The University Of Sheffield.

# ACS6501 MATLAB and Simulink

## Introductory guide and selected exercises

## Overview

### Week 1

- Lab Session 1 – Getting started
    - The IDE, help sources, using MATLAB interactively
    - Starting to write MATLAB programs
        - Variables
        - Arrays and Matrices

### Week 2

- Lab Session 2 – MATLAB programming continued
    - Branching code: if-else-end
    - Repeating code: loops
    - Writing your own functions
- Lab Session 3 – Simulink
    - Getting started with Simulink
    - Passing data between MATLAB and Simulink
    - Modelling ODEs

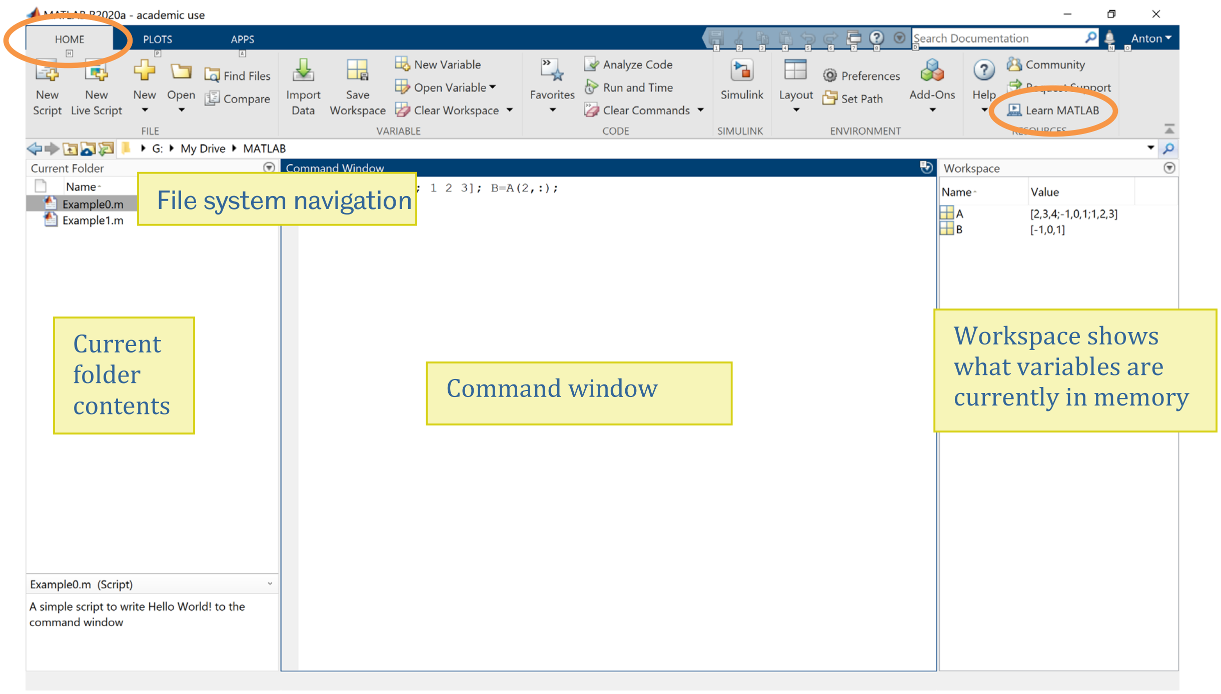
### Week 3

- Lab session 4 – Further MATLAB
    - More use of matrices in MATLAB and Simulink
    - Using MATLAB efficiently
    - Further plotting

# LAB SESSION 1

## 1.1 MATLAB preliminaries

Open MATLAB and have a quick look at the HOME tab menu.



```
; 1 2 3]; B=A(2,:);
```

| Name | Value |
|---|---|
| A | [2,3,4;-1,0,1;1,2,3] |
| B | [-1,0,1] |

Notice the "Learn MATLAB" resource (circled). If you click this, you will be taken to Mathworks' MATLAB academy web site. The free 2-hour MATLAB Onramp and 3-hour Simulink Onramp tutorials are good quick introductions.

### 1.1.1 MATLAB help

Task

a. At the command window prompt `>>` type `help log`

b. Click the button next to the MATLAB `>>` prompt, and you should see a function search dialog. For example, enter `sin` in its search bar. You can also search for help via the help menu. There is extensive help and there are plenty of coding examples.

### 1.1.2 Docking and undocking windows

You can arrange your MATLAB desktop the way you like it by docking or undocking windows. (Look for the little down arrow/triangle near the top right of the window.)

Task

Try this: Undock the command window, then dock the command window again.

You can dock/undock other windows – command history, workspace, etc. You can also drag windows around to change the MATLAB's desktop layout. More simply, you can use the Layout menu and use it to get back to the default set up. Click it to see the options available.

## 1.2 Basic Arithmetic

You can use MATLAB as a fancy calculator. For example, try these exercises:

Task

a. Use MATLAB to calculate $3 + 4 \times 5$ and $(3 + 4) \times 5$.

b. Using `^` to raise to a power, compute $\ln\left(\frac{0.4^2}{1+0.4^2}\right)$.

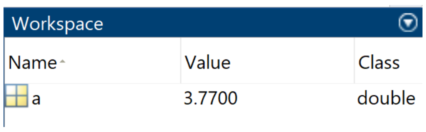
## 1.3 Variables

In MATLAB, the name of a variable (its identifier) can consist only of letters, underscores, and numbers. The first character must be a letter. Note that MATLAB is *case sensitive*.

### 1.3.1 Assignment

Task

a. Type at the command line `>> a=3.77`



| Name | Value | Class |
|---|---|---|
| a | 3.7700 | double |

*Explanation:* you have created a variable called `a` and assigned the value `3.77` to it. Notice it in the workspace. Note that the equal sign `=` is the "assignment operator". The assignment operator sets the value on the left hand side (LHS) to the value of the item on the right hand side. It is not a mathematics equals, it is an assignment operation: `a=b` is not the same thing as `b=a`.

Task

b. Type at the command line `>> a^2`
*Explanation:* You are computing `a` squared. The example also shows that if a LHS variable is not specified, the answer to the calculation is stored by MATLAB in a variable called `ans`.

Task

c. Type at the command line `>> a=1.1`
*Explanation:* you have overwritten `a` with a new value.

Task

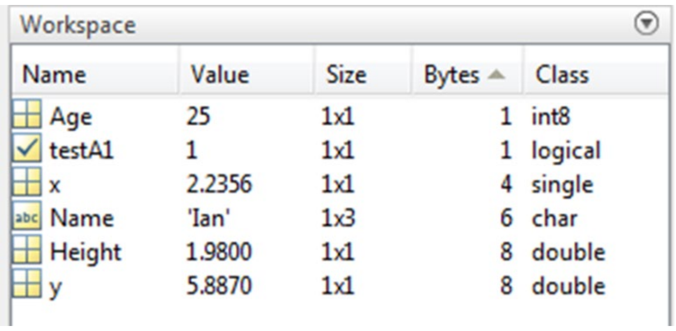
d. Type at the command line `>> sin(pi/2)`
Note: MATLAB has some pre-defined variables, e.g., common ones are

| | |
|---|---|
| `pi` | for $\pi$ |
| `i` and `j` | for $\sqrt{-1}$ in complex numbers |
| Inf | for $\infty$ |
| `NaN` | Not-a-Number (for undefined results, e.g., $0 * \infty$ or $0/0$) |
| `ans` | result of the last calculation with no variable assigned |

Task

e. Type at the command line `>> x=2`
Note the workspace and see the data type (look at the column called Class) for `x` is not an integer. MATLAB uses floating point numbers unless you specify otherwise. You can create other data types, e.g.,

```
>> Name='Ian';
>> Height=1.98;
>> Age=int8(25);
>> x=single(2.2356);
>> y=5.887;
```

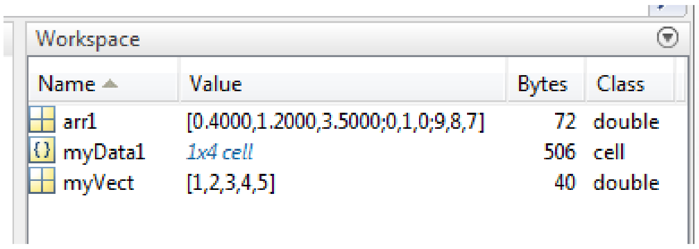


| Name | Value | Size | Bytes | Class |
|---|---|---|---|---|
| Age | 25 | 1x1 | 1 | int8 |
| testA1 | 1 | 1x1 | 1 | logical |
| x | 2.2356 | 1x1 | 4 | single |
| Name | 'Ian' | 1x3 | 6 | char |
| Height | 1.9800 | 1x1 | 8 | double |
| y | 5.8870 | 1x1 | 8 | double |

```
>> testA1=true;
```

You can also create Structured data types, e.g.,

```
>> myVect=[1,2,3,4,5];
>> arr1=[0.4,1.2,3.5;0,1,0;9,8,7];
>> myData1={'Peter',21,[50.1 63.2 66.1 59.0],'pass'};
```



| Name | Value | Bytes | Class |
|---|---|---|---|
| arr1 | [0.4000,1.2000,3.5000;0,1,0;9,8,7] | 72 | double |
| myData1 | *1x4 cell* | 506 | cell |
| myVect | [1,2,3,4,5] | 40 | double |

Task

f. Try to create variables called `2nd_var`, `my-var` and `var.3` and assign the value `1` to them. What happens? Why?

### 1.3.2 Deleting variables

Task

a. Enter at the command line: `>> clc`
It clears the command window, does not delete variables in the workspace

Task

b. Enter at the command line: `>> aa=1.2; bb=4.7; cc=5.98;`
Look at the workspace.
Enter at the command line: `>> clear bb` (Look again at the workspace).
If you want to erase all your variables, enter the command `clear`. You can also delete variables by highlighting them in the workspace and hitting the delete key.

### 1.3.3 Symbolic variables

It is possible to use MATLAB for analytical mathematics using *symbolic variables.* Some of MATLAB's functions are designed especially for symbolic mathematics. As you explore MATLAB's Help, you may see that some functions require symbolic variables. MATLAB symbolic variables are declared with the `syms` command. You may come across this in some examples but we won't be looking at the Symbolic Math Toolbox.

## 1.4 Saving and Loading data

You can save the workspace via the IDE menu save button or Right-click on the workspace or do it by typing `save` at the command line.

Save Workspace

You can load data from a data file using the Import Data button or by using `load` at the command line or from within MATLAB programs.

Import Data

Saving just some selected variables can also be done with `save` and `load` but more easily, you can save your data by right-clicking items in the workspace window and using the pop-up 'save as'.

Task

Save the workspace to a file called "myWorkspace.mat" with the command `save('myWorkspace.mat')`
Clear the workspace. Load the workspace again.

## 1.5 A first look at plotting

Task

MATLAB has the capability to plot numerical data. For example, enter these three command lines in the command window (exactly, including the dots and semicolons) and press the return key after each line:

```
>> x=1:100;
>> y=x.^2;
>> plot(x,y)
```

*Explanation:* with these commands you created a variable `x` (a one dimensional array of numbers from 1 to 100) and a variable `y` (a one dimensional array of numbers that are the squares of the values in `x`). You then plotted `y` versus `x`. We'll do more with arrays later.

## 1.6 MATLAB Scripts (writing your own programs)

Commands issued at the command line can be put into a text file but must be saved with extension ".m" for MATLAB to recognise it as a "script m-file".

Lines in the script m-file are executed sequentially. The workspace used by the script is the same as for command line you have been using so far.

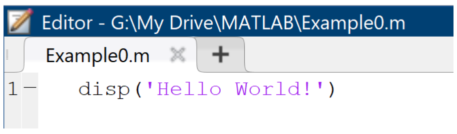
You can start a new script by clicking the New Script:

### 1.6.1 Editing a first script: Hello World

We will create a program that displays the string "Hello world!" on execution. We will use the `disp` function.

Task

Open a new script and in the editor type the line:


Editor - G:\My Drive\MATLAB\Example0.m

Example0.m

```
disp('Hello World!')
```

That's it! A very short program. Save the program with a suitable name by clicking on the Save button. You can run your program from the menu Run button or by typing the program name from the command line.

It is good practice to add comments to your code. Comment lines begin with %

### 1.6.2 Requesting user input and displaying output to the command window

You can use `input` for user input from the keyboard to assign to a variable

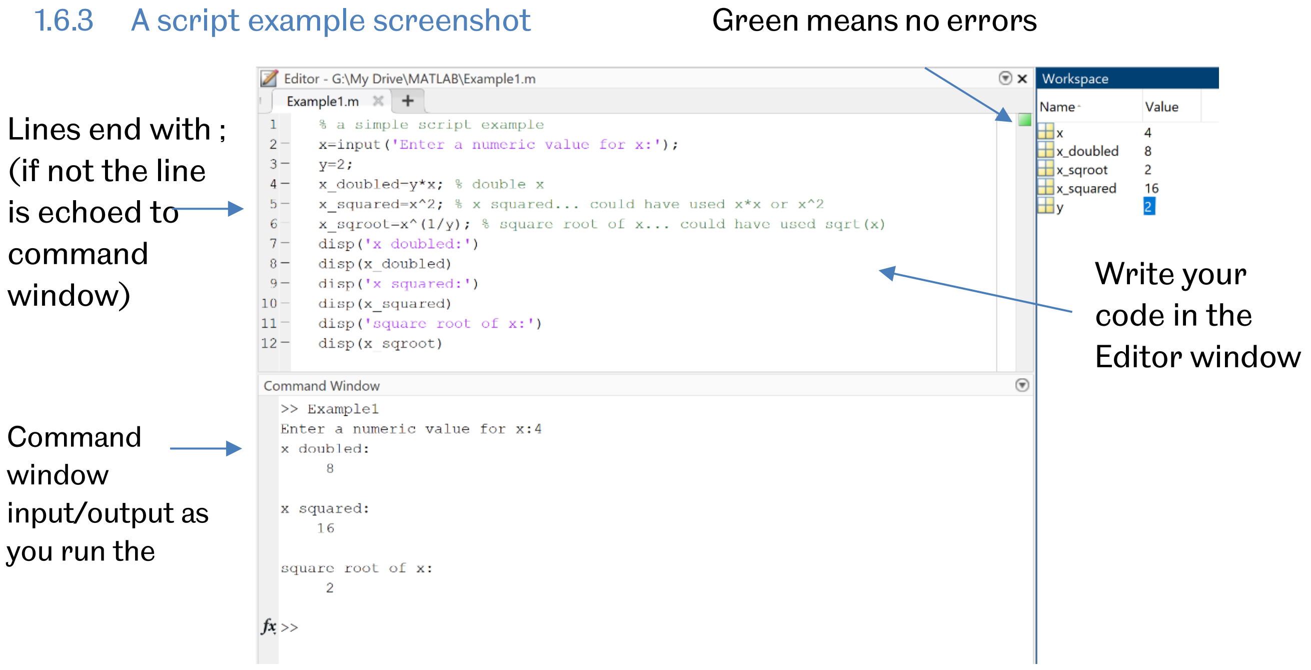
```
a=input('Enter radius in m:');
```

You can use `disp` to display values in the command window

```
array=[1.233 4.234 123.0];
```

```
disp('First Second Third')
disp(array)
```

