want to model whole systems, which can in many cases prove difficult to solve analytically, so we can use the method described here to approximate some solutions.

Topic 9.2.1: The Newton-Raphson Method for Systems of Equations

Section 10.1.3 detailed Newton's Method to solve for the roots of a nonlinear equation. But what if we have a whole system of equations? Newton-Raphson also works for systems of equations, and is relatively simple (especially in FreeMat!) when we use a matrix approach. I'll skip some theory here, but by the same assumptions made in Newton's Method, we can arrive at the following equation:

$$\begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \frac{\delta f_1}{\delta x_2} \\ \frac{\delta f_2}{\delta x_1} & \frac{\delta f_2}{\delta x_2} \end{bmatrix} * \begin{bmatrix} \Delta x_1 \\ \Delta x2 \end{bmatrix} = - \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

The first matrix seen here is called the Jacobian Matrix, simply a matrix with values for partial derivatives corresponding to their location inside the matrix. If you look at this equation, it has the classic Ax = b appearance, which means we can simply use A\b to solve for Δx. Given that the only unknowns in this equation are Δx_1 and Δx_2, we start with a guess (just as in Newton's Method), and use Gaussian Elimination to solve for these values. We then get our new values for x_1 and x_2 from our original x_1 and x_2 as follows:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{new} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{old} + \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

Straight away, we can make a simple program to evaluate this method for a system of two equations with two unknowns. Let's just put the equations and their derivatives right into the code and make a script .m file:

```
% newtsys.m script file

%% --- Two Simple Test Functions --- %%
f1 = @(x,y) (x^2 - 4*(y^3));
f2 = @(x,y) (sin(x) + 3*cos(3*y));

%% --- Their Derivatives --- %%
df1dx = @(x,y) (2*x);
df2dx = @(x,y) (cos(x));
df1dy = @(x,y) (-12*y^2);
df2dy = @(x,y) (-9*sin(3*y));

xy = [1;1]; % Initial Guess at Solution

deltax = [10; 10]; % starter value for delta x matrix

counter = 0; % initialize counter variable

%% --- Jacobian Matrix --- %%
J = @(x,y) [df1dx(x,y) df1dy(x,y);df2dx(x,y) df2dy(x,y)];

%% --- Function Matrix --- %%
F = @(x,y) [f1(x,y);f2(x,y)];

%% --- Iterative Technique --- %%
while abs(max(deltax)) > 1e-5
      deltax = J(xy(1),xy(2))\F(xy(1),xy(2)); %Gaussian Elim. Step
      xy = xy - deltax; % modify [x;y] per iterative technique
      counter = counter + 1; % count number of iterations
end
```

```
printf('%3d iterations\n',counter);
printf('x = %3f\t y = %3f\n',xy(1),xy(2));
```

If we run this script, we've set up a matrix J(x,y) which contains anonymous functions, each also a function of (x,y) and a matrix F(x,y) with similar properties. This way, we can simply put in our guess for x and y by evaluating J(x,y) and F(x,y). In this case, x is xy(1) and y is xy(2). We do our iterative technique by Gaussian Elimination using the backslash operator (A\b or in this case, J(x,y)\F(x,y)), and the resulting output is our delta matrix containing Δx and Δy. We simply loop these two commands, and we eventually reach a solution:

```
--> newtsys

  6 iterations

x = 4.339563       y = 1.676013
```

If we evaluate f_1 and f_2 at these coordinates, we should get something very close to zero:

```
--> f1(4.339563,1.676013)

ans =

 -7.1054e-15

--> f2(4.339563,1.676013)

ans =

 2.6645e-15
```

So, it worked! This technique can be easily extrapolated to systems of many equations and many unknowns, with two caveats: all partial derivatives have to be known (or numerically approximated), and in such systems, many times there are multiple solutions. This method can diverge, and the solutions reached are usually heavily dependent on your initial guesses, so it's necessary to have an idea of what your functions look like in the first place. You've been warned.

Now, let's look at a real-world example, along with a program that allows us some input and manipulation. Kinematic analysis of the four bar mechanism shown below results in the following system of equations:

$$F_1 = r_2\cos(\theta_2) + r_3\cos(\theta_3) + r_1\cos(\theta_1) + r_4\cos(\theta_4) = 0$$
$$F_2 = r_2\sin(\theta_2) + r_3\sin(\theta_3) + r_1\sin(\theta_1) + r_4\sin(\theta_4) = 0$$

Here, all radii (linkage lengths) are known, as are θ_1 and θ_4, so we really just have two functions, F_1 and F_2 of two variables, θ_2 and θ_3. For the Jacobian, we take the partial derivatives as shown previously:

$$\begin{bmatrix} \frac{\delta f_1}{\delta \theta_2} & \frac{\delta f_1}{\delta \theta_3} \\ \frac{\delta f_2}{\delta \theta_2} & \frac{\delta f_2}{\delta \theta_3} \end{bmatrix} = \begin{bmatrix} -r_2 \sin(\theta_2) & -r_3 \sin(\theta_3) \\ r_2 \cos(\theta_2) & r_3 \cos(\theta_3) \end{bmatrix}$$

Now, if we have an initial guess at θ_2 and θ_3, we can plug them into our final iterative technique, and get new values! Let's start by coding our functions and derivatives as anonymous functions (functions we can enter right from the command console, and pass around like variables).

Topic 9.3: Numerical Differentiation and Integration

Calculus consists of two parts. You're either summing things up (integration) or you're figuring out how quickly things are changing (differentiation). All of the calculus classes I took focused on proper and improper integrals that had closed-form solutions. A closed-form solution means one that leads to a definite answer, as opposed to another integral or to an infinite sum such as a Taylor series. The problem is that not all integrals have closed-form solutions. This is where numeric integration comes into play.

The other issue is differentiation, calculating the rate of change of some quantity. We'll look at differentiation on a theoretical level, as well as differentiation applied to a practical problem. The great thing is that the practical problem involves a ride at Six Flags America!

Topic 9.3.1: Numerical Integration

Numerical integration provides the ability to calculate solutions for definite integrals, meaning integrals that have numeric limits on the integration. While some integrals are easier (and better) to do by actually calculating closed form solutions, there are some functions for which an antiderivative does not exist. For example, the function for a Gaussian distribution, when integrated, does not have an antiderivative.

$$\int_{-1}^{2} \frac{1}{\sigma_x \sqrt{2\pi}} e^{\frac{-x^2}{2\sigma_x^2}} dx$$

If we ignore the constants, this integral boils down to $\int e^{-x^2} dx$. There's no antiderivative for $f(x)=e^{-x^2}$. Therefore, in order to have a numerical answer to this integral, we must use numerical integration.

Look at what an integral is doing. Its the sum of some function. The reason people have a problem with it is that it is an *infinite* sum, as in adding up an infinite number of parts. How is that possible? I'd have to point you to one of several texts on calculus. To understand it from the computer's point of view, look at this integral:

$$\int_{1}^{3} x^2 dx$$

If we create a graph of this equation, we'd get something such as this.

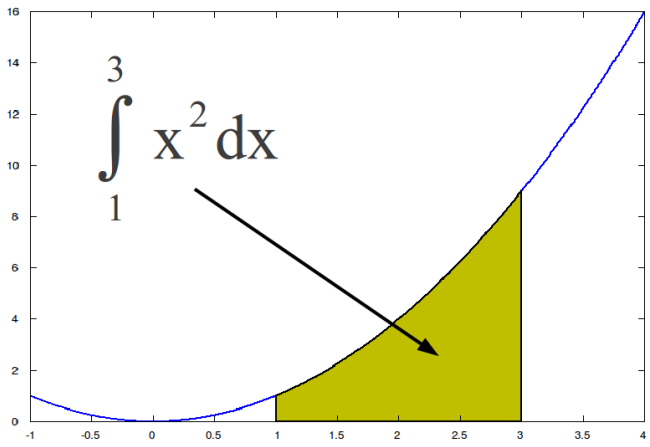


Figure 118: Graph of x^2 with integration from 1 to 3.

The fact that an integral consists of an infinite sum means that it requires some adjustment to work on a computer. That's because a computer cannot directly perform an infinite sum; that would take forever. Instead, *what we'll do with numerical integration is to calculate enough points to get an approximate answer.* If you're wondering how many points that is, that's based on the desired precision. The greater the desired precision, the greater the number of points needed.

So why use numerical integration? A big reason is that there are some equations that cannot be solved in closed form. And what's closed form? you ask. Closed form means you wind up with an answer that does not require adding up an infinite number of terms. For example, you can solve the equation x-5=2 in closed form. By simply rearranging the equation, you wind up with x=7. What of the previous equation:

$\int_1^3 x^2 dx$? This equation also has a closed form solution. If you actually perform the integration, you'll find that it equals 8/3. But what about this mess of an equation:

$f(x)=e^{-x^2}+2.5e^{-(x-3)^2}$ This is a curve created by two, different exponentials. The curve is shown in Figure 119.

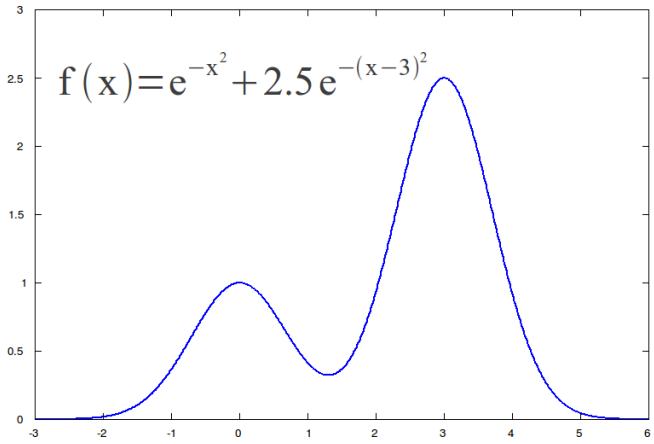


Figure 119: Graph created using two different exponentials.

Now what if we want to know the area under the curve from -1 to 2, as shown in this integral:

$$\int_{-1}^{2} e^{-x^2}+2.5e^{-(x-3)^2}=?$$

Don't panic. All the equation above does is calculate the area under this curve from -1 up to 2. The area that we want to calculate is shown in Figure 120.

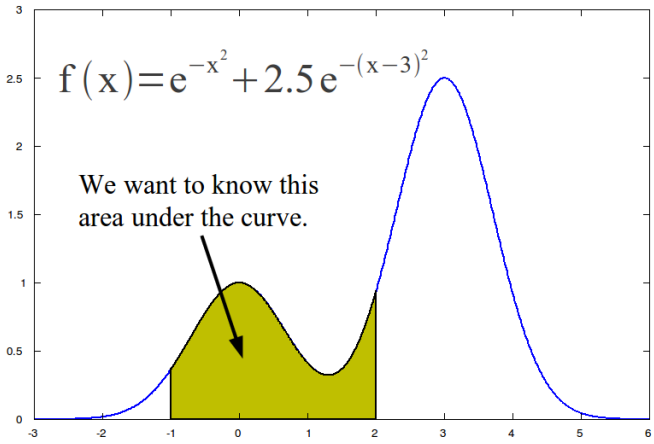


Figure 120: Graph of Gaussian curve showing example area to be calculated using numerical integration.

This is an example of an equation for which there is no closed form solution. The only way to handle this type of equation is through numerical integration. Thanks to computers and software such as Freemat, that's much easier than if you happened to be using the item shown in Figure 121.

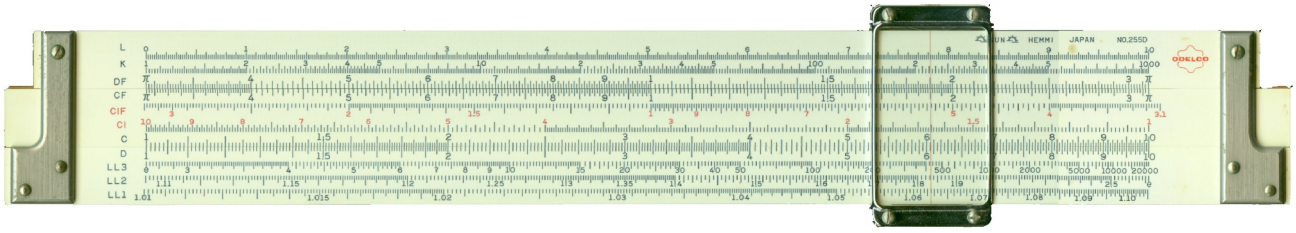


Figure 121: Hey, it's a Hemmi!

We're going to use this equation to demonstrate some some of the methods of numerical integration. Okay, back to the equation. It is:

$$f(x)=e^{-x^2}+2.5e^{-(x-3)^2}$$

Using Freemat, we can create a plot of this function, shown in Figure 122.

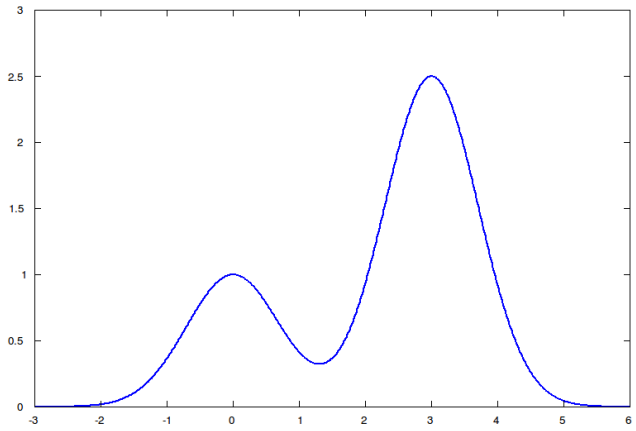


Figure 122: Plot of the double exponential function.

This was created using this small bit of code:

```
x=-3:0.01:6;
y=exp(-x.^2)+2.5*exp(-(x-3).^2);
plot(x,y,'b-','linewidth',2);
```

From this, we'll explain and demonstrate several methods for calculating the area underneath the curve. In general, there are many methods that can be used to calculate the area under a curve. We'll look at five of them. The ones we will cover here are the rectangular method, the trapezoidal method, Simpson's rule, Gaussian quadrature, and the Monte Carlo method. We'll go into a bit of detail about each one.

Topic 9.3.1.1 Rectangular Method

One way to add up the area is with the a bunch of rectangles. Just place a bunch of equal-width rectangles under the area. The rectangles can either touch on one corner (the so-called top-left corner approximation and the top-right corner approximation) or in the center of the rectangle (the midpoint approximation). We'll cover the midpoint approximation because, in my limited experience, it tends to provide the desired precision with fewer calculations than if a corner approximation is used.
Each rectangle stretches so that its top, center touches the curve and the bottom touches the bottom, which in this case is zero. This is shown in Figure 123.

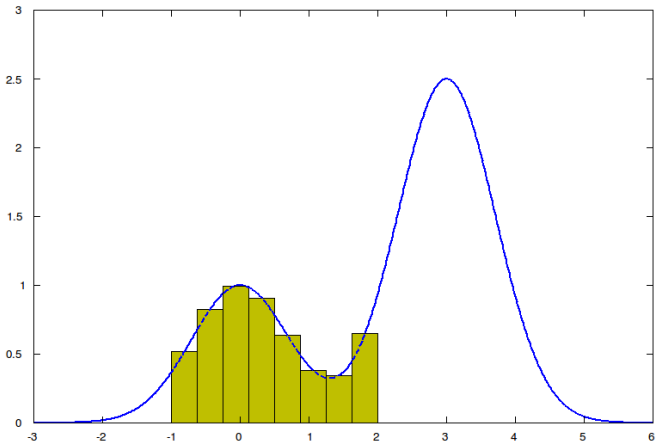


Figure 123: Example of calculating the area from -1 to 2 using 8 rectangular areas.

By adding up the areas of each rectangle, we can calculate the approximate area under the curve. If we look at each rectangle as having a width *w* and a height *h* equal to the distance from zero to where its center touches the graph, we'll have something as shown in Figure 124. The area of each rectangle is *hw*. We get the total area by summing the area of each rectangle.

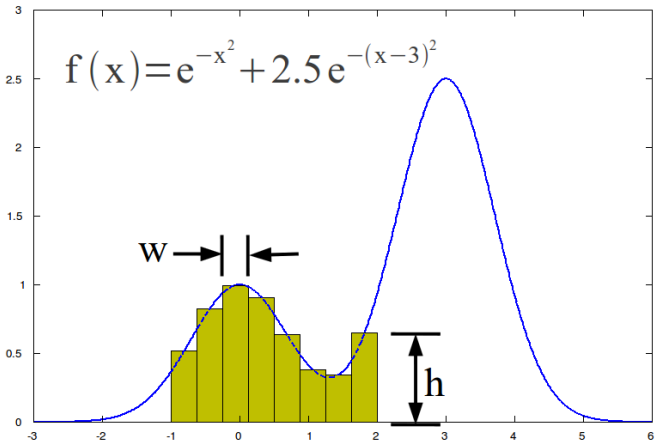


Figure 124: Exponential curve with the height and width of the rectangles annotated.

