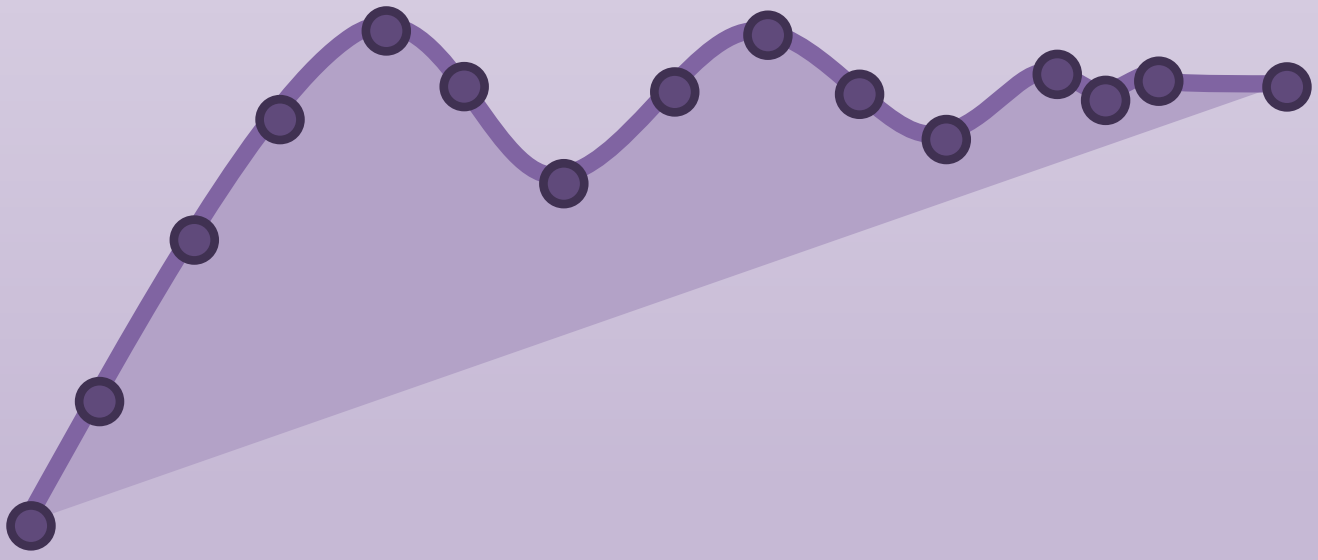
Simulink and Advanced Topics in MATLAB

Hans-Petter Halvorsen



https://www.halvorsen.blog

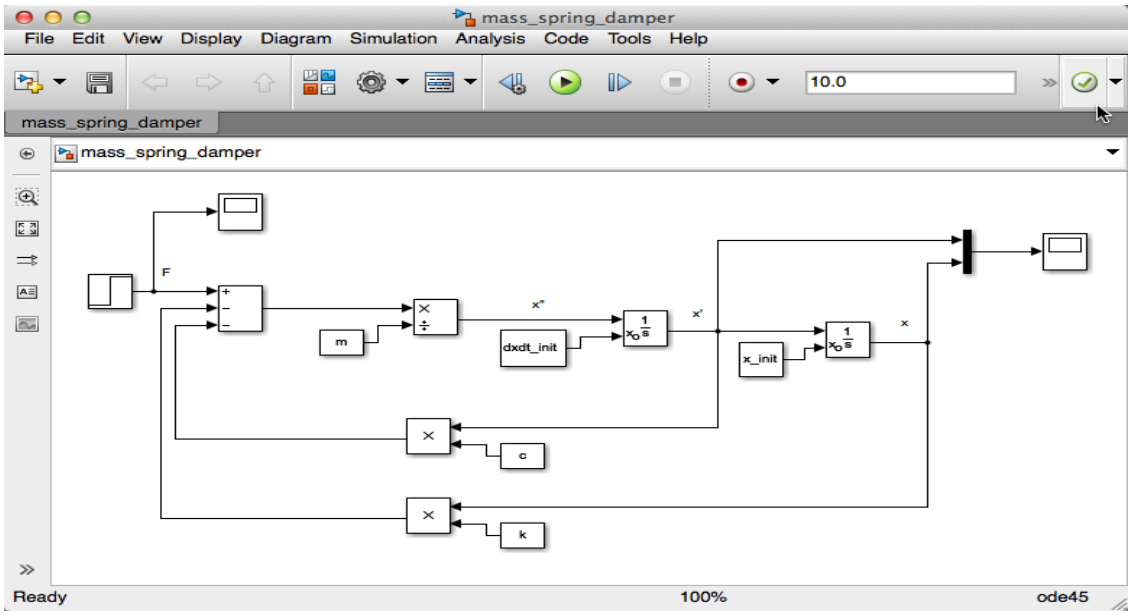
Simulink and Advanced Topics in MATLAB

University of South-Eastern Norway

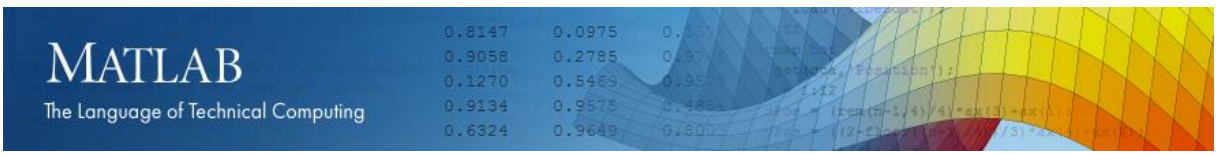
MATLAB

Simulink and Advanced Topics

Hans-Petter Halvorsen, 2022.08.17



http://www.halvorsen.blog



Preface

Copyright You cannot distribute or copy this document without permission from the author. You cannot copy or link to this document directly from other sources, web pages, etc. You should always link to the proper web page where this document is located, typically http://www.halvorsen.blog

In this MATLAB Course, you will learn basic MATLAB and how to use MATLAB in Control and Simulation applications. An introduction to Simulink and other Tools will also be given.

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, Modelling, Simulation and Control applications.

This is a self-paced course based on this document and some short videos on the way. This document contains lots of examples and self-paced tasks that the users will go through and solve on their own. The user may go through the tasks in this document in their own pace and the instructor will be available for guidance throughout the course.

The MATLAB Course consists of 3 parts:

1. Introduction to MATLAB
2. Modelling, Simulation and Control
3. Simulink and Advanced Topics

In Part 3 of the course, you will learn how to use some of the more advanced features in MATLAB. We will also take a closer look at **Simulink**, which is a Block Diagram Simulation Tool used together with MATLAB. We will also give an overview to other tools for numerical mathematics and simulation.

You must go through MATLAB Course – Part 1: Introduction to MATLAB before you start.

The course consists of lots of Tasks you should solve while reading this course manual and watching the videos referred to in the text.

Make sure to bring your **headphones** for the videos in this course. The course consists of several short videos that will give you an introduction to the different topics in the course.

Prerequisites: You should be familiar with undergraduate-level mathematics and have experience with basic computer operations.

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 is developed by The MathWorks. MATLAB is a short-term for MATrix LABoratory. MATLAB is in use world-wide by researchers and universities.

For more information, see www.mathworks.com

What is Simulink?

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

Simulink, developed by The MathWorks, is a commercial tool for modeling, simulating and analyzing dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in control theory and digital signal processing for simulation and design.

This training will give you the basic knowledge of Simulink and how you can use it together with MATLAB.

For more information about MATLAB, Simulink, etc., please visit http://www.halvorsen.blog

Online MATLAB Resources:

MATLAB:

http://www.halvorsen.blog/documents/programming/matlab/

MATLAB Basics:

http://www.halvorsen.blog/documents/programming/matlab/matlab_basics.php

Modelling, Simulation and Control with MATLAB:

http://www.halvorsen.blog/documents/programming/matlab/matlab_mic.php

MATLAB Videos:

http://www.halvorsen.blog/documents/video/matlab_basics_videos.php

MATLAB for Students:

http://www.halvorsen.blog/documents/teaching/courses/matlab.php

On these web pages you find video solutions, complete step by step solutions, downloadable MATLAB code, additional resources, etc.

Table of Contents

Preface ii

Table of Contents v

1 Introduction 1

2 Simulink 2

2.1 Start using Simulink 2

2.1.1 Block Libraries 4

2.1.2 Create a new Model 6

2.2 Wiring techniques 7

2.3 Help Window 8

2.4 Configuration 10

2.5 Examples 11

Task 1: Simulation in Simulink – Bacteria Population 20

2.6 Data-driven Modelling 21

2.6.1 Using the Command window 21

2.6.2 Using a m-file 24

2.6.3 Simulation Commands 25

Task 2: Mass-Spring-Damper System 26

Task 3: Simulink Simulation 29

3 Debugging in MATLAB 30

3.1 The Debugging Process 32

Task 4: Debugging 33

4 More about Functions 35

4.1 Getting the Input and Output Arguments 35
Task 5: Create a Function 36
Task 6: Optional Inputs: Using nargin and nargchk 37
Task 7: Optional Outputs: Using nargout and nargoutchk 38
5 More about Plots 39
5.1 LaTEX or TEX Commands 39
Task 8: LATEX Commands 40
Task 9: 3D Plot 40
6 Using Cells in the MATLAB Editor 42
Task 10: Using Cells 43
7 Importing Data 44
Task 11: Import Data 46
8 Structures and Cell Arrays 47
8.1 Structures 47
Task 12: Using Structures 48
9 Alternatives to MATLAB 49
9.1 Octave 49
9.2 Scilab and Scicos 49
9.3 LabVIEW MathScript 50
9.3.1 How do you start using MathScript? 51
9.3.2 Functions 52
9.3.3 ODE Solvers in MathScript 53
9.4 LabVIEW 54
9.4.1 The LabVIEW Environment 54
9.4.2 Front Panel 55
9.4.3 Block Diagram 58

9.4.4 LabVIEW Control Design and Simulation Module 59
9.5 Mathematics in LabVIEW 63
9.5.1 Basic Math 64
9.5.2 Linear Algebra 65
9.5.3 Curve Fitting 66
9.5.4 Interpolation 67
9.5.5 Integration and Differentiation 67
9.5.6 Statistics 68
9.5.7 Optimization 68
9.5.8 Differential Equations (ODEs) 69
9.5.9 Polynomials 69
9.6 MATLAB Integration (MATLAB Script) in LabVIEW 70
9.7 Python 71
Appendix A – MathScript Functions 73
Appendix B: Mathematics characters 75

1 Introduction

Additional Resources, Videos, etc. are available from:

http://www.halvorsen.blog/documents/programming/matlab

Part 3: Advanced Topics, Simulink and other Tools consists of the following topics:

- Introduction to Simulink
- Advanced Topics in MATLAB:
  - Debugging in MATLAB
  - More about functions
  - More about Plots
  - Using Cells in the MATLAB Editor
  - Importing Data
  - Structures and Cell Arrays
- Alternatives to MATLAB

2 Simulink

Simulink is an environment for simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing.

Simulink offers:

- A quick way of developing your model in contrast to text based-programming language such as e.g., C.
- Simulink has integrated **solvers**. In text based-programming language such as e.g., C you need to write your own solver.

Graphical Programming: In Simulink you program in a graphical way. LabVIEW is another programming language where you use graphical programming instead of text-based programming. LabVIEW is developed by National Instruments. You will use LabVIEW in a later chapter

Before you start, you should watch the following videos:

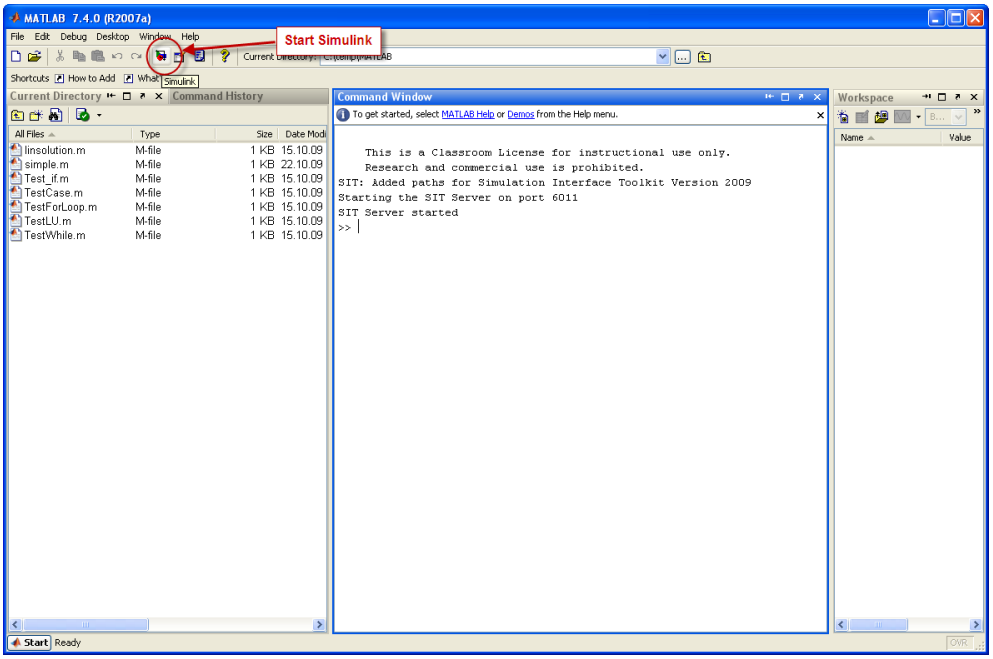
- "**Simulink Overview**"
- "**Getting Started with Simulink**"

The videos are available from:
https://www.halvorsen.blog/documents/teaching/courses/matlab/matlab3.php

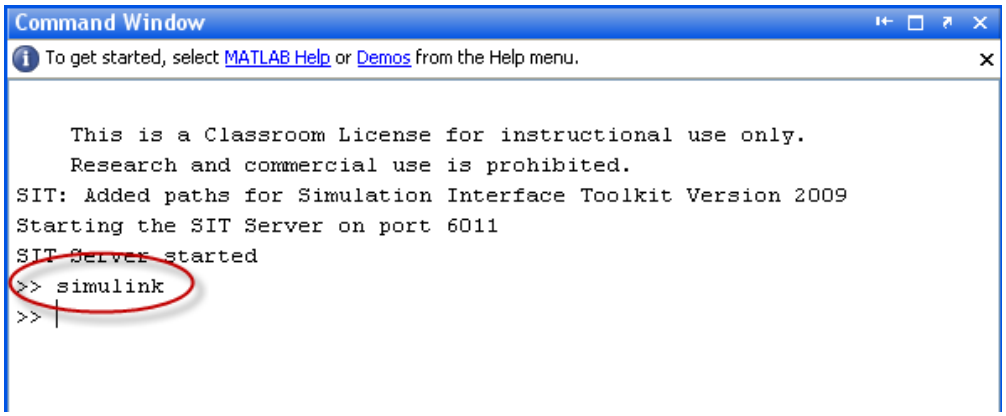
2.1 Start using Simulink

You start Simulink from the MATLAB IDE:

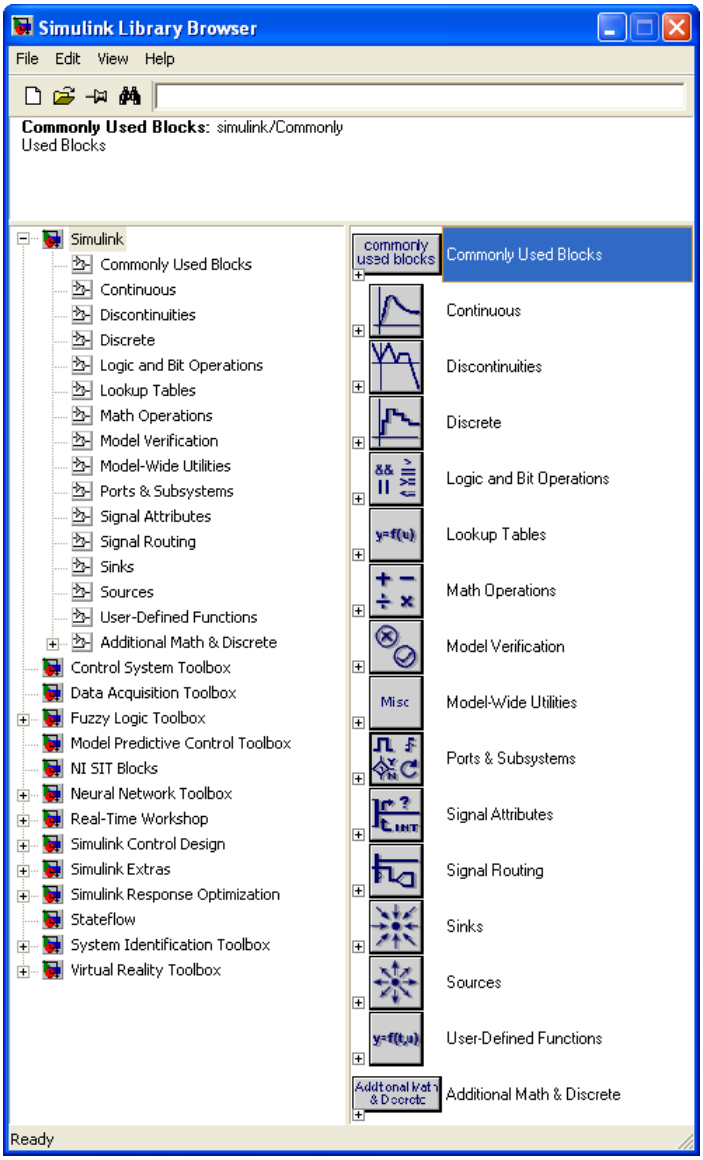
Open MATLAB and select the Simulink icon in the Toolbar:



Or type "**simulink**" in the Command window, like this:



Then the following window appears (**Simulink Library Browser**):



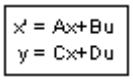
The **Simulink Library Browser** is the library where you find all the blocks you may use in Simulink. Simulink software includes an extensive library of functions commonly used in modeling a system. These include:

- Continuous and discrete dynamics blocks, such as Integration, Transfer functions, Transport Delay, etc.
- Math blocks, such as Sum, Product, Add, etc
- Sources, such as Ramp, Random Generator, Step, etc

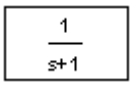
2.1.1 Block Libraries

Here are the most used "**Continuous"** Blocks:

Integrator



State-Space



Transfer Fcn

Transport Delay

Here are some commonly used "**Math Operations"** Blocks:

Add

Gain

Product

Sum

Here are some commonly used "**Signal Routing"** Blocks:

Mux

Demux

Here are some commonly used "**Sinks"** Blocks:

Scope

XY Graph

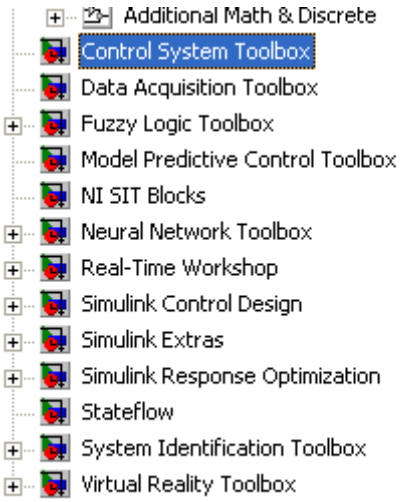
Here are some commonly used "**Sources"** Blocks:

Step

Signal Generator

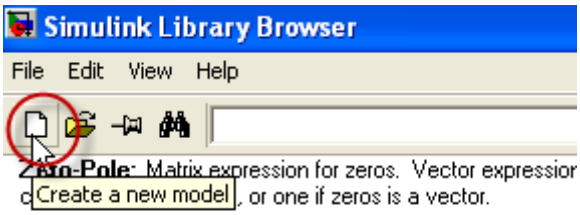
Ramp

In addition, there are lots of block in different Toolboxes:

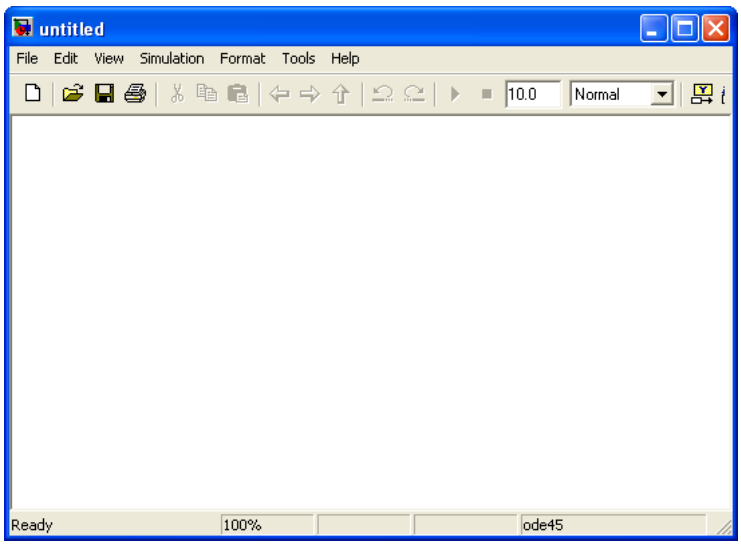


2.1.2 Create a new Model

Click the New icon on the Toolbar in order to create a new Simulink model:



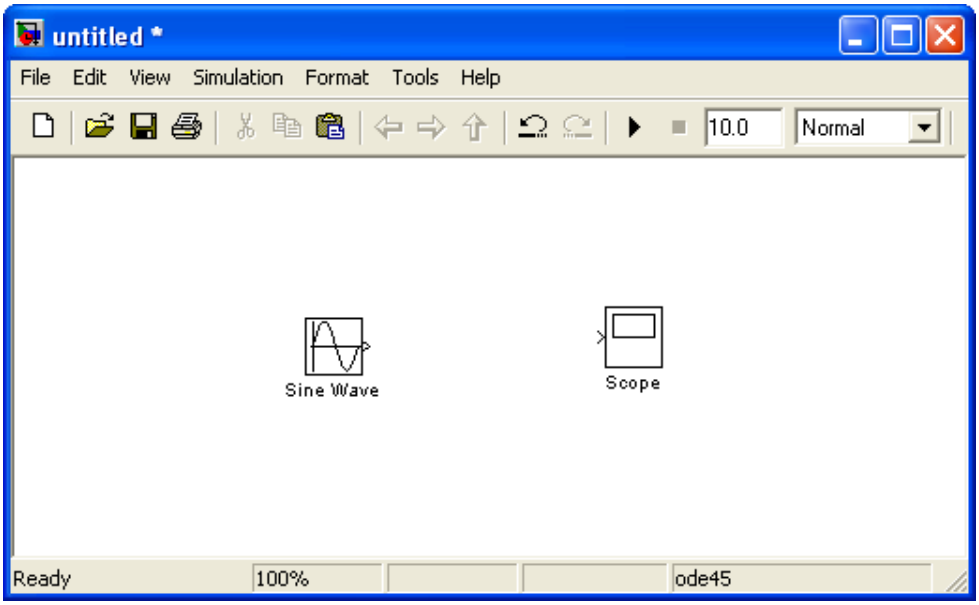
The following window appears:



You may now drag the blocks you want to use from the Simulink Library Browser to the model surface (or right-click on a block and select "Add to…").