### 1.6.3 A script example screenshot


Green means no errors
Lines end with ; (if not the line is echoed to command window)
Write your code in the Editor window
Command window input/output as you run the
Editor - G:\My Drive\MATLAB\Example1.m
Example1.m
Workspace
Command Window

```
% a simple script example
x=input('Enter a numeric value for x:');
y=2;
x_doubled=y*x; % double x
x_squared=x^2; % x squared... could have used x*x or x^2
x_sqroot=x^(1/y); % square root of x... could have used sqrt(x)
disp('x doubled:')
disp(x_doubled)
disp('x squared:')
disp(x_squared)
disp('square root of x:')
disp(x_sqroot)
```

| Name | Value |
|---|---|
| x | 4 |
| x_doubled | 8 |
| x_sqroot | 2 |
| x_squared | 16 |
| y | 2 |

```
>> Example1
Enter a numeric value for x:4
x doubled:
     8

x squared:
    16

square root of x:
     2

fx >>
```

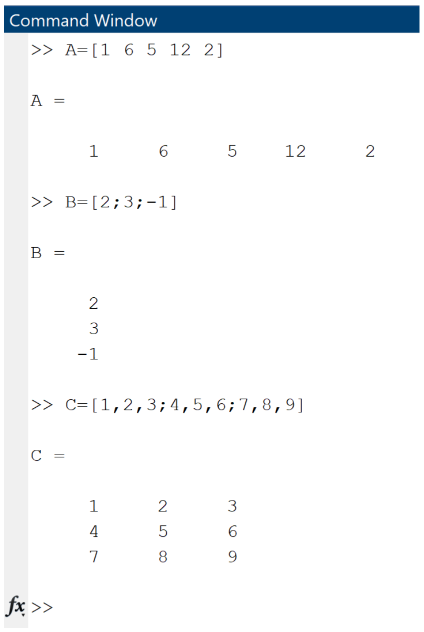
### 1.6.4 A script exercise: A simulator for throwing two six-sided dice

Task

Create a dice throwing simulator: a MATLAB script that, when it is run, outputs the scores for two dice (random integers between 1 and 6) and the sum of the scores. Use `rand` to create a (uniformly distributed) random number between zero and one. In your script you can multiply the random numbers by six and round them up using the `ceil` function. Add comments at the top of your script describing what the script does, the author and date. Save your script. Example solution: “sumTwoDice.m”

## 1.7 Arrays and Matrices

MATLAB arrays can be 1D, 2D, 3D, and of higher dimensions. To create or assign an array in MATLAB you use square brackets; you use spaces or commas to separate column elements and semicolons to separate rows. E.g.,



```
>> A=[1 6 5 12 2]

A =

     1     6     5    12     2

>> B=[2;3;-1]

B =

     2
     3
    -1

>> C=[1,2,3;4,5,6;7,8,9]

C =

     1     2     3
     4     5     6
     7     8     9

fx >>
```

a 1D array example

a 1D column array example

a 2D array example

For array indexing we use round brackets. Indexing starts at 1. For 2D arrays the order is *row,column*, e.g.,

```
>> A=[1 6 4 3 12 13 2]; % 1D array with 7 elements
>> A(1) % element at index position 1
ans = 1
>> x=A(5) % element at index position 5
x = 12
```

Using the 2D array called `arr1`, created earlier in 1.3.1e,

```
>> arr1(1,3) % this is the element at row 1, col 3
ans = 3.5000
```

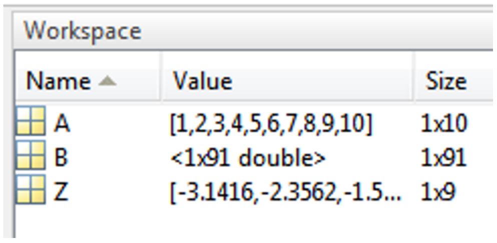
## 1.7.1 Using the colon as a quick way of creating arrays

MATLAB syntax allows use of the **colon** for more efficient matrix assignment.

- Assignment/Creating arrays

```
>> A=1:10;          % A(1)=1     A(2)=2 … A(10)=10
>> B=1:0.1:10;      % B(1)=1     B(2)=1.1 … B(91)=10
>> Z=-pi:pi/4:pi;   % X(1)=-pi   X(2)=-3/4 pi … X(9)=pi
```

*from* : *stepsize* : *to*
(step is 1 if stepsize omitted)



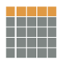
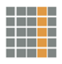
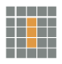
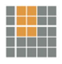
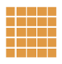
| Name ▲ | Value | Size |
|---|---|---|
| A | [1,2,3,4,5,6,7,8,9,10] | 1x10 |
| B | <1x91 double> | 1x91 |
| Z | [-3.1416,-2.3562,-1.5... | 1x9 |

### 1.7.2 Using the colon for array indexing

MATLAB allows use of the colon for more efficient matrix indexing .

*row , col*

Pictorial example... **A** is 5X5 array

| | | |
|---|---|---|
| • Select first row , all columns | `A(1,:)` |  |
| • Select all rows , 4th column | `A(:,4)` |  |
| • Select 2nd to 4th row , col.3 | `A(2:4,3)` |  |
| • Select rows 1 to 3 , cols 2 and 3 | `A(1:3,2:3)` |  |
| • Entire matrix | `A(:,:)` |  |

### 1.7.3 Array exercises

Task

a. Create the arrays

```
x=[0,1,2,3,4,5,6]; y1=[0,1,2,3,4,5,6]; y2=[0,1,4,9,16,25,36];
y3=[0,1,8,27,64,125,216];
```

b. Plot `y1` against `x`, `y2` against `x`, and `y3` against `x` all on the same graph (use the command `hold on` to hold the axes to receive further plots), and make use of different colours and line styles, e. g,

```
>> plot(x,y1,'-bo') % blue line with data point circles
```

Type `hold off` to end plotting to the current figure.

c. MATLAB has some useful array creation functions. Enter these commands:

```
>> eye(4)
>> ones(3)
>> zeros(2,3)
>> rand(5,2)
```

d. Create some matrices using the colon. Try these lines:

```
>> G=1:10
>> H=[1:10;1:10:100]
>> myD=[-2*pi:pi/4:2*pi]
>> plot(myD,sin(myD))
>> grid on
```

e. You can transpose an array with the operator ′
For example, enter the command `>> G′`

f. You can find out the size of an array with `size`, e.g., try `>> size(H)`
At command prompt enter

```
>> A=[2:4; -1:1; 1 2 3];B=A(2,:);size(B')
```

### 1.7.4 Array and Matrix Operations

Task

Enter the following from the command line:

```
>> A=[1,2;3,4]

   A =
      1     2
      3     4

>> B=[5,6;7,8]

   B =
      5     6
      7     8
```

and now enter:

| | | |
|---|---|---|
| (i) | `>> A+B` | |
| (ii) | `>> 2*A` | |
| (iii) | `>> A.*B` | |
| (iv) | `>> A.^2` | |
| (v) | `>> A*B` | (compare the answer to `A.*B`) |
| (vi) | `>> A^2` | (compare the answer to `A.^2`) |

*Explanation:* The result for (i) is as you might expect. With (ii) you are multiplying by a scalar value 2, and the result is as you would expect. However, with (iii), (iv), (v), and (vi) we are distinguishing between element-by-element operations and **matrix** mathematics. `A*B` is a matrix mathematics operation whereas `A.*B` is an element-by-element array operation.

For further help on “Array vs Matrix operations” and a table summarizing array operations see MATLAB Help.

# LAB SESSION 2

## 2.1 Program flow control

In your MATLAB programs, there will be situations where you need to choose between some different code depending on some condition, and, situations where you will want to repeat some code depending on some condition. You use `if` or `switch` to choose different code and to repeat code you use `while` or `for` loops.

### 2.1.1 Relational operators

You use relational and logical operators to create conditional expressions:

| | |
|---|---|
| `a == b` | True if a is equal to b. |
| `a ~= b` | True if a not equal to b. |
| `a > b` | True if a is greater than b. |
| `a >= b` | True if a is greater than or equal to b. |
| `a < b` | True if a is less than b. |
| `a <= b` | True if a is less than or equal to b. |
| `a & b` | logical AND, true only if a and b are both true |
| `a | b` | logical OR, true if any of a or b are true |
| `~a` | logical NOT, (in this example result is true if a is false) |

Logical AND is && in 'C'

### 2.1.2 `if` statement

If statements can be structured in any of these ways:

```
if condition
    statements;
end
```

```
if condition
    statements;
else
    statements;
end
```

```
if condition
    statements;
elseif condition2
    statements;
else
    statements;
end
```

Here's an example script using two `if` statements, one nested inside the other:

```
% if_else_example.m
A=input('Enter a value for A: ');
if A < 0
    disp(' A is less than zero ')
    disp(' setting the value to 0 ')
    A = 0;
else
    if A > 100
        disp(' A is greater than 100 ' )
        disp(' setting the value to 100 ')
        A = 100;
    else
        % do nothing
    end
end
```

Task

Run the code above and satisfy for yourself you understand how it works and then compare the code below, which does the same thing, but uses the if-elseif-else-end structure.

```
% if_elseif_example.m
A=input('Enter a value for A: ');
if A < 0
    disp(' A is less than zero ')
    disp(' setting the value to 0 ')
    A = 0;
elseif A > 100
    disp(' A is greater than 100 ' )
    disp(' setting the value to 100 ')
    A = 100;
else
    % do nothing
end
```

Previously, you wrote a program that simulates throwing dice. In some games, the total score will be doubled if the score of the first dice equals the score of the second dice. Alter your program so that the total score is doubled if both dice show the same score. Use `disp` to show the score and also output a message to let the user know. Compare your solution with “sumTwoDice.m”.

Optional

Here is a script that uses nested if statements to display the class of a degree according to a grade input.

```
% degreeclass.m
grade=input('Enter grade (integer value between 0 and 100): ');
if grade > 70
    disp('1st')
else
    if grade > 60
        disp('2:1')
    else
        if grade > 50
            disp('2:2')
        else
            if grade > 40
                disp('3rd')
            else
                disp('Fail')
            end
        end
    end
end
```

Recode the script so that it uses one if-elseif-elseif...end statement. Compare your solution with “degreeclass_elseif.m”. Note that code produced can sometimes be clearer when elseif’s (or `switch`) statements are used to simplify nested ifs.

### 2.1.3 `for` loop

A `for` loop is a loop that is executed *a specified number of times.* Syntax is like this:

```
for loopcounter = startvalue : step : endvalue
   % enter code here to do something for
   % each value of loopcounter
```

```
end
```

More generally:

```
for loopcounter = integerArray
   % do something for
   % each value of loopcounter
end
```

Note: If *step* is omitted the step size defaults to 1.

Examples:

```
% for loop
for k=1:100
    x(k)=pi*k/50;
    A(k)=sin(x(k));
end

% for loop with step size 5
for k=0:5:20
    disp(k)
end

% for loop, counting down
for k=10:-1:1
    disp(k)
end
```

Task

Write a program, using a `for` loop, that writes the numbers from 1 to 10 to an array `x`, and the squares of the numbers to an array `y`. Plot `y` versus `x`. Compare your solution with "loop_xsq_y.m".

Optional

A more challenging question: write a script, using a loop, to add all the natural numbers below one thousand that are multiples of 3 or 5, then, if you have used MATLAB before, see if you can remove the loop and do it in two lines. Compare your solution with "sum_natural_mults_of_3_and_5.m"

**Calculate $\pi$ by a Monte Carlo method:**
The area of a circle of radius $r = 1$ is $\pi$. To compute the area, we can create $N$ random points, where every point $N$ has a random $x$ coordinate and a random $y$ coordinate. We only consider random points where all the coordinates are in the range of $[-1, 1]$, i.e., a $2 \times 2$ square. A point is located inside the circle if $x^2 + y^2 < 1$.

To compute $\pi$, it is sufficient to multiply the area of the square with the number $N_k$ of points inside the circle divided by the overall number $N$ of points. To improve the precision, increase the number of points. Try $N = 1$, $N = 10$, $N = 100$, $N = 1{,}000$, $N = 10{,}000$, $N = 100{,}000$, $N = 1{,}000{,}000$. ("Pi_calc_by_area_of_circle.m.m")

### 2.1.4 `while` loop

`while` loops are used to loop through code until a specified condition is met. They are used in cases where the *number of loops is unknown.* Syntax:

```
while condition
```

```
    % do something
    % while the condition applies
end
```

Example m-file script using while loop to output numbers 1 to 10:

```
% while loop example
x=0;           % initialisation
while x<10     % execute the following command(s) while x<10
   x=x+1;    % increment value of x
   disp(x)     % display x
end
```

Optional

MATLAB provides a function to compute factorials. Write your own program, using a while loop, to compute the factorial of $n$ by multiplying each integer from 1 to $n$ with the previous product. Example solution: "factorial_n_while.m".