We'll now use this this graph and some Freemat code to add up the area of all of the rectangles. How many rectangles? Good question! It boils down to how little error we want. As we increase the number of sections (rectangles), the amount of error decreases. This is shown in Table 1. As the error decreases, we gain better accuracy as to the "actual" value.

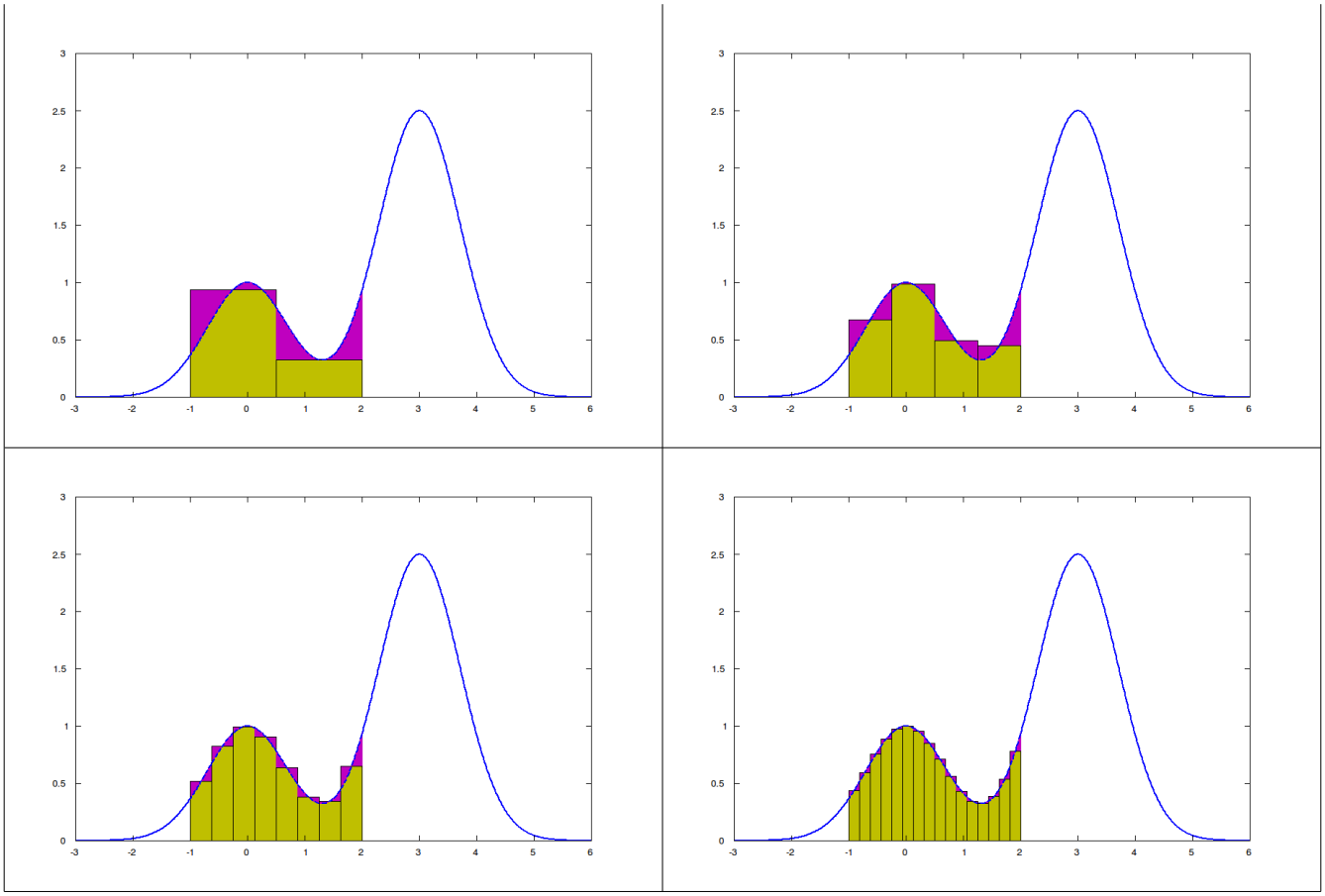


Table 1: These images show the amount of error (colored in magenta) due to the rectangular method of numerical integration. Note that as the number of rectangles increases, the amount of error decreases.

Think of "accuracy" in this sense as "the number of digits". The more accuracy (digits), the more rectangles we require. We'll create a function that we can provide the equation of the curve, the start and end points, and the number of rectangles to use to calculate the overall area.

```
% Script to calculate the specific area under a function using the
% rectangular method.
eqn=@(x) (exp(-x.^2)+2.5*exp(-(x-3).^2));
sects=8; % This determines how many sections or rectangles we'll use
         % to calculate the overall area.
start=-1; % This is the point at which we start summing the area.
stop=2; % This is the end point at which we sum the area.
w=(stop-start)/sects; % calculate the width of the rectangles.
x=(start+w/2):w:stop; % This creates a vector the center points
                      % of the rectangles along the x-axis.
area=eqn(x)*w; % Calculate the area of each rectangle.  The part
               % "eqn(x)" is the height, h, of each rectangle.
totalArea=sum(area); % Calculate the final area
```

Saving this script as **rectArea.m**, then running it, we get the following result.

```
--> rectArea
totalArea =
    1.9713
```

The calculated area is 1.9713. But is this correct? Another good question! One way to find out is to

test it using a function with a known answer. For example, we can use the integral $\int_0^2 x^2 dx$. This function x^2 has the antiderivative of $x^3/3$. The solution for $\int_0^2 x^2 dx$ becomes 8/3 = 2.667. Using this knowledge and changing the previous script for this function, we get:

```
% Script to calculate the specific area under a function using the
% rectangular method.
eqn=@(x) (x.^2);
sects=2; % This determines how many sections or rectangles we'll use
         % to calculate the overall area.
start=0; % This is the point at which we start summing the area.
stop=2; % This is the end point at which we sum the area.
w=(stop-start)/sects; % calculate the width of the rectangles.
x=(start+w/2):w:stop; % This creates a vector the center points
                      % of the rectangles along the x-axis.
area=eqn(x)*w; % Calculate the area of each rectangle.  The part
               % "eqn(x)" is the height, h, of each rectangle.
totalArea=sum(area); % Calculate the final area
```

Running this script, we get:

```
--> rectArea
totalArea =
    2.5000
```

This is not the correct answer. It's off by 2.6667 - 2.5 = 0.1667. Let's increase the number of sections.

```
% Script to calculate the specific area under a function using the
% rectangular method.
eqn=@(x) (x.^2);
sects=4; % This determines how many sections or rectangles we'll use
         % to calculate the overall area.
start=0; % This is the point at which we start summing the area.
stop=2; % This is the end point at which we sum the area.
w=(stop-start)/sects; % calculate the width of the rectangles.
x=(start+w/2):w:stop; % This creates a vector the center points
                      % of the rectangles along the x-axis.
area=eqn(x)*w; % Calculate the area of each rectangle.  The part
               % "eqn(x)" is the height, h, of each rectangle.
totalArea=sum(area); % Calculate the final area
```

Re-running the script, we get:

```
--> rectArea
totalArea =
    2.6250
```

As stated before, as the number of sections increases, the amount of error decreases. This means that we wind up with greater accuracy. The following table demonstrates this graphically.

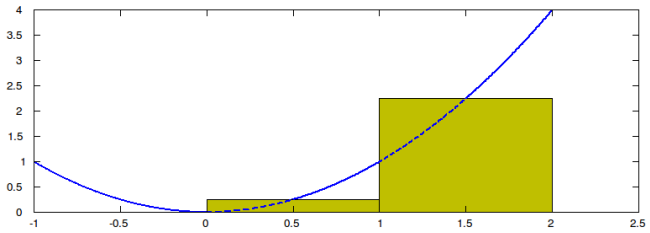
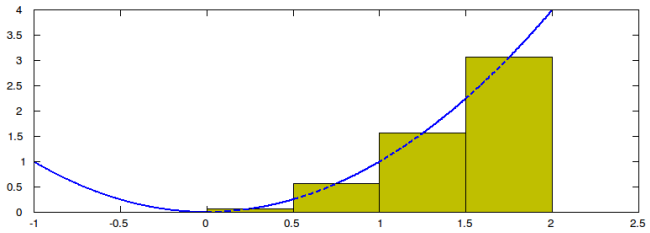
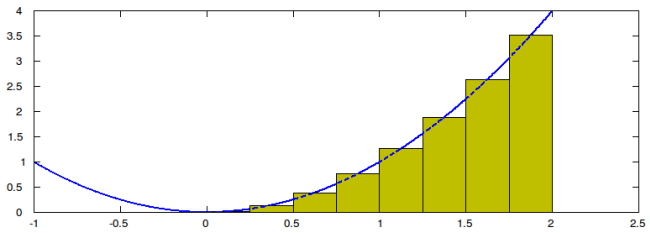
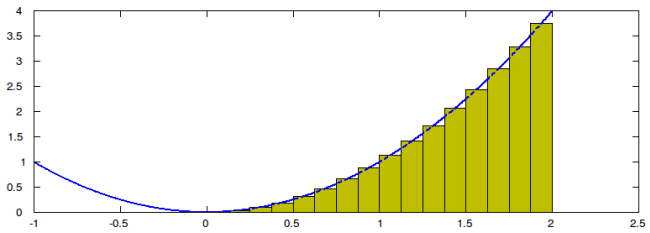
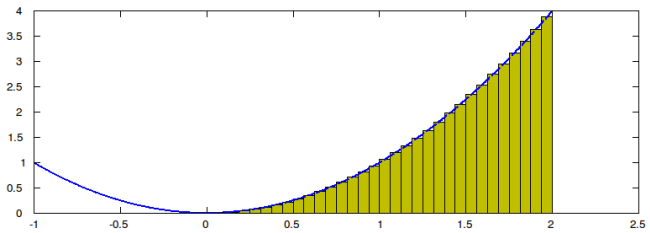
	Number of sections: 2 Calculated area: 2.5 Error: 0.1667
	Number of sections: 4 Calculated area: 2.625 Error: 0.0417
	Number of sections: 8 Calculated area: 2.6562 Error: 0.0105
	Number of sections: 16 Calculated area: 2.6641 Error: 0.0026
	Number of sections: 32 Calculated area: 2.6660 Error: 0.0004

Table 2: These images show the rectangular method for calculating $\int_0^2 x^2 dx$. *The right-most column shows the number of sections for each iteration, the calculated area for that number of sections, and the error between the calculated area and the actual area as found when solving the integral.*

Going back to our previous equation of $e^{-x^2} + 2.5\,e^{-(x-3)^2}$, let's double the number of rectangles to 16. The graph of this would appear as shown in Figure 125.

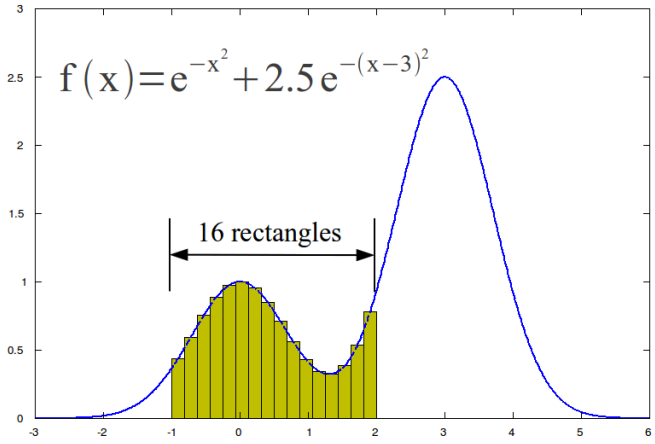


Figure 125: This shows the area to be integrated as it appears using the 16 rectangles as part of the rectangular method.

The code will appear as follows (with the changed part in a bold font):

```
% Script to calculate the specific area under a function using the
% rectangular method.
eqn=@(x) (exp(-x.^2)+2.5*exp(-(x-3).^2));
sects=16; % This determines how many sections or rectangles we'll use
          % to calculate the overall area.
start=-1; % This is the point at which we start summing the area.
stop=2; % This is the end point at which we sum the area.
w=(stop-start)/sects; % calculate the width of the rectangles.
x=(start+w/2):w:stop; % This creates a vector the center points
                      % of the rectangles along the x-axis.
area=eqn(x)*w; % Calculate the area of each rectangle.  The part
               % "eqn(x)" is the height, h, of each rectangle.
totalArea=sum(area); % Calculate the final area
```

Re-running the routine, we get:

```
--> rectArea
totalArea =
    1.9759
```

The area has gone up to 1.9759. Again, how do we know if this is correct? The problem is that we don't have a way to calculate an exact answer. But if we increase the number of sections, the amount of error will decrease. Eventually we'll get an answer that's close enough. But what is *close enough*? That's up to you to decide.

The great thing is that, once you decide on a particular accuracy, you can set up Freemat to run until it has reached that threshold. The way we'll do it here is by creating a loop and doubling the number of sections with each iteration. We'll start with an accuracy of 10^{-2} just to demonstrate the principle. To make the decision when this has been reached, we'll use a **while** loop, as shown below. Note that I've added several **printf** functions to provide feedback on the calculated area for each iteration. You would not want the printf commands in an actual function; again, I'm just using them here so that you can see what's going on with this code as it goes through its iterations.

```
% Script to calculate the specific area under a function using the
% rectangular method.
% The next line is the anonymous function containing the equation
% to be integrated.
```

```
eqn=@(x) (exp(-x.^2)+2.5*exp(-(x-3).^2));

% This first section calculates the initial value using 2 sections.

epsilon=1e-2; % Set the desired accuracy.
printf('The desired accuracy is %s\n',num2str(epsilon));
sects=2; % This determines how many sections or rectangles we'll use
         % to calculate the overall area.
start=-1; % This is the point at which we start summing the area.
stop=2; % This is the end point at which we sum the area.
w=(stop-start)/sects; % calculate the width of the rectangles.
x=(start+w/2):w:stop; % This creates a vector the center points
                      % of the rectangles along the x-axis.
area=eqn(x)*w; % Calculate the area of each rectangle. The part
               % "eqn(x)" is the height, h, of each rectangle.
totalArea=sum(area);
totalDiff=totalArea; % Difference between the consecutive calculations.
lastArea=totalArea; % This is used to calculate the differences
                    % between consecutive loops.
printf('With %i sections, the total area is %f\n',sects,totalArea);

% Run a while loop to determine when the accuracy has been met.

while (totalDiff>(totalArea*epsilon));
   sects=sects*2; % Double the number of sections with each loop.
   w=(stop-start)/sects; % calculate the width of the rectangles.
   x=(start+w/2):w:stop; % This creates a vector the center points
                         % of the rectangles along the x-axis.
   area=eqn(x)*w; % Calculate area of each rectangle.
   totalArea=sum(area); % Create a vector of cumulative sum of
                         % rectangular areas.
   totalDiff=abs(totalArea-lastArea);
   lastArea=totalArea;
   printf('With %i sections, the total area is %f\n',sects,totalArea);
end
printf('The calculated area from %f to %f is
%f\n',start,stop,totalArea);
```

Saving this script as **rectAreaPrec.m**, when we run it, we get:

```
--> rectAreaPrec
The desired accuracy is 0.01
With 2 sections, the total area is 1.899024
With 4 sections, the total area is 1.952193
With 8 sections, the total area is 1.971305
The calculated area from -1.000000 to 2.000000 is 1.971305
```

Next, we'll increase the accuracy from 10^{-2} to 10^{-3} to see how big of a change it makes. Again, I'm not going to display all of the code; instead, just change the line that sets the variable epsilon to:

```
epsilon=1e-3; % Set the desired accuracy.
```

Running the script, we get:

```
--> rectAreaPrec
The desired accuracy is 0.001
With 2 sections, the total area is 1.899024
With 4 sections, the total area is 1.952193
With 8 sections, the total area is 1.971305
With 16 sections, the total area is 1.975899
```

```
With 32 sections, the total area is 1.977035
The calculated area from -1.000000 to 2.000000 is 1.977035
```

The total area went up from 1.971305 to 1.977035. Note that the number of steps went up by a factor of 4. Let's next try an accuracy of precision of 10^{-6}.

```
epsilon=1e-6; % Set the desired accuracy.
```

Now, let's run the script again.

```
--> rectAreaPrec
The desired accuracy is 1e-06
With 2 sections, the total area is 1.899024
With 4 sections, the total area is 1.952193
With 8 sections, the total area is 1.971305
With 16 sections, the total area is 1.975899
With 32 sections, the total area is 1.977035
With 64 sections, the total area is 1.977318
With 128 sections, the total area is 1.977389
With 256 sections, the total area is 1.977407
With 512 sections, the total area is 1.977411
With 1024 sections, the total area is 1.977412
The calculated area from -1.000000 to 2.000000 is 1.977412
```

That took a few more sections. Is there a way to improve the code such that it doesn't take so many computations? Yes. That's where the trapezoid, Simpson's Rule, and the Gaussian quadrature method come into play.