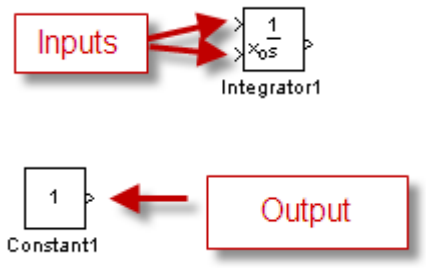
Example:

In this example we place (drag and drop) to blocks, a Sine Wave and a Scope, on the model surface:



2.2 Wiring techniques

Use the mouse to wire the **inputs** and **outputs** of the different blocks. Inputs are located on the left side of the blocks, while outputs are located on the right side of the blocks.



When holding the mouse over an input or an output the mouse changes to the following symbol.

Use the mouse, while holding the left button down, to drag wires from the input to the output.

Automatic Block Connection:

Another wiring technique is to select the source block, then hold down the **Ctrl** key while left-clicking on the destination block.

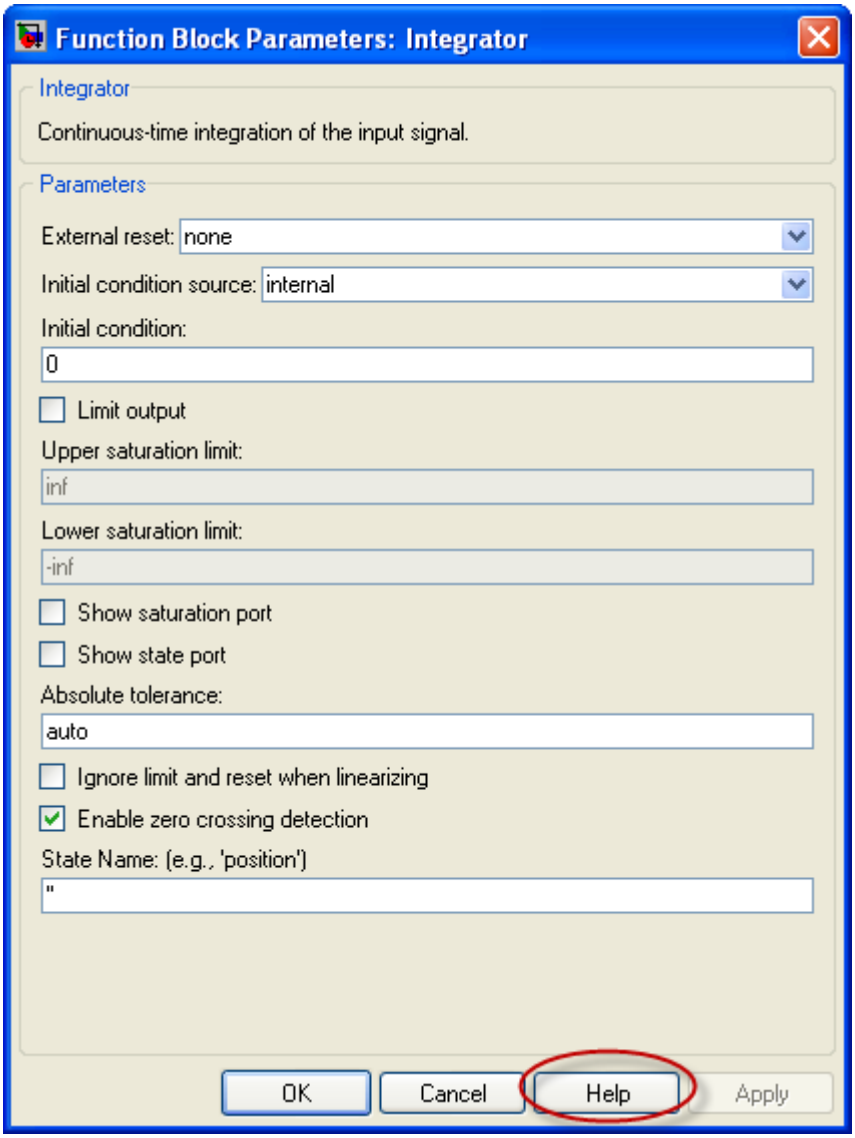
Try the different techniques on the example above.

Connection from a wire to another block

If wire a connection from a wire to another block, like the example below, you need to hold down the **Ctrl** key while left-clicking on the wire and then to the input of the desired block.

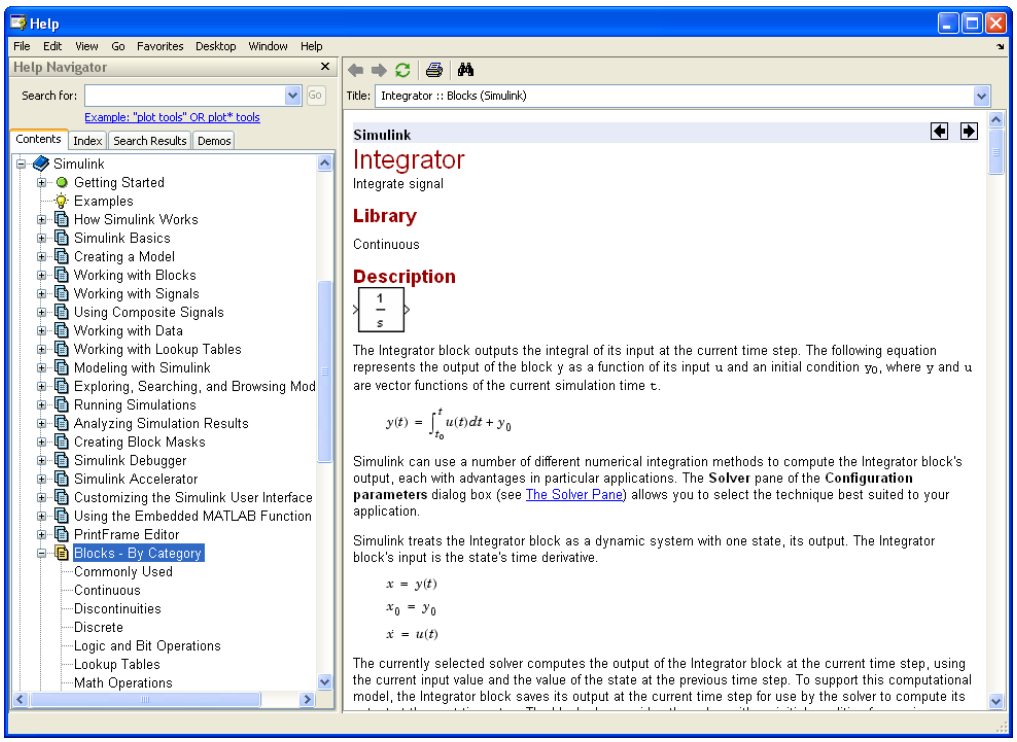
2.3 Help Window

To see detailed information about the different blocks, use the built-in Help system.



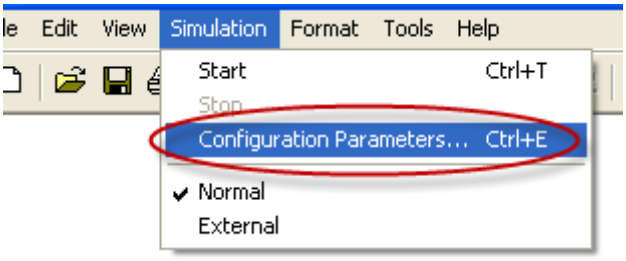
All standard blocks in Simulink have detailed Help. Click the Help button in the Block Parameter window for the specific block to get detailed help for that block.

The Help Window then appears with detailed information about the selected block:

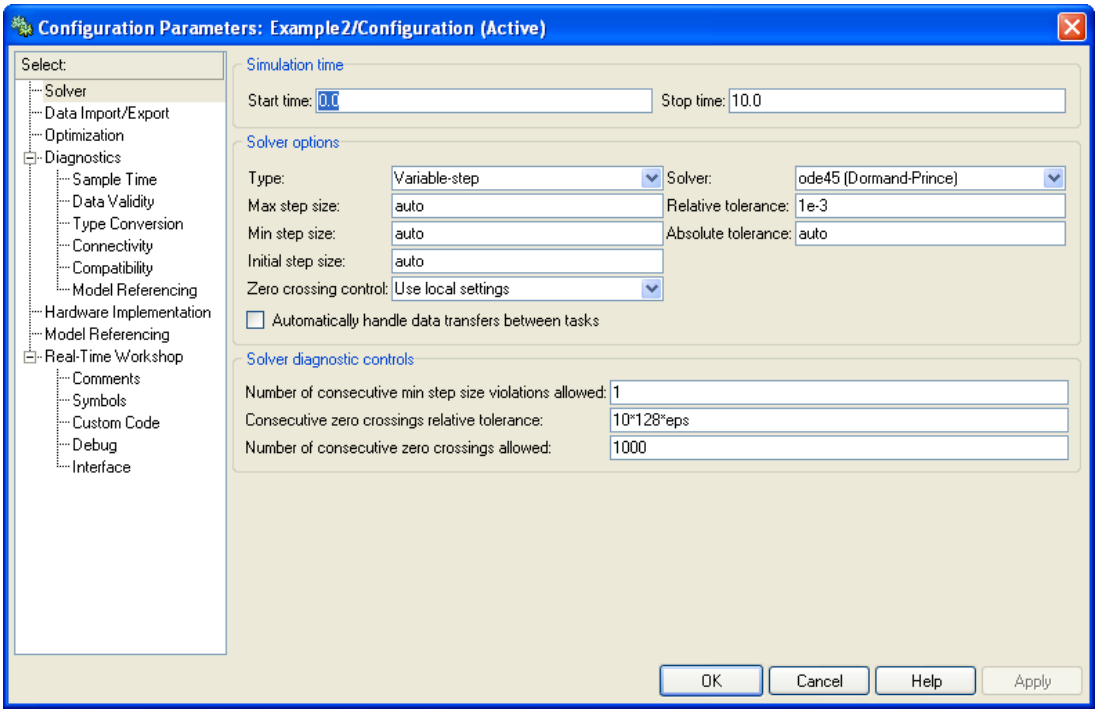


2.4 Configuration

There are lots of parameters you may want to configure regarding your simulation. Select "Configuration Parameters…" in the Simulation menu.



The following window appears:



Here you set important parameters such as:

- Start and Stop time for the simulation
- What kind of Solver to be used (ode45, ode23 etc.)
- Fixed-step/Variable-step

Note! Each of the controls on the Configuration Parameters dialog box corresponds to a configuration parameter that you can set via the "**sim**" and "**simset**" commands. You will learn more about these commands later.

Solvers are numerical integration algorithms that compute the system dynamics over time using information contained in the model. Simulink provides solvers to support the simulation of a broad range of systems, including continuous-time (analog), discrete-time (digital), hybrid (mixed-signal), and multirate systems of any size.

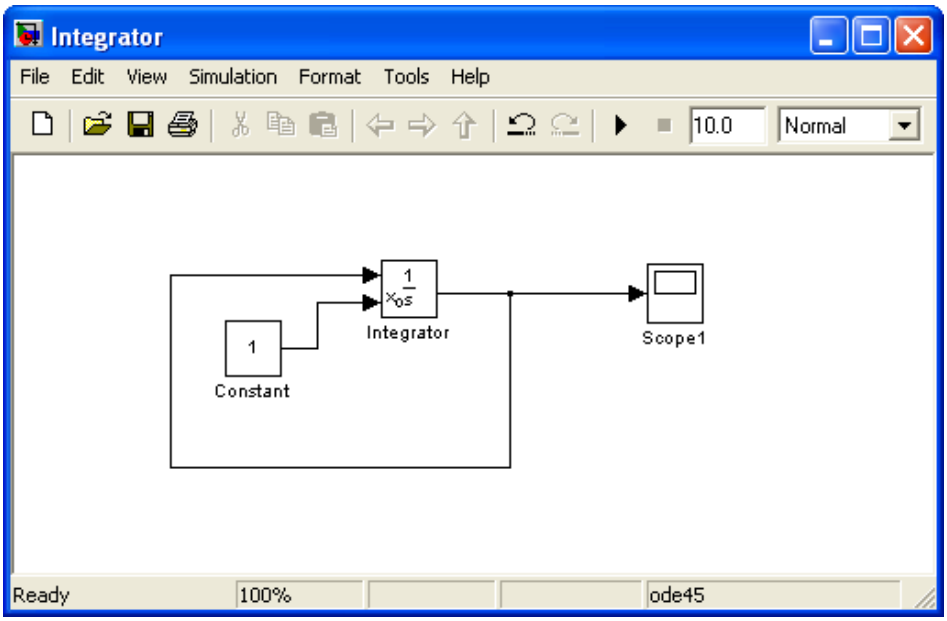
2.5 Examples

Below we will go through some examples in order to illustrate how to create block diagrams and related functionality.

Example:

Integrator with initial value

Create the following model (an integrator) and run the simulation:



Step1: Place the blocks on the model surface

This example uses the following blocks:

Integrator

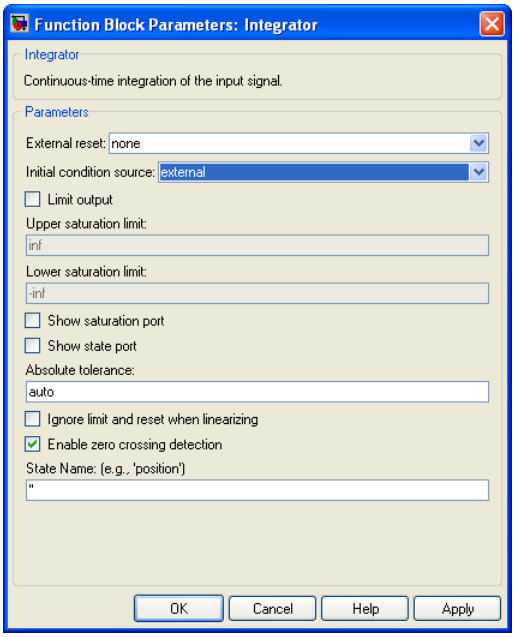
Constant

Scope

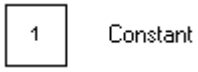
Step 2: Configuration

Integrator

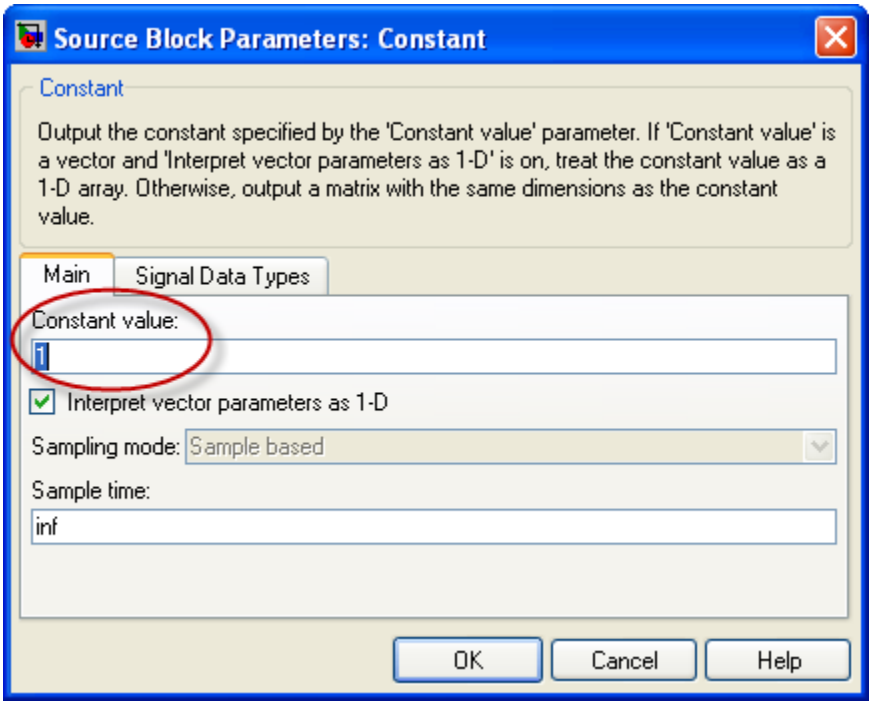
Double-click on the Integrator block. The Parameter window for the Integrator block appears:



Select "**Initial condition source=external**". The Integrator block now looks like this:



Double-click on the Constant block. The Parameter window for the Constant block appears:



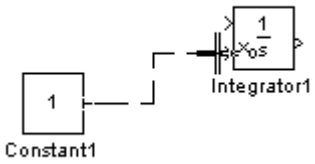
In the Constant value field we type in the initial value for the integrator, e.g., type the value 1.

Step 3: Wiring

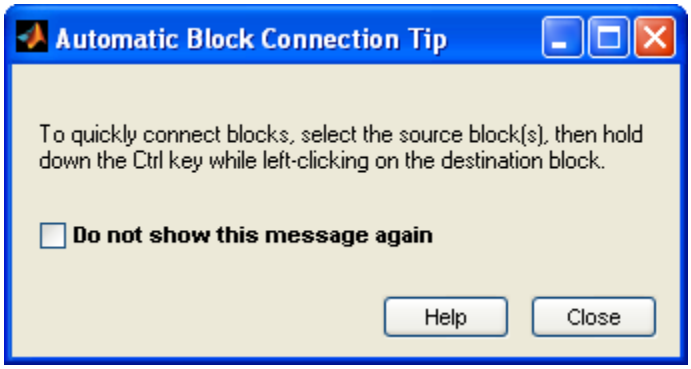
Use the mouse to wire the inputs and outputs of the different blocks.

When holding the mouse over an input or an output the mouse change to the following symbol.

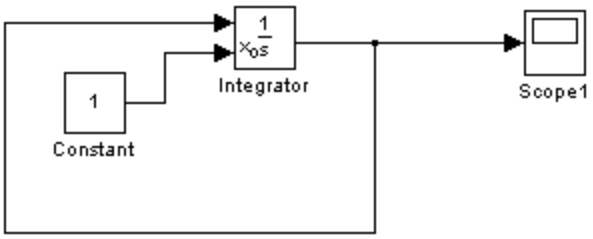
Draw a wire between the output on the Constant block to the lower input in the Integrator block, like this:



You could also do like this:



Wire the rest of the blocks together and you will get the following diagram:

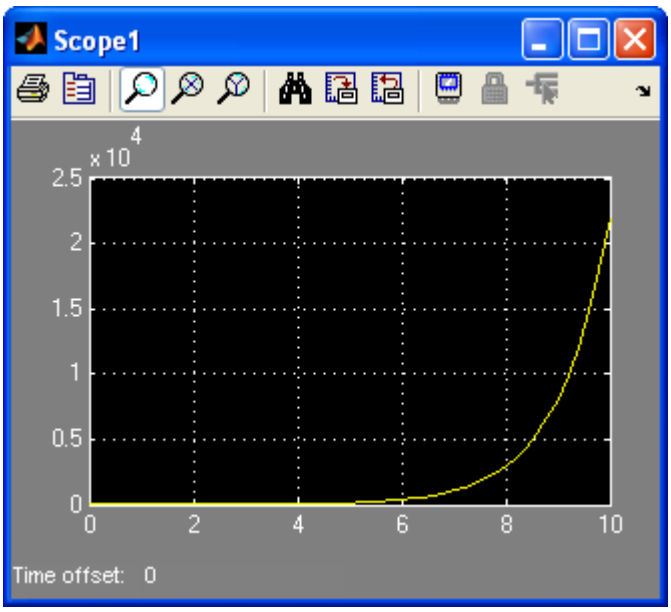


Step 4: Simulation

Start the simulation by clicking the "Start Simulation" icon in the Toolbar:

Step 5: The Results

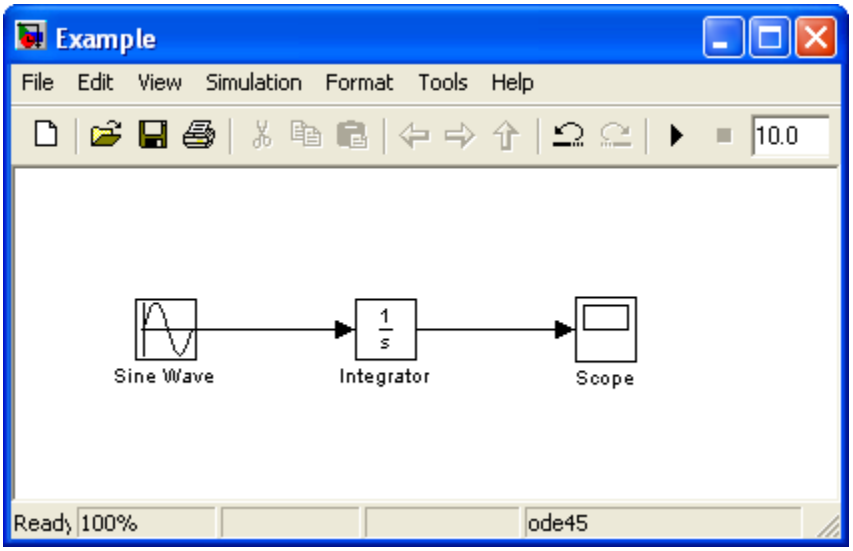
Double-click in the Scope block in order to see the simulated result:



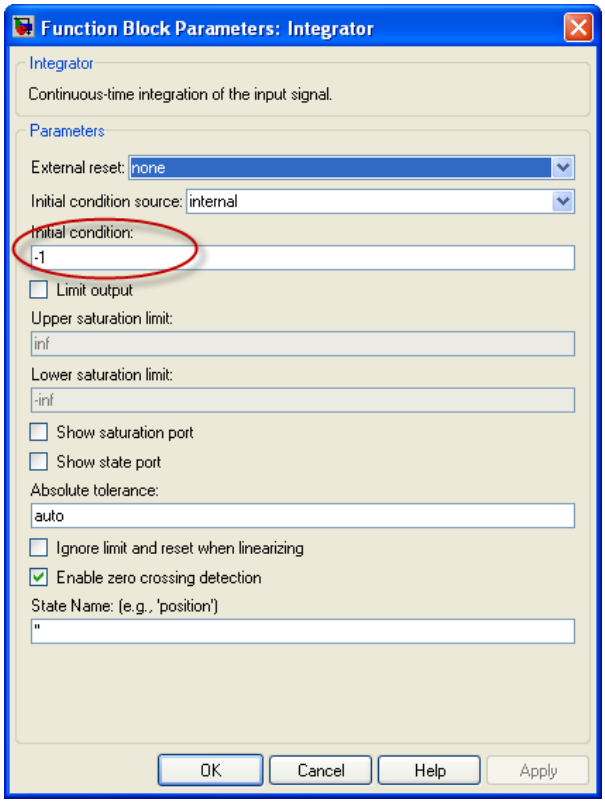
Example:

Sine Wave

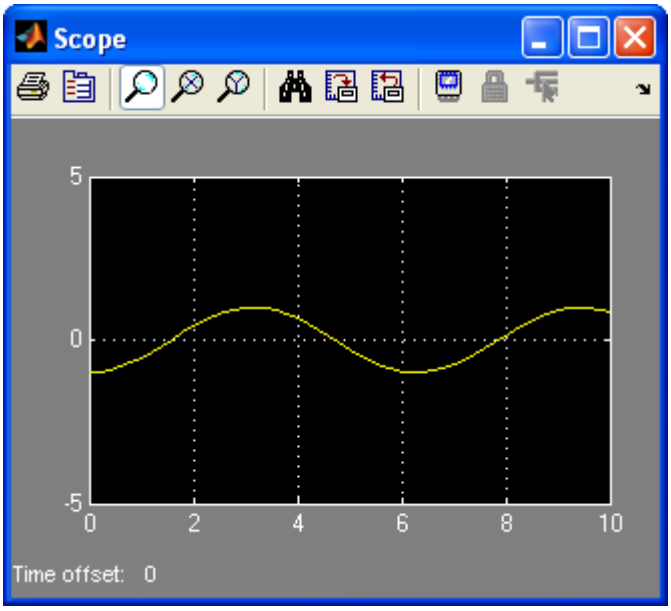
Create the block diagram as shown below:



Set the following parameter for the Integrator block:



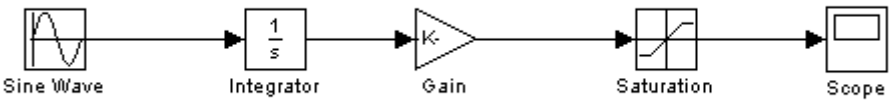
The result should be like this:



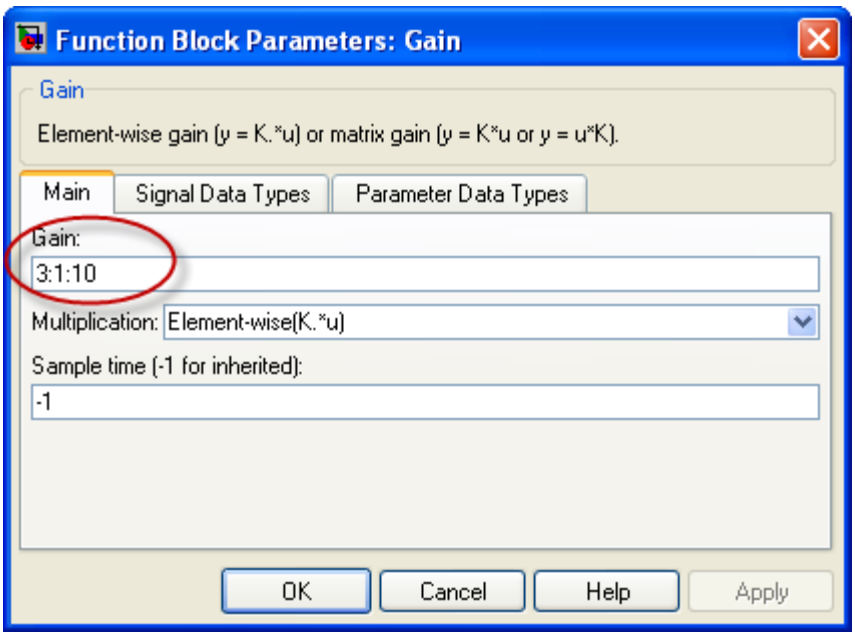
Example:

Using vectors

Create the following block diagram:

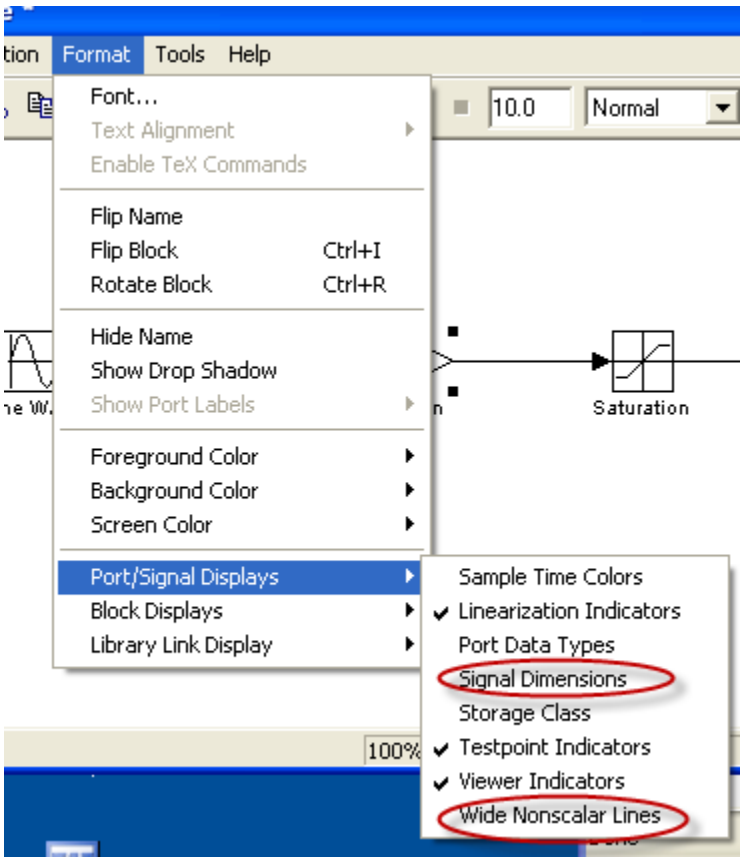


For the Gain block, type the following parameters:

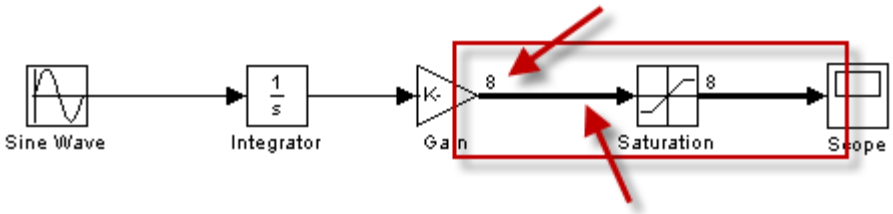


As you see, we can use standard MATLAB syntax to create a vector.

If you want to see the signal dimensions, select "Signal Dimensions" and "Wide Nonscalar Lines" as shown here:

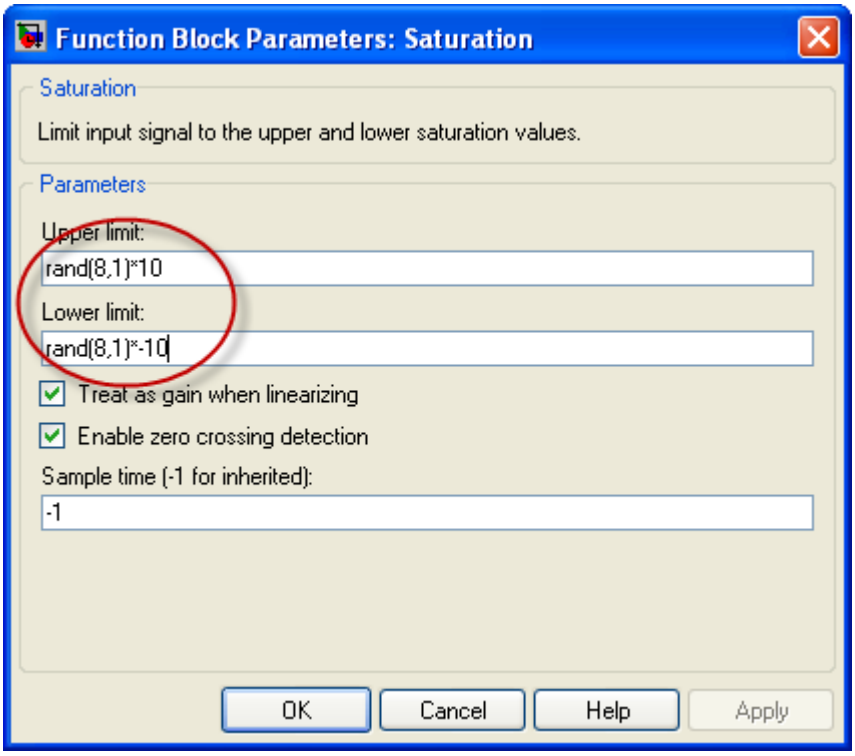


The block diagram should now look like this:



The thick lines indicate vectors, while the number (8) is the size of the vector.

Let’s change the Saturation block:



As you see you may use standard MATLAB functions and syntax.

Run the simulation and see the results in the Scope block.

Task 1: Simulation in Simulink – Bacteria Population

In this task we will simulate a simple model of a bacteria population in a jar (known from a previous task).

The model is as follows:

$$birth\ rate = bx$$

$$death\ rate = px^2$$

Then the total rate of change of bacteria population is:

$$\dot{x} = bx - px^2$$

Set ***b=1***/hour and ***p=0.5*** bacteria-hour

We will simulate the number of bacteria in the jar after **1 hour**, assuming that initially there are **100 bacteria** present.

Procedure:

1. Create the **block diagram** for the system using **"pen & paper"**

2. Start **Simulink** and create a New Model
3. Drag in the necessary blocks from the **Simulink Library Browser**
4. **Configure** the different blocks (double-click/right-click depending on what you need). Some blocks need to be "flipped" (Right-click → Format → Flip Block), while in other blocks you need to set a value (double-click)
5. **Draw lines** between the different blocks using the mouse
6. Set **Simulation Settings** (Simulation → Configuration Parameters). The simulation Time (Stop Time) should be set to **1** (hour)
7. Use a **Scope** to see the Simulated Result

You will need the following blocks:

- **Integrator** block To solve the differential equation. Note! Initial value $\mathbf{x_0=100}$
- Two **Gain** blocks For p (=0.5) and b (=1)
- **Product** block To compute $\mathbf{x^2}$
- **Sum** block Note! One plus (+) must be changed to minus (-)
- **Scope** block To show the simulated result. Note! Set to Autoscale

[End of Task]

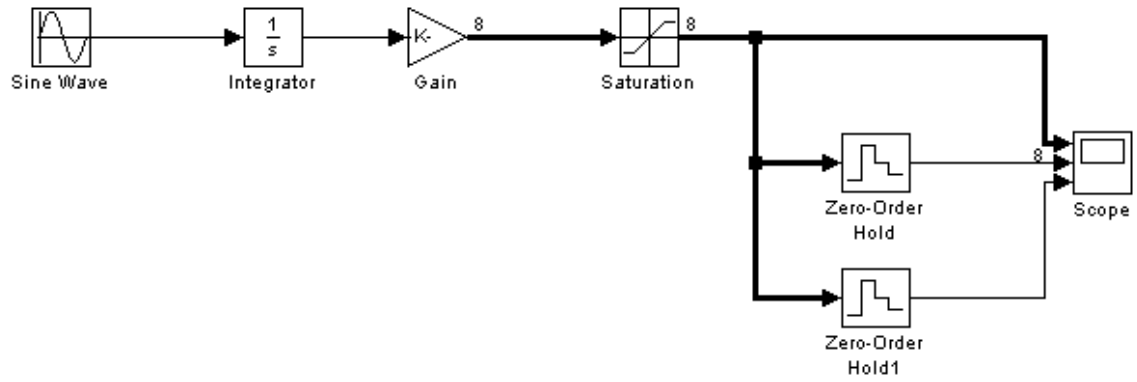
2.6 Data-driven Modelling

You may use Simulink together with MATLAB to specify data and parameters to your Simulink model. You may specify commands in the MATLAB Command Window or as commands in an m-file. This is called data-driven modeling.

2.6.1 Using the Command window

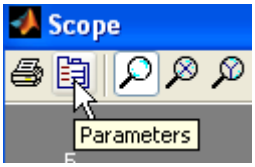
Example:

Given the following system:

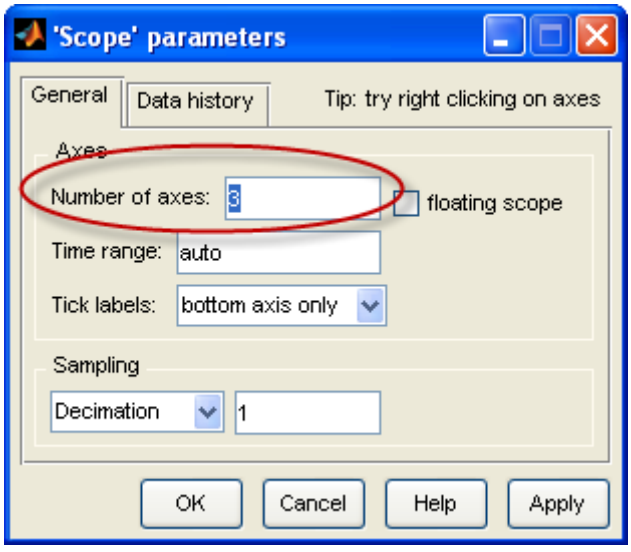


Note! In order to get 3 inputs on the Scope block:

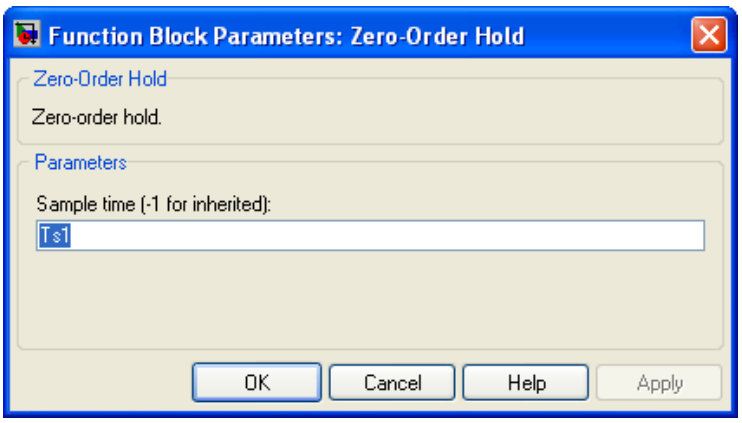
Double-click on the Scope and select the Parameters icon in the Toolbar:

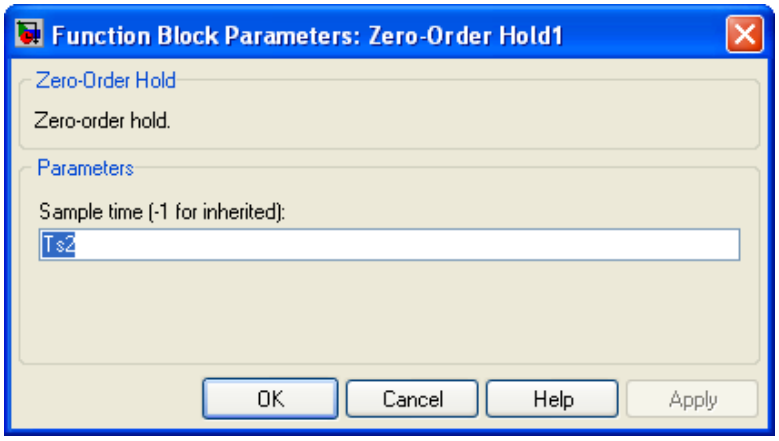


Then select Number of Axes=3:

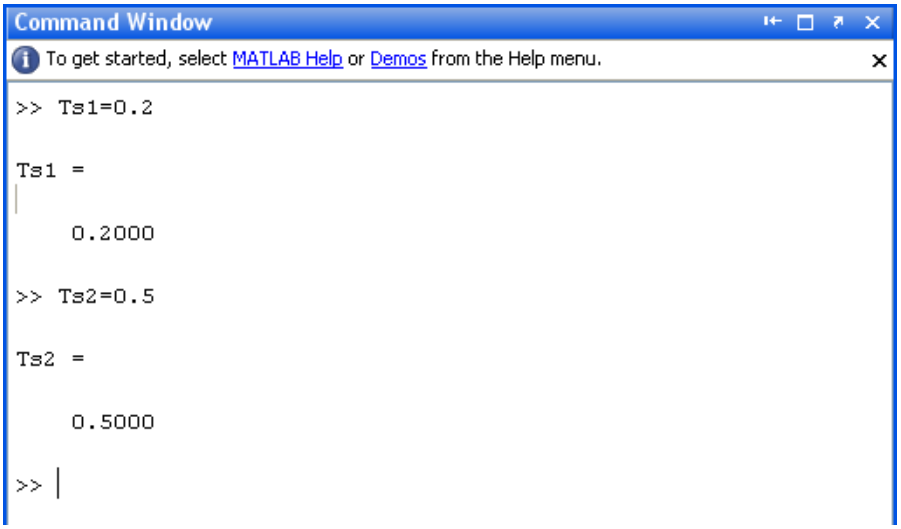


Configure the zero-order hold blocks like this:

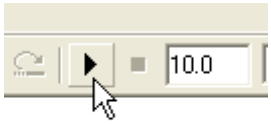




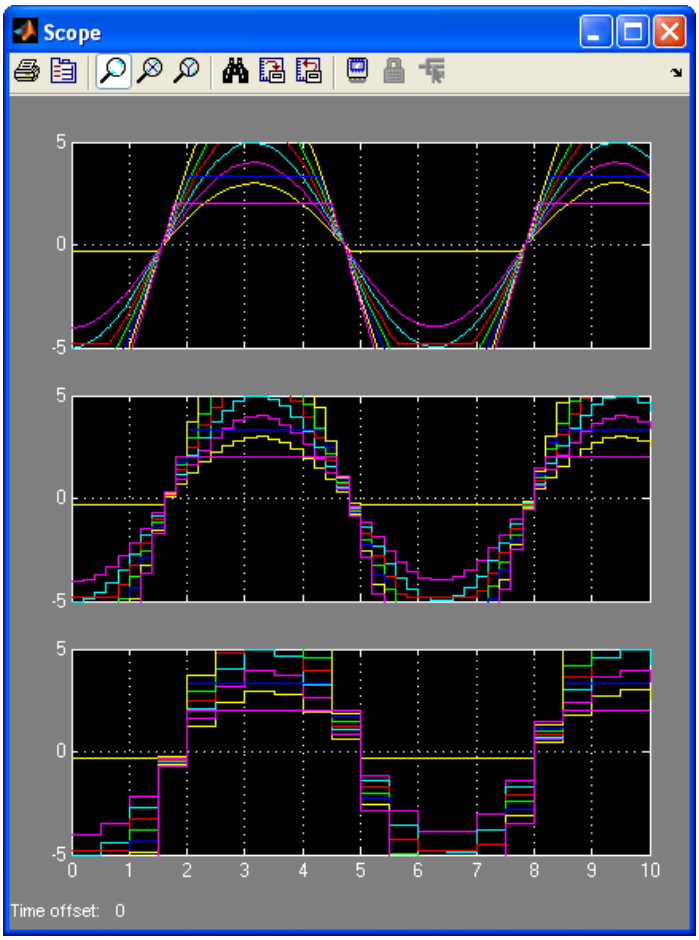
Write the following in the **Command window** in MATLAB:



Run the Simulink model from the Simulink:



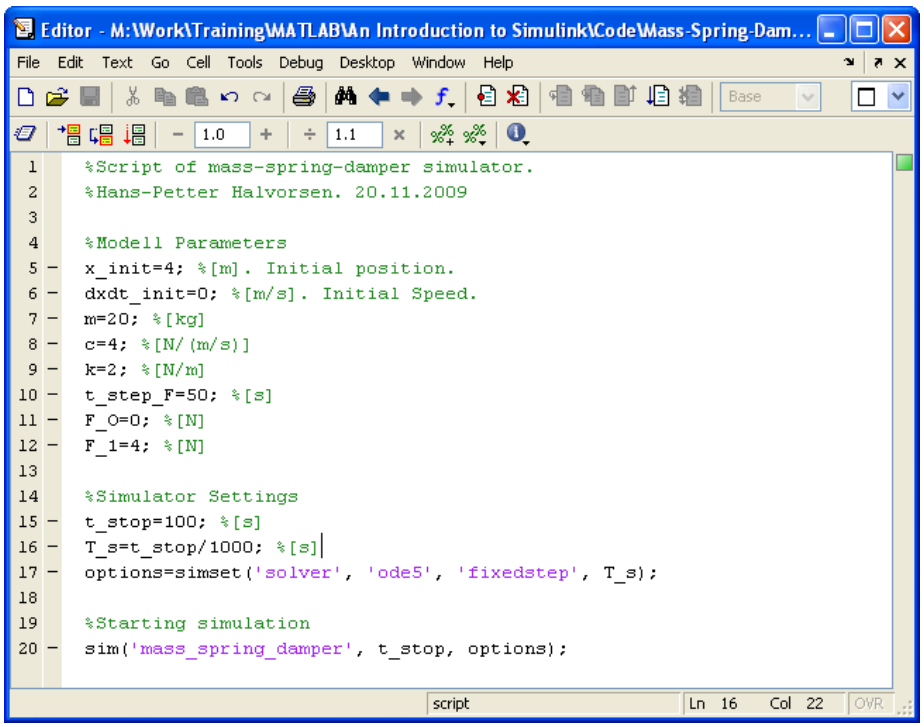
We then get the following results:



2.6.2 Using a m-file

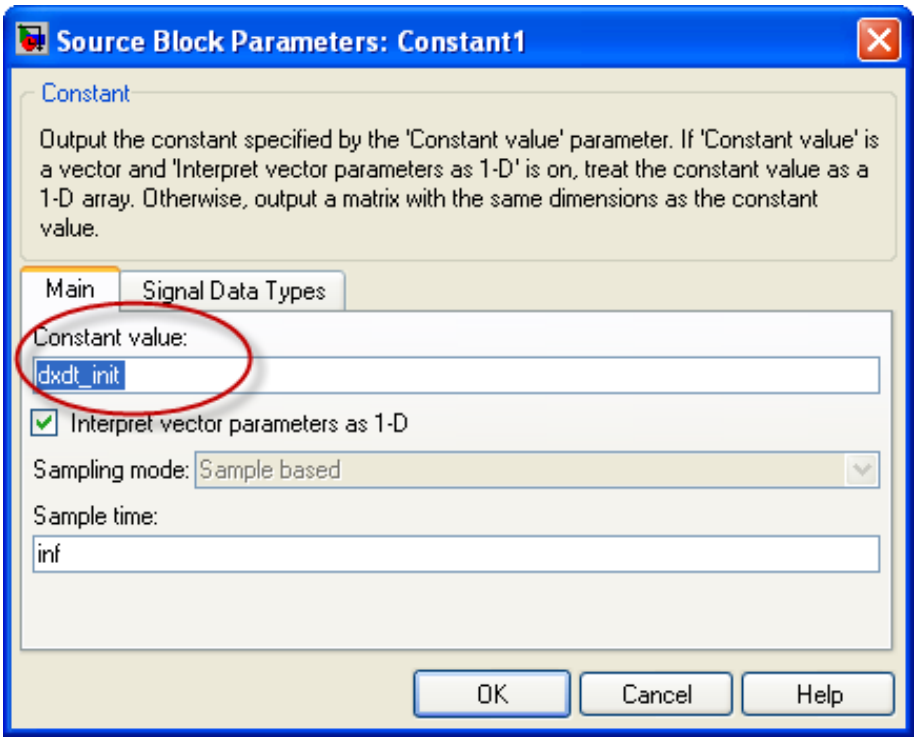
It is good practice to build your model in Simulink and configure and run the simulation from a MATLAB m-file.

