



Høgskolen i Telemark

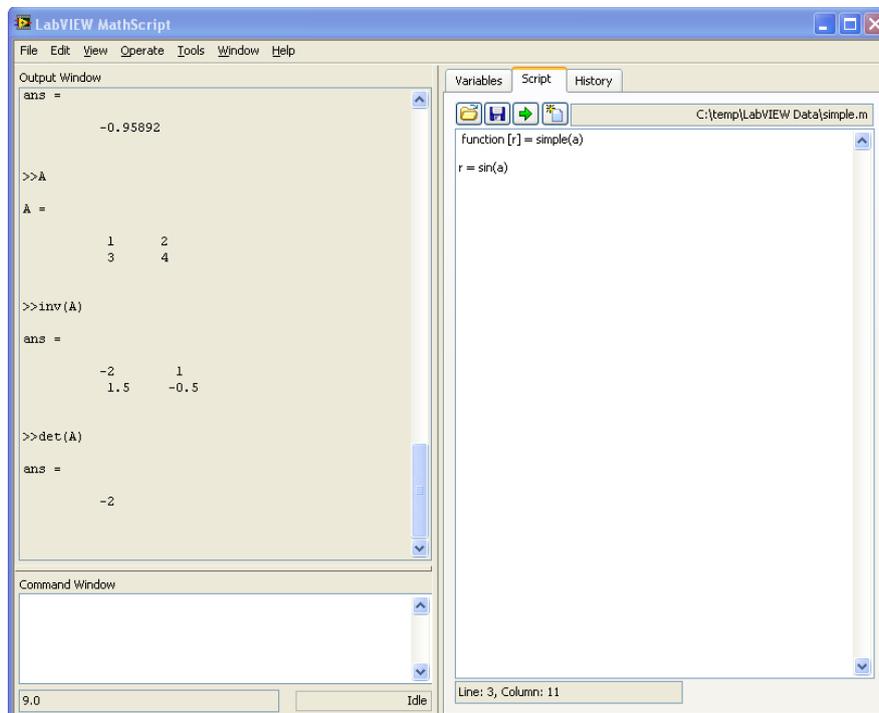
Telemark University College

Department of Electrical Engineering, Information Technology and Cybernetics

Tutorial

LABVIEW MATHSCRIPT

HANS-PETTER HALVORSEN, 2010.05.25





PREFACE

This document explains the basic concepts of using LabVIEW MathScript.

For more information about LabVIEW, visit my Blog: <http://home.hit.no/~hansha/>

What is LabVIEW?

LabVIEW (short for **L**aboratory **V**irtual **I**nstrumentation **E**ngineering **W**orkbench) is a platform and development environment for a visual programming language from National Instruments. The graphical language is named "G".

What is MATLAB?

MATLAB is a tool for technical computing, computation and visualization in an integrated environment. MATLAB is an abbreviation for MATrix LABoratory, so it is well suited for matrix manipulation and problem solving related to Linear Algebra.

MATLAB offers lots of additional Toolboxes for different areas such as Control Design, Image Processing, Digital Signal Processing, etc.

What is MathScript?

MathScript is a high-level, text- based programming language. MathScript includes more than 800 built-in functions and the syntax is similar to MATLAB. You may also create custom-made m-file like you do in MATLAB.

MathScript is an add-on module to LabVIEW but you don't need to know LabVIEW programming in order to use MathScript. If you want to integrate MathScript functions (built-in or custom-made m-files) as part of a LabVIEW application and combine graphical and textual programming, you can work with the **MathScript Node**.

In addition to the MathScript built-in functions, different add-on modules and toolkits installs additional functions. The **LabVIEW Control Design and Simulation Module** and **LabVIEW Digital Filter Design Toolkit** install lots of additional functions.

You can more information about MathScript here: <http://www.ni.com/labview/mathscript.htm>

How do you start using MathScript?

You need to install **LabVIEW** and the **LabVIEW MathScript RT Module**. When necessary software is installed, start MathScript by open LabVIEW:



In the **Getting Started** window, select **Tools -> MathScript Window...**:

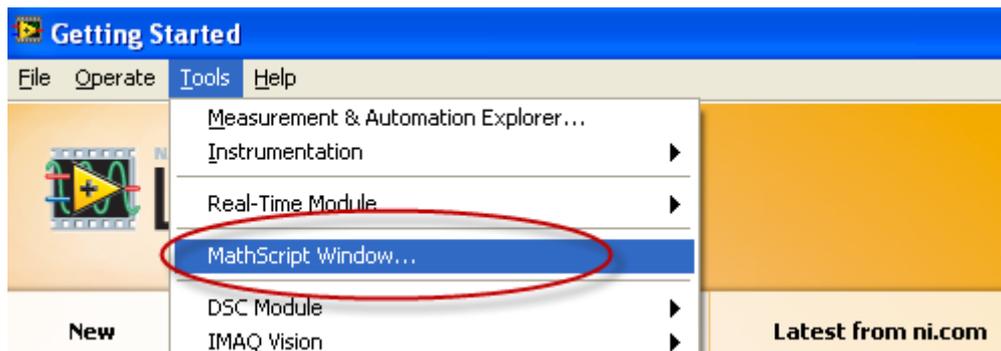




TABLE OF CONTENTS

| | |
|--|----|
| Preface | ii |
| Table of Contents | iv |
| 1 Introduction to LabVIEW | 1 |
| 1.1 Dataflow programming..... | 1 |
| 1.2 Graphical programming..... | 1 |
| 1.3 Benefits..... | 1 |
| 1.4 LabVIEW MathScript RT Module | 2 |
| 2 LabVIEW MathScript RT Module | 3 |
| 3 LabVIEW MathScript..... | 4 |
| 3.1 Introduction..... | 4 |
| 3.2 Help..... | 6 |
| 3.3 Examples..... | 6 |
| 3.4 Useful commands | 9 |
| Calling functions In MathScript | 9 |
| User-Defined Functions In MathScript | 10 |
| Scripts | 11 |
| 3.5 Flow Control | 13 |
| 3.5.1 If-else Statement..... | 13 |
| 3.5.2 Switch and Case Statement | 13 |
| 3.5.3 For loop..... | 14 |
| 3.5.4 While loop..... | 14 |
| 3.6 Plotting | 15 |
| 4 Linear Algebra Examples..... | 17 |
| 4.1 Vectors..... | 17 |
| 4.2 Matrices..... | 18 |
| 4.2.1 Transpose..... | 18 |

| | | |
|-------|--|----|
| 4.2.2 | Diagonal | 18 |
| 4.2.3 | Triangular | 19 |
| 4.2.4 | Matrix Multiplication | 20 |
| 4.2.5 | Matrix Addition | 21 |
| 4.2.6 | Determinant..... | 21 |
| 4.2.7 | Inverse Matrices | 22 |
| 4.3 | Eigenvalues | 23 |
| 4.4 | Solving Linear Equations..... | 24 |
| 4.5 | LU factorization | 25 |
| 4.6 | The Singular Value Decomposition (SVD)..... | 27 |
| 4.7 | Commands..... | 27 |
| 5 | Control Design and Simulation | 28 |
| 5.1 | State-space models and Transfer functions | 28 |
| 5.1.1 | PID..... | 29 |
| 5.1.2 | State-space model | 30 |
| 5.1.3 | Transfer function | 31 |
| 5.1.4 | First Order Systems..... | 32 |
| 5.1.5 | Second Order Systems | 32 |
| 5.1.6 | Padé-approximation | 34 |
| 5.2 | Frequency Response Analysis..... | 35 |
| 5.2.1 | Bode Diagram | 36 |
| | Time Response | 40 |
| 6 | MathScript Node | 42 |
| 6.1 | Transferring MathScript Nodes between Computers | 43 |
| 6.2 | Examples..... | 44 |
| 6.3 | Exercises | 47 |
| 7 | MATLAB Script | 48 |
| | Appendix A – MathScript Functions for Control and Simulation | 49 |



1 INTRODUCTION TO LABVIEW

LabVIEW (short for **L**aboratory **V**irtual **I**nstrumentation **E**ngineering **W**orkbench) is a platform and development environment for a visual programming language from National Instruments. The graphical language is named "G". Originally released for the Apple Macintosh in 1986, LabVIEW is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms including Microsoft Windows, various flavors of UNIX, Linux, and Mac OS X. The latest version of LabVIEW is version LabVIEW 2009, released in August 2009. Visit National Instruments at www.ni.com.

The code files have the extension ".vi", which is an abbreviation for "Virtual Instrument". LabVIEW offers lots of additional Add-Ons and Toolkits.

1.1 DATAFLOW PROGRAMMING

The programming language used in LabVIEW, also referred to as G, is a dataflow programming language. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

1.2 GRAPHICAL PROGRAMMING

LabVIEW ties the creation of user interfaces (called front panels) into the development cycle. LabVIEW programs/subroutines are called virtual instruments (VIs). Each VI has three components: a block diagram, a front panel, and a connector panel. The last is used to represent the VI in the block diagrams of other, calling VIs. Controls and indicators on the front panel allow an operator to input data into or extract data from a running virtual instrument. However, the front panel can also serve as a programmatic interface. Thus a virtual instrument can either be run as a program, with the front panel serving as a user interface, or, when dropped as a node onto the block diagram, the front panel defines the inputs and outputs for the given node through the connector pane. This implies each VI can be easily tested before being embedded as a subroutine into a larger program.

The graphical approach also allows non-programmers to build programs simply by dragging and dropping virtual representations of lab equipment with which they are already familiar. The LabVIEW programming environment, with the included examples and the documentation, makes it simple to create small applications. This is a benefit on one side, but there is also a certain danger of underestimating the expertise needed for good quality "G" programming. For complex algorithms or large-scale code, it is important that the programmer possess an extensive knowledge of the special LabVIEW syntax and the topology of its memory management. The most advanced LabVIEW development systems offer the possibility of building stand-alone applications. Furthermore, it is possible to create distributed applications, which communicate by a client/server scheme, and are therefore easier to implement due to the inherently parallel nature of G-code.

1.3 BENEFITS

One benefit of LabVIEW over other development environments is the extensive support for accessing instrumentation hardware. Drivers and abstraction layers for many different types of instruments and buses

are included or are available for inclusion. These present themselves as graphical nodes. The abstraction layers offer standard software interfaces to communicate with hardware devices. The provided driver interfaces save program development time. The sales pitch of National Instruments is, therefore, that even people with limited coding experience can write programs and deploy test solutions in a reduced time frame when compared to more conventional or competing systems. A new hardware driver topology (DAQmxBase), which consists mainly of G-coded components with only a few register calls through NI Measurement Hardware DDK (Driver Development Kit) functions, provides platform independent hardware access to numerous data acquisition and instrumentation devices. The DAQmxBase driver is available for LabVIEW on Windows, Mac OS X and Linux platforms.

For more information about LabVIEW, visit my Blog: <http://home.hit.no/~hansha/>

1.4 LABVIEW MATHSCRIPT RT MODULE

The LabVIEW MathScript RT Module is an add-on module to LabVIEW. With LabVIEW MathScript RT Module you can:

- Deploy your custom .m files to NI real-time hardware
- Reuse many of your scripts created with The MathWorks, Inc. MATLAB® software and others
- Develop your .m files with an interactive command-line interface
- Embed your scripts into your LabVIEW applications using the MathScript Node



2 LABVIEW MATHSCRIPT RT MODULE

You can work with LabVIEW MathScript through either of two interfaces: the “LabVIEW MathScript Interactive Window” or the “MathScript Node”.

You can work with LabVIEW MathScript RT Module through both interactive and programmatic interfaces. For an interactive interface in which you can load, save, design, and execute your .m file scripts, you can work with the “MathScript Interactive Window”. To deploy your .m file scripts as part of a LabVIEW application and combine graphical and textual programming, you can work with the “MathScript Node”.