But before we jump into those methods, let's finish this up by making this into a function that we can use for general purposes. The function I'm going to create uses four things as inputs: the equation to be integrated, the start point, the stop point, and a desired accuracy.

```
% Function to calculate the area under a curve using the rectangle
% method.
%
% The function will input the following variables:
% - eqn: This will be the equation which will be numerically
% integrated.
% It will be entered as an anonymous function.
% - start: This will be the start of the area to be integrated.
% - stop: This will be the end of the area to be integrated.
% - epsilon: This is the desired accuracy for the numerical
%   integration.

function finalValue=rectz(eqn,start,stop,epsilon)

sects=2;
w=(stop-start)/sects; % calculate the width of the rectangles.
x=(start+w/2):w:stop; % calculate the midpoints of each section.
area=eqn(x)*w; % calculate the height of each rectangle.
totalArea=sum(area); % sum up each section.
lastArea=totalArea; % this variable is used to determine when the
                    % desired accuracy has been reached.
diffErr=totalArea;

while(diffErr>(epsilon*totalArea))

sects=sects*2; % double the number of sections with each iteration.
w=(stop-start)/sects;
x=(start+w/2):w:stop;
area=eqn(x)*w;
```

```
totalArea=sum(area);
diffErr=abs(lastArea-totalArea); % calculate the difference between the
                                 % consecutive values.
lastArea=totalArea;

end

finalValue=totalArea; % read out the final calculated value
```

I saved this with the filename of **rectz.m**. Trying this out with a desired accuracy of 10-4, we get:

```
--> eqn=@(x) (exp(-x.^2)+2.5*exp(-(x-3).^2));
--> rectz(eqn,-1,2,1e-4)
ans =
    1.9774
```

Topic 9.3.1.2 Trapezoid Rule

The idea behind the trapezoid rule is to provide a shape that provides a better "fit" than the rectangular section. The concept is to use trapezoids as opposed to rectangles. In the rectangular method, only one point of each section touches the graph. In the trapezoidal method, both sides of each section touch the graph creating a piecewise linear approximation of the original graph. This is shown in Figure 126.

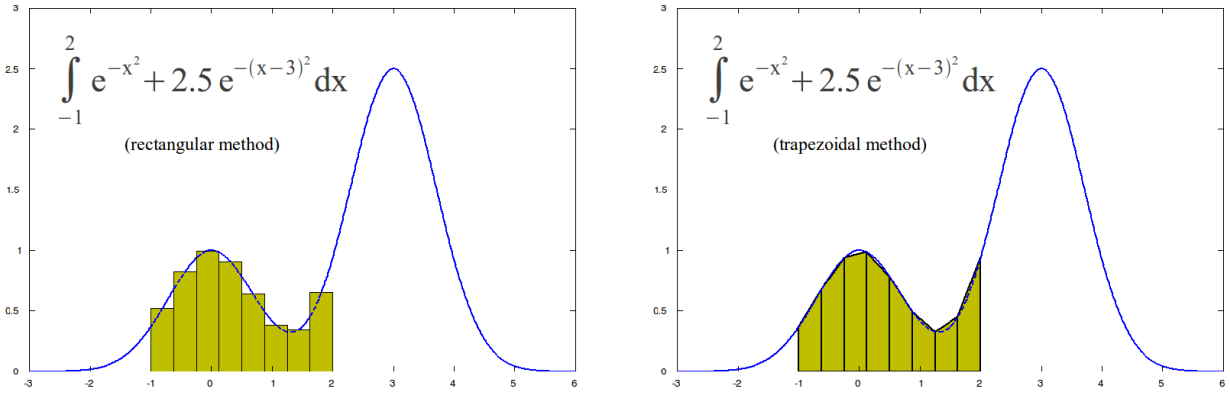


Figure 126: This is a comparison of the rectangular and trapezoidal methods of numerical integration. Note how the trapezoidal method follows the contour of the graph.

To calculate the area of each section, it's necessary to calculate the area of a trapezoid. This is shown in Figure 127.

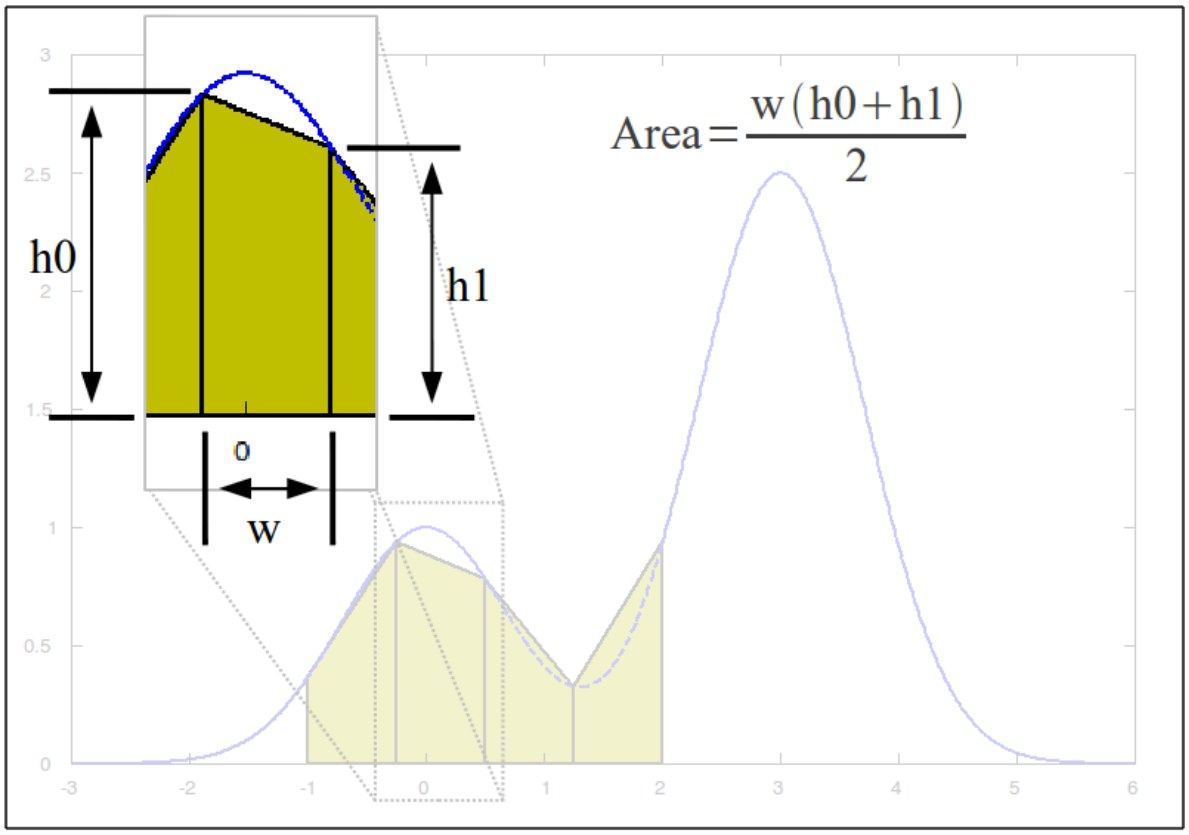


Figure 127: Calculation of the area of a trapezoid for numerical integration.

Rather than go through the steps from the previous section, we'll just modify the rectangular method function (saved as rectz.m) for the trapezoidal method. The changes involve the calculation for the area of each section. I've changed the calculation from that of a rectangle to that of a trapezoid. I did use a minor "trick" in the calculation in which I used the *circshift* function (*FD, p. 269*) rather than calculating each section independently. I saved this function as **trapz.m**.

```
% Function to calculate the area under a curve using the trapezoid
% method.
%
% The function will input the following variables:
% - eqn: This will be the equation which will be numerically
%        integrated.
% It will be entered as an anonymous function.
% - start: This will be the start of the area to be integrated.
% - stop: This will be the end of the area to be integrated.
% - epsilon: This is the desired accuracy for the numerical
%            integration.

function finalValue=trapz(eqn,start,stop,epsilon)

sects=2;
w=(stop-start)/sects; % calculate the width of the rectangles.
x=start:w:stop; % calculate the edge points of each section.
h0=eqn(x); % calculate the height of both edges of each section.
h1=circshift(h0,[0,-1]); % rotate the vector to allow adding
```

```
                          % consecutive parts to calculate the area.
area=(h0+h1)*w/2; % calculate the area of a trapezoid
totalArea=sum(area); % sum up each section.
totalArea=areaSum(length(x)-1); % read out the accumulated sum.
lastArea=totalArea; % this variable is used to determine when the
                    % desired accuracy has been reached.
diffErr=totalArea;

while(diffErr>(epsilon*totalArea))

sects=sects*2; % double the number of sections with each iteration.
w=(stop-start)/sects;
x=start:w:stop;
h0=eqn(x);
h1=circshift(h0,[0,-1]);
area=(h0+h1)*w/2;
areaSum=cumsum(area);
totalArea=areaSum(length(x)-1);
diffErr=abs(lastArea-totalArea); % calculate the difference between the
                                 % consecutive values.
lastArea=totalArea;

end

finalValue=totalArea; % read out the final calculated value
```

Testing this new function with the previous equation, we get this:

```
--> eqn=@(x) (exp(-x.^2)+2.5*exp(-(x-3).^2));
--> trapz(eqn,-1,2,1e-4)
ans =
    1.9775
```

Topic 9.3.1.3 Simpson's Rule

The idea behind Simpson's rule is that rather than using trapezoids to form a piecewise linear approximation of the graph to instead use an equation of the form $f(x)=ax^2+bx+c$. This equation creates a parabola. Hence, each section will be a piecewise parabolic shape that follows the graph. This is shown in Figure 128.

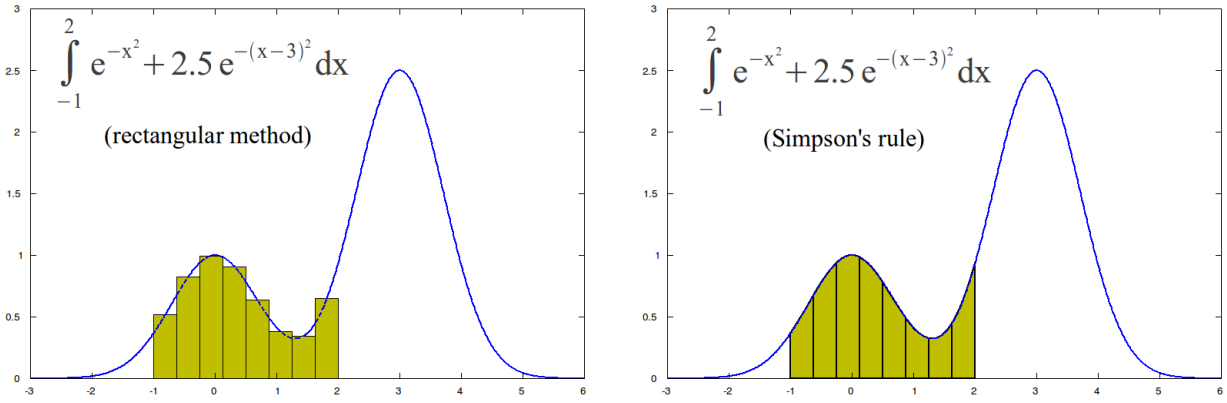


Figure 128: Comparison of rectangular method and Simpson's rule.

The area for each section is actually a straightforward calculation, as shown in Figure 129. (The

derivation of this calculation is another matter.)

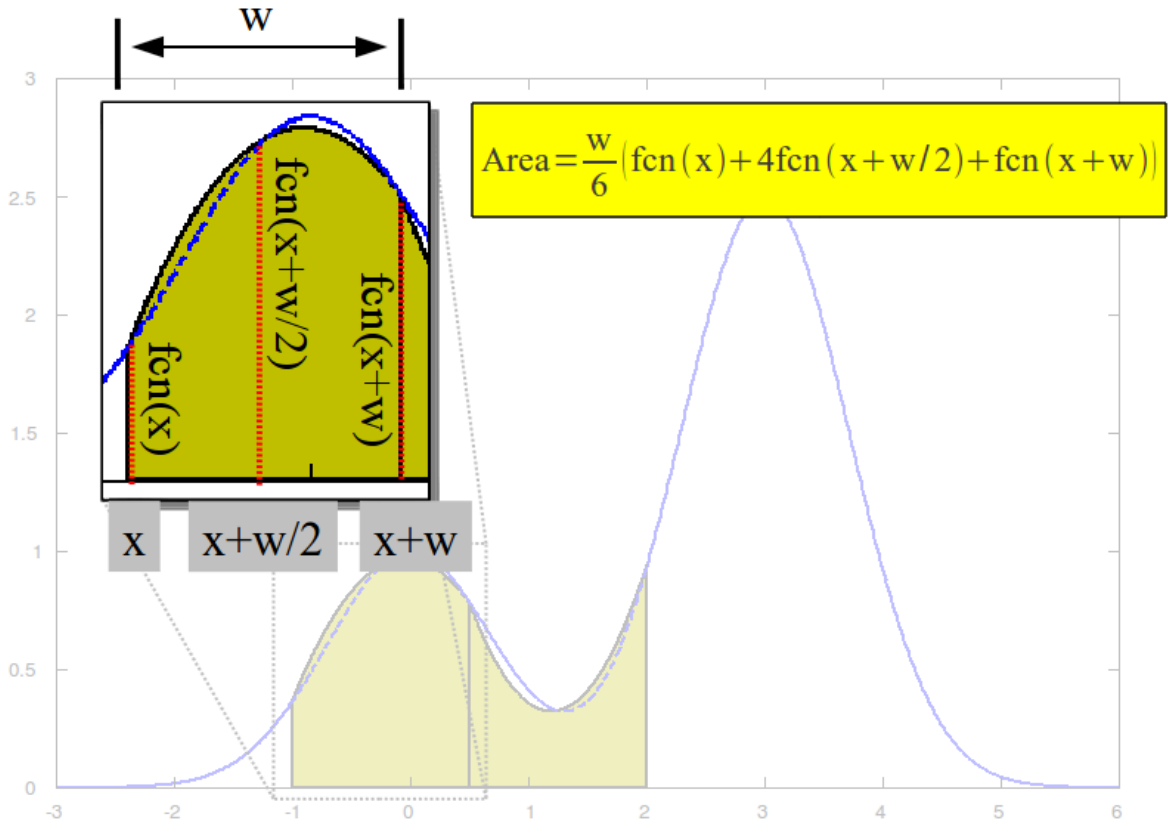


Figure 129: Calculation of the area for each section using Simpson's rule.

Once again, we'll use the original function **rectz.m** and modify it for the area according to Simpson's rule.

```
% Function to calculate the area under a curve using Simpsons rule.
%
% The function will input the following variables:
% - eqn: This will be the equation which will be numerically
%        integrated.
% It will be entered as an anonymous function.
% - start: This will be the start of the area to be integrated.
% - stop: This will be the end of the area to be integrated.
% - epsilon: This is the desired accuracy for the numerical
%            integration.

function finalValue=quadv(eqn,start,stop,epsilon)

sects=2;
w=(stop-start)/sects; % calculate the width of each section.
x=start:w:stop; % calculate the endpoints of each section.
xmid=(start+w/2):w:stop; % calculate the midpoint of section.
h0=eqn(x); % calculate the height of the beginning of each section.
hmid=[eqn(xmid) 0]; % calculate the height of the midpoint of each
                    % section.
h1=circshift(h0,[0,-1]); % rotate the h0 vector to create another
```

```
                          % vector of the endpoint heights of each
                          % section.
area=(w/6)*(h0+4*hmid+h1); % calculate the area of each section
                            % according to Simpsons rule.
areaSum=cumsum(area); % sum up each section.
totalArea=areaSum(length(x)-1); % read out the accumulated sum.
lastArea=totalArea; % this variable is used to determine when the
                    % desired accuracy has been reached.
diffErr=totalArea;

while(diffErr>(epsilon*totalArea))

sects=sects*2; % double the number of sections with each iteration.
w=(stop-start)/sects;
x=start:w:stop; % calculate the endpoints of each section.
xmid=(start+w/2):w:stop; % calculate the midpoint of section.
h0=eqn(x);
hmid=[eqn(xmid) 0];
h1=circshift(h0,[0,-1]);
area=(w/6)*(h0+4*hmid+h1);
areaSum=cumsum(area);
totalArea=areaSum(length(x)-1);
diffErr=abs(lastArea-totalArea); % calculate the difference between the
                                 % consecutive values.
lastArea=totalArea;

end

finalValue=totalArea; % read out the final calculated value
```

Checking this with the previous routines, we get:

```
--> quadv(eqn,-1,2,1e-4)
ans =
    1.9774
```

This is the same as the **rectz** and **trapz** routines (used with the rectangular method and the trapezoidal method, respectively).

Topic 9.3.1.4: Gauss-Legendre Method

The concept of the Gaussian quadrature rule is to use "a weighted sum of function values at specified points within the domain of integration."[3] In other words, this uses some high-level math to arrive at an answer for a definite integral. In mathematical terms, we're going to use this equation:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right)$$

What's happening here is that we're calculating the value of our function at several different points, weighting those values, then summing them up. Note that on the right-hand side, the value of x_i is determined by the roots of a Legendre polynomial; the values used from the limits of the integral are from the part (a+b)/2, which is the midpoint of each section calculated. Clear as mud?

