Introduction to Computational Science

Eng. Phd. Giuseppe Lombardo
CEMIF.CAL, INFM-Licryl Laboratory,
University of Calabria, Italy

16 May – 25 June 2005

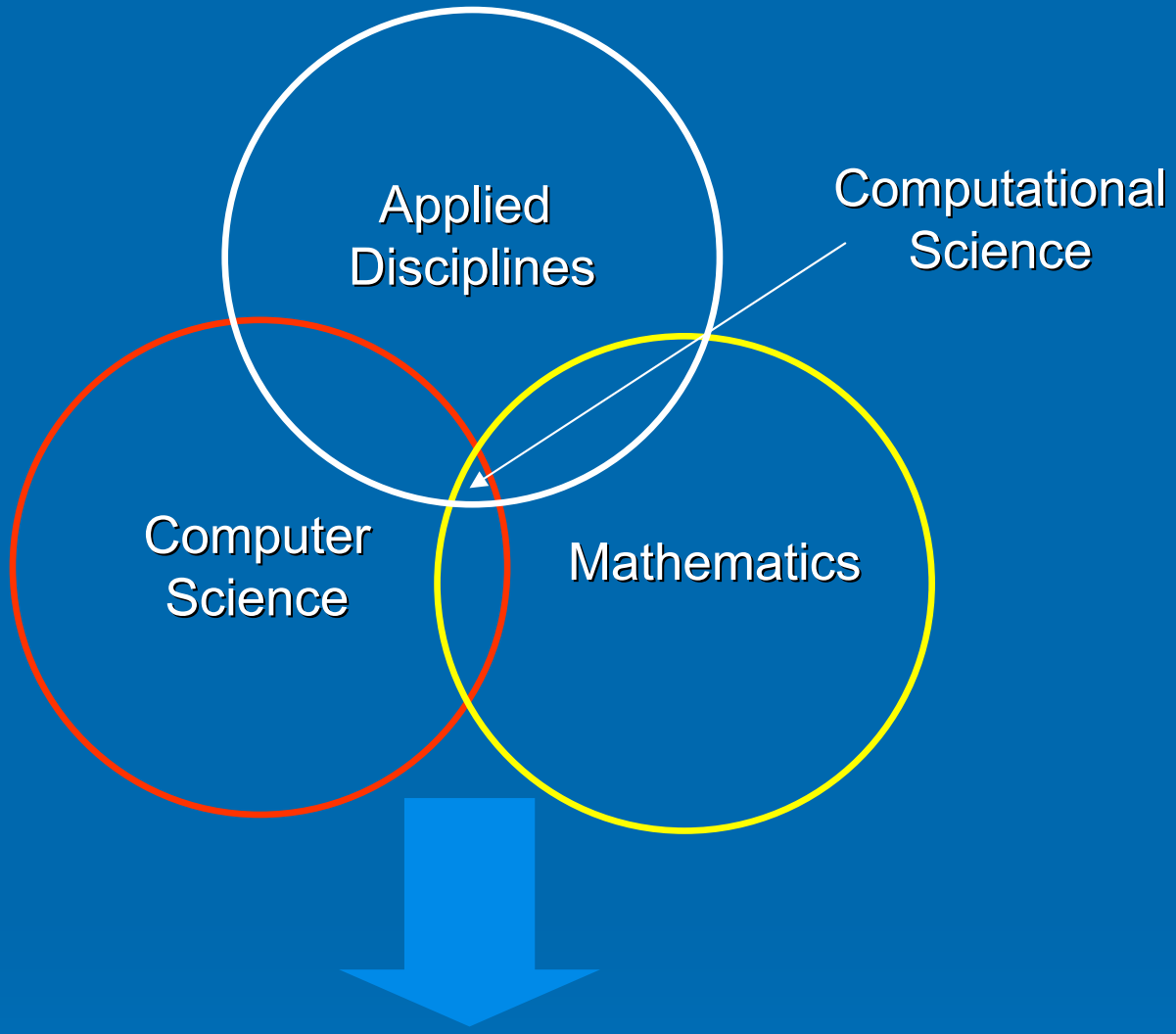
Table of Contents

- Computational Science
- MatLab: the Language of technical computing
- Mathematical models: Ordinary Differential Equation (ODE)
  - Initial Value Problem (IVP)
  - Boundary Value Problem (BVP)
- Introduction to Simulink (extension of MatLab): simulation of dynamical systems.

What is Computational Science?

- **Computational Science:**
  The study of scientific or engineering problems using computers + mathematical models as tools.

- **Computer Science:**
  The study of computer algorithms, languages, and machines for solving problems.



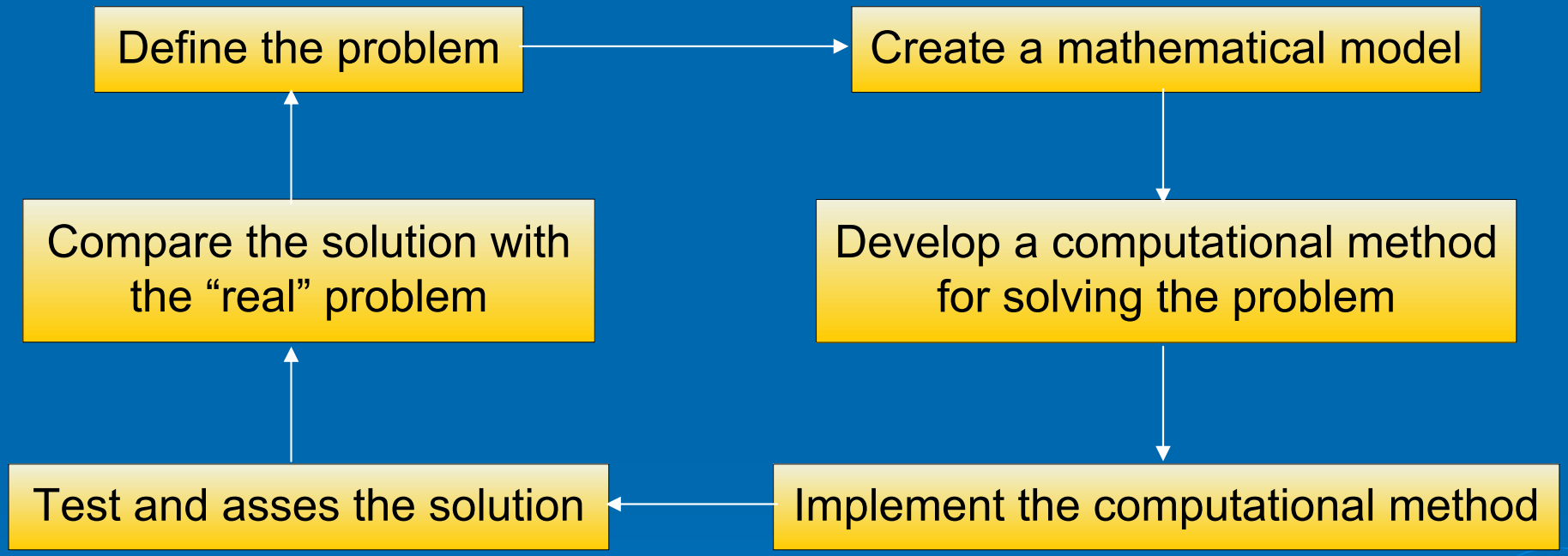
Computational Science is multidisciplinary.

What is Expected From this lecture?

- **Goals:**
  - Know how to use a computer to solve scientific problems;
  - Become familiar and competent in the use of MatLab for solving mathematical problems;

- **Not Goals:**
  - Be an amazing programmer;
  - Teach you about maths;

Problem – Solving Process



Problem solving
Example – Ball Trajectory

- Problem definition:

  A small object is launched into flight from the ground at a speed of 50 m/s at 30 degrees above the horizontal over level ground. Determine the trajectory covered by the ball

- Mathematical Model:

  Time: t(s) with t=0 when the object is launched;

  Initial velocity magnitude: v_0=50 m/s;

  Initial angle: θ_0=30°;

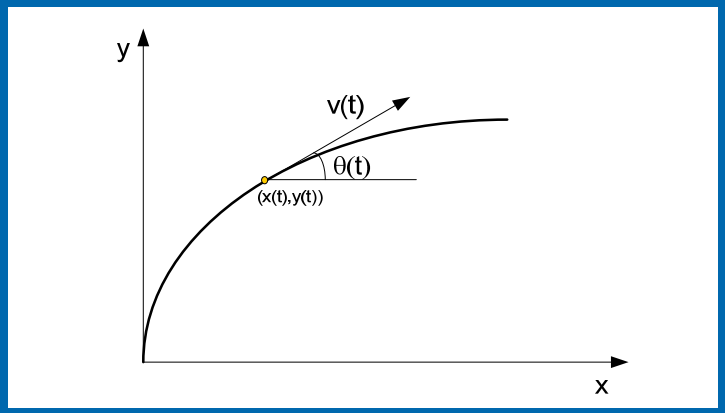
  Vertical position of ball: y(t) m and y(0)=0;

  Horizontal position of the ball: x(t) m and x(0)=0;

  Acceleration of gravity: g= 9.8 m/s^2, directed in negative y direction.

Example – Ball Trajectory (2)

The key step in developing a mathematical model is to divide the trajectory into its horizontal and vertical components. Hence to describe the problem, we adopt the following reference frame



The velocity of the ball at time t is described by a vector: $\bar{v}(t) = (\dot{x}(t), \dot{y}(t))$, having as magnitude: $v(t) = \sqrt{\dot{x}(t)^2 + \dot{y}(t)^2}$. θ(t) is the angle that the velocity vector form with the horizontal: $\theta(t) = \arctan \frac{\dot{y}(t)}{\dot{x}(t)}$

Example – Ball Trajectory (3)

Applying the Newton law to the problem, we obtain the mathematical model of our system:

$$\frac{d}{dt}(m\bar{v}(t)) = F \Rightarrow \left\{ \begin{array}{l} \dot{x}(t) = v(t) * \cos(\theta(t)) \\ \dot{y}(t) = v(t) * \sin(\theta(t)) \\ \dot{v}(t) = -g * \sin(\theta(t)) \\ \dot{\theta}(t) = -\dfrac{g}{v(t)} * \cos(\theta(t)) \end{array} \right\}$$

with initial condition:

$$\left\{ \begin{array}{l} x_0 = 0 \\ y_0 = 0 \\ v_0 = 50 \\ \theta_0 = 30 \end{array} \right\}$$

Example – Ball Trajectory (4)

- Implementation of the computation method to solve the differential equations:

The equations defined in the mathematical model can be readily implemented using MatLab. The commands used in the following solution will be discussed in detail later, but observe that the MatLab steps match closely to the mathematical model just developed.

Example – Ball Trajectory (5)

```
global g
g=9.8; %m/s2
tf=5.1;
x0=0;
y0=0;
v0=50; %m/s;
teta0=30*pi/180;
```

Parameters for the problem

```
function xdot=flighttraj(t,x)
%Compute the system of differential equations:
global g
xdot(1)=x(3)*cos(x(4));
xdot(2)=x(3)*sin(x(4));
xdot(3)=-g*sin(x(4));
xdot(4)=-(g/x(3))*cos(x(4));
xdot=[xdot(1);xdot(2);xdot(3);xdot(4)];
```

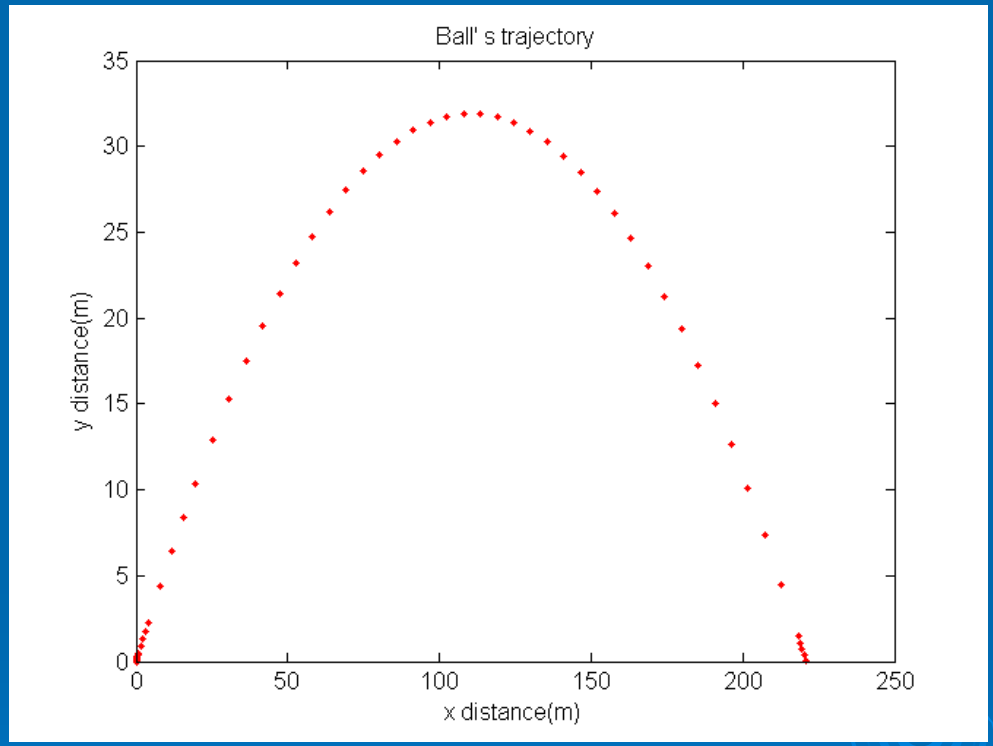
Mathematical model

```
%Solve the system of differential equations
using as solver a Runge Kutta algorithm
[t,y]=ode45('flighttraj',[0 tf],[x0 y0 v0 teta0]);
```

Computational method

```
%Plot the calculate trajectory
plot(y(:,1),y(:,2),'r.');
```

Plot the solution



Example – Lotka Volterra

- **Problem definition:**

It is the classical Predator-Prey model, where are two species systems: P predator and V prey.

- Predators are the only factor that cuts down the prey population;
- The remaining prey population breeds without limitations;
- Predator's breeding rate is proportional to their catch
- The number of predators is limited by death rate.

- **Mathematical Model:**

$$\begin{cases} \dot{V}=kV-aVP \\ \dot{P}=bVP-dP \end{cases}$$

$V(0)=0.5;$

$P(0)=0.5;$

k: is the birth rate of prey

a: is catch rate

b: characterizes the efficency with which predators use their catch to produce more predators

d: is the death rate for predators

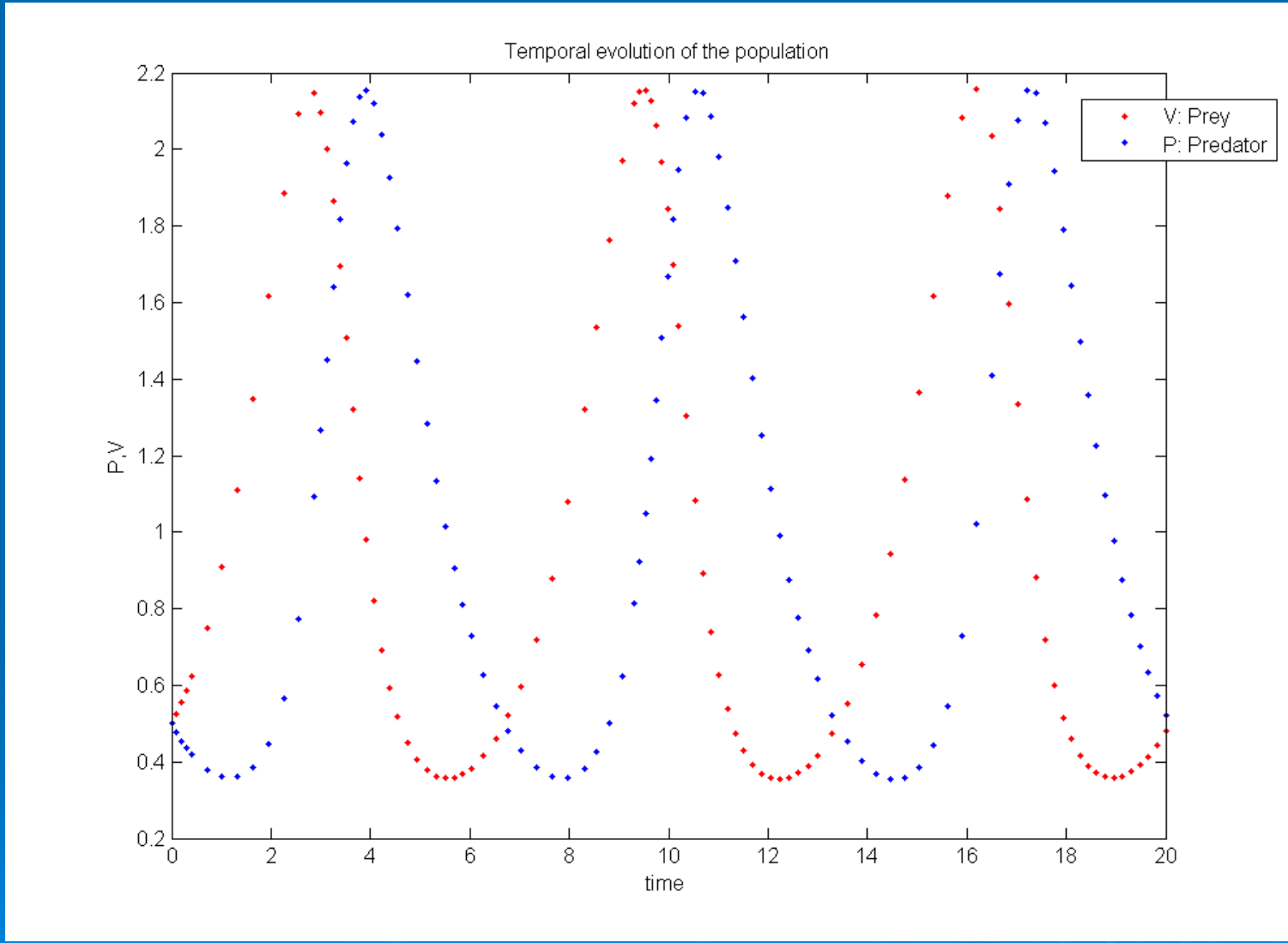
Example – Lotka Volterra (2)