The LabVIEW MathScript RT Module complements traditional LabVIEW graphical programming for such tasks as algorithm development, signal processing, and analysis. The LabVIEW MathScript RT Module speeds up these and other tasks by giving users a single environment in which they can choose the most effective syntax, whether textual, graphical, or a combination of the two. In addition, you can exploit the best of LabVIEW and thousands of publicly available .m file scripts from the web, textbooks, or your own existing m-script applications. LabVIEW MathScript RT Module is able to process your files created using the current MathScript syntax and, for backwards compatibility, files created using legacy MathScript syntaxes. LabVIEW MathScript RT Module can also process certain of your files utilizing other text-based syntaxes, such as files you created using MATLAB software. Because the MathScript RT engine is used to process scripts contained in a MathScript Windows or MathScript Node, and because the MathScript RT engine does not support all syntaxes, not all existing text-based scripts are supported.

LabVIEW MathScript RT Module supports most of the functionality available in MATLAB, the syntax is also similar.

For more details, see <http://zone.ni.com/devzone/cda/tut/p/id/3257>

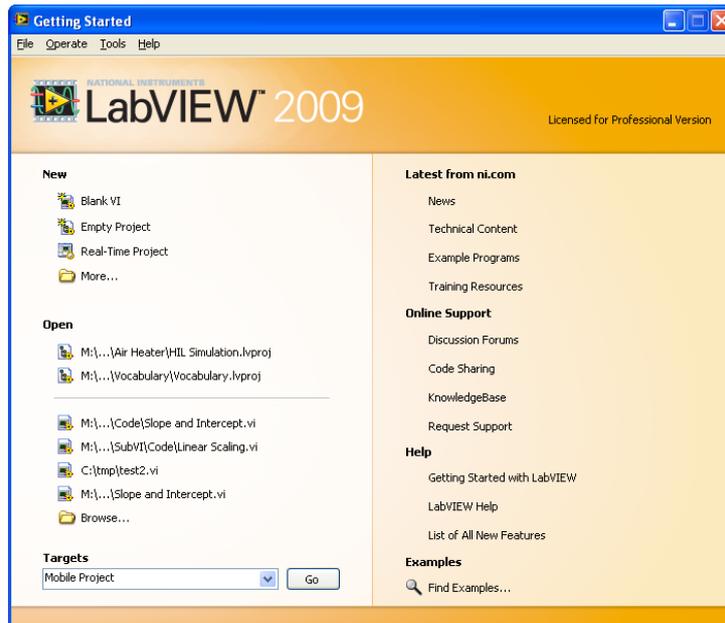


3 LABVIEW MATHSCRIPT

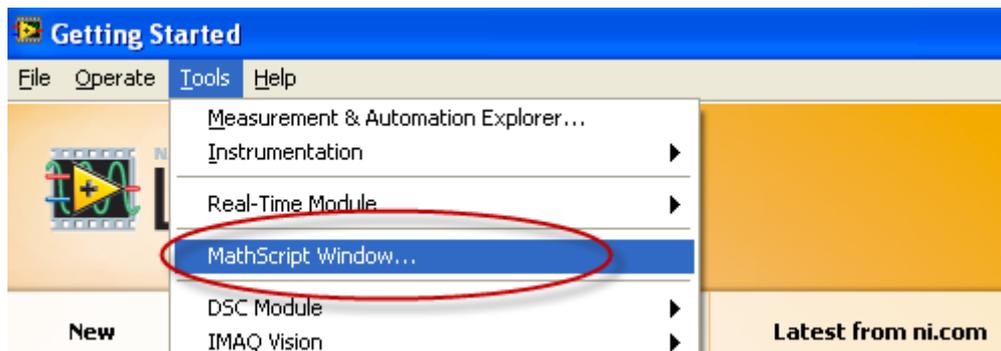
3.1 INTRODUCTION

Requires: **MathScript RT Module**

How do you start using MathScript? You need to install **LabVIEW** and the **LabVIEW MathScript RT Module**. When necessary software is installed, start MathScript by open LabVIEW:



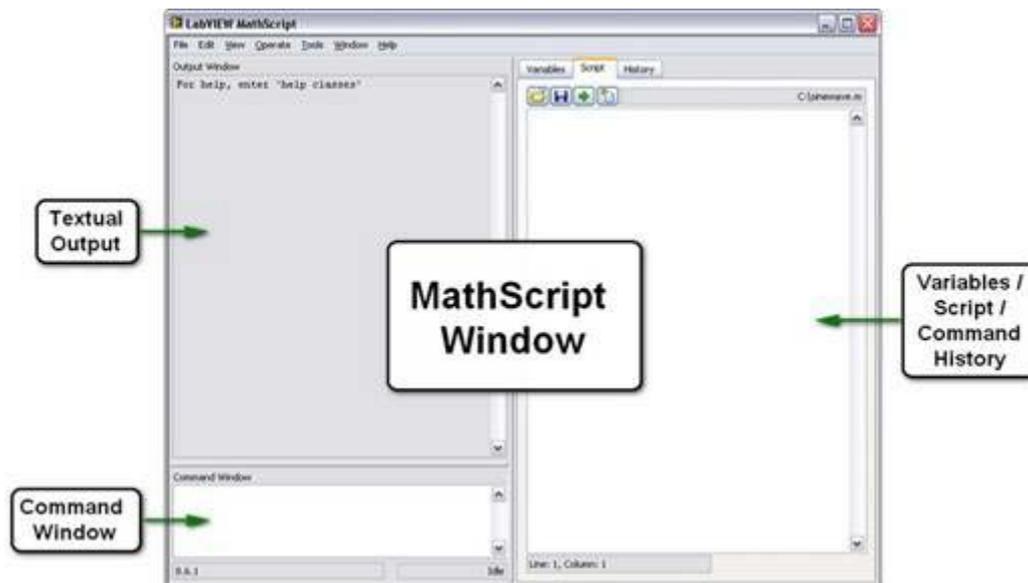
In the **Getting Started** window, select **Tools -> MathScript Window...**:



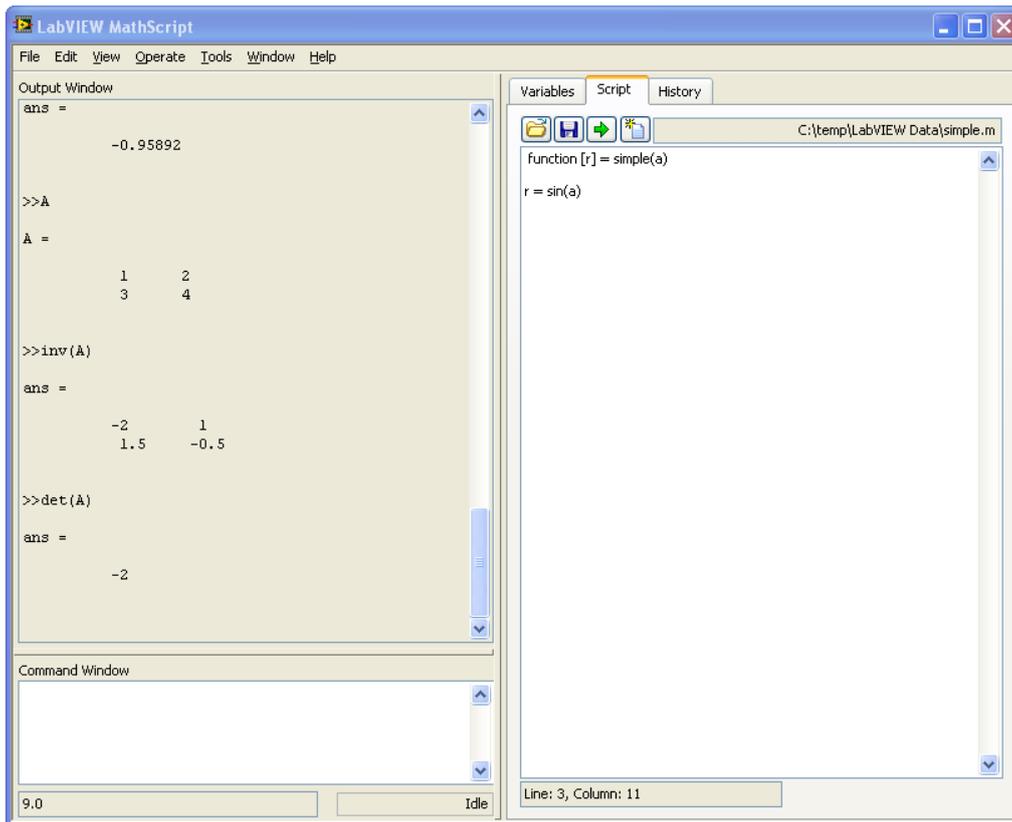
The “LabVIEW MathScript Window” is an interactive interface in which you can enter .m file script commands and see immediate results, variables and commands history. The window includes a command-line interface where you can enter commands one-by-one for quick calculations, script debugging or learning. Alternatively, you can enter and execute groups of commands through a script editor window.

As you work, a variable display updates to show the graphical / textual results and a history window tracks your commands. The history view facilitates algorithm development by allowing you to use the clipboard to reuse your previously executed commands.

You can use the “LabVIEW MathScript Window” to enter commands one at a time. You also can enter batch scripts in a simple text editor window, loaded from a text file, or imported from a separate text editor. The “LabVIEW MathScript Window” provides immediate feedback in a variety of forms, such as graphs and text.



Example:



[End of Example]

3.2 HELP

You may also type help in your command window

```
>>help
```

Or more specific, e.g.,

```
>>help plot
```

3.3 EXAMPLES

I advise you to test all the examples in this text in LabVIEW MathScript in order to get familiar with the program and its syntax. All examples in the text are outlined in a frame like this:

```
>>  
...
```

This is commands you should write in the **Command Window**.

You type all your commands in the Command Window. I will use the symbol “>>” to illustrate that the commands should be written in the Command Window.

Example: Matrices

Defining the following matrix

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}$$

The syntax is as follows:

```
>> A = [1 2;0 3]
```

Or

```
>> A = [1,2;0,3]
```

If you, for an example, want to find the answer to

$$a + b, \text{ where } a = 4, b = 3$$

```
>>a=4  
>>b=3  
>>a+b
```

MathScript then responds:

```
ans =  
    7
```

MathScript provides a simple way to define simple arrays using the syntax: “**init:increment:terminator**”. For instance:

```
>> array = 1:2:9  
  
array =  
  
1 3 5 7 9
```

The code defines a variable named array (or assigns a new value to an existing variable with the name array) which is an array consisting of the values 1, 3, 5, 7, and 9. That is, the array starts at 1 (the init value), increments with each step from the previous value by 2 (the increment value), and stops once it reaches (or to avoid exceeding) 9 (the terminator value).

The increment value can actually be left out of this syntax (along with one of the colons), to use a default value of 1.

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

The code assigns to the variable named `ari` an array with the values 1, 2, 3, 4, and 5, since the default value of 1 is used as the incrementer.

Note that the indexing is one-based, which is the usual convention for matrices in mathematics. This is atypical for programming languages, whose arrays more often start with zero.

Matrices can be defined by separating the elements of a row with blank space or comma and using a semicolon to terminate each row. The list of elements should be surrounded by square brackets: `[]`. Parentheses: `()` are used to access elements and subarrays (they are also used to denote a function argument list).

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
16  3  2 13
 5 10 11  8
 9  6  7 12
 4 15 14  1
>> A(2,3)
ans =
11
```

Sets of indices can be specified by expressions such as `"2:4"`, which evaluates to `[2, 3, 4]`. For example, a submatrix taken from rows 2 through 4 and columns 3 through 4 can be written as:

```
>> A(2:4, 3:4)
ans =
11 8
 7 12
14 1
```

A square identity matrix of size `n` can be generated using the function `eye`, and matrices of any size with zeros or ones can be generated with the functions `zeros` and `ones`, respectively.