### 2.1.5 Nested loops

Optional

Create a multiplication table for the products of all integers from 1 to 10. Use a two-dimensional array called `table(m,n)`, where `table(m,n)=m*n`. Use nested `for` loops over `m` and `n` to create the table. Example solution: "loop_multiplication_table.m"

## 2.2 Program layout

Especially important when using if statements and loops:

- Structure your programs
- Use indentation

Here is an example of bad layout ("bad_layout.m"):

```
nrows=4;ncols=4;A=ones(nrows,ncols);
for c=1:ncols,for r=1:nrows
if r==c,A(r,c)=2;elseif abs(r-c)==1,A(r,c)=-1;
else A(r,c)=0;end,end,end,A
```

Run it to see what it does. Below is the same code with indentation and comments. Look how much easier it is to read!

```
% good_layout.m
% Create a matrix on 1s
nrows=4;
ncols=4;
A=ones(nrows,ncols);

% Loop through the matrix, assign 2 on the main diagonal,
% -1 on the adjacent diagonals, and 0 everywhere else
for c=1:ncols
    for r=1:nrows
        if r==c
            A(r,c)=2;
        elseif abs(r-c)==1
            A(r,c)=-1;
        else
            A(r,c)=0;
        end
    end
end
disp(A)
```

## 2.3 Functions

Plain text files with extension “.m“ are interpreted by MATLAB as either Script or Function m-files. They are treated as functions if the keyword `function` is found at the start of first line. A script can be run from command line by invoking its name whereas functions must be called. Functions, unlike scripts, can use input/output arguments. Variables used by scripts are part of main workspace whereas functions have their own workspace.

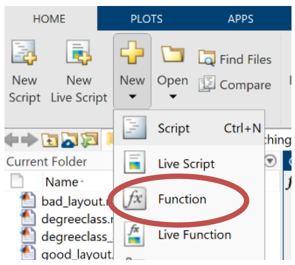
### 2.3.1 Function syntax

You define a function like this:

```
function [outputArgs] = funcName(inputArgs)
% function help comment line
% your code goes here
end
```

Notes:

- first line of a function m-file should be the function definition line.
- second line is known as the H1 line *if it is a comment* and is then treated by MATLAB as the start of the function’s help
- use square brackets [ ] for outputArgs
- outputArgs is a comma-separated list of identifiers
- [ ] brackets are optional for only one output
- functions with no output are allowed, e.g., `pause(n)`
- use round brackets ( ) for inputArgs
- inputArgs is a comma-separated list of identifiers
- functions with no input are allowed, e.g., `cputime`
- Order of variables used to pass data is important!
- Calling `plot(x,y)` is not the same as `plot(y,x)`



### 2.3.2 Creating a new function

If you create a new function from the MATLAB “New” menu you get a skeleton template created in the Edit window:

```
function [outputArg1,outputArg2] = untitled2(inputArg1,inputArg2)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

Here’s an example function to calculate the volume of a Sphere:

```
function vol=SphereVol(radius)
% SphereVol calculates the volume of a sphere
% Input arguments: radius
% Return value: vol

vol=4/3*pi*radius^3;  % volume calculation

end
```

You can call SphereVol from within scripts or from the command line:

```
>> SphereVol(1)
ans = 4.1888
```

Note that adding comments immediately after the function definition line are used by MATLAB's help command:

```
>> help SphereVol
  SphereVol calculates the volume of a sphere
  Input arguments: radius
  Return value: vol
```

### 2.3.3 Exercises

Task

(i) Create a new function called SphereVM that computes and returns the volume and mass `[vol,mass]` of a sphere given inputs `radius` and `density`.

Optional

(ii) Write functions mySum2, mySum4 and myOperations2. Functions with "2" in their name will accept two arguments, the function mySum4 will take four arguments. The functions will return:

- mySum2: The sum of the two arguments.
- mySum4: The sum of the four arguments.
- myOperations2: The sum, difference, product, and ratio of the arguments.

Test your functions.

### 2.3.4 Anonymous functions

In addition to creating functions as we have done above, MATLAB also provides an 'anonymous function' syntax which allows you to create functions without needing a separate function m-file. For example, try these commands:

Task

a. At the command prompt, enter `f = @(x) sin(3*pi*x).*x.^3`
MATLAB creates a function handle `f`. You can now use this function handle:

```
>> f(0.75)             (evaluate f at 0.75)
>> fplot(f,[-5,5])     (plot f over the domain [−5,5])
>> fzero(f,2.3)        (find zero near the initial guess 2.3)
```

Optional

b. At the command prompt, here we define a function with two arguments:

```
>> g = @(x,y)(x.^4+y.^4)-4*x.^2.*y.^2
```

Evaluate $g$ at $x = 0.75$, $y = 0.75$

```
>> g(0.75,0.75)
```

Plot $g$ over the domain $x \in [-2,2]$ and $y \in [-2,2]$.

```
>> fsurf(g,[-2,2,-2,2])
```

# LAB SESSION 3

## 3.1 Getting started with Simulink

Simulink is a graphical add-on to MATLAB for simulation, model based design and analysis dynamic systems.

### 3.1.1 Building a first model[1]

#### First steps

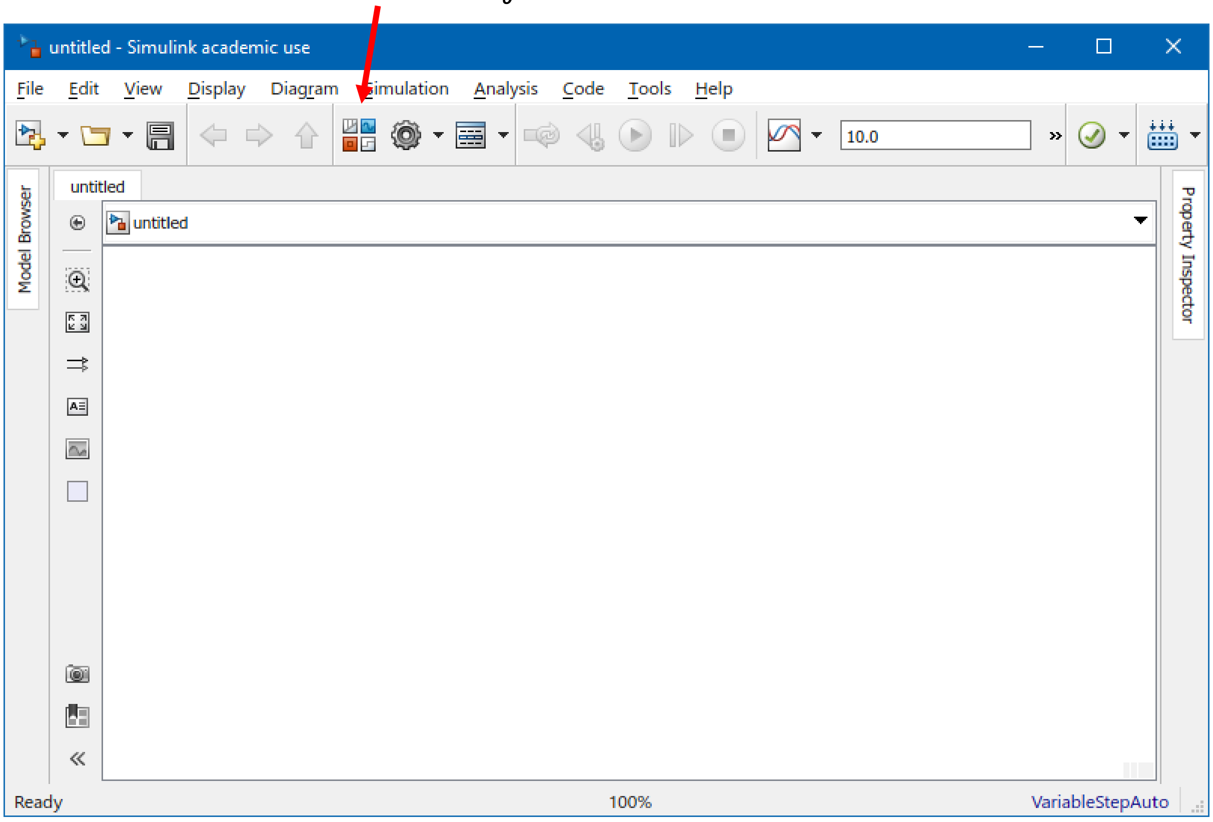
Task

Start Simulink by typing `simulink` at the MATLAB command prompt or by clicking the Home tab's Simulink icon.

Simulink

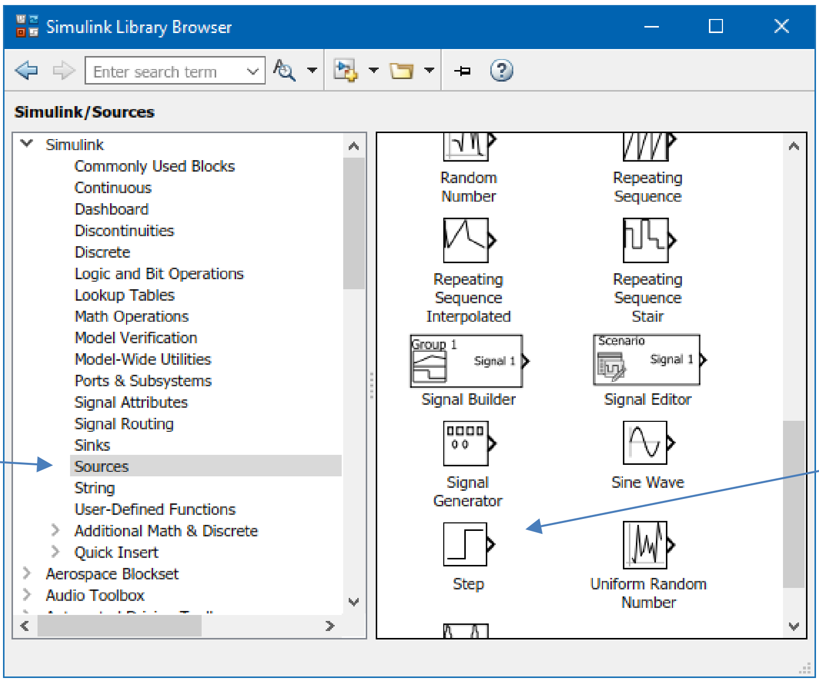
Create a blank Simulink Model by choosing '**Blank Model**' from the Simulink Start Page.

An 'untitled' blank model window pops up. To start to build your model, you can find blocks with the Simulink Library browser.



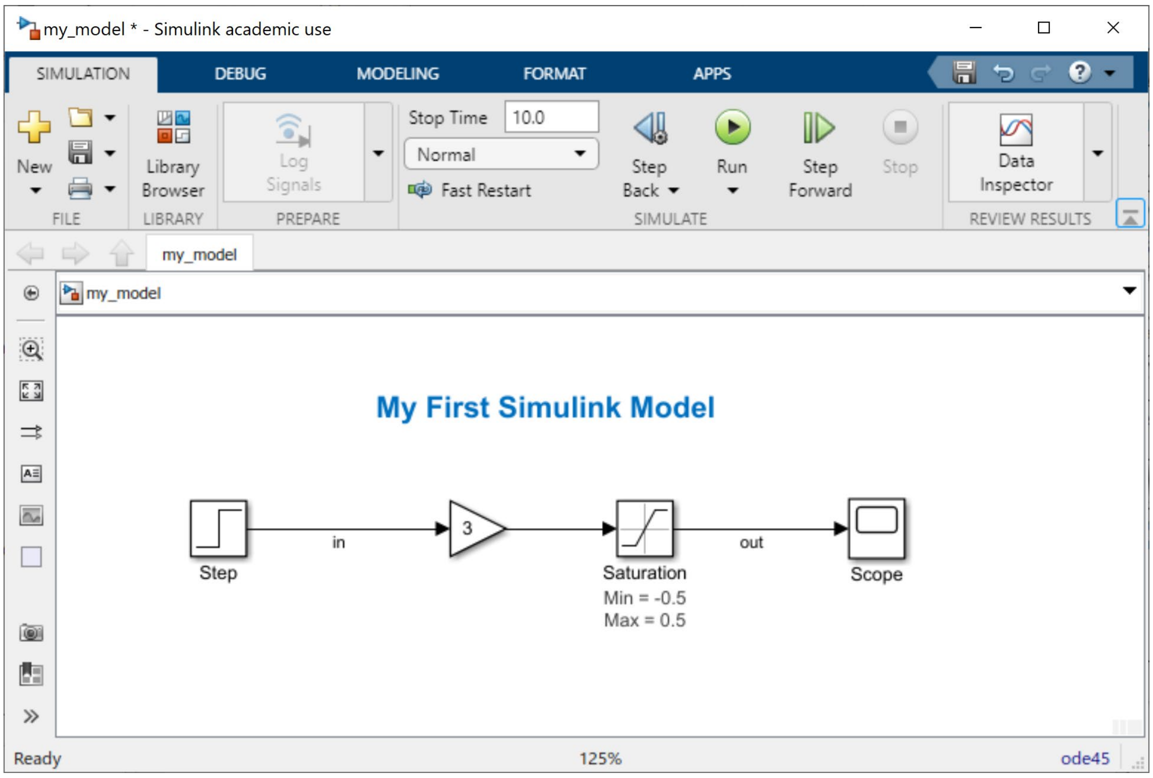
[1] Following "Getting Started with the Simulink Environment: Build and simulate a model" https://uk.mathworks.com/videos/getting-started-with-simulink-118723.html 02/11/2018

For example, the 'step' block is in the sources section of the library:



Task

Create the model below by locating the blocks with the Simulink Library browser and dragging them to your blank model. `Scope` is in `Sinks`, `Gain` is in `Math Operations`, `Saturation` is in `Commonly Used Blocks`.



You connect the blocks by dragging the mouse from the arrow-out shown on a block to the arrow-in shown on another. The block label will disappear. To show the block name, right-click the block, then select *Format* → *Show Block Name* → *On*. Save your model.

### Changing block parameters

Task

The `Step` block has a default step time of `1`. This is the time at which you want the step change to happen in the simulation. Double click the `step` block, and in the block's *parameters* dialog box change 'Step time' to something else, e.g., change it to 5. Click OK. The `Gain` block has a default of `1`. Change it to another value, e.g., 3. Click OK.

### Running the model

Run the simulation with and without the `Saturation`. Compare the results. To run the simulation, click on the green arrow run button. Double-click on the `Scope` to view the simulation result.

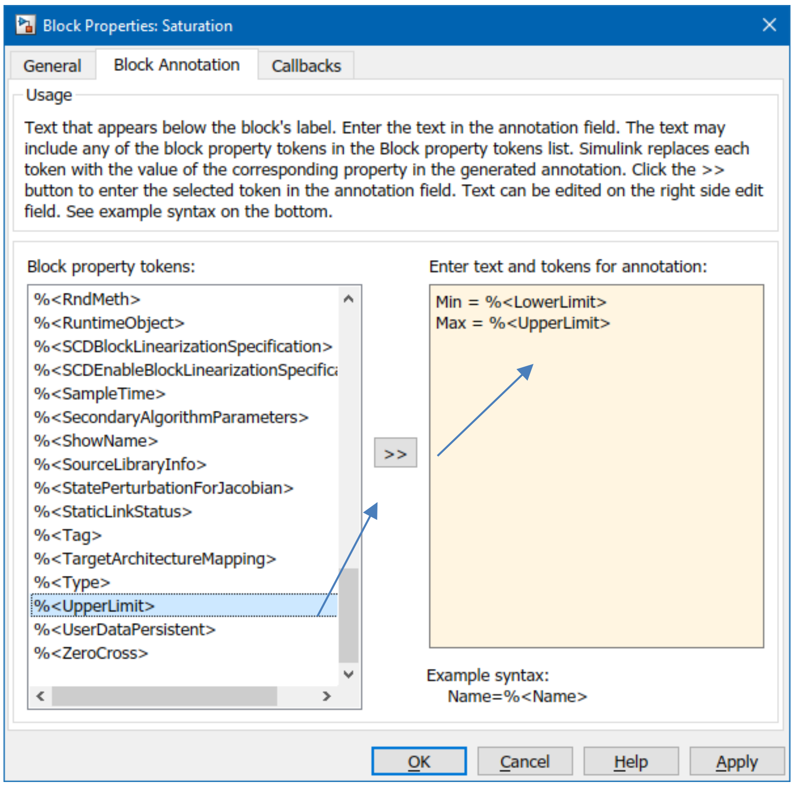
Remember that for Help about a block you can right-click it and select help.

### 3.1.2 Labelling your models (making them more readable)

Adding labels to well-positioned blocks and their connections can improve the readability of a model. This is especially true for big models with many blocks. To quickly rename a block, double-click on the *label* text and retype the label. To label a signal (i.e., a connecting line), double-click it and enter your label text. You can also use block properties to make annotations to models.

Task

Try this for the `Saturation` block, in properties choose the "Block Annotation" tab. Then type in the 'Min = ' and 'Max = ' text and select the LowerLimit and the UpperLimit annotations as shown below:



Text that appears below the block's label. Enter the text in the annotation field. The text may include any of the block property tokens in the Block property tokens list. Simulink replaces each token with the value of the corresponding property in the generated annotation. Click the >> button to enter the selected token in the annotation field. Text can be edited on the right side edit field. See example syntax on the bottom.