Rather than calculating the roots of the polynomial, we're going to used already-calculated values for a 5-point polynomial[4]. These are:

3 From Wikipedia article "Gaussian quadrature", http://en.wikipedia.org/wiki/Gaussian_quadrature, pulled 19 June 2011.

4 From Wikipedia article "Gaussian quadrature", http://en.wikipedia.org/wiki/Gaussian_quadrature, pulled 19 June 2011

n	Root Value	Weight
1	0	2
2	$\pm 1/\sqrt{3}$	1
3	0	8/9
	$\pm\sqrt{3/5}$	5/9
4	$\pm\sqrt{\frac{(3-2\sqrt{6/5})}{7}}$	$\frac{18+\sqrt{30}}{36}$
	$\pm\sqrt{\frac{(3+2\sqrt{6/5})}{7}}$	$\frac{18-\sqrt{30}}{36}$
5	0	128/225
	$\pm\frac{1}{3}\sqrt{5-2\sqrt{10/7}}$	$\frac{322+13\sqrt{70}}{900}$
	$\pm\frac{1}{3}\sqrt{5+2\sqrt{10/7}}$	$\frac{322-13\sqrt{70}}{900}$

Once again adapting the original code from the **rectz.m** function, we get this function which we will name **quadgl.m**:

```
% Numerical integration using the Gaussian quadrature method
% This routine will use a five-point Gaussian quadrature rule.

function finalValue=quadgl(eqn,a,b,epsilon)

% Setup constants and weights for five-point Gaussian
% quadrature based on Gauss-Legendre polynomial.

c1=(1/3)*(5-2*(10/7)^0.5)^0.5;
c2=(1/3)*(5+2*(10/7)^0.5)^0.5;
w0=128/225;
w1=(322+13*(70)^0.5)/900;
w2=(322-13*(70)^0.5)/900;

% Calculate values based on 2 sections.

sects=2;
w=(b-a)/sects;
wh=w/2;
x=(a+wh):w:b;
h=(w0*wh*eqn(x))+(w1*wh*eqn(-wh*c1+x))+(w1*wh*eqn(wh*c1+x))+
(w2*wh*eqn(-wh*c2+x))+(w2*wh*eqn(wh*c2+x));
totalArea=sum(h);
diffErr=totalArea;
lastArea=totalArea;

% Increase the number of sections until the desired
% accuracy is reached.

while (diffErr>(epsilon*totalArea))
```

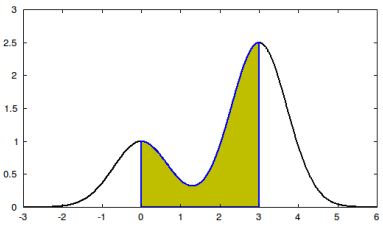
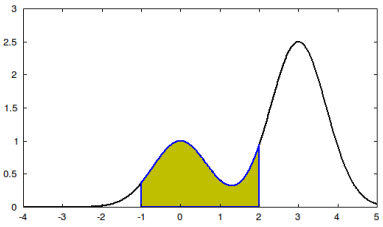
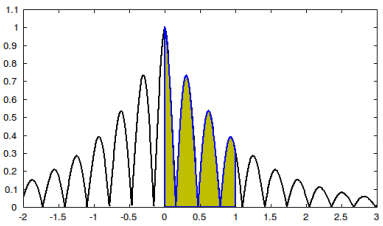
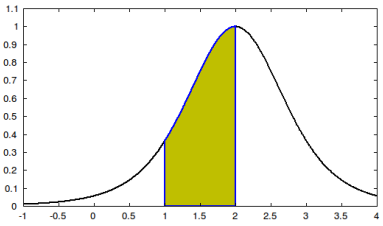
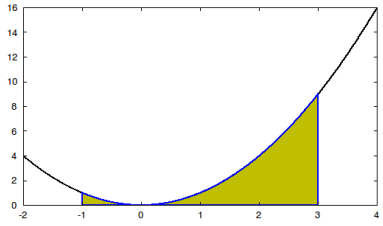
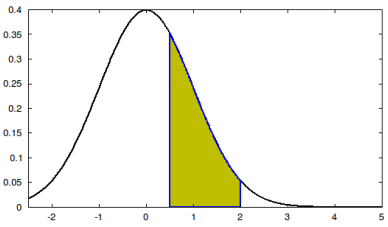
```
sects=sects*2;
w=(b-a)/sects;
wh=w/2;
x=(a+wh):w:b;
h=(w0*wh*eqn(x))+(w1*wh*eqn(-wh*c1+x))+(w1*wh*eqn(wh*c1+x))+
(w2*wh*eqn(-wh*c2+x))+(w2*wh*eqn(wh*c2+x));
totalArea=sum(h);
diffErr=abs(totalArea-lastArea);
lastArea=totalArea;

end

finalValue=totalArea;
```

Checking this with our double exponential equation and an accuracy of 10^{-4}, we get:

```
--> eqn=@(x) (exp(-x.^2)+2.5*exp(-(x-3).^2));
--> quadgl(eqn,-1,2,1e-4)
ans =
    1.9774
```

If you're wondering why we are generating so many different routines that provide the same answer, then you're asking a good question. The answer is that we're essentially independently verifying that we're probably getting a good solution. We tried to integrate the equation $e^{-x^2}+2.5\,e^{-(x-3)^2}$ using the rectangular method, the trapezoidal method, Simpson's rule, and the Gauss-Legendre method. In all cases, we achieved the same solution, approximately 1.9774. That probably means that our routines are working as we hoped. If we had arrived at different answers, then we would probably have wondered about our underlying assumptions.

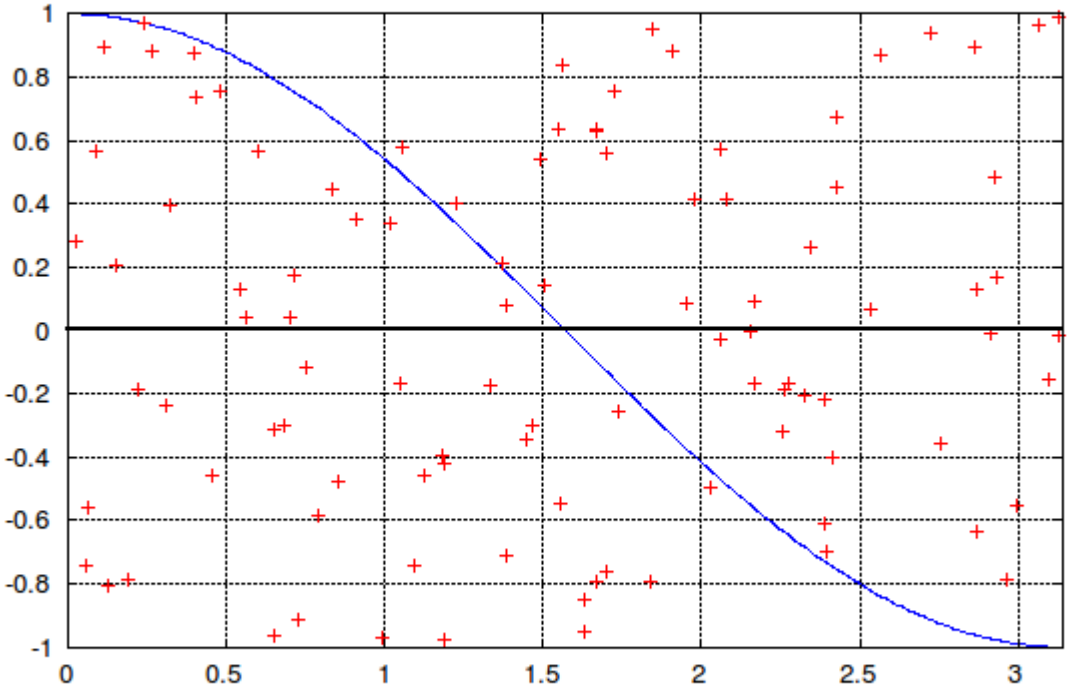
In practical terms, however, we're going to probably stick with one of these. That will be the one that provides the answer with the fewest possible iterations or sections. The following table provides some examples of the number of iterations of the **while** loop in order to achieve the desired accuracy. Note that this is not an indication of the total number of calculations required. On a per-iteration basis, the rectangular method uses the fewest calculations, while the Gauss-Legendre requires many. However, by looking at the number of iterations required, we can get an idea of how quickly we converge on an answer. As the table below shows, the Gauss-Legendre method provides the fastest convergence in cases of proper integrals with no discontinuities (the fewest iterations) of the four methods used thus far.

Equation and Limits	Graph of Equation	Iterations Required for 10^{-5} Accuracy			
		Rect	Trap	Simpson	Gauss
$e^{-x^2}+2.5\,e^{-(x-3)^2}$ from 0 to 3.		3	4	3	2
$e^{-x^2}+2.5\,e^{-(x-3)^2}$ from -1 to 2.		8	9	4	2
$\left\|(e^{-\|(x)\|})\cos(10x)\right\|$ from 0 to 1.		10	8	9	8
$\frac{125}{(5+2*(x-2)^2)^3}$ from 1 to 2.		7	8	2	2
x^2 from -1 to 3.		9	10	2	2
$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$ from 0.5 to 2.		7	7	4	2

Topic 9.3.1.5: Monte Carlo Integration

Monte Carlo methods are methods that use random numbers to do different mathematical tasks. Probably the most common usage is for numerical integration, as we'll describe here. This method can work on anything from simple functions to image analysis. Here's how it works:

Say we were to lay out the domain [0 π] onto a dart board, and plot cos(x) on that domain. If we throw a random distribution of darts at this board, it might look like this: (red crosses are darts)



Numerically, the proportion of darts “under the curve” (between the curve and the x-axis) to the total number of darts thrown is equal to the proportion of area under the curve to total area of the domain. Note that darts above the curve but below the x-axis subtract from this total.

Knowing that there are 100 darts in the above figure, we can manually count and come up with 18 darts below the curve in the left half of the graph, and 18 darts above the curve (but below the axis) in the right half. We subtract this from our first number and get 0. The total area of our domain is 2π, so 0*2π is the area under the curve. We know, of course, that the value of this integral should be zero, and see that we come up with that here.

Pretty simple, right? Let's look at some simple code then! Let's start by making a function we can pass arguments to. We need to tell it three things: our function (which we can pass as a variable using an anonymous function), the minimum x and the maximum x (the extrema of our domain).

```
function return_value = montecarlo(fx, xmin, xmax)

n_darts = 100; % number of darts to throw; more darts = more accurate
```

```
%% --- Determine Domain Size --- %%
x = linspace(xmin, xmax, 200);
for n = 1:length(x)
      y(n) = fx(x(n)); % draw curve to be integrated
end
ymin = min(y);

if ymin > 0 % if y is all positive, min is 0
      ymin = 0;
end

ymax = max(y);

%% --- Throw Darts! --- %%
dartcounter = 0; % This will count the number of darts below the curve
for i = 1:n_darts
      rx = rand()*(xmax-xmin) - xmin; % generates random dart
      ry = rand()*(ymax-ymin) - ymin;

      %% --- Count Darts for Calculation --- %%
      if ry < fx(rx) && ry > 0
            dartcounter = dartcounter + 1;
      else if ry > f(rx) && ry < 0
            dartcounter = dartcounter - 1;
            end
      end
end

totalarea = abs(xmax-xmin)*abs(ymax-ymin);
return_value = (dartcounter/n_darts)*totalarea; % area under curve
```

Now, let's pass an anonymous function (a function stored in a variable) and a domain to our program and see what happens:

```
--> f = @(x) (exp(-x.^2); % VERY hard to integrate!
```

I'll take a quick aside and explain what I just did – the name of our function (f) can be passed around just like any other variable. When we say f = @(x), the @ symbol denotes that it's a function (not a vector or cell or other entity), and the (x) denotes that it's a function of x. If it were a function of x,y and z, we would have f = @(x,y,z). The rest of the expression is simply the function to be evaluated, in this case:

$$f(x)=e^{-x^2}$$

When we call f(3), for example, it will return exp(-3^2). Let's continue:

```
--> x0 = 0; x1 = 2; % evaluate integral from x = 0 to 2

-->montecarlo(f,x0,x1)

ans =
    0.8840
```

Straight away we notice two things. First, our answer 0.8840 is close (but not exactly) the analytical solution to the integral, 0.8821. Second, if we run the same input several times, we get different answers. The more darts you throw, the more accurate the answer will become, at the expense of computational time.

Topic 9.3.2: Numerical Differentiation

We use differentiation to find the the rate of change of some quantity. This quantity can be an equation, or it can be actual data that is collected over some range. The most common range is time. For example, we can use differentiation to calculate the change of a position over time which gives us velocity. Then we can calculate the change of velocity over time and get acceleration.
Another purpose of differentiation is to find the maxima or minima, meaning the peaks, of a function.

Topic 9.3.2.1: Calculating the Rate of Change

The first derivative of a function is nothing more than calculating the slope of that function at every point. The slope is the ratio of the change in the y-axis to the x-axis, sometimes called the "rise over the run". This is shown in Figure 130.

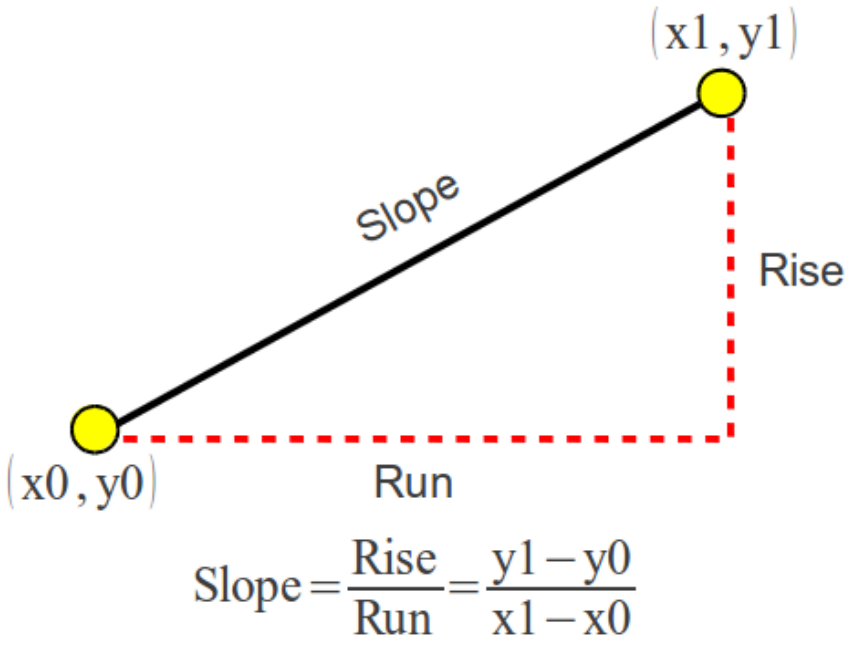


Figure 130: The slope is the ratio of the change in the y-axis over the change in the x-axis.

The slope can be identified as being a positive slope, a negative slope, or no slope at all. This is shown in Figure 131.

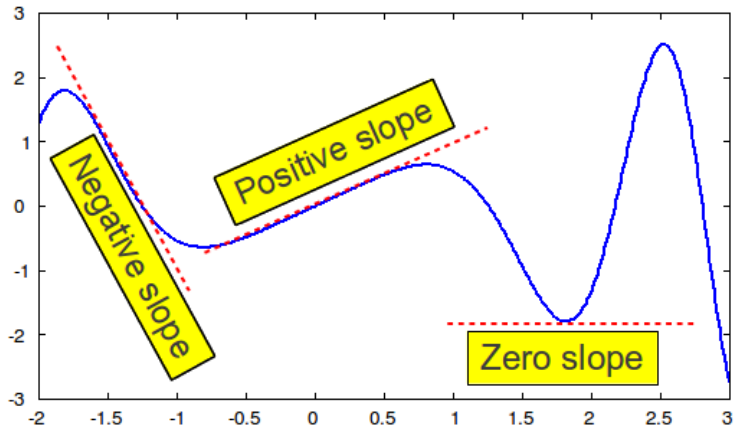


Figure 131: A slope can be described as positive, negative or zero (meaning it's flat).

To calculate the slope of a function at a particular point, there are several methods. These are:

1. Backwards difference: This uses a point, and some point after it.
2. Forward difference: This uses a point, and some point before it.
3. Three-point estimation: This uses a point, a point before it and a point after it. This is shown in Figure 132.
4. Five-point estimation: This uses a point, two points before it and two points after it.

With each of these methods, the concept is to start with some distance between the points, then move the points inwards towards the point of interest, re-calculate the slope, and compare it to the previous calculation. Once the difference is less than some desired precision, you've calculated the slope at that point.

There is a problem with all of these methods. That problem is the fact that computers use a finite number of significant digits. Any algorithm that attempts to make the width smaller will eventually reach a point where the floating point numbers used in the computer do not have the precision necessary. Frankly, we're lucky if we can get an answer with a precision that is better than the square root of the floating point precision of the processor we're using. We can find out this value in Freemat using the **eps** command (FD, p. 185). Here's the value for my computer (a quad-core, Intel-based, 64-bit system):

```
--> eps
ans =
 2.22044604925031e-16
--> sqrt(eps)
ans =
 1.49011611938477e-08
```

This means that the best precision I can expect out of my system is roughly 1e-8. If we get better than that, we're doing well.