```
>> eye(3)
ans =
1 0 0
0 1 0
```

```

0 0 1
>> zeros(2,3)
ans =
0 0 0
0 0 0
>> ones(2,3)
ans =
1 1 1
1 1 1

```

3.4 USEFUL COMMANDS

Here are some useful commands:

| Command | Description |
|--------------------------------------|----------------------------|
| eye (x), eye (x,y) | Identity matrix of order x |
| ones (x), ones (x,y) | A matrix with only ones |
| zeros (x), zeros (x,y) | A matrix with only zeros |
| diag ([x y z]) | Diagonal matrix |
| size (A) | Dimension of matrix A |
| A' | Inverse of matrix A |

CALLING FUNCTIONS IN MATHSCRIPT

MathScript includes more than 800 built-in functions that you can use, e.g., in a previous task you used the **plot** function.

Example: Built-in Functions

Given the vector:

```
>>x=[1 2 5 6 8 9 3]
```

→ Find the **mean** value of the vector x.

→ Find the **minimum** value of the vector x.

→ Find the **maximum** value of the vector x.

The MathScript Code is:

```
x=[1 2 5 6 8 9 3]
mean(x)
min(x)
max(x)
```

[End of Example]

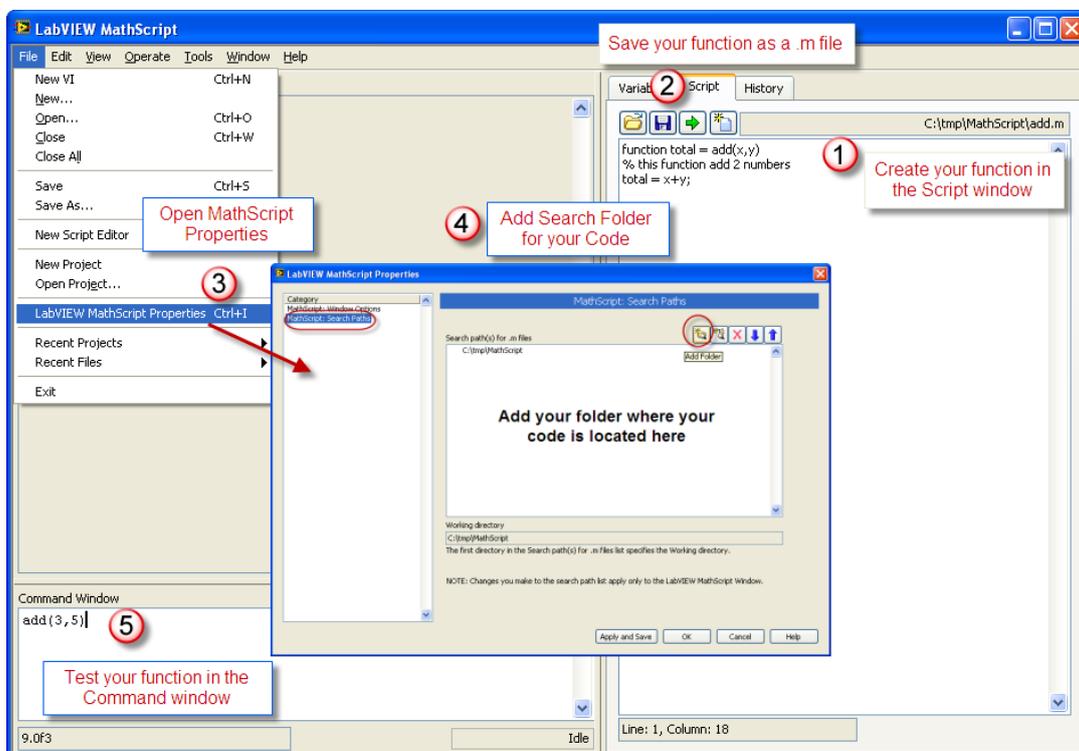
USER-DEFINED FUNCTIONS IN MATHSCRIPT

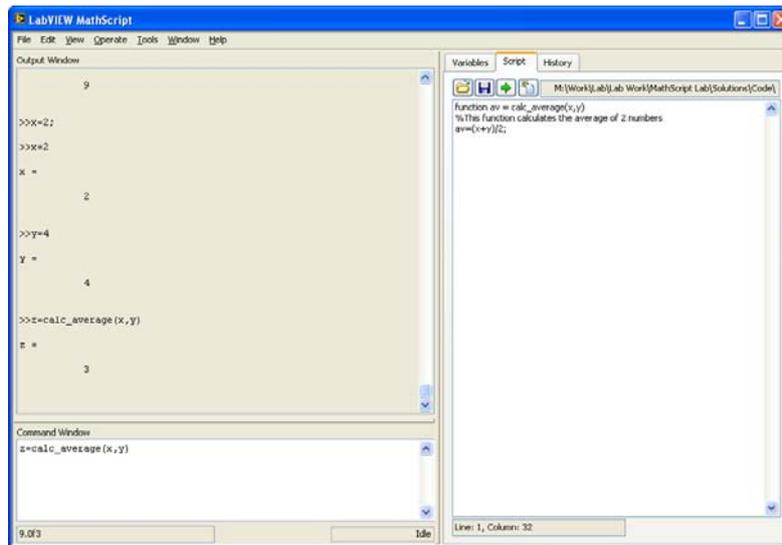
MathScript includes more than 800 built-in functions that you can use but sometimes you need to create your own functions.

To define your own function in MathScript, use the following syntax:

```
function outputs = function_name(inputs)
% documentation
...
```

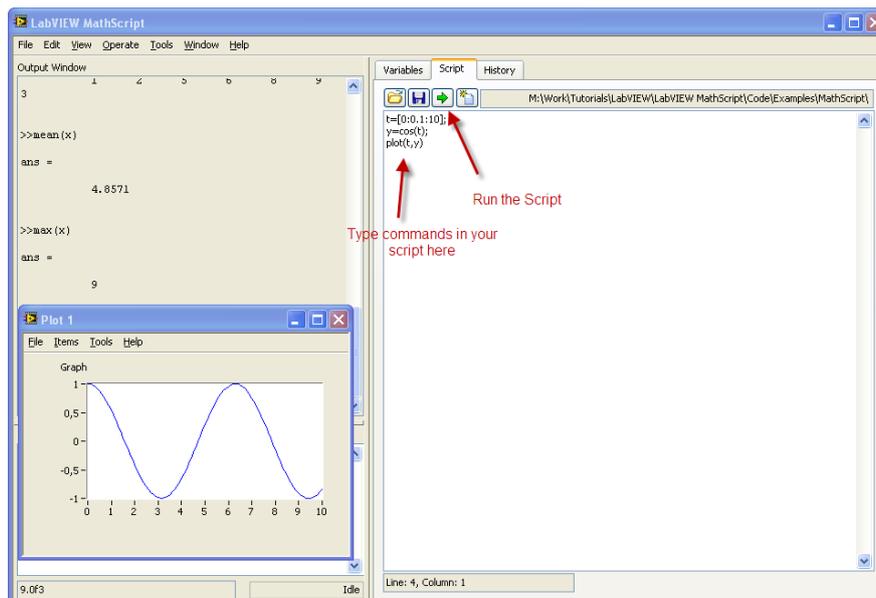
Here is the procedure for creating a user-defined function in MathScript:



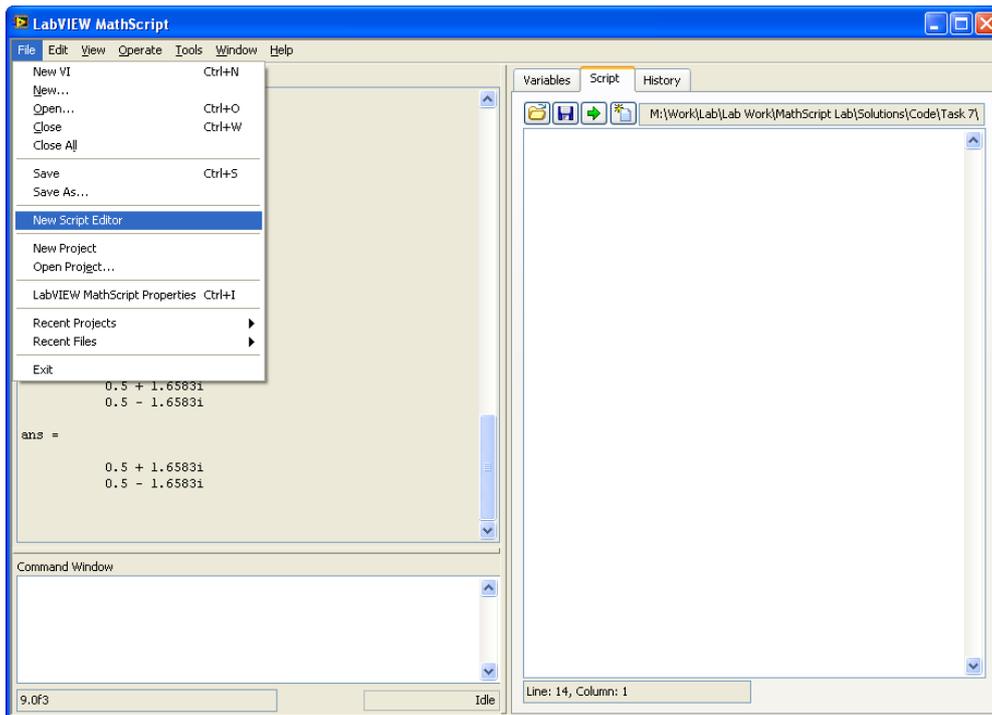


SCRIPTS

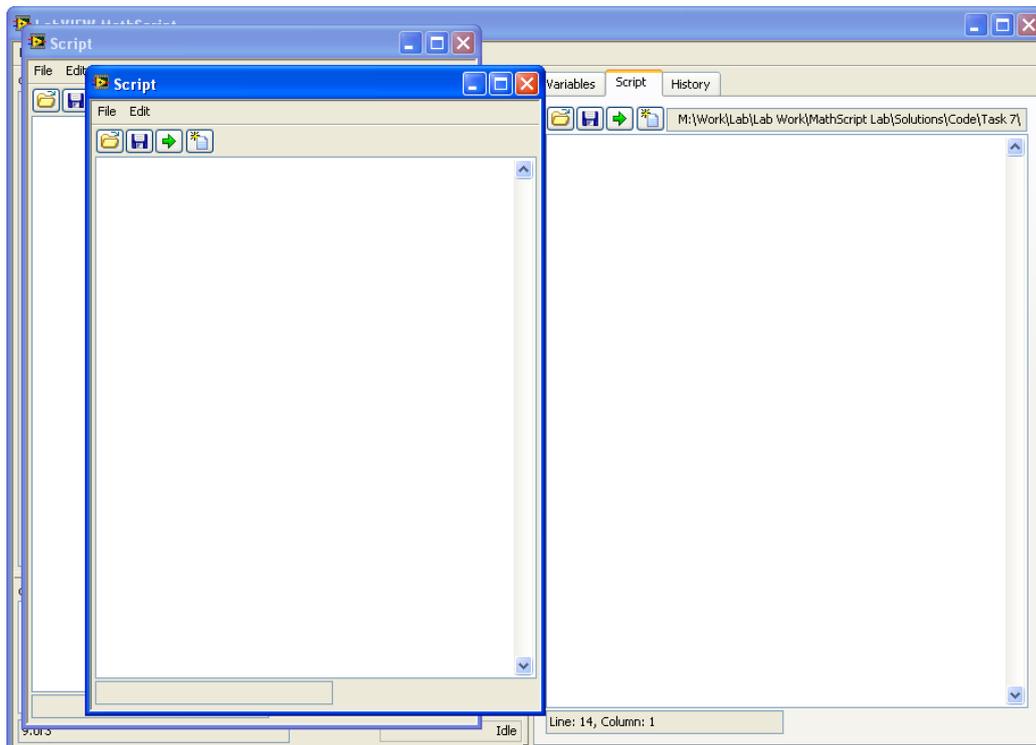
A script is a sequence of MathScript commands that you want to perform to accomplish a task. When you have created the script you may save it as a m-file for later use.



You may also have multiple Script Windows open at the same time by selecting **"New Script Editor"** in the File menu:



This gives:



3.5 FLOW CONTROL

This chapter explains the basic concepts of flow control in MathScript.

The topics are as follows:

- If-else statement
- Switch and case statement
- For loop
- While loop

3.5.1 IF-ELSE STATEMENT

The if statement evaluates a logical expression and executes a group of statements when the expression is true. The optional elseif and else keywords provide for the execution of alternate groups of statements. An end keyword, which matches the if, terminates the last group of statements. The groups of statements are delineated by the four keywords—no braces or brackets are involved.

Example: If-Else Statement

Test the following code:

```
n=5
if n > 2
    M = eye(n)
elseif n < 2
    M = zeros(n)
else
    M = ones(n)
end
```

[End of Example]

3.5.2 SWITCH AND CASE STATEMENT

The switch statement executes groups of statements based on the value of a variable or expression. The keywords case and otherwise delineate the groups. Only the first matching case is executed. There must always be an end to match the switch.

Example: Switch and Case Statement

Test the following code:

```
n=2
switch(n)
    case 1
        M = eye(n)
    case 2
        M = zeros(n)
```

```
    case 3
        M = ones(n)
    end
```

[End of Example]

3.5.3 FOR LOOP

The for loop repeats a group of statements a fixed, predetermined number of times. A matching end delineates the statements.

Example: For Loop

Test the following code:

```
m=5
for n = 1:m
    r(n) = rank(magic(n));
end
r
```

[End of Example]

3.5.4 WHILE LOOP

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching end delineates the statements.

Example: While Loop

Test the following code:

```
m=5;
while m > 1
    m = m - 1;
    zeros(m)
end
```

[End of Example]

3.6 PLOTTING

This chapter explains the basic concepts of creating plots in MathScript.

Topics:

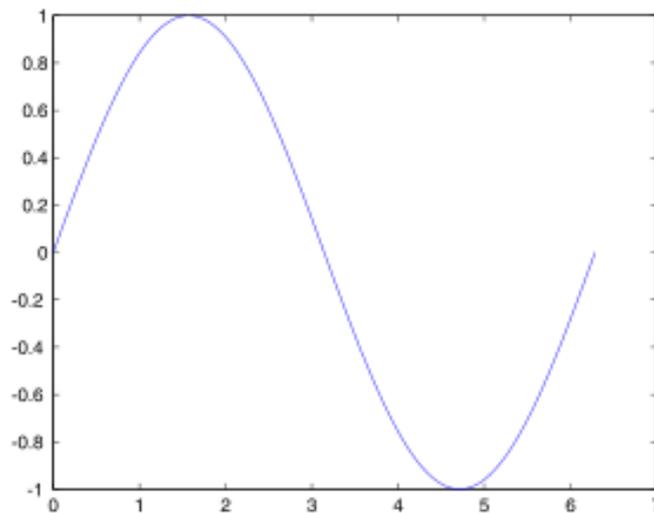
- Basic Plot commands

Example: Plotting

Function plot can be used to produce a graph from two vectors x and y. The code:

```
x = 0:pi/100:2*pi;  
y = sin(x);  
plot(x,y)
```

produces the following figure of the sine function:



[End of Example]

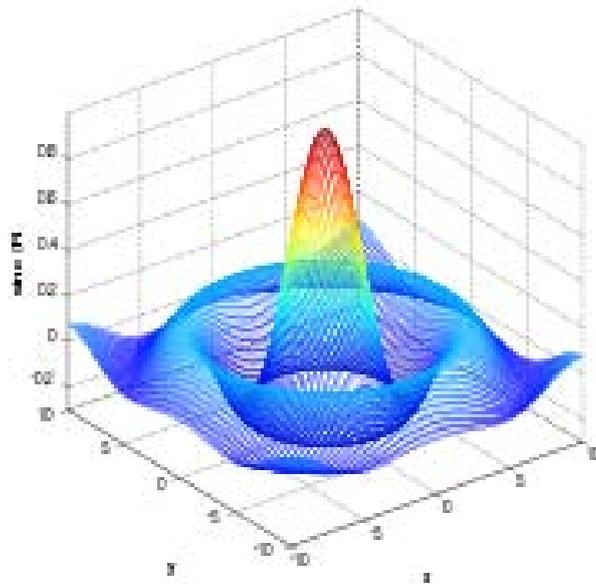
Example: Plotting

Three-dimensional graphics can be produced using the functions surf, plot3 or mesh.

```
[X,Y] = meshgrid(-10:0.25:10,-10:0.25:10);  
f = sinc(sqrt((X/pi).^2+(Y/pi).^2));  
mesh(X,Y,f);
```

```
axis([-10 10 -10 10 -0.3 1])  
xlabel('\bfx')  
ylabel('\bfy')  
zlabel('\bfsinc) ({\bfR})')  
hidden off
```

This code produces the following 3D plot:



[End of Example]



4 LINEAR ALGEBRA EXAMPLES

Requires: **MathScript RT Module**

Linear algebra is a branch of mathematics concerned with the study of matrices, vectors, vector spaces (also called linear spaces), linear maps (also called linear transformations), and systems of linear equations.

MathScript are well suited for Linear Algebra.

4.1 VECTORS

Given a vector x

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in R^n$$

Example: Vectors

Given the following vector

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

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

The **Transpose** of vector x :

$$x^T = [x_1 \ x_2 \ \dots \ x_n] \in R^{1 \times n}$$

```
>> x'
ans =
     1     2     3
```

[End of Example]

The **Length** of vector x :

$$\|x\| = \sqrt{x^T x} = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

Orthogonality:

$$x^T y = 0$$

4.2 MATRICES

Given a matrix A :

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \in R^{n \times m}$$

Example: Matrices

Given the following matrix:

$$A = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}$$

```
>> A=[0 1;-2 -3]
```

```
A =
     0     1
    -2    -3
```

[End of Example]

4.2.1 TRANSPOSE

The **Transpose** of matrix A :

$$A^T = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \in R^{m \times n}$$

Example: Transpose

Given the matrix:

$$A^T = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}^T = \begin{bmatrix} 0 & -2 \\ 1 & -3 \end{bmatrix}$$

```
>> A'
```

```
ans =
     0    -2
     1    -3
```

[End of Example]

4.2.2 DIAGONAL

The **Diagonal** elements of matrix A is the vector

$$\text{diag}(A) = \begin{bmatrix} a_{11} \\ a_{22} \\ \vdots \\ a_{pp} \end{bmatrix} \in R^{p=\min(x,m)}$$

Example: Diagonal

Find the diagonal elements of matrix A :

```
>> diag(A)
ans =
     0
    -3
```

[End of Example]

The **Diagonal** matrix Λ is given by:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \in R^{n \times n}$$

Given the **Identity** matrix I :

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \in R^{n \times m}$$

Example: Identity Matrix

Get the 3x3 Identity matrix:

```
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

[End of Example]

4.2.3 TRIANGULAR

Lower Triangular matrix L :

$$L = \begin{bmatrix} \cdot & 0 & 0 \\ \vdots & \ddots & 0 \\ \cdot & \dots & \cdot \end{bmatrix}$$

Upper Triangular matrix U :

$$U = \begin{bmatrix} \cdot & \dots & \cdot \\ 0 & \ddots & \vdots \\ 0 & 0 & \cdot \end{bmatrix}$$

4.2.4 MATRIX MULTIPLICATION

Given the matrices $A \in R^{n \times m}$ and $B \in R^{m \times p}$, then

$$C = AB \in R^{n \times p}$$

where

$$c_{jk} = \sum_{l=1}^m a_{jl} b_{lk}$$

Example: Matrix Multiplication

Matrix multiplication:

```
>> A=[0 1;-2 -3]
A =
     0     1
    -2    -3
>> B=[1 0;3 -2]
B =
     1     0
     3    -2
>> A*B
ans =
     3    -2
    -11     6
```

[End of Example]

Note!

$$AB \neq BA$$

$$A(BC) = (AB)C$$

$$(A + B)C = AC + BC$$

$$C(A + B) = CA + CB$$

4.2.5 MATRIX ADDITION

Given the matrices $A \in R^{n \times m}$ and $B \in R^{n \times m}$, then

$$C = A + B \in R^{n \times m}$$

Example: Matrix Addition

Matrix addition:

```
>> A=[0 1;-2 -3]
>> B=[1 0;3 -2]
>> A+B
ans =
     1     1
     1    -5
```

[End of Example]

4.2.6 DETERMINANT

Given a matrix $A \in R^{n \times n}$, then the **Determinant** is given:

$$\det(A) = |A|$$

Given a 2x2 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in R^{2 \times 2}$$

Then

$$\det(A) = |A| = a_{11}a_{22} - a_{21}a_{12}$$

Example: Determinant

Find the determinant:

```
A =
     0     1
```

```

-2  -3
>> det(A)
ans =
     2

```

Notice that

$$\det(AB) = \det(A) \det(B)$$

and

$$\det(A^T) = \det(A)$$

[End of Example]

Example: Determinant

Determinants:

```

>> det(A*B)
ans =
    -4
>> det(A)*det(B)
ans =
    -4
>> det(A')
ans =
     2
>> det(A)
ans =
     2

```

[End of Example]

4.2.7 INVERSE MATRICES

The **inverse** of a quadratic matrix $A \in R^{n \times n}$ is defined by:

$$A^{-1}$$

if

$$AA^{-1} = A^{-1}A = I$$

For a 2x2 matrix we have:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in R^{2 \times 2}$$

The inverse A^{-1} is given by

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \in R^{2 \times 2}$$

Example: Inverse Matrices

Inverse matrix:

```
A =
    0     1
   -2    -3

>> inv(A)

ans =
   -1.5000   -0.5000
    1.0000     0
```

[End of Example]

Notice that:

$$AA^{-1} = A^{-1}A = I$$

→ Prove this in MathScript

4.3 EIGENVALUES

Given $A \in R^{n \times n}$, then the Eigenvalues is defined as:

$$\det(\lambda I - A) = 0$$

Example: Eigenvalues

Find the Eigenvalues:

```
A =
    0     1
   -2    -3

>> eig(A)

ans =
   -1
   -2
```

[End of Example]

4.4 SOLVING LINEAR EQUATIONS

Given the linear equation

$$Ax = b$$

with the solution:

$$x = A^{-1}b$$

(Assuming that the inverse of A exists)

Example: Solving Linear Equations

Solving the following equation:

The equations

$$\begin{aligned}x_1 + 2x_2 &= 5 \\ 3x_1 + 4x_2 &= 6\end{aligned}$$

may be written

$$Ax = b$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

where

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$b = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

The solution is:

```
A =
     1     2
     3     4
>> b=[5;6]
b =
     5
     6
>> x=inv(A)*b
x =
    -4.0000
```

```
4.5000
```

In MathScript you could also write “ $x=A\b$ ”, which should give the same answer. This syntax can also be used when the inverse of A don't exists.

[End of Example]

Example: Solving Linear Equations

Illegal operation:

```
>> A=[1 2;3 4;7 8]
>> x=inv(A)*b
??? Error using ==> inv
Matrix must be square.
>> x=A\b
x =
    -3.5000
     4.1786
```

[End of Example]

4.5 LU FACTORIZATION

LU factorization of $A \in R^{n \times m}$ is given by

$$A = LU$$

where

L is a lower triangular matrix

U is a upper triangular matrix

The MathScript syntax is `[L,U]=lu(A)`

Example: LU Factorization

Find L and U :

```
>> A=[1 2;3 4]
>> [L,U]=lu(A)
L =
    0.3333    1.0000
    1.0000     0
```

```

U =
    3.0000    4.0000
         0    0.6667

```

[End of Example]

Or sometimes LU factorization of $A \in R^{n \times m}$ is given by

$$A = LU = LDU$$

where

D is a diagonal matrix

The MathScript syntax is `[L,U,P]=lu(A)`

Example: LU Factorization

Find L , U and P :

```

>> A=[1 2;3 4]
A =
    1    2
    3    4
>> [L,U,P]=lu(A)
L =
    1.0000    0
    0.3333    1.0000
U =
    3.0000    4.0000
         0    0.6667
P =
    0    1
    1    0

```

[End of Example]

4.6 THE SINGULAR VALUE DECOMPOSITION (SVD)

The **Singular value Decomposition** (SVD) of the matrix $A \in R^{n \times m}$ is given by

$$A = USV^T$$

where

U is a orthogonal matrix

V is a orthogonal matrix

S is a diagonal singular matrix

Example: SVD Decomposition

Find S , V and D :

```
>> A=[1 2;3 4];
>> [U,S,V] = svd(A)
U =
    -0.4046    -0.9145
    -0.9145     0.4046
S =
    5.4650     0
         0     0.3660
V =
    -0.5760     0.8174
    -0.8174    -0.5760
```

[End of Example]

4.7 COMMANDS

| Command | Description |
|------------------------------|-------------------------------------|
| [L,U]=lu(A) [L,U,P]=lu(A) | LU Factorization |
| [U,S,V] = svd(A) | Singular Value Decomposition (SVD) |



5 CONTROL DESIGN AND SIMULATION

Using LabVIEW MathScript for Control Design purposes you need to install the “Control Design and Simulation Module” in addition to the “MathScript RT Module” itself.