- Implement computational method

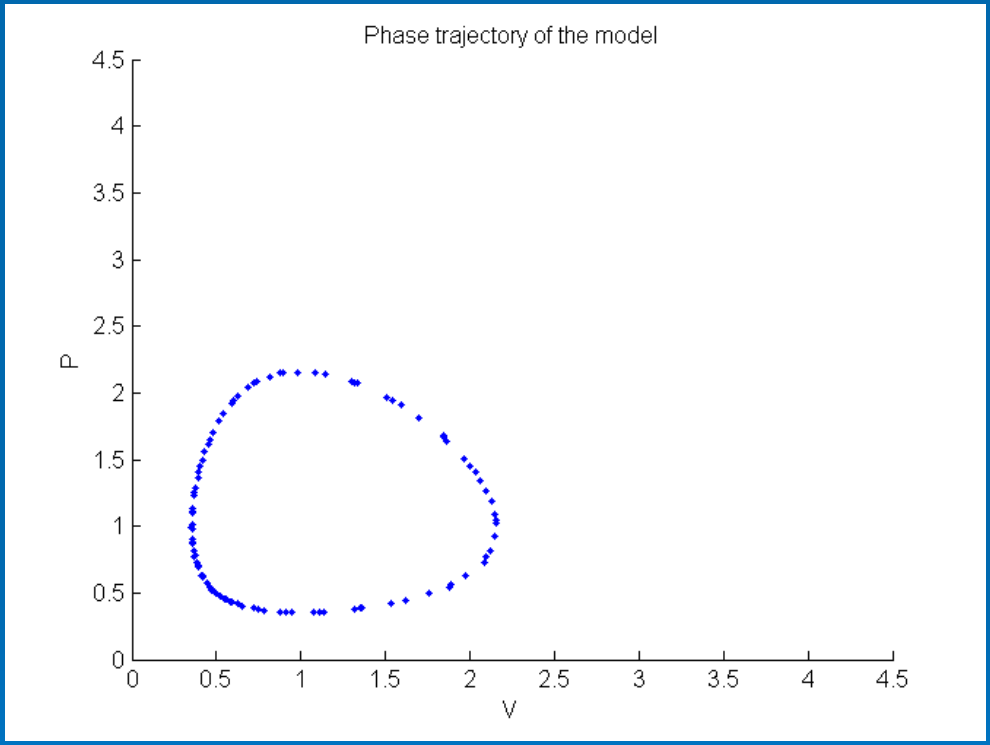
```
k=1;a=1;b=1;d=1;

function xdot=predator_prey(t,x,flag,k,a,b,d)
%Predator-Prey model:
%Solution of Lotka-Volterra
%differential model:
 %Input:
% k , a, b, d, describe the two
% species in exam
%System of differential equations
xdot(1)=k*x(1)-a*x(1)*x(2);
xdot(2)=b*x(1)*x(2)-d*x(2);
xdot=[xdot(1);xdot(2)];
```

```
[t,x]=ode45('predator_prey',[0 20],[.5 .5],options,k,a,b,d);

%Plot the solution
plot(t,x(:,1),'r.');
hold on
plot(t,x(:,2),'b.');
xlabel('time');
ylabel('V,P');
legend('V: Prey','P: Predator');
title('Temporal evolution of the population');
figure;
hold on;
plot(x(:,1),x(:,2),'k.');
title('Phase trajectory of the model');
xlabel('V');
ylabel('P');
```

Example – Lotka Volterra (3)



Example – Lotka Volterra (4)



Stable equilibrium of the systems:

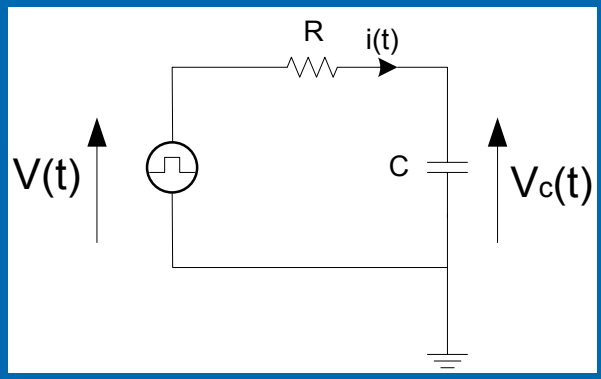
$$kV\text{-}aVP=0 \Rightarrow P_{eq} = \frac{k}{a} = 1$$

$$bVP\text{-}dP=0 \Rightarrow V_{eq} = \frac{d}{b} = 1$$

Example – RC circuit

- **Problem definition:**

  Considering the RC network shown in figure, calculate the $v_c(t)$ across the capacitor when we apply a source v(t);



- **Mathematical model:**

  Resistance: R= 10KΩ;

  Capacitance: C= 33 μF;

  Voltage across the capacitance: $V_c(t)$ with $V_c(0)=0$;

  Current into the circuit: i(t)

  Source voltage:

$$V(t) = \begin{cases} 0 & 0 \le t<10 \\ 1 & 10 \le t<15 \\ 0 & t \ge 15 \end{cases}$$

Example – RC circuit (2)

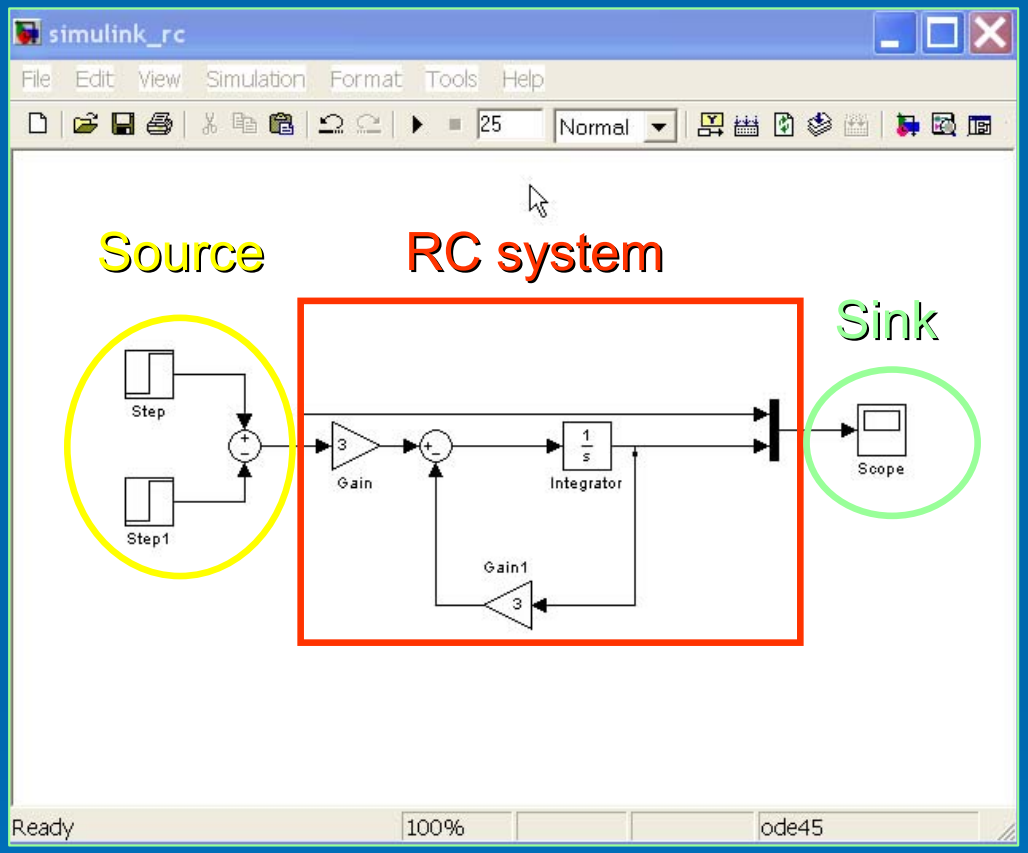
Using the Kirchoff current law to the rc network, we obtain the differential

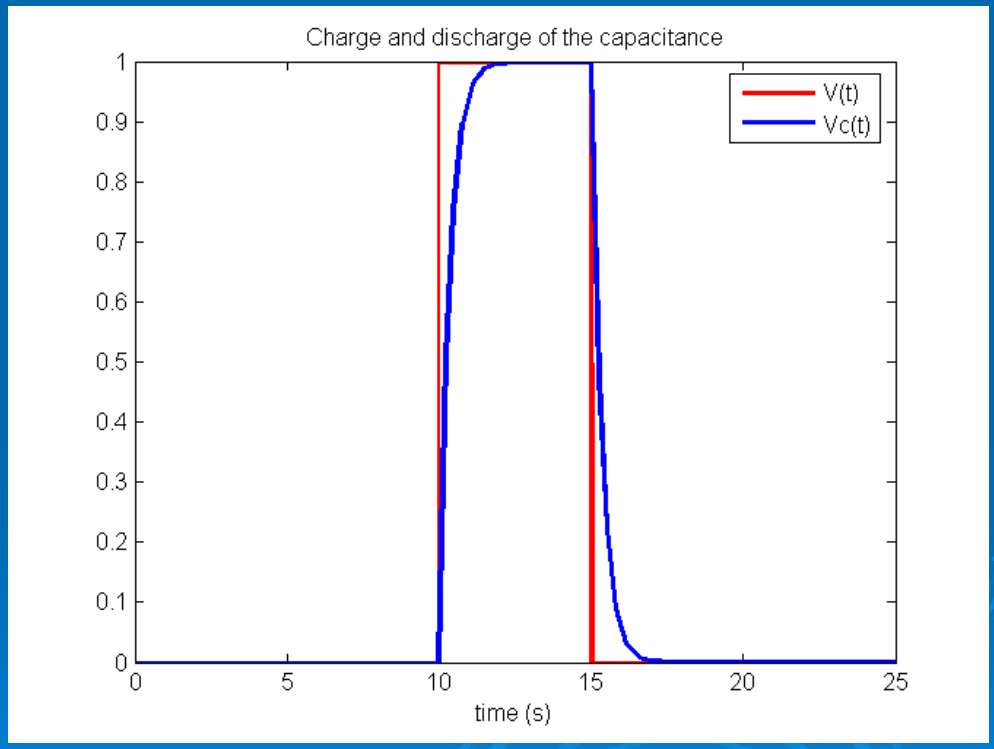
voltage equation: $C\dot{V}_c(t) + \frac{V_c(t)}{R} = \frac{V(t)}{R}$

- Implementation of the computational method:

In this case, to solve the differential equation, we have used a new computational tool: Simulink.
Simulink is a graphical tool that allows modelling and simulation of complex dynamical systems through block diagrams. At its core, it involves an automated solution of ODE's based on input block diagrams.

Example – RC circuit (2)





Mathematical Models

The behaviour of physical systems is obtained by utilizing the physical laws of mechanical, electrical, fluid and thermodynamic systems and etc.. .
We generally model physical systems with ordinary or partial differential equations:

- By far the most common method
- Ample supply of numerical methods and computer programs

Errors arising from modelling

- Modelling errors:
  - The gap between the model and “reality”;
  - Errors due to (approximate) mathematical methods in solving the model.
  - Errors arise from numerical methods used to solve the problem

Errors arising from modelling (2)

- Errors induced by the use of computers
  - Computer arithmetic (rounding)
  - Errors in programming
  - Errors in post processing (visualization, data analysis)
  - Wrong interpretation of the results.

Why use models?

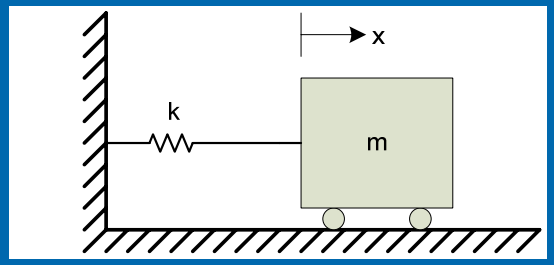
- To understand better the behaviour of a systems;
- To predict the future development of a system or even to manipulate the system in order to achieve a desired result;
- Sometimes it's the only feasible way to find a solution is trough modelling.
- Models are usually cheaper and faster than conventional experiments (i.e. simulation of a control unit for an aeroplane)

Computing Software

MatLab and Simulink are computer programs that combine computation and visualization power that make them particularly useful tools to develop rapidly and accurately mathematical models.
They are both computer programming language and software environment for using that language effectively.

The potential productivity improvement realized from these tools is dramatic. As an example, considering the simple spring-mass system of the following figure.

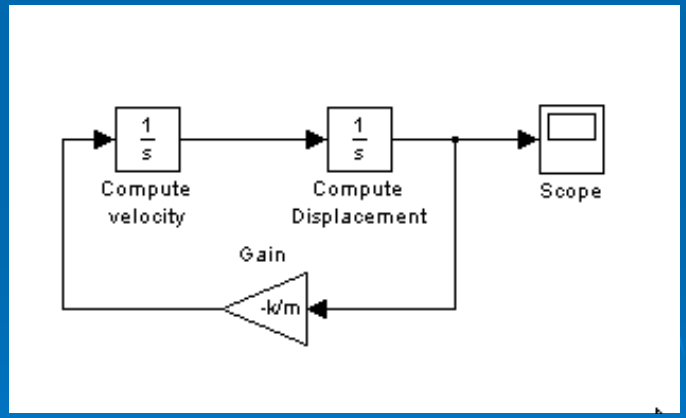
Computing Software (2)



MatLab

```
global k m
k=2;
m=1;
x1=0;
x2=2;
[t,x]=ode45(@example,[0 5],[x1 x2]);
plot(t,x(:,1),'r-');
%---------------------------------------
function xdot=example(t,x);
global k m
xdot(1)=x(2);
xdot(2)=(-k/m)*x(1);
xdot=[xdot(1);xdot(2)];
```

Simulink



Computing Software (3)

Program size Comparison

Programming Language	Lines of Code (Blocks)	Approximate Keystrokes
8086 Assembly Language	300	5000
FORTRAN	80	1200
MatLab	4	90
Simulink	4	20

MatLab

- **MatLab: Matrix Laboratory**
- **Key Features of This Software:**
  - Computation
  - Visualization
  - Programming
- **Typical uses:**
  - Math and computation
  - Algorithm development
  - Modelling, simulation, and prototyping
  - Data analysis, exploration, and visualization
  - Scientific and engineering graphics
  - Application development, Graphical User Interface

Why use MatLab?

- Extremely powerful and simple environment for solving complex engineering problems.
- A huge range of built-in functions from simple mathematical functions to symbolic equation solvers (come from specialized Toolbox).
- Sophisticated built-in graphics from simple plotting to animated 3D graphics and Image display.
- Simple to develop complex software systems: eg: F14 flight control system.
- Typically a “prototyping environment”

Getting Started

The prompt:
>>

Matlab it is an interactive language much like Basic. What you type at the screen is what you get. However, these are not compiled like C, Java or Fortran.

Works with basic numbers as a calculator:
>>10*5
>>5/10

Also variable assignment
>>a=10
>>b=5
>>c=a*b

Most obvious things work:
>>a=2.99e8
>>b=5+7
>>c=a/b

Basic Calculations

Basic Maths operations are as expected:

```
>>a=2
>>b=300
>>c=b^a
>>d=a+b
>>e=(a*b +c)/d
```

There are many basic built-in functions in Matlab:

```
>>f=cos(a)+sin(b)
>>g=sqrt(a^2+b^2);
>> h=exp(e*a);
```

Arrays of Numbers, Vectors

Matlab treats every variable as an **array** or **Matrix**
A scalar number is really only a special case of a matrix

```
>> a=[10];
>> b=[2];
>> c=a+b
>> ans = 12
```

Matlab thinks only about matrices:
Row Vectors:

```
>> a=[10, 11];
>> b=[2, 3];
>> c=a+b
>> ans = [12 14]
```

it is a row vector!..1X2: 1 row and 2 columns

Or Column Vectors:

```
>> a=[10; 11]
>> b=[2; 3]
>> c=a+b
>> ans = [12;14]
```

it is a column vector!..2X1: 2 rows and 1 column

Vector and Matrix manipulations

```
>> A=[1 3 6; 2 7 8; 0 3 9];
A =
    1    3    6
    2    7    8
    0    3    9
>>size(A)
ans =
    3    3
```

Transpose of A:

```
>>A'
ans =
    1    2    0
    3    7    3
    6    8    9
```

Column or row components:

```
>>A(:,3)
ans =
    6
    8
    9
>>A(2,:)
ans =
  2    7    8
```

Matrices and Arrays in Matlab

Another general arrays (matrices)

```
>> a=[1, 2; 3, 4; 5, 6]
ans
      1 2
      3 4
      5 6
```

```
>> b=[6, 5; 4, 3; 2, 1]
ans
      6 5
      4 3
      2 1
```

```
>> c=a+b
ans  =
    7    7
    7    7
    7    7
```

Note: writing in **C** this simple addition between the two matrices it will be:

```
main()
{
int i,j, c[3][3]
for (i=0; i<3;++i)
for (j=0; j<3; ++j)
c[i][j]=a[i][j]+b[i][j]
}
```

Matrices and Arrays in Matlab (2)

However

```
>> a=[10; 11]
>> b=[2, 3]
>> c=a+b
```

ERROR! Matrix Dimensions must agree!

To add or subtract matrices, arrays must have compatible dimensions.

Matrices and Arrays in Matlab (3)

Matrix subtraction:

>>c=a-b

Matrix multiplication:

>>c=a*b matrix dimensions must agree

Generally:

1. To add two matrices, they must have the same number of rows and columns;
2. To multiply two matrices "**a*b**", **a** must have the number of columns as there are rows of **b**.

Matrices and Arrays in Matlab (4)

Matrix division (matrix inversion) is non-trivial so be careful about typing
>>c=a/b
when a and b are matrices.

If, what you really wanted was all the elements in a to be multiplied (divided) by all the elements in b, then MatLab has a special syntax:
>>c=a.*b
>>d=a./b

Note the ‘dot’. In this case a and b must have the same dimensions.

Matrix functions

Matrix inverse:
>>inv(a)

Determinant of a matrix:
>>d=det(a)

Rank of a matrix:
>>rank(a)

Condition number of a matrix:
>>cond(a)

Matrix of random numbers:
>>rand(3,3) produce a matrix 3X3 of random numbers

Zero matrix:
>>zeros(3,4)

Identity matrix:
>>eye(3)

Systems of Linear Equations

Solving ordinary differential equations by means Finite Difference Method (FDM) or Finite Element Method (FEM) involve to find solutions of systems of linear or non linear equations. The typical form of a linear system of algebraic equations is written as:

$$\mathbf{A}x=b$$

and the solution is obtained by:

>> *x*=inv(**A**)**b*

or

>> *x*=**A***b*

Note:

It takes almost two and one half times as long to compute the solution with x = inv(A)*b as with x = A\b. The first, calculate the inverse of the Matrix A instead of the second use Gaussian elimination to find x.

Systems of Linear Equations (2)

Solution example:

$$\begin{aligned} 10x + 9y + 6z &= 1 \\ 7x + 8y - 3z &= -3 \\ -x + 10y - 4z &= 9 \end{aligned}$$

$$\begin{bmatrix} 10 & 9 & 6 \\ 7 & 8 & -3 \\ -1 & 10 & -4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ 9 \end{bmatrix}$$

In MatLab:

```
>> A=[10 9 6; 7 8 -3; -1 10 -4];
>> b=[1;-3;9];
>> tic, x=inv(A)*b,toc;
x =
-1.3274
1.0743
0.7675
Elapsed time is 0.120000 seconds
>> tic, x=A\b,toc;
x =
-1.3274
1.0743
0.7675
Elapsed time is 0.050000 seconds.
```

Nonlinear Algebraic Equations

Nonlinear algebraic equations are frequently adopted in many different areas. In MatLab there are two main function to solve nonlinear equations: fzero and fsolve (in Optimization Toolbox):

- **fzero('func',x_0)** tries to find a zero of the function 'func' near x_0;
- **fsolve**('func',x_0) tries to solve a system of nonlinear equations 'func' of the type: F(x)=0 for x, where x is a vector and F(x) is a function that returns a vector value, starting from x_0.