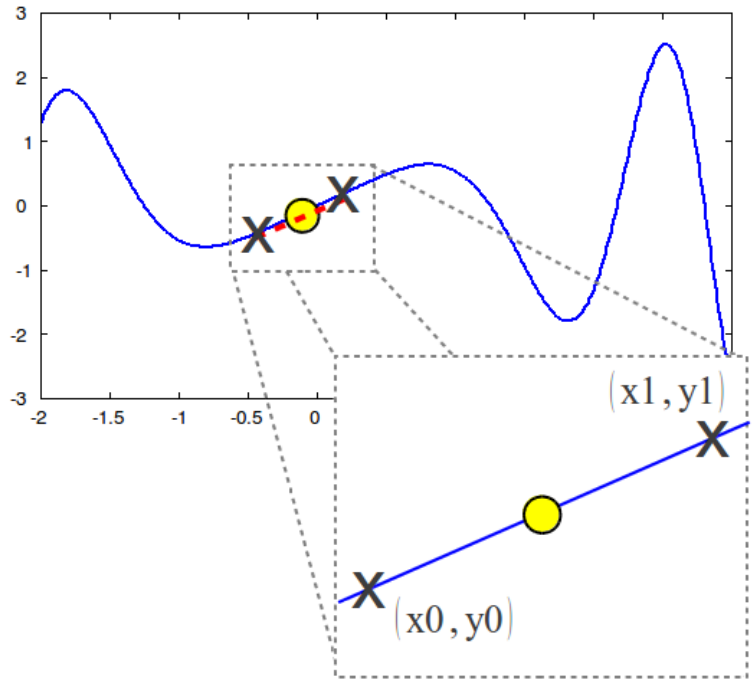


Figure 132: To calculate the slope on a graph at a particular point, use the slope between two points, one just above and one just below the point of interest. Then slowly move the points in towards the point of interest and refine the slope til it meets some precision value. This is known as a three-point difference estimation method.

Here is a short function, called **slopeCompare**, that can be used to calculate the slope on a curve at a defined point.

```
% The purpose of this function is to calculate the slope
% of an equation at a particular point. This is the
% first derivative of the equation at that point.
% This uses the three-point difference method.
% filename = slopeCompare.m
function slopeCompare(f,fp,x,epsilon);
  w=0.1; % starting value for points above and below x
         % to calculate the slope.
  s=(f(x+w)-f(x-w))/(2*w);
  oldSlope=s;
  diffErr=abs(s);
  printf('The initial width is %f.\n',w);
  printf('The initial slope is %14.11f.\n',s);
  loop=0;
  while (diffErr>abs(epsilon*s))
  loop=loop+1;
  w=w*0.1;
  s=(f(x+w)-f(x-w))/(2*w);
  printf('%d) The slope is %14.11f with a width of %e.\n',loop,s,w);
  diffErr=abs(s-oldSlope);
  oldSlope=s;
  end
```

```
  printf('The actual value is %14.11f.\n',fp(x));
printf('The error from the actual value is %e.\n',abs(s-fp(x)));
```

This function takes an equation, the first derivative of the equation, a point on the x-axis, and the desired precision to demonstrate how the slope can be calculated using smaller and smaller lengths of the tangent line.

To demonstrate this function, we'll use the equation that we've been using quite a bit up til now. That equation is:

$$f = x * \cos\left(x^2\right)$$

The first derivative of this equations is:

$$f' = \cos\left(x^2\right) - 2x^2 \sin\left(x^2\right)$$

We'll use the point on the x-axis of -2, and a precision of 1e-5.

```
--> clear all
--> f=@(x) (x.*cos(x.^2));
--> fp=@(x) (cos(x.^2)-(2*(x.^2).*sin(x.^2)));
--> slopeCompare(f,fp,-2,1e-5)
The initial width is 0.100000.
The initial slope is  5.34982302312.
1) The slope is  5.40028317212 with a width of 1.000000e-02.
2) The slope is  5.40077141156 with a width of 1.000000e-03.
3) The slope is  5.40077629230 with a width of 1.000000e-04.
The actual value is  5.40077634160.
The error from the actual value is 4.929508e-08.
```

The error from the actual value is roughly 5e-8. We'll the function again, but this time we'll adjust the precision down to 1e-8.

```
--> slopeCompare(f,fp,-2,1e-8)
The initial width is 0.100000.
The initial slope is  5.34982302312.
1) The slope is  5.40028317212 with a width of 1.000000e-02.
2) The slope is  5.40077141156 with a width of 1.000000e-03.
3) The slope is  5.40077629230 with a width of 1.000000e-04.
4) The slope is  5.40077634115 with a width of 1.000000e-05.
The actual value is  5.40077634160.
The error from the actual value is 4.530429e-10.
```

Note how we're getting values closer and close to the actual value. Our error from the actual value is roughly 4.5e-10. Let's run the function one more time, but dial down the precision to 1e-10.

```
--> slopeCompare(f,fp,-2,1e-10)
The initial width is 0.100000.
The initial slope is  5.34982302312.
1) The slope is  5.40028317212 with a width of 1.000000e-02.
2) The slope is  5.40077141156 with a width of 1.000000e-03.
3) The slope is  5.40077629230 with a width of 1.000000e-04.
4) The slope is  5.40077634115 with a width of 1.000000e-05.
5) The slope is  5.40077634181 with a width of 1.000000e-06.
6) The slope is  5.40077633504 with a width of 1.000000e-07.
7) The slope is  5.40077630395 with a width of 1.000000e-08.
8) The slope is  5.40077682576 with a width of 1.000000e-09.
9) The slope is  5.40077649269 with a width of 1.000000e-10.
10) The slope is  5.40079092559 with a width of 1.000000e-11.
11) The slope is  5.40123501480 with a width of 1.000000e-12.
12) The slope is  5.39679412270 with a width of 1.000000e-13.
13) The slope is  5.46229728116 with a width of 1.000000e-14.
14) The slope is  5.32907051820 with a width of 1.000000e-15.
```

```
15) The slope is  0.00000000000 with a width of 1.000000e-16.
16) The slope is  0.00000000000 with a width of 1.000000e-17.
The actual value is  5.40077634160.
The error from the actual value is 5.400776e+00.
```

What happened? Note that as the solution was converging, it came closest with a width of 1e-6 (Line 5). After that, it started getting further away from the actual value. This is due to the available precision on this computer. If we try to use a precision beyond the **sqrt(eps)**, such calculations may become unstable.

We can eke out a bit more precision by slowing down the change in the width between iterations. Right now, the width is dropped by a factor of 10 (0.1) from one iteration to the next. We can change that to 1/2 to see if it will become more stable. To do this, we change the one line in the function that changes the width between iterations. It should now appear as follows:

```
w=w*0.5;
```

Then we can run the function again with the same parameters as before.

```
--> slopeCompare(f,fp,-2,1e-10)
The initial width is 0.100000.
The initial slope is  5.34982302312.
1) The slope is  5.38834692329 with a width of 5.000000e-02.
2) The slope is  5.39768853988 with a width of 2.500000e-02.
3) The slope is  5.40000561708 with a width of 1.250000e-02.
4) The slope is  5.40058373715 with a width of 6.250000e-03.
5) The slope is  5.40072819528 with a width of 3.125000e-03.
6) The slope is  5.40076430532 with a width of 1.562500e-03.
7) The slope is  5.40077333255 with a width of 7.812500e-04.
8) The slope is  5.40077558934 with a width of 3.906250e-04.
9) The slope is  5.40077615353 with a width of 1.953125e-04.
10) The slope is  5.40077629458 with a width of 9.765625e-05.
11) The slope is  5.40077632986 with a width of 4.882813e-05.
12) The slope is  5.40077633869 with a width of 2.441406e-05.
13) The slope is  5.40077634084 with a width of 1.220703e-05.
14) The slope is  5.40077634136 with a width of 6.103516e-06.
The actual value is  5.40077634160.
The error from the actual value is 2.423999e-10.
```

That worked! What appears to be happening if we change the width too much, we overshoot the answer. Once that happens, we're in unstable territory and the function just goes off into the wild blue yonder (so to speak). With this width, we can try to drop the precision again to 1e-11.

```
--> slopeCompare(f,fp,-2,1e-11)
The initial width is 0.100000.
The initial slope is  5.34982302312.
1) The slope is  5.38834692329 with a width of 5.000000e-02.
2) The slope is  5.39768853988 with a width of 2.500000e-02.
3) The slope is  5.40000561708 with a width of 1.250000e-02.
4) The slope is  5.40058373715 with a width of 6.250000e-03.
5) The slope is  5.40072819528 with a width of 3.125000e-03.
6) The slope is  5.40076430532 with a width of 1.562500e-03.
7) The slope is  5.40077333255 with a width of 7.812500e-04.
8) The slope is  5.40077558934 with a width of 3.906250e-04.
9) The slope is  5.40077615353 with a width of 1.953125e-04.
10) The slope is  5.40077629458 with a width of 9.765625e-05.
11) The slope is  5.40077632986 with a width of 4.882813e-05.
12) The slope is  5.40077633869 with a width of 2.441406e-05.
13) The slope is  5.40077634084 with a width of 1.220703e-05.
14) The slope is  5.40077634136 with a width of 6.103516e-06.
```

```
15) The slope is  5.40077634167 with a width of 3.051758e-06.
16) The slope is  5.40077634163 with a width of 1.525879e-06.
The actual value is  5.40077634160.
```

The error from the actual value is 3.044853e-11

Again, it worked. Now what happens if we try a precision of 1e-12? Let's try it:

```
--> slopeCompare(f,fp,-2,1e-12)
The initial width is 0.100000.
The initial slope is  5.34982302312.
1) The slope is  5.38834692329 with a width of 5.000000e-02.
2) The slope is  5.39768853988 with a width of 2.500000e-02.
3) The slope is  5.40000561708 with a width of 1.250000e-02.
4) The slope is  5.40058373715 with a width of 6.250000e-03.
5) The slope is  5.40072819528 with a width of 3.125000e-03.
6) The slope is  5.40076430532 with a width of 1.562500e-03.
7) The slope is  5.40077333255 with a width of 7.812500e-04.
8) The slope is  5.40077558934 with a width of 3.906250e-04.
9) The slope is  5.40077615353 with a width of 1.953125e-04.
10) The slope is  5.40077629458 with a width of 9.765625e-05.
... (a bunch more lines)
47) The slope is  5.78125000000 with a width of 7.105427e-16.
48) The slope is  6.87500000000 with a width of 3.552714e-16.
49) The slope is  3.12500000000 with a width of 1.776357e-16.
50) The slope is  0.00000000000 with a width of 8.881784e-17.
51) The slope is  0.00000000000 with a width of 4.440892e-17.
The actual value is  5.40077634160.
The error from the actual value is 5.400776e+00.
```

Oops, again. That didn't work.

There are some ways to eke out a bit more precision if (for some reason) you need that much precision. The two ways we'll do this are:

1. Use a slightly more precise estimation method, namely a five-point difference estimation method (vice the three-point method we've used up til now).
2. Use a method known as Richardson's extrapolation method to eke out a bit more precision from whatever estimation method we choose.

Up til now, we've used the three-point difference estimation method to calculate the slope. The equation for this is:

$$\text{Slope}_{3pt} = \frac{f(x+w) - f(x-w)}{2w}$$

We're going to use a five-point difference estimation method, which is as follows:

$$\text{Slope}_{5pt} = \frac{f(x-2w) - 8f(x-w) + 8f(x+w) - f(x+2w)}{12w}$$

The value for the slope, "s", in our function now changes to this:

```
s=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
```

Time to try it. We'll use our previous parameters (the same equation and differential equation as well as the same point, -2) and a precision of 1e-8.

```
--> slopeCompare5(f,fp,-2,1e-8)
The initial width is 0.100000.
The initial slope is  5.40703542861.
1) The slope is  5.40118822335 with a width of 5.000000e-02.
2) The slope is  5.40080241207 with a width of 2.500000e-02.
3) The slope is  5.40077797615 with a width of 1.250000e-02.
4) The slope is  5.40077644384 with a width of 6.250000e-03.
```

```
5) The slope is  5.40077634799 with a width of 3.125000e-03.
6) The slope is  5.40077634200 with a width of 1.562500e-03.
The actual value is  5.40077634160.
The error from the actual value is 3.987459e-10.
```

Note two things. First, we accomplished a slightly better precision (4.0e-10) compared to the three-point method (4.5e-10). Second, it took a few more steps than before. However, let's see what happens if we dial down the desired precision to 1e-12. Remember that with the three-point method and this precision, we wound up with an unstable routine that never converged.

```
--> slopeCompare5(f,fp,-2,1e-12)
The initial width is 0.100000.
The initial slope is  5.40703542861.
1) The slope is  5.40118822335 with a width of 5.000000e-02.
2) The slope is  5.40080241207 with a width of 2.500000e-02.
3) The slope is  5.40077797615 with a width of 1.250000e-02.
4) The slope is  5.40077644384 with a width of 6.250000e-03.
5) The slope is  5.40077634799 with a width of 3.125000e-03.
6) The slope is  5.40077634200 with a width of 1.562500e-03.
7) The slope is  5.40077634162 with a width of 7.812500e-04.
8) The slope is  5.40077634160 with a width of 3.906250e-04.
9) The slope is  5.40077634160 with a width of 1.953125e-04.
10) The slope is  5.40077634159 with a width of 9.765625e-05.
The actual value is  5.40077634160.
The error from the actual value is 7.637446e-12
```

That worked! So with this method, it appears that we're able to converge to a more precise value. The problem with using this method alone is that it makes calculating the slope over arrays difficult. This is due to the presence of the **while** loop. Since this loop has to be run for each value of x in an array, it would take a long time to calculate each value. This leads to another method for eking out a bit more precision using the Richardson extrapolation method.

We won't go into the math behind the Richardson extrapolation method. This method is not a method for calculating numeric differentials; it's a numeric method that makes *other* numeric methods converge faster than they would otherwise. Given a particular function f(x), we calculate the first values using our already-defined five-point difference method. Thus:

$$R_1 = \frac{f(x-2w)-8f(x-w)+8f(x+w)-f(x+2w)}{12w}$$

The Richardson extrapolate can be calculated as follows:

$$R_{j+1}(w) = \frac{4^j R_j(w/2) - R_j(w)}{4^j - 1}$$

In order to make this work, we start by calculating several initial values, then we begin the Richardson extrapolation method. Since each iteration of this method uses two from the previous iteration (with overlap), we lose one with each iteration. For example if we were to start with 4 initial values, in the next iteration we would wind up with three values, then two, then the final one value. As we increase the number of iterations, we increase the precision.

Example - The Richardson Extrapolation Method

Let's look at a basic function that uses two five-point estimation methods, followed by a double iteration of the Richardson extrapolation.

```
% filename = slopeRich.m
function returnValue=slope(f,x);
```

```
if(nargin<2)
printf('Error: This function requires at least a function and point as input.\n');
return
end
w=0.1; % starting value for points above and below x
% to calculate the slope.
s1=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
w=w*0.5;
s05=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
s21=(4*s05-s1)/3;
returnValue=s21;
```

Now we can compare this with the actual derivative value:

```
--> f=@(x) (x.*cos(x.^2));
--> fp=@(x) (cos(x.^2)-(2*(x.^2).*sin(x.^2)));
--> abs(fp(-2)-slopeRich(f,-2))
ans =
    0.0015
```

We can continue adding smaller widths, as well as more Richardson extrapolates.

```
% filename = slopeRich.m
function returnValue=slope(f,x);
if(nargin<2)
printf('Error: This function requires at least a function and point as input.\n');
return
end
w=0.1; % starting value for points above and below x
% to calculate the slope.
s1=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
w=w*0.5;
s05=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
w=w*0.5;
s025=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
w=w*0.5;
s0125=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
w=w*0.5;
s00625=(f(x-2*w)-8*f(x-w)+8*f(x+w)-f(x+2*w))/(12*w);
s21=(4*s05-s1)/3;
s205=(4*s025-s05)/3;
s2025=(4*s0125-s025)/3;
s20125=(4*s00625-s0125)/3;
s31=(16*s205-s21)/15;
s305=(16*s2025-s205)/15;
s3025=(16*s20125-s2025)/15;
s41=(64*s305-s31)/63;
s405=(64*s3025-s305)/63;
s51=(256*s405-s41)/255;
returnValue=s51;
```

Running this enhanced script, we get this:

```
--> f=@(x) (x.*cos(x.^2));
--> fp=@(x) (cos(x.^2)-(2*(x.^2).*sin(x.^2)));
--> abs(fp(-2)-slopeRich(f,-2))
ans =
 1.3145e-13
```

The new script uses five different widths (starting with a width of 0.1, and going down by 1/2 of that for each smaller value), then using four Richardson extrapolation loops. It allowed for the precision for

this problem to drop from 1.5e-3 down to the miniscule 1.3e-13.
The other thing is that we accomplished this without using a **while** loop. That means that we can pass an array to this function without having to use any looping.