Use the Control Design MathScript RT Module functions to design, analyze, and simulate linear controller models using a text-based language. The following is a list of Control Design MathScript RT Module classes of functions and commands that LabVIEW MathScript supports.

Getting help about MathScript functions regarding the Control Design Toolkit (CDT), type “[help cdt](#)” in the Command Window in the MathScript environment.

The following function classes exist:

| Class | Description |
|---------------------------|---------------------------------------|
| cdops | Arithmetic operator functions |
| cdplots | XY plane functions |
| cdsolvers | Equation solver functions |
| connect | Model interconnection functions |
| construct | Model construction functions |
| convert | Model conversion functions |
| dynchar | Dynamic characteristics functions |
| frqrsp | Frequency response analysis functions |
| info | Model information functions |
| reduce | Model reduction functions |
| ssanals | State-space analysis functions |
| ssdesign | State-feedback design functions |
| timeresp | Time response analysis functions |

We will go through some of the classes and function in detail below:

5.1 STATE-SPACE MODELS AND TRANSFER FUNCTIONS

MathScript offers lots of functions for defining and manipulate state-space models and transfer functions.

Class: [construct](#)

Description:

Use functions in the construct class to construct linear time-invariant system models and to convert between model forms.

Below we see the different functions available in the construct class:

| Function | Description |
|----------------------------|---|
| drandss | Generates a random discrete state-space system model |
| drandtf | Generates a discrete random system model in transfer function form |
| drandzpk | Generates a discrete random system model in zero-pole-gain (ZPK) form |
| pid | Constructs a proportional-integral-derivative (PID) controller model |
| randss | Generates a continuous random state-space system model |
| randtf | Generates a continuous random system model in transfer function form |
| randzpk | Generates a continuous random system model in zero-pole-gain (ZPK) form |
| ss | Creates a system model in, or converts a model to, state-space form |
| sys_filter | Constructs a digital filter in transfer function form |
| sys_order1 | Constructs the components of a first-order system model |
| sys_order2 | Constructs the components of a second-order system model |
| tf | Creates a system model in, or converts a model to, transfer function form |
| zpk | Constructs a system model in, or converts a model to, zero-pole-gain form |

Below we will give some examples of how to use the most import functions in this class.

5.1.1.1 PID

Currently, the Proportional-Integral-Derivative (PID) algorithm is the most common control algorithm used in industry.

In PID control, you must specify a process variable and a setpoint. The process variable is the system parameter you want to control, such as temperature, pressure, or flow rate, and the setpoint is the desired value for the parameter you are controlling.

A PID controller determines a controller output value, such as the heater power or valve position. The controller applies the controller output value to the system, which in turn drives the process variable toward the setpoint value.

Then the PID controller calculates the controller action, $u(t)$:

$$u(t) = K_c \left(e + \frac{1}{T_i} \int_0^t e dt + T_d \frac{de}{dt} \right)$$

Where

K_c Controller gain

T_i Integral time

T_d Derivative time

And e is the error

$$e = SP - PV$$

SP – Setpoint

PV – Process Variable

Function: **pid**

Description:

Constructs a proportional-integral-derivative (PID) controller model in either parallel, series, or academic form.

Examples:

```
Kc = 0.5;
Ti = 0.25;
SysOutTF = pid(Kc, Ti, 'academic');
```

[End of Example]

5.1.1.2 STATE-SPACE MODEL

A state-space model is just a structured form or representation of the differential equations for a system.

A linear State-space model:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

where x is the state vector and u is the input vector. A is called the system-matrix, and is square in all cases.

Example:

The differential equations:

$$\dot{x}_1 = -2x_1 + 6u$$

$$\dot{x}_2 = 2x_1$$

May be written on state-space form:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 6 \\ 0 \end{bmatrix} u$$

Function: **ss**

Description:

This function constructs a continuous or discrete linear system model in state-space form. You also can use this function to convert transfer function and zero-pole-gain models to state-space form.

Examples:

```
% Creates a state-space model
A = eye(2)
```

```

B = [0; 1]
C = B'
SysOutSS = ss(A, B, C)

% Converts a zero-pole-gain model to state-space form
z = 1
p = [1, -1]
k = 1
SysIn = zp(z, p, k)
SysOutSS = ss(SysIn)

```

[End of Example]

5.1.3 TRANSFER FUNCTION

The transfer function of a linear system is defined as the ratio of the Laplace transform of the output variable to the Laplace transform of the input variable.

$$H(S) = \frac{y(s)}{u(s)}$$

Function **tf**

Description:

This function creates a continuous or discrete linear system model in transfer function form. You also can use this function to convert zero-pole-gain and state-space models to transfer function form.

Examples:

```
>>s = tf('s')
```

This specifies that you want to create the continuous transfer function $s / 1$. After you enter this command, you can use LabVIEW MathScript operands on this transfer function to define a zero-pole-gain or transfer function model.

```
SysOutZPK = 4*(s + 2) / (s + 1)
```

This example constructs a zero-pole-gain model with a gain of 4, a zero at -2, and a pole at -1.

```
SysOutTF = (3*(s*s*s) + 2) / (4*(s*s*s*s) + 8)
```

This example constructs the transfer function model $3s^3 + 2 / 4s^4 + 8$.

[End of Example]

5.1.4 FIRST ORDER SYSTEMS

The following transfer function defines a first order system:

$$H(s) = \frac{K}{Ts + 1}$$

Where

K is the gain

T is the Time constant

Function [sys_order1](#)

Description:

This function constructs the components of a first-order system model based on a gain, time constant, and delay that you specify. You can use this function to create either a state-space model or a transfer function model, depending on the output parameters you specify.

Inputs:

K Specifies the gain matrix. K is a real matrix.

tau Specifies the time constant, in seconds, which is the time required for the model output to reach 63% of its final value. The default value is 0.

delay Specifies the response delay of the model, in seconds. The default value is 0.

Examples:

```
K = 0.5;
tau = 1.5;
SysOutTF = sys_order1(K, tau);
```

[End of Example]

5.1.5 SECOND ORDER SYSTEMS

A standard **second order transfer** function model may be written like this:

$$H(s) = \frac{y(s)}{u(s)} = \frac{K\omega_0^2}{s^2 + 2\zeta\omega_0s + \omega_0^2} = \frac{K}{\left(\frac{s}{\omega_0}\right)^2 + 2\zeta\frac{s}{\omega_0} + 1}$$

Where

K is the gain

ζ (zeta) is the relative damping factor

ω_0 [rad/s] is the undamped resonance frequency.

Function `sys_order2`

Description:

This function constructs the components of a second-order system model based on a damping ratio and natural frequency you specify. You can use this function to create either a state-space model or a transfer function model, depending on the output parameters you specify.

Example:

Examples of how to use the `sys_order2` function:

```
dr = 0.5
wn = 20
[num, den] = sys_order2(wn, dr)
SysTF = tf(num, den)
[A, B, C, D] = sys_order2(wn, dr)
SysSS = ss(A, B, C, D)
```

[End of Example]

Class: `connect`

Description:

Use members of the `connect` class to connect systems models together in various configurations.

Below we see the different functions available in the `connect` class:

| Function | Description |
|-----------------------|---|
| <code>append</code> | Appends system models together |
| <code>diag</code> | Constructs a system model whose diagonal contains copies of another model |
| <code>feedback</code> | Connects two system models in a closed-loop configuration |
| <code>hconcat</code> | Horizontally concatenates two or more system models. |
| <code>parallel</code> | Connects two system models together in a parallel configuration |
| <code>series</code> | Connects two system models together in a serial configuration |
| <code>vconcat</code> | Vertically concatenates two or more system models |

Function `series`

Description:

This function connects two system models in series to produce a model SysSer with input and output connections you specify. The input models must be either continuous models or discrete models with identical sampling times.

Example:

Here is an example of how to use the **series** function.

```

SysIn_1 = tf([1, 1], [1 -1, 3])
SysIn_2 = zpk([1], [1, -1], 1)
SysSer = series(SysIn_1, SysIn_2)

```

[End of Example]

Class: **convert**

Description:

Use members of the convert class to convert a continuous system model to a discrete model, convert a discrete model to a continuous model, and resample a discrete model. You also can use members of this class to incorporate delays into a system model.

Below we see the different functions available in the convert class:

| Function | Description |
|---------------------------------|---|
| c to d | Converts a continuous system model to a discrete model |
| d to c | Converts a discrete system model to a continuous one |
| d to d | Resamples a discrete system model |
| delay to z | Incorporates delays into a discrete system model |
| distributedelay | Minimizes transport delay in a system model |
| pade | Incorporates delays into a continuous system model by using Pade approximation |
| polycoef | Specifies whether transfer function coefficients are in ascending or descending direction |
| ss to ss | Applies a state transformation to a system model |

5.1.6 PADÉ-APPROXIMATION

The Transfer function of a time-delay is:

$$H(s) = e^{-\tau s}$$

In some situations it is necessary to substitute $e^{-\tau s}$ with an approximation, e.g., the **Padé-approximation**:

$$e^{-\tau s} \approx \frac{1 - k_1 s + k_2 s^2 + \dots \pm k_n s^n}{1 + k_1 s + k_2 s^2 + \dots + k_n s^n}$$

Function: **pade**

Description:

This function incorporates time delays into a system model using the Pade approximation method, which converts all residuals. You must specify the delay using the set function. You also can use this function to calculate coefficients of numerator and denominator polynomial functions with a specified delay.

Example:

Examples of how to use the `pade` function:

```

SysCon = zpk(1, 3.2, 6)
SysCon = set(SysCon, 'inputdelay', 6, 'outputdelay', 1.1)
SysDel = pade(SysCon, 2)

delay = 1.2
order = 3
[num, den] = pade(delay, order)

```

[End of Example]

5.2 FREQUENCY RESPONSE ANALYSIS

The frequency response of a system is a frequency dependent function which expresses how a sinusoidal signal of a given frequency on the system input is transferred through the system. Each frequency component is a sinusoidal signal having a certain amplitude and a certain frequency.

The frequency response is an important tool for analysis and design of signal filters and for analysis and design of control systems.

The frequency response can found experimentally or from a transfer function model.

The frequency response of a system is defined as the steady-state response of the system to a sinusoidal input signal. When the system is in steady-state it differs from the input signal only in amplitude (A) and phase angle (ω).

If we have the input signal:

$$u(t) = U \sin \omega t$$

The steady-state output signal will be:

$$y(t) = UA \sin (\omega t + \phi)$$

A and ϕ is a function of the frequency ω so we may write $A = A(\omega), \phi = \phi(\omega)$

For a transfer function

$$H(S) = \frac{y(s)}{u(s)}$$

We have:

$$A(\omega) = |H(j\omega)|$$

$$\phi(\omega) = \angle H(j\omega)$$

Where $H(j\omega)$ is the frequency response of the system, i.e., we may find the frequency response by setting $s = j\omega$ in the transfer function.

5.2.1 BODE DIAGRAM

Bode diagrams are useful in frequency response analysis. The Bode diagram consists of 2 diagrams, the Bode magnitude diagram, $A(\omega)$ and the Bode phase diagram, $\phi(\omega)$.

