The Freemat Primer

By
Gary Schafer

Ver. 1.0
January 2009

Table of Contents

About the Author

I am an electrical engineer. I know a little bit about computer programming (just enough to be trouble) and mathematics (more than enough to cause trouble).   My main emphasis is digital signal processing and radio-frequency (RF) signals. ***If you're looking for help in complex programming using Freemat, this is not the place.*** This is a basic book that I started putting together for my own use, but then realized that others might find it helpful.

Any suggestions for improvements, corrections, or the like can be sent to garystar1 at comcast dot net.

This book is not exhaustive. I don't know how to use every, single button, icon, command, switch, dial, gauge, and lever within Freemat. If you are looking for a book with details of all Freemat commands, you will want the *Freemat v3.6 Documentation* by Samit Basu. It is available for free online at http://freemat.sf.net/FreeMat-3.6.pdf.

User Assumptions

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

How I Put this Book Together

I used Freemat Ver 3.6 running within Microsoft XP (desktop system) or MS Vista Basic Home Edition (laptop). The book was written with OpenOffice Writer 2.4.1, while I used Jasc's Paint Shop Pro Ver 9 for the graphics.

While Freemat on Linux platforms, such as Ubuntu, this workbook was put together for the Windows version of Freemat.

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 author is given credit where due.

Using with Freemat v3.6 Documentation

Samit Basu, the man who wrote Freemat, created a document, the *Freemat v3.6 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 3.6

This is how Freemat looks when you first start it up (Figure 1).
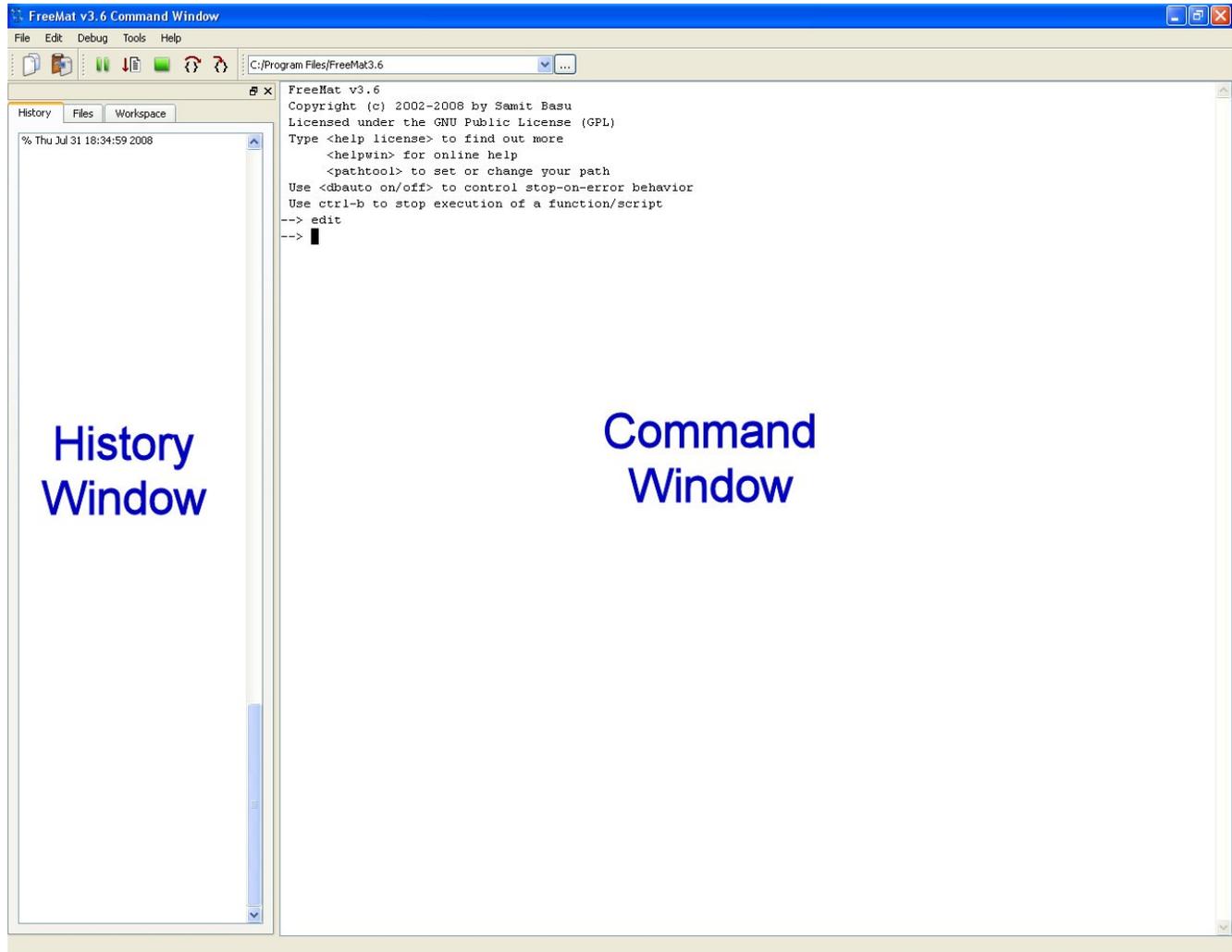
Figure 1: Main Freemat Window

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 3.6 is shown below.

Example - Beginning lines of Freemat 3.6

```
FreeMat v3.6
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
-->
```

We'll be spending most of our time covering the Command Window. This is where the action happens. However, let's look briefly at the other windows.

Topic 1.1.1: The History Window

The history window 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 1.1.2: The Files Window

Figure 2: Files Window

Click on the "Files" tab. This window shows a listing of all files within the current directory. See Figure 2. Typically, this directory will be the "Freemat" directory under "Program Files". This is the directory into which all of the files necessary to run Freemat are stored. In this example, I've changed it to a special directory into which I store all of my Freemat-related files.

You can use this window to enter a filename by simply double-clicking on that file.

Note: The list in the Files Window is only updated when Freemat is started. Any additions or deletions to the directory will not be shown in Freemat until the next time you start the program.

Warning to MS Vista Users: There are issues saving files into the *Program Files* directory. If you are running Windows Vista as your operating system, I recommend that you jump to *Topic 3.1: Setting the Working Directory for Saving Files*. Then come back here.
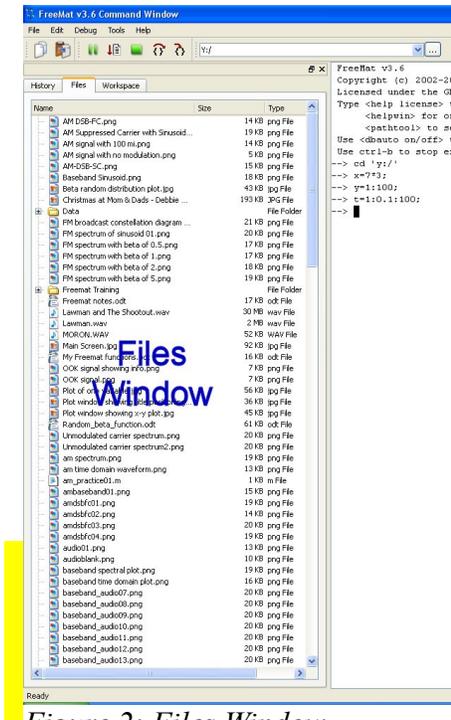
Topic 1.1.3: The Workspace Window

Figure 3: Workspace Window

Click on the "Workspace" tab. This window shows all variables and, for single variables, it shows their values. See Figure 3. Frankly, I keep this window open more often than the others. I find this window the most useful. It helps me to confirm variable values and tells me when a matrix might be the wrong size for operating (adding, multiplying, etc) with another matrix.

You can erase all variables and their values by typing "clear all" at the command prompt and pressing <Enter>.

Topic 1.2: 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

At the command prompt, type "3 +2" and press <Enter>.

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

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.

Topic 1.3: 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, especially if this were part of a function or script. I recommend one line per command just to keep things neat and tidy.

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.
- Type "who" onto the command line and press <Enter>. *NOTE: This only shows the variable name, type and array size. It does not show the value of the variable itself.*

Example - Viewing a Variable Value using the Command Line

```
--> x=7*3;
--> x
ans =
21
--> who
Variable Name     Type    Flags          Size
         ans    int32                    [1 1]
           x    int32                    [1 1]
```

Topic 1.4: How Many Decimal Points Do You Want?

Freemat uses the **format** command (*FD*, p. 389) 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 1.5: 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*.

A variable can store numbers of many different types (see *Topic 1.6: Variable Types*) as well as text.

A quick note on the allowable length of a variable name. In Matlab, a variable can be up to 19 characters long. 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.

Topic 1.6: 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 are:
- **int8** - a signed, 8 bit integer. Allowable values are -128 to 127.
- **int16** - a signed, 16 bit integer. Allowable values are -32,768 to 32,767.
- **int32** - a signed, 32 bit integer. Allowable values are -2,147,483,648 to 2,147,483,647. *Default Freemat type for integers.*
- **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 x 10^{38} to 3.4 x 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 x 10^{308} and slightly more than 1.79 x 10^{308}. This is also called a *double precision floating point number. Default Freemat type for floating point numbers.*
- **complex** - a signed, 32 bit complex floating point number. I *believe* that the values are 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*.

Topic 1.7: The Variable "Ans"

You may have already noticed a variable, "ans". This variable is used by Freemat to display results. This can be when you want to display a variable, or if you perform a calculation without storing it to a variable. However, if you specify a variable, such as "x = 1+1", then the result of "1+1" will be stored in "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.**

"Ans" can also be used for calculations. For example, if you perform a calculation without storing the result in a variable, the result is stored in "ans". Then you can use "ans" as a variable in any follow-on calculations. If you do not store the results in the follow-on calculations, the results are again stored in "ans".

A way to view all of the variables currently in use is the **who** (*FD*, p. 267) function. Simply type *who* at the command prompt and hit <Enter>.

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

Note that the **who** function provides all of the information on each variable. In this case, only one variable has been defined. If there had been more than one, it would be listed here as well.

Topic 1.8: 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.

Example - The Built-In Variables

```
--> format long
--> e
ans =
```

```
2.71828182845905
--> i
ans =
0.0000000 + 1.0000000i
--> inf
ans =
1.#INF0000000000
--> pi
ans =
3.14159265358979
```

Topic 1.9: 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.sf.net/FreeMat-3.6.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 30º 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.

- <u>Does the function already exist?</u> Maybe. Again, check the documentation. Samit Basu and his colleagues have done a masterful job of putting a lot of the commonly-needed functions into Freemat. But I imagine that they have day jobs; it would take several people of their caliber working more than full-time to put them all in there. And making your own is not difficult. That will be for later. *Topic 8: Scripts & Functions*, to be precise.

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
```

---

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.

```
ans =
 19
--> (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.

Topic 2.3: Cumulative Sum, Products and Factorials

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

- **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. 219)
- **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. 217)
- **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. 182)

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.

Example - Raising a Number or Variable to a Power

```
--> x=5.6;
--> x^2.5
ans =
74.2113
--> 8^3
ans =
512
```

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

- **exp(x)** - This is exactly the same as if you had typed e^x. Since Freemat provides *e* as a standard variable, I've found its easier to simply type *e^x* than *exp(x)*. Oh, and in case you're worried whether there are differences, there are. But I've only seen a difference in the very last digit of the double type variables (14 digits) I've tested with. Not being a mathematician, I don't know which one is correct. I just go with the assumption that they're both the same. (*FD*, p. 180)
- **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. 181)

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. 186)
- **log2(x)** - calculates the log, base 2, of the variable x. (*FD*, p. 187)
- **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. 185)
- **log1p(x)** - calculates the logarithm, base e, for the argument plus one. In other words, calculate the natural logarithm for x+1. (*FD*, p. 187)

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 3: Setting Up the Freemat Environment

Before we get too deep into Freemat, let's set a few things up.

Topic 3.1: Setting the Working Directory for Saving Files

Figure 4: Working Directory box at top of Freemat window
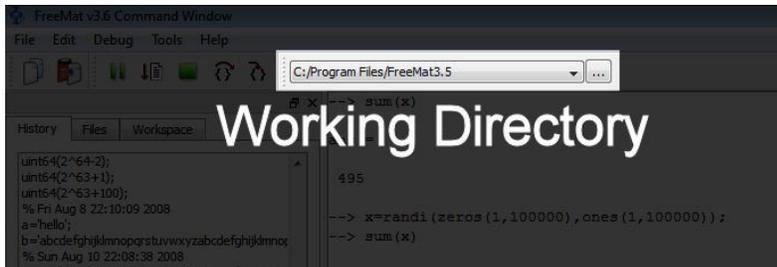
To be truly useful, Freemat needs to work with data. Whether its scripts, functions, images, or other data, Freemat needs someplace to put this stuff. When Freemat is first installed in Windows, all data is stored in the *Freemat* folder under the *Program Files* directory. This is shown in the small box at the top of the Freemat

window. See Figure 4.

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 and writes, you should take heed and change the working directory to something *other* than the *Program Files* directory.

WARNING TO VISTA USERS

Since WinVista protects 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 <u>that funny "Cancel or Allow" Mac commercial</u> was just being sarcastic? No. They were serious. Unfortunately, neither Freemat nor Vista 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, change your working directory to some folder within your "User" folder. You'll be very glad you did!

You have two options. First, change your working directory. This is the one I recommend. Keeps things simple. To do so, use the **cd** (change directory) command (*FD*, p. 453).

NOTE: As of this writing, I have not been able to use the browse button next to the Working Directory box to actually change my working directory. This applies to both the WinXP and WinVista. Every time I've tried, I get the following error: *Error: cd function requires exactly one argument*. If you have the same problem, you'll have to change your working directory manually.

To change the working directory, type:

cd 'full path name'

where 'full path name' is just that, the full working path of your desired directory. For example, I created a folder under "My Documents" called "Freemat". To make it the working directory, I'd use the following command:

```
cd 'C:\Documents and Settings\Owner\My Documents\Freemat'
```

If you're afraid of mis-typing the full path (and who wouldn't be?), you can use Windows Explorer to help out. Open up Windows Explorer, click around to open the directory where you want to store your files, then look in the Address Bar. See Figure 5.
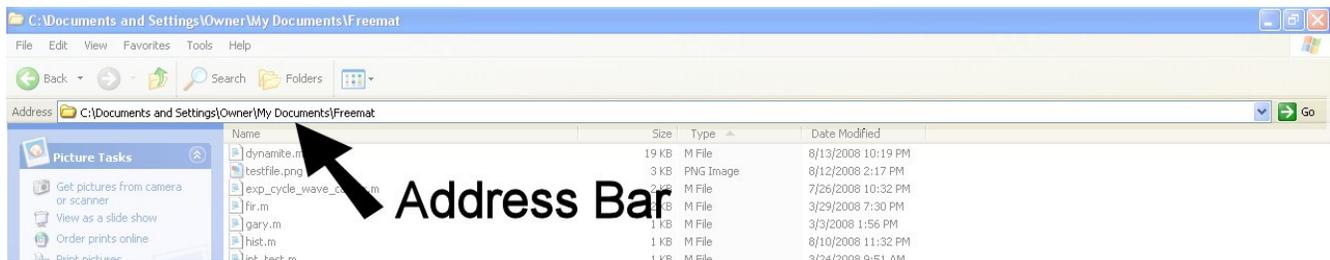

Figure 5: Windows Explorer showing Address Bar

If you do not see the Address Bar in the Windows Explorer window, click **View -> Toolbars** and click on **Address Bar** so that it has a check mark next to it.

With the Address Bar showing:

- Highlight the text in the Address Bar (which is your desired path name)
- Click **Edit -> Copy** (or just hit *Ctrl+C*)
- Switch over to the Freemat window
- Put your cursor wherever you want to enter the path name (the Command Window, the Editor Window, wherever)
- Press *Ctrl+V* to copy the path name into Freemat.

If you don't want to change the working directory, your second option is to include the full path in the command needing the filename. As an example, look at the **print** command (*FD*, p. 520) used to save plots as images. Rather than just the filename, I could include the path and file name in the **print** command. This would look like:

```
print('C:\Documents and Settings\Owner\My Documents\Freemat\testfile.png')
```

Adding the full path name along with the filename can be unwieldy. But if you want to make absolutely, positively certain that your file is stored in a particular directory, this is the way to go.

One, last thing about the working directory. Freemat will not remember it when you close the program. The next time you start Freemat, you're back to the Program Files/Freemat folder as your working directory. I have three remedies.