Topic 9.3.2.2: Numeric Differentiation of Sampled Data

Differentation of a theoretical function, such as $f(x)=e^{-x^2}$, means that we can calculate any number of points that we want. With the Richardson extrapolation method, for example, we need to select some width "w", then calculate for half of that width. Again, no problem with a theoretical equation. We have access to an infinite number of points.
It's a different story when calculating with sampled data. In this case, we have a couple of problems. These are a finite amount of data and the fact that quantizing effects become more pronounced with differentiation. Depending on the conditions of the sampling, such as the sampling rate and the amount of time over which something was sampled, we might have a very small amount of data with which to work.
Still, we can use several of the methods we discussed previously. For example, we can still make use of the three-point and five-point difference estimation methods. However, we have to be careful with the ends of the array used to make the calculations.

Example - Using Numeric Differentials to Calculate Velocity and Acceleration

Okay, the first derivative or its numeric equivalent, the numeric differential, provides the rate of change of some function. Let's put this concept to something concrete. We've discussed the American Physical Society's outing to Six Flags America in Largo, Maryland (See Topic 6.3.4: Adding a Grid on page 133). You can read about it yourself at http://physicsbuzz.physicscentral.com/2011/07/physics-of-six-flags.html. One of the rides they went on was the Tower of Doom. This ride consists of a tower with seats connected to rails on the side. The seats slowly rise up the tower, then are suddenly dropped. This is shown in Figure 133.

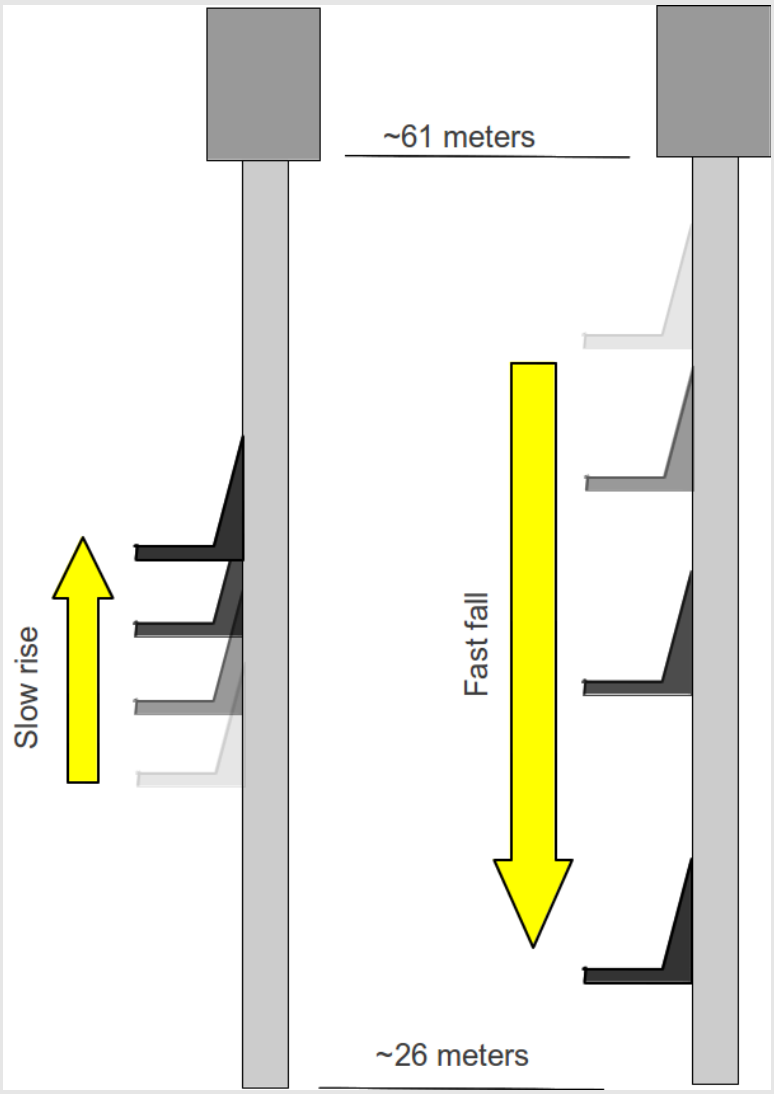


Figure 133: General overview of the Tower of Doom ride.

As part of their outing, they collected four sets of data. This data was acceleration data (using accelerometers) in three axes (x, y and z, which I believe corresponds to forward/backward, up/down and left/right), and altitude (height). It's the altitude data that we're going to use for this example. Here's a short script that will read in the altitude data, plot it and label the important parts of the graph.

```
% plotToDAltitude.m
clear all;
close('all')
oldDir=pwd; % Store the current directory so that we can return to it.
cd('/home/gary/Freemat/Data'); % Change to the directory holding the data files.
data=real(dlmread('todAltitude.txt',char(9))); % Read in the data.
accelToD=real(dlmread('todAccelY.txt',char(9))); % Read acceleration data.
cd(oldDir); % Now that the data is read in, return to the old directory.
t=data(3:length(data),1);
altitude=data(3:length(data),2);
plot(t,altitude);
%sizefig(1200,700);
xDataset;
```

```
grid on;
title('Altitude on Tower of Doom Ride','fontsize',16);
xlabel('Time (sec)','fontsize',12);
ylabel('Altitude (meters)','fontsize',12);
labelSet;
text(18,45,'Seats rising up tower','horizontalalignment','center','rotation',60)
text(40,45,'Seats falling down tower','horizontalalignment','center','rotation',-
85)
```

Running this script, we get this graph.

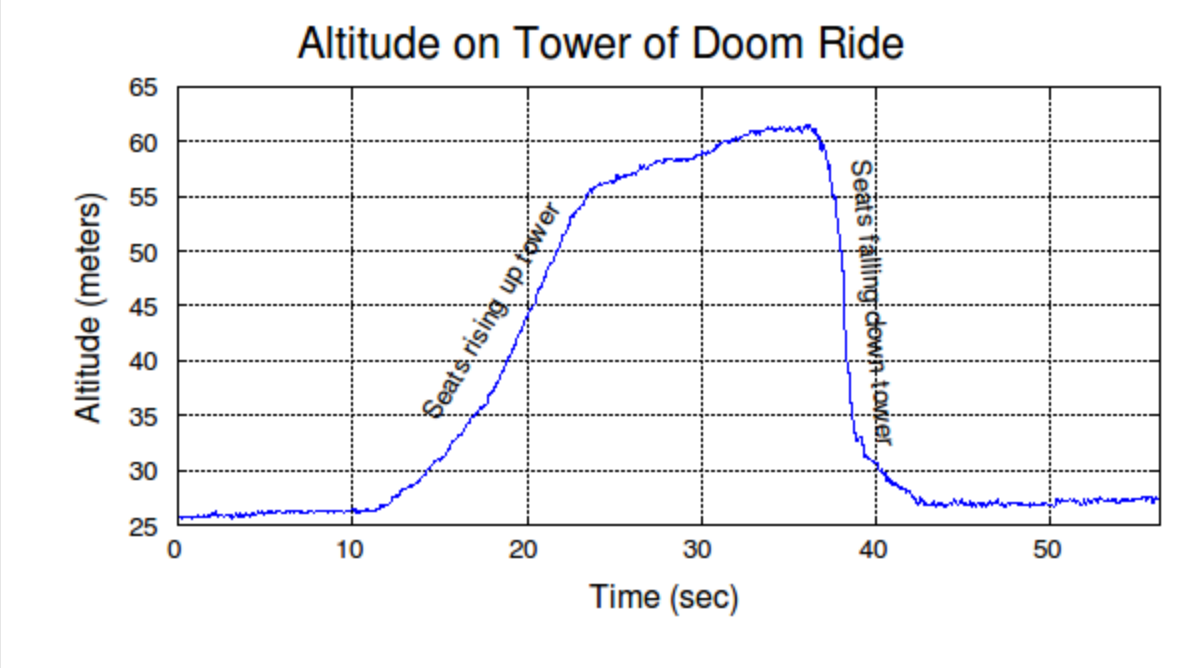


Figure 134: This is the data from the Tower of Doom ride at Six Flags America in Largo, Maryland (Courtesy of the American Physical Society Public Outreach Program). The lefthand slope is relatively shallow, showing the slow movement up the tower. The slope on the right is steep due to the fact that the altitude changes quickly when the seats are dropped.

We can now use this data to calculate velocity and acceleration. Let's make sure we understand the basics, though. Velocity is the change of distance over time. Acceleration is the change of velocity over time. Putting this into equations, it would appear like this:

$$v = \frac{dy}{dt}$$

$$a = \frac{dv}{dt}$$

If you're not deep into math and these equations cause you to hyperventilate, try looking at them a little differently. The change of position, labeled "dy", is nothing more than the difference between two successive position points. The change in time, "dt", is the difference between two successive time points. Look at Figure 135 which shows this concept graphically.

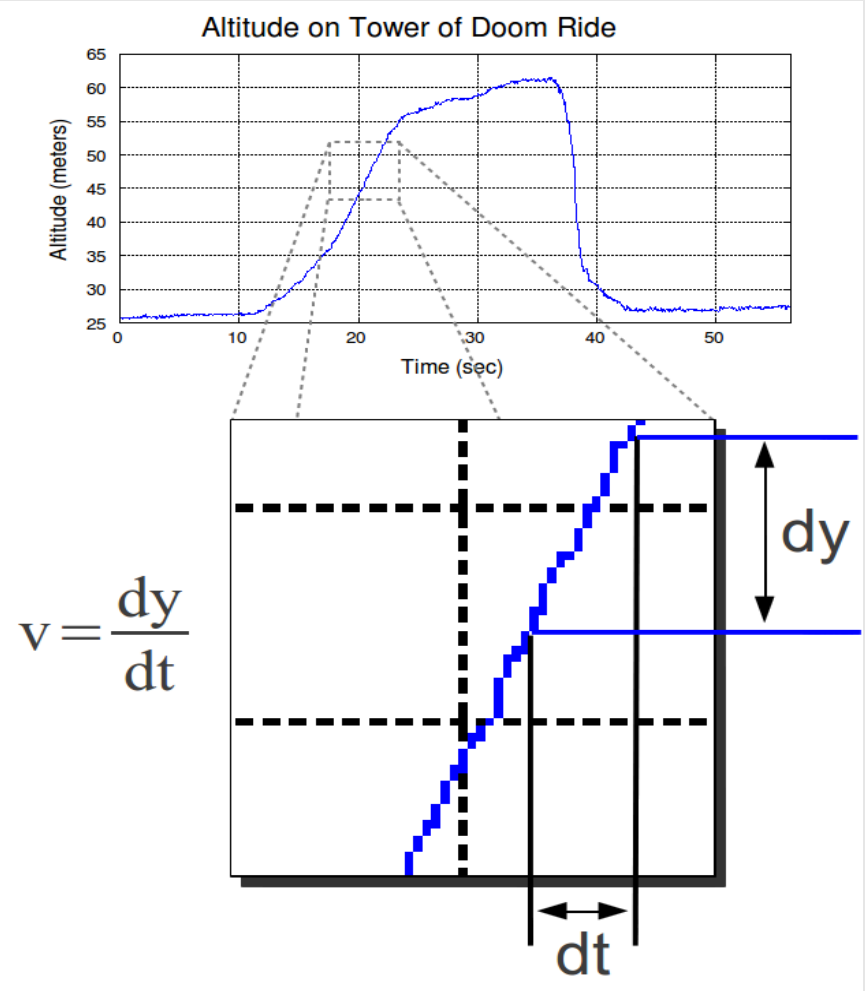


Figure 135: The velocity of the ride is the change in position over time. All we have to do to calculate this is to calculate the difference in position, then divide that by the difference in time.

We'll create a short script that makes use of the **diff** function (FD, p. 202) to calculate the difference in successive points of the position. Then we'll divide that by the difference in time samples to calculate the velocity over time.

```
% ToDVelocity.m
clear all;
close('all')
oldDir=pwd; % Store the current directory so that we can return to it.
cd('/home/gary/Freemat/Data'); % Change to the directory holding the data files.
data=real(dlmread('todAltitude.txt',char(9))); % Read in the data.
accelToD=real(dlmread('todAccelY.txt',char(9))); % Read acceleration data.
cd(oldDir); % Now that the data is read in, return to the old directory.
t=data(3:length(data),1);
altitude=data(3:length(data),2);
% Calculate velocity
altDiff=diff(altitude);
timeDelta=diff(t);
timeDiff=timeDelta(1);
dataVel=altDiff/timeDiff;
```

```
t2=t(1:length(dataVel));
plot(t2,dataVel);
xDataset;
grid on;
title('Vertical Velocity on Tower of Doom Ride','fontsize',16);
xlabel('Time (sec)','fontsize',12);
ylabel('Velocity (m/sec)','fontsize',12);
labelSet;
```

This results in the graph shown in .

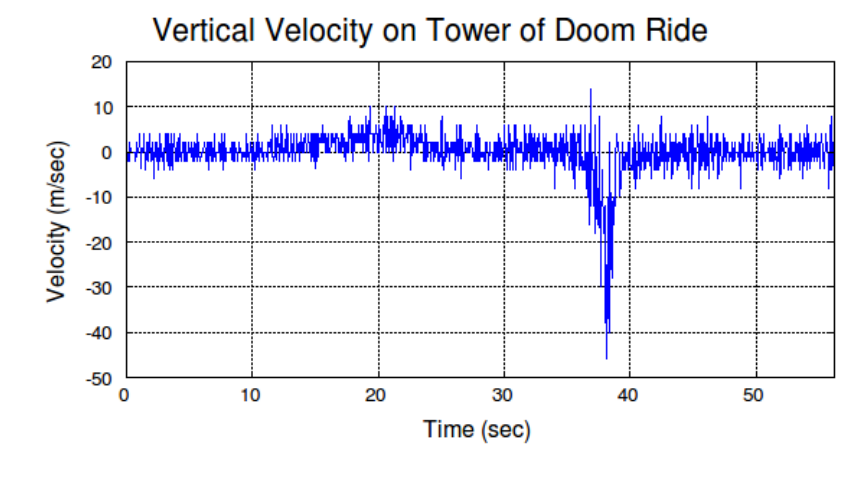


Figure 136: Calculated velocity on the Six Flags Tower of Doom ride. Note that due to the noise in the measurements, it's very difficult to see subtle changes in the velocity.

Looking at the calculated velocity, there appears to be a lot of noise in the data. We can clean this up somewhat using a simple moving average. This function is provided below.

```
% Calculate a simple moving average
% moveAve.m
%
% x = the array over which the moving average
% will be calculated.
% n = the number of points to calculate the average.
% (the default = 3)
function returnArray=moveAve(x,n);
% Check to see if an array has been provided.
% If not, print an error and quit the function.
if(nargin==0);
printf('ERROR: You have to supply an array.');
return
end
arraySize=size(x);
if(min(arraySize)>1);
printf('ERROR: The provided array is has a minimum\n');
printf('dimension of %d. It requires a one-dimensional vector.\n',min(arraySize));
return
end
% Check to see if an amount of averaging has been
% provided. If not, set a default value.
if(nargin==1);
n=3;
```

```
end
% If for some reason the average is set to 1,
% just return the original array.
if(n==1);
returnArray=x;
return
end
% Set the array to determine if it needs to be
% flipped.
flipFlag=0;
if(arraySize(1)>arraySize(2))
x=x';
flipFlag=1;
end
sumX=0;
for(ii=1:n);
sumX=sumX+x;
x=circshift(x,[0 -1]);
end
if(flipFlag==1)
sumX=sumX';
end
returnArray=(sumX(1:length(x)-n))/n;
```

Now we'll add this function to our previous script. That script now appears as follows:

```
% ToDVelocity.m
clear all;
close('all')
oldDir=pwd; % Store the current directory so that we can return to it.
cd('/home/gary/Freemat/Data'); % Change to the directory holding the data files.
data=real(dlmread('todAltitude.txt',char(9))); % Read in the data.
accelToD=real(dlmread('todAccelY.txt',char(9))); % Read acceleration data.
cd(oldDir); % Now that the data is read in, return to the old directory.
t=data(3:length(data),1);
altitude=data(3:length(data),2);
% Calculate velocity
altDiff=diff(altitude);
timeDelta=diff(t);
timeDiff=timeDelta(1);
dataVel=altDiff/timeDiff;
velAve=moveAve(dataVel,30);
tvA=t(1:length(velAve));
plot(tvA,velAve);
xDataset;
grid on;
title('Vertical Velocity on Tower of Doom Ride','fontsize',16);
xlabel('Time (sec)','fontsize',12);
ylabel('Velocity (m/sec)','fontsize',12);
labelSet;
```

Here's how the smoothed data appears:

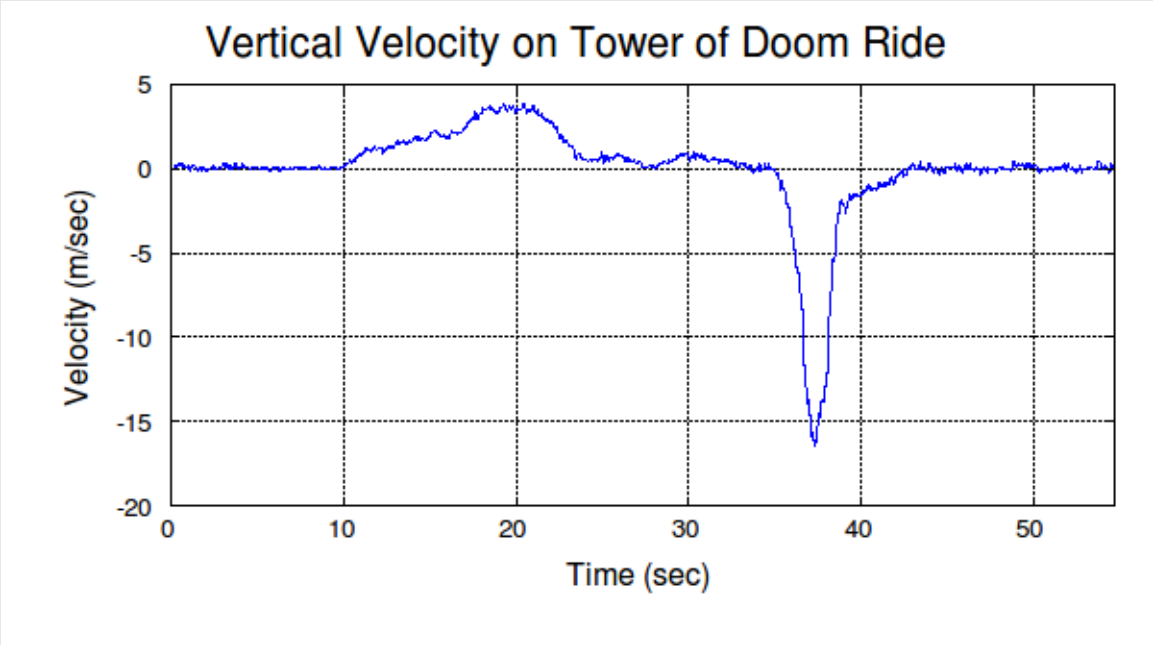


Figure 137: This is the velocity data, but after having been passed through a smoothing function to "clean up" some the noise of the original graph. Note how the ride starts up at about the 10 second mark. Also note how the velocity changes at roughly the 16 second mark, then goes down to a very slow crawl roughly the 23 second mark. Finally, it drops at roughly the 35 second mark.