A Typical m-file could look like this:



You use the **simset** command to configure your simulation parameters and the **sim** command to run the simulation.

The variables you refer to in the m-file is set in the Constant value field in the Parameter window for each block.



2.6.3 Simulation Commands

The most used command is:

- simset
- sim

Use these commands if you configure and run your Simulink model from a m-file.

Example:

```
%Simulator Settings
t_stop=100; %[s]
T_s=t_stop/1000; %[s]
options=simset('solver', 'ode5', 'fixedstep', T_s);

%Starting simulation
sim('mass_spring_damper', t_stop, options);
```

[End of Example]

Task 2: Mass-Spring-Damper System

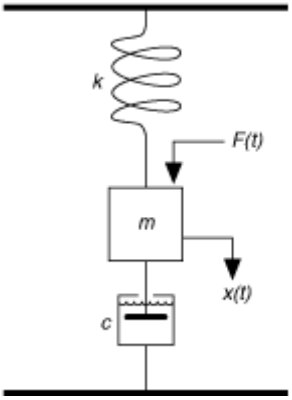
In this example we will create a mass-spring-damper model in Simulink and configure and run the simulation from a MATLAB m-file.

In this exercise you will construct a simulation diagram that represents the behavior of a dynamic system. You will simulate a spring-mass damper system.

$$F(t) - c\dot{x}(t) - kx(t) = m\ddot{x}(t)$$

where t is the simulation time, F(t) is an external force applied to the system, c is the damping constant of the spring, k is the stiffness of the spring, m is a mass, and x(t) is the position of the mass. $\dot{x}$ is the first derivative of the position, which equals the velocity of the mass. $\ddot{x}$ is the second derivative of the position, which equals the acceleration of the mass.

The following figure shows this dynamic system.



The goal is to view the position x(t) of the mass m with respect to time t. You can calculate the position by integrating the velocity of the mass. You can calculate the velocity by integrating the acceleration of the mass. If you know the force and mass, you can calculate this acceleration by using Newton's Second Law of Motion, given by the following equation:

Force = Mass × Acceleration

Therefore,

Acceleration = Force / Mass

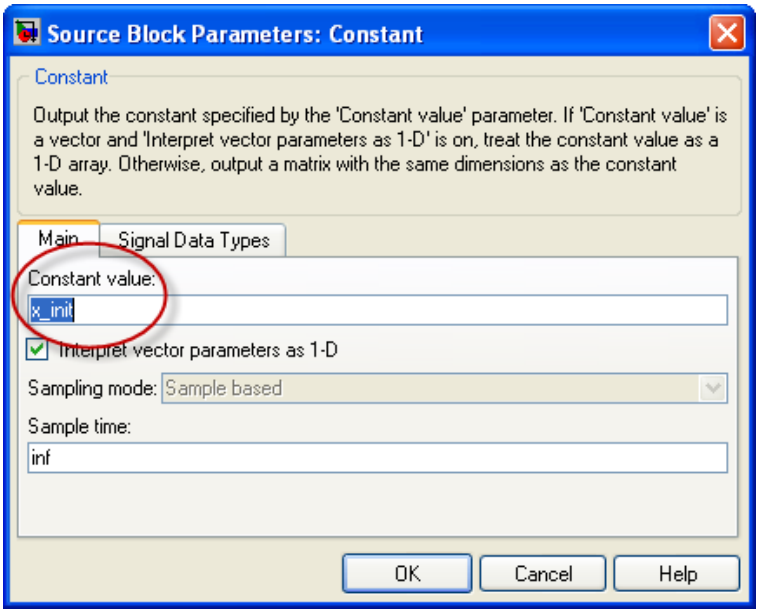
Substituting terms from the differential equation above yields the following equation:

$$\ddot{x} = \frac{1}{m}(F - c\dot{x} - kx)$$

You will construct a simulation diagram that iterates the following steps over a period of time.

→ Create the block diagram for the mass-spring-damper model above.

Instead of hard-coding the model parameters in the blocks you should refer to them as variables set in an m-file.



These variables should be configured:

- x_init
- dxdt_init
- m=
- c=
- k
- t_step_F
- F_O
- F_1

m-File

The following variables should then be set in the m-file:

```
x_init=4; %[m]. Initial position.
dxdt_init=0; %[m/s]. Initial Speed.
m=20; %[kg]
c=4; %[N/(m/s)]
k=2; %[N/m]
t_step_F=50; %[s]
F_O=0; %[N]
F_1=4; %[N]
```

→ Create the model of the system in Simulink, and then create a m-file where you specify model and simulation parameters. Then use the **sim** function in order to run the simulation within the m-file.

[End of Task]

Task 3: Simulink Simulation

Given the autonomous system:

$$\dot{x} = ax$$

where $a = -\frac{1}{T}$,where T is the time constant

The solution for the differential equation is found to be:

$$x(t) = e^{at}x_0$$

Set $T = 5$ and the initial condition $x(0) = 1$.

Simulate the system in Simulink where we plot the solution $x(t)$ in the time interval $0 \leq t \leq 25$

[End of Task]

3 Debugging in MATLAB

Debugging is about different techniques for finding bugs (errors that make your code not work as expected) in your code.

In all but the simplest programs, you are likely to encounter some type of unexpected behavior when you run the program for the first time. Program defects can show up in the form of warning or error messages displayed in the command window, programs that hang (never terminate), inaccurate results, or some number of other symptoms.

It is difficult to write code without errors (bugs), but MATLAB have powerful Debugging functionality, similar to other tools like, e.g., Visual Studio.

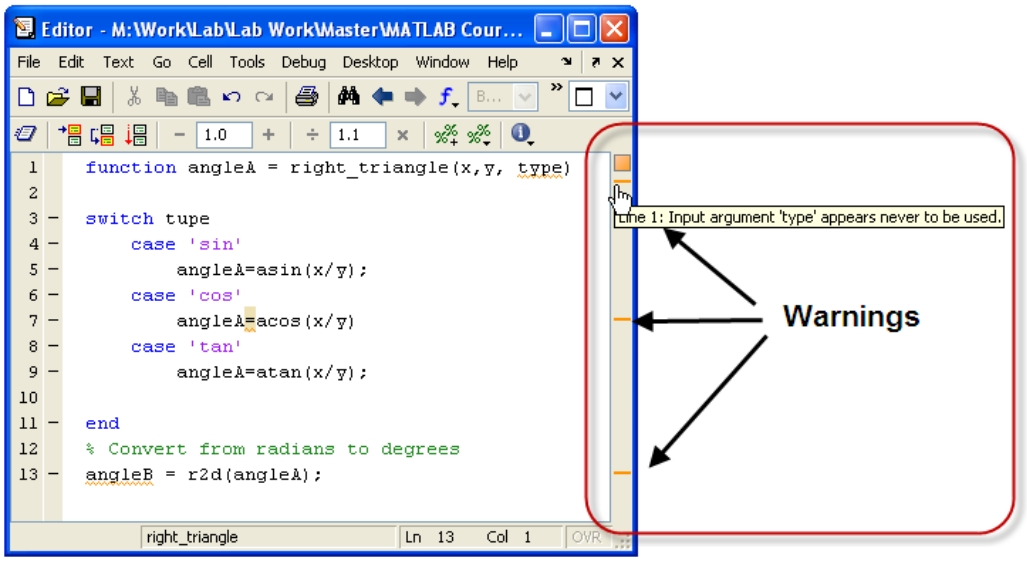
Why we call it debugging? They found a bug (actually a moth) inside a computer in 1947 that made the program not behaving as expected. This was the "first" real bug which was debugged.

Step 1: Removing Warnings and Errors notified by MATLAB

The first step in order to avoid errors is to remove all warnings and errors notifies by MATLAB in the Editor. On the right side of the Editor there will be shown symbols to illustrate that MATLAB have found potential errors in your code.

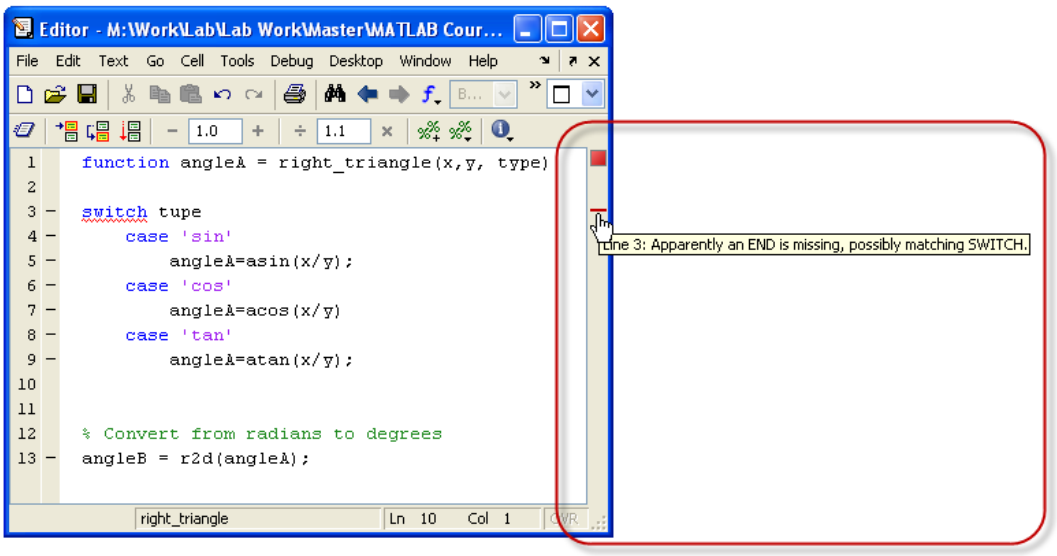
Warnings - Click the symbols to get more information about a specific warning

Example:



Errors - Click the symbols to get more information about a specific error

Example:

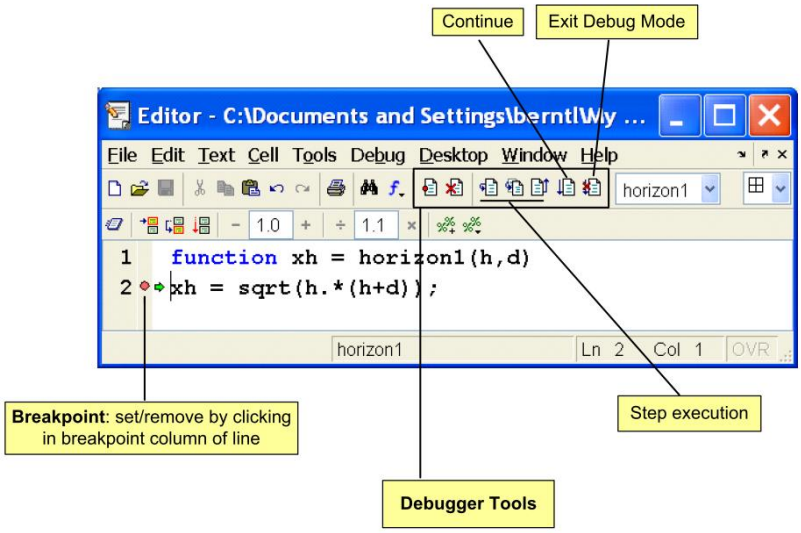


→ Take necessary actions in order to remove these Errors and Warnings!

Step 2: Using Debugging Tools and Techniques in the MATLAB Editor

In addition MATLAB have more sophisticated debugging tools we will learn more about below. These are tools you use when your program is running.

Below we see the basic debugging functionality in MATLAB:

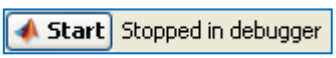


The MATLAB Debugger enables you to examine the inner workings of your program while you run it. You can stop the execution of your program at any point and then continue from that point, stepping through the code line by line and examining the results of each operation performed. You have the choice of operating the debugger from the Editor window that displays your program, from the MATLAB command line, or both.

3.1 The Debugging Process

You can step through the program right from the start if you want. For longer programs, you will probably save time by stopping the program somewhere in the middle and stepping through from there. You can do this by approximating where the program code breaks and setting a stopping point (or ***breakpoint***) at that line. Once a breakpoint has been set, start your program. The MATLAB Editor/Debugger window will show a green arrow pointing to the next line to execute. From this point, you can examine any values passed into the program, or the results of each operation performed. You can step through the program line by line to see which path is taken and why. You can step into any functions that your program calls, or choose to step over them and just see the end results. You can also modify the values assigned to a variable and see how that affects the outcome.

When the program is in debug-mode, the command prompt is changed to "K>>" and the following message appears in the status bar:



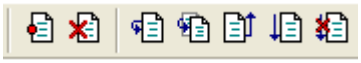
Your code could look something like this:

```
1 -  clear, clc
2 -  m=5;
3 -  for n = 1:m
4 ●➡     r(n) = rank(magic(n))
5 -  end
6
```

● A red circle indicates that you have set a breakpoint, which means your program will stop at this place in your code and wait for further instructions from you.

The green arrow indicates at what line your program is at the moment.

Now you can use the "debugging toolbar" to step through your code:



The "debugging toolbar" contains the following buttons:

Set/Clear Breakpoint

Clear all Breakpoints

Step through the program, line by line

Step in (to a function, etc.)

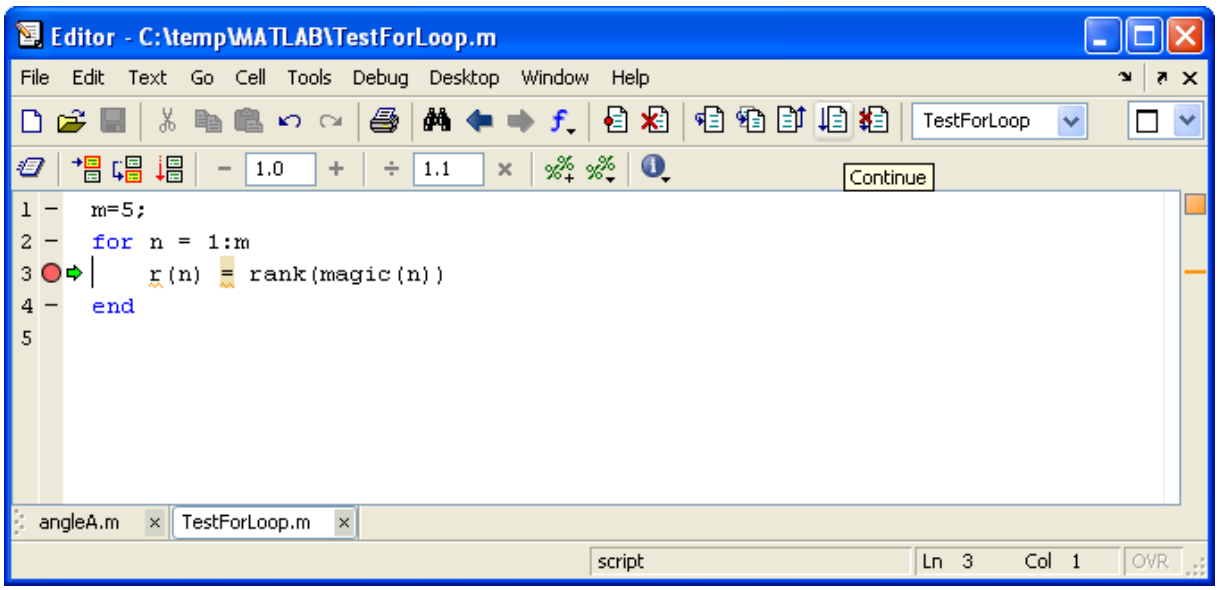
Step out (of a function, etc.)

Continue

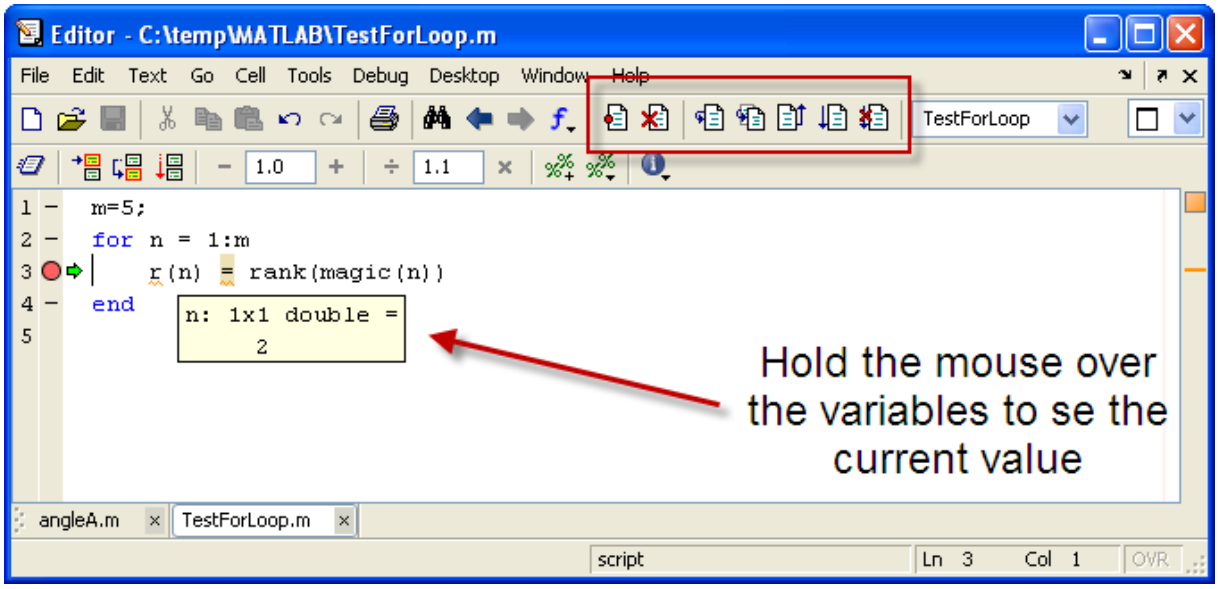
Exit Debug mode

Task 4: Debugging

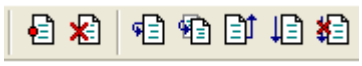
Create a similar program like this:



Set a Breakpoint inside the loop and use the Debugging functionality to step through the program and watch the result in each iteration.



Test all the different buttons in the "debugging toolbar":



In addition you should open some of your previous programs you have made, and try the debugging tools on them.

[End of Task]

4 More about Functions

4.1 Getting the Input and Output Arguments

A function may have several inputs and several outputs. Use **nargin** and **nargout** to determine the number of input and output arguments in a particular function call. Use **nargchk** and **nargoutchk** to verify that your function is called with the required number of input and output arguments.