First, create a one-line script that changes your working directory. (See *Topic 8: Scripts & Functions*). The entire script will be the one line **cd** (change directory) command with the path name of your desired working directory. Store this script into your primary Freemat programs folder (typically the already-mentioned *C:/Program Files/Freemat* folder. When you first start up Freemat, run this script and (Voila!), your working directory will be changed to something of your choosing.

Changing the Working Directory for Windows Vista Users

Windows Vista and Freemat are not incompatible. But Vista has thrown in a few twists for... fun. Here's my recommendation for changing the working directory for Vista users. This will be a two-step process.

Step #1. Make a one-line **cd** script. When you save the script, save it to your User directory. Why? Because Vista does not allow Freemat to save files to the Program File directory, or any of its sub-directories.
Step #2. Use Windows Explorer to move (or just copy) the file from your User directory to your Freemat directory within the Program Files folder.

Now, when you first run Freemat, it will look for scripts within the Freemat directory within the Program Files folder. As the first command you run when you open Freemat, type the name of your script. Mine, for example, is called *workdir*. It changes my working directory to one within my User directory, not in my Program Files folder.

Second, put a *cd* command at the beginning of each script you run. Slows down the execution a little bit, but it's only necessary if you plan on saving any data as part of the script. And it will also ensure your data from that script goes to the desired folder.

My third, and last, suggestion has already been mentioned. If you save any data (**print**, **wavwrite**, etc), include the full path in the filename.

Topic 3.2: Setting the Path for Retrieving Data Files

Retrieving or reading data files is different than saving or storing them. When Freemat saves a data file, it wants to go to the working directory by default. However, when you want to *retrieve* a file, Freemat will look at your working directory and any folders listed in your Path Tool. This includes any data files, images, scripts, or functions you may have written.

To set the path for Freemat, click on **Tools -> Path Tool** in the Freemat menus. This will open a window, as shown in Figure 6. Use the directory tree on the left of the window to select the folder that
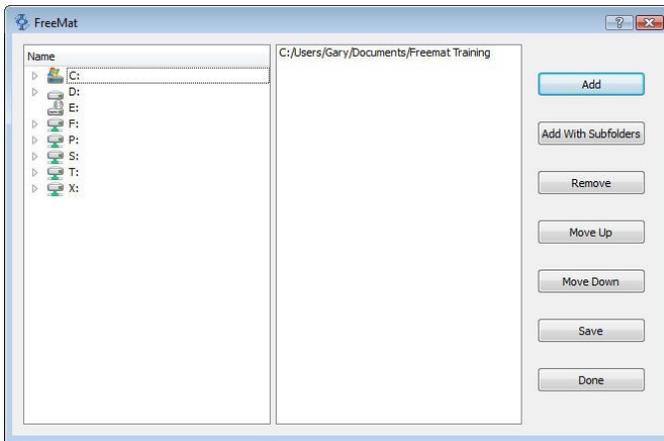
Figure 6: Path Tool window

Figure 7: Path Tool window with folder added

you want to include in the path. Then either click the "Add" or "Add With Subfolders" button.

Figure 7 shows the Path Tool window with a new folder added. In this case, I selected the "Documents" folder under the "Users" directory. Then, I clicked the "Add" button.

Topic 3.3: Using a Fixed Width Font in the Freemat Editor

The first time that you open the Editor Window in Freemat, you may see a warning that you should use a fixed width font. Why? I have no idea. The type and size of the font makes absolutely no difference to how it is executed. To stop seeing this message, change the font type. To change the font, click on the *Edit* menu, followed by *Preferences -> Font*. A window will open that allows you to change the standard font. The one I use is *Courier New.* And use a size that is easy for you to read.
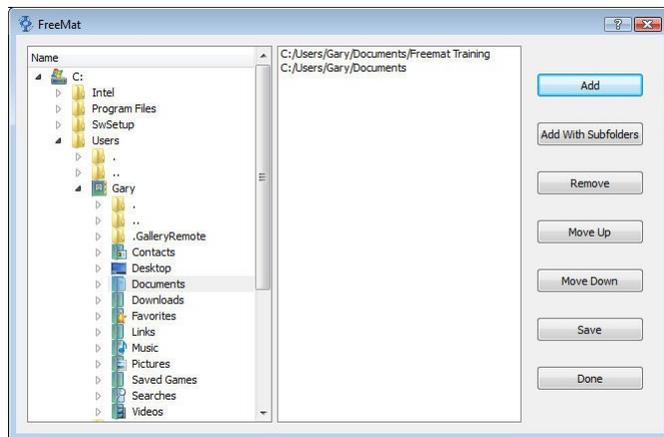
Topic 4: Working with WAV Files

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 4.1: Reading a WAV File Size

NOTE: As of this writing, any .wav files to be read <u>must</u> 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. 411). 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 4.2: 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 = the audio file as listed in the directory.

NOTE: In the Freemat Version 3.6 Help directory, the example used does not show using the ' marks. To use this function properly, you must use these marks otherwise you will get an error when attempting a wavread() function.

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 Monophonic WAV File

I'll use a monophonic WAV file, **myaudio_mono.wav**. It's a monophonic audio file of a human voice.

```
--> y=wavread('myaudio_mono.wav');
--> plot(y)
```
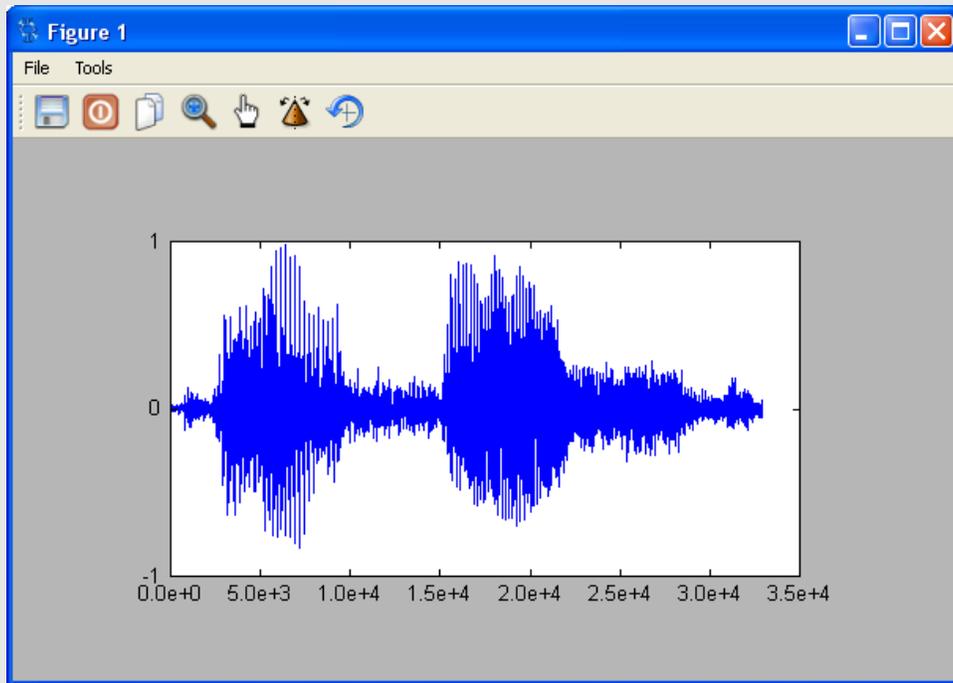
*Figure 8: Plot of monophonic audio file, **myaudio_mono.wav**. This is a WAV audio file of human voice. Note that the amplitudes are normalized to fall within the range of -1 to 1.*

Example - Reading from a Stereo WAV File

I'll use the WAV audio file from the previous example named **myaudio.wav**.

```
--> y=wavread('myaudio.wav')
```

The next line pulls out the left channel (column 1) in order to display it.
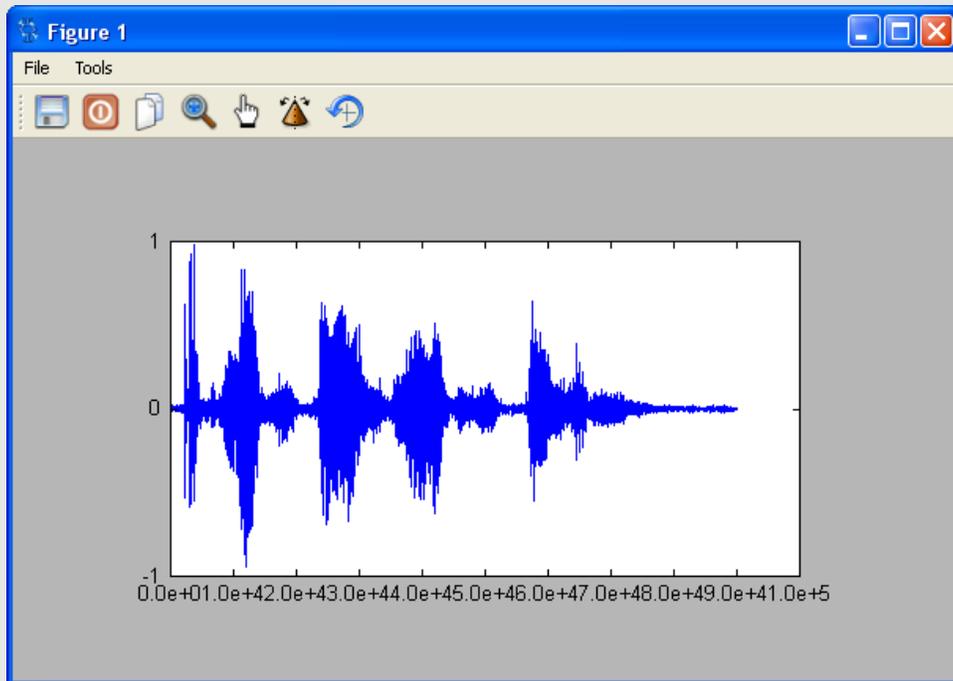
```
--> L=y(:,1);
--> plot(L)
```

*Figure 9: Plot of left channel of **myaudio.wav**, a stereo audio file of human voice. Again, note how the amplitudes are normalized to fall within the values of -1 to 1.*

Topic 4.3: 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

Here's an example showing how to read in the sample rate, then create a time axis for the plot.

```
--> [y,sample_rate]=wavread('myaudio.wav');
--> sample_rate
ans =
  44100
--> L=y(:,1);
```

The next line creates a time counter in seconds. It uses the function **length(x)**, which returns the length of the matrix **x**. By dividing the counter by the sample rate, the counter becomes time in seconds.

```
--> t=(1:length(L))/sample_rate;
--> plot(t,L)
```
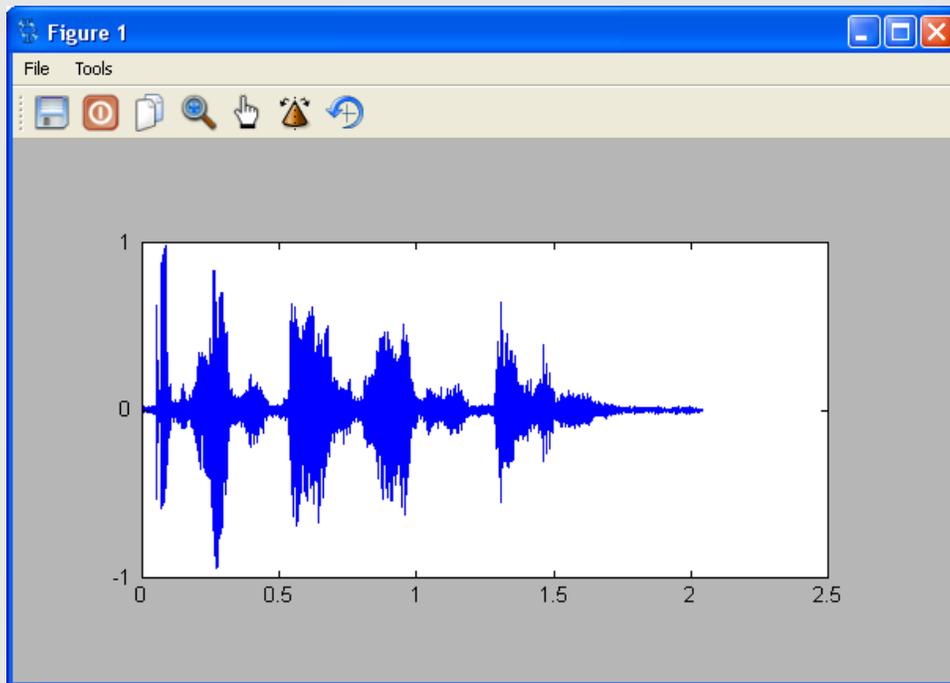

*Figure 10: Plot of left channel of **myaudio.wav** file with time (in seconds) on horizontal axis.*

Example - Read the Bit Depth

Remember: In order to read the bit depth, you have to also read in the sample rate.

This example reads in the bit depth and sample rate from **myaudio_mono.wav**. The sample rate is read in as 44100 Hz (44.1 kHz) and the bit depth is 16, meaning 16 bits are used to digitize each sample.

```
--> [y,sample_rate,bit_depth]=wavread('myaudio_mono.wav');
--> sample_rate
ans =
  44100
--> bit_depth
```

```
ans =
 16
--> t=(1:length(y))/sample_rate;
--> plot(t,y)
```
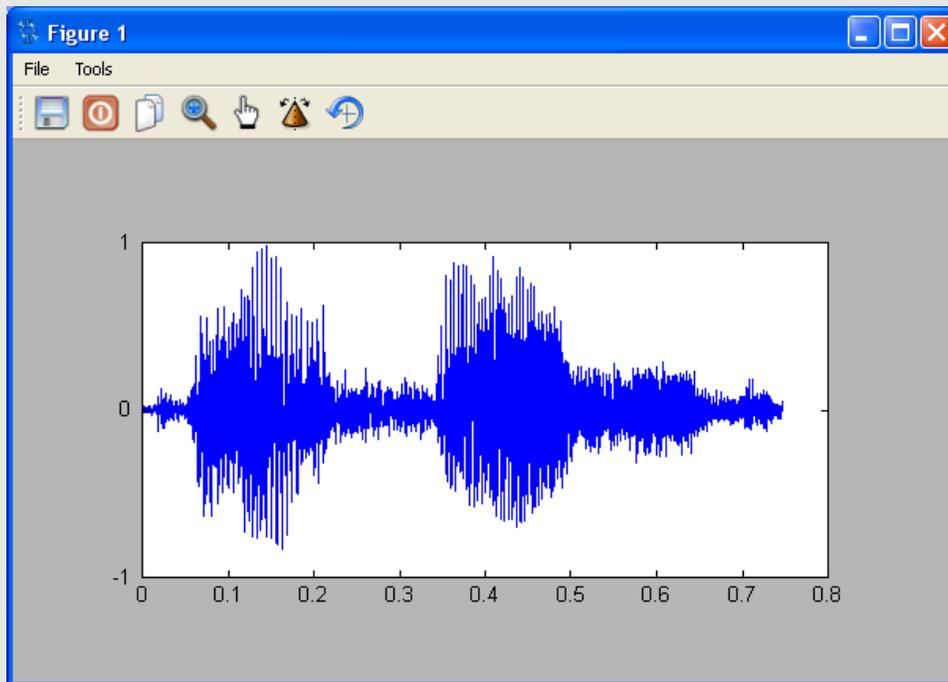

*Figure 11: Plot of mono signal **myaudio_mono.wav**, which is has a sample rate of 44.1 kHz and a bit depth of 16 bits. The horizontal axis is time in seconds.*

Topic 4.4: Reading Only a Portion of a WAV File

Freemat provides handles that allow you to read in only a portion of a WAV file. The two syntaxes are:

y=wavread('filename.wav', [start stop])
y=wavread('filename.wav', samples)

where:
y = variable used to store samples from WAV file.
start = sample number at which to start reading the WAV file.
stop = sample number at which to stop reading the WAV file.
samples = number of samples, starting at the beginning of the WAV file, to import.