Next, we'll calculate the numeric differential of the velocity data, which will tell use the acceleration of the ride. We'll expand on our previous script and use another **diff** command to calculate the acceleration over time.

```
% ToDAcceleration.m
clear all;
close('all')
oldDir=pwd; % Store the current directory so that we can return to it.
cd('/home/gary/Freemat/Data'); % Change to the directory holding the data files.
data=real(dlmread('todAltitude.txt',char(9))); % Read in the data.
accelToD=real(dlmread('todAccelY.txt',char(9))); % Read acceleration data.
cd(oldDir); % Now that the data is read in, return to the old directory.
t=data(3:length(data),1);
altitude=data(3:length(data),2);
altAve=moveAve(altitude,20);
% Calculate the velocity.
altDiff=diff(altAve);
timeDelta=diff(t);
timeDiff=timeDelta(1);
dataVel=altDiff/timeDiff;
t2=t(1:length(dataVel));
velAve=moveAve(dataVel,20);
tvA=t(1:length(velAve));
% Calculate acceleration
velDiff=diff(velAve);
dataAccel=velDiff/timeDiff/1;
aveAccel=moveAve(dataAccel,10);
t3=t(1:length(aveAccel));
```

```
plot(t3,aveAccel,'linewidth',2);
xDataset;
grid on;
title('Vertical Acceleration on Tower of Doom Ride','fontsize',16);
xlabel('Time (sec)','fontsize',12);
ylabel('Acceleration (m/s/s)','fontsize',12);
labelSet;
```

This results in the acceleration graph shown below.

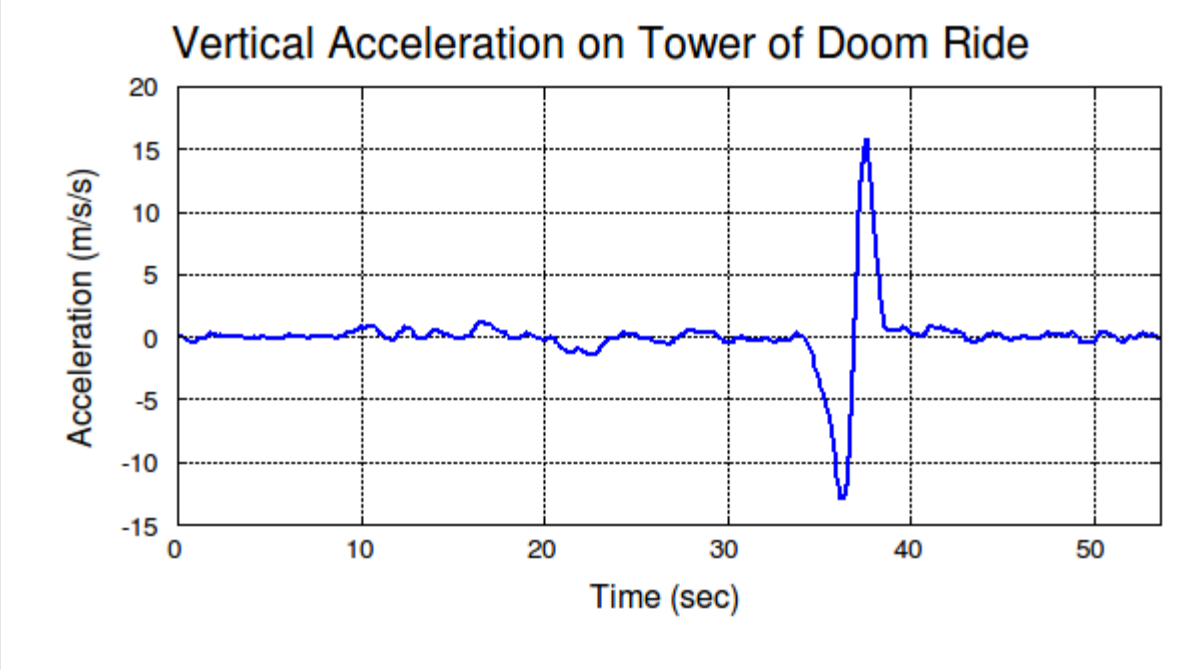


Figure 138: This shows the calculated acceleration of the Tower of Doom ride. Note that the data has been smoothed using a simple moving average.

Since another one of the data sets is the y-axis (up/down) acceleration (using accelerometers), we can plot these two data sets together to see how well they match. We showed this data set in Figure 90.

```
% ToDAcceleration.m
clear all;
close('all')
data=real(dlmread('todAltitude.txt',char(9))); % Read in the data.
t=data(3:length(data),1);
altitude=data(3:length(data),2);
altAve=moveAve(altitude,20);
% Calculate the velocity.
altDiff=diff(altAve);
timeDelta=diff(t);
timeDiff=timeDelta(1);
dataVel=altDiff/timeDiff;
t2=t(1:length(dataVel));
velAve=moveAve(dataVel,20);
tvA=t(1:length(velAve));
% Calculate acceleration
velDiff=diff(velAve);
dataAccel=velDiff/timeDiff/1;
aveAccel=moveAve(dataAccel,10);
t3=t(1:length(aveAccel));
```

```
plot(t3,aveAccel,'linewidth',2);
xDataset;
grid on;
title('Vertical Acceleration on Tower of Doom Ride','fontsize',16);
xlabel('Time (sec)','fontsize',12);
ylabel('Acceleration (m/s/s)','fontsize',12);
labelSet;
% Plot y-axis accelerometer data
accelToD=real(dlmread('todAccelY.txt',char(9))); % Read acceleration data.
ta=accelToD(3:length(accelToD),1);
accel=accelToD(3:length(accelToD),2)-9.80655; % Normalize the acceleration to zero.
accelAve=moveAve(accel,10); % Smooth out the accelerometer data.
accelAve=circshift(accelAve,[-18 0]); % Shift the data sets so that they line up in
time.
taAve=ta(1:length(accelAve));
line(taAve,accelAve,zeros(1,length(taAve)),'color','r','linewidth',2);
legend('Differentiation of Altitude Data','Accelerometer
Data','location','northwest')
```

The result is shown in Figure 139.

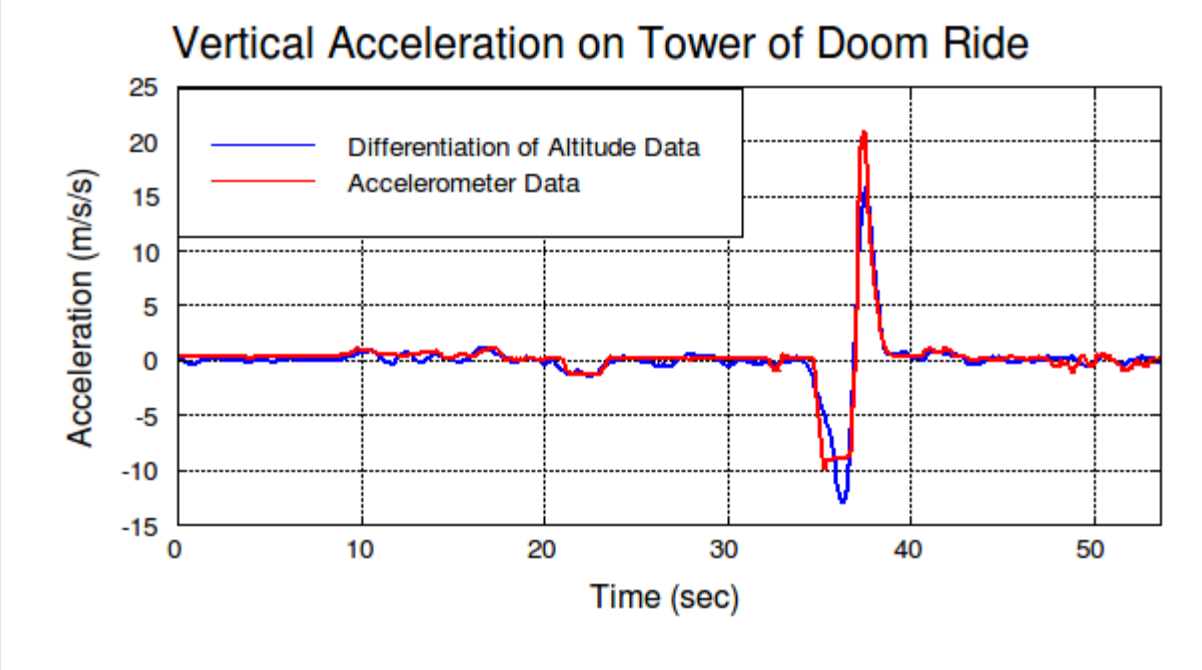


Figure 139: This is the y-axis accelerometer data plotted over the calculated acceleration data. Note the excellent match between the two, with the exception of the sharp drop near the 37 second mark). This difference probably has to do with the smoothing used on the altitude and velocity data.

Note that, with the exception of the acceleration when the chairs are dropped, the two data sets match pretty well. While I don't know for certain, I believe that the reason for the difference is due to the moving average filter applied to the data sets.

Topic 9.3.2.3: Finding the Maxima and Minima

A maxima or a minima is a point on a graph where the slope goes to zero. A maxima or minima is where the slope (the first derivate) goes to zero. This is shown in Figure 140.

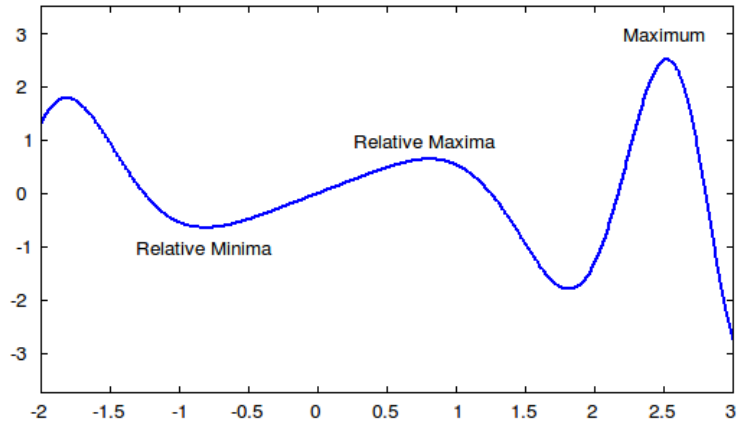


Figure 140: Overview of maxima and minima.

That means that if we plot the first differential, any points where this plot goes through zero will be points at which the original graph has a maxima or minima. This is shown below.

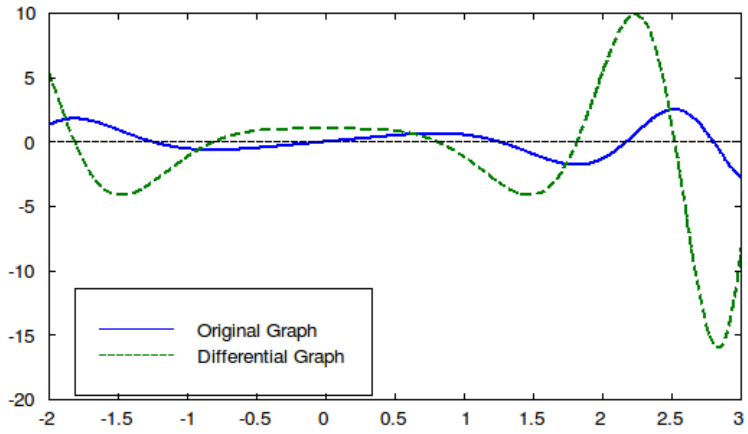


Figure 141: This shows how an original graph compares with a differential graph.

We can find the actual zero-crossing points using numerical differentiation. To do that, we can use the **diff** command (FD, p.). Here's a short bit of code to provide an example.

```
x=linspace(-2,3,1000); % Create the x-axis range.
y=x.*cos(x.^2); % Create the points of the graph.
plot(x,y,'linewidth',2); % Plot the original function graph
yLimits=get(gca,'ylim'); % Save the y-axis limits for later.
dx=x(1:end-1); % Create an x-axis range for the differential graph.
diffX=x(2)-x(1); % Create the differential graph.
dy=diff(y)/diffX; % Calculate the slope of the differential graph.
% The next line plots the differential graph.
line(dx,dy,zeros(1,length(dx)),'linewidth',2,'color','r');
% Create and plot the zero line.
```

```
line([-2 3],[0 0],[0 0],'linewidth',2,'color','k','linestyle','--');
ylim(yLimits); % Reset the y-axis limits to the original levels.
legend('Original graph','Differential graph','location','southwest');
```

Here's the result of running this script.

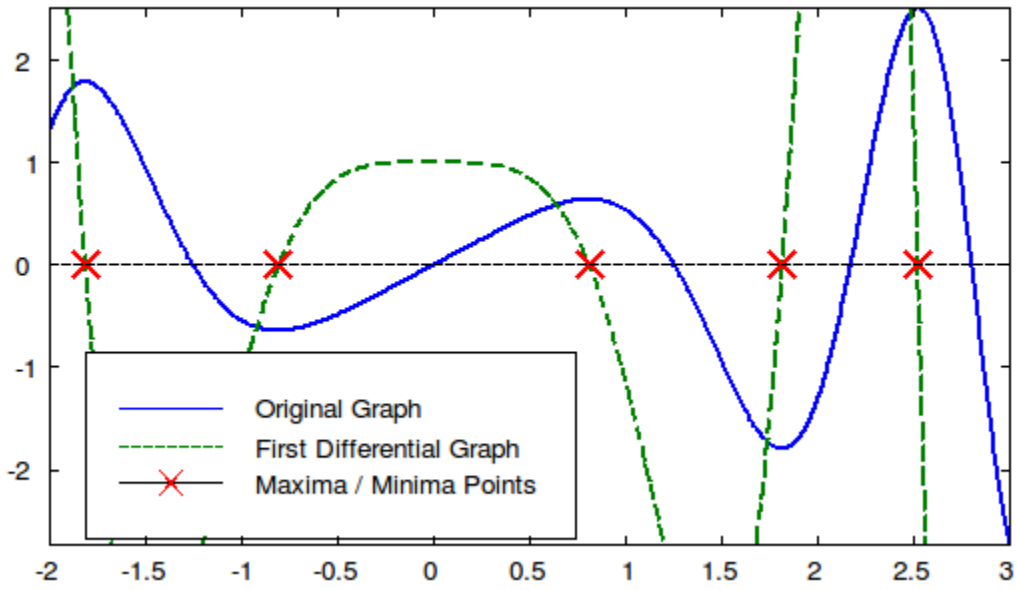


Figure 142: This shows a graph comparing the original graph (in blue) with the differential graph (the green, dotted line). Note that the maxima and minima of the original graph occur at the same points as when the differential graph passes through zero. These zero-crossing ponts of the differential are marked with a red "X". Also note that a negative slope on the differential graph means a maxima on the original graph; a positive slope means a minima.