Nonlinear Algebraic Equations (2)

```
>> fzero('(x^3)+x*cos(x)-4*x',-5)
ans =
-2.1281
>> fzero('cos(x)-x^2',0)
ans =
-0.8241
```

Nonlinear Algebraic Equations (3)

This example finds a zero of the system of two equations and two unknowns:

$$\begin{cases} x_1 - x_2 = e^{-x_1} \\ -x_1 + x_2 = e^{-x_2} \end{cases} \Rightarrow F(x)=0 \Rightarrow \begin{cases} x_1 - x_2 - e^{-x_1} = 0 \\ -x_1 + x_2 - e^{-x_2} = 0 \end{cases}$$

```
function F = fexample(x)

F=[x(1)-x(2)-exp(x(1));
   -x(1)+x(2)-exp(x(2))];
```

Nonlinear Algebraic Equations (4)

Now we are ready to find the solution of our systems, but we have to indicate the starting point of the non linear routine to solve the problem:

```
>>x0=[1;8];                    %starting guess point for the solution
>>options=optimset('Display','iter'); % Option to display output
>>[x,fval] = fsolve(@fexample,x0,options)  % Call optimizer
```

Nonlinear Algebraic Equations (5)

Iteration	Norm of Func-count	First-order f(x)	Trust-region step	optimality	radius
0	3	8.84452e+006		8.86e+006	1
1	6	1.19429e+006	1	1.2e+006	1
2	9	160846	1	1.62e+005	1
3	12	21556.6	1	2.2e+004	1
4	15	2868.14	1	2.98e+003	1
5	18	380.564	1	405	1
6	21	52.9237	1	55.3	1
7	24	11.5836	1	8.93	1
8	27	2.65943	1	1.72	1
9	30	0.83611	1.44257	0.896	2.5
10	33	0.112921	1.37225	0.3	2.5
11	36	0.0101145	1.40974	0.0356	2.5
12	39	0.00130067	1.41417	0.00181	3.52
13	42	0.000175934	1.41421	0.000103	3.52
14	45	2.38101e-005	1.41421	1.2e-005	3.52
15	48	3.22234e-006	1.41421	1.62e-006	3.52
16	51	4.36097e-007	1.41421	2.2e-007	3.52

Optimization terminated: first-order optimality is less than options.TolFun.

x =

-7.6693
-7.6693

fval =

1.0e-003 *

-0.4670
-0.4670

The m-file

Files that contain a computer code are called the *m-files*. There are two kinds of m-files: the *script files* and the *function files*:

- Script files do not take the input arguments or return the output arguments.
- The function files may take input arguments or return output arguments.

Script files

A script file is simply a listing of MatLab commands that you might otherwise have typed at the Keyboard:

Starting up the built-in MatLab editor. Type in the following:

```
t=0:0.1:1;
x=t.^2;
y=x+t;
```

Note:
The editor “recognises” the syntax. This might help you not make mistakes. Save this in a file called eg1.m (the “m” extension is the important bit).

Function files

Here is an example of the function file

```
function [b, j] = descsort(a)
% Function descsort sorts, in the descending order, a real array
% a. Second output parameter j holds a permutation used to
% obtain array b from the array a.

[b ,j] = sort(-a);
b = -b;
```

Function files (2)

This function takes one input argument, the array of real numbers, and returns a sorted array together with a permutation used to obtain array **b** from the array **a**.
MatLab built-in function **sort** is used here. Recall that this function sort numbers in the ascending order. A simple trick used here allows us to sort an array of numbers in the descending order.

To demonstrate functionality of the function under discussion let

```
>>a = [pi –10 35 0.15];
>>[b, j] = descsort(a)

b =
35.0000 3.1416 0.1500 -10.0000
j =
3 1 4 2
```

M - files (2)

These two new tools, by which we can write programs using collections of MatLab commands and/or built-in functions, are similar to other common programming languages. So if we have to solve a complex problem, we will divide the problem using “intermediate” step (subroutine or function) and then we combined the “intermediate” result to obtain the final solution: its is the “divide et conquer” algorithm (bottom up or top – down approach)

Flow Control

A basic problem now is that a script file is just a list of instructions. What we need is to be able to change the "flow" of instructions depending on the values of the variables. To do this, we need to

a) Test variables: is ***a*** greater than ***b*** ? Is ***c*** equal to 105 ? etc.
b) To do something (other than the next instruction) depending on the outcome of a test: if ***a*** is greater than ***b*** then plot ***a***, else plot ***b***. While ***c*** is greater than ***0*** subtract one.

MatLab provides both tests and flow control structures. With these we will be able to write proper "intelligent" programmes.

Relational Operators

MatLab has six relational operators

Operator	**Interpretation**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

Zero is logical false.
One is logical true.

Relational Operators (2)

Logical expressions may be formed as follows:

```
>>x=10; y=11;
>>x>y
ans= 0
>> x<y
ans = 1
>>x ~= y
ans= 1
```

As always, MatLab makes no distinction between scalars and arrays:

```
>> a=[2 4 6]; b=[3, 5, 1];
>> a<b
[1 1 0]
>>a ~= b
[1 1 1]
```

Relational operators work with any equal sized vectors or matrices.
The result of a relational operator is either true or false (1 or 0)

Logical Expressions

MatLab can also do logical manipulation (combinations of true and false)

MatLab has three logical operators:

Logical Operator	**Symbol**
and	&
or	|
not	~

Zero is taken as logical false.
Any other number (including negative numbers) are taken as logical true.

Control Flow

MatLab has three decision-making or control-flow structures:

- For Loops
- While Loops
- If-Then-Else structures

FOR LOOPS

For Loops:
A simple example

```
>>for n=1:10
      x(n)=sin(n*pi/10);
    end
```

```
n=1:10;
x=sin(n*pi/10)
```

The assignment n=1:10 is a general array assignment

Other array assignments work too:

```
>> for n=1:2:20
      y(n)=sin(n*pi/10);
     end
```

or even

```
>>data=[3 9 45 6; 7 16 -5 5]
>>for n=data
      x=n(1)-n(2);
    end
```

Generally:

```
for n=vector
            [ calculations involving nth element
            of vector]
end
```

WHILE LOOPS

For loops execute a group of statements a fixed number of times.

While loops execute a group of statements an indefinite number of time until “expression” is true:

```
while {expression}
    [Commands]
End
```

where expression is any logical expression evaluating to true or false.

IF-THEN-ELSE Statements

```
if {expression}
   [command 1]
else
   [command 2]
end
```

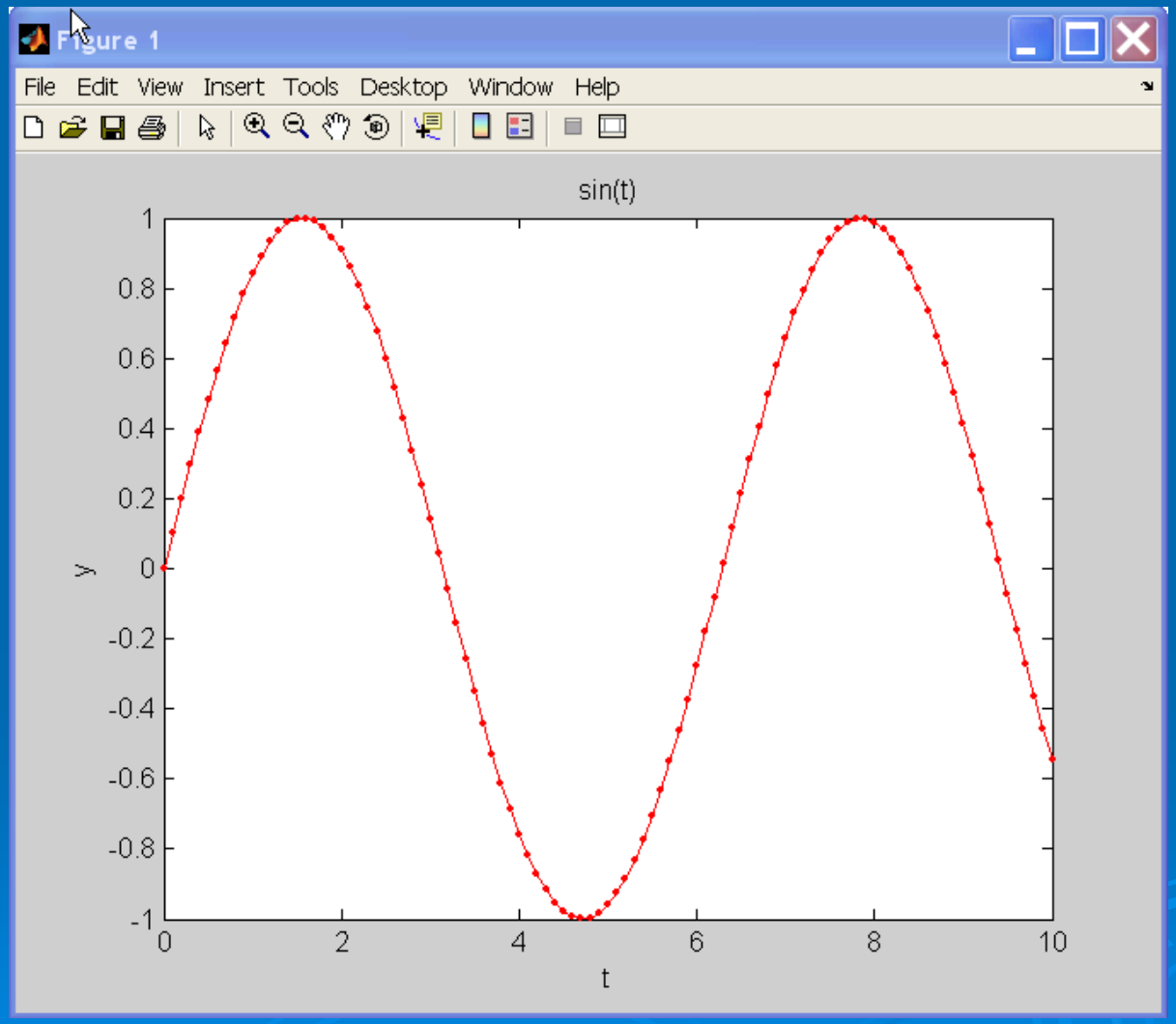
If statements, produces command1 if “expression” is valuated true otherwise produces command 2.

Basic Plotting

MatLab has extensive facilities for plotting and graphically analysing data. Here we will look at basic plotting capabilities:

```
>>t=0:0.1:10;          %form a t = vector
>>y=sin(t);            %for each t value,
                       %calculate sine
>>plot(t,y,'r.-');     %basic plot of y in function
                       %of t, also connect each value
                       %with a straight line and put a
                       %dot for each calculate value
>>title('sin(t)');     %put a title into the figure
>>xlabel('t');         %put a label into x axis
>>ylabel('y');         %put a label into y axis
```

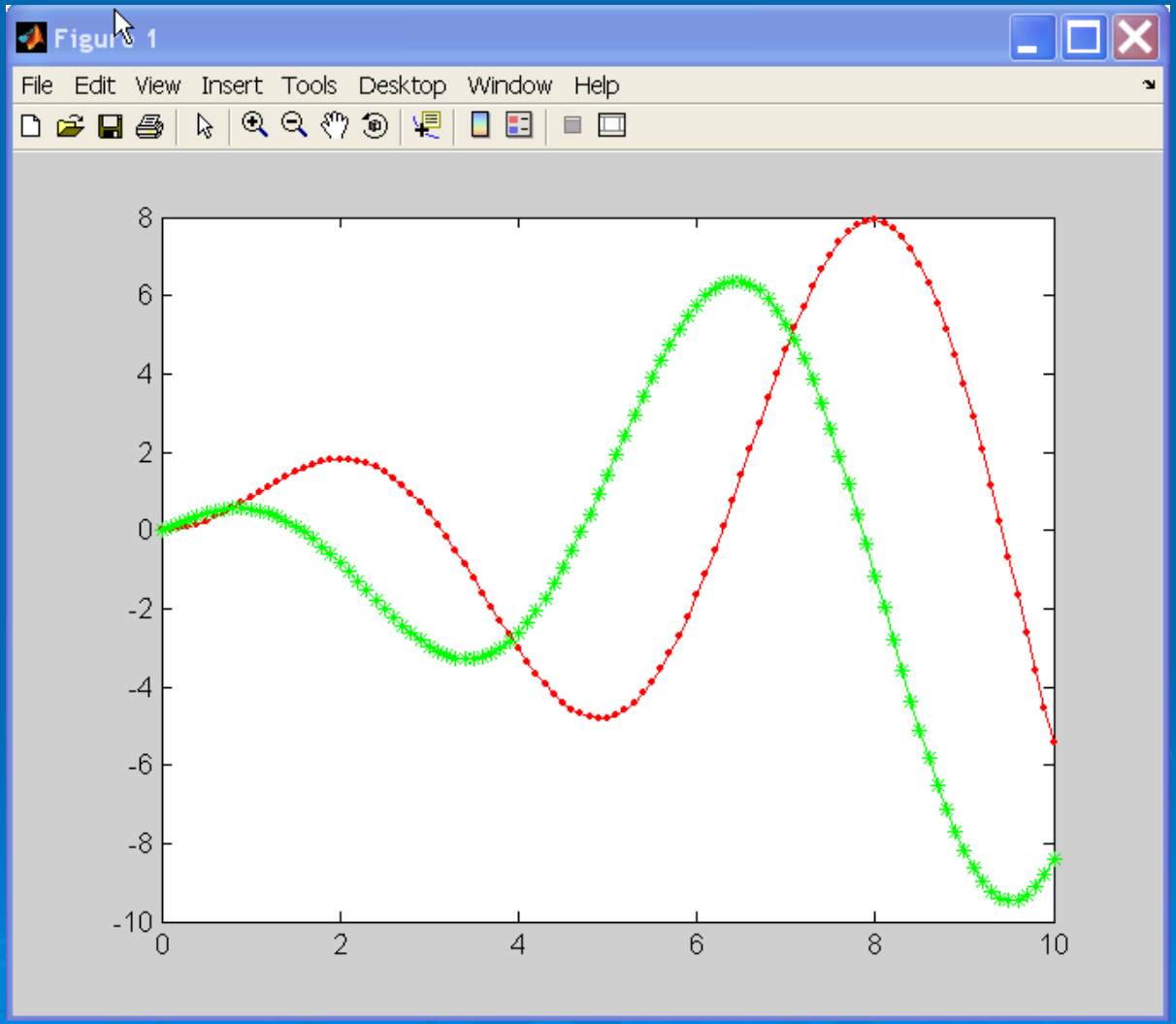
Basic Plotting (2)



Basic Plotting (3)

Plotting multiple data:

```
>>t=0:0.1:10;
>>y1=sin(t).*t;
>>y2=cos(t).*t;
>>plot(t,y1,'r.-',t,y2,'g*-');
```



Mathematical models of Systems

- Ordinary differential equations (Odes) describe phenomena that change continuously: they arise in models throughout chemistry, science, engineering, medicine, biology, anthropology and so on.
- With a differential equation, we can associate ***initial conditions*** or ***boundary conditions;***
  - If these conditions are specified at a single value of the independent variable, they are referred to as initial conditions and the combination of the differential equation and an appropriate number of initial conditions is called an ***Initial Value Problem*** (**IVP**);
  - If these conditions are specified at more than one value of the independent variable, they are referred to as boundary conditions and the combination of the differential equation and the boundary conditions is called a ***Boundary Value Problem*** (**BVP**).

Initial Value Problem

An ordinary differential equation contains one or more derivatives of a dependent variable y(t) with respect to a single independent variable t, usually referred to as *time*. Generally there are many functions y(t) that satisfy a given ODE, and additional information is necessary to specify the solution of interest.

In an *initial value problem,* the solution of interest satisfies a specific *initial condition*, that is, y is equal to $\mathbf{y_0}$ at a given initial time $\mathbf{t_0}$. An initial value problem for an ODE is then:

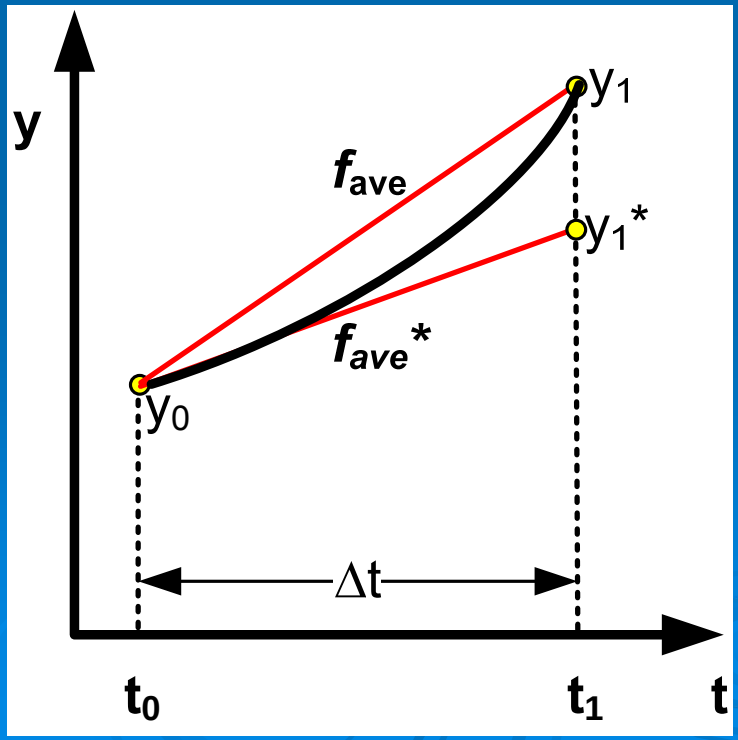
$$y^n(t) = F[t, y(t), y'(t)..y^{n-1}(t)]$$

$$y(t_0) = a_0, y'(t_0) = a_1, ...y^{n-1}(t_0) = a_{n-1}$$

If the function F[t,y(t),y'(t),..,yn-1(t)] is sufficiently smooth, this problem has one and only one solution. Generally there is no analytic expression for the solution, so it is necessary to approximate by numerical means, such as using one of the ODE solvers.

Initial Value Problem (2)

Consider the differential equation $\frac{dy}{dt} = f(t, y)$ subject to $y(t_0)=y_0$. The approach to integrate the differential equation numerically is to start at $t=t_0$, where we know the solution, $y=y_0$. Then we move to $t_1=t_0+\Delta t$. The solution at that point is: $y_1=y_0+\Delta t^* f_{ave}$. Then we proceed in the same way to obtain $y(t_0+2^*\Delta t)=y(t_2)=y_2$ and so on.