```
%<RndMeth>
%<RuntimeObject>
%<SCDBlockLinearizationSpecification>
%<SCDEnableBlockLinearizationSpecifica
%<SampleTime>
%<SecondaryAlgorithmParameters>
%<ShowName>
%<SourceLibraryInfo>
%<StatePerturbationForJacobian>
%<StaticLinkStatus>
%<Tag>
%<TargetArchitectureMapping>
%<Type>
%<UpperLimit>
%<UserDataPersistent>
%<ZeroCross>
```

```
Min = %<LowerLimit>
Max = %<UpperLimit>
```

Example syntax:
Name=%<Name>

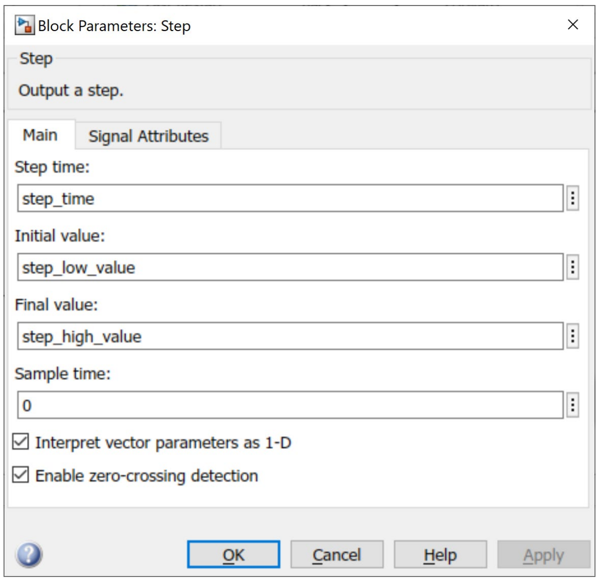
You can add text, e.g., model title, directly to your model. Simply double-click on the white background and type your text. You can format the text.

## 3.2 Getting data to and from MATLAB

### 3.2.1 Passing block parameter values from MATLAB

Task

As an example showing the use of MATLAB variables to determine block parameters, edit the properties of the `Step` block as shown below:



Output a step.

Step time: step_time

Initial value: step_low_value

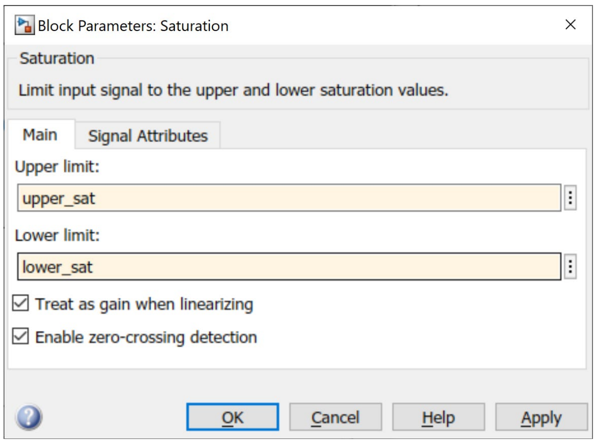
Final value: step_high_value

Sample time: 0

☑ Interpret vector parameters as 1-D

☑ Enable zero-crossing detection

Also edit the `Saturation` block as shown below:



Limit input signal to the upper and lower saturation values.

Upper limit: upper_sat

Lower limit: lower_sat

☑ Treat as gain when linearizing

☑ Enable zero-crossing detection

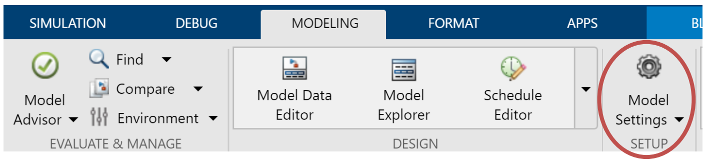
Then create script in MATLAB containing the lines ("run_my_model1.m"):

```
% Script to set sine amplitudes and run the simulink model "my_model1"
clear
step_time=input('Enter time step-change occurs (e.g. 5): ');
step_low_value=input('Enter step low value (e.g. 0): ');
step_high_value=input('Enter step high values (e.g. 1): ');
upper_sat=input('Enter upper saturation values (e.g. 0.5): ');
lower_sat=input('Enter lower saturation values (e.g. -0.5): ');
sim('my_model1')  % run the model
```

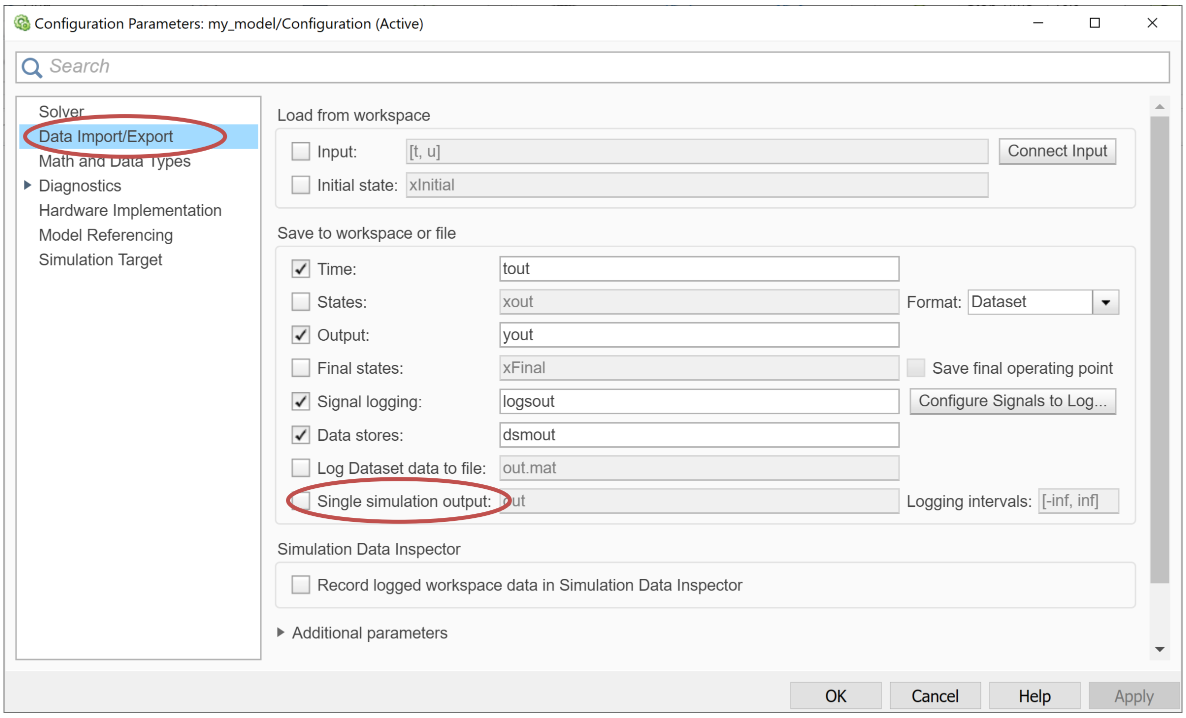
Save your model. Then run the script. Double-click on the `Scope` to view the simulation result. Experiment with a few different inputs to satisfy yourself the model is working as you expect.

## Configuring your model output to save data to MATLAB

**IMPORTANT: In the examples that follow, a Simulink configuration change has been made, via the "Model Setting" button in the "Modelling" menu.**



The "Single simulation output" box has been cleared. (*Beware that it is ticked by default when you start a new model on Managed Desktop MATLAB R2020a.*)



Load from workspace

| | | |
|---|---|---|
| ☐ Input: | [t, u] | Connect Input |
| ☐ Initial state: | xInitial | |

Save to workspace or file

| | | |
|---|---|---|
| ✓ Time: | tout | |
| ☐ States: | xout | Format: Dataset |
| ✓ Output: | yout | |
| ☐ Final states: | xFinal | ☐ Save final operating point |
| ✓ Signal logging: | logsout | Configure Signals to Log... |
| ✓ Data stores: | dsmout | |
| ☐ Log Dataset data to file: | out.mat | |
| ☐ Single simulation output: | out | Logging intervals: [-inf, inf] |

Simulation Data Inspector

☐ Record logged workspace data in Simulation Data Inspector

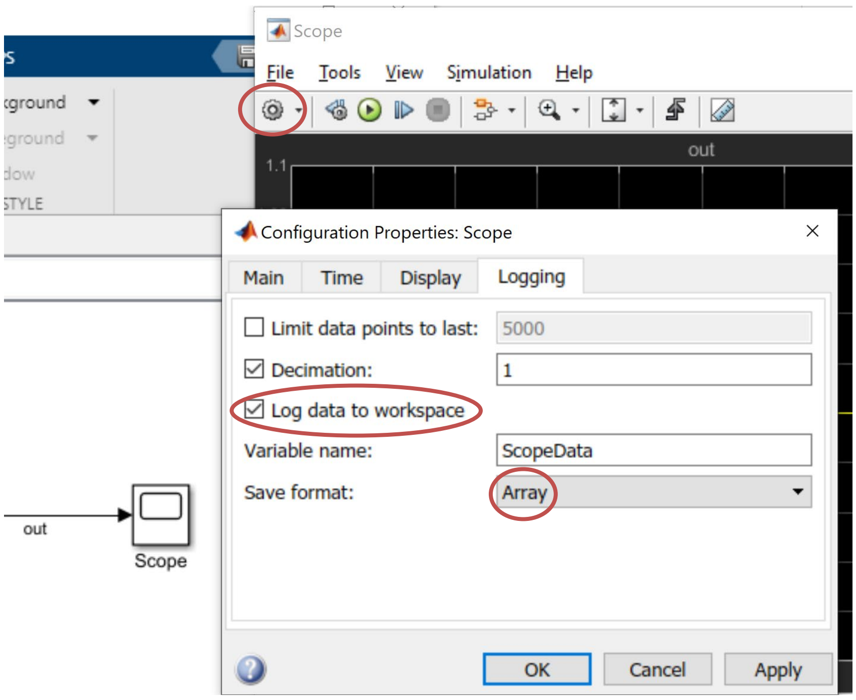
Additional parameters

You should make this change for each model.

### Logging the Scope data to the MATLAB workspace

Task

To do this, click the scope configure settings icon, tick Log data, and save in the format you want. Here we save as an array:


Scope
File Tools View Simulation Help
out
Scope
Configuration Properties: Scope
Main Time Display Logging
OK Cancel Apply

☐ Limit data points to last: 5000
☑ Decimation: 1
☑ Log data to workspace
Variable name: ScopeData
Save format: Array

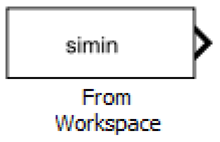
To make a MATLAB plot of the data you could add the following lines of code to the script that runs *my_model1*