Example - Reading in Only a Portion of a WAV File

In this first example, I'll read in only 8192 samples starting from the beginning of the file.

```
[y,sample_rate]=wavread('myaudio_mono.wav',8192);
t=(1:length(y))/sample_rate;
plot(t,y)
```
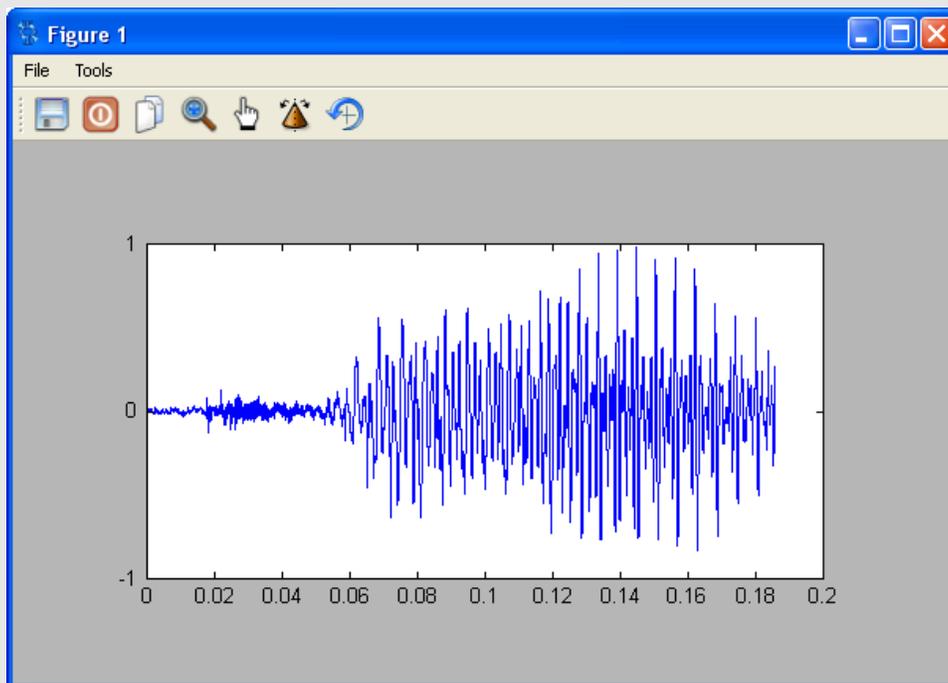

Figure 12: Plot of portion of WAV audio file with only 8192 samples.

For the next example, I'll read in 8192 samples starting at sample 6500.

```
[y,sample_rate]=wavread('myaudio_mono.wav',[6500 (6500+8192)]);
t=(1:length(y))/sample_rate;
plot(t,y)
```
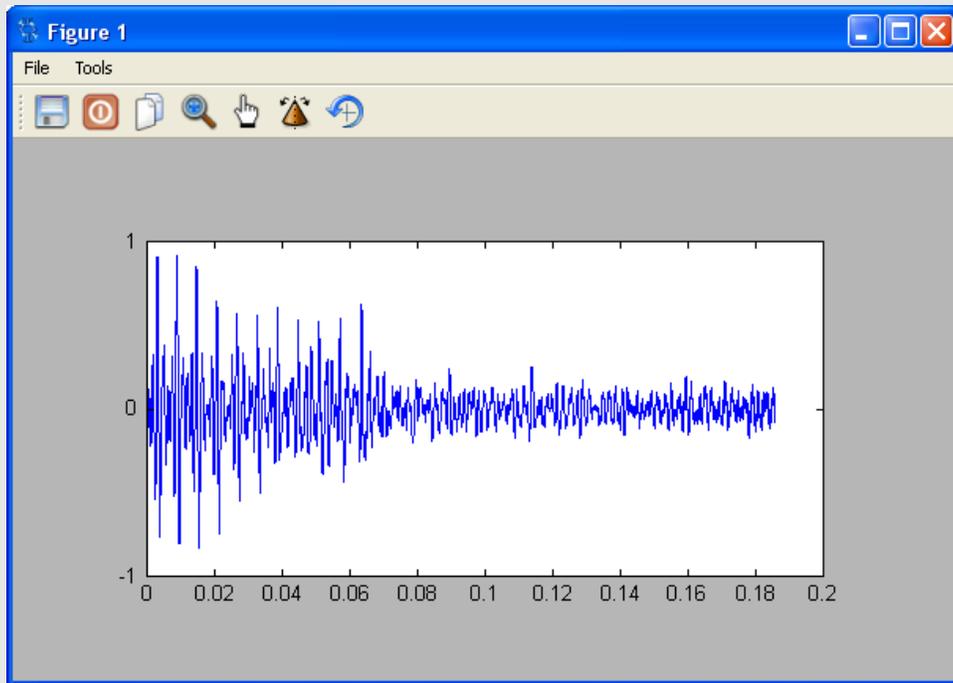
Figure 13: Plot of WAV file with 8192 samples imported starting at sample 6500.

Topic 4.5: 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 will need to change the working directory to something other than your Program Files folder. Otherwise, you won't be saving your files. They will simply not be stored. See *Topic 3.1: Setting the Working Directory for Saving Files*.

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 5: Plots

The default plot function provides a blue colored, solid line plot in a window that is 600 x 400. The area minus the toolbars is 600 x 344. The actual plot itself, which I call the *plot area,* is roughly 420 x 210. The basic **plot** function (*FD*, p. 514) is as follows:

plot(y)

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

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. More on that later.

Topic 5.1: Plot of a One-Dimensional Array

Let's start by looking at the plot of a single variable, an array with one dimension.

Example - Making a Plot of a Single Variable

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

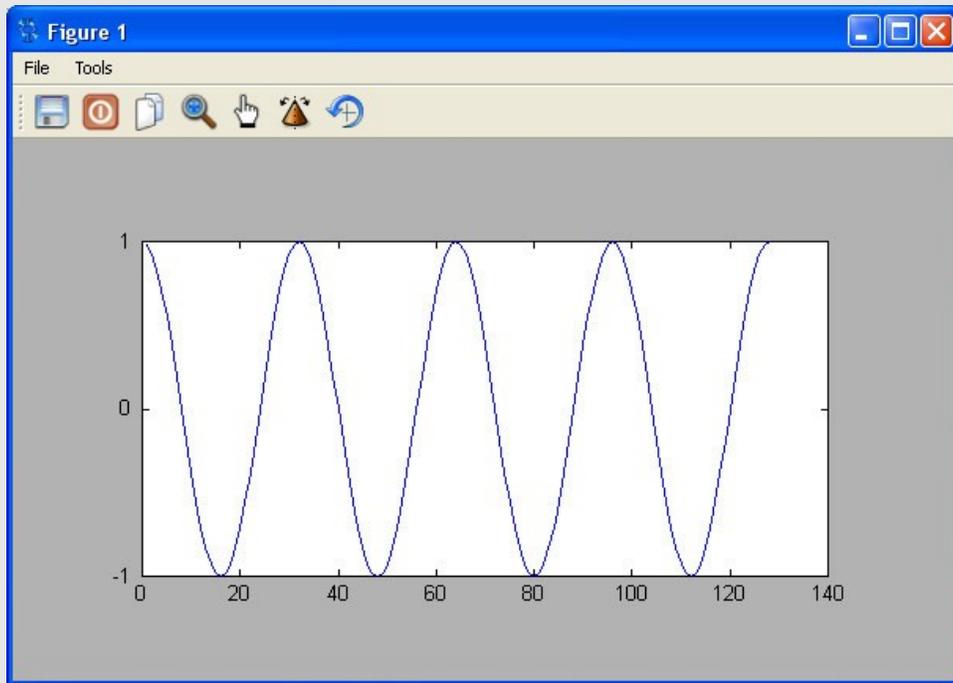The resulting figure is shown in Figure 14.

Figure 14: Plot of One Variable

Topic 5.2: Setting the Plot Color

The **plot** function comes with a few handles that you can set when you invoke the function itself.

NOTE: The only time possible to set the handles for plot color, line type, and point markers is when invoking the plot function. It is not possible to change these after the plot is created.

Simple colors can be added as a single-character handle for the function. These colors are:

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

NOTE: *Plot Colors* have to be used with a *Line Type* and/or a *Point Marker* (listed below); otherwise, your plot will be invisible. *Line Types* or *Point Markers*, however, do not require a *Plot Color*; they can be used individually.

Topic 5.3: Setting the Line Type

Freemat provides several different line types. 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 - Plots using Different Line Types

Using the following commands, or as a script, you get the plot shown in Figure 15. This is a plot using a green (the letter "g" in the **plot** function), dotted (the colon in the **plot** function) line.

```
t=1:256;
y=2*pi*t;
x=sin(y*1/100)-1/3*sin(y*3/100)+1/5*sin(y*5/100);
plot(x,'g:')
```
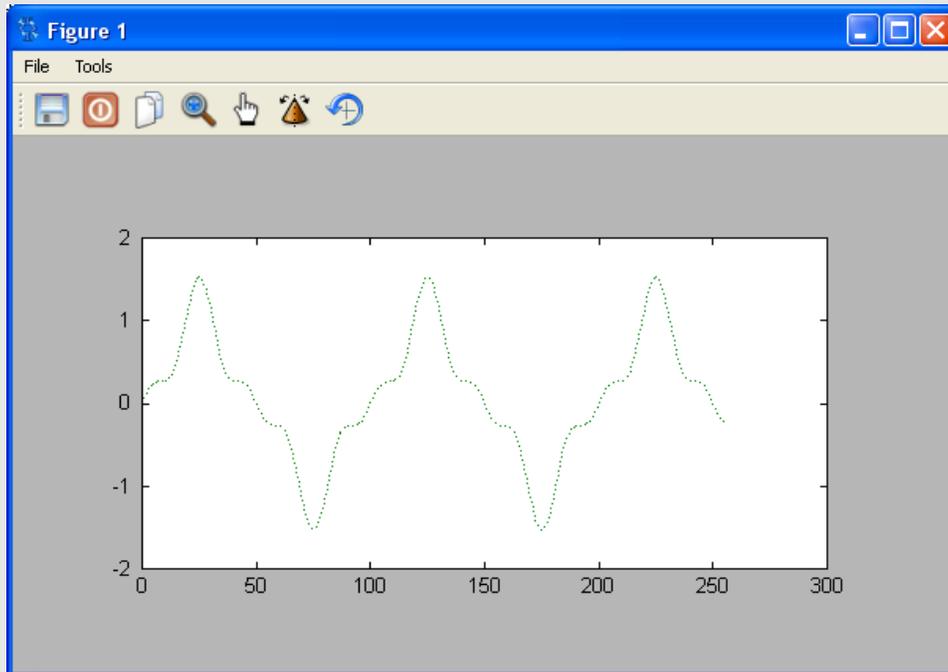

Figure 15: Plot using a green, dotted line

Or change the command to create a solid, black line, as shown in Figure 16.

```
t=1:256;
y=2*pi*t;
x=sin(y*1/100)-1/3*sin(y*3/100)+1/5*sin(y*5/100);
plot(x,'k-')
```
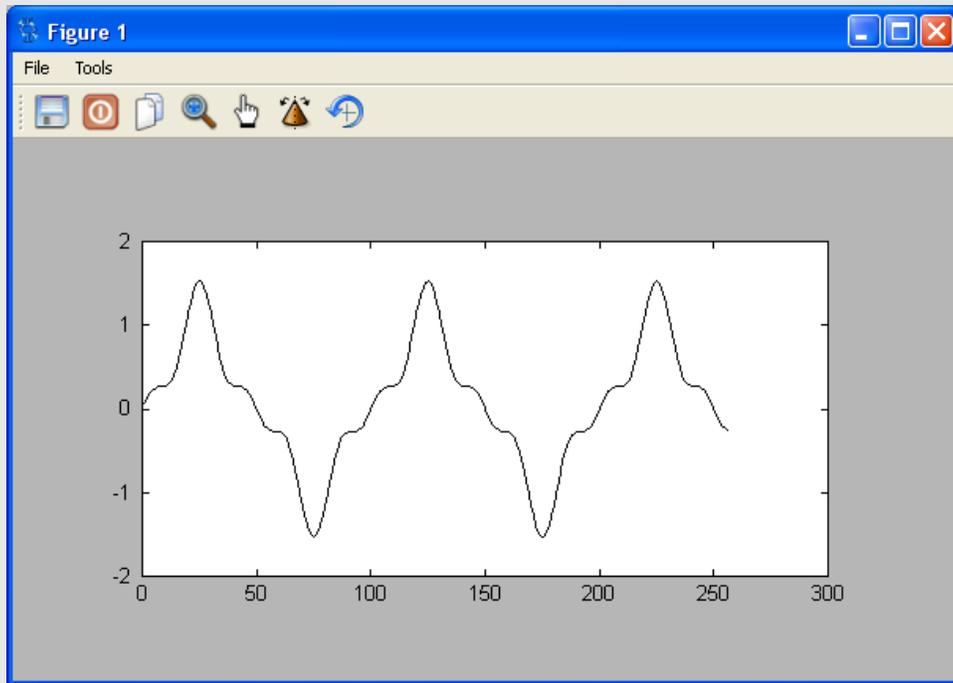
Figure 16: Plot using a black, solid line

Topic 5.4: 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 a solid, black line.

```
t=1:64;
y=2*pi*t;
x=cos(y*5/100);
plot(x,'k-')
```
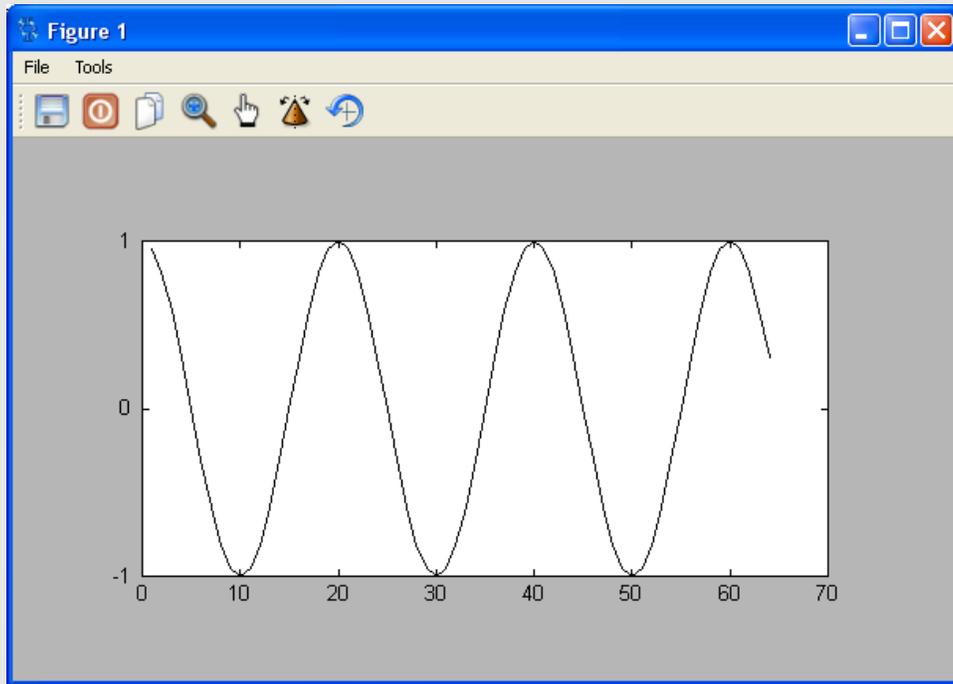

Figure 17: Plot with a solid black line

Next, we add square point markers (the added "s" in the plot function).

```
t=1:64;
y=2*pi*t;
x=cos(y*5/100);
plot(x,'k-s')
```
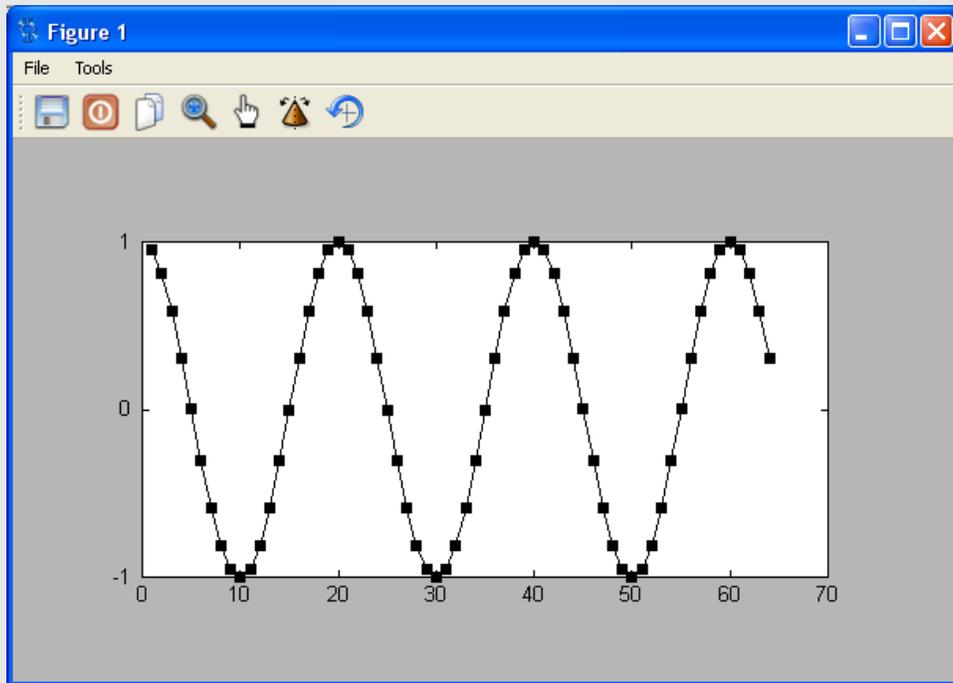
Figure 18: Plot with a solid, black line and square markers