Initial Value Problem (3)

It means that these methods are referred to as discrete variable methods and generate a sequence of approximate values for $y(t)$, $y_1,y_2,y_3,..y_n$, at points $t_1,t_2,t_3,..t_n$. In our development, we will assume a constant spacing **Delta**=t_{i+1}-t_i between t points. In realistic implementations of these methods, however, **Delta** is chosen to satisfy a user-specified accuracy request.

Errors enter into the numerical solution of IVPs from two sources:

1. *Discretization error* and depends on the method being used;
2. *Computational error* which includes such things as *roundoff error*, the error in evaluating implicit formulas, etc;

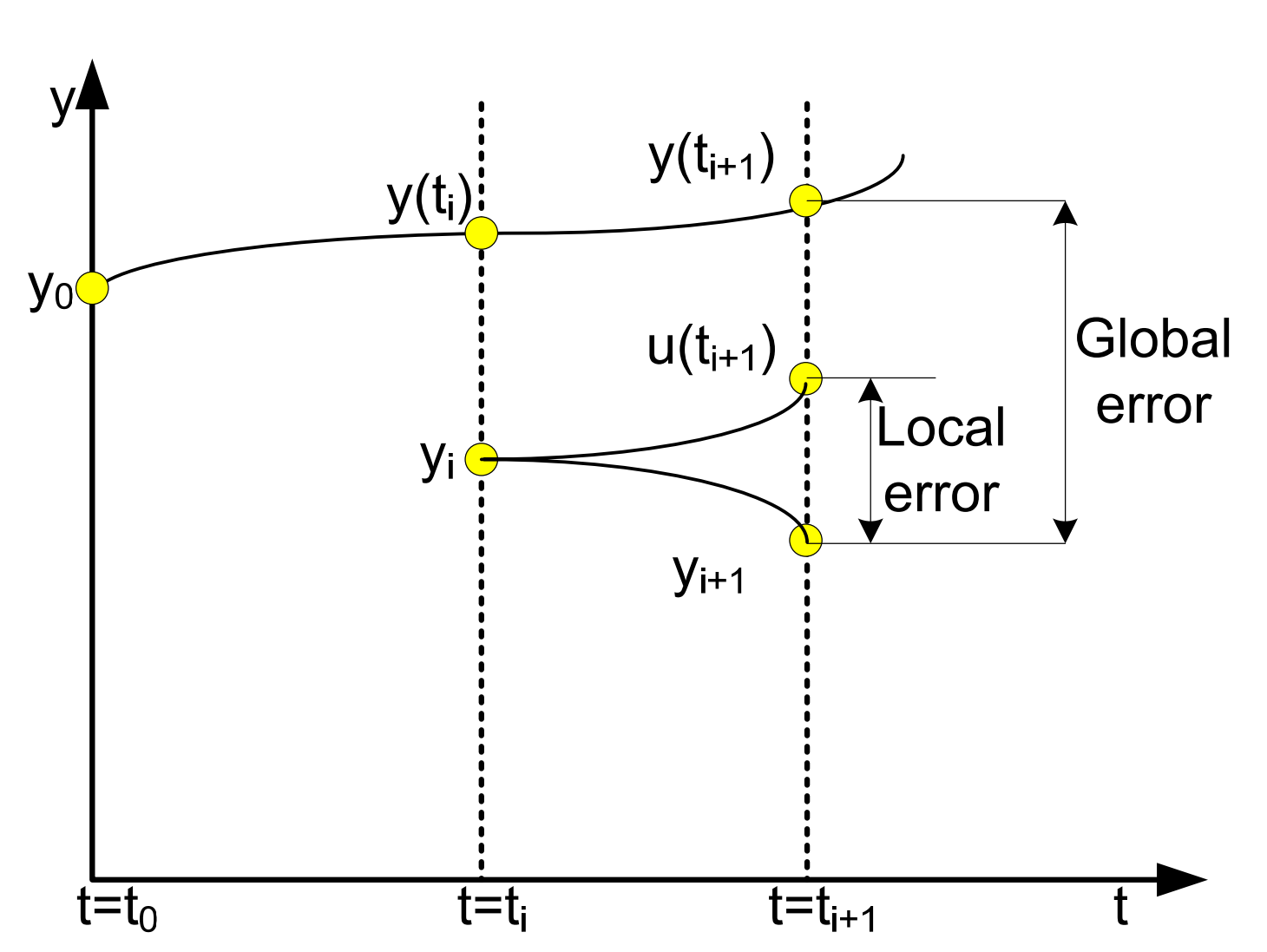
Initial Value Problem (4)

If we denote with $y(t_i)$ the exact solution of $\frac{dy}{dt} = f(t,y)$ at t_i and with y_i the numerical approximation of $y(t_i)$, we can distinguish two type of discretization error:

- Global error
- Local error

Global error: $e_{i+1}=y(t_{i+1})-y_{i+1}$
Local error: is the error that we take in a single step.
Letting u(t) the solution of $\frac{du(t)}{dt} = f(t,u)$ with $u(t_i)=y_i$ the local error is:
$d_{i+1}=u(t_{i+1})-y_{i+1}$.

Initial Value Problem (5)



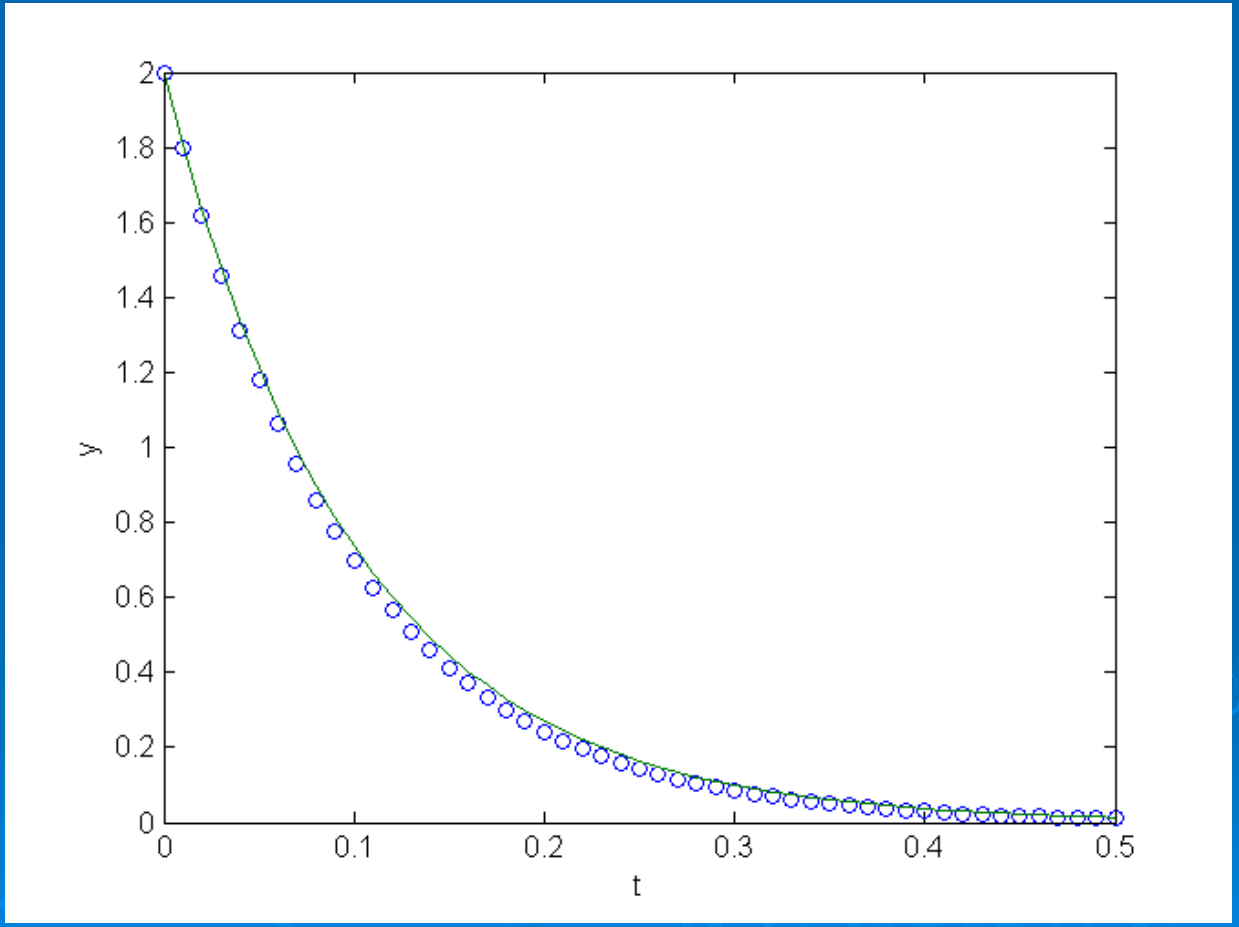
Initial Value Problem (6)

The simplest method is Euler's Method. In Euler's method, to estimate y_{i+1}, f_{ave} is approximated by its value at (t_i,y_i), that is $f(t_i,y_i)$, or its value at (t_{i+1},y_i), $f(t_{i+1},y_i)$. The former case is called the forward Euler Method and the latter case is called the backward Euler Method.

Initial Value Problem (7)

$$\begin{cases} \dfrac{dy}{dt} = r(t)y(t) \\ y(0) = y_0 \end{cases}$$

```
r=-10; delta=0.04; y(1)=2;
k=0;
for time = delta:delta:0.5
    k=k+1;
    y(k+1)=y(k)+delta*(r*y(k));
end
t=[0:delta:0.5];
y_true=2*exp(-10*t);
plot(t,y,'o',t,y_true);
xlabel('t');
ylabel('y');
```



Initial Value Problem (8)

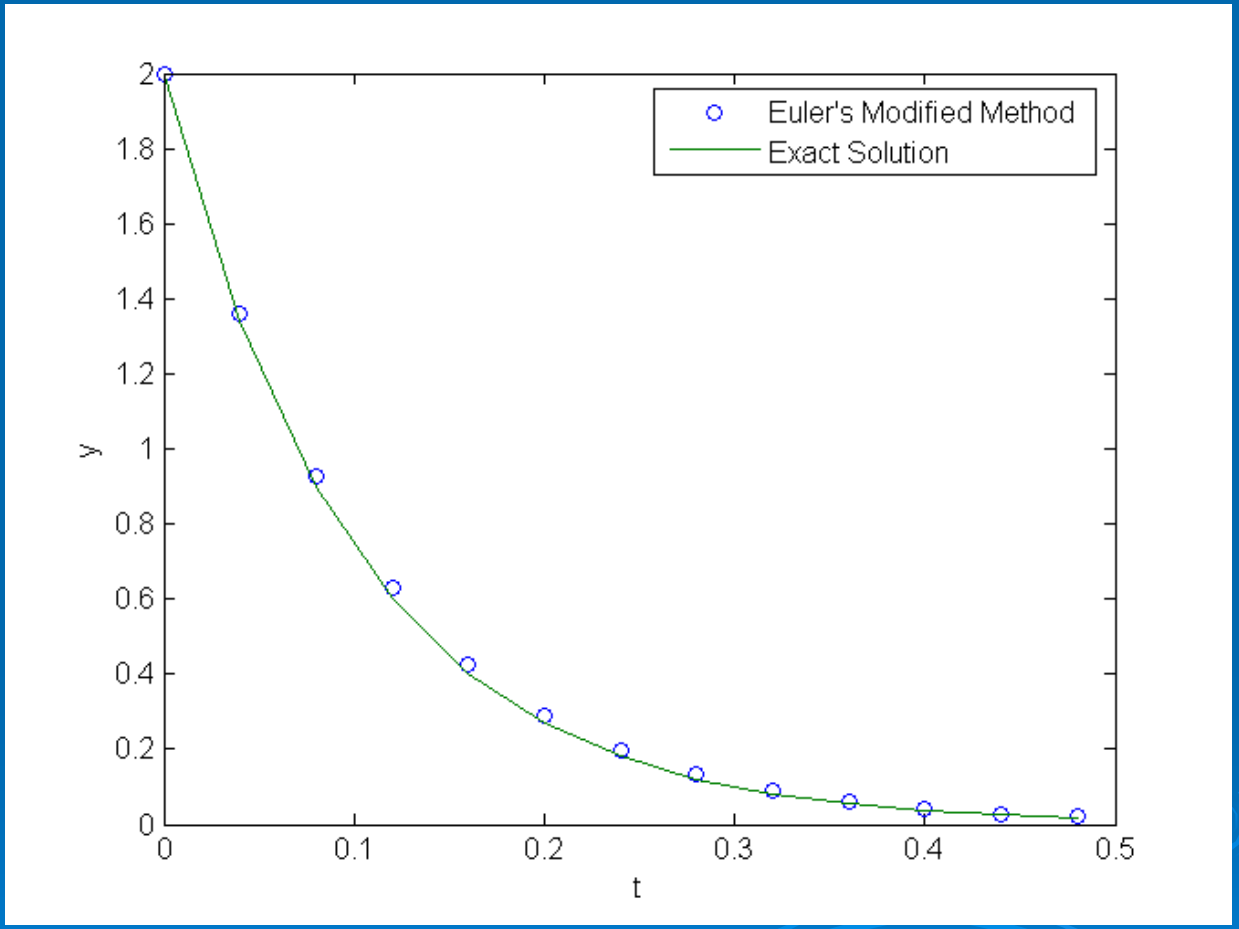
The forward Euler method uses the derivative at the initial point of the interval.
The backward Euler method uses the derivative close to the end point of the interval.
A numerical solution with improved accuracy is obtained by using the average of the initially computed end points.

The algorithm steps are:

1. Evaluate the slope at the start of the interval, that is $f(t_i,y_i)$;
2. Estimate yi+1 at the end of the interval using Euler's Method: $y_{i+1}=y_i+\Delta t*f(t_i,y_i)$;
3. Evaluate the slope at the end of the interval: $f(t_{i+1},y_i+\Delta t*f(t_i,y_i))$;
4. Calculate the average of the two slopes, f_{ave} from step 1 and 3;
5. Calculate a revised value of y_{i+1} using the average slope
   $y_{i+1}=y_i+\Delta t*f_{ave}$

Initial Value Problem (9)

```
r=-10; delta=0.04; y(1)=2;
k=0;
for time = delta:delta:.5
    k=k+1;
    x(k+1)=y(k)+delta*r*y(k);
    y(k+1)=y(k)+(delta/2)*(r*y(k)+r*x(k+1));
end
t_true=0:delta:0.5;
y_true=2*exp(-10*t_true);
plot(t_true,y,'o',t_true,y_true);
xlabel('t');
ylabel('y');
```



Initial Value Problem (10)

The various ways of making the estimation of f_{ave} are called methods:

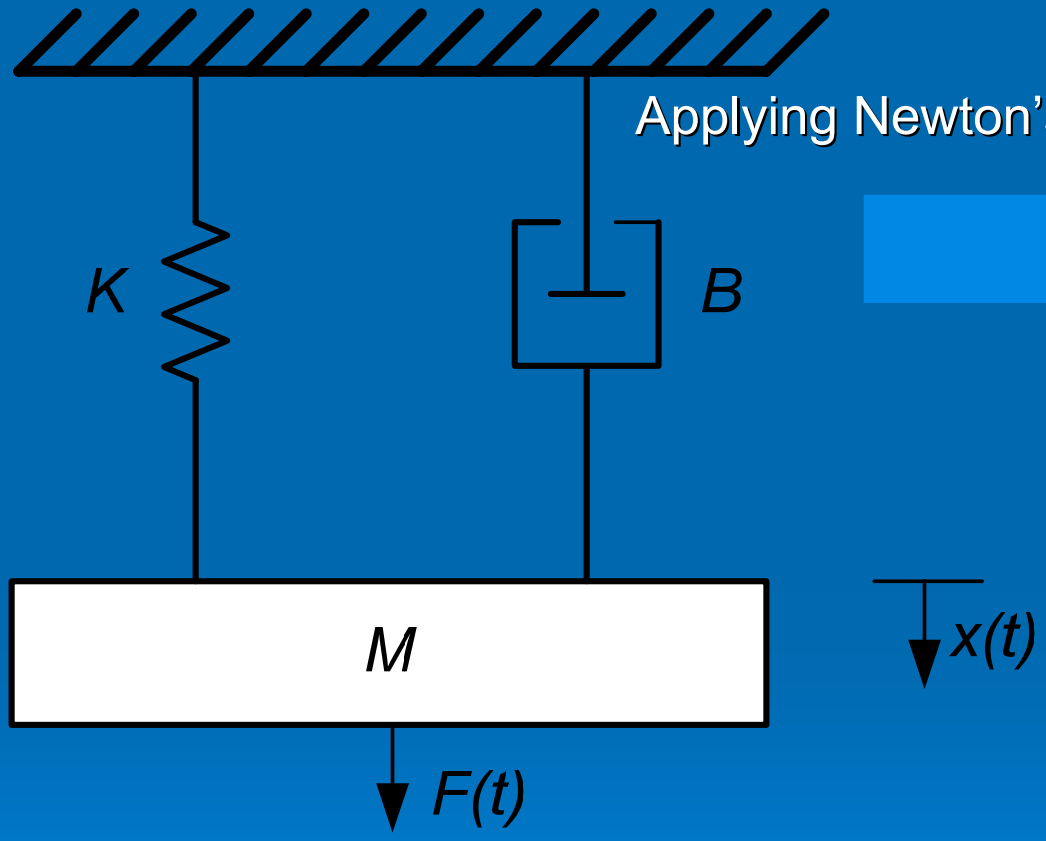
- One step Method (without memory, i.e. Eulero, Runge-Kutta)
- Multi step Method (with memory, i.e.Adams Methods)

Multistep methods tend to be more efficient than single-step methods for problems with smooth solutions and high accuracy requirements

Using MatLab to solve Odes

- MatLab provides some functions for numerical solutions of differential equations:
- For IVP problem:
  - “ode23” Runge Kutta Algorithm;
  - “ode45” Runge Kutta Algorithm;

Example: Mechanical system



Applying Newton's law of motion

$$M\frac{d^2x}{dt^2} + B\frac{dx}{dt} + Kx = F(t)$$

$$Let\ x_1 = x\ and\ x_2 = \frac{dx}{dt},\ then$$

$$\begin{cases} \dfrac{dx_1}{dt} = x_2 \\ \dfrac{dx_2}{dt} = \dfrac{1}{M}\left[F(t) - Bx_2 - Kx_1\right] \end{cases}$$

Example: Mechanical system (2)