```
plot(ScopeData(:,1),ScopeData(:,2))
xlabel('time')
ylabel('y')
title('Scope data log')
```

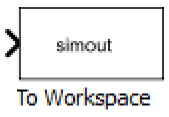
Optional

Create a model with two `Sine Wave` blocks sources. Set the amplitude of one to be 3 and the other to be 2. Use a `Product` block to multiply the signals and view the result on a `Scope`. Add a title and labels to your model. Run your model to see if it works. Then edit your model to accept variables in MATLAB's workspace to define the amplitudes of each initial sine wave. Run it again with few different inputs. Set your scope so that it opens on running the simulation. Example solution: "simple_sine_wave_product".

### 3.2.2 Simulink blocks to pass data to and from the workspace



The `simin` block can be used to read data values specified in timeseries, matrix, or structure format and pass them into your simulation model.



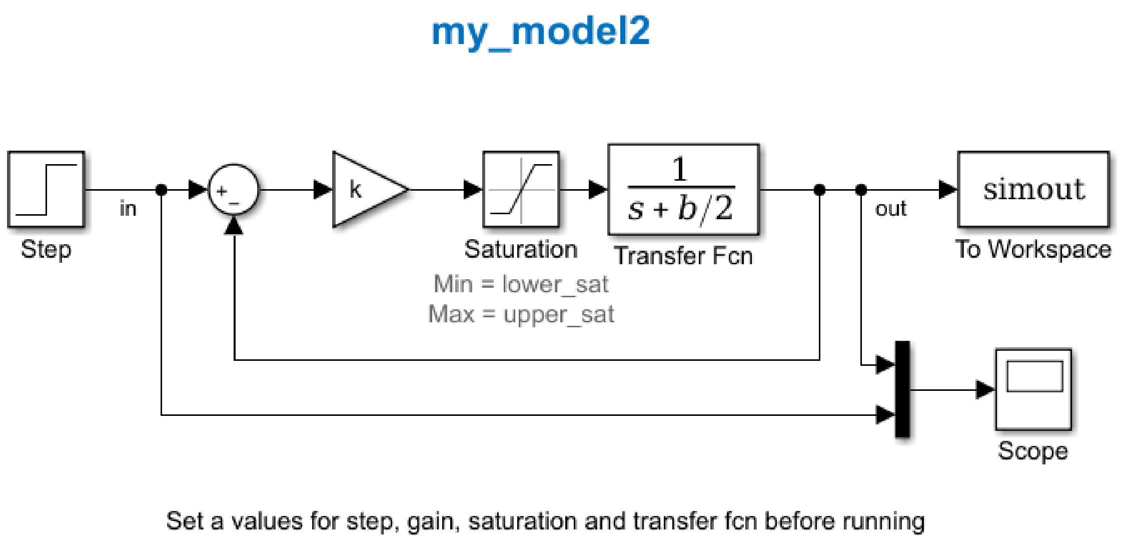
With the `simout` block you can output data to specified timeseries, array, or structure in MATLAB's workspace

Task

Now make a copy of your first model and edit it to it to produce the one shown below. Save it as '*my_model2*'. Note the use of a mux block to show two signals on

Task

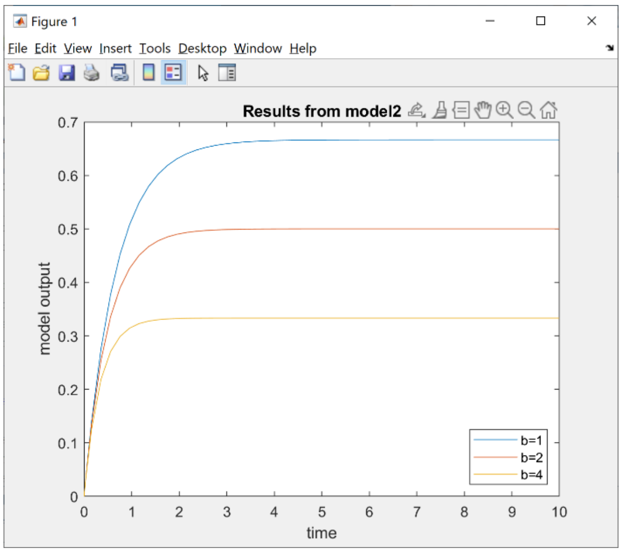
the scope. Alternatively, you can set the number of scope inputs to 2 on the scope block and then you could change the scope layout to show both signals on one plot or show them on separate subplots.



Create a MATLAB script to set values for the step, gain, saturation, and transfer function blocks, e.g., set `step_time=0; step_low_value=0; step_high_value=1; upper_sat=1; lower_sat=-1; k=1; b=1;`

Run your model and check the MATLAB workspace to see what the `simout` block creates. Make a plot of model output vs time.

Now edit your MATLAB script so that it runs the simulation and obtains the simout for three successive values for b (b = 1, 2, and 4) and plot them on one figure as shown opposite. Note that `plot` function can take a timeseries data type as an arg, i.e., you can `plot(simout).`



Next, try changing the input signal from the step block to one that uses `simin`. We could use data from a file or a variable. Let's create a timeseries data type variable. At the command line type:

```
>> myinputdata=timeseries([zeros(100,1);ones(500,1);zeros(400,1)],(linspace(0,10,1000))');
```

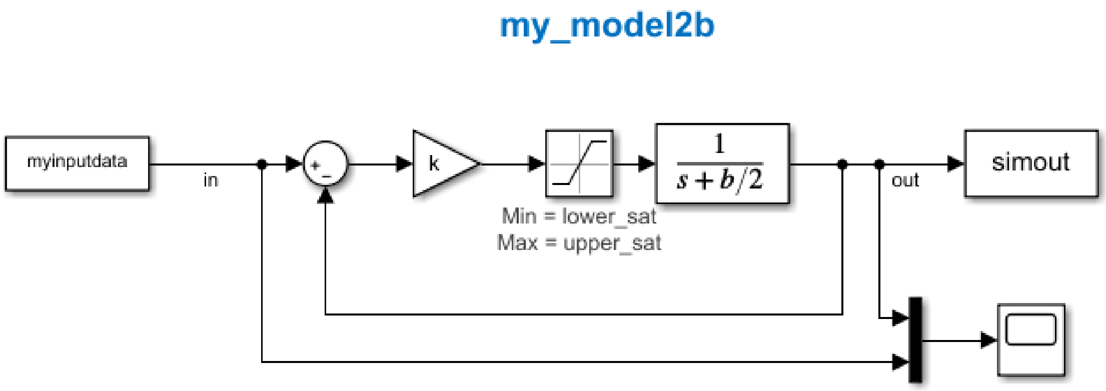
*Explanation:*

- `timeseries` is a function that creates a timeseries data type. It requires some data values and some time values. For the data values let's make a step as follows `zeros(100,1)` makes a column of 100 zeroes

- `ones(500,1)` makes a column of 500 ones
- `zeros(400,1)` makes a column of 400 zeroes we join them together in one array to get 1000 data values
- `linspace` is a function that creates a linearly spaced 1D array. We are using it for our time values. We choose a start of zero and end at 10, with 1000 values to match our number of data values. We transpose it with ' to get a column rather than a row.

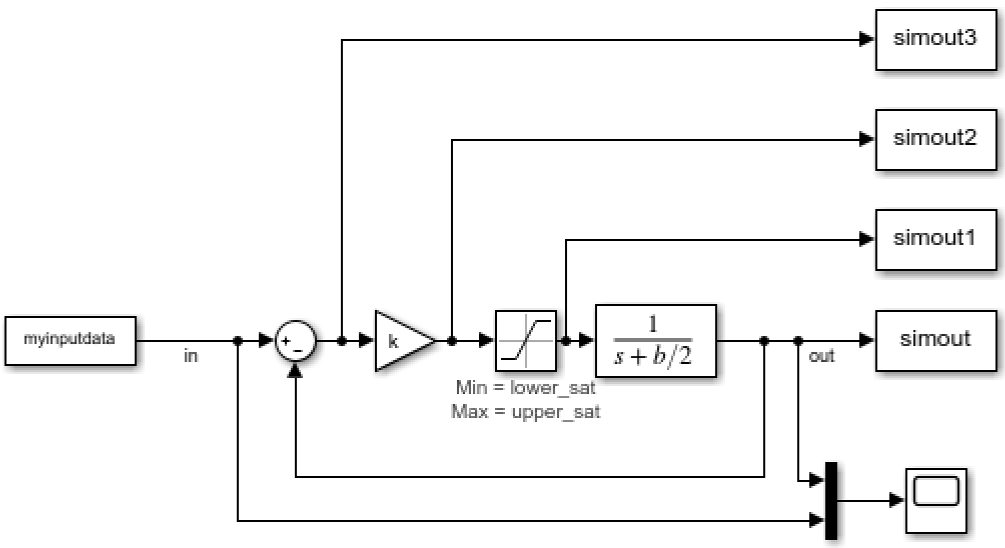
Task

Now, edit the model and save as my_model2b so that it is like the one below. Use `simin` as the input block. Then rename the `simin` to be `myinputdata` to use the variable you created above.



Run the Simulink model. You should see a response like the one you got for the step block input, and a decay after 6 seconds.

Next, you could add further outputs that you might be interested in, e.g.,

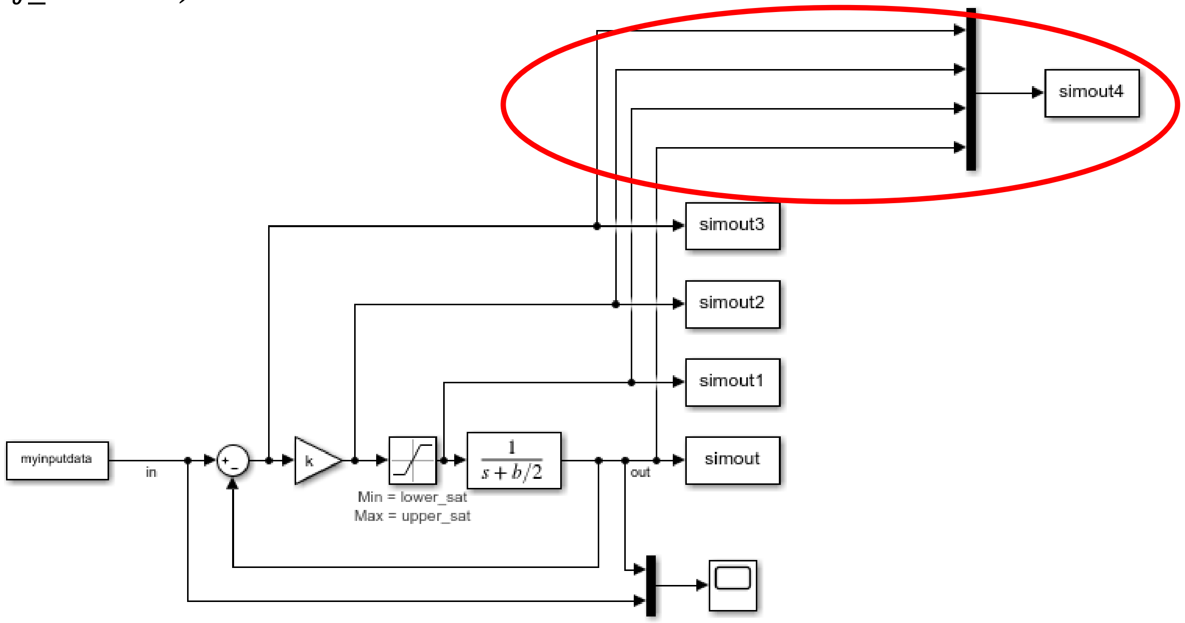


Set `k=2`. Run the model. You could plot the outputs on one axes, e.g., try this:

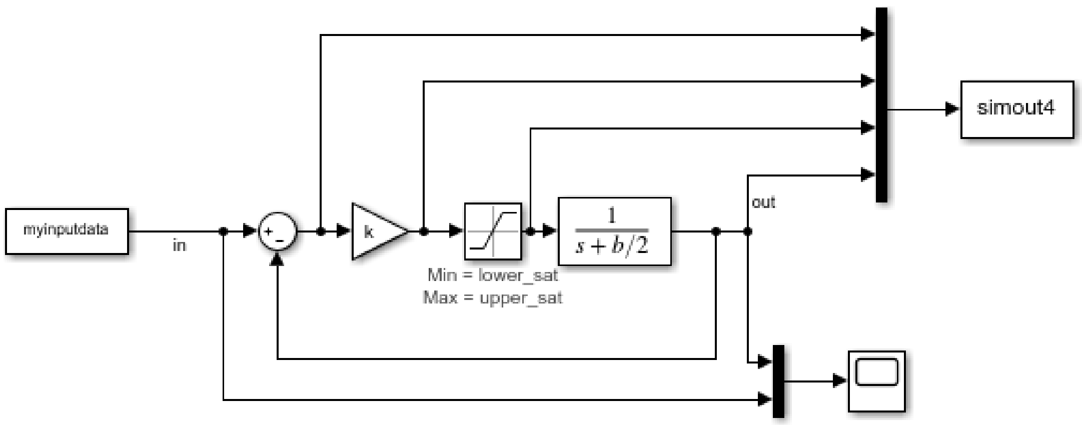
```
plot(myinputdata,'b-')
hold on
plot(simout3,'g.')
plot(simout2,'r:')
plot(simout1,'c--')
plot(simout,'k-')
```

Task

Or you could combine the outputs into one `simout` variable, like this ("my_model2c"):



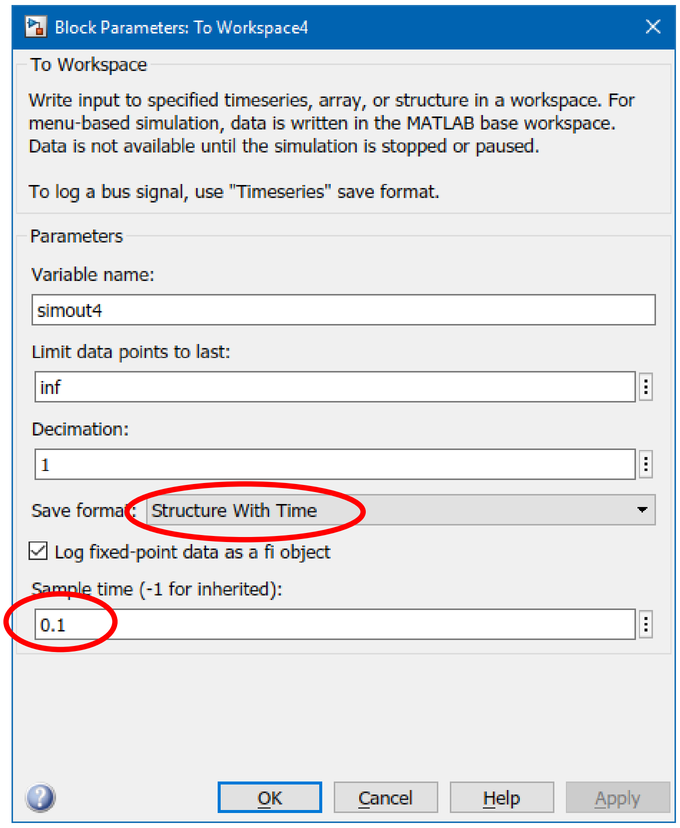
Let's simplify the model to remove several `simout` blocks, to end up with something like this ("model_2d"):



Then, to plot all the outputs in one go, simply use `plot(simout4)`. If you wanted to plot just one of the data sets from `simout4`, e.g., the fourth data column vs time, one way to do it is this: `plot(simout4.Time,simout4.Data(:,4))`

What if you wanted samples at regular spaced time intervals? You can change the sampling time for your output. E.g., change the block parameter sample time value for `simout4` to 0.1s. It is convenient to save the sampled times that match your data points too, so choose the save format as timeseries or a structure with time, as shown:

Task


Block Parameters: To Workspace4

To Workspace

Write input to specified timeseries, array, or structure in a workspace. For menu-based simulation, data is written in the MATLAB base workspace. Data is not available until the simulation is stopped or paused.

To log a bus signal, use "Timeseries" save format.

Parameters

Variable name: simout4

Limit data points to last: inf

Decimation: 1

Save format: Structure With Time

☑ Log fixed-point data as a fi object

Sample time (-1 for inherited): 0.1

Look at the variables produced in the workspace. You can plot the data in the `simout4` structure like this:
```
plot(simout4.time,simout4.signals.values)
```

If you wanted just the 4th data column, you could use
```
plot(simout4.time,simout4.signals.values(:,4))
```

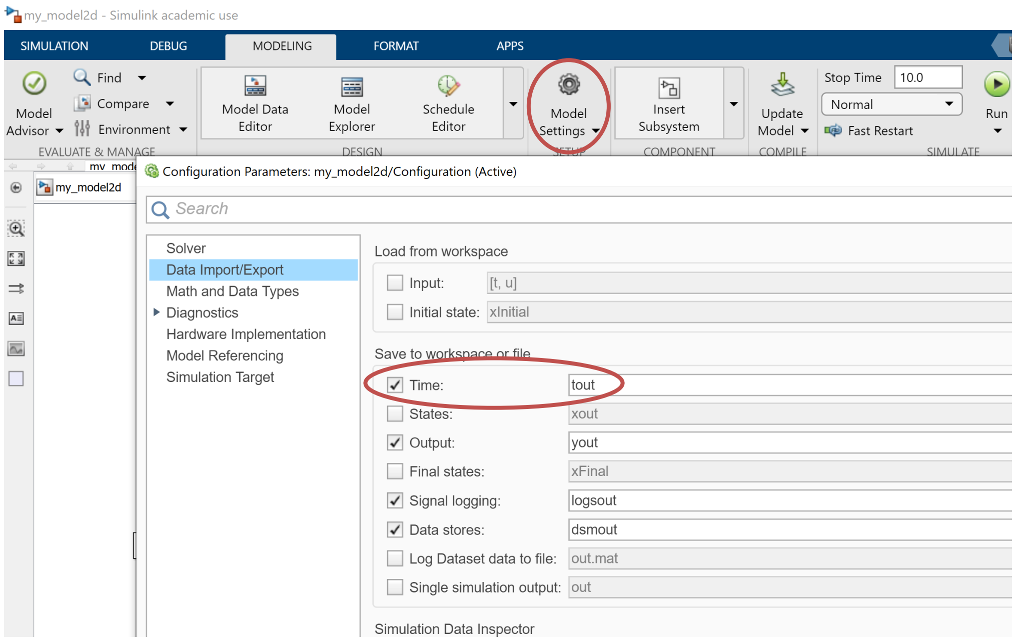
If you wanted to compare two signals you could plot them:

```
plot(simout4.time,simout4.signals.values(:,2))
hold on
plot(simout4.time,simout4.signals.values(:,3))
```

or use them with other MATLAB functions, e.g.,

```
mean(simout4.signals.values(:,2))
mean(simout4.signals.values(:,3))
```

Note: You might be wondering about `tout`. You'll see that `tout` might not be regularly spaced or the same size as the time column of the timeseries. That is because tout are solver step times and the solver is 'variable step'. You can change the solver to be fixed step to get regularly spaced tout values. See model config properties:


Configuration Parameters: my_model2d/Configuration (Active)

Load from workspace

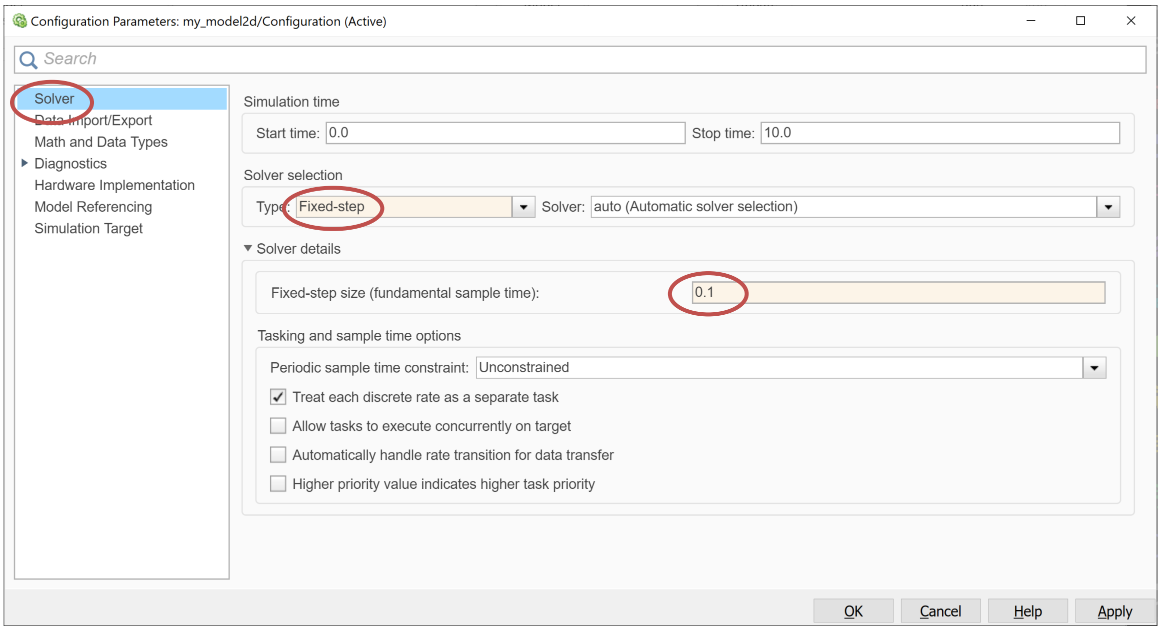
- [ ] Input: [t, u]
- [ ] Initial state: xInitial

Save to workspace or file

- [✓] Time: tout
- [ ] States: xout
- [✓] Output: yout
- [ ] Final states: xFinal
- [✓] Signal logging: logsout
- [✓] Data stores: dsmout
- [ ] Log Dataset data to file: out.mat
- [ ] Single simulation output: out

Simulation Data Inspector

And, here, for example the step is set to 0.1s


Configuration Parameters: my_model2d/Configuration (Active)

Simulation time

Start time: 0.0 Stop time: 10.0

Solver selection

Type: Fixed-step Solver: auto (Automatic solver selection)

Solver details

Fixed-step size (fundamental sample time): 0.1

Tasking and sample time options

Periodic sample time constraint: Unconstrained

- [✓] Treat each discrete rate as a separate task
- [ ] Allow tasks to execute concurrently on target
- [ ] Automatically handle rate transition for data transfer
- [ ] Higher priority value indicates higher task priority

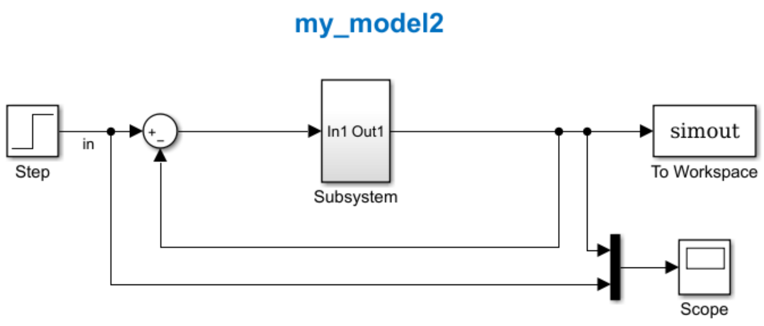
## 3.3 Creating subsystems

You can combine multiple blocks and signals into a ‘subsystem’. To do this use the mouse to drag a box around all the blocks you want in your subsystem. The MULTIPLE tab will appear. Click on “Create Subsystem” there. (A fast way is to enter Ctrl+G.)

Optional

Using my_model2, make a subsystem comprising of the gain, saturation and transfer function blocks. You should end up with model that looks like this:

Optional



Double-click on the subsystem to view what's inside it. Use the menu navigation arrows to get back to the top level of the model.

*Note 1*: You wouldn't normally create a subsystem for such a small model. However, creating subsystems is useful in the design of large models and also provides a way mask the underlying design and/or prompt for parameter values. You will see subsystems used in many of the demos that come with Simulink.

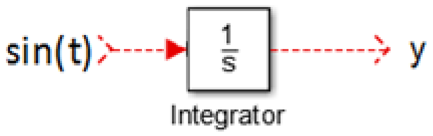
*Note 2*: If you wish to remove a subsystem: right-click the subsystem and select *Subsystem & Model Reference* → *Expand Subsystem* (or Shift+Ctrl+G), then drag the revealed blocks out of the shaded area, then delete the shaded area.

## 3.4 Modelling ODEs: a first look

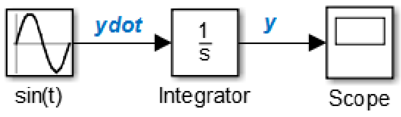
Consider

$$\frac{dy}{dt} = \dot{y} = \sin(t)$$

We can solve it by integrating $\sin(t)$ since $y = \int \dot{y}\, dt$. In Simulink, we use the integrator block for this:

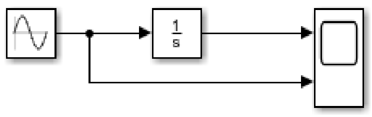


In Simulink, there is a *sin* block we can use for the input and we can send the solution *y* to a scope, so our ODE model is:



Task

Build the model. Check the scope output is correct, maybe view the input also on the scope, like this:



Optional

Below is another, more complicated, example to build: The Kermack-McKendrick SIR disease model is defined by the following system of equations[2]:

$$\dot{S}(t) = -aS(t)I(t)$$
$$\dot{I}(t) = aS(t)I(t) - bI(t)$$
$$\dot{R}(t) = bI(t)$$

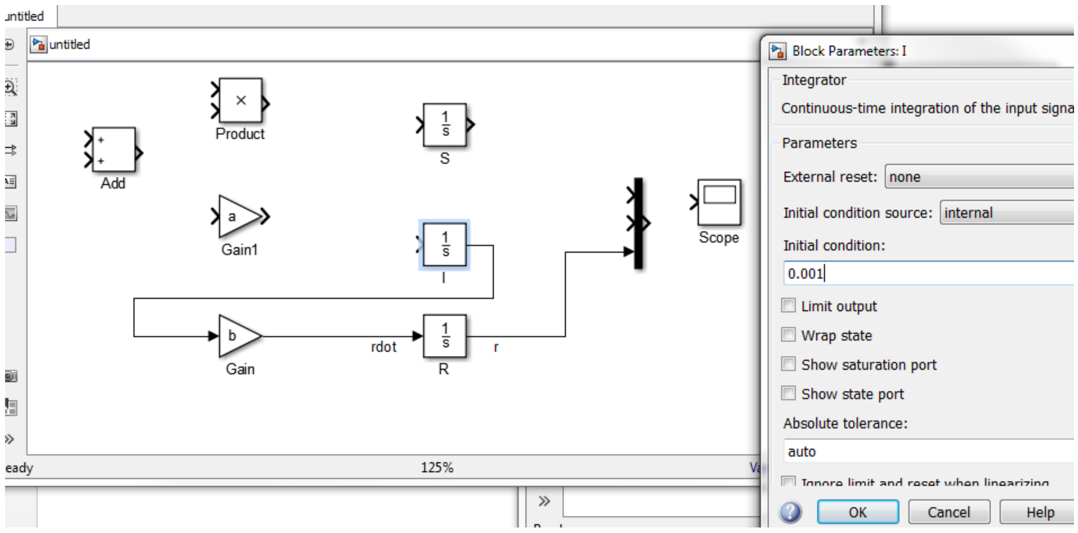
where $t$ is time, $S(t)$ is the number of susceptible people $I(t)$ is the number of people infected $R(t)$ is the number of people who have recovered and are now immune to reinfection $a$ is the infection rate, $b$ is the recovery rate.

To model in Simulink we can use

- three integrator blocks: one for $\dot{S}$, one for $\dot{I}$, one for $\dot{R}$
- a gain (or product) block for $a$ and for $b$ multiplication
- a subtraction block
- a scope to see the output

Let's also set initial conditions $S(0) = 1$ and, to infect a small part of the population, set $I(0) = 0.001$.

Start building integrator blocks for $S$, $I$, $R$, gain block for $a$ and $b$, a subtraction (use add and change sign) and product block, a scope, and connections for $\dot{R}(t) = bI(t)$



Block Parameters: I

Integrator

Continuous-time integration of the input signa

Parameters

External reset: none

Initial condition source: internal

Initial condition: 0.001

☐ Limit output

☐ Wrap state

☐ Show saturation port

☐ Show state port

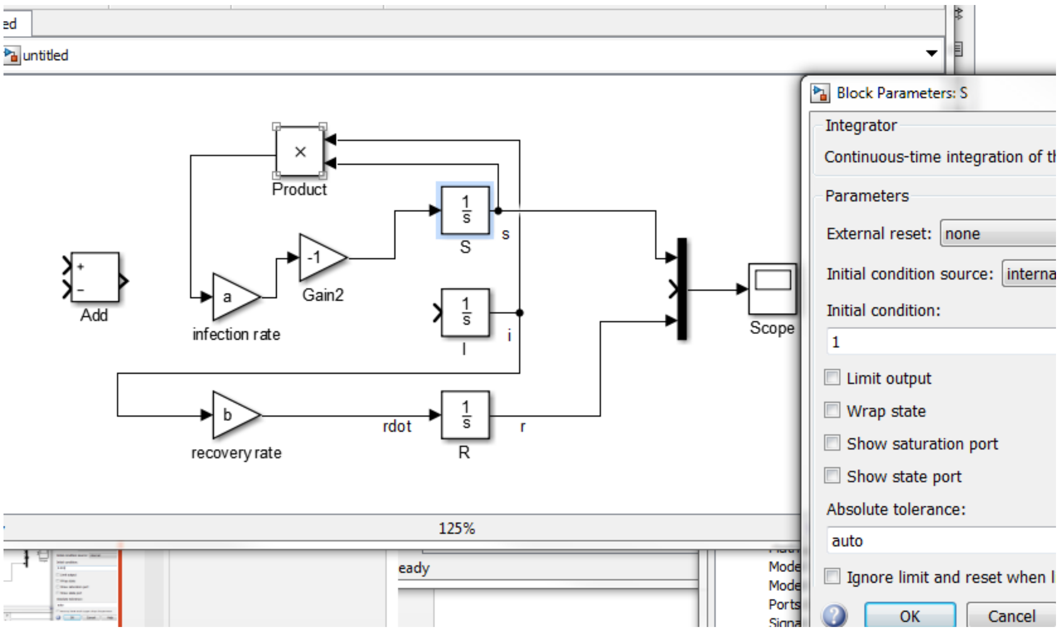
Absolute tolerance: auto

☐ Ignore limit and reset when linearizing

Now connect for $\dot{S}(t) = -aS(t)I(t)$ product $SI$ multiply by $a$ and by -1, set $S$ initial condition, edit labels:

[2] http://mathworld.wolfram.com/Kermack-McKendrickModel.html

Optional



Block Parameters: S

Integrator

Continuous-time integration of t

Parameters

External reset: none

Initial condition source: interna

Initial condition:

1

☐ Limit output

☐ Wrap state

☐ Show saturation port

☐ Show state port

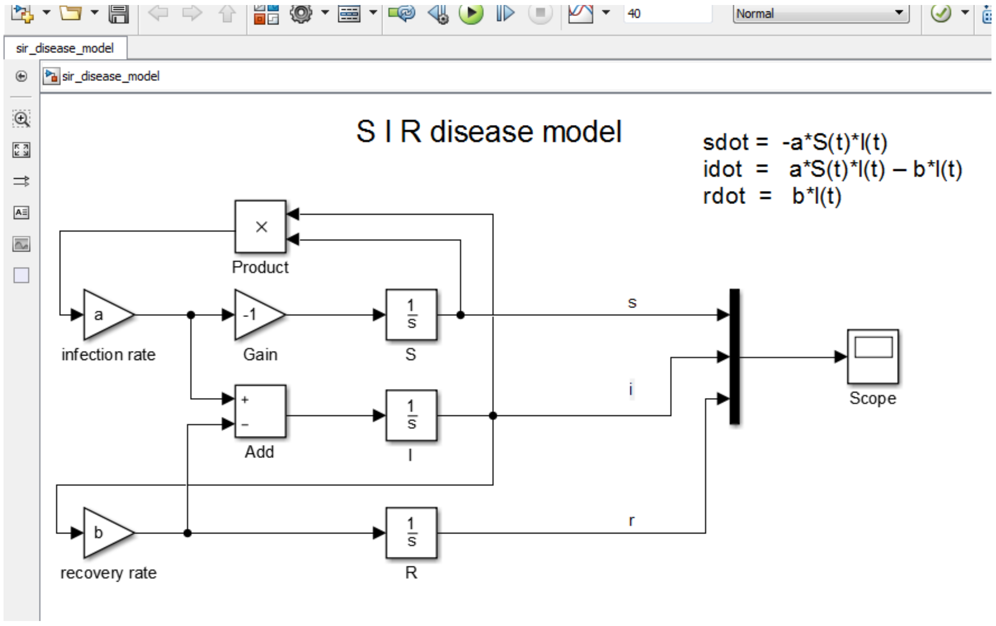
Absolute tolerance:

auto

☐ Ignore limit and reset when l

OK Cancel

Do similar for equation 3 and tidy up the positioning of the blocks...

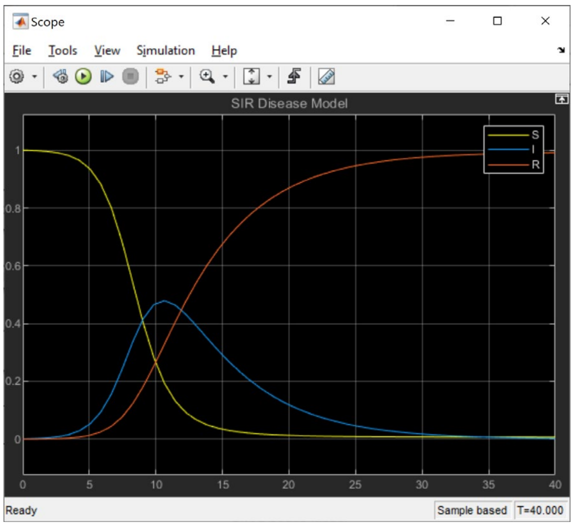


sdot = -a*S(t)*I(t)
idot = a*S(t)*I(t) – b*I(t)
rdot = b*I(t)

In MATLAB, set some values for a and b