Finally, create a plot just using square point markers. The plot is blue because of the fact that, unless explicitly stated, Freemat plots will be blue.

```
t=1:64;
y=2*pi*t;
x=cos(y*5/100);
plot(x,'s')
```
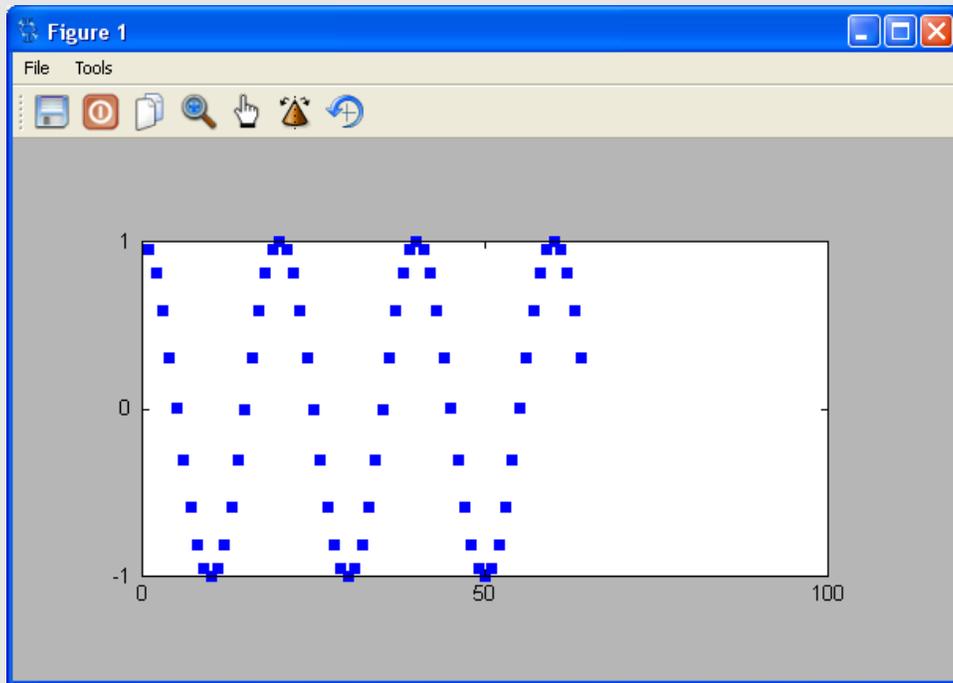
Figure 19: Square point marker plot of sine wave.

Topic 5.5: Making an X-Y Plot

If you want to plot one array against another, typically called an X-Y plot, use the following format:

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

Example - Making an X-Y Plot

The first example shows a cosine wave plotted on the horizontal axis against a sine wave on the vertical axis.

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

The result is shown in Figure 20.

Figure 20: Plot of an X-Y Graph

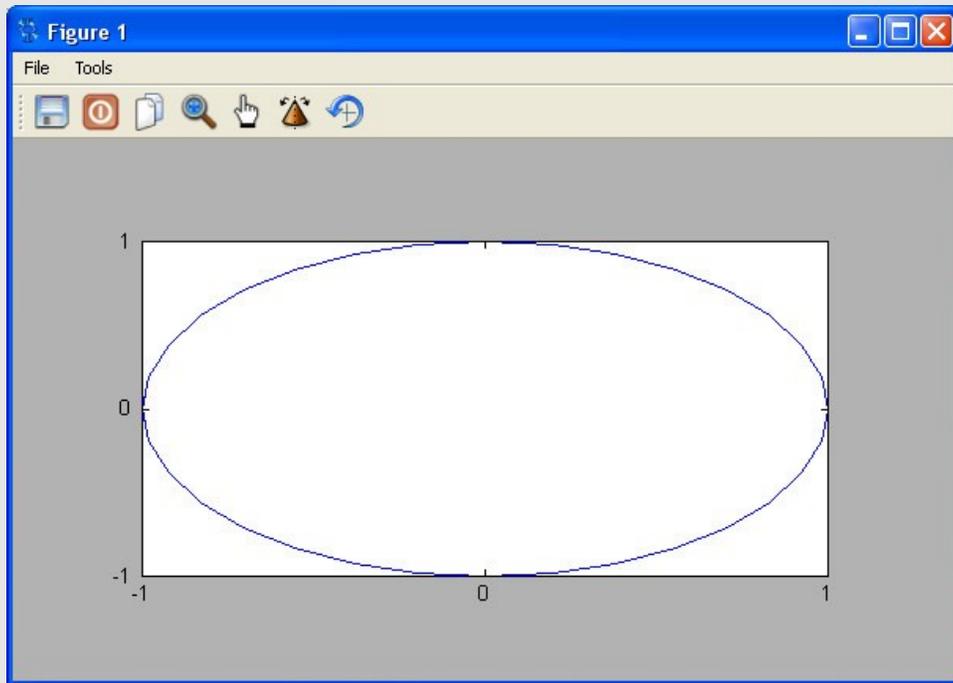Using a dot point marker type on an X-Y plot, we can create a scatter plot. In this case, we can see two, Gaussian random variables plotted against each other.

```
x=randn(1,10000);
y=randn(1,10000);
plot(x,y,'k.')
```
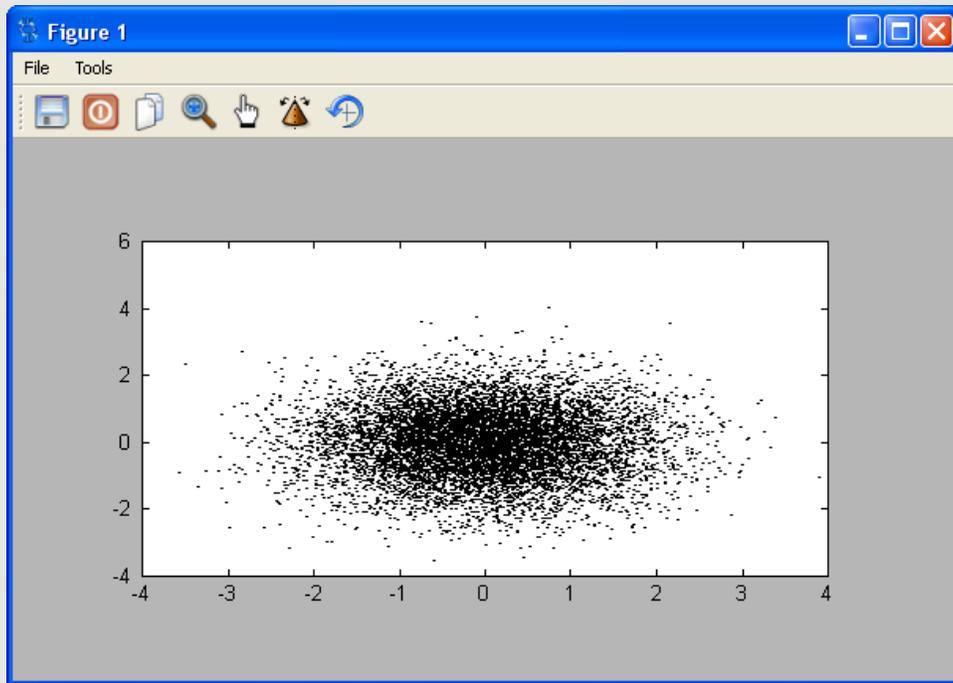
Figure 21: Scatter plot using black dot markers

Topic 5.6: Plots of Multiple, Independent Variables

To plot two or more independent variables, put a string containing a line color, line type, and/or point marker between the variables. This can be shown as follows:

plot(x,y): This will plot the variables in an X-Y plot.
plot(x,'-',y): This will plot the variables **x** and **y** both along the horizontal axis. Both will be solid lines. The **x** variable will be blue. The **y** variable will be green. (See discussion of plot colors below.)
plot(x,'s',y,'-',z): This will plot the variables **x**, **y**, and **z** along the horizontal axis. In this case, the **x** variable will be blue square markers, the y variable will be a solid green line, and the z variable will be solid red line.

 Assuming you do not explicitly tell it a color for the respective variable, Freemat will default to the following colors (in order) for the first five independent variables:
1. Blue
2. Green
3. Red
4. Cyan
5. Purple

Example - Plots of Multiple, Independent Variables

In the first example, we'll use the same script as used to create Figure 20, but with one difference. We'll

put a line type (in this case, a solid line "-") between the two, independent variables.

```
t=1:128;
x=cos(2*pi*t/32);
y=sin(2*pi*t/32);
plot(x,'-',y)
```
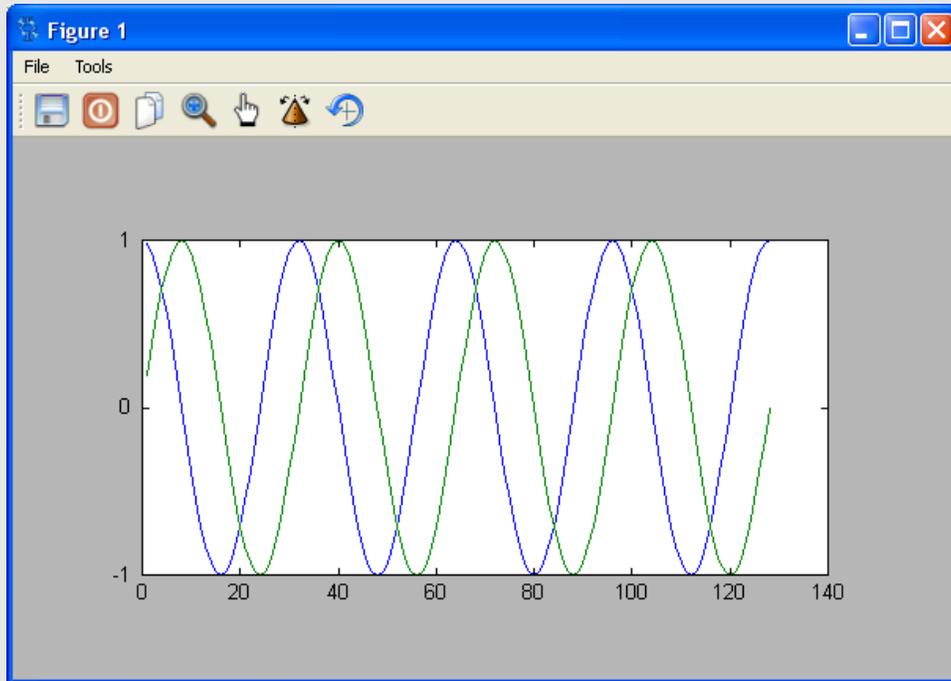

Figure 21: Plot of two independent variables using default colors

In this next plot, we'll create five different waveforms, plot each as a solid line (again, the "-" symbol), and use the default colors.

```
t=1:100;
w=2*pi*t;
x=cos(w*5/100);
y=0.5*sin(w*7/100);
z=0.7*cos(w*10/100);
a=0.3*abs(sin(w*3/100));
b=-0.3*abs(cos(w*3/100))-0.1;
plot(x,'-',y,'-',z,'-',a,'-',b)
```
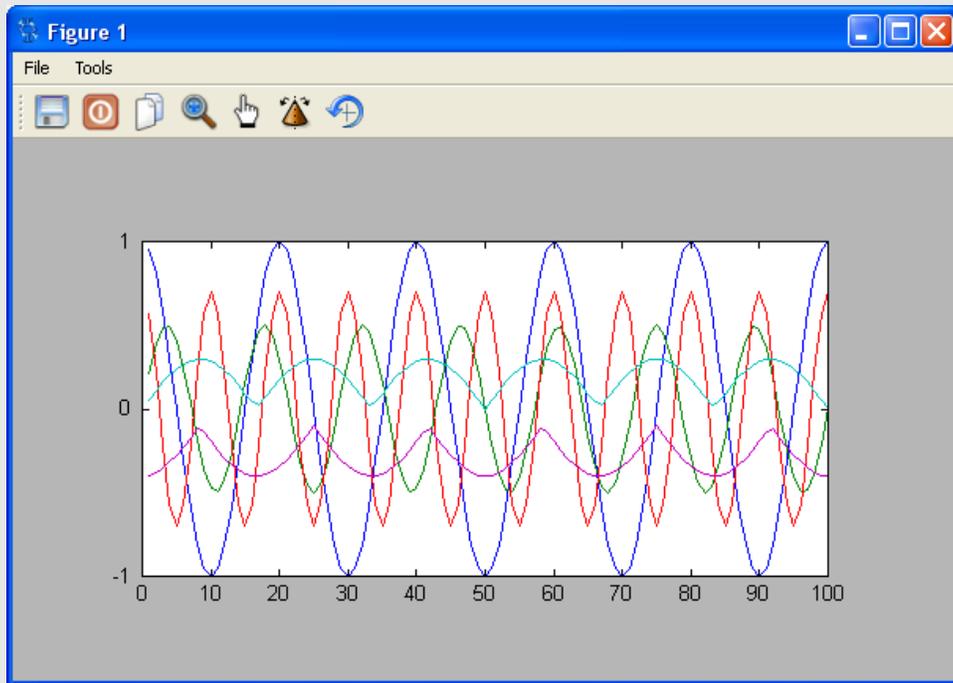
Figure 22: Plot showing five independent variables using default colors

Topic 5.7: 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. 542 & 545, respectively).

These functions are separate from the **plot()** function and are invoked *after* the plot 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 room for setting the horizontal and vertical limits.