M-Function

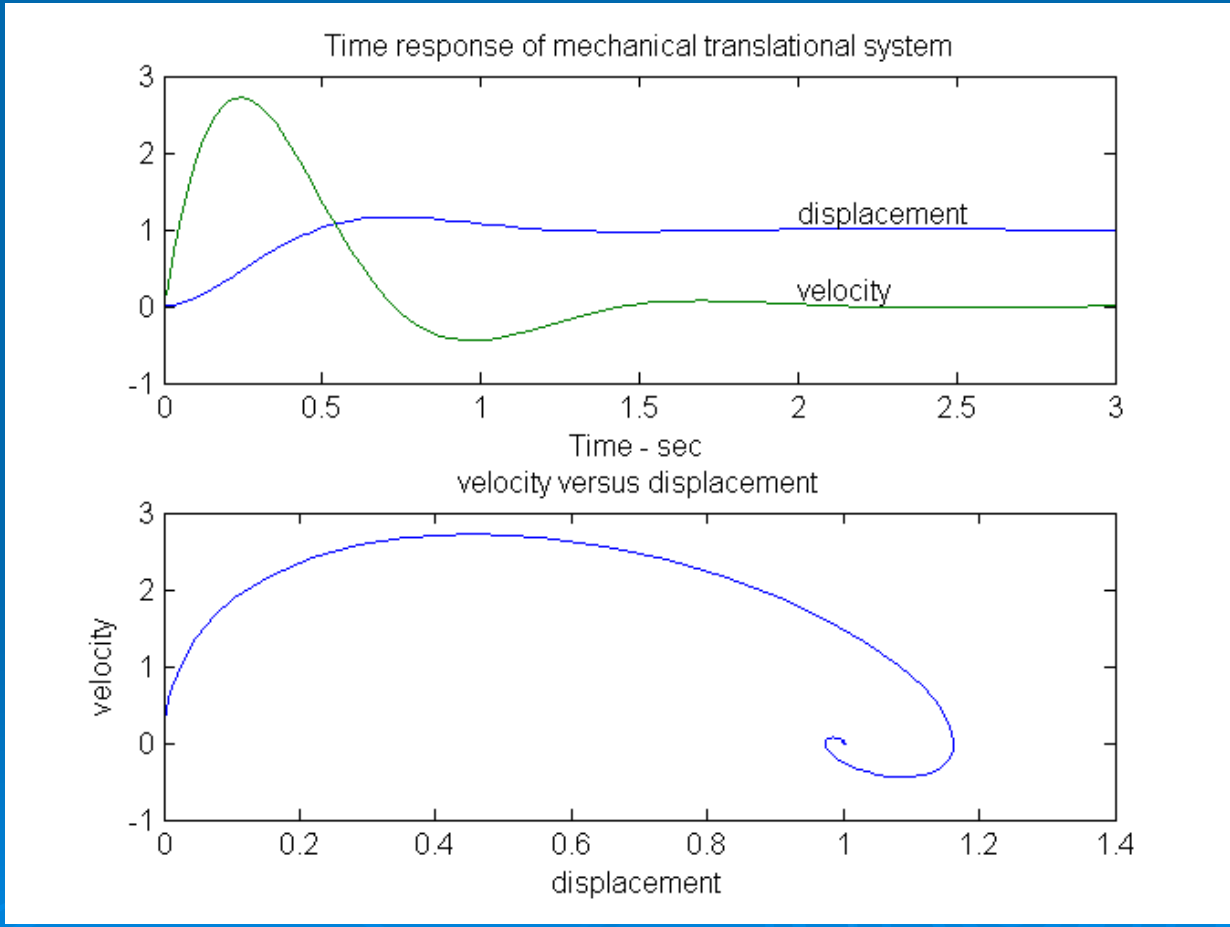
```
function xdot=mechsys(t,x,flag,M,K,B,F);

xdot(1)=x(2);
xdot(2)=(1/M)*(F-B*x(2)-K*x(1));
xdot=[xdot(1);xdot(2)];
```

M-Script

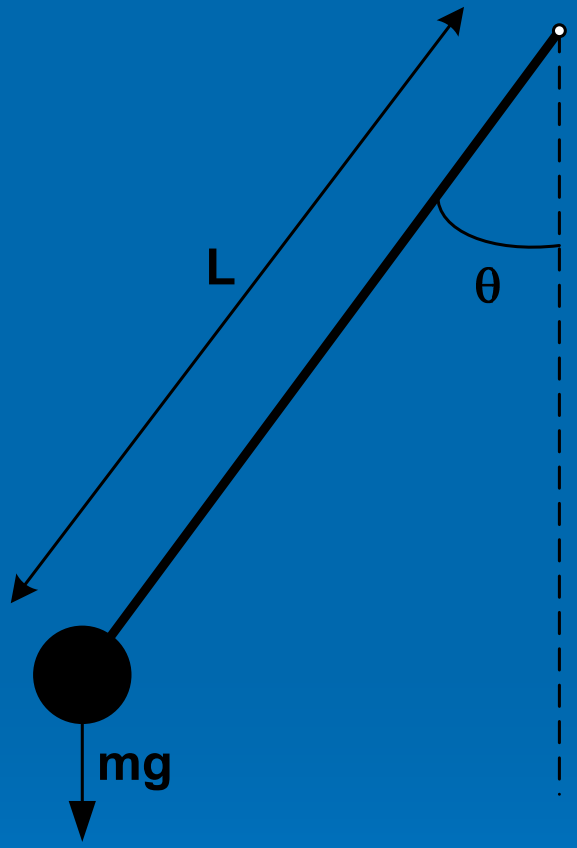
```
tspan=[0,3];          % time interval
x0=[0,0];             % initial conditions
M=1;                  % Mass value Kg
K=25;                 % Elastic Constant N/m
B=5;                  % Damping coefficent Ns/m
F=25;                 % Force

[t,x]=ode45('mechsys',tspan,x0,[],M,K,B,F);
```

Example: Mechanical system (3)

```
subplot(2,1,1),plot(t,x);
title('Time response of mechanical translational system');
xlabel('Time - sec');
text(2,1.2,'displacement');
text(2,.2,'velocity');

d=x(:,1);
v=x(:,2);
subplot(2,1,2);
plot(d,v);
title('velocity versus displacement');
xlabel('displacement');
ylabel('velocity');
```



Example: Pendulum



$$F_T = -mg\sin(\theta(t)) - BL\dot{\theta}(t)$$

Newton's law

$$F_T = mL\ddot{\theta}(t)$$

$$mL\ddot{\theta}(t) + BL\dot{\theta}(t) + mg\sin(\theta(t)) = 0$$

$$Let\ x_1 = \theta\ and\ x_2 = \dot{\theta},\ then$$

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -\dfrac{B}{m}x_2 - \dfrac{g}{L}\sin(x_1) \end{cases}$$

Example: Pendulum (2)

M-Function

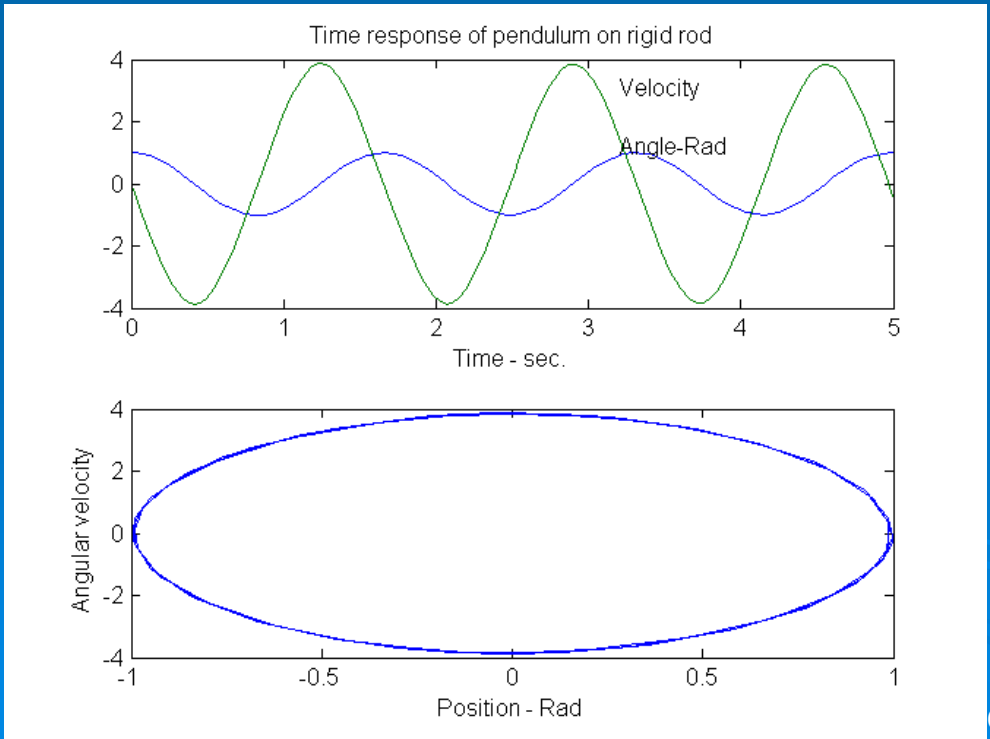
```
function xdot=pendulum(t,x,flag,m,B,L)

g=9.81;%m/s^2
xdot=[x(2)
      -(B/m)*sin(x(2))-(g/L)*sin(x(1))];
```

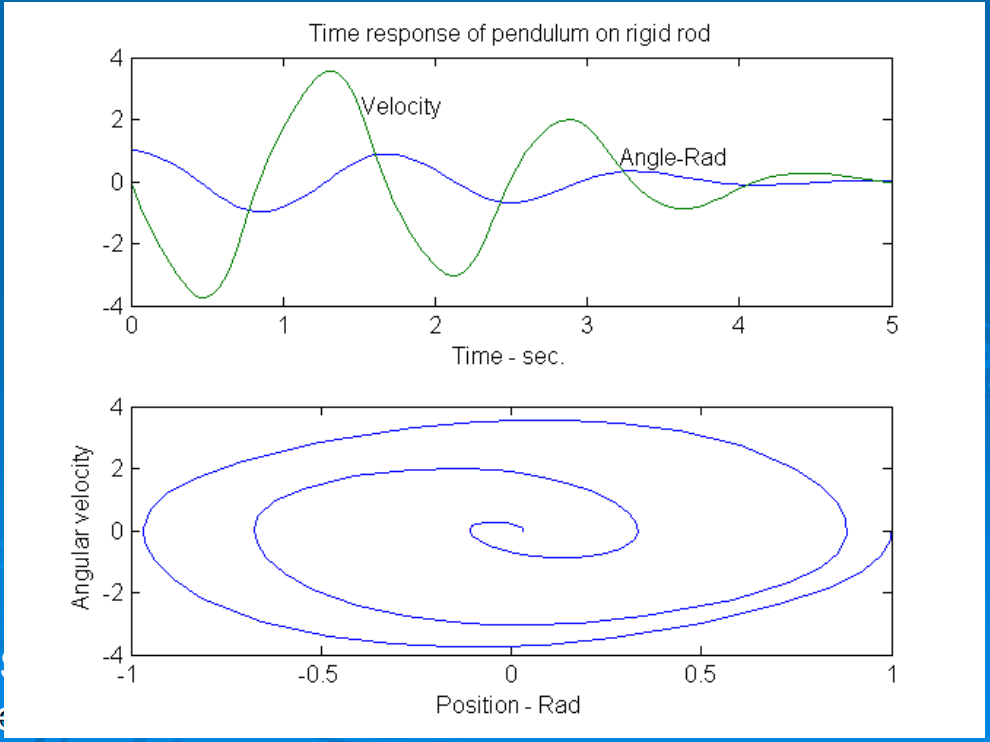
M-Script

```
tspan=[0 5];
x0=[1 0];
m=2/9.81;
L=0.6;
B=0.02
[t,x]=ode23('pendulum',tspan,x0,[],m,B,L);
```

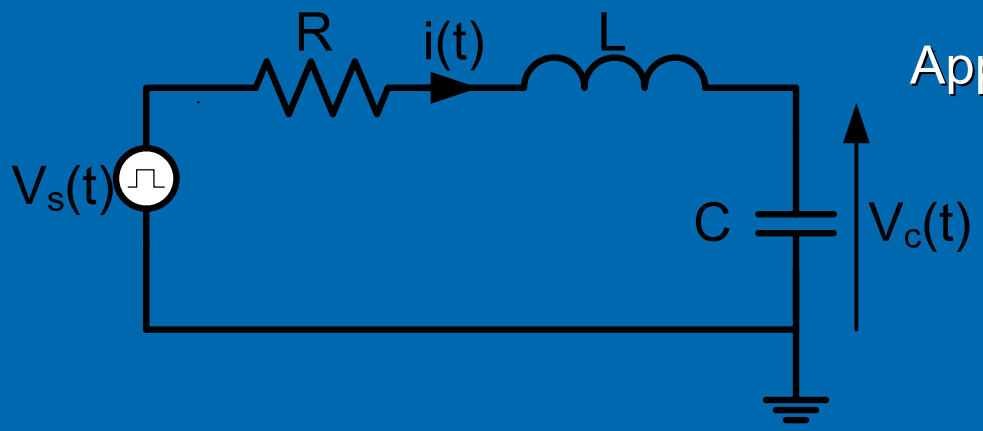
Example: Pendulum (3)

```
subplot(2,1,1),plot(t,x);
title('Time response of pendulum on rigid rod');
xlabel('Time - sec.');
text(3.2,3.1,'Velocity');
text(3.2,1.2,'Angle-Rad');

th=x(:,1);
w=x(:,2);
subplot(2,1,2);
plot(th,w);
xlabel('Position - Rad'),ylabel('Angular velocity');
```





Example: RLC circuit



Applying KVL

$$Ri(t) + L\frac{di(t)}{dt} + V_c(t) = V_s(t)$$

having

$$i(t) = C\frac{dV_c(t)}{dt}$$

with initial condition

$$i(0) = 0A$$

$$V_c(0) = 0.5\ Volts$$

Let $x_1 = Vc,\ x_2 = i;$

$$\begin{cases} \dot{x}_1 = \dfrac{1}{C}x_2 \\ \dot{x}_2 = \dfrac{1}{L}(V_s - x_1 - Rx_2) \end{cases}$$

Example: RLC circuit (2)

M-Function

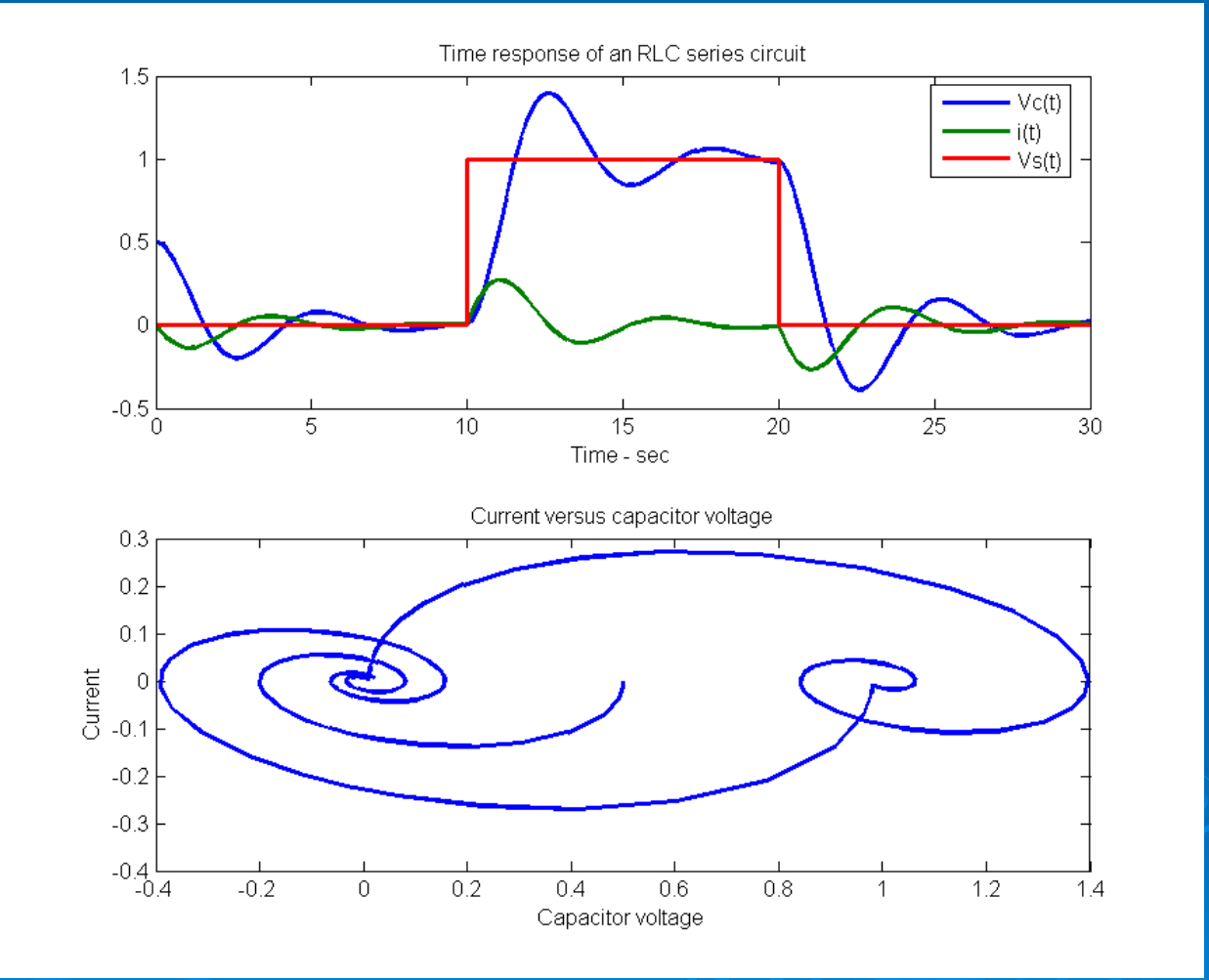
```
function xdot=electsys(t,x,flag,R,L,C);
if t<10
    V=0;
elseif t<=20
    V=1;
else
    V=0;
end
xdot=[x(2)/C;(1/L)*(V-x(1)-R*x(2))];
```

M-Script

```
x0=[0.5,0];
tspan=[0,30];
R=1.4;L=2;C=0.32;
[t,x]=ode23('electsys',tspan,x0,[],R,L,C);
```

Example: RLC circuit (3)

```
t1=find(t<10);
t2=find((t<=20)&(t>=10));
t3=find(t>20);
V=zeros(length(t),1);
V(t2)=1;
subplot(2,1,1),plot(t,x,t,V);
title('Time response of an RLC series circuit');
xlabel('Time - sec');

vc=x(:,1);i=x(:,2);
subplot(2,1,2),plot(vc,i)
title('Current versus capacitor voltage');
xlabel('Capacitor voltage');
ylabel('Current');
```



Boundary Value Problem

A Boundary Value problem specifies values or equations for solution components at more than one point x_i. Hence, unlike IVPs, a boundary problem may not have a solution, or may have a finite number, or may have infinitely number.

Because of this, programs for solving BVPs require users to provide a guess for the solution desired.