```
>> a=1;
>> b=0.2;
```

and run the model ("sir_disease_model"). `Scope` output should be something like the figure opposite.



# LAB SESSION 4

## 4.1 More with matrices

### 4.1.1 Using matrices to solve simultaneous equations

Task

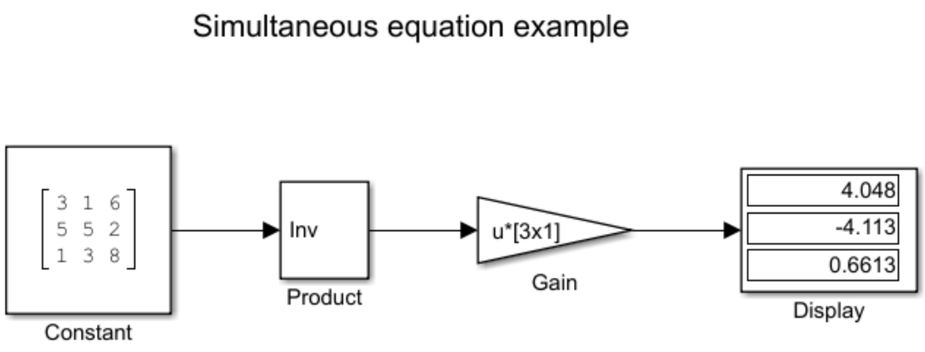
a. Use MATLAB to solve the following system of equations:

$$3x_1 + 1x_2 + 6x_3 = 12$$
$$5x_1 + 5x_2 + 2x_3 = 1$$
$$1x_1 + 3x_2 + 8x_3 = -3$$

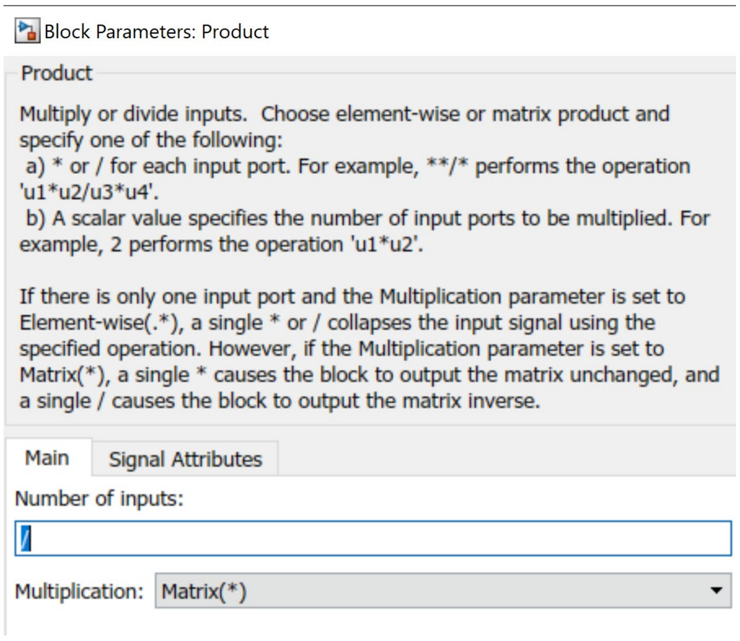
Put the equations in matrix form $Ax = b$, so that you can compute $x = A^{-1}b$. (Before matrix inversion, you should use `det` to check that the matrix is not singular.) Example solution: “simulteqn_example.m”.

Optional

b. You can use Simulink to do the same thing. Construct or run this model (available in Blackboard “matrices_in_simulink.slx”)



Note the order selected is u*K for the gain block. Also, note the product block setting to produce the matrix inverse is “/” as explained in the block help:


Block Parameters: Product

Product

Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:
a) * or / for each input port. For example, **/* performs the operation 'u1*u2/u3*u4'.
b) A scalar value specifies the number of input ports to be multiplied. For example, 2 performs the operation 'u1*u2'.

If there is only one input port and the Multiplication parameter is set to Element-wise(.*), a single * or / collapses the input signal using the specified operation. However, if the Multiplication parameter is set to Matrix(*), a single * causes the block to output the matrix unchanged, and a single / causes the block to output the matrix inverse.

Number of inputs: /

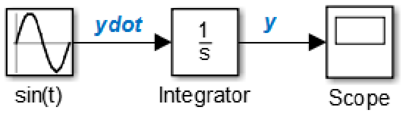
Multiplication: Matrix(*)

### 4.1.2 Using matrices for Simultaneous ODEs in Simulink

We have seen that it is possible to model ODEs such as

$$\frac{dy}{dt} = \dot{y} = \sin(t)$$

by expressing them in the form $y = \int \dot{y}\,dt$ and then using integrator blocks in Simulink to make the ODE model:

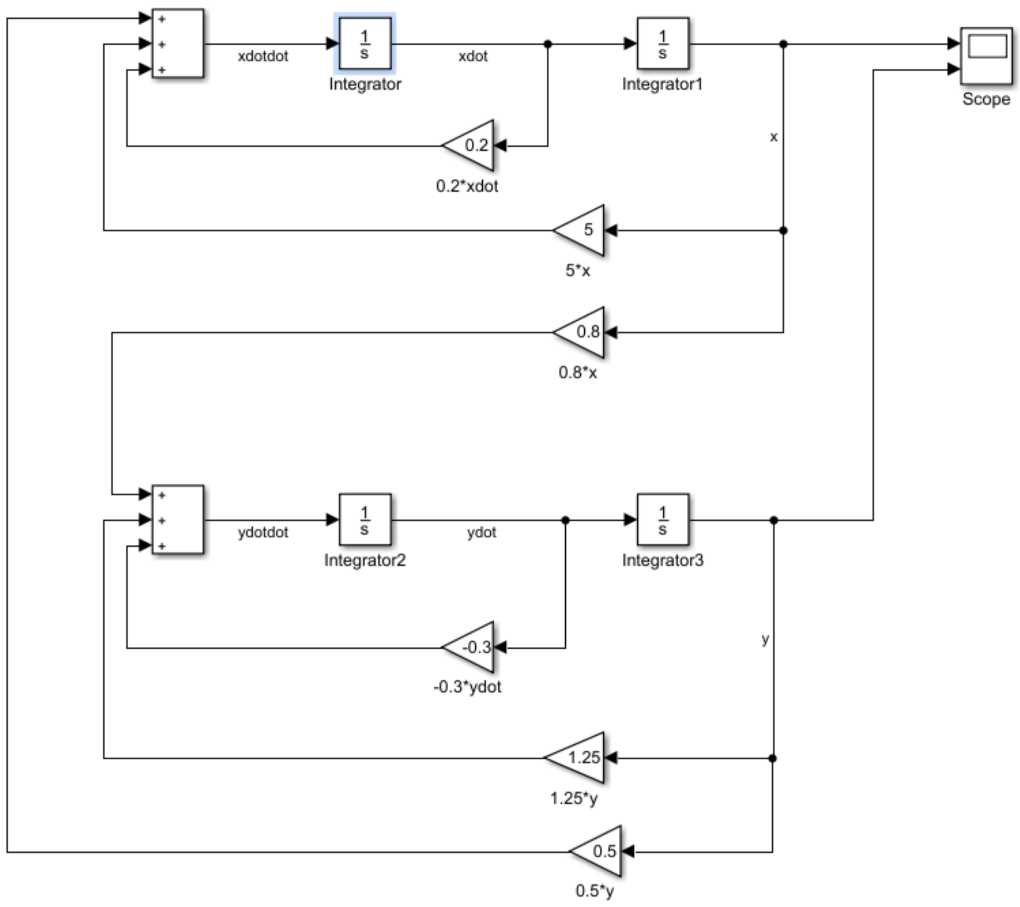


Suppose now that we have a system that is described by the two ODE's:

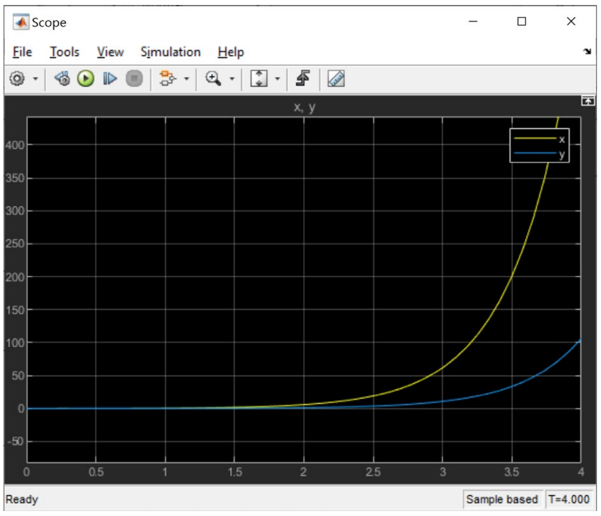
$$\ddot{x} = 0.2\dot{x} + 5\mathrm{x} + 0.5\mathrm{y}$$
$$\ddot{y} = -0.3\dot{y} + 0.8\mathrm{x} + 1.25\mathrm{y}$$

Where $\dot{x}(0) = 0, x(0) = 0.1, \dot{y}(0) = 0, y(0) = 0.1$, and we wish to obtain a solution over the domain $0 \le t \le 4$. We could construct a Simulink model for the system like the one below (also available in Blackboard as "simultODEexample1.slx").

Example of Simultaneous Ordinary Differential equations in Simulink



If we, for example, set the initial conditions of the integrator1 and intergrator3 blocks to 0.1, leave the others at default 0, and change the Simulink run time to 4 and then run the model the scope output is:



Task

Now, using matrices, our two ODEs could be represented like this:

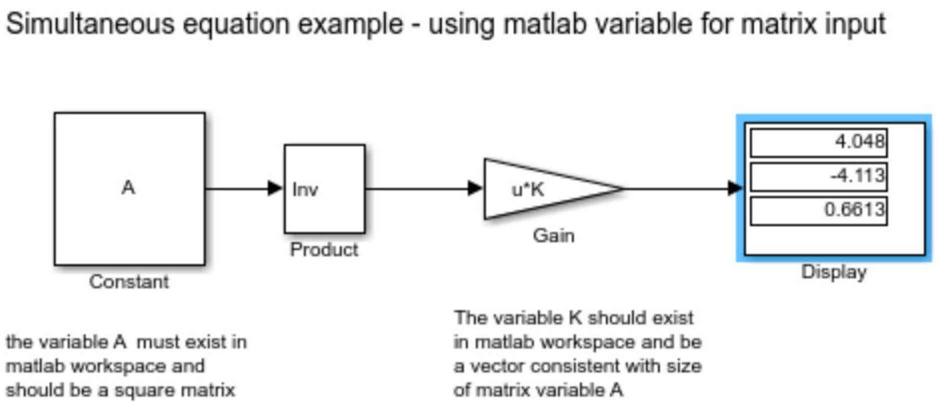
$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\end{bmatrix} = \begin{bmatrix}0.2\dot{x}\\ -0.3\dot{y}\end{bmatrix} + \begin{bmatrix}5 & 0.5\\ 0.8 & 1.25\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix}$$

Using two integrator blocks, and one gain block for the `[xdot, ydot]` multiplication by `[0.2;-0.3]`, and one gain block for the `[x, y]` multiplication by `[5 0.5; 0.8 1.25]`, and one two-input sum block, and one demux block and one scope, construct a Simulink model of the ODEs in matrix form. Check and compare the scope output to the non-matrix model. If stuck, see “simultODEexample2.slx” available in Blackboard.

### 4.1.3 Using MATLAB matrix variables in Simulink

Remember that variables in the MATLAB workspace can be used as block parameters. So, for example, in the optional simultaneous equation task above, we could have assigned variables in MATLAB and used them in a Simulink model as shown below (in Blackboard as “matrices_in_simulink_2.slx”):