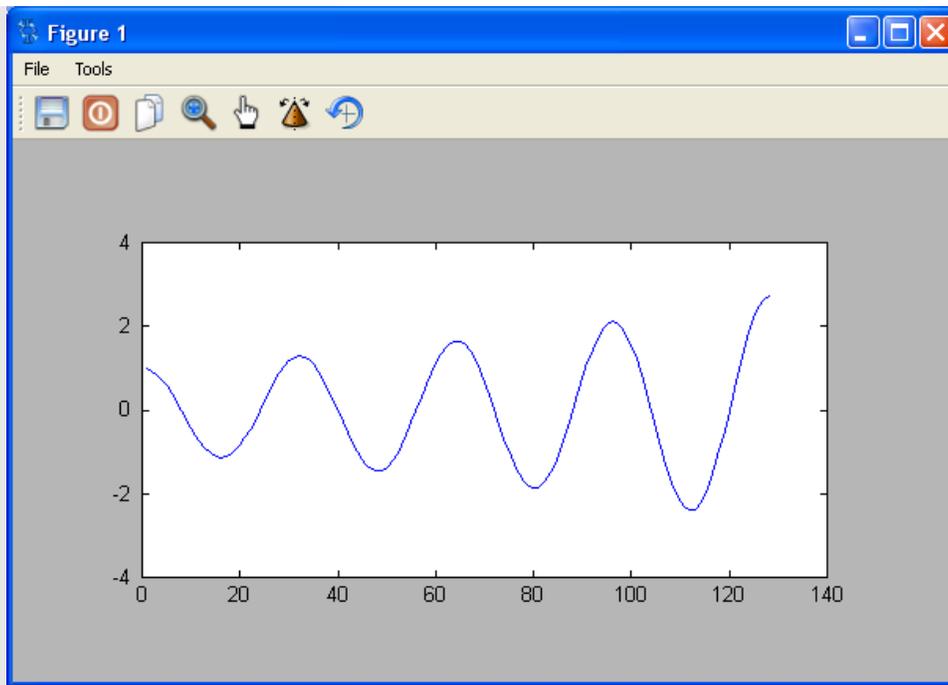```
t=0:128;
x=cos(2*pi*t/32).*exp(t/128);
plot(x)
```

*Figure 23: Plot that does not fill up the plot area.*

To fix this, we'll first set the horizontal limits. The result is shown in Figure 24:

```
xlim([1,128])
```
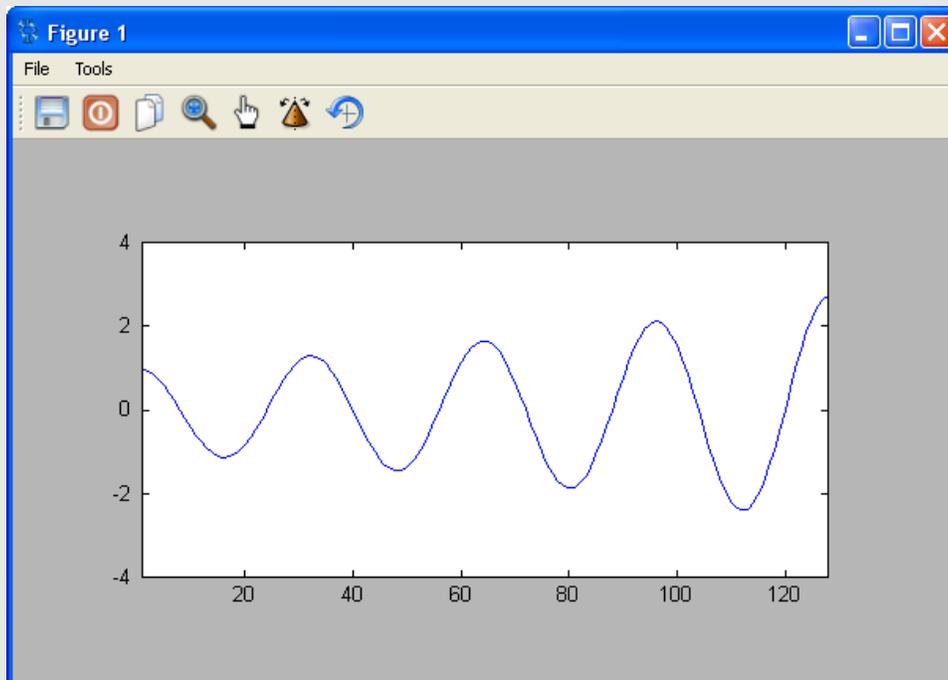

Figure 24: Plot with horizontal limits manually set

Finally, we'll set the vertical limits:

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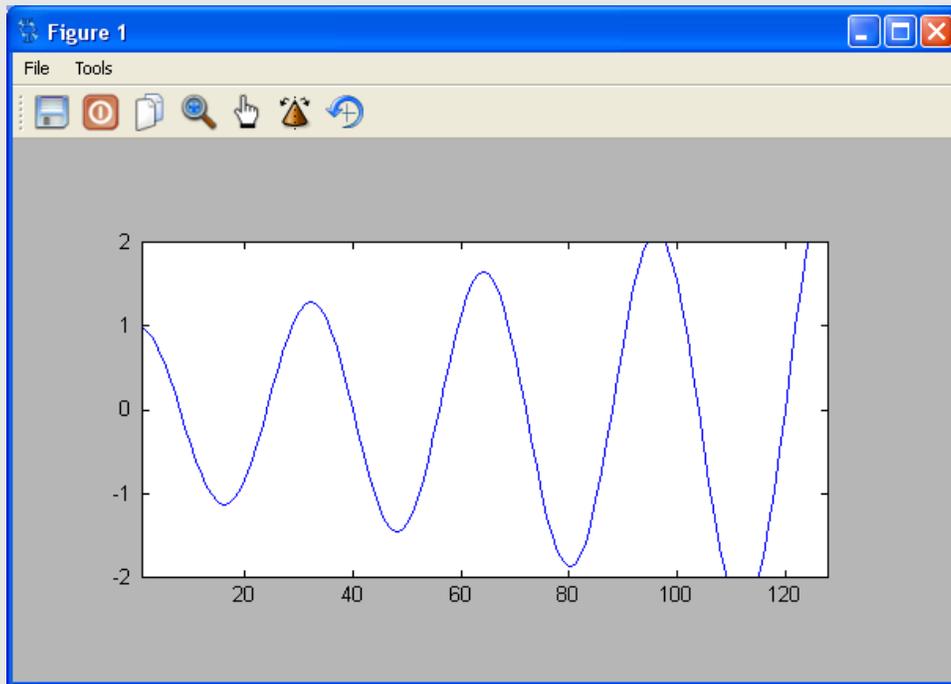

Figure 25: Plot after setting both the horizontal and vertical limits

Topic 5.8: Resizing a Plot

You can resize the plot using the **sizefig** function (*FD*, p. 524). It has 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. This is the same as using either the **xlim** or the **ylim** functions.

NOTE: The size of the actual plot area will be less than that stated in **sizefig**.

The plot area will be roughly 0.735 of the stated horizontal number and 0.67 of the stated vertical number. Therefore, to have the plot area be a certain size, you need to perform the calculations as follows:

entered_number_horizontal ≈ desired_horizontal_size/0.735

entered_vertical_size ≈ desired_vertical_size/0.67

Example - Resizing a Plot

In this example, we'll look at resizing a plot.

```
t=1:1024;
x=sin(2*pi*t/100);
plot(x)
```

These commands will create the plot shown in Figure 26.
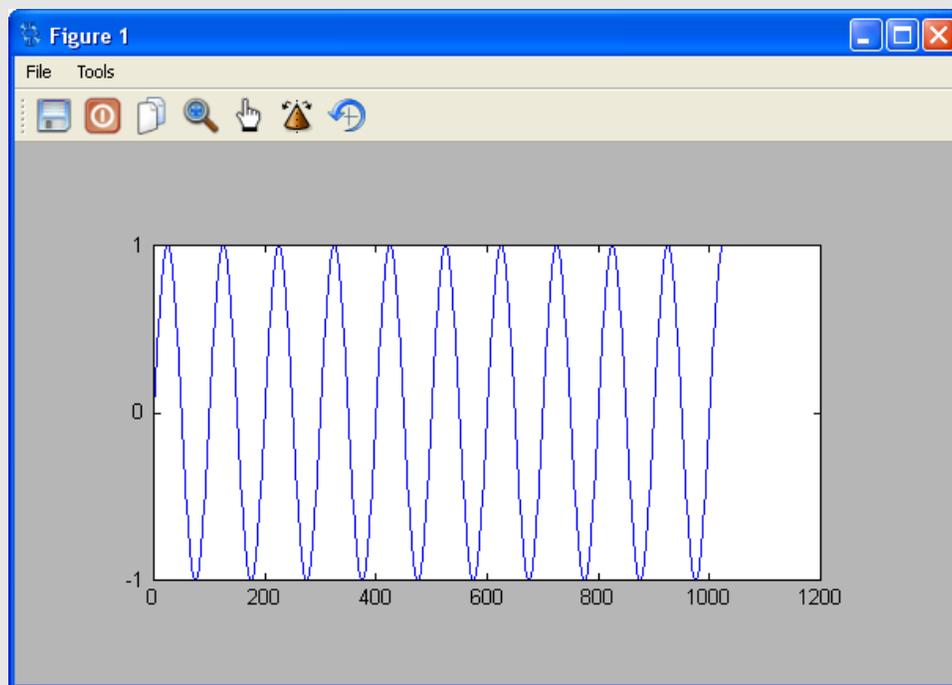
Figure 26: Plot shown in default size

Once we run the sizefig function, we get the plot shown in Figure 27.

```
sizefig(300,200)
```
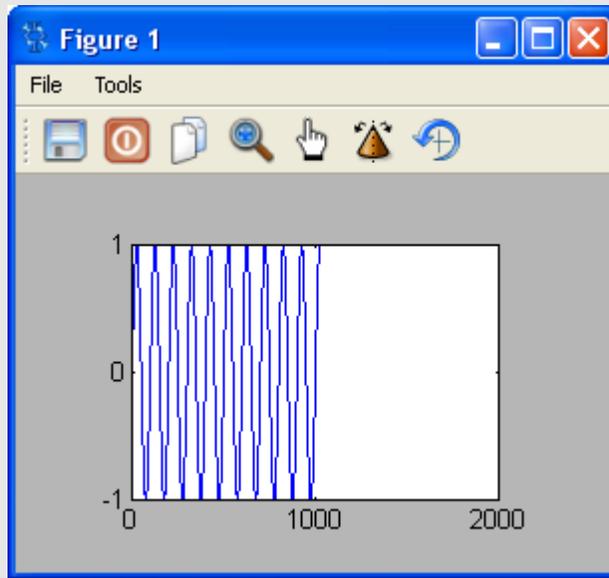
Figure 27: Resized plot at a smaller size

Topic 5.9: Creating a Plot Title

Now that you've created your plot, you can add labels. One of these is a plot title, invoked using the **title** function (*FD*, p. 533).

title('Put Your Title Here')

The **title** function comes with several handles, including:

position - use this to set the horizontal and vertical position of the title. The syntax is :

title('Put your Title Here','position',[x-position y-position])

where:

- **x-position** = a number from 0 - 1, inclusive. 0 is the left side of the window; 1 is all the way on the righthand side.
- **y-position** = a number from 0 - 1, inclusive. 0 is the bottom of the window; 1 is all the way on the top.

The *position* point of the title string is the center, topmost point of the string. This is shown graphically by the black dot in the text string below.

If no position is provided, the default position is [0.5 1], which is centered at the top of the display window.

The stated position is from the center of the provided text horizontally and at its very top vertically.

Example - Positioning a Plot Title

The following is an example of creating a plot, setting limits on the horizontal axis, and creating and positioning a title.

```
y=wavread('mywavfile.wav',[10000 12500]);
plot(y,'k-');
xlim([1000 1100]);
title('Audio Plot','position',[0.5 0.92]);
```

The result is shown in Figure 28.
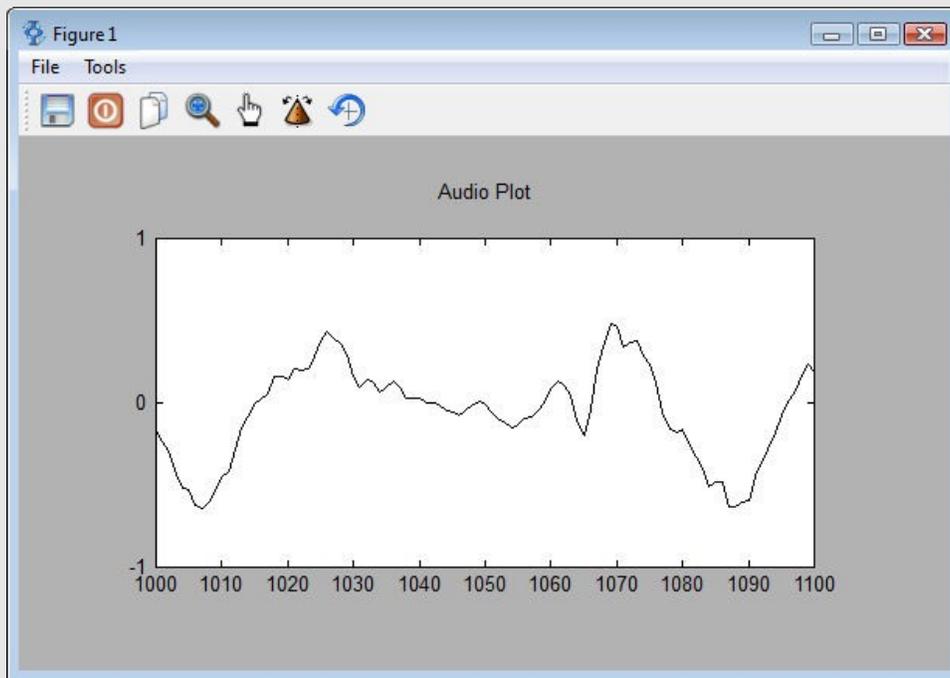
Figure 28: Positioning a Plot Title

Topic 5.10: Setting the X-Axis (Horizontal) Label

It's possible to provide a label to the horizontal axis of the graph. This is done using the **xlabel** command (*FD*, p. 542). An example is below:

xlabel('Time (samples)')

Example - Putting in a Horizontal Axis Label

The following is an example of how to establish a label for the horizontal axis of a plot.

```
y=wavread('mywavfile.wav',[10000 12500]);
plot(y,'k-');
xlim([1000 1100]);
title('Audio Plot','position',[0.5 0.92]);
xlabel('Time (samples)','fontsize',20);
```

The result is shown in Figure 29.
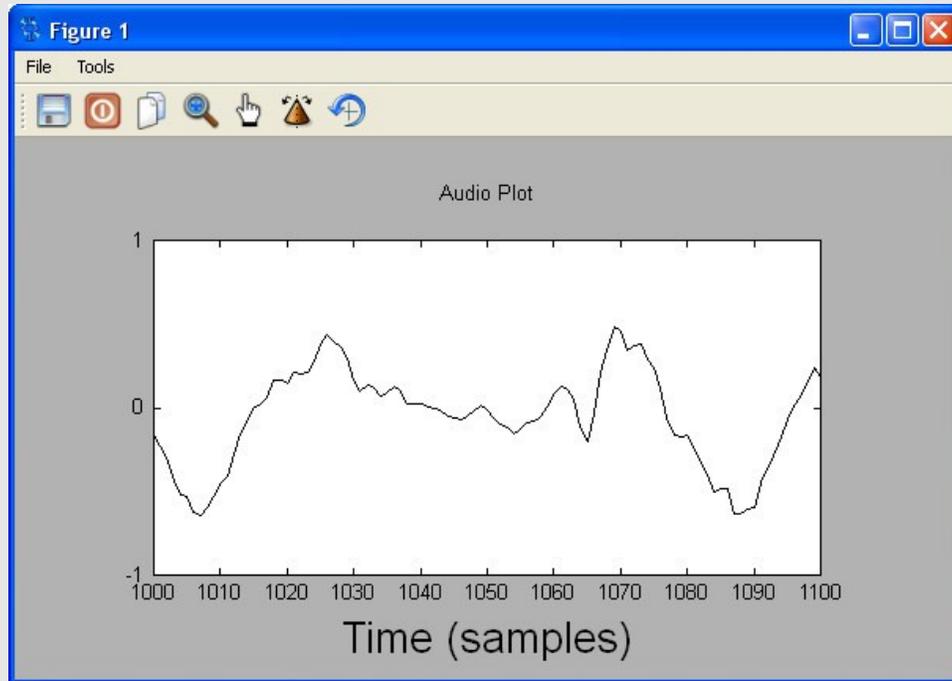
Figure 29: Making a Horizontal Axis Label

In this example, the horizontal axis is first written then it is resized to a larger size to make it more readable.

Topic 5.11: Setting the Y-Axis (Vertical) Label

You can also provide a label for the vertical or y-axis. You do this using the **ylabel** command (FD, p. 544).

Example - Adding a Vertical Axis Label to a Plot

```
y=wavread('moron.wav',[10000 12500]);
plot(y,'k-');
```

```
xlim([1000 1100]);
title('AudioPlot','position',[0.5 0.92]);
xlabel('Time(samples)','fontsize',20);
ylabel('Amplitude','fontsize',20);
```
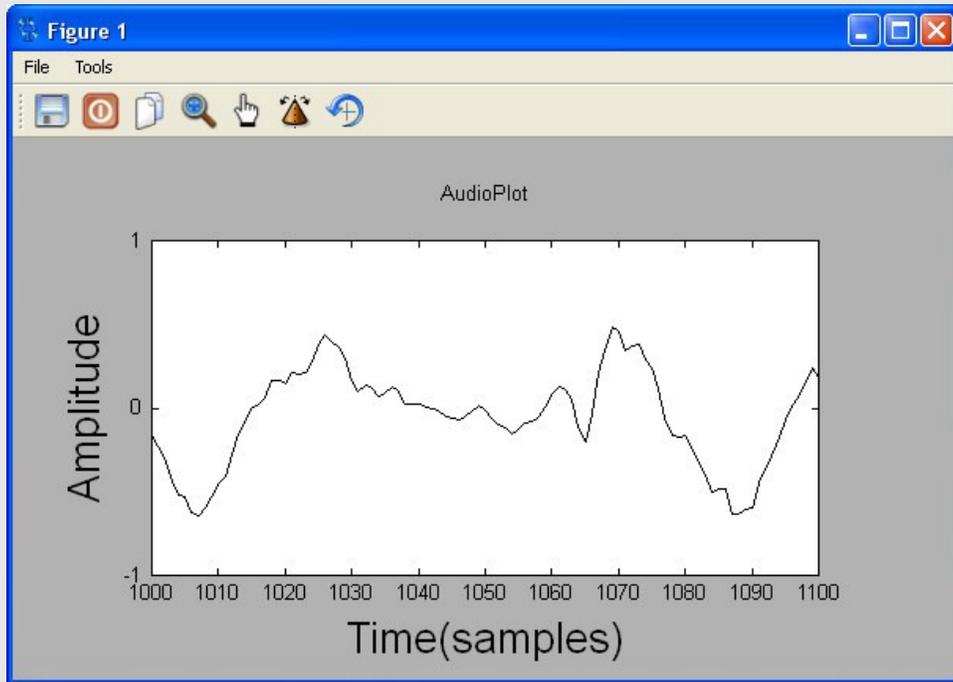

Figure 30: Plot showing addition of a vertical label

Topic 5.12: Working with Multiple Figures

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. 499). 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. But there are two 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.

Example - Creating Multiple Plots

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