Example:

We create the following function:

```
function [x,y] = myfunc(a,b,c,d)
disp(nargchk(2,4,nargin)) % Allow 2 to 4 inputs
disp(nargoutchk(0,2,nargout)) % Allow 0 to 2 outputs

x = a + b;
y = a * b;

if nargin == 3
    x = a + b + c;
    y = a * b * c;
end

if nargin == 4
    x = a + b + c + d;
    y = a * b * c * d;
end
```

We test the function in the Command window with different inputs and outputs:

```
>> [x, y] = myfunc(1,2)
x =
     3
y =
     2
```

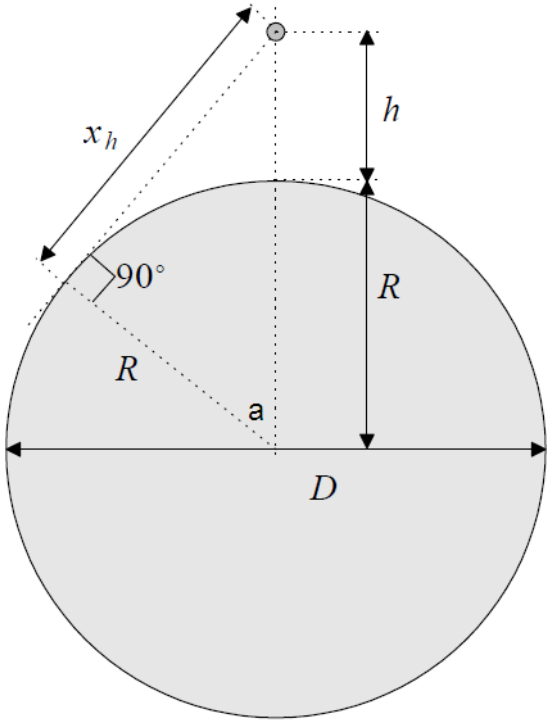
```
>> [x, y] = myfunc(1,2,3)
x =
     6
y =
     6
>> [x, y] = myfunc(1,2,3,4)
x =
    10
y =
    24
>> [x] = myfunc(1,2)
x =
     3
>> [x, y, z] = myfunc(1,2)
??? Error using ==> myfunc
Too many output arguments.

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

Note! In newer versions of MATLAB, the **error** function is recommended instead of the **disp** function, but both should work.

[End of Example]

Task 5: Create a Function

You may have experienced standing on top of a hill or mountain, and it feels you can see everything. How far can you really see? It depends on the height of the mountain and the radius of the earth, as shown in the sketch below.



In this task we will create a function that finds the distance to the horizon x_h.

You may use the Pythagorean law to find x_h:

$$R^2 = x_h^2 = (R+h)^2 \Leftrightarrow x_h = \sqrt{h(2R+h)}$$

D is the diameter of the earth, R is the radius of the earth, h is your height above the earth, where you are standing on a mountain. The radius on the earth is $R = 6378km$.

→ **Create a function that finds xh from input parameter h**,

```
>>xh=horizon(h)
```

How far can you see if you are on top of the Mount Everest?

Make sure the function may handle vector inputs and create a help text for the function that describes what the function is doing.

→ Create a script where you use the function to plot h vs. xh where h is a vector from 1 to 8000 meters. Create labels, title and a legend in the plot.

[End of Task]

Task 6: Optional Inputs: Using nargin and nargchk

The distance to the horizon is quite different on the moon than on the earth because the radius is different for each.

→ **Extend your function so that R could be an optional input to the function**, e.g.:

```
>>xh=horizon(h,R)
```

If `xh=horizon(h)` is used, R is assumed to be R=6378km (the earth).

→ Use **nargin** to solve the problem. Use also **nargchk** to validate the number of inputs.

How far could you see if the moon had a mountain similar to Mount Everest? The radius on the moon is $R = 1737km$.

[End of Task]

Task 7: Optional Outputs: Using nargout and nargoutchk

Let say we also may want to find the angle (a) between radius to the horizon and the observer (you are standing on top of the mountain). See the illustration above.

→ **Extend your function so that the angle a could be an optional output from the function**, e.g.:

```
>>[xh,a]=horizon(h,R)
```

If `xh=horizon(h,R)` is used, a should be ignored (only xh is calculated).

→ Use **nargout** to solve the problem. Use also **nargoutchk** to validate the number of outputs.

The angle a is given by:

$$\tan a = \frac{x_h}{R} \Leftrightarrow a = \operatorname{atan}\left(\frac{x_h}{R}\right)$$

Note! You have to convert from radians to degrees ($2\pi = 360^o$). Use your function **r2d** which you created in a previous task.

[End of Task]

5 More about Plots

MATLAB have advance Plot functionality. We have already used the plot functionality in MATLAB in a dozen of examples. In this chapter we will learn more about the advanced plotting functionality that MATLAB offers.

5.1 LaTEX or TEX Commands

When using labels, legends, and titles in a plot you sometimes want use more advanced labels, titles, legend such as:

"Solution of $\int_1^5 \sin(x)\,dx$"

This is done by using LaTeX or TeX commands.

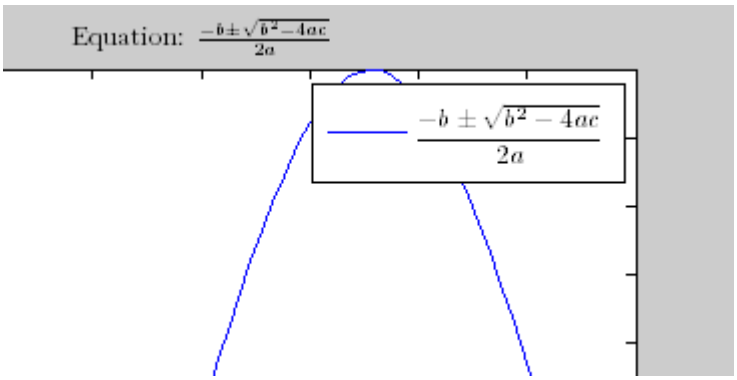
In LaTeX typesetting, mathematical expressions are bracketed by the $$ symbol. Using $ bracketing, indicates in-line math.

Example:

The MATLAB code:

```
legend({'$$\frac{-b  \pm  \sqrt{b^{2}-4ac}}{2a}$$'},
'Interpreter', 'LaTeX')

title({'Equation: $\frac{-b  \pm  \sqrt{b^{2}-4ac}}{2a}$'},
'Interpreter', 'LaTeX')
```

gives the following plot:

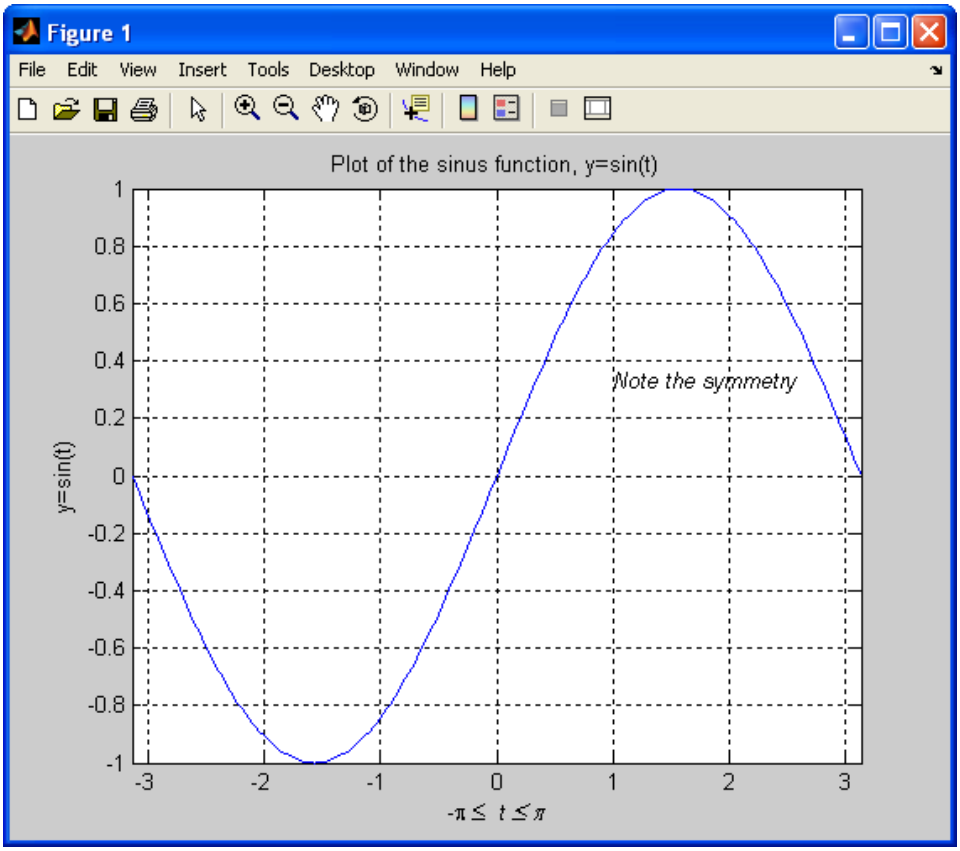


[End of Example]

See Appendix B: Mathematics characters.

Task 8: LATEX Commands

Use MATLAB to create the following plot:

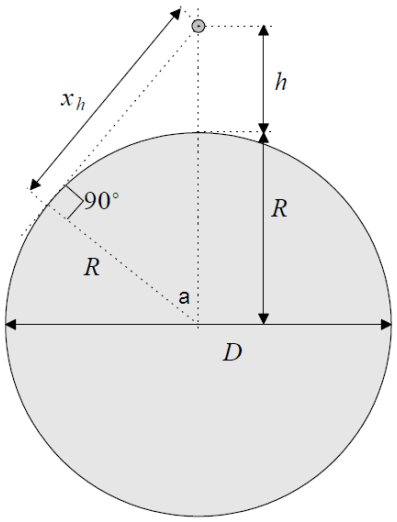


[End of Task]

Task 9: 3D Plot

Use your `xh=horizon(h,R)` from a previous task to create a **mesh** plot where you plot xh for different values of h and R respectively.

Tip! Call the function in nested For Loops for different values for h and R respectively.

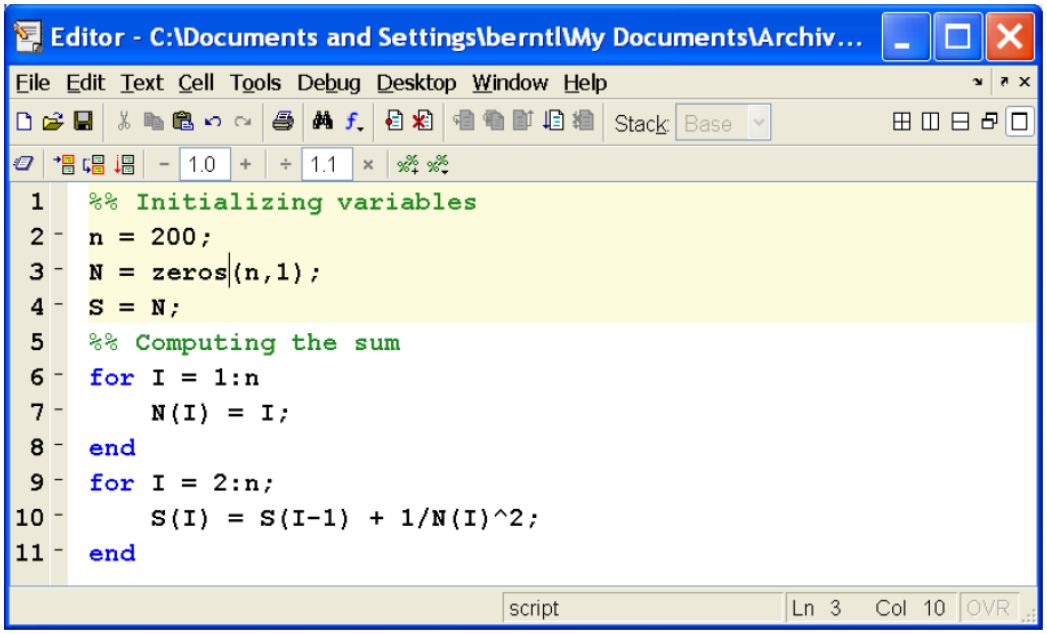


[End of Task]

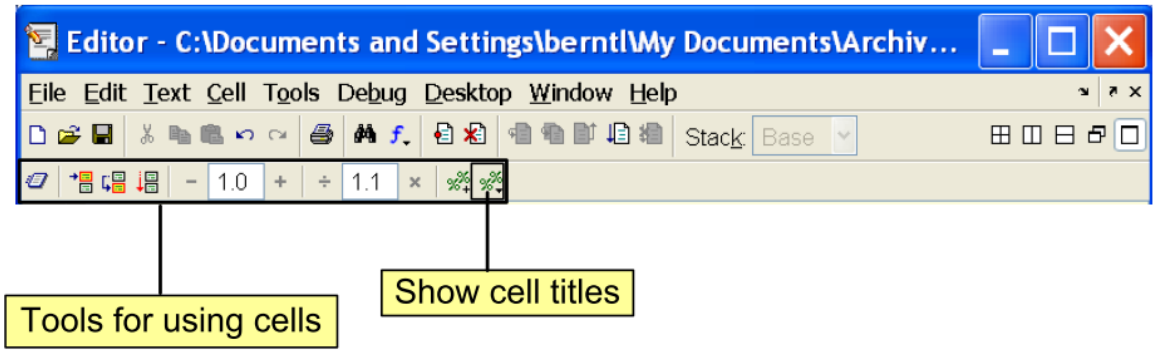
6 Using Cells in the MATLAB Editor

You may structure MATLAB code in the editor by defining text cells. A text cell is initiated by putting the symbol **%%** (double % character) in the first position of a line. The cell ends on the line preceding the next %% symbol. After the %% symbol, a space should be inserted, followed by a descriptive text. The MATLAB Editor marks a cell by framing it with a yellow box: when you put the cursor in a cell, the frame is shown. In order for this to work, the Cell Mode must have been Enabled, see the Cell menu of the MATLAB editor.

Below we see an example:



The Cells Toolbar in the Editor:



Why Use Cells?

M-files often have a natural structure consisting of multiple sections. Especially for larger files, you typically focus efforts on a single section at a time, working with the code in just that section. Similarly, when conveying information about your M-files to others, often you describe the sections of the code. To facilitate these processes, use M-file cells, where cell means a section of code. Specifically, MATLAB uses cells for Rapid code iteration in the Editor/Debugger — this makes the experimental phase of your work with M-file scripts easier.

Task 10: Using Cells

Use one of your previous scripts and divide your code into different cells. Run the different Cells individually.

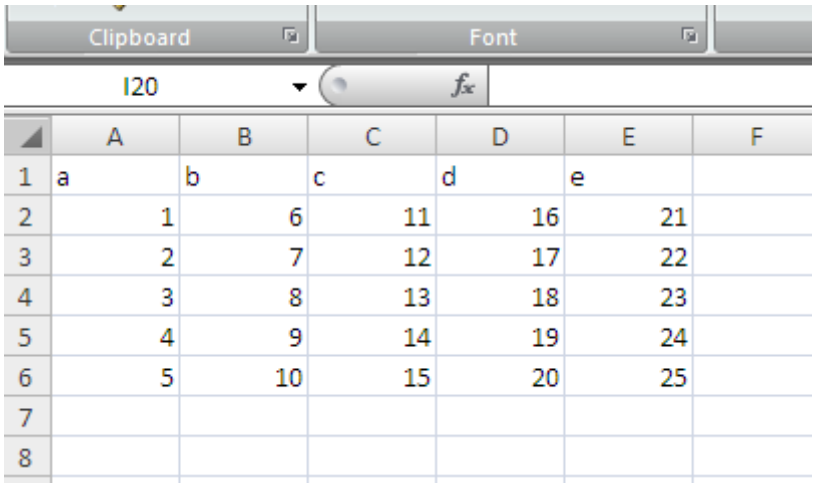
Use the Cells tools to browse between the different Cells in your script.

[End of Task]

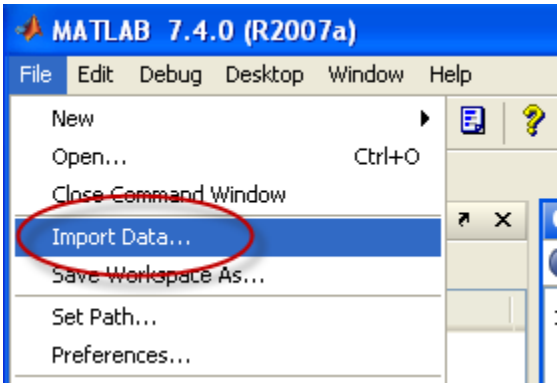
7 Importing Data

It is often needed to import data into MATLAB for analysis and calculations, it could be data in a spreadsheet or logged data from a DAQ device that you want to analyze. MATLAB have powerful tools for both importing and exporting data.

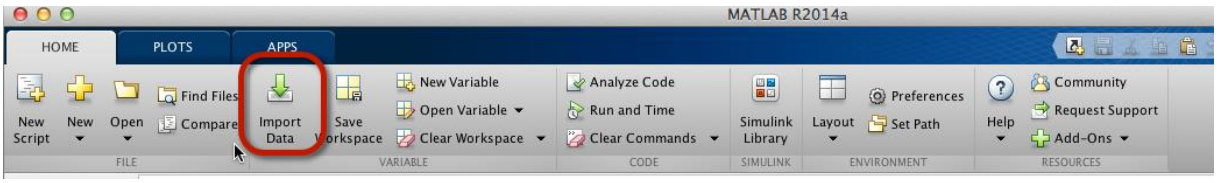
Given an Excel file:



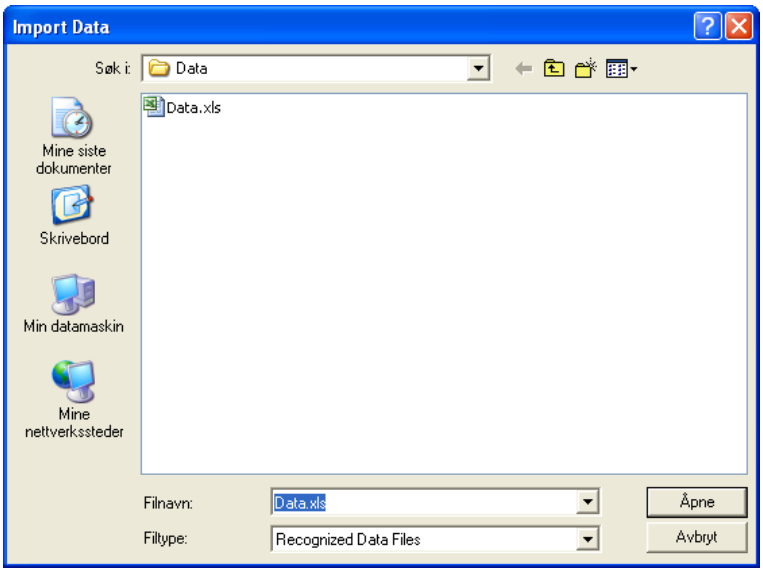
To open the Import Wizard, use File → Import Data …:



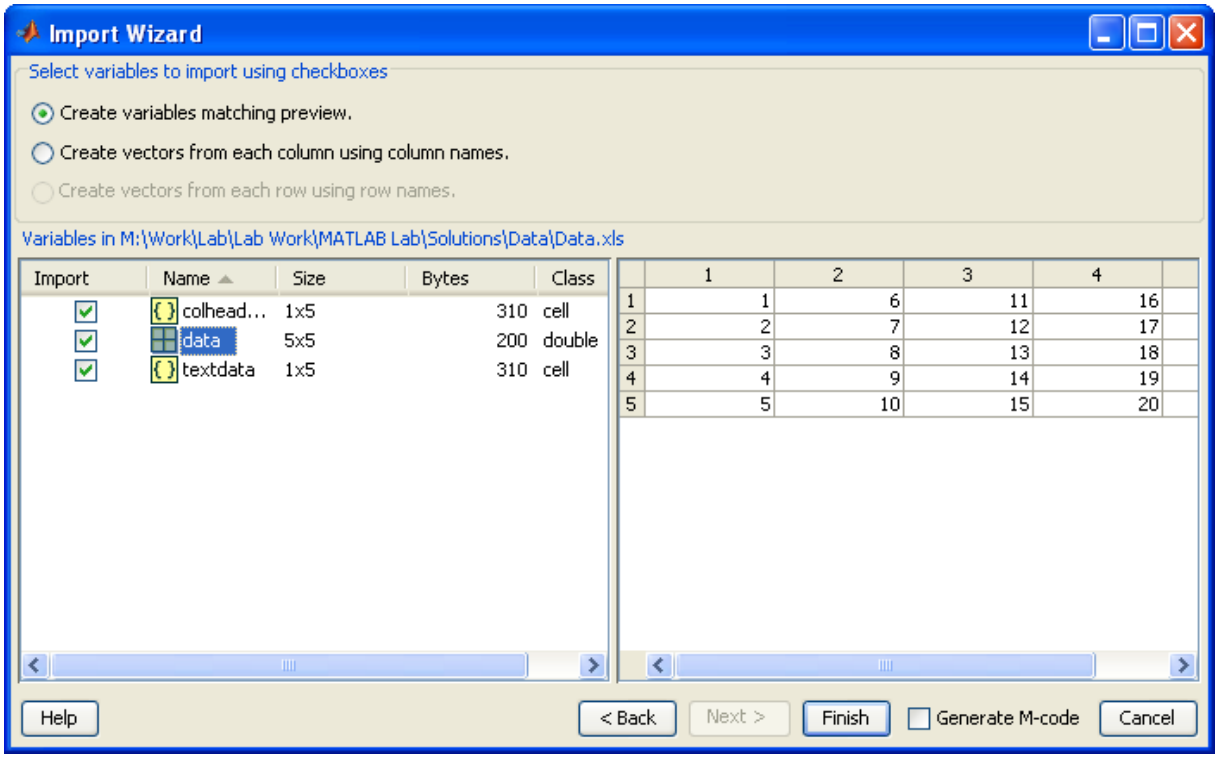
Or in newer versions of MATLAB:



A File dialog appears:



Select, e.g., a Excel Spreadsheet File, and the Import Wizard appears:



Clicking Finish and the data from the Excel file will be available in MATLAB:

```
K>> who

Your variables are:

colheaders  data        textdata

K>> data
data =
     1     6    11    16    21
     2     7    12    17    22
     3     8    13    18    23
     4     9    14    19    24
     5    10    15    20    25
K>>
```

Before you start, you should watch the video "**Importing Data from Files**".

The video is available from:
https://www.halvorsen.blog/documents/teaching/courses/matlab/matlab3.php

Task 11: Import Data

Create a spreadsheet file with some data (or use an existing spreadsheet with data if you have) and import the data into MATLAB.

Plot the data in MATLAB.

[End of Task]

8 Structures and Cell Arrays

Historically, the matrix was the only data type in MATLAB. Vectors and scalars are special cases of the general matrix. Now some new and important data structures have arrived. One is the multi-dimensional array, which just extends the matrix to more than two dimensions. More important are the **structure** and the **cell array**.

In this chapter we will use these new data structures.

Before you start, you should watch the video "**Introducing Structures and Cell Arrays**"

The video is available from:
https://www.halvorsen.blog/documents/teaching/courses/matlab/matlab3.php

8.1 Structures

A structure is a data structure that can hold diverse data types, not necessarily numbers, and with named data containers called fields, similar to a record with fields in a database.

Example:

```
>>tank.height  =  0.4;
>>tank.diameter  =  0.5;
>>tank.type  =  'cylinder';
>>tank
tank = height: 0.4000
      diameter: 0.5000
      type: 'cylinder'
```

[End of Example]

Task 12: Using Structures

Create a function that calculates the volume of different objects, such as a cylinder, a sphere, etc.

Use Structures to solve the problem.

[End of Task]

9 Alternatives to MATLAB

Here are some other alternatives to MATLAB worth mention:

- Octave
- Scilab and Scicos
- LabVIEW MathScript
- LabVIEW
- Python

9.1 Octave

Octave is a free software tool for numerical analysis and visualization. The function and command syntax are very similar to MATLAB. Many contributed functions packages (like the toolboxes in MATLAB) are available. They cover control theory, signal processing, simulation, statistics etc. They are installed automatically when you install Octave.

There is no SIMULINK-like tool in Octave, but there are many simulation functions (as in Control System Toolbox in MATLAB).

- Read more about Octave on their Homepage: http://www.gnu.org/software/octave/
- Read more about Octave on Wikipedia: http://en.wikipedia.org/wiki/GNU_Octave

9.2 Scilab and Scicos

Scilab is a free scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications.

Scilab is an open-source software. Since 1994 it has been distributed freely along with the source code via the Internet. It is currently used in educational and industrial environments around the world.

Scilab is quite like MATLAB, and the range of functions are comparable.

Octave is more like MATLAB than to Scilab. One problem with Octave has been that data plotting is more cumbersome in Octave than in Scilab.

One nice thing about Scilab is that you get Scicos automatically installed when you install Scilab. Scicos is a block-diagram based simulation tool like Simulink and LabVIEW Simulation Module.

- Read more about Scilab on their Homepage: http://www.scilab.org/
- Read more about Scilab on Wikipedia: http://en.wikipedia.org/wiki/Scilab

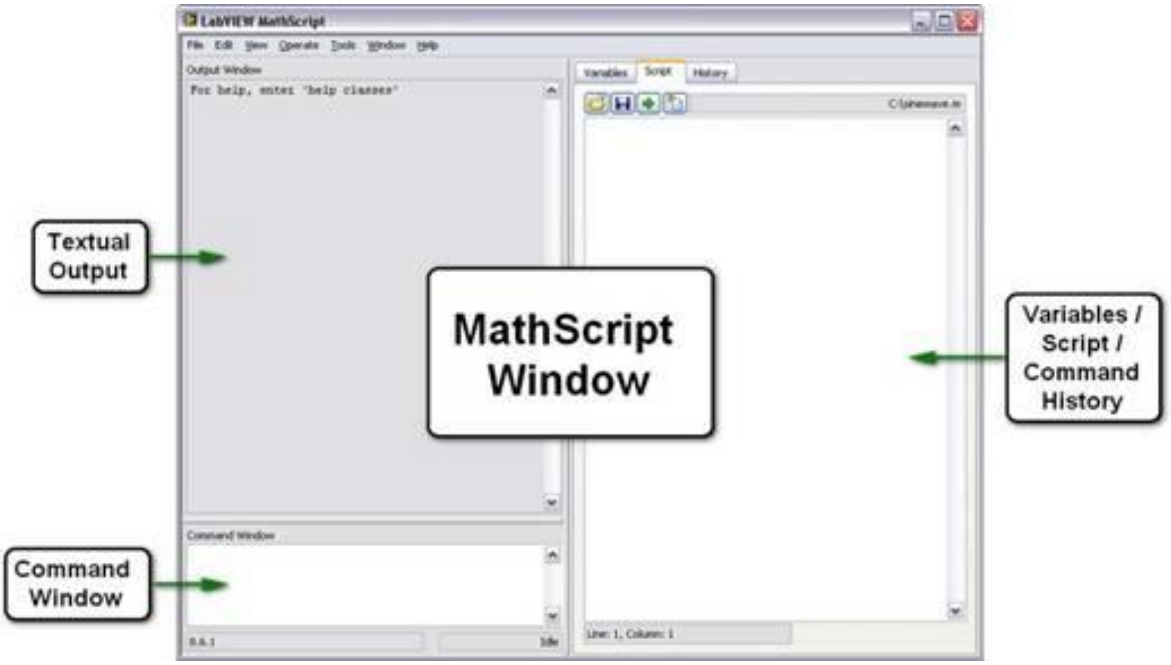
9.3 LabVIEW MathScript

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.

For more information about MathScript, please read the Tutorial “LabVIEW MathScript”.

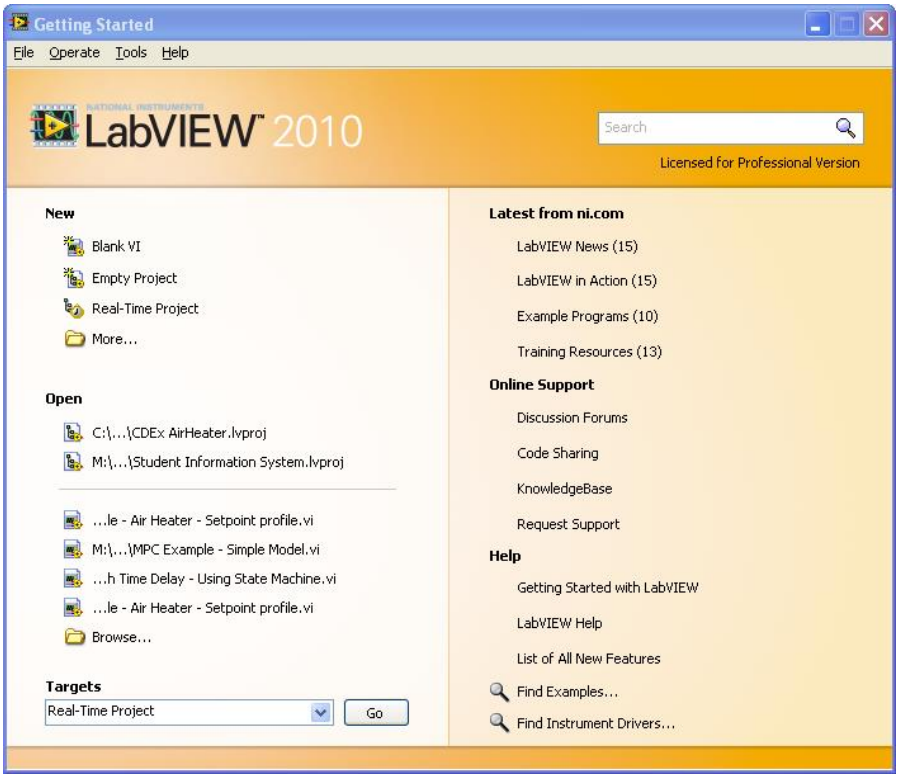
MathScript is an add-on module to LabVIEW but you don’t need to know LabVIEW programming in order to use MathScript.



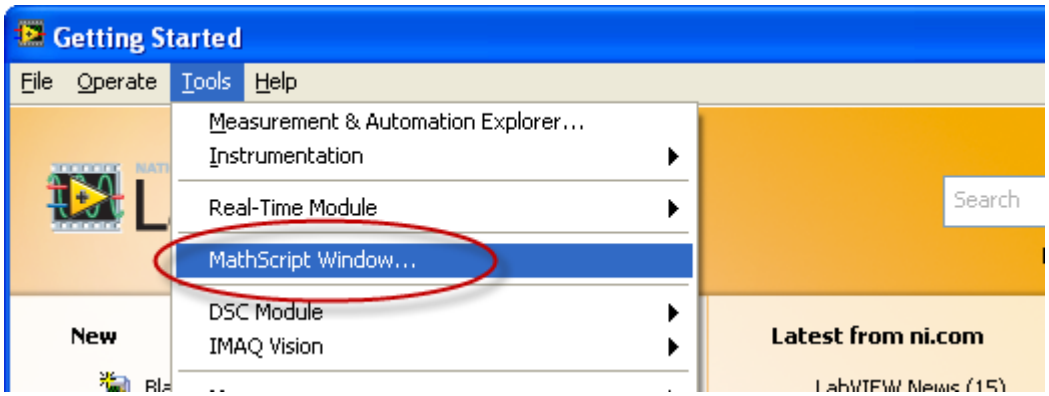
For more information about MathScript, please read the Tutorial "LabVIEW MathScript".

9.3.1 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:

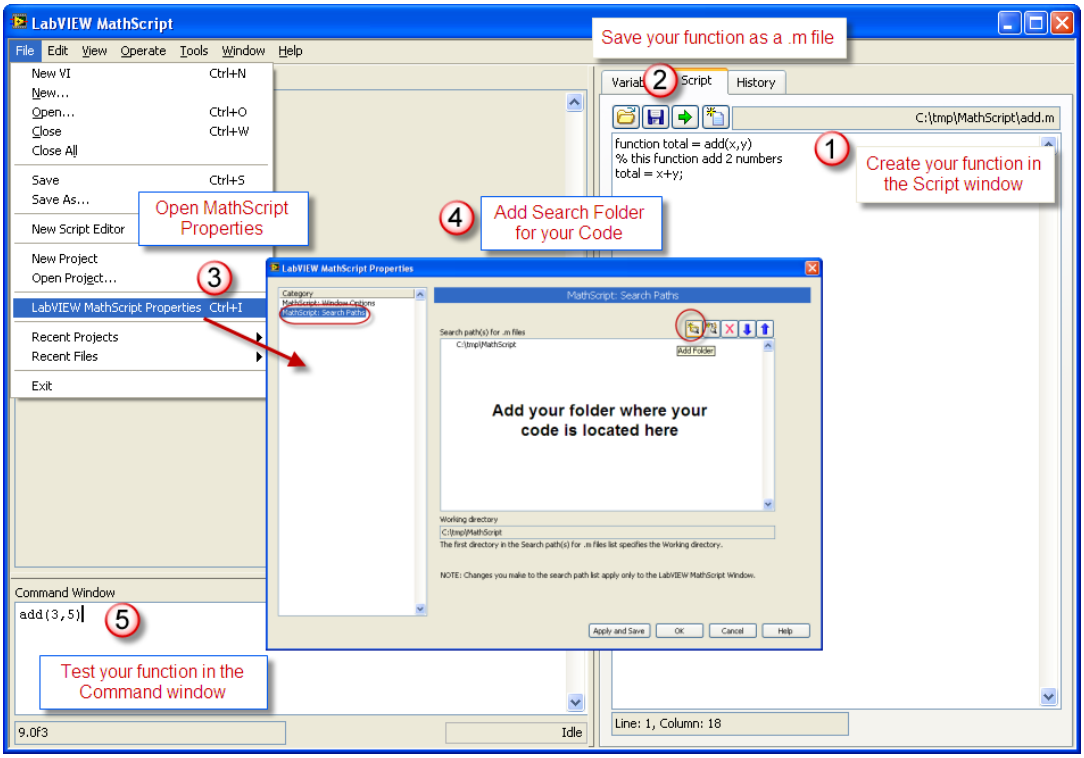


In the **Getting Started** window, select **Tools -> MathScript Window**...:



9.3.2 Functions

The figure below illustrates how to create and use functions in MathScript:



9.3.3 ODE Solvers in MathScript

MathScript also offers some ODE solvers, not as many as MATLAB and other names, but the principle is quite the same.

Below we see a list with available ODE solvers:

ode (MathScript RT Module Class)

Requires: MathScript RT Module

Use members of the ode class to solve continuous ordinary differential equations.

Function	Description
ode_adams	Adams-Moulton ODE solver
ode_bdf15	BDF ODE solver
ode_bdf23	BDF ODE solver
ode_radau5	Determines the y-values in an ODE system using the RADAU 5 ODE solver
ode_rk23	Runge-Kutta 23 ODE solver
ode_rk45	Runge-Kutta 45 ODE solver
ode_rosen	Rosenbrock ODE solver
odepset	Gets or sets the parameters for an ODE solver

Below we see the description for the ode_rk23 function:

ode_rk23 (MathScript RT Module Function)

Owning Class: ode

Requires: MathScript RT Module

Syntax

```
[t, y] = ode_rk23(fun, times, y0)
```

Legacy Name: ode23

Description

Uses the Runge-Kutta 23 ODE solver to determine the y-values in an ODE system.

Details

Examples

Inputs

Name	Description
fun	Specifies the name of the function that describes the ODE system. The function must have the following form: `function dy = fun(times, y)`. **fun** is a string.
times	Specifies either the starting time and ending time for the time span, such as [0, 20], or the time points, such as 1:20. **times** must be strictly monotonic and is a real, double-precision vector.
y0	Specifies the y-value at the starting time. **y0** is a real, double-precision vector.

Outputs

Name	Description
t	Returns the time values at which LabVIEW evaluates the y-values. **t** is a real, double-precision vector.
y	Returns the y-values that LabVIEW approximates. **y** is a real, double-precision matrix.

Details

If you specify the starting and ending time for the time span in **times**, LabVIEW automatically adjusts the time step size throughout the calculation to ensure that the error per step remains at a given relative and absolute tolerance. If you specify the time points in **times**, LabVIEW also automatically adjusts the time step size to ensure a more accurate calculation. However, LabVIEW returns only the y-values at the time points you specify, and **t** equals **times**.

You can use the `odepset` function to set the relative and absolute tolerance as well as other parameters the MathScript ODE solver uses.

This function is not supported in the LabVIEW Run-Time Engine. Do not use this function in a stand-alone application or shared library.

Examples