Boundary Value problem (2)

Numerical methods that solve BVPs can be classified in two main categories:

- Methods that are specific for one-dimensional problems: **Shooting Methods**;
- Methods that are independent from the dimension of the problem to solve: **Finite Difference Methods (FDM)** and **Finite Elements Methods (FEM)**;

Shooting – Methods

The idea is to transform the BVP into an equivalent IVP, having an unknown s parameters that solve another problem.

For example consider:

$$y'' = f(x, y, y')$$
$$y(a) = \alpha, y(b) = \beta$$

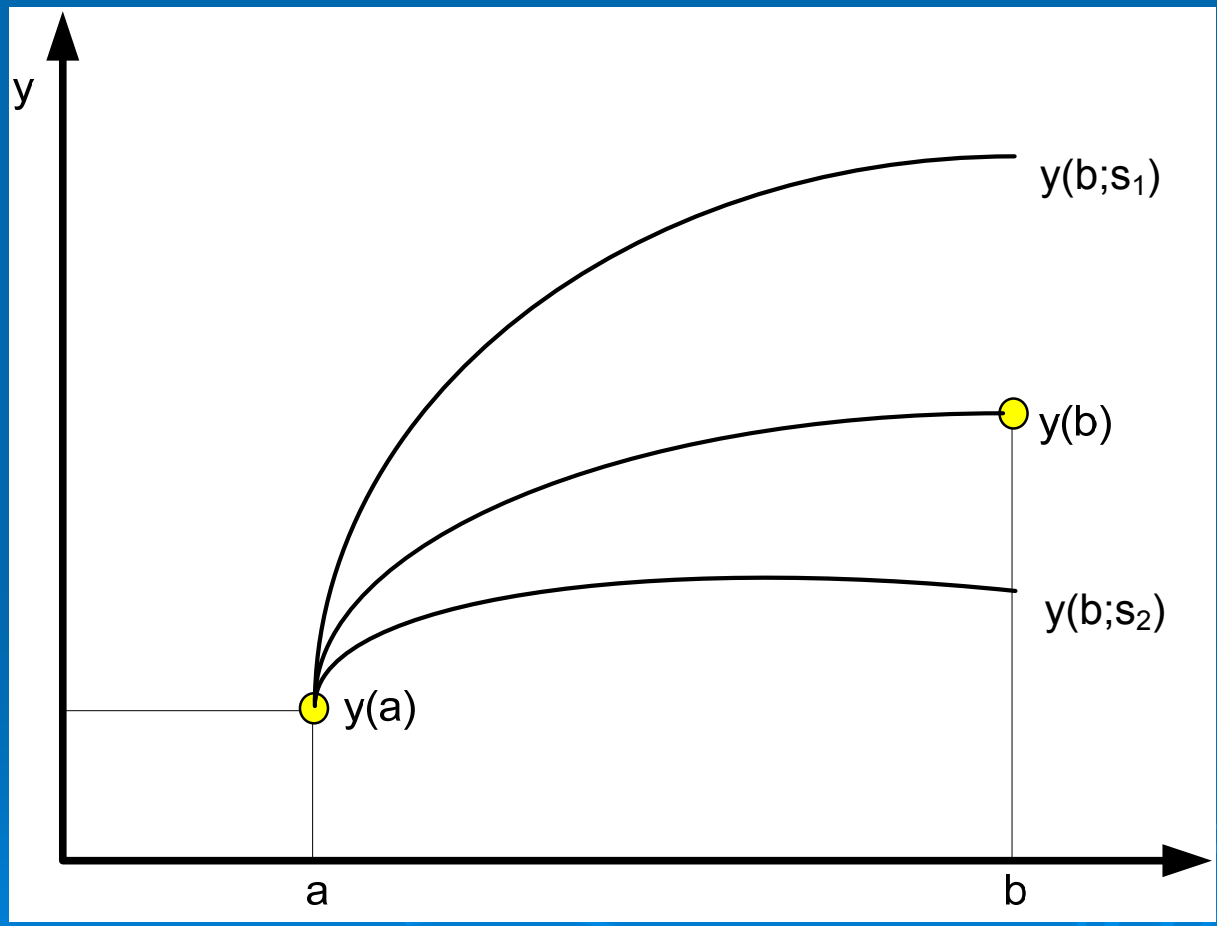
IVP

$$y'' = f(x, y, y')$$
$$y(a) = \alpha, y'(a) = s$$

$$s \to y(x; s)$$

$$F(s) := y(b; s) - \beta = 0$$

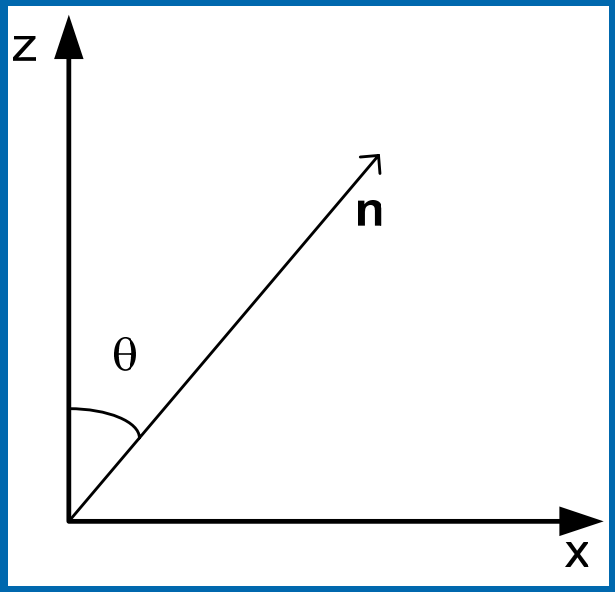
Shooting – Methods (2)



Example- Freedericksz Transition (1)

The process of the electric field-induced reorientation of the director, inside a nematic liquid cell is called Freedericksz Transition. All commercial liquid crystal display are based on this phenomenon.

Example- Freedericksz Transition (2)



$$\begin{cases} n_x = \sin(\theta(z)); \\ n_y = 0; \\ n_z = \cos(\theta(z)); \end{cases}$$

Applying the elastic continuum theory

$$\xi_E{}^2 \frac{d^2\theta(z)}{dz^2} + \sin(\theta(z))\cos(\theta(z)) = 0 \quad with \quad \xi_E = \frac{1}{E}\sqrt{\frac{4\pi K}{|\varepsilon_a|}}$$

with boundary conditions:

$$\begin{cases} \theta(0) = 0; \\ \theta(d) = 0; \end{cases}$$

Example- Freedericksz Transition (3)

Rewriting the second order differential equation as a system of two first order differential equation, and let $y_1 = \theta(z) \;\; and \;\; y_2 = \dot{\theta}(z)$ we obtain:

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = -\dfrac{1}{2\xi_E{}^2}\sin(2y_1) \end{cases} \; with \; \begin{cases} y_1(0) = 0 \\ y_1(d) = 0 \end{cases}$$

Example- Freedericksz Transition (4)

Defining MatLab function:

```
function dydx=free_ode(x,y)
global z
dydx=[y(2)
   -z*sin(2*y(1))];        %ea<0
```

Mathematical model

```
function res=free_bc(ya,yb)
res=[ya(1)
   yb(1)];
```

Boundary constraints

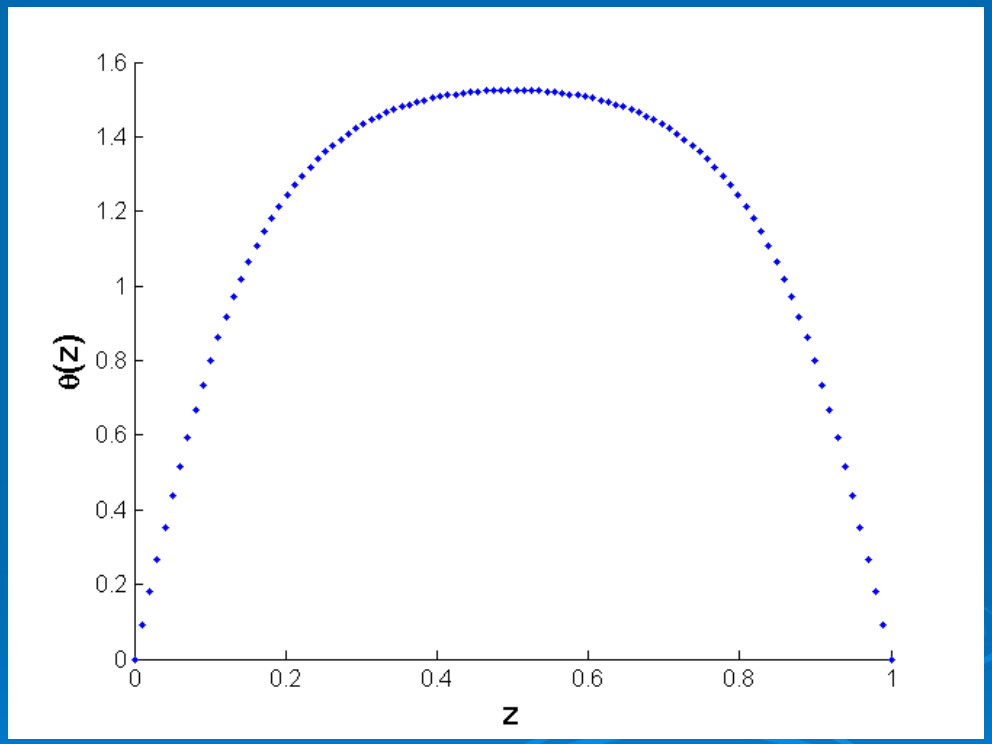
Example- Freedericksz Transition (5)

```
function [soly,yc]=lc_freedericksz_iter

solinit=bvpinit(linspace(0,1,10),[1 0]);      %Initial Gauss Solution
sol=bvp4c(@free_ode,@free_bc,solinit);        %Solver

x=linspace(0,1);                              %Building of a vector of
                                              %100 equally spaced points between
                                              %0 and 1
y=deval(sol,x);                               %Evaluate the numerical solution “sol”
                                              %in the x vector points

soly=y(1,:);
yc=deval(sol,0.5,1);                          %Evaluate the solution in the middle of
                                              %the cell
```

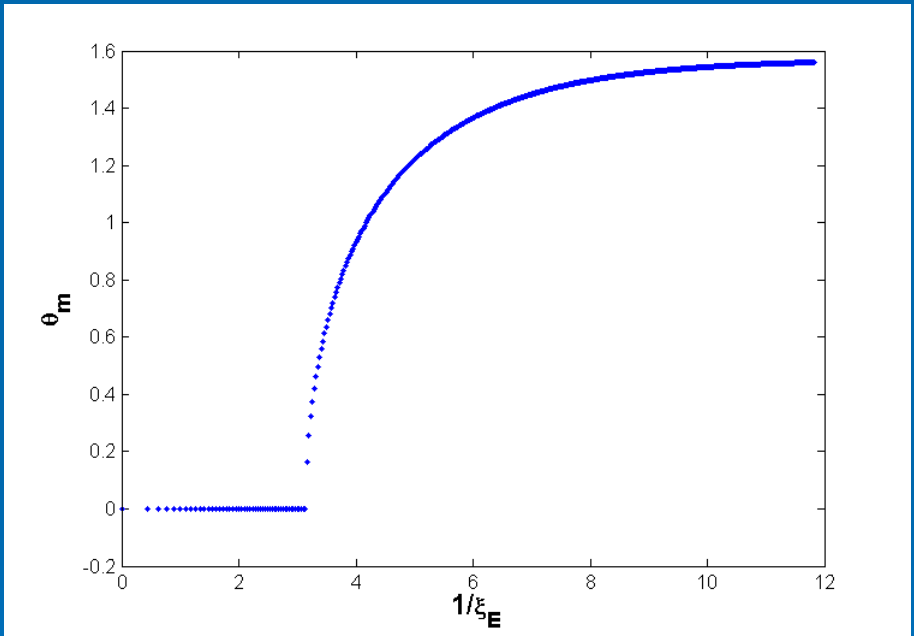
Example- Freedericksz Transition (6)

M-file that call the functions defined before:

```
global z;
zi=0:.1:70;
x=linspace(0,1);
set(gca,'nextplot','replacechildren');
h = waitbar(0,'Please wait...');
for i=1:length(zi)
    waitbar(i/length(zi),h)
    z=zi(i);
    [s(:,i),yc(i)]=lc_freedericksz_iter;
    %subplot(length(zi),1,i)
    plot(x,s(:,i),'b.');
    axis([0 1 0 1.6])
    S(i)=getframe;
end
close(h)
figure;
suze=sqrt(2*zi);
plot(suze,yc,'b.');
```



Example- Freedericksz Transition (7)



In agreement with theoretical threshold value

$$\xi_{E_c} = \frac{d}{\pi}$$

Finite Difference Method

- The computational domain is broken into points (in one dimension) or into rectangular or cubes (two or three dimensional dimension):

$$x_i = a + ih_x,\ i = 0,...,l+1 \text{ where } h_x = \frac{b-a}{l+1} \text{ and } a \le x \le b$$

$$y_j = c + jh_y,\ j = 0,...,m+1 \text{ where } h_y = \frac{d-c}{m+1} \text{ and } c \le y \le d$$

$$z_k = e + kh_z,\ k = 0,...,n+1 \text{ where } h_z = \frac{f-e}{n+1} \text{ and } e \le z \le f$$

Finite Difference Method (2)

- The derivates are then estimated on this grid. To derive a formula that can be used to estimate the derivates, we use the Taylor series expansions:

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{f''(x)}{2}(\Delta x)^2 + \frac{f'''(x)}{6}(\Delta x)^3 + \ldots (1)$$

$$f(x-\Delta x) = f(x) - f'(x)\Delta x + \frac{f''(x)}{2}(\Delta x)^2 - \frac{f'''(x)}{6}(\Delta x)^3 + \ldots (2)$$

Finite Difference Method (3)

Subtracting equation (1) from equation (2) and rearranging gives the centered difference formula of equation, which is second-order accurate:

$$f'(x) = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} - \frac{f'''(x)}{6}(\Delta x)^2 + \ldots$$

$$\approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x}$$

Finite Difference Method (4)

For second derivatives, we add equations (1) and (2). This yields the second-order accurate centered difference formula:

$$f''(x) = \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2} - \frac{f^{iv}(x)}{12}(\Delta x)^2 + \ldots$$

$$\approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2}$$

Finite Difference Method (5)

In this way we reduce the linear or non linear differential problem into a linear or non linear system of equations.

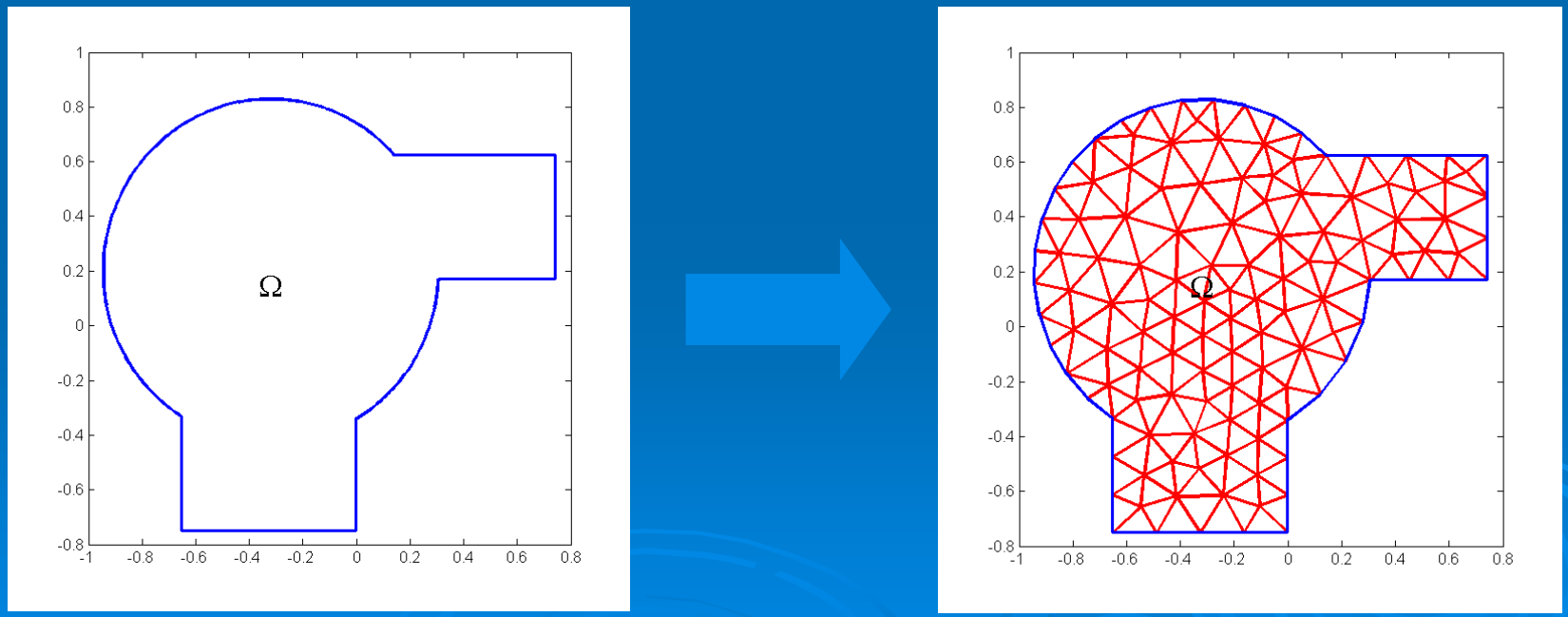
In one dimension will have:

$$\frac{d^2u(x)}{dx^2} = f(x) \rightarrow Au = b \ where$$

$$A = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

Finite Element Methods

- The domain of a BVP is divide in simple elements (*finite elements*). The vertices of the elements are called *nodes*, in which the value of the field variable is to be explicitly calculated



Finite Element Methods (2)

- The values of the field variable computed at the nodes are used to approximate the values at non nodal points (that is, in the element interior) by interpolation of the nodal values. For the three-node triangular example, the field variable inside each triangular is described by:

$$u(x,y) = N_1(x,y)u_1 + N_2(x,y)u_2 + N_3(x,y)u_3$$

where u_1, u_2, u_3 are the unknown values of the field at the nodes, and N_1,N_2,N_3 are the known interpolation functions, called *shape functions.*

FEM Example

Consider the following second order ordinary differential equation:

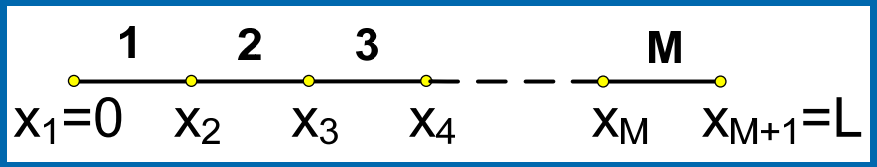
$$a\frac{d^2u}{dx^2}+b\frac{du}{dx}+cu=f(x),\ 0<x<L \qquad (1)$$

subject to boundary conditions:

$$u(0)=0 \ \ and \ \ u(L)=0$$

First, we divide the one-dimensional domain into *M elements* bounded by *M+1* values x_i of the independent variable, so that $x_1=0$ and $x_{M+1}=L$

FEM Example (2)



M elements, M+1 nodes

An approximate solution for the our Ode is assumed in the form:

$$\tilde{u}(x) = \sum_{i=1}^{M+1} N_i(x) u_i \qquad (2)$$

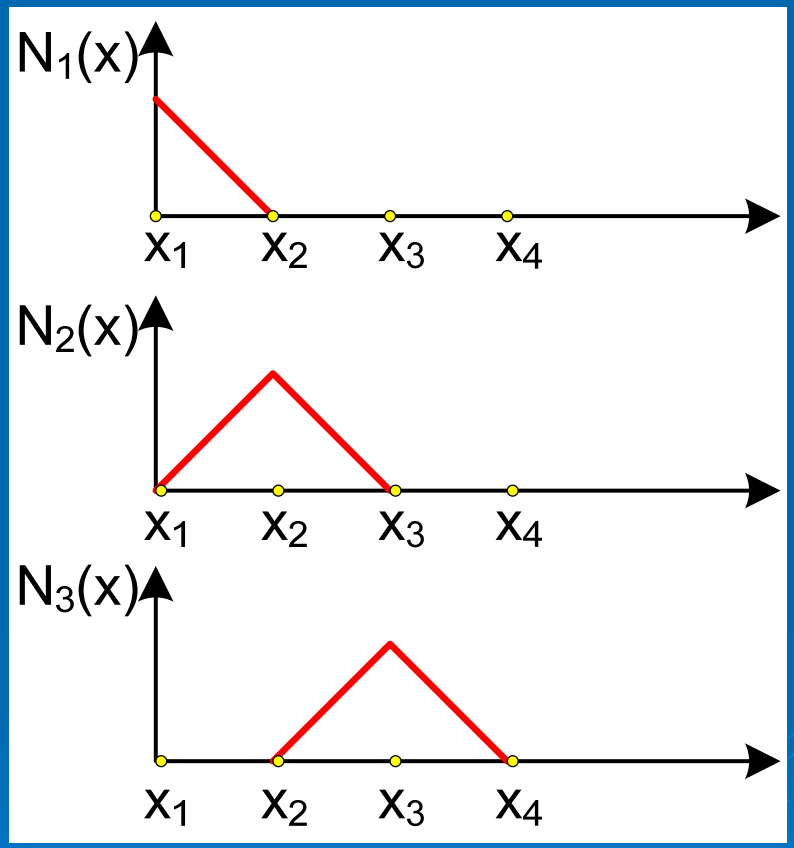
Where u_i is the value of the solution at $x=x_i$ and $N_i(x)$ is the corresponding shape function. The shape function used, $N_i(x)$ are non zero over only a small portion of the global problem domain.