```
plot(x)
y=cos(2*pi*t/16);
figure(2)
plot(y)
```

Topic 5.13: 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. 520). 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=1:128;
x=sin(2*pi*t/32);
plot(x)
print('testfile.png')
```

The resulting image is shown in Figure 31.
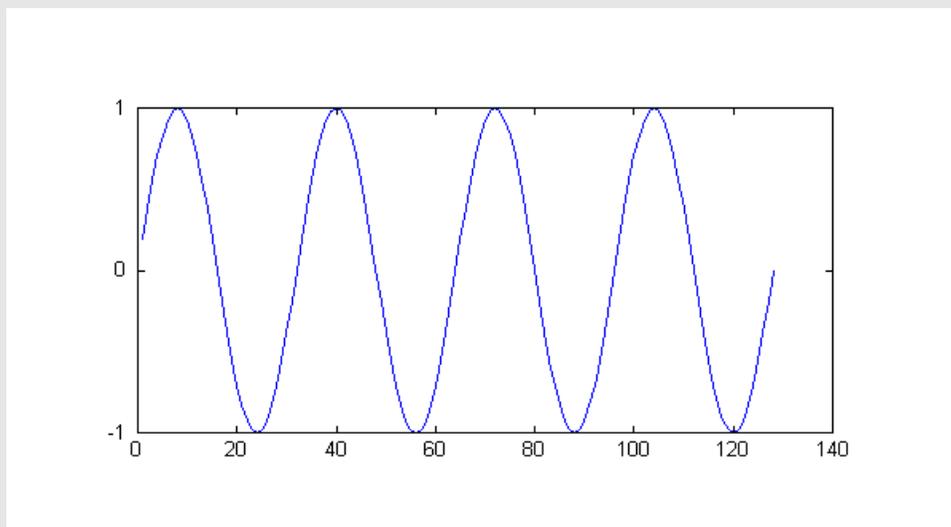
Figure 31: Plot as shown when saved as an image

Topic 5.14: 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 3.1: Setting the Working Directory for Saving Files* for more information.) To confirm that your file has been properly saved, you'll need to use the standard Windows Explorer window. Unfortunately, the Files Window in Freemat is only updated when Freemat is first started. Therefore, if you save an image, it will not show up in the Files Window until the next time you open Freemat.

Topic 6: Working with Arrays & Matrices

To Freemat, every number is a matrix. Single numbers, such as 5.342, are simply a matrix of 1, or [1 1] according to Freemat. That is why if you store a single variable, it will be listed as a size of [1 1].

Here's an example:

```
--> y=5.342;
--> who
Variable    Name        Type        Flags       Size
            ans         double                  [0 0]
            y           double                  [1 1]
```

The matrix size is listed in the *[row column]* format. Thus, a matrix listed as a size of [2 3] means that it has 2 rows and 3 columns.

Oh, and in case you are curious about the **ans** variable with a size of [0 0]. That's due to the **who** command (*FD*, p. 267) itself. It appears that any command will cause the **ans** variable to be created, even if it is a null value.

A matrix can have more than two dimensions. The simplest matrix has one dimension.

Topic 6.1: 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 (:). 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 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;
```

Example - Creating a Sequential Array

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

```
   1   2   3   4   5   6   7   8   9  10
t=1:3:10
t =
   1   4   7  10
```

Topic 6.2: 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. 361)
- **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. 363)
- **randi(low,high)** - creates an array of random integers between the values of *low* and *high* (inclusive). (*FD*, p. 370)

Topic 6.3: 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 6.4: 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 6.4.1: 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 32. 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

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

```
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
```

Figure 32: Matrix Addition in Freemat

Topic 6.4.2: 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 6.4.3: 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 6.4.4: Matrix Division

Topic 6.4.4.1: 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 6.4.4.2: 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 6.4.4.3: 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 6.4.5: Element-wise Matrix Math

It's possible to perform multiplication and division on matrices on an element-wise basis. 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. 267) to see the variable dimensions or
  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. 104) 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: ./

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
```

Topic 7: 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.

If you want to delve more into this subject, I highly recommend the following books:
- *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, Steven W. Smith, Newnes Publishing, 2003
- *Understanding Digital Signal Processing (Second Edition)*, Richard G. Lyons, Prentice-Hall Publishing, 2004
- *Digital Signal Analysis (Second Edition)*, Samuel D. Stearns & Don R. Hush, Prentice-Hall Publishing, 1990
- *Digital Signal Processing: A Practical Approach*, Emmanuel C. Ifeachor & Barrie W. Jervis, Addison-Wesley Publishers, 1993

Topic 7.1: Using the FFT

The **fft()** function (*FD*, p. 436) is used as follows:

y=fft(x)

where x = an array, preferably one whose length is a power of 2. The **fft** function works most efficiently when its length is 2, 4, 8, 16, or some other power of 2. 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. This is normally how FFT analyzers operate. They collect a power-of-2 number of real samples, window the data (more on that later), perform the FFT, then display the magnitude of 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=1:13;
--> x=cos(2*pi*t/10);
--> y=fft(x)
y =
Columns 1 to 2
```

```
0.8090 +  0.0000i    0.8090 +  4.9185i
Columns 3 to 4
0.8090 -  2.7632i    0.8090 -  1.1665i
Columns 5 to 6
0.8090 -  0.6501i    0.8090 -  0.3444i
Columns 7 to 8
0.8090 -  0.1088i    0.8090 +  0.1088i
Columns 9 to 10
0.8090 +  0.3444i    0.8090 +  0.6501i
Columns 11 to 12
0.8090 +  1.1665i    0.8090 +  2.7632i
Columns 13 to 13
0.8090 -  4.9185i
```

The first number (0.8090 + 0.0000i) is the DC component, or DC bias, of the signal. The next number, 0.8090 + 4.9185i, is the complex conjugate of the last number, 0.8090 - 4.9185i. The second number, 0.8090 -2.7632i, is the complex conjugate of the second-to-last number, 0.8090 + 2.7632i. 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 easily accomplished using the absolute value or **abs** function. The magnitude of a complex number is its absolute value.

Example - Creating a Frequency Domain Display

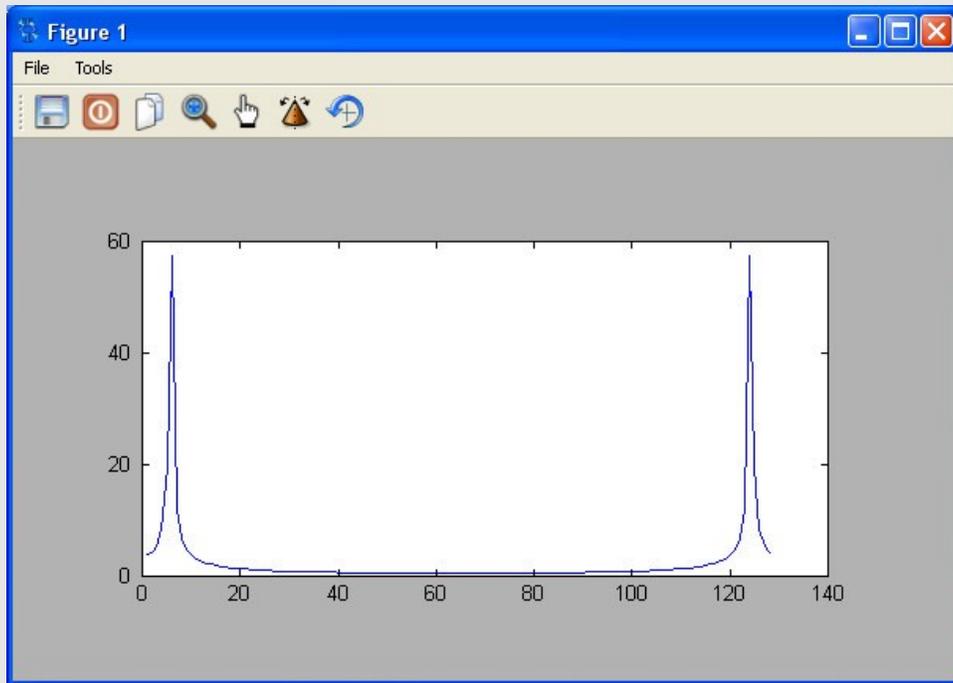```
t=1:128;
x=sin(2*pi*t/27);
y=fft(x);
plot(abs(y))
```

Figure 33: FFT plot of sine wave

Topic 7.2: Windowing the Time-Domain Samples

Due to issues of scalloping loss and spectral leakage, you may need to window the time-domain samples before you use the *fft* function. A window, in this instance, 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);
```

Blackman-Harris window

```
t=1:N;
w=0.42-0.5*cos(2*pi*t/N)+0.08*cos(4*pi*t/N);
```

Example - Adding a Window for the FFT Plot

```
N=128;
t=1:N;
x=sin(2*pi*t/27);
```

```
w=0.5-0.5*cos(2*pi*t/128);
y=x .* w;
z=fft(y);
plot(abs(z))
```
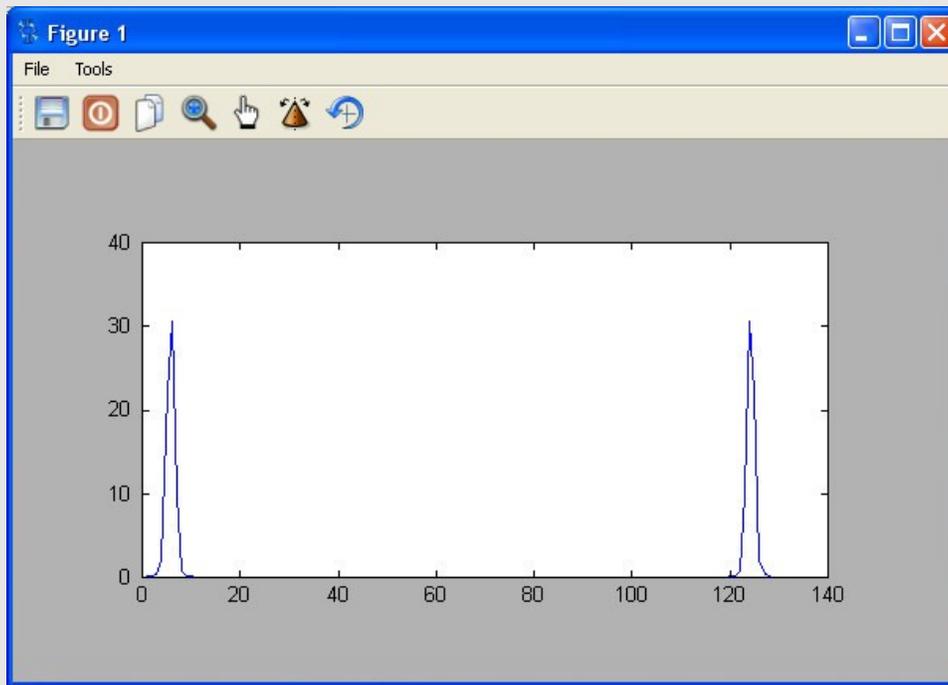

Figure 34: FFT plot of sine wave with Hanning window

Topic 7.3: Adjusting the Values of the FFT

You may have noticed the magnitudes of the peaks in the previous FFT magnitude display. The peaks are just above 30. This is due to the manner in which the FFT is calculated. To better represent the magnitudes of the actual display, multiply each FFT bin value by 2/N, where N is the number of samples in the array.

If you do so, you'll need to multiply each bin by the inverse (N/2) before calculating the inverse FFT (**ifft**).

Example - Adjusting the FFT Magnitude Values

In this example, the frequency bins created by the FFT will be each multiplied by 2/N in order to normalize them.

```
t=1:512;
x=cos(2*pi*t/23);
y=fft(x);
z=(2/512)*abs(y);
```
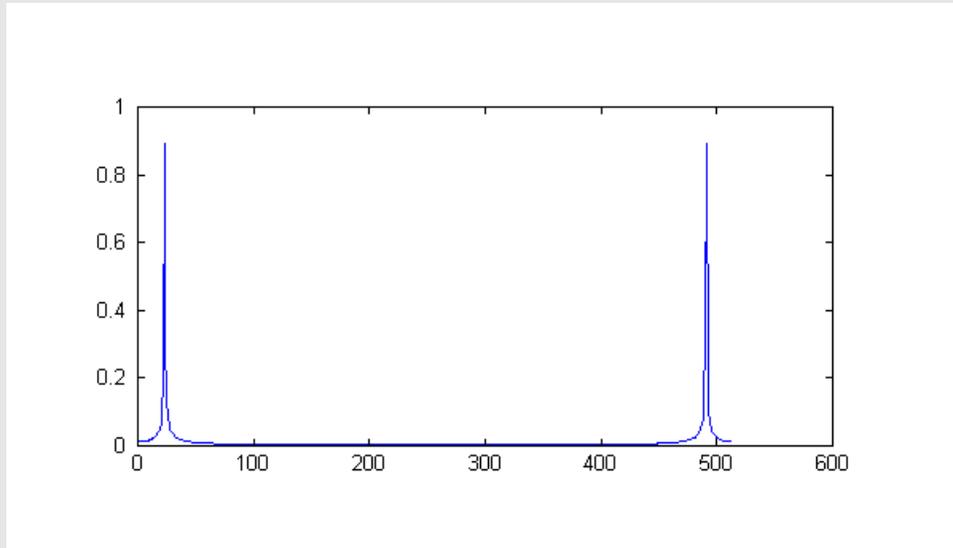
```
plot(z)
```


Figure 35: Plot of normalized FFT magnitudes. Note that the peaks are less then 1. This is due to scalloping loss and spectral leakage.

Topic 7.4: 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. 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

> The **ifft** function is not listed in the online reference nor in the *Freemat v3.6 Documentation*. Not to worry. It's there and it works.

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.

```
t=1:512;
x=cos(2*pi*t/90);
y=fft(x);
z=ifft(y);
```

The **real** function will be used to recover the real waveform from the complex array output from the **ifft** function.

The Freemat Primer

```
s=real(z);
```

In the final **plot** function, the variables will be offset by +1 and -1 in order to more easily compare them.
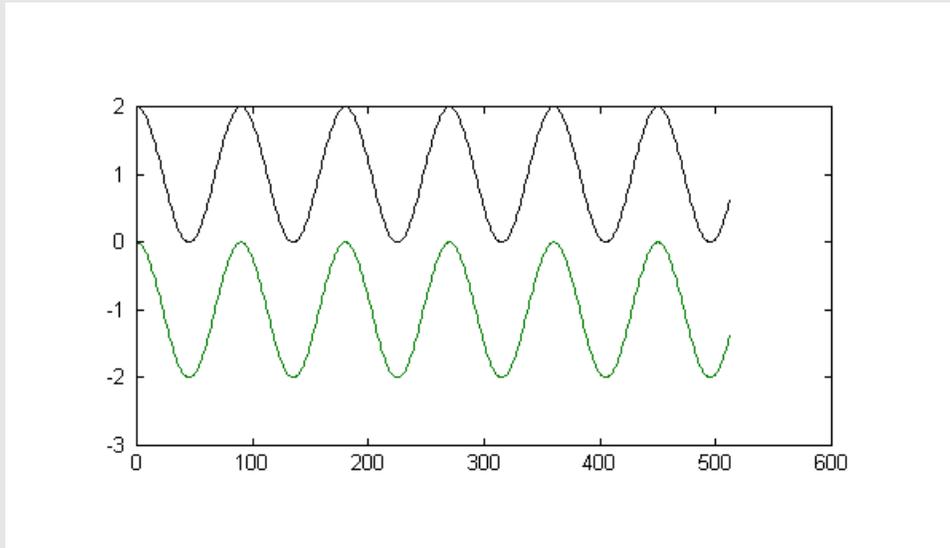
```
plot((x+1),'k-',(s-1),'g-');
```


Figure 36: Comparison of original waveform (top) and the waveform resulting from the FFT followed by inverse FFT (bottom).

Topic 7.5: 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 (or phase) of the result, the default is to display the sample number on the horizontal axis. For example, look at Figure 36. The horizontal axis is the sample number for each point of the FFT.

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 a Fractional Sample Frequency Horizontal Axis

I'll create a 512-point sine wave, calculate the FFT of that sine wave, then change the horizontal axis to a fractional display.