```
>> A = [3 1 6; 5 5 2; 1 3 8];
>> K = [12; 1; -3];
```


Simultaneous equation example - using matlab variable for matrix input
A
Constant
Inv
Product
u*K
Gain
4.048
-4.113
0.6613
Display
the variable A must exist in matlab workspace and should be a square matrix
The variable K should exist in matlab workspace and be a vector consistent with size of matrix variable A

### 4.1.4 Indexing with Logical arrays

It is sometimes useful to be able to find non-zero values or to locate specified values within an array. Logical arrays and `find` can be used to do this.

Task

Enter the following commands at the command line and ensure you understand the results:

a. To find the indices of non-zero elements

```
>> X=[1 0 2 -3 0 0 10 0]; find(X)
```

b. To replace zeros with 9:

```
>> X(find(X==0))=9
```

Optional

c. Here we use a Logical array to extract some values from a 2D array. (Think of it as a mask.) In the command window, enter the following lines without semi-colons at the end so you see the result of each line.

```
>> AA=[10,9,8;7,6,5;4,3,2]
>> L=logical(eye(3))
>> AA(L)
```

In the above lines we created a 3x3 numeric data type array variable `AA`, we created a 3x3 logical data type array variable `L`, we indexed `AA` using our logical array to select the elements on the diagonal.

## 4.2 Using MATLAB efficiently: vectorisation of code

Vectorisation is the term given to the use of code and operations to process vectors, arrays, or matrices 'en masse' (rather than looping through elements). In MATLAB, vectorised code is typically more compact and executes faster.

To write vectorised code

1. Use vector, array and matrix operations in expressions where possible.
2. Pre-allocate memory for vectors and matrices where possible.
3. Make use of logical and/or vectorised indexing (i.e., avoid loop counted indexing if possible).

(a) An example using vectorised indexing

Assume we have two same sized matrices A, B and want to copy the 1st column from A to B

- Unvectorised loop method:

```
[numr,numc]=size(A);
for k=1:numr
  B(k,1)=A(k,1);
end
```

- Vectorised method:

```
B(:,1)=A(:,1);
```

(b) An example using an array 'en masse' in an expression

You can make use of functions and operations on entire arrays at once. E.g. assume `x` is a vector of numbers. Assume we want a vector `y` to have sine values of each `x` element.

- Loop method

```
for k=1:length(x)
  y(k)=sin(x(k))
end
```

- Vectorised method (sin function can take arrays as arguments):

```
y=sin(x)
```

(c) Preallocation example

Task

- Here's some example code with no pre-allocation of memory for the array. This means the loop changes size of variable `y` each cycle. Try it and see the warning in the editor.

```
x=-100:0.001:100;
for k=1:length(x)
    if x(k)>0
        y(k)=sqrt(x(k));
    else
        y(k)=0;
    end
end
```

- Here's the same code but preallocating variable `y` before its use within a loop. (It may also be possible to preallocate in such a way as to populate some of the elements to values we finally want.)

```
x=-100:0.001:100;
y=zeros(size(x));
for k=1:length(x)
  if x(k)>0
    y(k)=sqrt(x(k));
  end
end
```

(d) Using logical indexing

You might be able to use logical indexing to change the values of the elements in the array without having to use a loop. For example, here's a vectorised version of the code above. The loop and if statement has been replaced by a logical indexing expression.

```
x=-100:0.001:100;
y=zeros(size(x));
y(x>0)=sqrt(x(x>0));
```

Task

Compare run times of the no-preallocation loop code in (c) with the vectorised version in (d) above. (Use `tic` and `toc` to start and stop the timing. Available in Blackboard as "vectorised_example.m".)

## 4.3 Further plotting

### 4.3.1 Basic plotting reminder

Task

Run the following code in the file “plot_intro.m” (also shown below) section by section and see how each plot is produced.

```
% Introduction to plotting
% (examples by J A Rossiter)

%% Defining a domain
% try following 3 lines to see how x-axis values might be defined
x=0:10          % set x values as 0 1 2 3 4 5 6 7 8 9 10
xx=0:.1:2       % set xx values as 0 0.1 0.2 0.3 ....... 1.8 1.9 2.0
xxx=-4:0.4:2    %set xxx values as -4.0 -3.6 -3.2 .... 1.2 1.6 2.0

%% Simple plotting examples
x=-1.5:.02:1.5; % define domain
y=tan(x);       % evaluate tan over this domain
figure(1)
plot(x,y)
title('Tangent curve')

figure(2)
x2=-3:.1:5;
p=poly([-1 -2]);    % assumes polynomial (x+1)(x+2), roots -1 and -2
px=polyval(p,x2);   % computes polynomial values
plot(x2,px,'r:')    % change from default blue line plot
title(['Plot of polynomial with coefficients ',num2str(p)])

%% 2 lines in one plot without using hold on
figure(3)
plot(x,y,'go-',x2,px,'c--');     % two lines in plot command
name=poly2str(p,'x');            % Generates string/text of polynomial
title(['Plot of ',name,' and tan(x)'],'Fontsize',18,'Color',[.4,.6,.1])
legend('tangent','Polynomial')
xlabel('x-axis'), ylabel('y-axis')
text(2,-1,'Text in the graph')

%% Modify an existing plot, refer to fig number
figure(1)
axis([-0.5,1,-2,2])

%% Using subplots
figure(4)
subplot(231), plot(x,y), title('tangent')
subplot(235), plot(x2,px), title('Polynomial')
subplot(234), plot(x,y*2,'g'), title('Scaled tangent')

%% Some subplots with handles, created using a loop
figure(5)
X=[1:10]'*[2:.1:2.4];
nrows=size(X,2);
for k=1:nrows
    h(k)=subplot(nrows,1,k); % creates an array of handles
    plot(X(:,k))
end

hold on
plot(1:.5:5,'r')% alters just the last plot
hold off

subplot(h(2))   % use a handle to modify one of the subplots
hold on
plot(1:.5:5,'g')
hold off
```

### 4.3.2 A 3D plotting example

Below is an example that loads some array data and creates a 3D plot. (Blackboard files “mesh_plane_example.m” and “expsincosdata.mat”.) The data have a 41x41 array.

```
% meshgrid and surf plot with horiz plane intercept example

load expsincosdata % load the data in expsincosdata
surf(z)

% add a horiz plane to the plot
hold on
[x,y]=meshgrid(1:size(z,1),1:size(z,2)); % set up same grid size
zheight=1.5;
z2=zheight*ones(length(x),length(y)); % zheight*one for all grid points
mesh(x,y,z2) % a mesh plot, we could use surf again if we wished

% add title and labels
title('expsincosdata plot with horizontal plane intercept')
xlabel('x'), ylabel('y'), zlabel('z')
hold off
```

Task

Run the script and explore how it works.

Optional

You don’t always have to create scripts to use MATLAB’s plotting capabilities. Try this:

1. Click on the MATLAB’s PLOTS tab.
2. In the workspace Click on the z variable (don’t double-click it) and you will see the greyed-out plot icons in the plots tab on the main menu become coloured.
3. Explore a few plot types suitable to the data you have, e.g., `surf` or `bar3h`. If you wish to explore further, see Help and navigate to the Graphics section.

### 4.3.3 Symbols and subscripts in plot axes and titles

Task

A simplified form of Lorenz’s equations to model a chaotic system is given by:

$$\dot{x}_1 = \sigma(x_2 - x_2)$$
$$\dot{x}_2 = x_1(\rho - x_3) - x_2$$
$$\dot{x}_3 = x_1 x_2 - \beta x_3$$

Assume that a soluntion has been found and results were saved to “yxBRS_loz.mat” (available on Blackboard). The code below produces a comet 3D plot of $\dot{x}_1$, $\dot{x}_2$ $\dot{x}_3$ (data saved as `y`) and gives the plot a two-line title that includes the values for $\sigma$, $\rho$, $\beta$ (saved as `S`, `R`, `B`) and the initial conditions for $x_1$, $x_2$, $x_3$, and uses symbols and subscripts in the title text. Run the following code to see the resulting axesa and title (“plot_title_example.m”)

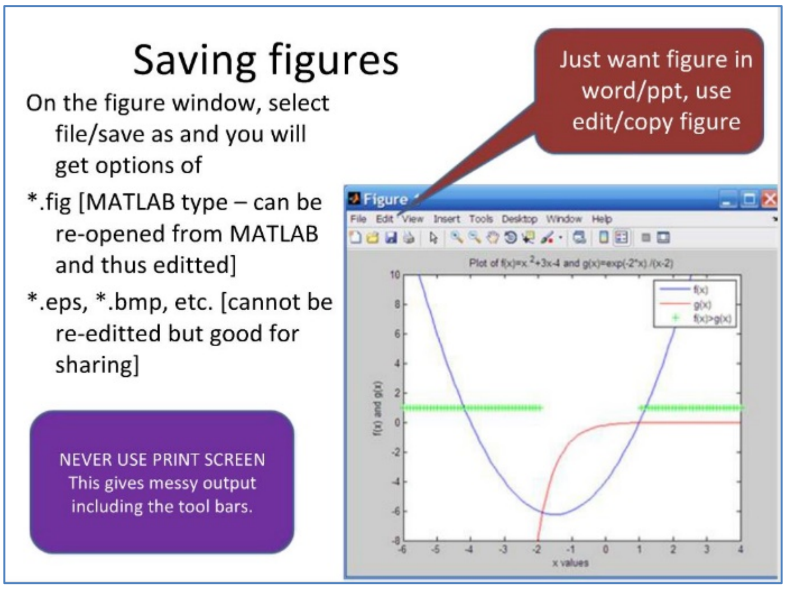
```
% An example plot with subscripts and symbols in title
figure(6)
load yxBRS_Loz

comet3(y(:,1),y(:,2),y(:,3))     % create a 3D comet plot of y data
xlabel('dx_1/dt'), ylabel('dx_2/dt'), zlabel('dx_3/dt')
title({['Lorenz equations solution with \beta=',num2str(B),...
    ', \rho=',num2str(R),', \sigma=',num2str(S)];...
    ['and initial conditions x_1=',num2str(x(1)),', x_2=',num2str(x(2)),...
```

```
     ', x_3=',num2str(x(3))]})
grid on
```

## 4.4 Using MATLAB output in reports

### 4.4.1 Saving figures



Saving figures

On the figure window, select file/save as and you will get options of

*.fig [MATLAB type – can be re-opened from MATLAB and thus editted]

*.eps, *.bmp, etc. [cannot be re-editted but good for sharing]

Just want figure in word/ppt, use edit/copy figure

NEVER USE PRINT SCREEN
This gives messy output including the tool bars.

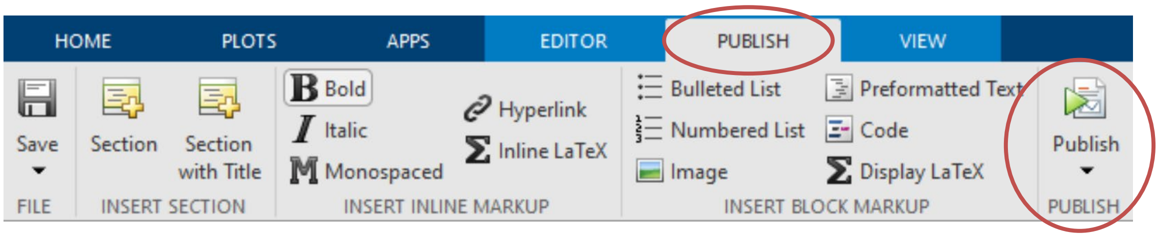
Figure
Plot of f(x)=x.^2+3x-4 and g(x)=exp(-2*x)./(x-2)
f(x)
g(x)
f(x)>g(x)
f(x) and g(x)
x values

### 4.4.2 Publisher tool

MATLAB's Publisher tool may sometimes be useful as an aid to producing reports that include your code, results, and plots. E.g., open the file "fourier_square_wave_publish.m" in the MATLAB Editor. (The file is available on Blackboard.) Notice how the double %% highlights each section as you move the cursor through the file.

Task

Make a report: click on the PUBLISH tab and then click the Publish Run button.



HOME
PLOTS
APPS
EDITOR
PUBLISH
VIEW
Save
Section
Section with Title
Bold
Italic
Monospaced
Hyperlink
Inline LaTeX
Bulleted List
Numbered List
Image
Preformatted Text
Code
Display LaTeX
Publish
FILE
INSERT SECTION
INSERT INLINE MARKUP
INSERT BLOCK MARKUP
PUBLISH