FEM Example (3)

$$N_i(x) = \frac{x - x_{i-1}}{x_i - x_{i-1}} \qquad x_{i-1} \le x \le x_i$$

$$N_i(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i} \qquad x_i \le x \le x_{i+1}$$

$$N_i(x) = 0 \qquad x < x_{i-1} \quad x > x_{i+1}$$



Clearly the shape functions are simply linear interpolation functions such that the value of the solution *u(x)*, in $x_i<x<x_{i+1}$ is a linear combination of adjacent "nodal" values u_i and u_{i+1}. For example in $x_2 \le x \le x_3$:

$$\tilde{u}(x) = N_2(x)u_2 + N_3(x)u_3$$

FEM Example (4)

Substitution of the assumed solution (2) into the governing equation (1) yields the residual:

$$R(x;u_i)=\sum_{i=1}^{M+1}\left[a\frac{d^2}{dx^2}\{N_i(x)u_i\}+b\frac{d}{dx}\{N_i(x)u_i\}+c\{N_i(x)u_i\}-f(x)\right]$$

Applying Galerkin's weighted residual method, using each shape function as a weighting function, we obtain:

$$I=\int_0^L N_j(x)R(x;u_i)dx=\sum_{i=1}^{M+1}\int_{xi}^{x_{i+1}} N_j(x)*\left[a\frac{d^2\tilde{u}_e}{dx^2}+b\frac{d\tilde{u}_e}{dx}+c\tilde{u}_e-f(x)\right]dx=0 \qquad j=i,i+1 \quad (3)$$

$$\tilde{u}_e(x)=N_i(x)u_i+N_{i+1}(x)u_{i+1}$$

FEM Example (5)

$$R_e(x;u_i) = \left[a\frac{d^2\tilde{u}_e}{dx^2} + b\frac{d\tilde{u}_e}{dx} + c\tilde{u}_e - f(x) \right]$$ Residual inside each element

$$I_i = \int_{x_i}^{x_{i+1}} N_j(x) * R_e(x;u_i)dx \qquad j = i, i+1$$ Galerkin's Method inside the element

$$I = \sum_{i=1}^{M+1} I_i$$

Applying integration by parts to the first integral of (3) results in:

$$\int_0^L \left\{ -a\frac{dN_j(x)}{dx}\frac{du(x)}{dx} + bN_j(x)\frac{du(x)}{dx} + cN_j(x)u(x) \right\} dx = \int_0^L N_j(x) f(x)dx \qquad j = 1, M+1 \quad (4)$$

FEM Example (6)

Expressing the integral (4) for each element we obtain:

$$\sum_{e=1}^{M} K_e * U_e = \sum_{e=1}^{M} F^e$$

Element matrix

$$K_e = \int_{x_i}^{x_{i+1}} \left(-a \begin{Bmatrix} N_i'(x) \\ N_{i+1}'(x) \end{Bmatrix} \left[N_i'(x) N_{i+1}'(x) \right] + b \begin{Bmatrix} N_i(x) \\ N_{i+1}(x) \end{Bmatrix} \left[N_i'(x) N_{i+1}'(x) \right] + c \begin{Bmatrix} N_i(x) \\ N_{i+1}(x) \end{Bmatrix} \left[N_i(x) N_{i+1}(x) \right] \right) dx$$

Element Force vector

$$F_e = \int_{x_i}^{x_{i+1}} f(x) \begin{Bmatrix} N_{i+1}(x) \\ N_i(x) \end{Bmatrix} dx$$

Unknown Nodal vector

$$U_e = \begin{Bmatrix} u_{i+1} \\ u_i \end{Bmatrix}$$

FEM Example (5)

Hence the integration of the equation (4) yields, M+1 algebraic equations in the M+1 unknown nodal solution u_i, and these equation is rewritten in the matrix form:

$$KU=F$$

Where K is the system matrix, U is the vector of unknown nodal “displacement” and F is the vector of nodal “forces”

Finite Element Methods (3)

- The skeleton of the program structure of the finite element analysis:
  - Read input data and allocate proper array size
  - Calculate element matrices and vectors for very element
  - Assemble element matrices and vectors into the system matrix and vector
  - Apply constraints to the system matrix and vector
  - Solve the matrix equation.
  - Plot and/or print desired results.

M-file Program

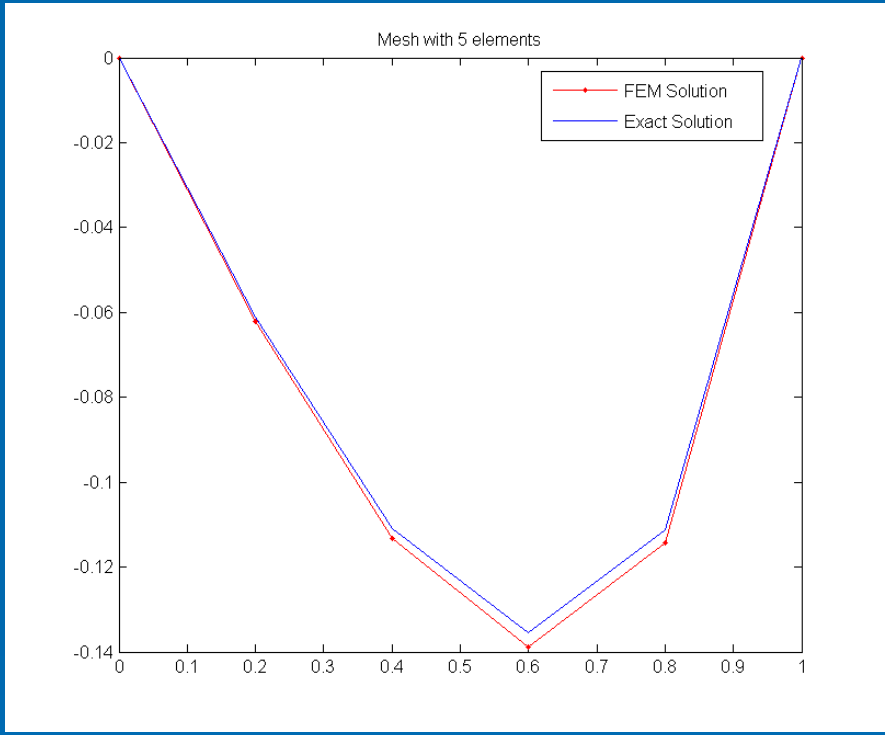
```
%-------------------------------------------------------------------------
% to solve the ordinary differential equation given as
%   a u'' + b u' + c u = 1,  0 < x < 1
%   u(0) = 0  and  u(1) = 0
% using 5 linear elements
%
% Variable descriptions
%   k = element matrix
%   f = element vector
%   kk = system matrix
%   ff = system vector
%   index = a vector containing system dofs associated with each element
%   bcdof = a vector containing dofs associated with boundary conditions
%   bcval = a vector containing boundary condition values associated with
%           the dofs in 'bcdof'
%-------------------------------------------------------------------------
clear;
Close all;
%----------------------------------
%  input data for control parameters
%----------------------------------

nel=20;                 % number of elements
nnel=2;                 % number of nodes per element
ndof=1;                 % number of dofs per node
nnode=nel+1;            % total number of nodes in system
sdof=nnode*ndof;        % total system dofs

%---------------------------------------
%  input data for nodal coordinate values
%---------------------------------------

%gcoord(1)=0.0;  gcoord(2)=0.2;  gcoord(3)=0.4;  gcoord(4)=0.6;
%gcoord(5)=0.8;  gcoord(6)=1.0;
gcoord=0.0:1/nel:1; %this are the coordinates of the nodes
%-----------------------------------------------------
%  input data for nodal connectivity for each element
%-----------------------------------------------------

%nodes(1,1)=1;  nodes(1,2)=2;  nodes(2,1)=2;  nodes(2,2)=3;
%nodes(3,1)=3;  nodes(3,2)=4;  nodes(4,1)=4;  nodes(4,2)=5;
%nodes(5,1)=5;  nodes(5,2)=6;

nodes(:,1)=1:(length(gcoord)-1);
nodes(:,2)=2:length(gcoord);
%---------------------------------------
%  input data for coefficients of the ODE
%---------------------------------------

acoef=1;            % coefficient 'a' of the diff eqn
bcoef=-3;           % coefficient 'b' of the diff eqn
ccoef=2;            % coefficient 'c' of the diff eqn
```

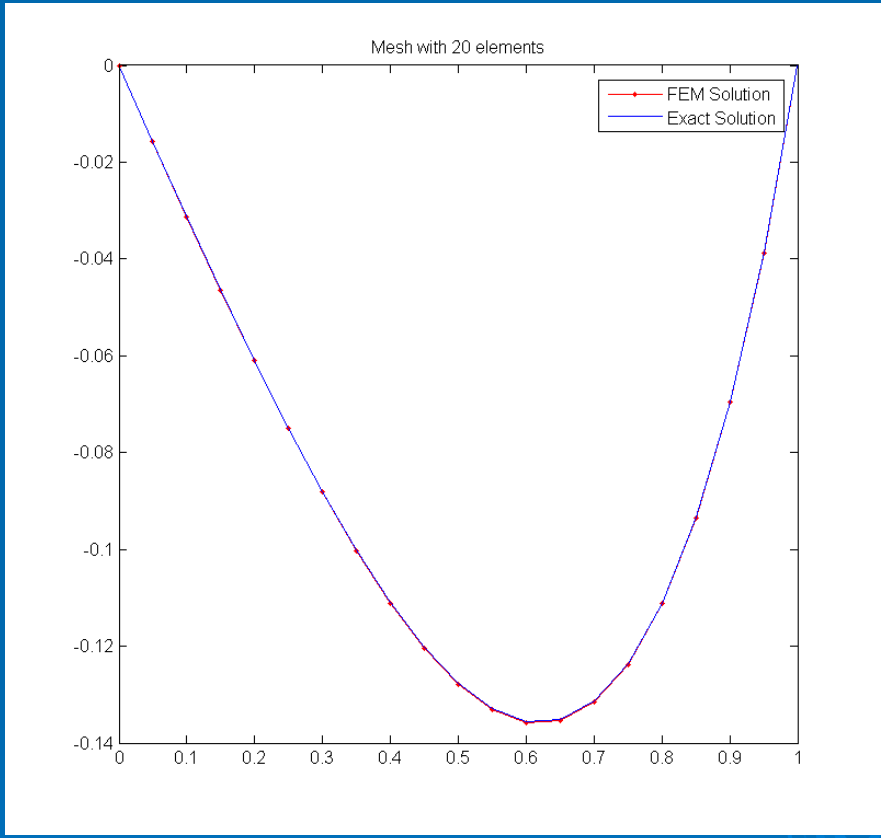
M-file Program (2)

```
%-----------------------------------
%  input data for boundary conditions
%-----------------------------------

bcdof(1)=1;             % first node is constrained
bcval(1)=0;             % whose described value is 0
bcdof(2)=nnode;         % 101th node is constrained
bcval(2)=0;             % whose described value is 0

%---------------------------------------
%  initialization of matrices and vectors
%---------------------------------------

ff=zeros(sdof,1);       % initialization of system force vector
kk=zeros(sdof,sdof);    % initialization of system matrix
index=zeros(nnel*ndof,1);  % initialization of index vector

%-----------------------------------------------------------------
%  computation of element matrices and vectors and their assembly
%-----------------------------------------------------------------

for iel=1:nel           % loop for the total number of elements

nl=nodes(iel,1); nr=nodes(iel,2); % extract nodes for (iel)-th element
xl=gcoord(nl); xr=gcoord(nr);% extract nodal coord values for the element
eleng=xr-xl;            % element length
index=feeldof1(iel,nnel,ndof);% extract system dofs associated with element

k=feode2l(acoef,bcoef,ccoef,eleng); % compute element matrix
f=fef1l(xl,xr);                     % compute element vector
[kk,ff]=feasmbl2(kk,ff,k,f,index);  % assemble element matrices and vectors

end
```

```
%---------------------------
%   apply boundary conditions
%---------------------------
[kk,ff]=feaplyc2(kk,ff,bcdof,bcval);

%---------------------------
%  solve the matrix equation
%---------------------------
fsol=kk\ff;

%--------------------
% analytical solution
%--------------------
c1=0.5/exp(1);
c2=-0.5*(1+1/exp(1));
for i=1:nnode
x=gcoord(i);
esol(i)=c1*exp(2*x)+c2*exp(x)+1/2;
end

%-----------------------------------
% print both exact and fem solutions
%-----------------------------------
num=1:1:sdof;
store=[num' fsol esol']

figure;
plot(gcoord,fsol,'r.-');
hold on;
plot(gcoord,esol)
legend('FEM Solution','Exact Solution');
titolo=['Mesh with ',int2str(nel),' elements'];
title(titolo);
%-----------------------------------------------------------------
```

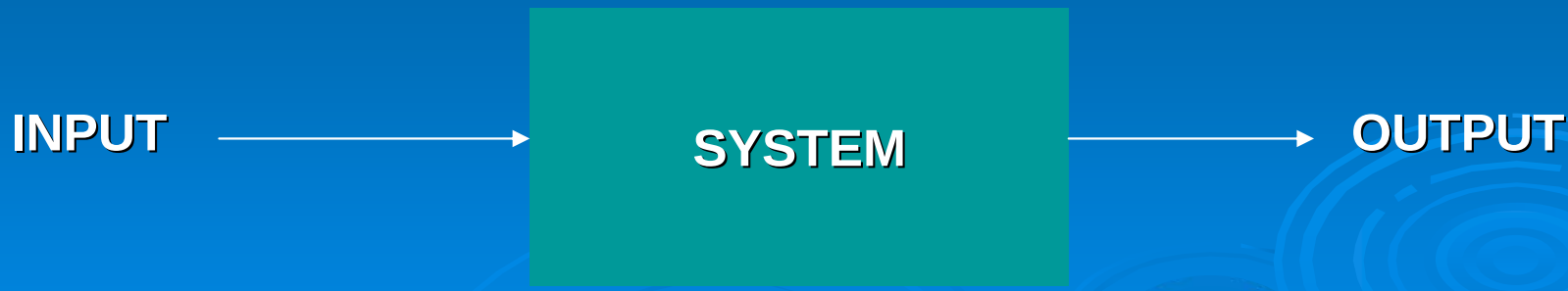
M-file Program (3)





Introduction to Simulink

Simulink is an extension to MatLab which uses a icon-driven interface for the construction of a block diagram representation of a process.
A block diagram is simply a graphical representation of a process (which is composed of an input, the system, and an output).



Why use Simulink?