```
% The lorenz function is defined by:
% function dy = lorenz(times, y)
% dy = zeros(3, 1);
% dy(1) = 10*(y(2)-y(1));
% dy(2) = 28*y(1)-y(2)-y(1)*y(3);
% dy(3) = y(1)*y(2)-8/3*y(3);
[t, y] = ode_rk23('lorenz', [0, 5], [1; 1; 1])
```

9.4 LabVIEW

LabVIEW is a graphical programming language, and it is well suited for Control and Simulation applications.

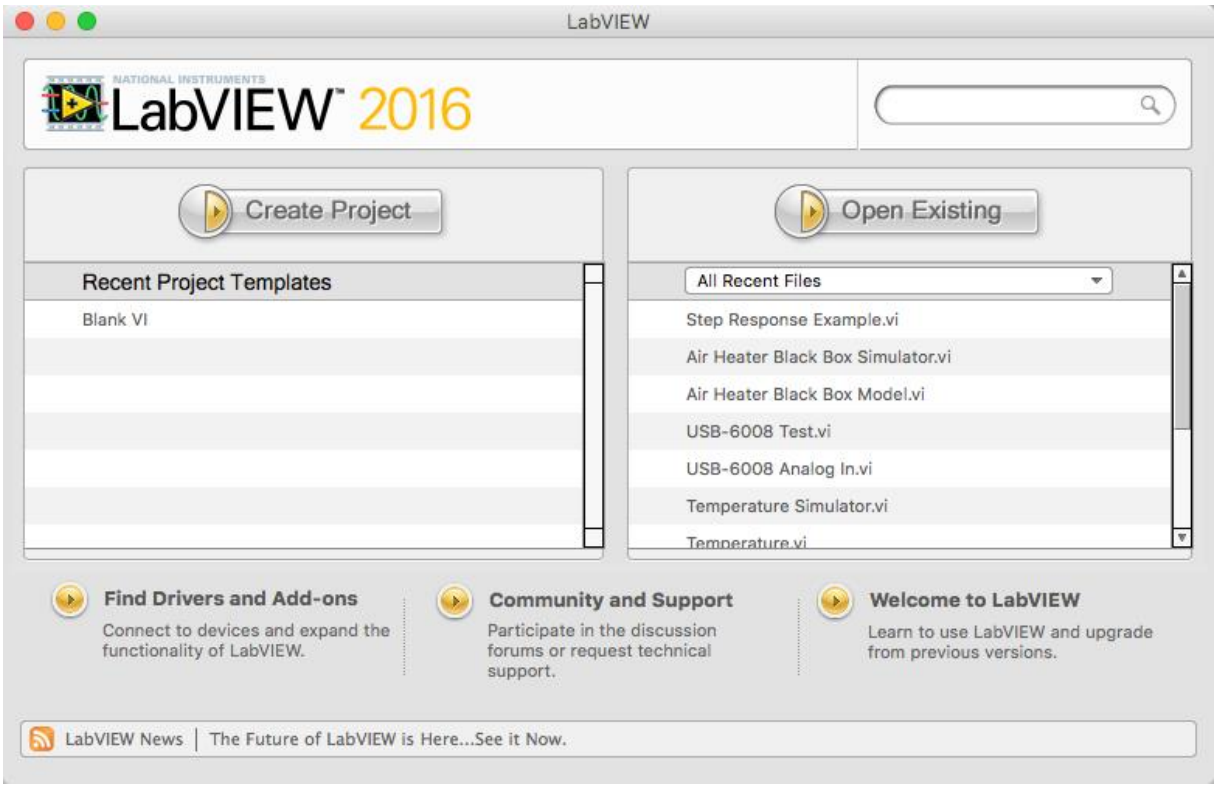
In this chapter we will use LabVIEW to create a block diagram model and simulate it, similar to what we have done in Simulink.

9.4.1 The LabVIEW Environment

LabVIEW programs are called **V**irtual **I**nstruments, or VIs, because their appearance and operation imitate physical instruments, such as

oscilloscopes and multimeters. LabVIEW contains a comprehensive set of tools for acquiring analyzing, displaying, and storing data, as well as tools to help you troubleshoot your code.

When opening LabVIEW, you first come to the "Getting Started" window.

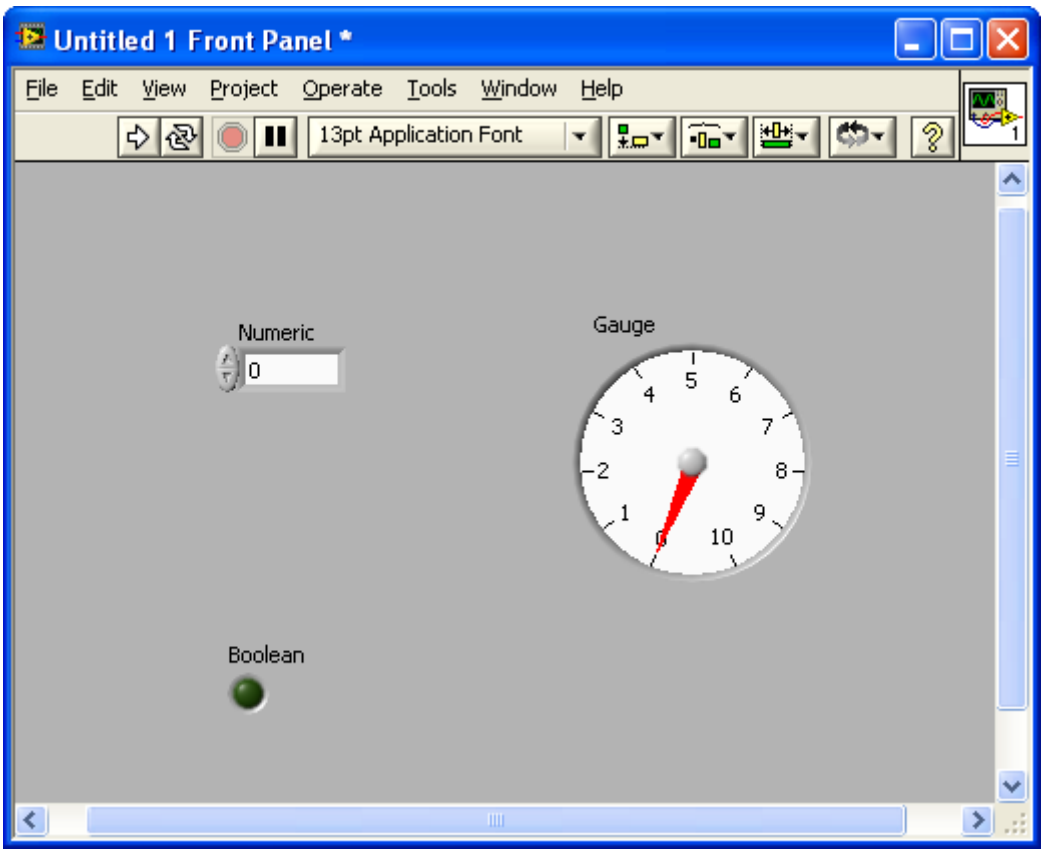


In order to create a new VI, select "Blank VI" or in order to create a new LabVIEW project, select "Empty project".

When you open a blank VI, an untitled front panel window appears. This window displays the front panel and is one of the two LabVIEW windows you use to build a VI. The other window contains the block diagram. The sections below describe the front panel and the block diagram.

9.4.2 Front Panel

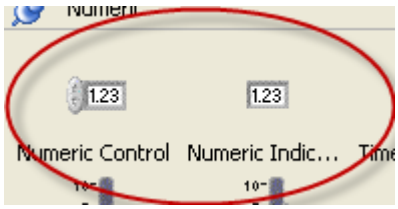
When you have created a new VI or selected an existing VI, the Front Panel and the Block Diagram for that specific VI will appear.



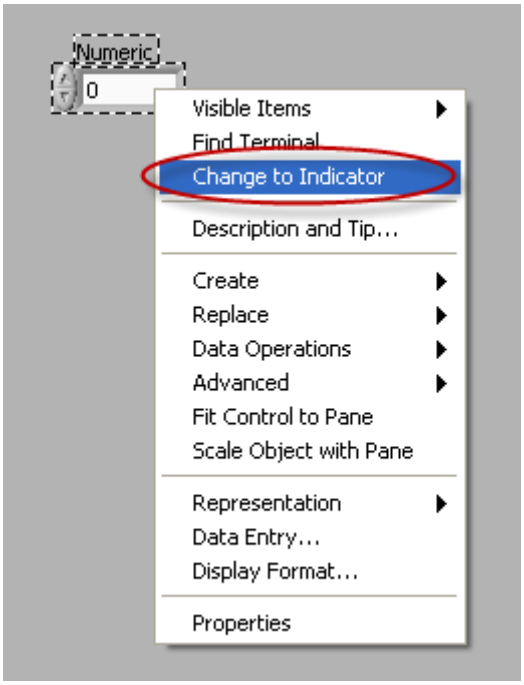
In LabVIEW, you build a user interface, or front panel, with controls and indicators. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays.

You build the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays. Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices and display data the block diagram acquires or generates.

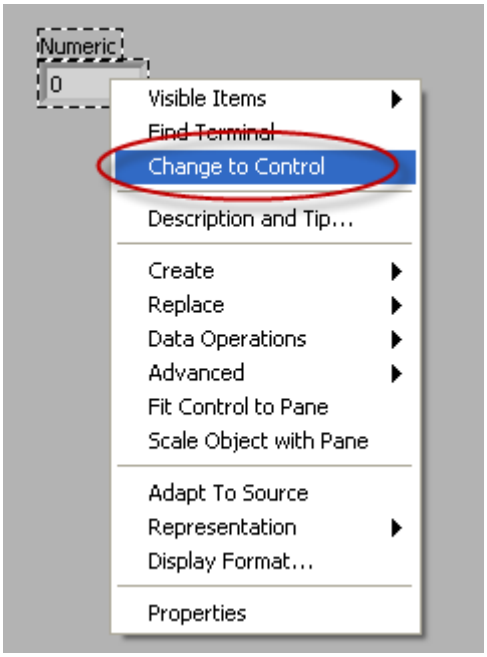
E.g., a "Numeric" can either be a "Numeric Control" or a "Numeric Indicator", as seen below.



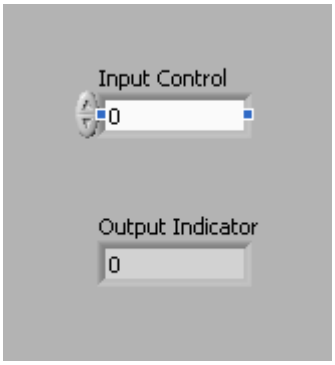
I you select a "Numeric Control", it can easy be changed to an "Numeric Indicator" by right click on the object an select "Change to Indicator"



Or opposite, I you select a "Numeric Indicator", it can easy be changed to an "Numeric Control" by right click on the object an select "Change to Control"



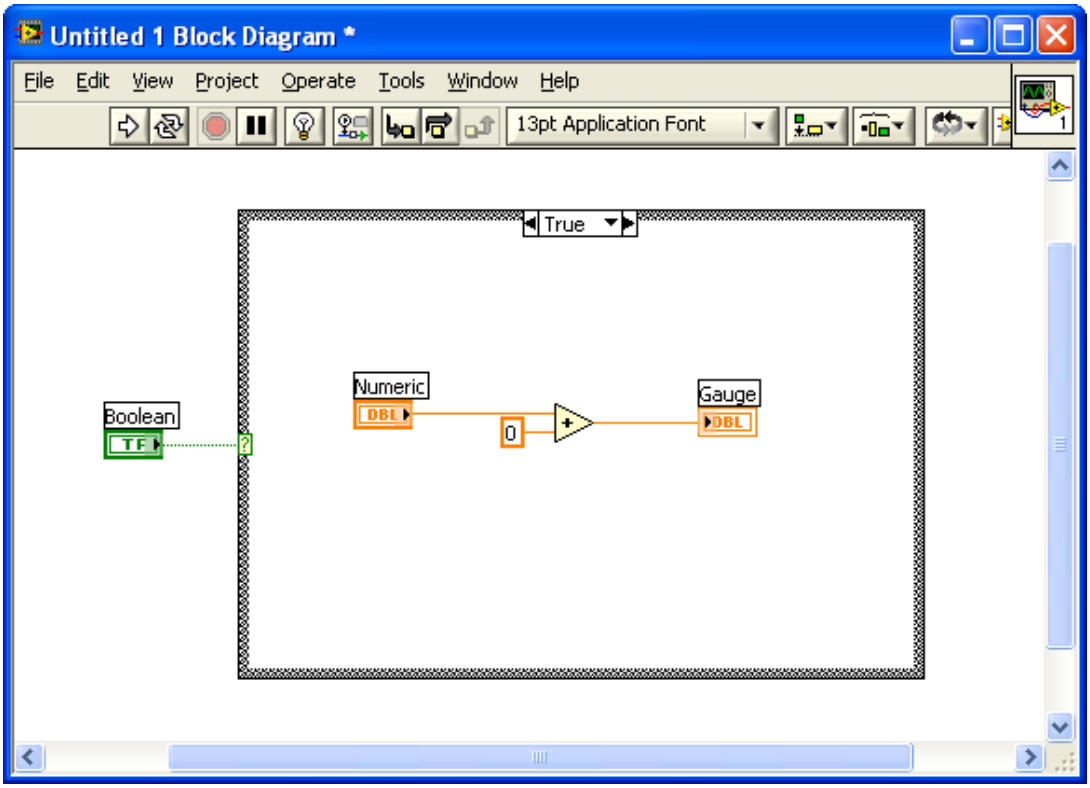
The difference between a "Numeric Control" and a "Numeric Indicator" is that for a "Numeric Control" you may enter a value, while the "Numeric Indicator" is read-only, i.e., you may only read the value, not change it.



The appearance is also slightly different, the "Numeric Control" has an increment and an decrement button in front, while the "Numeric Indicator" has a darker background color in order to indicate that its read-only.

9.4.3 Block Diagram

After you build the user interface, you add code using VIs and structures to control the front panel objects. The block diagram contains this code. In some ways, the block diagram resembles a flowchart.



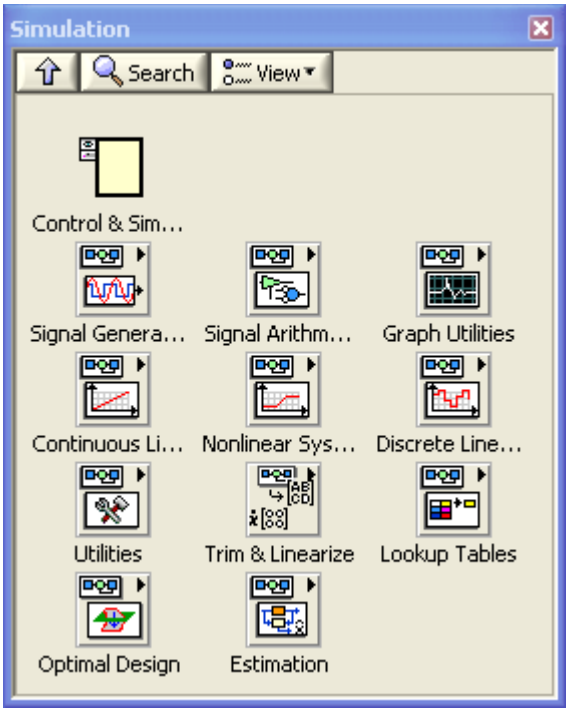
After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code. Front panel objects appear as

terminals, on the block diagram. Block diagram objects include terminals, subVIs, functions, constants, structures, and wires, which transfer data among other block diagram objects.

9.4.4 LabVIEW Control Design and Simulation Module

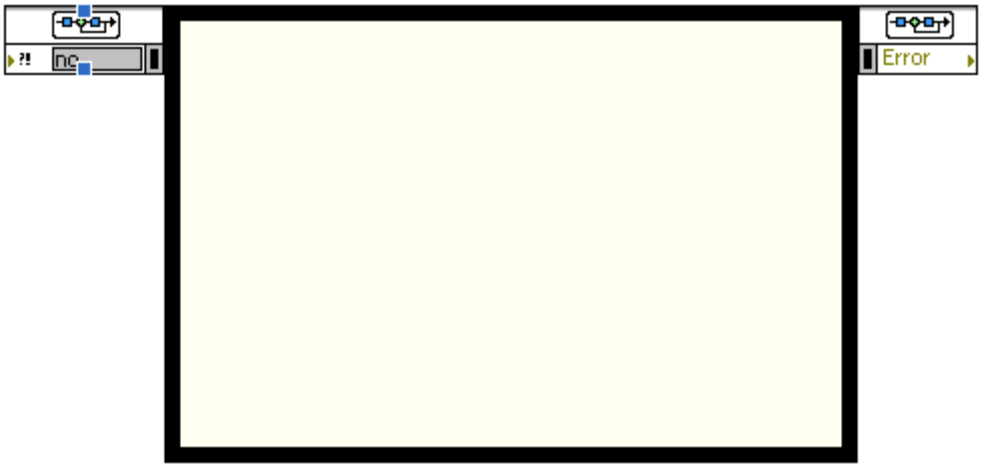
In this chapter we will focus on how to create and simulate a model using the "**Simulation Loop**" and the corresponding blocks available in LabVIEW.

Below we see the **Simulation palette** in LabVIEW with "Simulation Loop" and the corresponding blocks available:



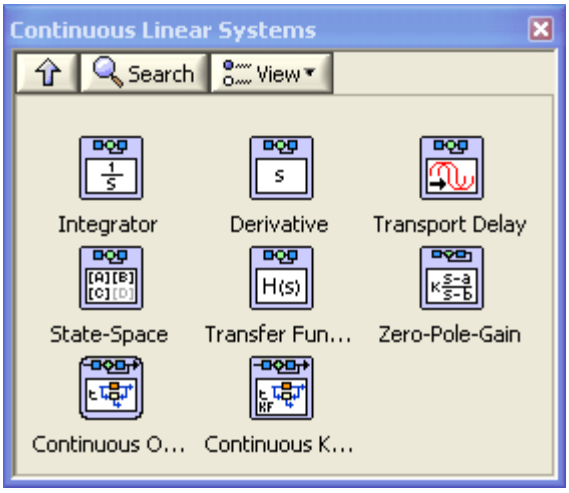
In the "**Simulation**" Sub palette we have the "Control and Simulation Loop" which is very useful in simulations:

You must place all Simulation functions within a Control & Simulation Loop or in a simulation subsystem. You also can place simulation subsystems within a Control & Simulation Loop or another simulation subsystem, or you can place simulation subsystems on a block diagram outside a Control & Simulation Loop or run the simulation subsystems as stand-alone VIs.



The Control & Simulation Loop has an Input Node (upper left corner) and an Output Node (upper right corner). Use the Input Node to configure simulation parameters programmatically. You also can configure these parameters interactively using the Configure Simulation Parameters dialog box. Access this dialog box by double-clicking the Input Node or by right-clicking the border and selecting Configure Simulation Parameters from the shortcut menu.

In the "**Continuous Linear Systems**" Sub palette we have important blocks for we will use when creating our model:



The most used blocks probably are Integrator, Transport Delay, State-Space and Transfer Function.

When you place these blocks on the diagram you may double-click or right-click and then select "Configuration…"

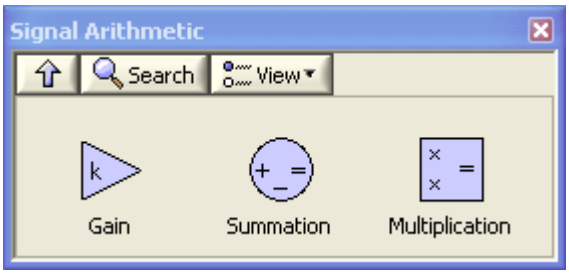
Integrator - Integrates a continuous input signal using the ordinary differential equation (ODE) solver you specify for the simulation.

Transport Delay - Delays the input signal by the amount of time you specify.

Transfer Function - Implements a system model in transfer function form. You define the system model by specifying the Numerator and Denominator of the transfer function equation.

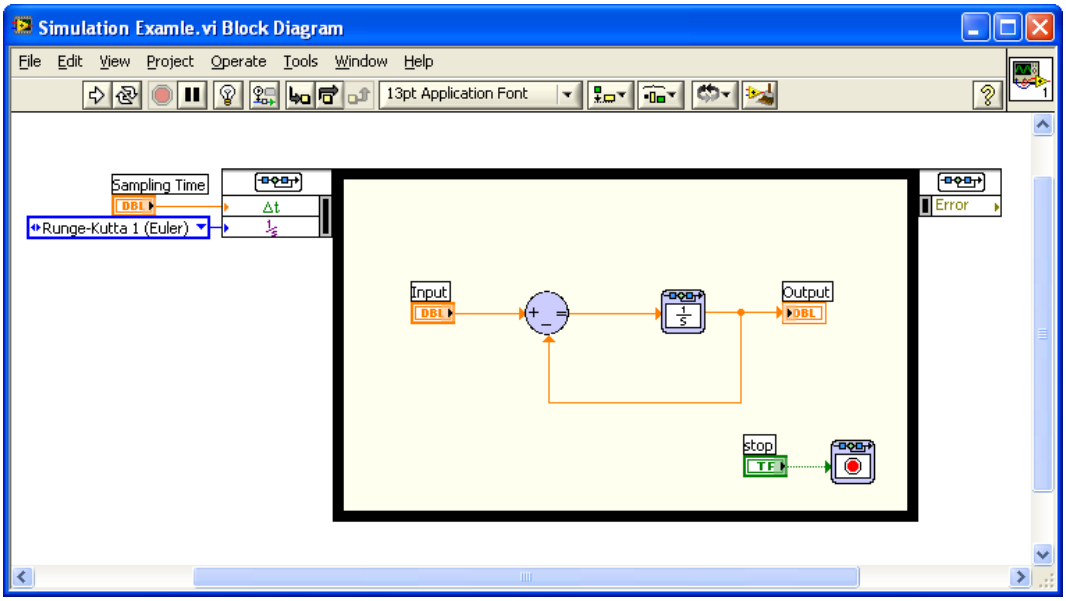
State-Space - Implements a system model in state-space form. You define the system model by specifying the input, output, state, and direct transmission matrices.

The "**Signal Arithmetic**" Sub palette is also useful when creating a simulation model:



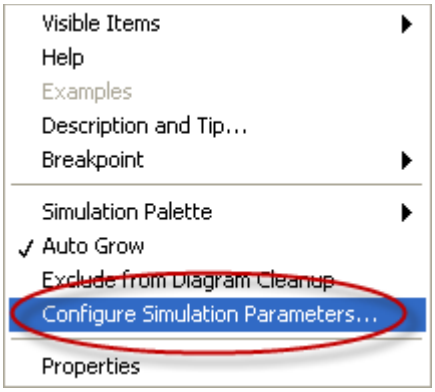
Example:

Below we see an example of a simulation model using the Control and Simulation Loop.

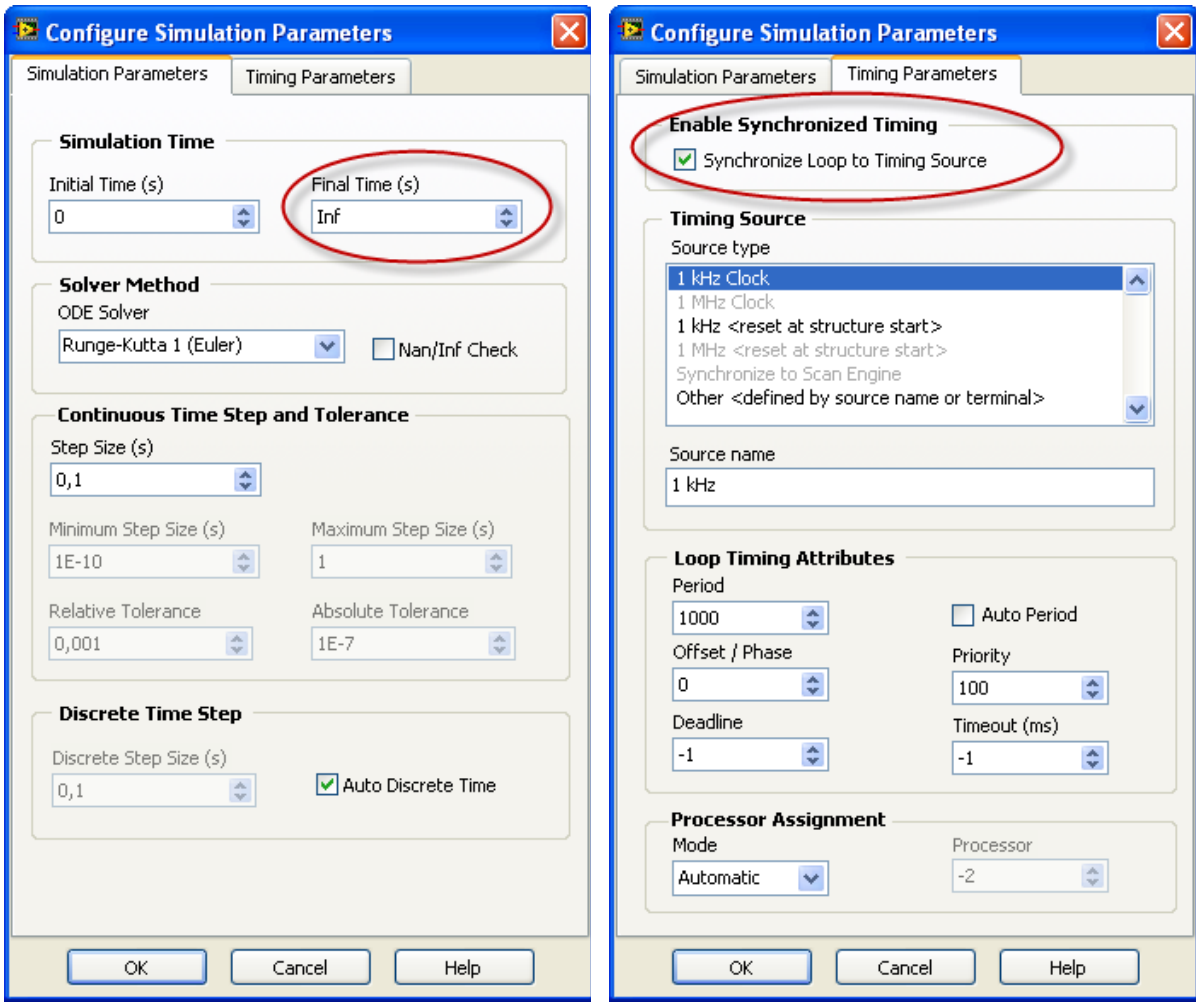


Notice the following:

Click on the border of the simulation loop and select "**Configure Simulation Parameters...**"



The following window appears (Configure Simulation Parameters):

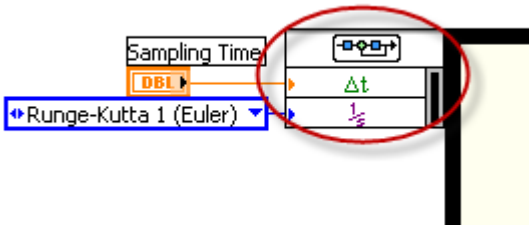


In this window you set some Parameters regarding the simulation, some important are:

- **Final Time (s)** – set how long the simulation should last. For an infinite time set "**Inf**".
- **Enable Synchronized Timing** - Specifies that you want to synchronize the timing of the Control & Simulation Loop to a timing source. To enable synchronization, place a checkmark in this checkbox and then choose a timing source from the Source type list box.

Click the Help button for more details.

You may also set some of these Parameters in the Block Diagram:



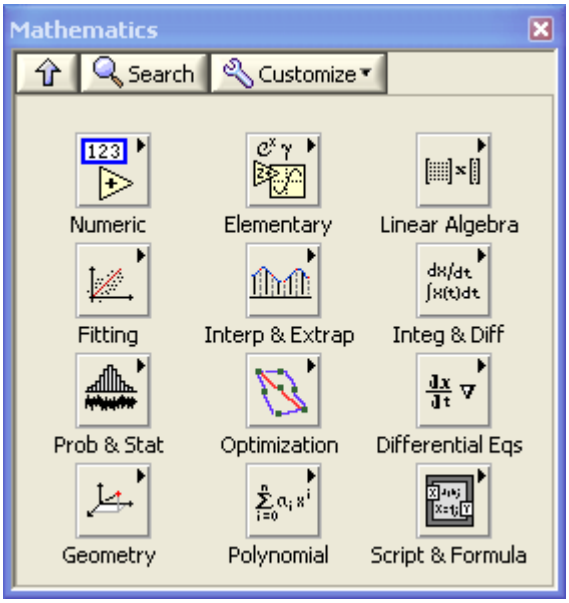
You may use the mouse to increase the numbers of Parameters and right-click and select "Select Input".

[End of Example]

9.5 Mathematics in LabVIEW

When it comes to mathematics and numerical techniques, LabVIEW offers functionality similar to what exists in MATLAB.

Below we see the **Mathematics** palette in LabVIEW:



Here we have functionality for:

- Basic math operations
- Linear Algebra
- Curve Fitting
- Interpolation
- Integration and Differentiation
- Statistics
- Optimization
- Differential Equations (ODEs)
- Polynomials
- MATLAB integration (MATLAB Script)
- etc.

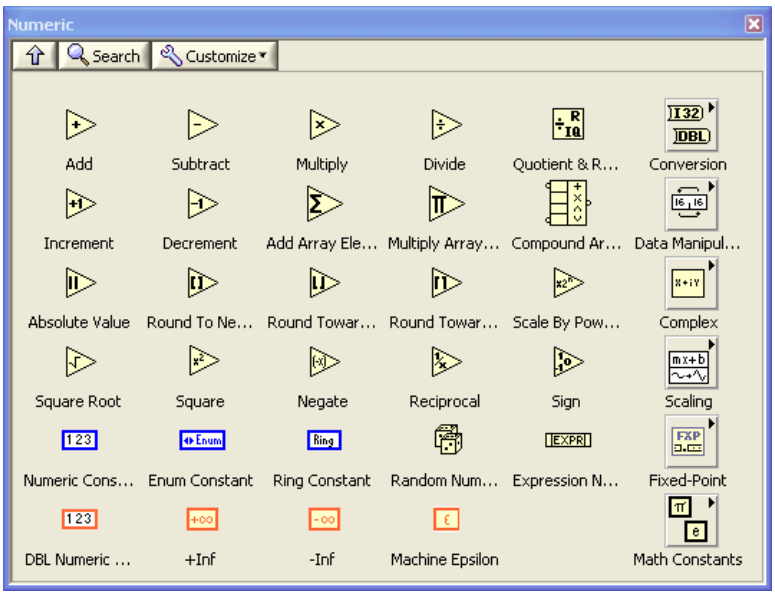
Below we will take a closer look at some of these functions.

9.5.1 Basic Math

LabVIEW have lots of functionality for basic math operations, trigonometric functions, etc.

Numeric Palette in LabVIEW:

Below we see the **Numeric** palette in LabVIEW:

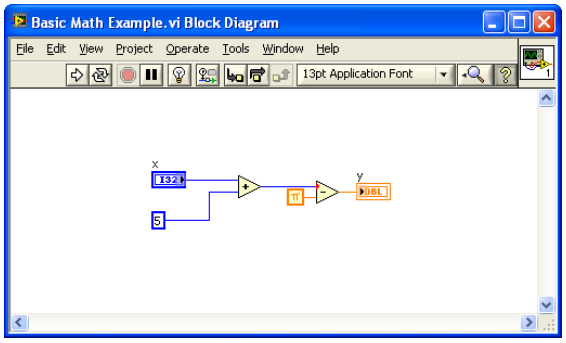


Here we have basic math functions, such as Add, Subtract, Multiply, Divide, etc.

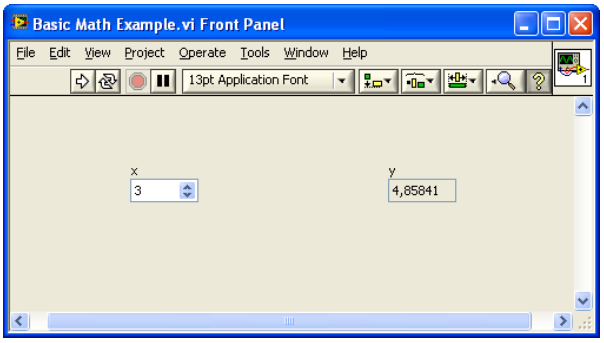
Example:

Below we see a simple example using the basic math features:

Block Diagram:



Front Panel:

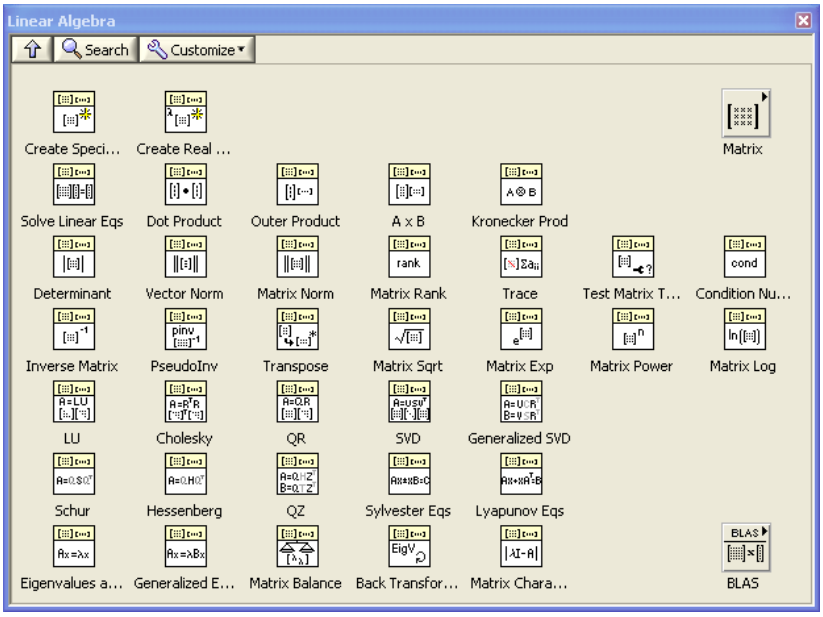


[End of Example]

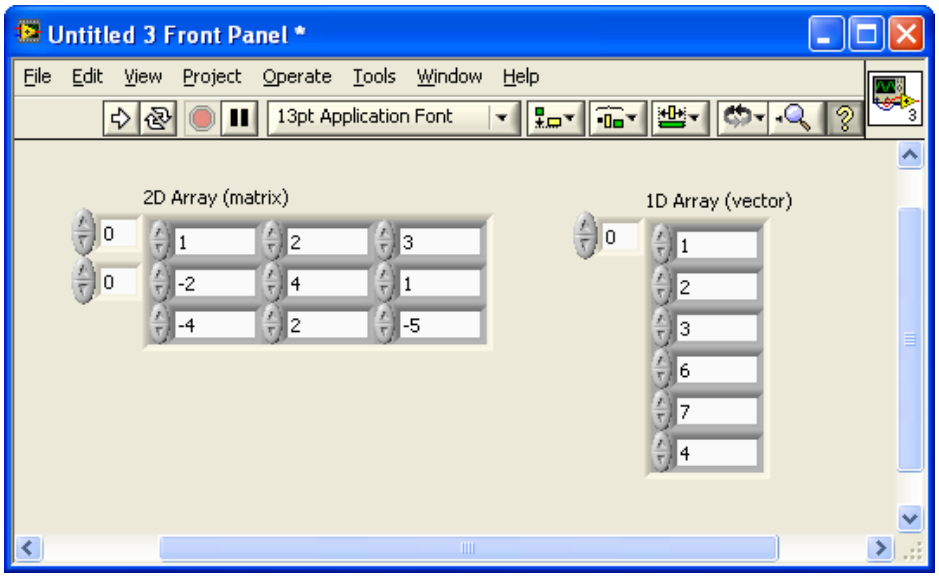
9.5.2 Linear Algebra

LABVIEW have lots of VIs (functions) for Linear Algebra. Below we see the Linear Algebra palette in LabVIEW:

Linear Algebra Palette in LabVIEW:



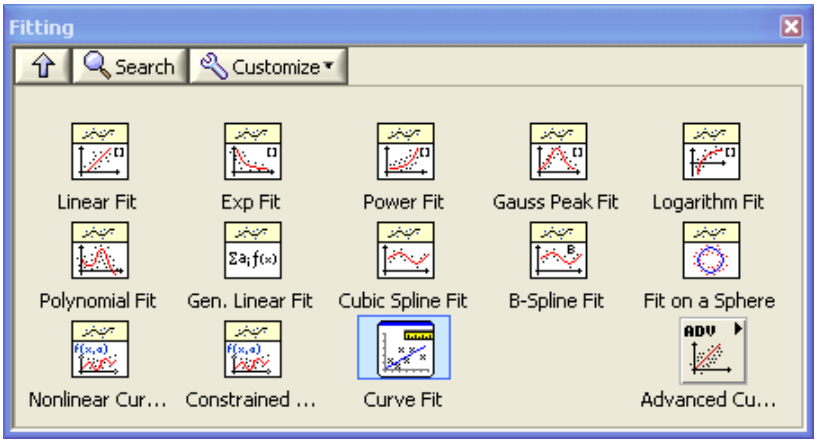
In LabVIEW is a matrix defined as a 2-dimensional array, while a vector is defined as a 1 dimensional array, see Figure below:



9.5.3 Curve Fitting

LabVIEW offers lots of functionality for Curve Fitting.

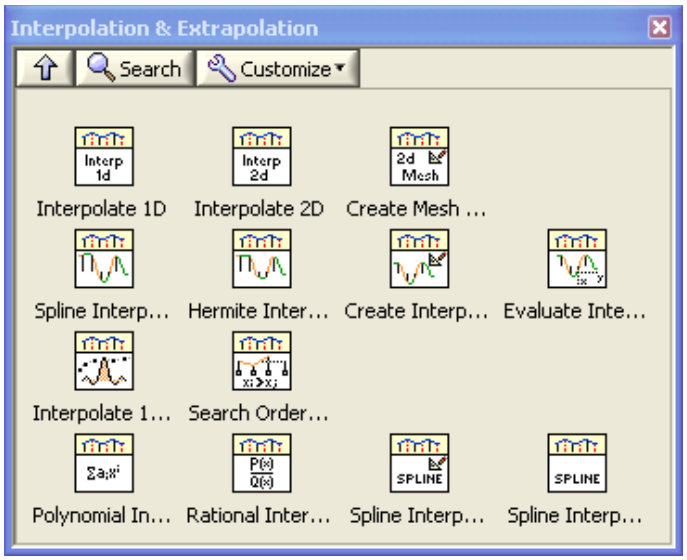
Fitting palette in LabVIEW:



9.5.4 Interpolation

LabVIEW offers lots of functionality for Interpolation.

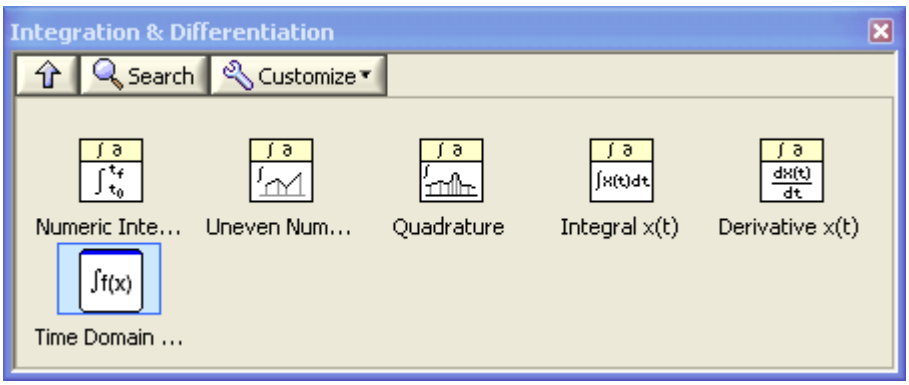
Interpolation & Extrapolation palette in LabVIEW:



9.5.5 Integration and Differentiation

LabVIEW offers lots of functionality for numerical integration and differentiation.

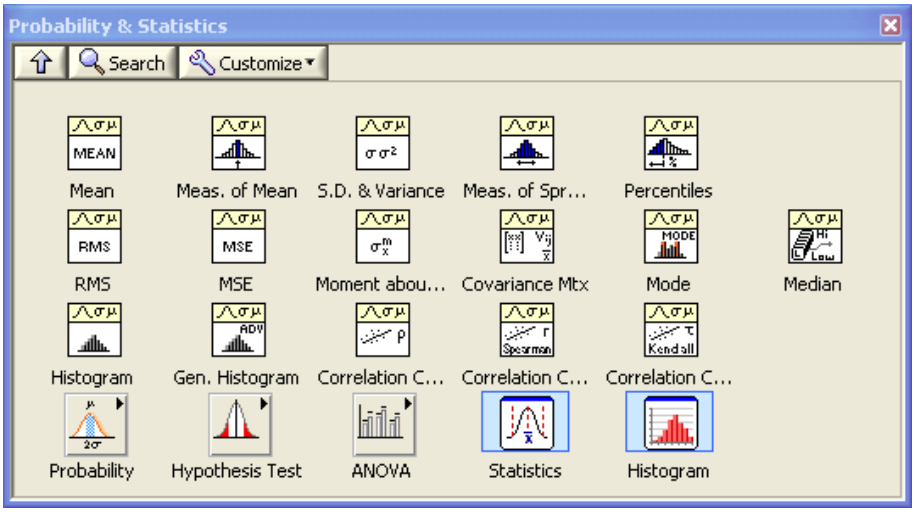
Integration and Differentiation palette in LabVIEW:



9.5.6 Statistics

LabVIEW offers lots of functionality for Statistics, including basic functionality for finding mean, median, standard deviation, etc.

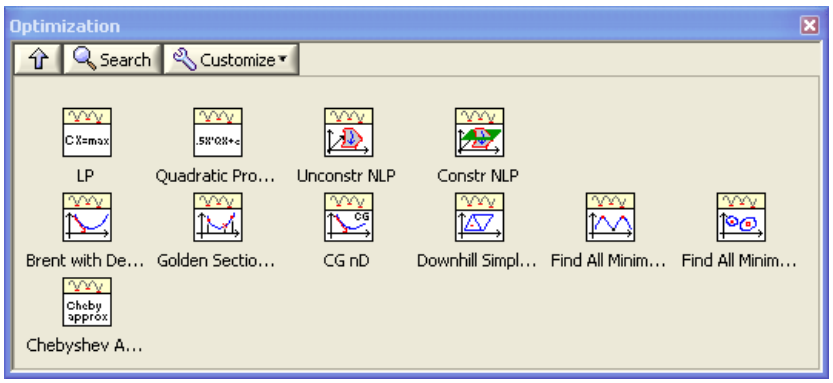
Probability and Statistics palette in LabVIEW:



9.5.7 Optimization

LabVIEW offers lots of functionality for Optimization.

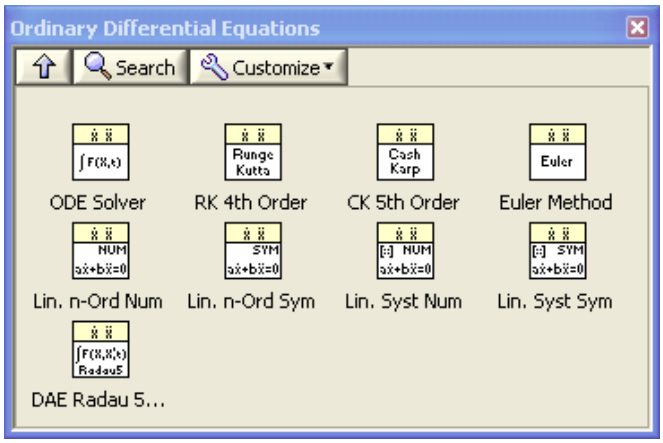
Optimization palette in LabVIEW:



9.5.8 Differential Equations (ODEs)

LabVIEW offers lots of functionality for solving Differential Equations.

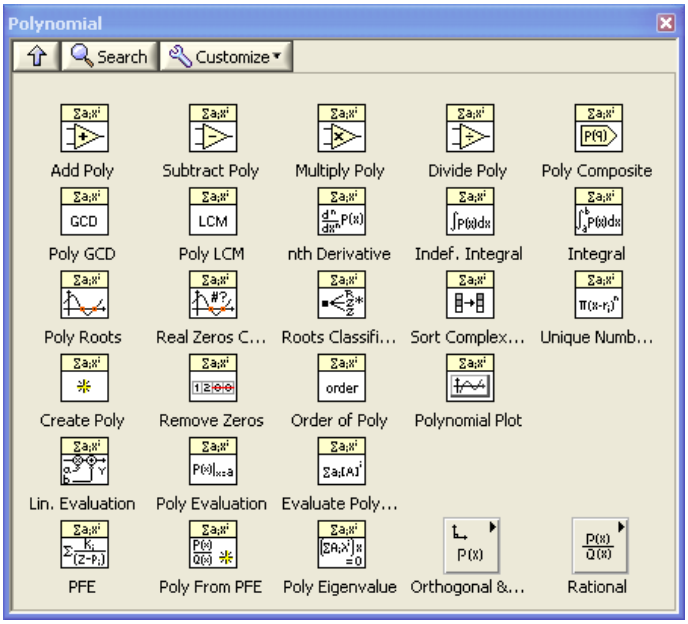
Ordinary Differential Equations palette in LabVIEW:



9.5.9 Polynomials

LabVIEW offers lots of functionality for creating and manipulating Polynomials.

Polynomial palette in LabVIEW:



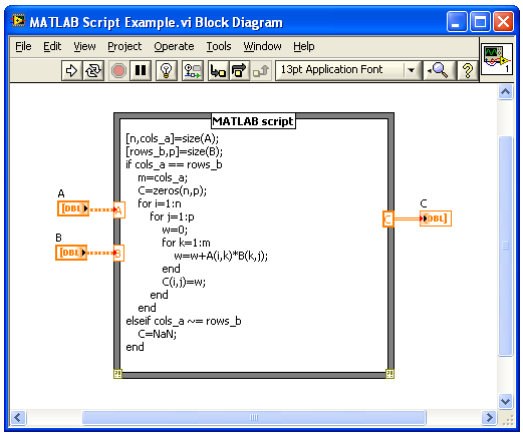
9.6 MATLAB Integration (MATLAB Script) in LabVIEW

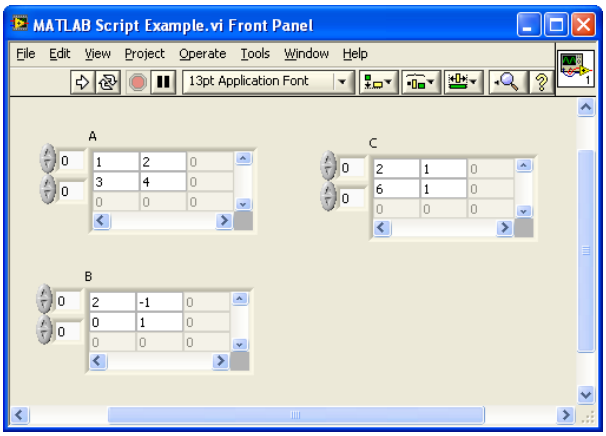
It is possible to integrate MathScript code in LabVIEW, which has similar syntax as MATLAB (see previous chapter about MathScript). In addition, there is also possible to integrate MATLAB code directly using something called the "MATLAB Script".

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.

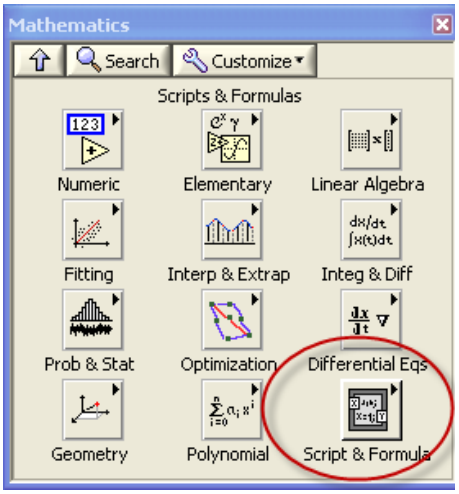
Example:

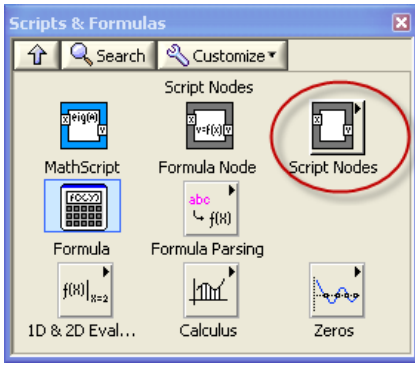
Block Diagram: Front Panel:

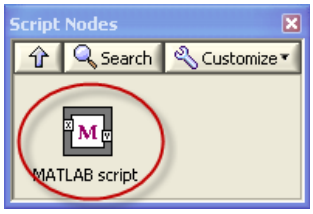




The MATLAB Script Node is found in the Mathematics palette (Mathematics → Scripts & Formulas → Script Nodes):







9.7 Python

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language.

You can install a basic Python IDE from www.python.org.

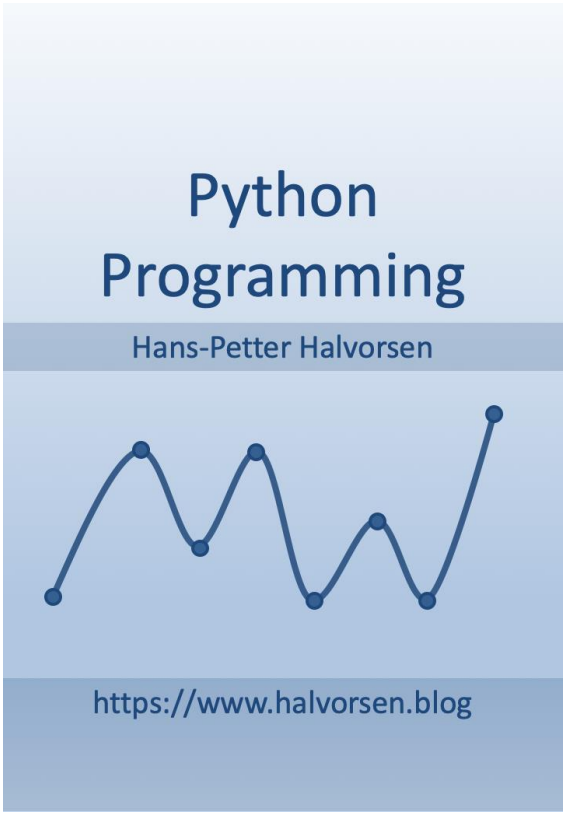
For more MATLAB look and feel the Anaconda Python distribution is recommended (with all major Scientific packages included, such as NumPy, SciPy, Matplotlib...).

The Spyder IDE is also included (which has much more features than the basic IDE).

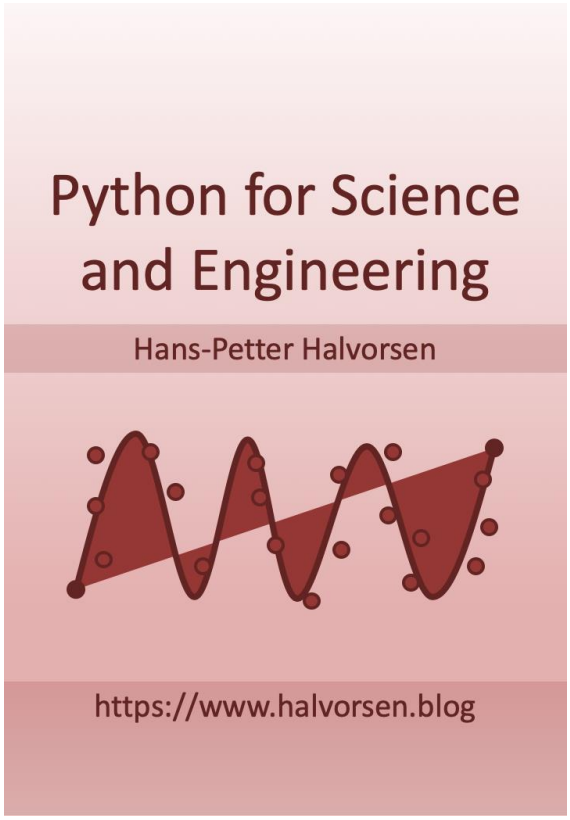
I have written the following Python textbooks:

Python Programming: This is a textbook in Python Programming with lots of Practical Examples and Exercises. You will learn the necessary foundation for basic programming with focus on Python.

Python for Science and Engineering: This is a textbook in Python Programming with lots of Examples, Exercises, and Practical Applications within Mathematics, Simulations, etc. The focus is on numerical calculations in mathematics and engineering. Necessary theory is presented in addition to many practical examples.



Python Programming



Python for Science and Engineering

These textbooks can be downloaded for free. These textbooks have many of the same examples and exercises you find in the MATLAB textbooks.

For more information, please see the following:

https://www.halvorsen.blog/documents/programming/python/

Appendix A – MathScript Functions

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

Function	Description	Example
plot	Generates a plot. plot(y) plots the columns of y against the indexes of the columns.	>X = [0:0.01:1]; >Y = X.*X; >plot(X, Y)
tf	Creates system model in transfer function form. You also can use this function to state-space models to transfer function form.	>num=[1]; >den=[1, 1, 1]; >H = tf(num, den)
poles	Returns the locations of the closed-loop poles of a system model.	>num=[1] >den=[1,1] >H=tf(num,den) >poles(H)
tfinfo	Returns information about a transfer function system model.	>[num, den, delay, Ts] = tfinfo(SysInTF)
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.	>num=[1,1]; >den=[1,-1,3]; >H=tf(num,den); >t=[0:0.01:10]; >step(H,t);
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.	>t = [0:0.1:10] >u = sin(0.1*pi*t)' >lsim(SysIn, u, t)
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.	>K = 1; >tau = 1; >H = sys_order1(K, tau)
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.	>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)
damp	Returns the damping ratios and natural frequencies of the poles of a system model.	>[dr, wn, p] = damp(SysIn)
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.	>Kc = 0.5; >Ti = 0.25; >SysOutTF = pid(Kc, Ti, 'academic');

conv	Computes the convolution of two vectors or matrices.	`>C1 = [1, 2, 3];`
		`>C2 = [3, 4];`
		`>C = conv(C1, C2)`

series	Connects two system models in series to produce a model SysSer with input and output connections you specify	`>Hseries = series(H1,H2)`
feedback	Connects two system models together to produce a closed-loop model using negative or positive feedback connections	`>SysClosed = feedback(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 = eye(2)`
		`>B = [0; 1]`
		`>C = B'`
		`>SysOutSS = ss(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 = ss(A, B, C, D)`
		`>[A, B, C, D, Ts] = ssinfo(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] = pade(delay, order)` `>[A, B, C, D] = pade(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 = tf(num, den)`
		`>bode(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] = bodemag(SysIn)`
		`>[mag, wout] = bodemag(SysIn, [wmin wmax])`
		`>[mag, wout] = bodemag(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 = tf(num, den)`
		`margin(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] = margins(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**".

Appendix B: Mathematics characters

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	$\sim$
\beta	β	\phi	φ	\leq	$\leqslant$
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	$\clubsuit$
\epsilon	ϵ	\omega	ω	\diamondsuit	$\diamondsuit$
\zeta	ζ	\Gamma	Γ	\heartsuit	$\heartsuit$
\eta	η	\Delta	Δ	\spadesuit	$\spadesuit$
\theta	θ	\Theta	Θ	\leftrightarrow	$\leftrightarrow$
\vartheta	ϑ	\Lambda	Λ	\leftarrow	$\leftarrow$
\iota	ι	\Xi	Θ	\uparrow	$\uparrow$
\kappa	κ	\Pi	Π	\rightarrow	$\rightarrow$
\lambda	λ	\Sigma	Σ	\downarrow	$\downarrow$
\mu	μ	\Upsilon	Υ	\circ	$\circ$
\nu	ν	\Phi	Φ	\pm	$\pm$
\xi	ξ	\Psi	Ψ	\geq	$\geqslant$
\pi	π	\Omega	Ω	\propto	$\propto$
\rho	ρ	\forall	$\forall$	\partial	∂
\sigma	σ	\exist	$\exists$	\bullet	$\bullet$
\varsigma	ς	\ni	$\ni$	\div	$\div$
\tau	τ	\cong	$\cong$	\neq	$\neq$
\equiv	$\equiv$	\approx	$\approx$	\aleph	$\aleph$
\Im	$\Im$	\Re	$\Re$	\wp	$\wp$
\otimes	$\otimes$	\oplus	$\oplus$	\oslash	$\oslash$
\cap	$\cap$	\cup	$\cup$	\supseteq	$\supseteq$
\supset	$\supset$	\subseteq	$\subseteq$	\subset	$\subset$
\int	$\int$	\in	$\in$	\o	o

Hans-Petter Halvorsen

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

Blog: http://www.halvorsen.blog

University of South-Eastern Norway

www.usn.no

Simulink and Advanced Topics in MATLAB