The $A(\omega)$ -axis is in decibel (dB)

Where the decibel value of x is calculated as: $x[dB] = 20\log_{10}x$

The $\phi(\omega)$ -axis is in degrees (not radians)

Function: [bode](#)

Description:

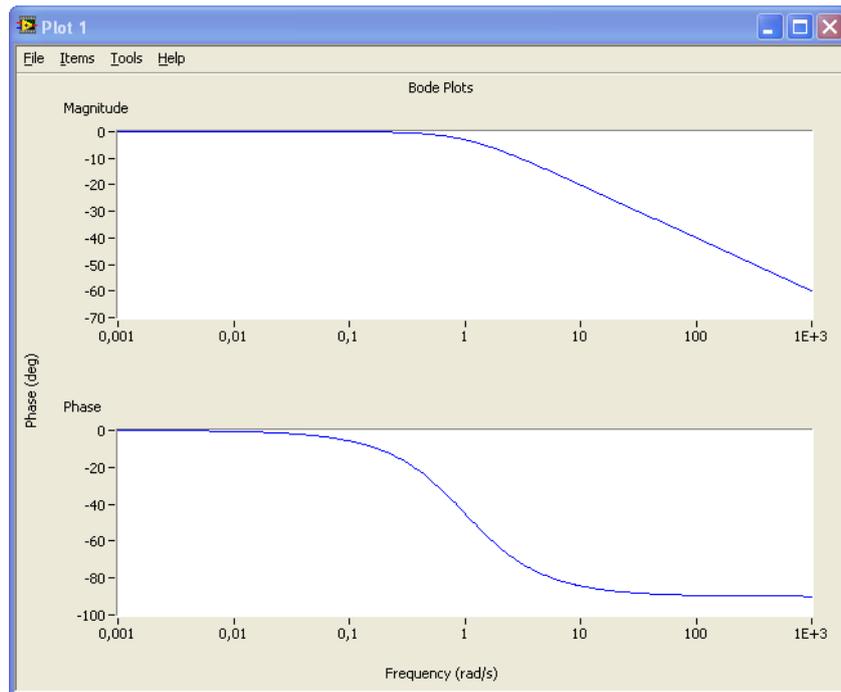
This function creates the Bode magnitude and Bode phase plots of a system model. You also can use this function to return the magnitude and phase values of a model at frequencies you specify. If you do not specify an output, this function creates a plot.

Examples:

We have the following transfer function

$$H(s) = \frac{y(s)}{u(s)} = \frac{1}{s + 1}$$

We want to plot the Bode diagram for this transfer function:



In MathScript we could write:

```
num=[ 1 ];
den=[ 1, 1 ];
H1=tf(num,den)
bode(H1)
```

[End of Example]

Function: [margin](#)

Description:

This function calculates and/or plots the smallest gain and phase margins of a single-input single-output (SISO) system model.

The gain margin indicates where the frequency response crosses at 0 decibels ("crossover frequency", ω_c).

$$|H(j\omega_c)|$$

ω_c is also the bandwidth of the system

The phase margin indicates where the frequency response crosses -180 degrees ("crossover frequency", ω_{180}).

$$\angle H(j\omega_{180})$$

Examples:

The following example illustrates the use of the margin function.

```
num = [1]
den = [1, 5, 6]
H = tf(num, den)
margin(H)
```

[End of Example]

Example:

Given the following system:

$$H(S) = \frac{1}{s(s+1)^2}$$

We want to plot the Bode diagram and find the crossover-frequencies for the system using MathScript.

We use the following functions: **tf**, **bode**, **margins** and **margin**.

- **gmf** is the gain margin frequencies, in radians/second. A gain margin frequency indicates where the model phase crosses -180 degrees.
- **gm** Returns the gain margins of the system.
- **pmf** Returns the phase margin frequencies, in radians/second. A phase margin frequency indicates where the model magnitude crosses 0 decibels.
- **pm** Returns the phase margins of the system.
- We get:

```

LabVIEW MathScript
File Edit View Operate Tools Window Help
Output Window
phase_data =
    0.01    -91.146
    0.1     -101.42
    0.2     -112.62
    0.5     -143.13
    1       -180
    10      -258.58
    100     -268.85

gmf =
    1.9972

gm =
    0.99931

pmf =
    21.386

pm =
    0.68233

Command Window

9.0f3 Idle
Line: 6, Column: 1

% Transfer function
num=1;
den1=[1,0];
den2=[1,1]
den3=[1,1]

den = conv(den1,conv(den2,den3));
H = tf(num, den)

% Bode Plot
bode(H)

% Margins and Phases
wlist=[0.01, 0.1, 0.2, 0.5, 1, 10, 100];

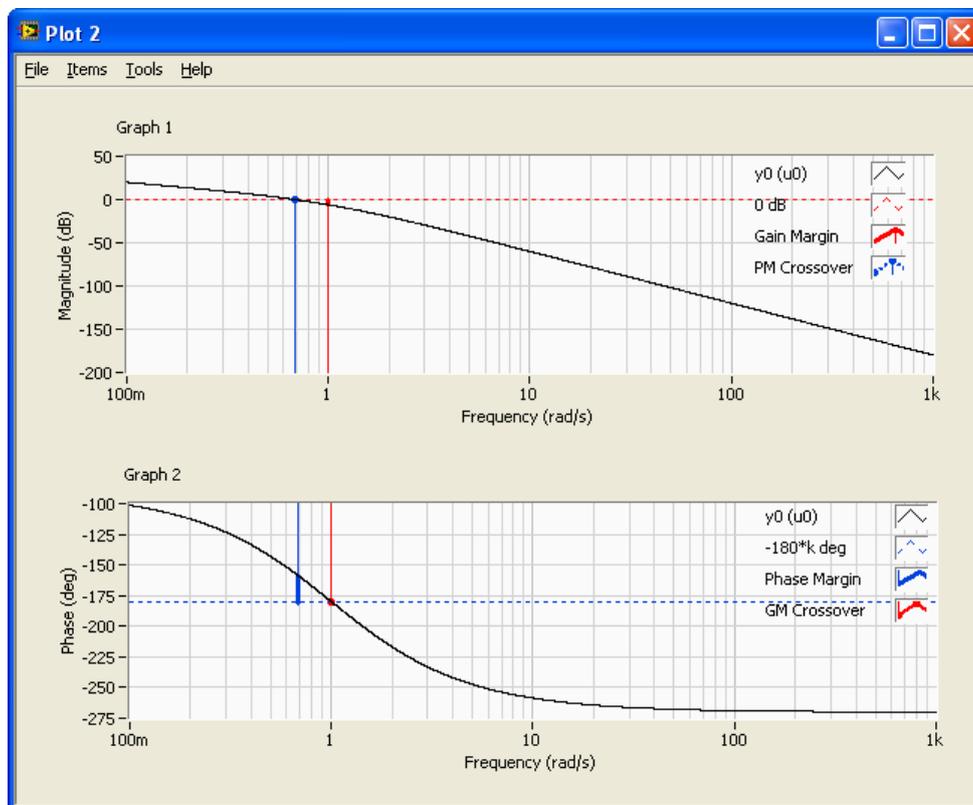
[mag, phase,w] = bode(H, wlist);
magdB=20*log10(mag); %convert to dB

% [mag, phase,w] = bode(H);
mag_data = [w, magdB];
phase_data = [w, phase]

% Crossover Frequency-----
[gmf, gm, pmf, pm] = margins(H)
margin(H)

```

Below we see the Bode diagram with the crossover-frequency and the gain margin and phase margin for the system plotted in:



[End of Example]

TIME RESPONSE

Class: [timeresp](#)

Description:

Use members of the timeresp class to create generic linear simulations and time domain plots for step inputs, impulse inputs, and initial condition responses.

Below we see the different functions available in the timeresp class:

| Function | Description |
|----------------------------|--|
| impulse | Creates the impulse response plot of a system model |
| initial | Creates the initial response plot of a system model |
| lsim | Creates the linear simulation plot of a system model |
| randvector | Generates one or two random vectors |
| step | Creates the step response plot of a system model |

Function: [step](#)

Description:

This function creates a step response plot of the system model. You also can use this function to return the step response of the model outputs. If the model is in state-space form, you also can use this function to return the step response of the model states. This function assumes the initial model states are zero. If you do not specify an output, this function creates a plot.

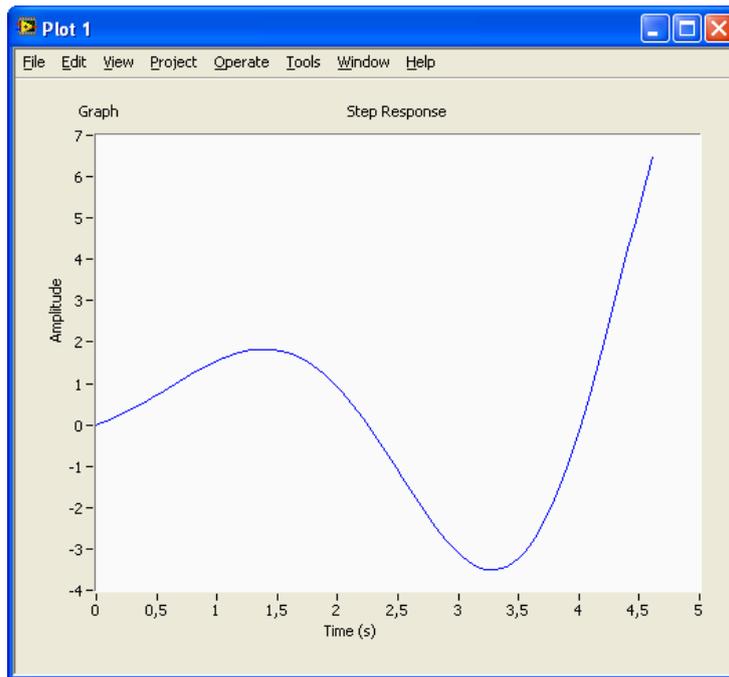
Example:

Given the following system:

$$H(s) = \frac{s + 1}{s^2 - s + 3}$$

We will plot the time response for the transfer function using the **step** function

The result is as follows:



The MathScript code:

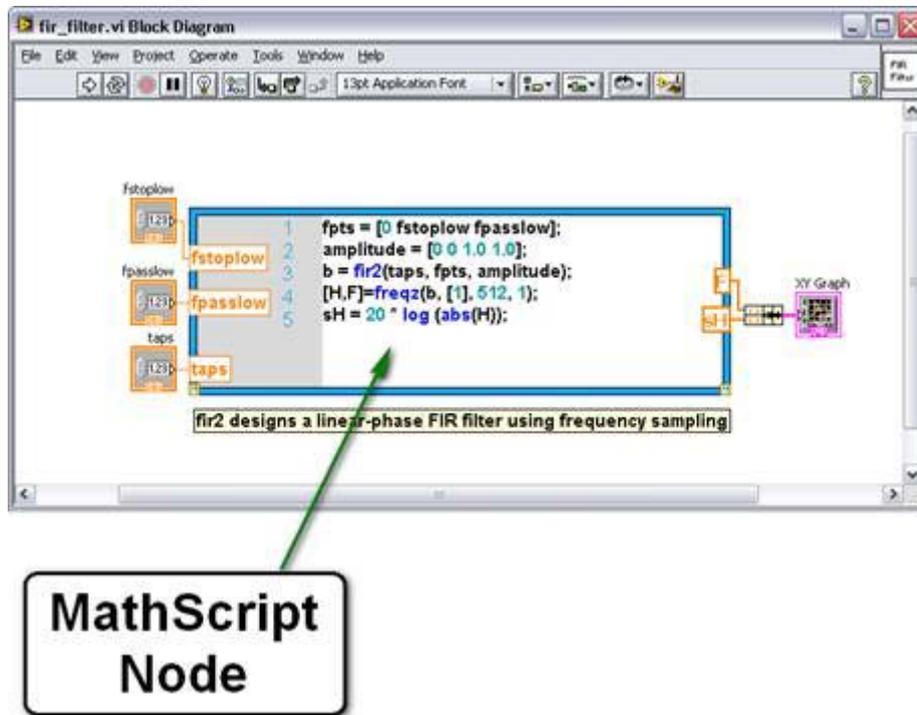
```
H = tf([1, 1], [1, -1, 3])  
step(H)
```

[End of Example]



6 MATHSCRIPT NODE

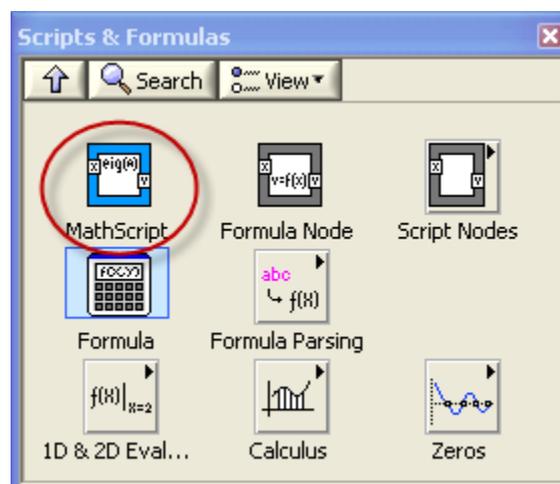
The “MathScript Node” offers an intuitive means of combining graphical and textual code within LabVIEW. The figure below shows the “MathScript Node” on the block diagram, represented by the blue rectangle. Using “MathScript Nodes”, you can enter .m file script text directly or import it from a text file.