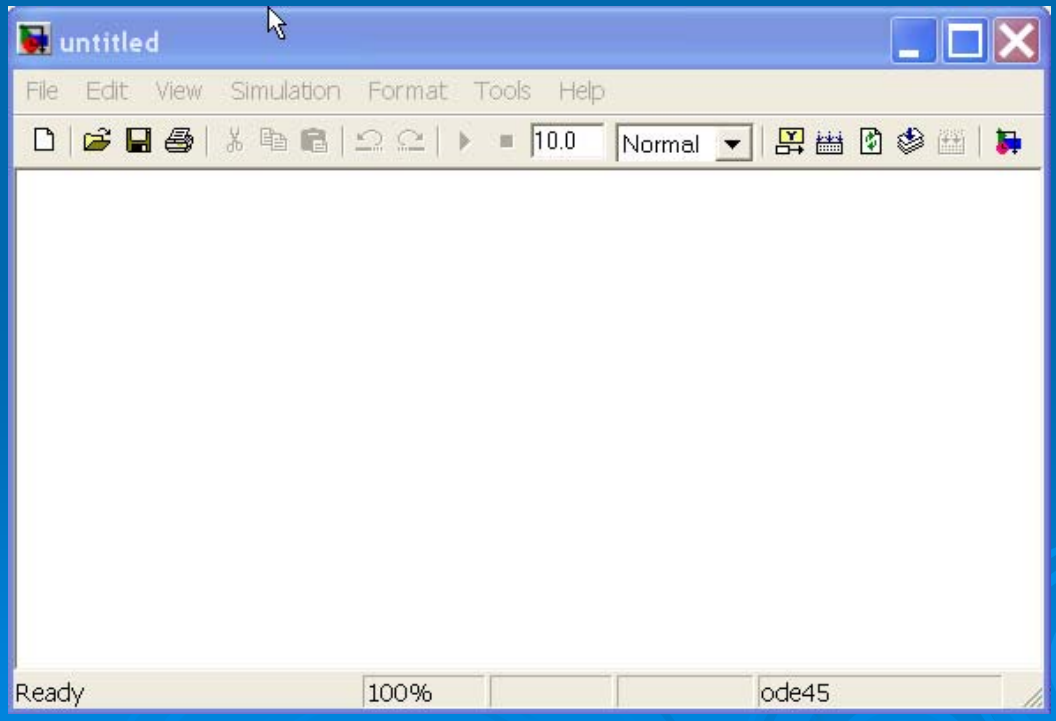
Realistic physical and/or engineering systems can be modelled as several blocks (“particles”) connected by links

- Inputs are often known
- Outputs may also be measured
- Systems may be too complex to solve manually

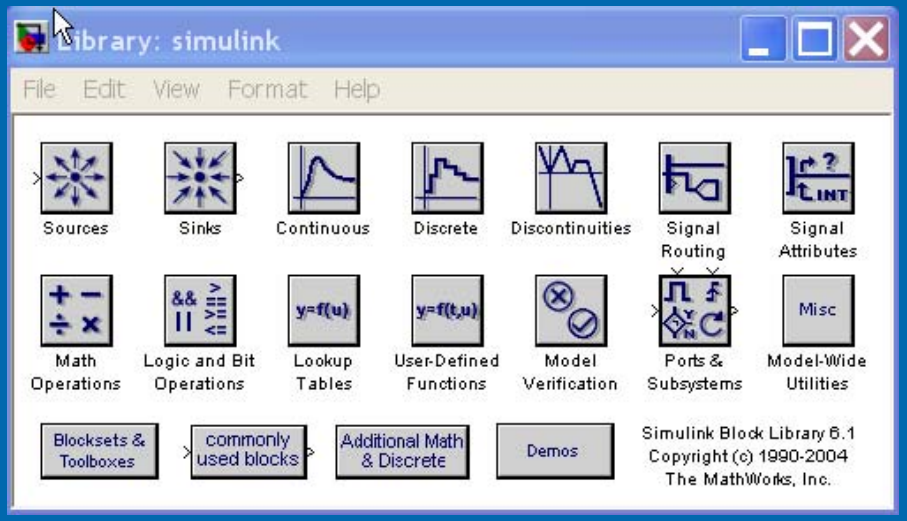
Instead of writing MatLab code, we simply connect the necessary **block** together to construct the block diagram. The **block** represent possible inputs to the system, parts of the systems, or outputs of the system. Simulink allows the user to easily simulate systems of linear and nonlinear ordinary differential equations.

Starting Simulink

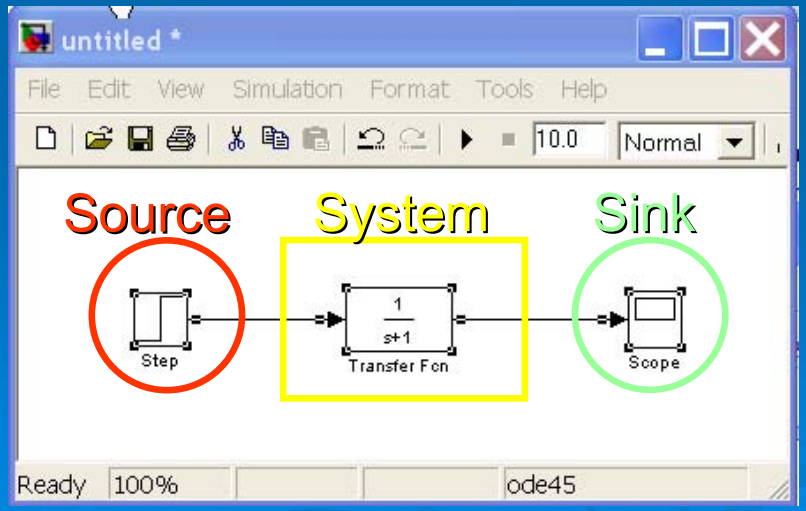
Type “simulink” in the command window or select “new simulink model” at the top of the MatLab command window



Main Simulink Window

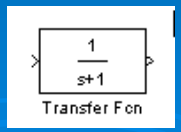


To build a system, you have to drag blocks from the main windows and drop into the model window.



Basic Elements

- Blocks and lines are the two major items in a Simulink Model
  - Blocks are used to generate, modify, combine, output and display signals
  - Sources – Used to generate various signals (e.g. a step function, harmonic force etc)
  - Sinks – Used to output or display signals
  - Lines are used to transfer signals between blocks
- Blocks may have zero to several input or output terminals.



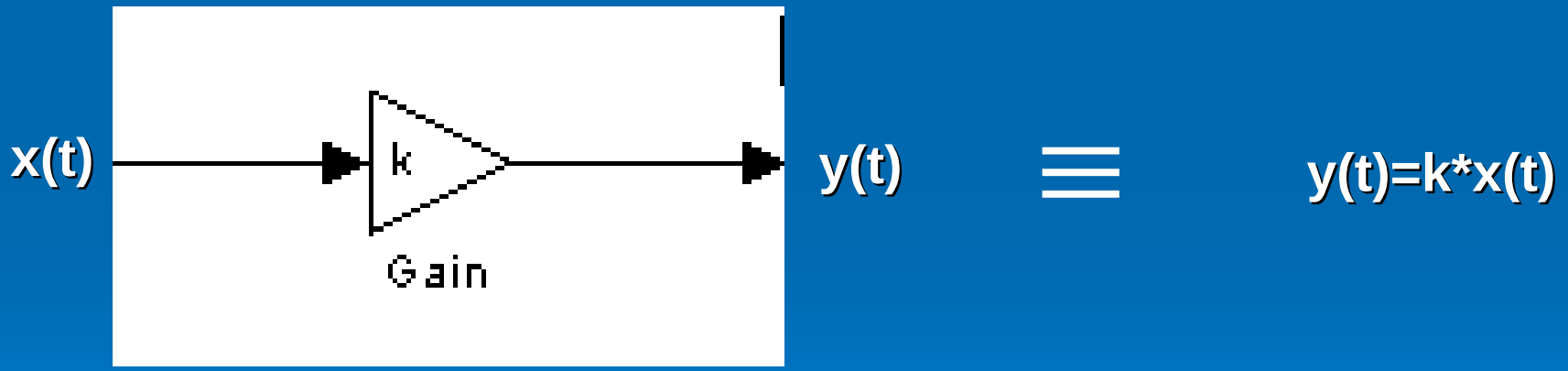
Continuous systems

Most physical systems are modelled as continuous systems, since they can be described using differential equations. Any systems that can be described using linear differential equations may be modelled using four primitive blocks:

- Gain Blocks
- Sum Blocks
- Derivative Blocks
- Integrator Blocks

Gain Blocks

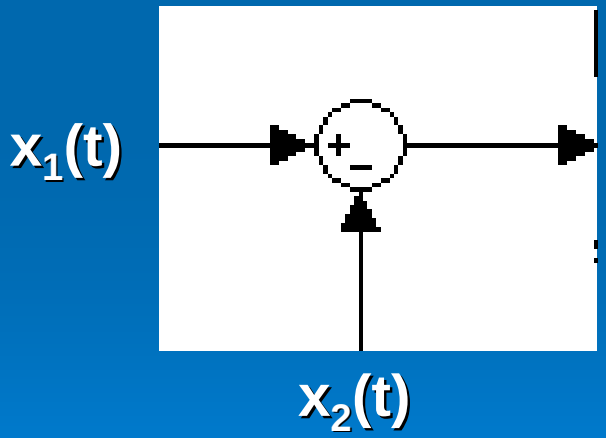
The simplest block diagram element is the gain block. The output of the gain block is the input multiplied by a constant:



This block, i.e. can model an linear electric resistance or a linear elastic spring

Sum Block

- The sum block permits us to add two or more inputs. The output of the sum block, is the algebraic sum of the inputs:



$\equiv$

$x_1(t)-x_2(t) = y(t)$

Derivative Block

The derivative block computes the time rate of its input. The block represents the differential equation y(t)=dx(t)/dt. Another useful representation derived by Laplace transform of the derivative of a function (ignoring initial condition) is:

$$L\left(\frac{dx(t)}{dt}\right) = sL(x(t)) = sX(s)$$

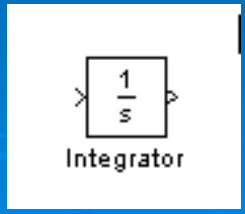
$$\frac{d}{dt} \longrightarrow s$$

Integrator Block

The integrator block computes the time integral of its input from the starting time to present. The integrator block represents the equation:

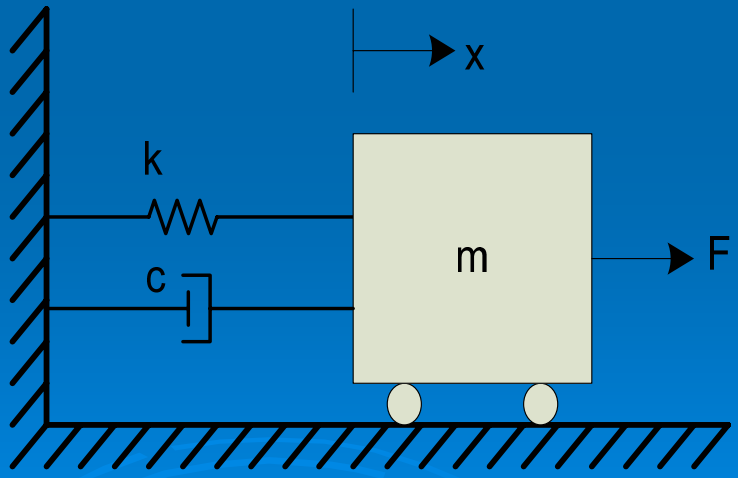
$$y(t) = y(t_0) + \int_{t_0}^{t} x(\tau)\, d\tau$$

Using again a Laplace transform for the integral, we have another representation for the block:



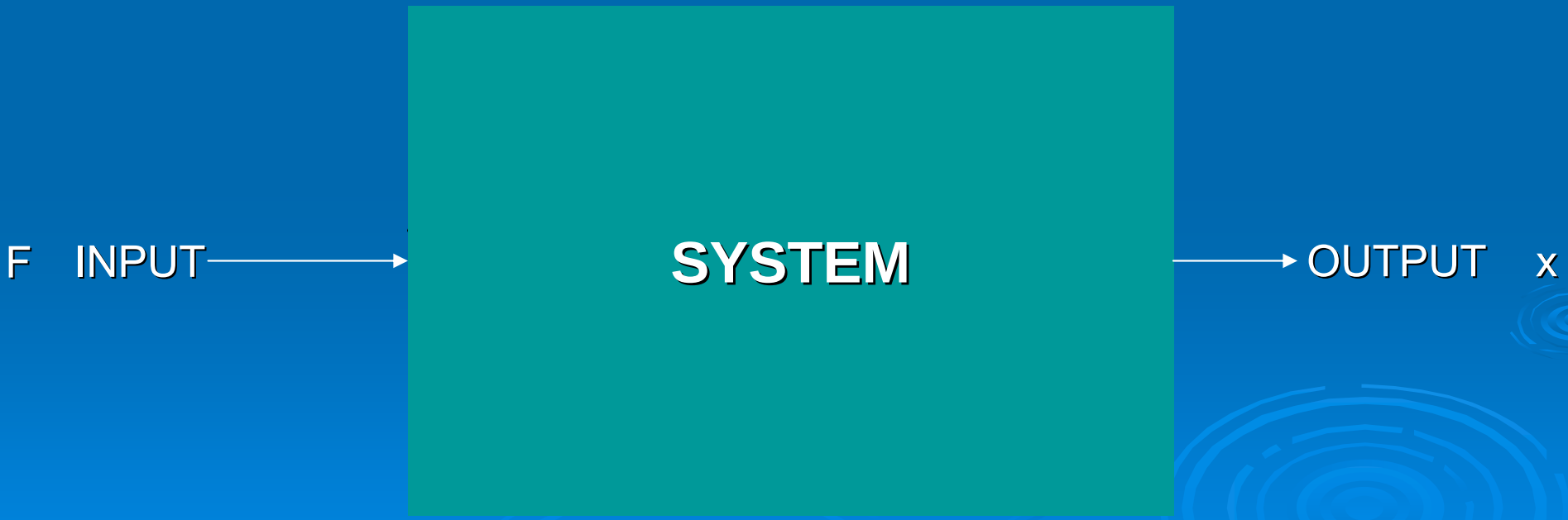
Example of a continuous system

Considering the spring-mass-dashpot system depicted in figure. Ignoring friction, we obtain the following differential equation of motion for the System: $m\ddot{x} + c\dot{x} + kx = F$



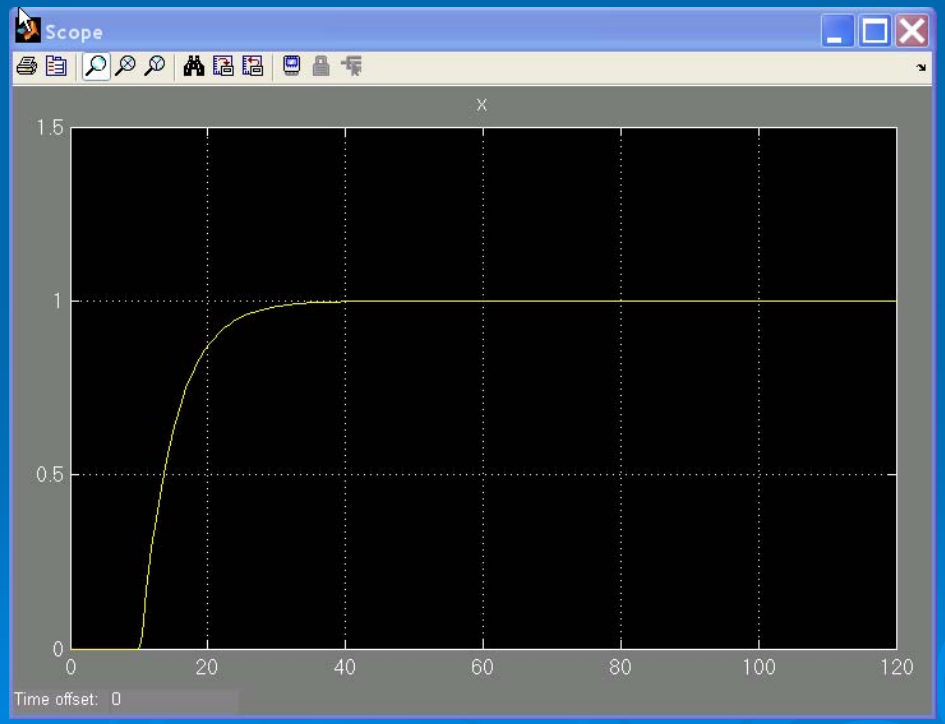
Example of a continuous system (2)

Rewriting the model for the system in this way: $\ddot{x} = \frac{F}{m} - \frac{c}{m}\dot{x} - \frac{k}{m}x$, the block diagram of the system will be:



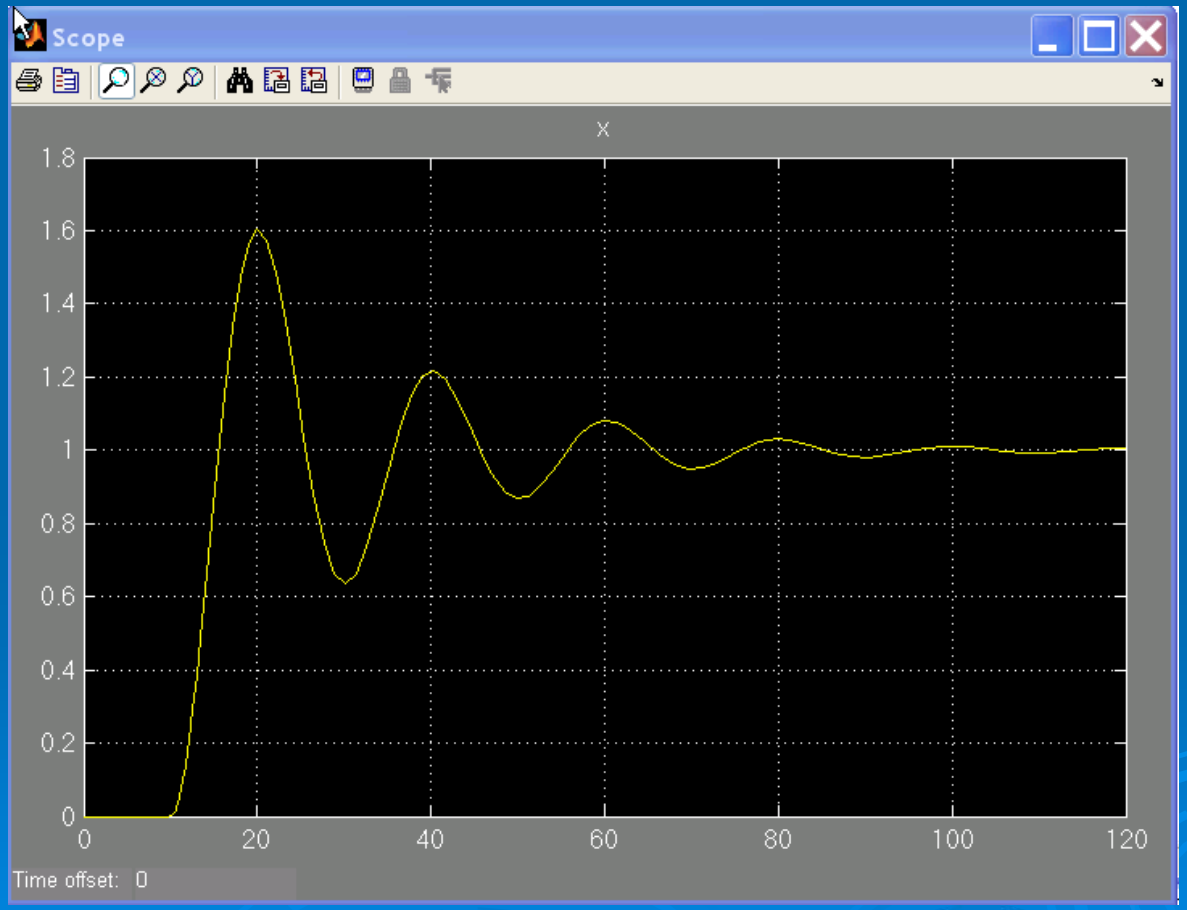
Example of a continuous system (3)

Now, with m= 1Kg, k=1 N/m , c=5N*s/m,, F(t)= u(t-10) (step function that is 0 for t ≤ 10s and 1 for t>10s) and initial condition $x(0) = 0, \dot{x}(0) = 0$, we obtain the following dynamics



Example of a continuous system (4)

Instead of with m=10, k= 1 and c= 1, we will have:



Many thanks for

your kind attention