```
--> t=1:512;
--> x=cos(2*pi*t/90);
--> y=fft(x);
--> z=abs(y);
--> plot(z,'k-')
```
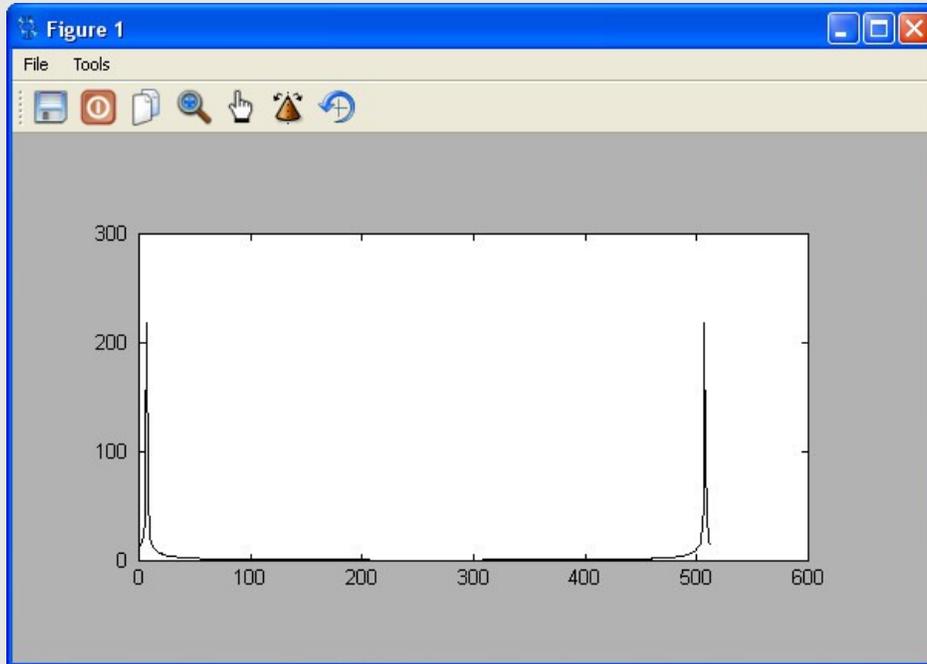

Figure 37: FFT display with sample numbers for the horizontal axis

Next, we'll change the horizontal axis so that it is the fraction of the sample rate. This uses the following commands:

```
--> h=t/512;
--> plot(h,z,'k-')
```
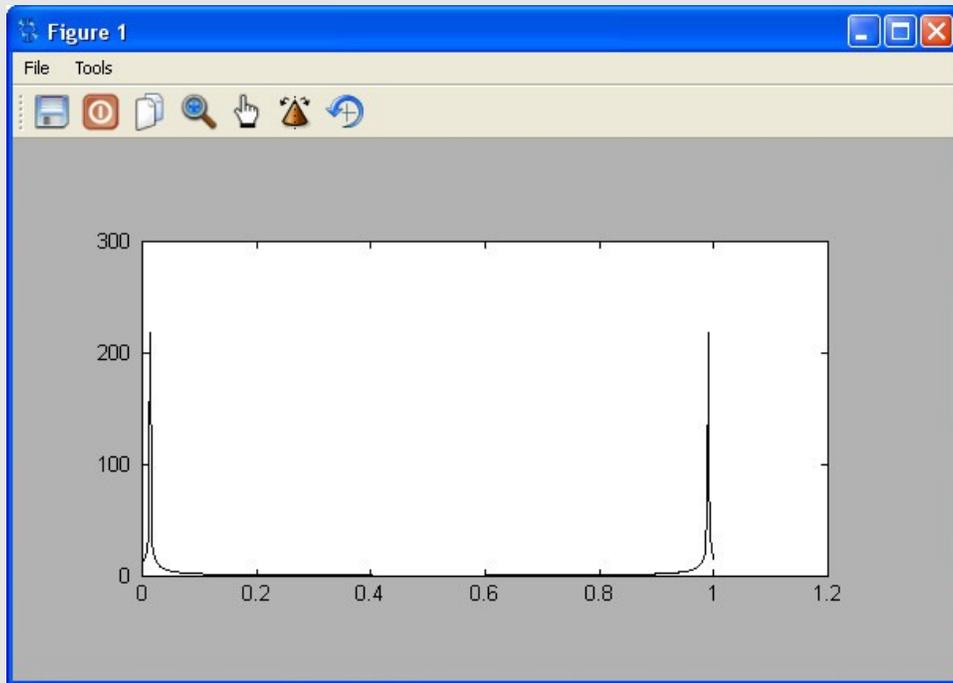
The result is shown in Figure 38.

Figure 38: 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:

$$x_f = \frac{f}{samples} * (sample\ rate)$$

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.

```
sr=1e6;
t=1:512;
freq=1e5;
x=cos(2*pi*t*freq/sr);
y=fft(x);
```

```
z=abs(y);
f=t*sr/512;
plot(f,z,'k-')
```
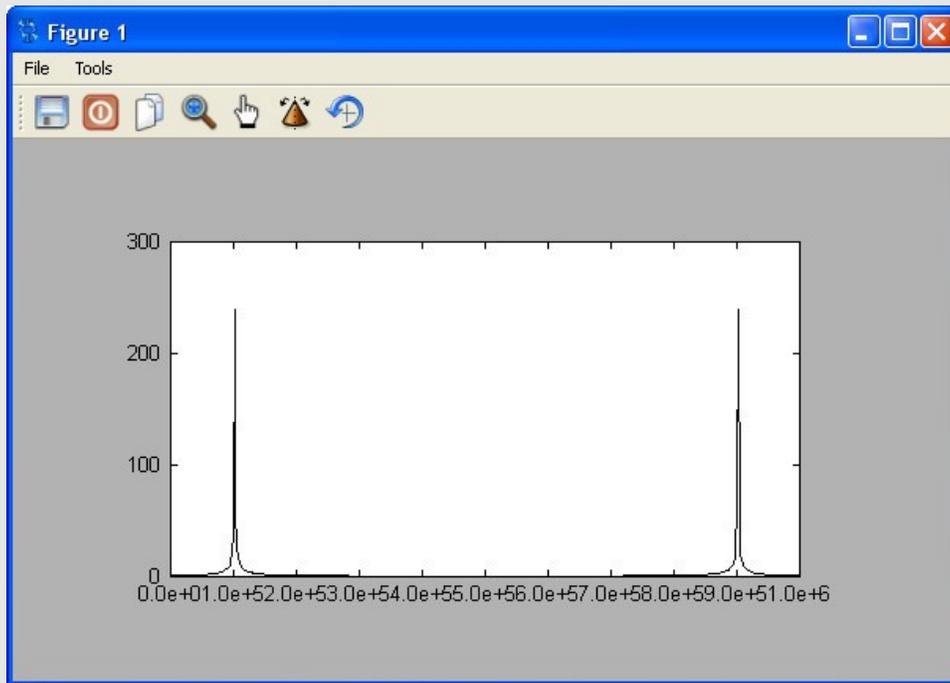

Figure 39: FFT display using actual frequencies of a 100 kHz sine wave sampled at 1 MHz.

Note that, in Figure 39, the horizontal axis is difficult to read. All of the numbers are jumbled together. 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. To make it easier to read, I'll divide the horizontal axis by 1000 to make the units kHz. This is shown in Figure 40.

```
sr=1e6;
t=1:512;
freq=1e5;
x=cos(2*pi*t*freq/sr);
y=fft(x);
z=abs(y);
f=t*sr/512000;
plot(f,z,'k-')
```
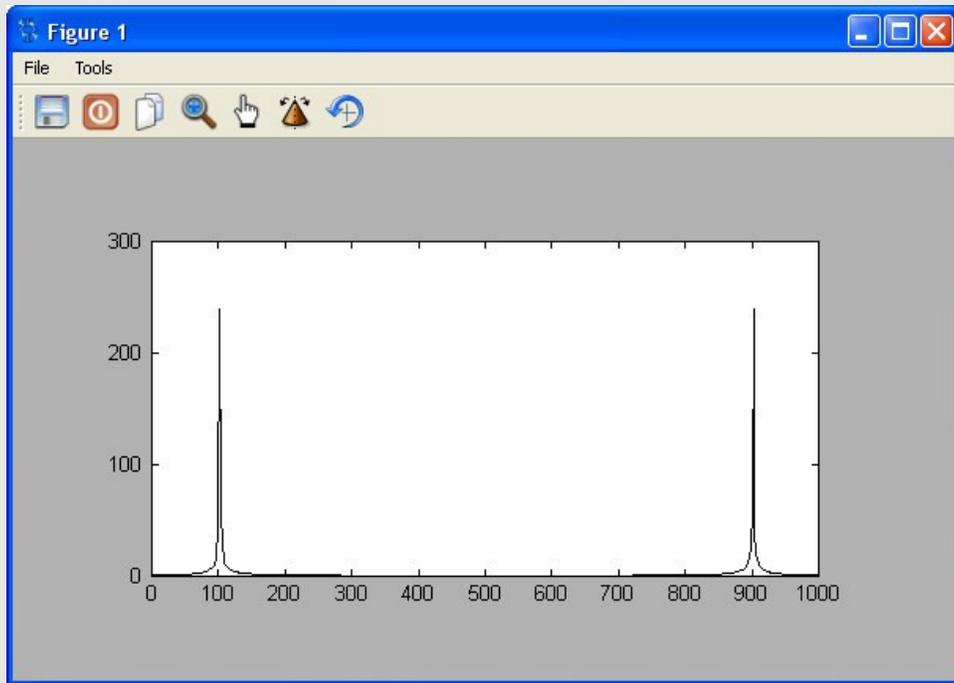
Figure 40: FFT display of 100 kHz sine wave. This is using a 1 MHz (1000 kHz) sample rate and the display has horizontal units of kHz.

Topic 8: Scripts & Functions

Scripts in Freemat are nothing more than a bunch of commands chained together. It's a computer program. 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 <u>so long as</u> 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 <u>absolutely insisted</u> 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 8.1: The Freemat Editor

One of the great things about Freemat is that it comes with a built-in editor. This is the one I <u>highly</u> 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.
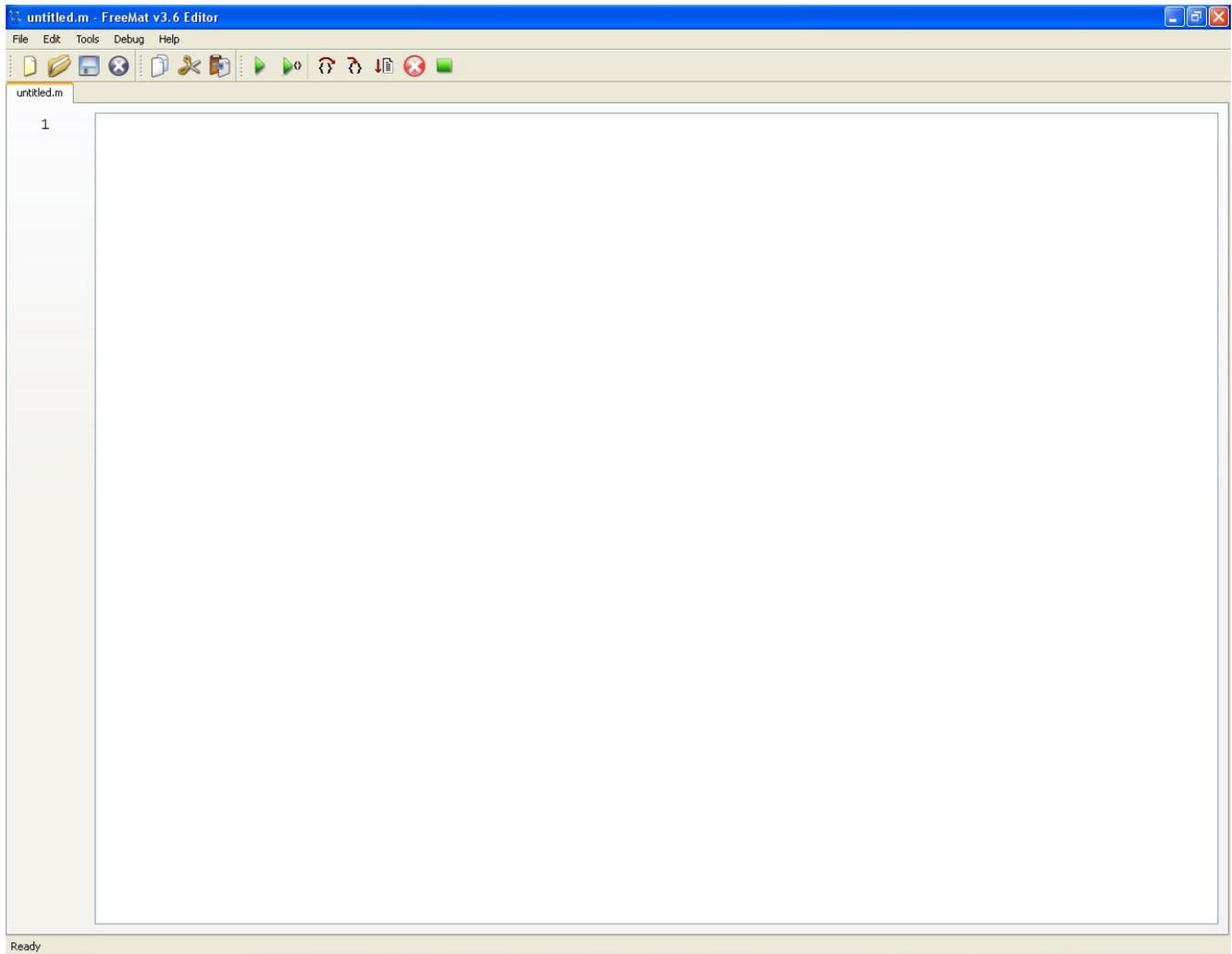
Figure 40: Editor Window

When you do, you'll get the window shown in Figure 40.

Topic 8.2: 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.
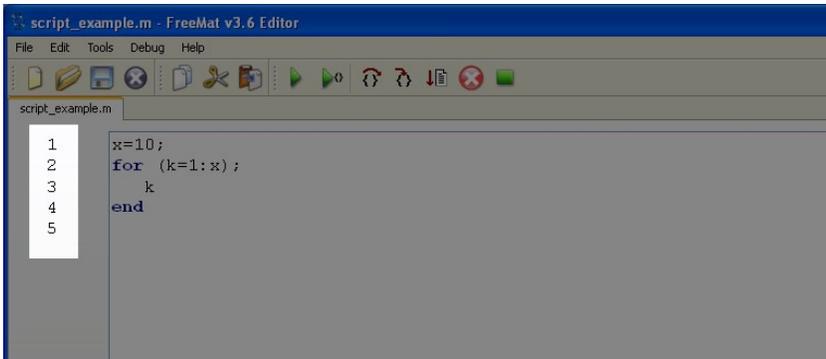
Line Numbering: Figure 41 shows a short, (essentially) useless script that shows the line numbering in the Editor Window. The line numbers are automatically generated. If an error occurs in the script or function, Freemat provides an error statement listing the line number where the error occured.

Figure 41: Editor Window showing line numbering
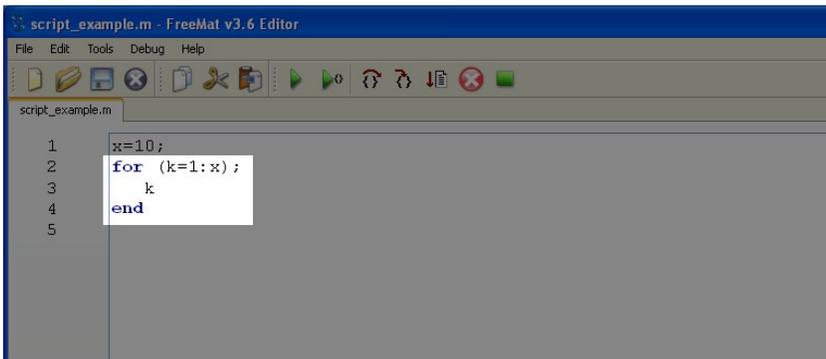
Color Coding & Indentation: Figure 42 shows the Editor Window with a **for** loop. (**For** loops will be covered in *Topic 9: Flow Control*.) Note the color coding used in the **for** loop. Also note the indentation of the statements within the **for** loop.

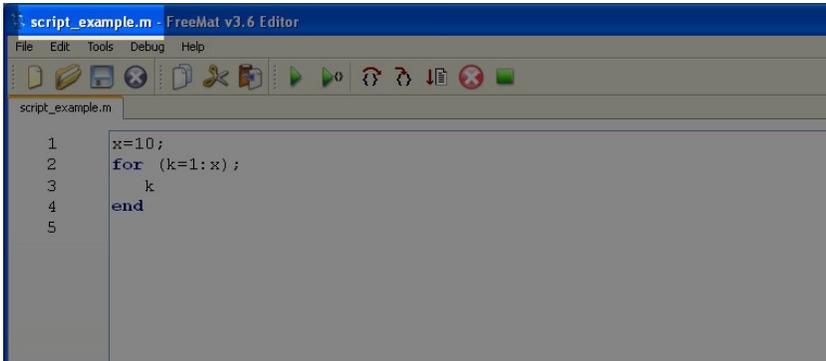*Figure 42: Editor Window showing color coding and indentation.*

Figure 43: Editor Window showing the filename

Filename: Figure 43 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.
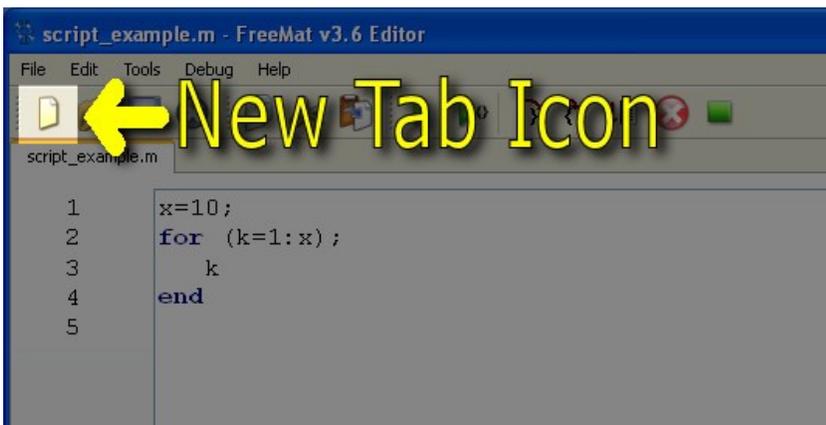

Figure 44: Editor Window showing the icon used to create a new tab.

New Tab Icon: Figure 44 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.

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.)