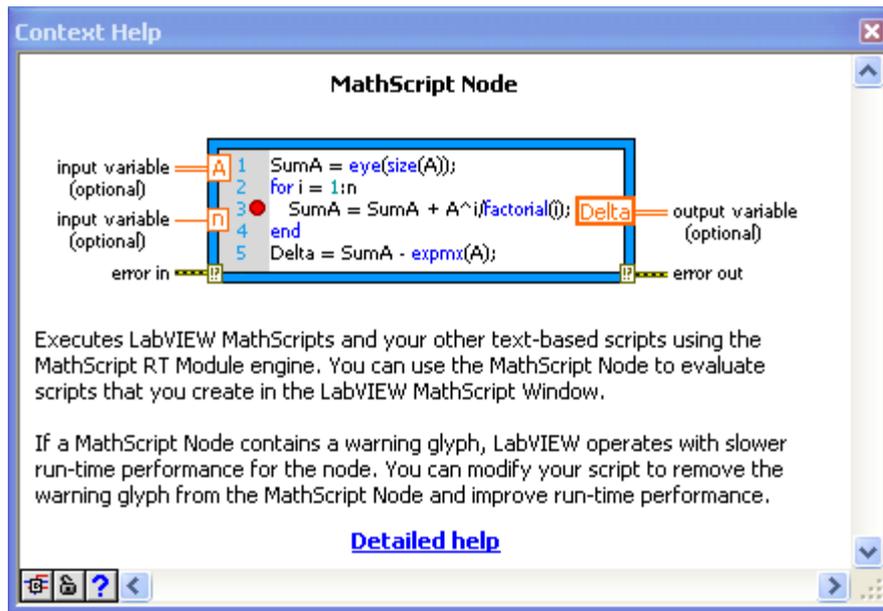
You can define named inputs and outputs on the MathScript Node border to specify the data to transfer between the graphical LabVIEW environment and the textual MathScript code.

You can associate .m file script variables with LabVIEW graphical programming, by wiring Node inputs and outputs. Then you can transfer data between .m file scripts with your graphical LabVIEW programming. The textual .m file scripts can now access features from traditional LabVIEW graphical programming.

The MathScript Node is available from LabVIEW from the Functions Palette: Mathematics → Scripts & Formulas

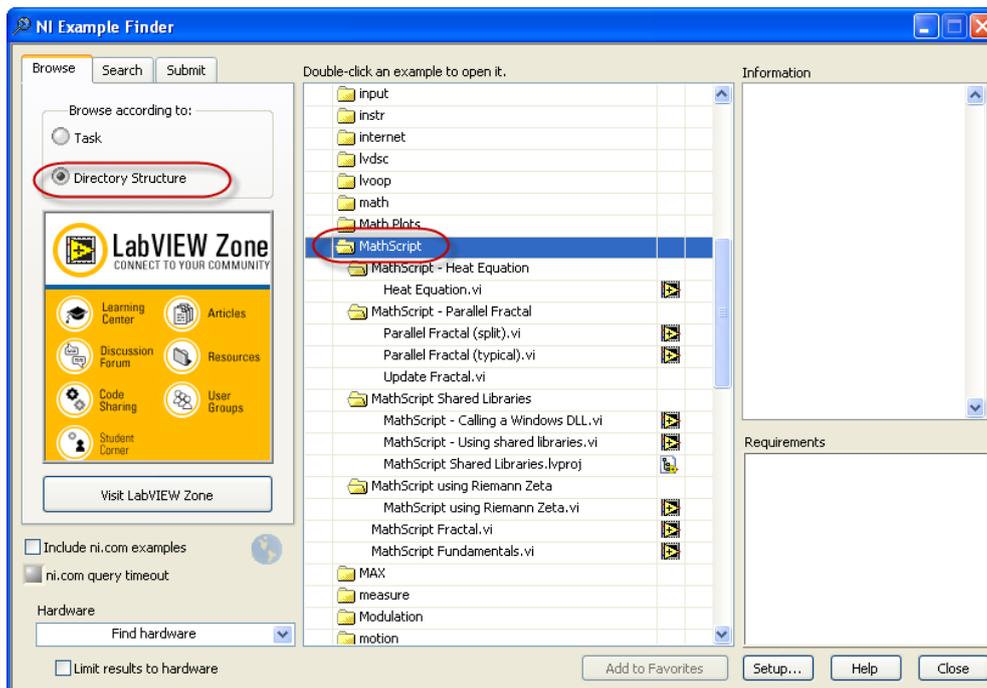


If you click Ctrl+H you get help about the MathScript Node:



Click “Detailed help” in order to get more information about the MathScript Node.

Use the NI Example Finder in order to find examples:



6.1 TRANSFERRING MATHSCRIPT NODES BETWEEN COMPUTERS

If a script in a MathScript Node calls a user-defined function, LabVIEW uses the default search path list to link the function call to the specified .m file. After you configure the default search path list and save the VI that contains the MathScript Node, you do not need to reconfigure the MathScript search path list when you open the VI on a different computer because LabVIEW looks for the .m file in the directory where the .m file was

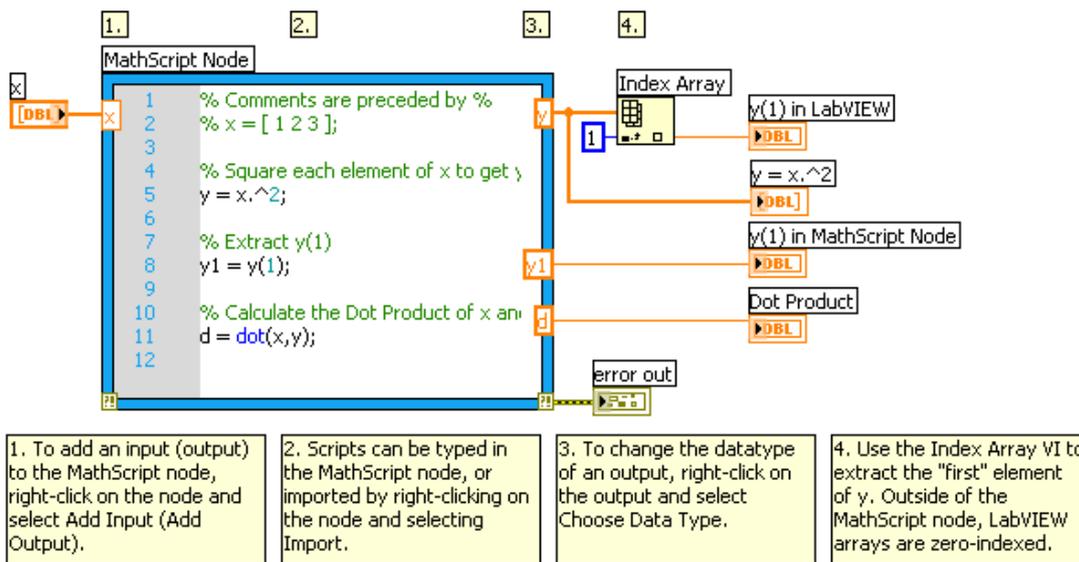
located when you last saved the VI. However, you must maintain the same relative path between the VI and the .m file.

6.2 EXAMPLES

Example: Using the MathScript Node

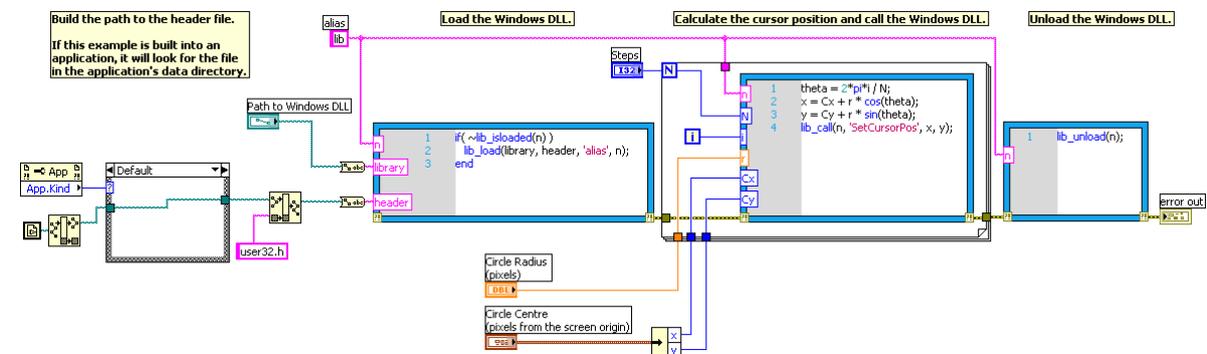
Here is an example of how you use the MathScript Node. On the left border you connect input variables to the script, on the right border you have output variables. Right-click on the border and select “Add Input” or “Add Output”.

The MathScript Node can be found in the Functions >> Mathematics >> Scripts & Formulas Palette.



[End of Example]

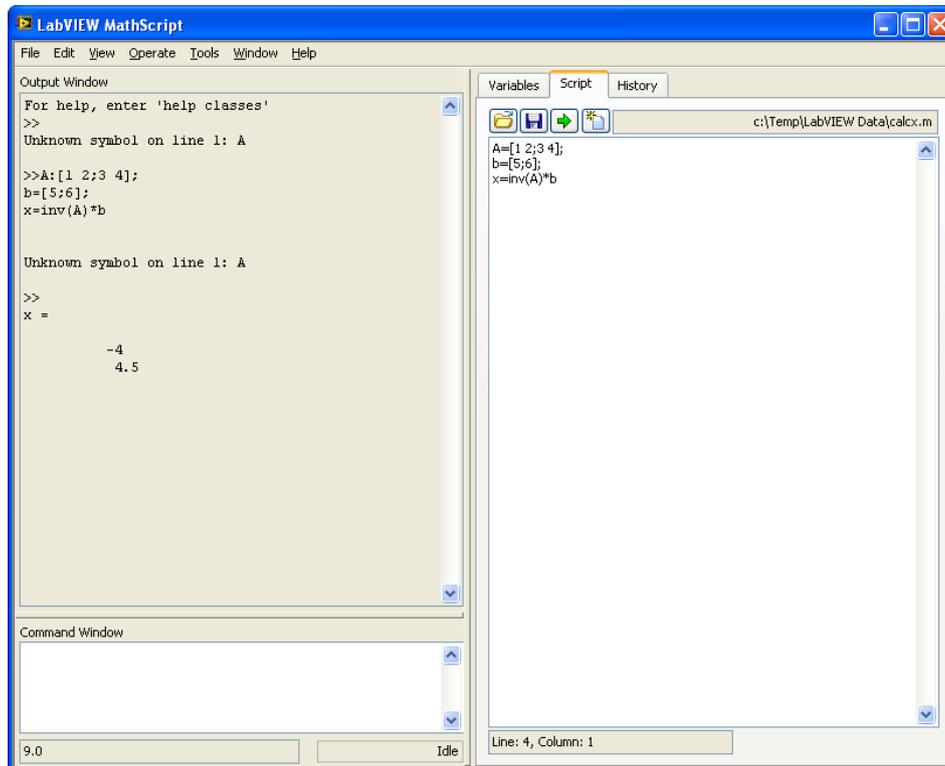
Example: Calling a Windows DLL:



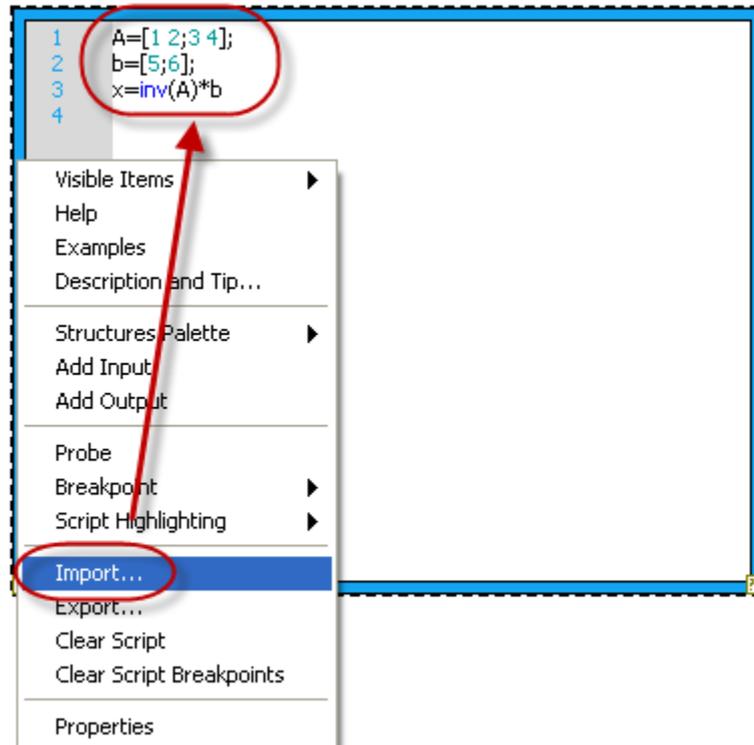
[End of Example]

Example: Using m-files in the MathScript Node:

Use the LabVIEW MathScript to create a m-file script (or you may use MATLAB to create the same script):

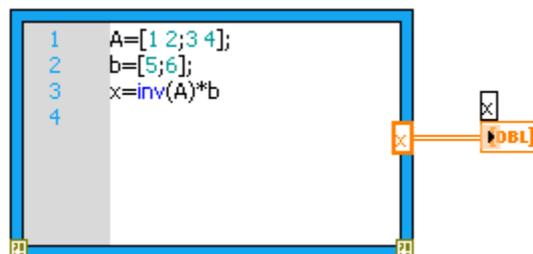


Right-click on the border of the MathScript Node and select "Import", and then select the m-file you want to import into the Node.

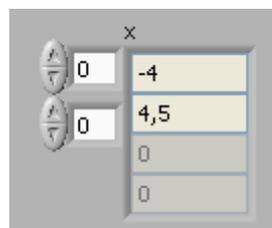


Right-click on the right border and select “Add Output”. Then right-click on the output variable and select “Create Indicator”.

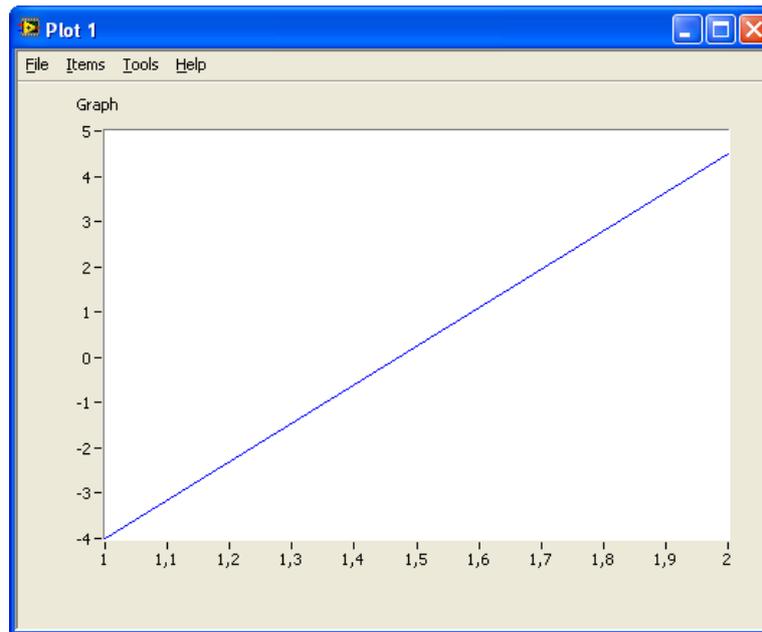
Block Diagram:



The result is as follows (click the Run button):



If you, e.g., add the following command in the MathScript Node: plot(x), the following window appears:



[End of Example]

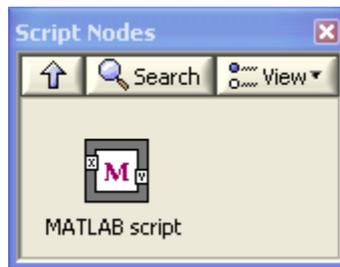
6.3 EXERCISES

Use the MathScript Node and test the same examples you did in the previous chapter (Chapter 4 - "Linear Algebra Examples")



7 MATLAB SCRIPT

The MATLAB Script calls the MATLAB software to execute scripts. You must have a licensed copy of the MATLAB software version 6.5 or later installed on your computer to use MATLAB script nodes because the script nodes invoke the MATLAB software script server to execute scripts written in the MATLAB language syntax. Because LabVIEW uses ActiveX technology to implement MATLAB script nodes, they are available only on Windows.





APPENDIX A – MATHSCRIPT FUNCTIONS FOR CONTROL AND SIMULATION

Here are some descriptions for the most used MathScript functions used in this Lab Work.

| Function | Description | Example |
|-------------------|---|---|
| plot | Generates a plot. <code>plot(y)</code> plots the columns of <code>y</code> against the indexes of the columns. | <pre>>X = [0:0.01:1]; >Y = X.*X; >plot(X, Y)</pre> |
| tf | Creates system model in transfer function form. You also can use this function to state-space models to transfer function form. | <pre>>num=[1]; >den=[1, 1, 1]; >H = tf(num, den)</pre> |
| poles | Returns the locations of the closed-loop poles of a system model. | <pre>>num=[1] >den=[1,1] >H=tf(num,den) >poles(H)</pre> |
| tfinfo | Returns information about a transfer function system model. | <pre>>[num, den, delay, Ts] = tfinfo(SysInTF)</pre> |
| step | Creates a step response plot of the system model. You also can use this function to return the step response of the model outputs. If the model is in state-space form, you also can use this function to return the step response of the model states. This function assumes the initial model states are zero. If you do not specify an output, this function creates a plot. | <pre>>num=[1,1]; >den=[1,-1,3]; >H=tf(num,den); >t=[0:0.01:10]; >step(H,t);</pre> |
| lsim | Creates the linear simulation plot of a system model. This function calculates the output of a system model when a set of inputs excite the model, using discrete simulation. If you do not specify an output, this function creates a plot. | <pre>>t = [0:0.1:10] >u = sin(0.1*pi*t) >lsim(SysIn, u, t)</pre> |
| Sys_order1 | Constructs the components of a first-order system model based on a gain, time constant, and delay that you specify. You can use this function to create either a state-space model or a transfer function model, depending on the output parameters you specify. | <pre>>K = 1; >tau = 1; >H = sys_order1(K, tau)</pre> |
| Sys_order2 | Constructs the components of a second-order system model based on a damping ratio and natural frequency you specify. You can use this function to create either a state-space model or a transfer function model, depending on the output parameters you specify. | <pre>>dr = 0.5 >wn = 20 >[num, den] = sys_order2(wn, dr) >SysTF = tf(num, den) >[A, B, C, D] = sys_order2(wn, dr) >SysSS = ss(A, B, C, D)</pre> |
| damp | Returns the damping ratios and natural frequencies of the poles of a system model. | <pre>>[dr, wn, p] = damp(SysIn)</pre> |
| pid | Constructs a proportional-integral-derivative (PID) controller model in either parallel, series, or academic form. Refer to the LabVIEW Control Design User Manual for information about these three forms. | <pre>>Kc = 0.5; >Ti = 0.25; >SysOutTF = pid(Kc, Ti, 'academic');</pre> |
| conv | Computes the convolution of two vectors or matrices. | <pre>>C1 = [1, 2, 3]; >C2 = [3, 4];</pre> |

| | | |
|-----------------|---|---|
| | | >C = <code>conv</code> (C1, C2) |
| series | Connects two system models in series to produce a model SysSer with input and output connections you specify | >Hseries = <code>series</code> (H1,H2) |
| feedback | Connects two system models together to produce a closed-loop model using negative or positive feedback connections | >SysClosed = <code>feedback</code> (SysIn_1, SysIn_2) |
| ss | Constructs a model in state-space form. You also can use this function to convert transfer function models to state-space form. | >A = <code>eye</code> (2) >B = [0; 1] >C = B' >SysOutSS = <code>ss</code> (A, B, C) |
| ssinfo | Returns information about a state-space system model. | >A = [1, 1; -1, 2] >B = [1, 2]' >C = [2, 1] >D = 0 >SysInSS = <code>ss</code> (A, B, C, D) >[A, B, C, D, Ts] = <code>ssinfo</code> (SysInSS) |
| pade | Incorporates time delays into a system model using the Pade approximation method, which converts all residuals. You must specify the delay using the set function. You also can use this function to calculate coefficients of numerator and denominator polynomial functions with a specified delay. | >[num, den] = <code>pade</code> (delay, order) >[A, B, C, D] = <code>pade</code> (delay, order) |
| bode | Creates the Bode magnitude and Bode phase plots of a system model. You also can use this function to return the magnitude and phase values of a model at frequencies you specify. If you do not specify an output, this function creates a plot. | >num=[4]; >den=[2, 1]; >H = <code>tf</code> (num, den) > <code>bode</code> (H) |
| bodemag | Creates the Bode magnitude plot of a system model. If you do not specify an output, this function creates a plot. | >[mag, wout] = <code>bodemag</code> (SysIn) >[mag, wout] = <code>bodemag</code> (SysIn, [wmin wmax]) >[mag, wout] = <code>bodemag</code> (SysIn, wlist) |
| margin | Calculates and/or plots the smallest gain and phase margins of a single-input single-output (SISO) system model. The gain margin indicates where the frequency response crosses at 0 decibels. The phase margin indicates where the frequency response crosses -180 degrees. Use the margins function to return all gain and phase margins of a SISO model. | >num = [1] >den = [1, 5, 6] >H = <code>tf</code> (num, den) <code>margin</code> (H) |
| margins | Calculates all gain and phase margins of a single-input single-output (SISO) system model. The gain margins indicate where the frequency response crosses at 0 decibels. The phase margins indicate where the frequency response crosses -180 degrees. Use the margin function to return only the smallest gain and phase margins of a SISO model. | >[gmf, gm, pmf, pm] = <code>margins</code> (H) |

For more details about these functions, type “[help cdt](#)” to get an overview of all the functions used for Control Design and Simulation. For detailed help about one specific function, type “[help <function_name>](#)”.

Plots functions: Here are some useful functions for creating plots: [plot](#), [figure](#), [subplot](#), [grid](#), [axis](#), [title](#), [xlabel](#), [ylabel](#), [semilogx](#) – for more information about the plots function, type “[help plots](#)”.



Høgskolen i Telemark

Telemark University College

Faculty of Technology

Kjølnes Ring 56

N-3914 Porsgrunn, Norway

www.hit.no

Hans-Petter Halvorsen, M.Sc.

Telemark University College

Department of Electrical Engineering, Information Technology and Cybernetics

Phone: +47 3557 5158

E-mail: hans.p.halvorsen@hit.no

Blog: <http://home.hit.no/~hansha/>

Room: B-237a