We can use this differential graph along with the **circshift** function (FD, p. 269) to get a better idea of where the differential graph passes through zero.

```
% Script to demonstrate how to calculate where zero crossings occur.
% zeroCross.m
f=@(x) (x.*cos(x.^2)); % Equation to graph
x=linspace(-2,3,1024);
y=f(x); % graph of original equation
dx=x(1:end-1);
diffX=x(2)-x(1);
dy=diff(y)/diffX;
ys=circshift(dy,[0,-1]); % this creates the shifted version of the
                         % original graph.  The shifted version will
                         % be shifted to the left (-1) compared to
                         % the original.
yz=(dy.*ys)<0; % This looks for only points that are negative (less
               % than zero).
plot(x,y,'linewidth',2);
% plot zero crossings in red
line(dx,yz,zeros(1,length(dx)),'color','r');
legend('Original Graph','Maxima / Minima Points','location',
'southwest');
```

This results in the figure shown in Figure 143.

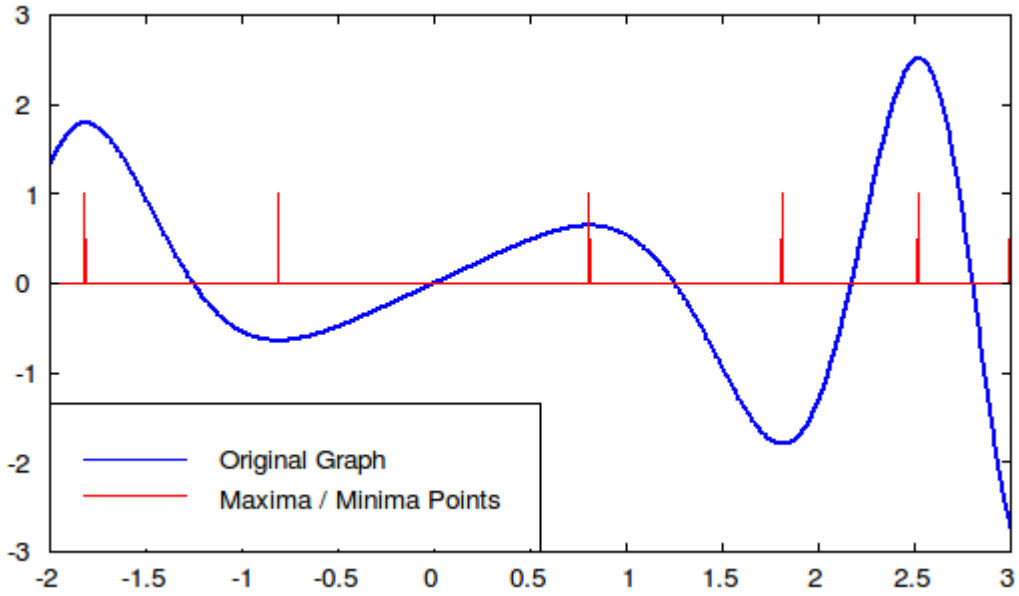


Figure 143: The original graph (in blue) with pulses (in red) showing where the maxima and minima occur on the original graph.

We can add additional code to our **zeroCross.m** script that will go through the the vector of pulses and determine their approximate x-axis positions. We can then use that knowledge to calculate the location of these maxima and minima more precisely.

```
% Script to demonstrate how to calculate where zero crossings occur.
% zeroCross.m
clear all;
close('all');
f=@(x) (x.*cos(x.^2)); % Equation to graph
x=linspace(-2,3,1024);
y=f(x); % graph of original equation
dx=x(1:end-1);
diffX=x(2)-x(1);
dy=diff(y)/diffX;
ys=circshift(dy,[0,-1]); % this creates the shifted version of the
% original graph.
yz=(dy.*ys)<0; % This looks for only points that are negative (less
% than zero).
yz=yz(1:end-1); % This removes the wrap-around point on the end.
dx=dx(1:length(yz)); % This ensures the x-axis is the same length
% as the y-axis.
plot(x,y,'linewidth',2);
% plot zero crossings in red
line(dx,yz,zeros(1,length(dx)),'color','r');
legend('Original Graph','Maxima / Minima Points','location',
'southwest');
c=0;
for(ii=1:length(yz)-1);
if(yz(ii)>0);
c=c+1; % If the pulse goes positive, increase the counter.
xz(c)=x(ii); % Add the x-axis position to the array.
```

```
end
end
xz % Print out the array of points.
```

This results in the same graph as shown in Figure 143, but it also provides the following print-out.

```
--> zeroCross
ans =
   -1.8192   -0.8123    0.8055    1.8074    2.5161
```

We can use these points as starting points for finding more precise values of the maxima and minima. We can use the Newton-Raphson method outlined in Topic 9.1.2: The Newton-Raphson Method on page 163. However, we have to use a slightly different version. For our problem finding the maxima and minima, we have to find the roots of the first differential. This means we need to calculate the first differential, and then calculate the slope of that to home in on the more precise value of the maxima and minima using the second differential. In other words, need to calculate the slope of the slope. We can generate a new function, which we'll call **sloperich2**, that will allow us to calculate the second numerical differentiation. This is identical to the original **sloperich.m** function in that it uses the five-point differentiation method along with Richardson extrapolation to achieve good precision. All we're doing here is substituting the slope calculations for the calculations of the original function. In other words, we're replacing f(x) with sloperich(f(x)).

```
% The purpose of this function is to calculate the slope
% of an equation at a particular point. This is the
% second derivative of the equation at that point.
% This uses the five-point differentiation method along
% with Richardson's extrapolation.
% filename = sloperich2.m
function returnValue=slope(f,x);
if(nargin<2)
printf('Error: This function requires at least a function and point as
input.\n');
return
end
w=0.1; % starting value for points above and below x
% to calculate the slope.
s1=(sloperich(f,x-2*w)-8*sloperich(f,x-w)+8*sloperich(f,x+w)-
sloperich(f,x+2*w))/(12*w);
w=w*0.5;
s05=(sloperich(f,x-2*w)-8*sloperich(f,x-w)+8*sloperich(f,x+w)-
sloperich(f,x+2*w))/(12*w);
w=w*0.5;
s025=(sloperich(f,x-2*w)-8*sloperich(f,x-w)+8*sloperich(f,x+w)-
sloperich(f,x+2*w))/(12*w);
w=w*0.5;
s0125=(sloperich(f,x-2*w)-8*sloperich(f,x-w)+8*sloperich(f,x+w)-
sloperich(f,x+2*w))/(12*w);
w=w*0.5;
s00625=(sloperich(f,x-2*w)-8*sloperich(f,x-w)+8*sloperich(f,x+w)-
sloperich(f,x+2*w))/(12*w);
s21=(4*s05-s1)/3;
s205=(4*s025-s05)/3;
s2025=(4*s0125-s025)/3;
s20125=(4*s00625-s0125)/3;
s31=(16*s205-s21)/15;
s305=(16*s2025-s05)/15;
s3025=(16*s20125-s2025)/15;
```

```
s41=(64*s305-s31)/63;
s405=(64*s3025-s305)/63;
s51=(256*s405-s41)/255;
returnValue=s51;
```

We can use this in our previous script to calculate the precise point of the maxima and minima.

```
% Script to demonstrate how to calculate where zero crossings occur.
% zeroCross.m
clear all;
close('all');
f=@(x) (x.*cos(x.^2)); % Equation to graph
x=linspace(-2,3,1024);
y=f(x); % graph of original equation
dy=slope1(f,x);
ys=circshift(dy,[0,-1]); % this creates the shifted version of the
% original graph.
yz=(dy.*ys)<0; % This looks for only points that are negative (less
% than zero).
yz=yz(1:end-1); % This removes the wrap-around point on the end.
dx=x(1:length(yz)); % This ensures the x-axis is the same length
                    % as the y-axis.
% Calculate the approximate points where the maxima and minima occur.
% This is possible by looking at which points have a positive impulse.
c=0;
for(ii=1:length(x)-1);
if(yz(ii)>0);
c=c+1; % If the pulse goes positive, increase the counter.
xz(c)=x(ii); % Add the x-axis position to the array.
end
end
printf('Approximate maxima and minima points');
xz'
% Refine the location of the maxima and minima by using the
% Newton-Raphson method. This is done by starting with the zero-
% crossing points of the first differential and calculating the
% slope of them using the second differential. Then use these two
% components in the Newton-Raphson method to converge on a (fairly
% accurate) solution of where the maxima and minima occur.
epsilon=1e-5;;
for(ii=1:c);
clear p;
k=1; % Start the counter at 1.
p(k)=xz(ii); % This is the initial guess for the NR method.
fp=slope1(f,p(k));
fp2=slope2(f,p(k));
p(k+1)=p(k)-fp/fp2;
while(abs(p(k+1)-p(k))>epsilon)
k=k+1;
fp=slope1(f,p(k));
fp2=slope2(f,p(k));
p(k+1)=p(k)-fp/fp2;
end
xz(ii)=p(end);
end
printf('Precise points');
xz'
```

Running this script, we get this:

```
--> format long
--> zeroCross
Approximate maxima and minima points
ans =
  -1.81915933528837
  -0.81231671554252
   0.80547409579668
   1.81231671554252
   2.52101661779081
Precise points
ans =
  -1.81447238096425
  -0.80825193293577
   0.80825193293577
   1.81447238101386
   2.52222528574970
```

This gives us the relatively precise points where the maxima and minima occur.

Topic 9.3.2.4: Telling if its Maxima or Minima

The next question we can ask is this: did we find a maxima or a minima. It's going to be one or the other. Sure, we can just look at the graph and tell. But wouldn't it be nice if Freemat could figure it out for us? There's a way to do that. Look back at Figure 142 on page 210. Notice how the graph of the first differential has a particular slope depending on whether the original graph is a maxima or minima? The graph of the first differential will have a positive slope for a minima, and a negative slope for the maxima. This means that if we calculate the second differential at the maxima or minima, it will be negative for the maxima and positive for the minima. From the previous script, we already have the variable xz that has stored all of the x-axis points where the maxima / minima occur. We can create another short script that will go through each value, use the **sloperich2** function we created above, and determine if the peak at that point is a maxima or a minima.

```
% Determine if point is a maxima or a minima
% filename=maxMinDetermine.m
% NOTE: This script requires the function
% zeroCross.m be run before this script is run.
for(ii=1:length(xz));
if(sloperich2(f,xz(ii))<0);
% If the second differential is negative, its a maxima.
printf('The point at %f is a maxima.\n',xz(ii));
else
% Otherwise, its a minima.
printf('The point at %f is a minima.\n',xz(ii));
end
end
```

Let's give it a try.

```
--> maxMinDetermine
The point at -1.814472 is a maxima.
The point at -0.808252 is a minima.
The point at 0.808252 is a maxima.
The point at 1.814472 is a minima.
The point at 2.522225 is a maxima.
```

Comparing these results with the graph shown in Figure 142, it appears that this works just fine.

Appendix A: Special Function to Resize Plots with Titles and Labels

As of this writing, there are a couple of bugs in way that Freemat handles figures and plots. One bug is the position of the text when using the **xlabel** command (as shown in Topic 6.3.3.2 Setting X-Axis (Horizontal) & Y-Axis (Vertical) Labels). Another bug is the fact with the command:

```
get(gca,'position')
```

This command works fine when entered manually in the Command Window. However, if you attempt to use it in either a script or a function, it returns incorrect data. Until these are corrected, here's a function that you can save as the file labelSet.m somewhere in your path. And you can use it whenever you create a graph that has a title and/or labels (x-axis, y-axis or both). While it is far from perfect, your graph should have a decent enough appearance for publishing.

```
% This function will maximize the graph size
% based on the size of any titles or labels
% on the graph.
% THIS FUNCTION SHOULD BE RUN AFTER ALL TITLES
% AND LABELS HAVE BEEN ADDED!
function labelSet
xaxisMarks=24; % This is the space needed for the x-axis numeric
labels.
yaxisMarks=40; % This is the space needed for the y-axis numeric
labels.
marginX=20; % Extra padding for the x-axis.
marginY=20; % Extra padding for the y-axis.
curHand=gca; % Get the current figure handle.
curSize=get(gcf,'figsize'); % Get the size of the current figure.
curTitle=get(curHand,'title'); % Get the handle of the title (if any).
if(isempty(curTitle)); % If there is no title, skip this section.
titleFontsize=0;
else
titleFontsize=1.5*get(curTitle,'fontsize');
set(curTitle,'verticalalignment','middle');
set(curTitle,'horizontalalignment','center');
if(titleFontsize<20)
titleFontsize=20;
end
end
curXlabel=get(curHand,'xlabel');
if(isempty(curXlabel));
xFontsize=0;
else
xFontsize=2.5*get(curXlabel,'fontsize');
set(curXlabel,'horizontalalignment','center');
set(curXlabel,'verticalalignment','bottom');
if(xFontsize<20);
xFontsize=20;
end
end
curYlabel=get(curHand,'ylabel');
if(isempty(curYlabel));
yFontsize=0;
else
```

```
yFontsize=3*get(curYlabel,'fontsize')
set(curYlabel,'horizontalalignment','center');
set(curYlabel,'verticalalignment','top');
if(yFontsize<20);
yFontsize=20;
end
end
% Calculate new sizes of margins and plot area size.
leftSide=(yFontsize+yaxisMarks)/curSize(1)
rightSide=marginX/curSize(1);
plotWidth=1-leftSide-rightSide;
bottomSide=(xaxisMarks+xFontsize)/curSize(2);
topSide=(marginY+titleFontsize)/curSize(2);
plotHeight=1-bottomSide-topSide;
% Set new size.
set(curHand,'position',[leftSide bottomSide plotWidth plotHeight]);
% If the title is set, adjust the title position so that it's centered
% between the top of the window and the top of the plot area.
if(titleFontsize>0);
titleTop=1-topSide/2;
set(curTitle,'position',[0.5 titleTop]);
end
```

Appendix B: The Histogram Function

As of this writing, there is a problem with the **hist** function. It does not appear to operate correctly. Until it is corrected, you can use this function, saving it as **hist.m** in one of the folders in your path. It should provide the proper functionality as described in the Freemat Documentation.

```
% This function calculates the histogram of an array
% (input_array). The variables used are:
% bins = either the number of bins (if a scalar) or
%        the center of the bins (if a vector)
% normal = normalize the values such that the sum of all of
%          the bins equals this value.
% partial = a string that is either 'partial' or 'full'.  The
%           default is 'full'.  In this case, any values outside
%           of the limits of the bin values will be lumped into
%           then ends.  If the string is 'partial', those values
%           outside of the ends will not be counted.
function [rv r]=hist(input_array,bins,normal,partial)
if(nargin>4);
printf('ERROR: The hist function requires no more than 4
arguments.\n');
return
end
if(nargin<4)
partial='full';
end
if(nargin==1);
bins=10;
normal=0;
max_amp=max(input_array);
min_amp=min(input_array);
bin_val=linspace(min_amp,max_amp,bins+1);
w=bin_val(2)-bin_val(1);
r=min_amp+w/2:w:max_amp-w/2;
elseif(nargin==2);
normal=0;
s=size(bins);
if(s(1)>s(2))
bins=bins';
end
if(sum(diff(diff(s)))>0)
printf('The bin array must be equally spaced.\n');
return
end
if(isscalar(bins))
max_amp=max(input_array);
min_amp=min(input_array);
bin_val=linspace(min_amp,max_amp,bins+1);
w=(max_amp-min_amp)/bins;
r=(min_amp+w/2):w:(max_amp-w/2);
else
r=bins;
w=bins(2)-bins(1);
max_amp=max(bins)+w/2;
min_amp=min(bins)-w/2;
```

```
bins=length(bins);
bin_val=linspace(min_amp,max_amp,bins+1);
end
elseif(nargin>=3);
s=size(bins);
if(s(1)>s(2))
bins=bins';
end
if(sum(diff(diff(s)))>0)
printf('The bin array must be equally spaced.\n');
return
end
if(isscalar(bins))
max_amp=max(input_array);
min_amp=min(input_array);
bin_val=linspace(min_amp,max_amp,bins+1);
w=(max_amp-min_amp)/bins;
r=(min_amp+w/2):w:(max_amp-w/2);
else
r=bins;
w=bins(2)-bins(1);
max_amp=max(bins)+w/2;
min_amp=min(bins)-w/2;
bins=length(bins);
bin_val=linspace(min_amp,max_amp,bins+1);
end
end
for(k=1:bins)
t=(input_array>bin_val(k)).*(input_array<(bin_val(k+1)));
cdf_array(k)=sum(t);
end
if(strcmp('full',partial))
t=input_array<bin_val(1);
cdf_array(1)=cdf_array(1)+sum(t);
t=input_array>bin_val(bins+1);
cdf_array(bins)=cdf_array(bins)+sum(t);
end
if(normal>0);
rv=cdf_array*normal/length(input_array);
else
rv=cdf_array;
end
```