```
t=1:128;
x=sin(2*pi*t/16);
plot(x)
```

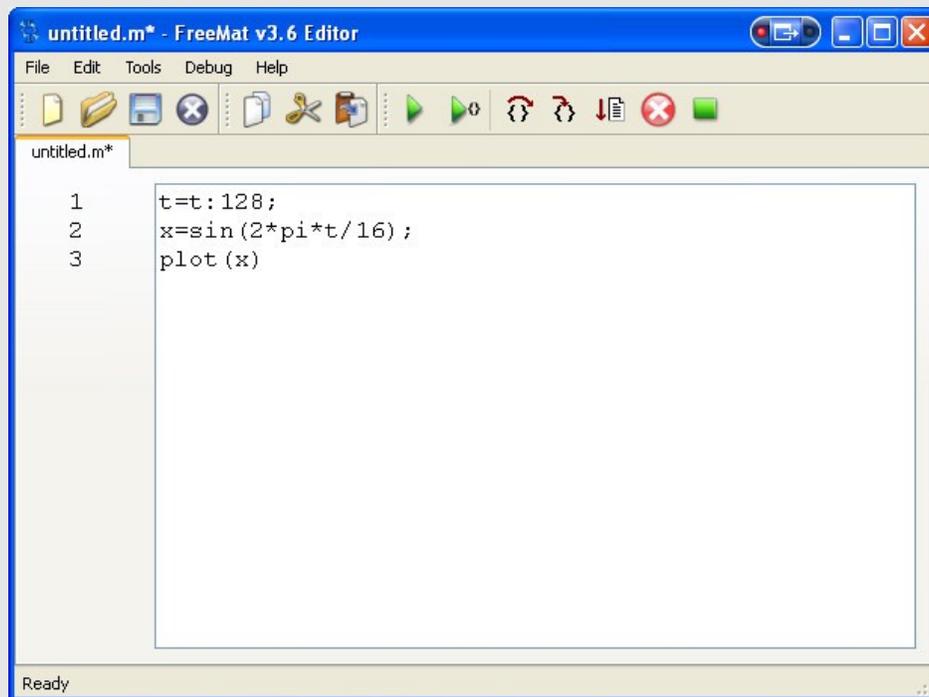Figure 45 shows the Editor Window with the program.

Figure 45: Editor window with test script

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

```
test_script
```

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 46.
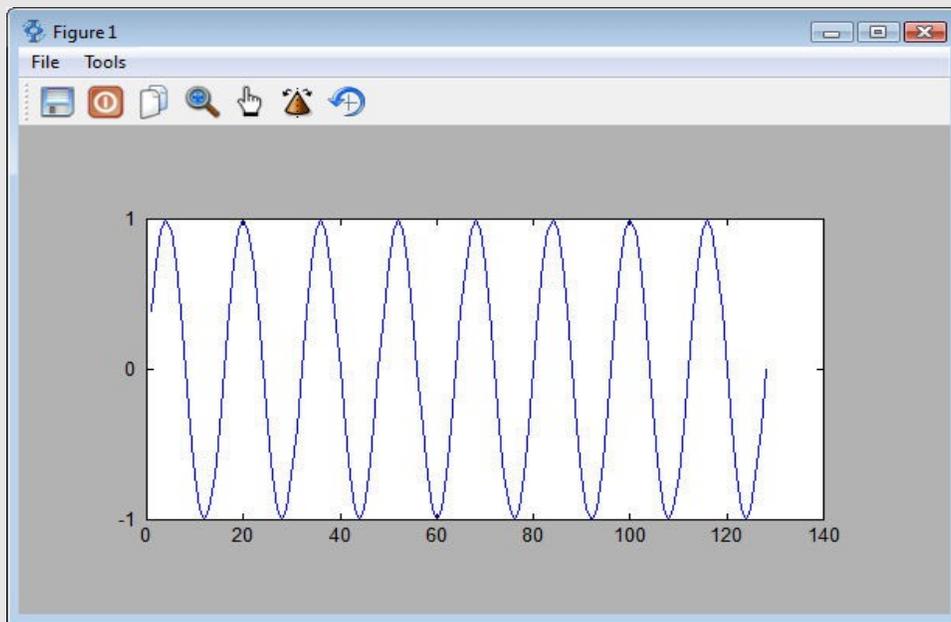
Figure 46: Basic plot, in this case a sine wave, generated from a script.

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.

Example - A Script to Change Your Working Directory

Earlier, I explained how you could create a short script to change your working directory. Here's one that I use to change mine, and I have it stored in my *Program Files/Freemat* folder in Windows. That way, every time I start up Freemat, I can type in the filename (workdir) in the Command Window and instantly I have a working directory where I can keep my data.

```
cd 'C:\Documents and Settings\Owner\My Documents\Freemat'
```

I saved the file into the *C:/Program Files/Freemat 3.6* folder under the name *workdir.m*. Therefore, when I want to run the file after starting up Freemat, I simply type *workdir* in the Command Window.

```
--> workdir
-->
```

Figure 47 shows a screen shot taken right after starting up Freemat. Figure 48 shows a screen shot taken right after running the *workdir* script.
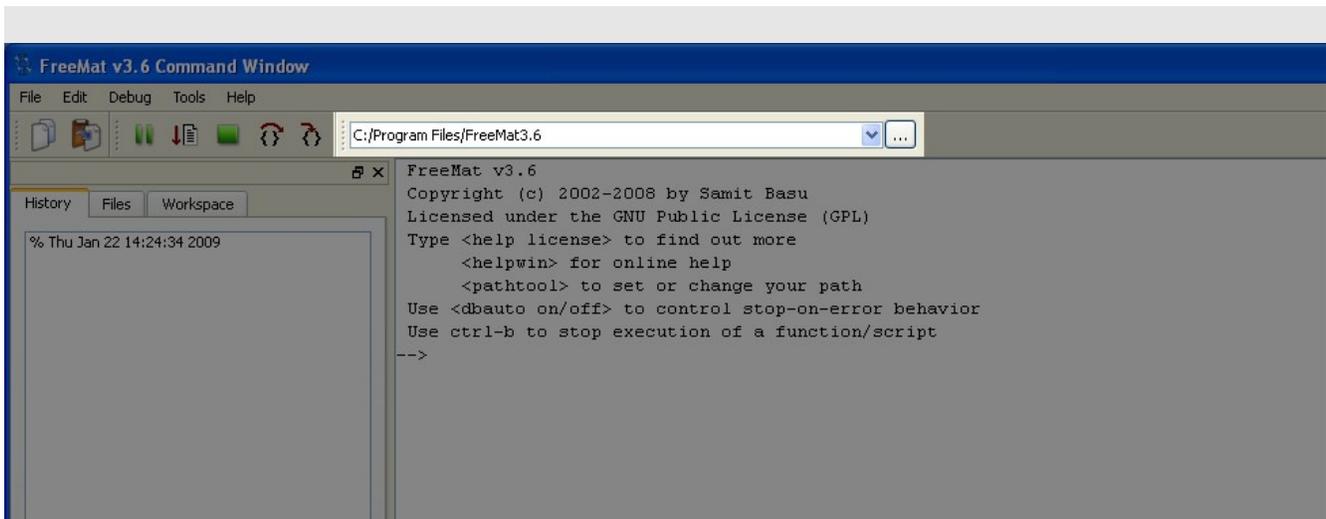
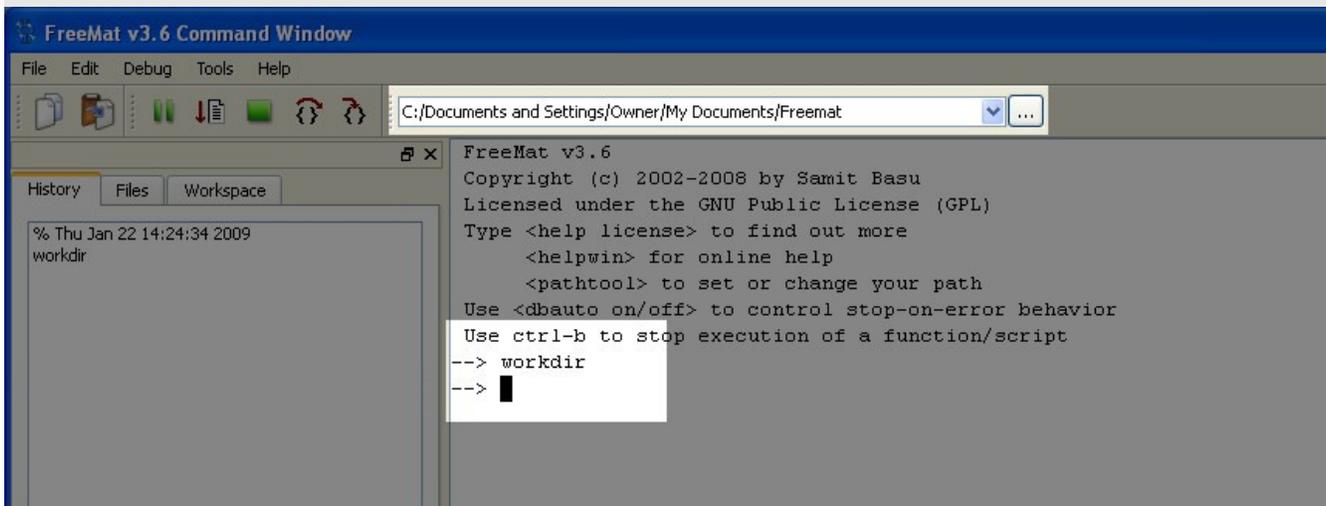*Figure 47: Screen shot showing working directory after starting Freemat.*

Figure 48: Screen shot taken after running the workdir script. It shows the new working directory.

Topic 8.3: 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:

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

```
myscript
```

It's that simple.

Topic 8.4: Creating a Function

A function is a short script that performs an operation. 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 [return_value1, return_value2, ... , return_valueM] =
function_name (parameter1, parameter2, ... , parameterN)
        statements
```

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. Once it's completed, it returns the value (if so designed) and goes back to wherever the function was called.

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. In the

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

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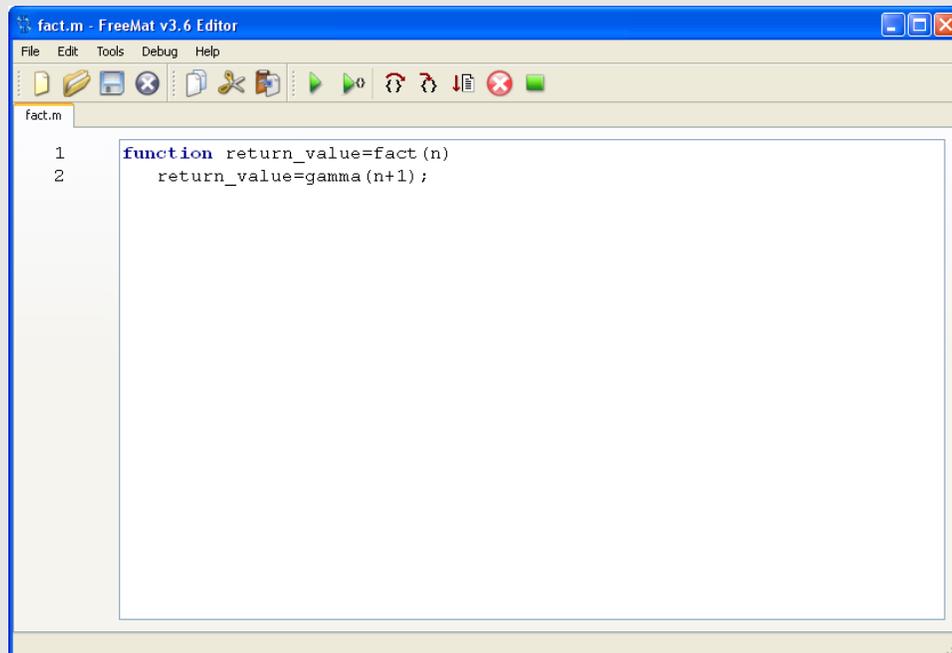

Figure 49: The editor window with a short function written

This function calculates the factorial value of the input variable *n*. I saved it under the filename *fact.m*.
Below shows the result of calculating 5!.

```
--> fact(5)
ans =
120
```

The variables *return_value* are not necessary if the function does not return any values. Let's look at an example:

Topic 9: Flow Control

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

Topic 9.1: For Loops

The **for** loop (*FD*, p. 110) 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.
-->
```

Topic 9.2: Comparison / Equality Operators

Before going into **while** loops (*FD*, p. 121) and **if-elseif-else** statements (*FD*, p. 111), 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.

Topic 9.3: While Loops

A **while** loop (*FD*, p. 121) is one which is run until a certain condition is met. In the for loop, the condition is the number of loops to run. In the while loop, you'll use a comparison operator to decide when to stop.

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

```
      end
```

Example - Using the While Loop

This example of the **while** loop will create a function to calculate the greatest common divisor of two integers.

```
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. 188). 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.

Topic 9.4: If-Elseif-Else Statements

This is similar to the *if-then-else* statements many programmers are familiar with. The difference is that this type of decision statement can be carried on indefinitely. 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
```