3.2 Numerical solution of ODEs

Learn and master Euler's method. Write it yourself. After you master the concepts, write it using a separate function for the Right-Hand-Side (RHS) file. And a separate function for the Euler integrator.

Once that works well for you. Then learn the syntax of a numerical method with your favorite software, for example MATLAB's `ODE45`.

SAMPLE 3.7 A simple first order ODE on the computer: Solve $\dot{x}-x = t$, with initial condition $x(0) = 0$, numerically using Euler's method. Plot the solution for $0 \leq t \leq 2$ where t is dimensionless time.

Solution As explained in the text, Euler's method is the simplest, although not very efficient or stable, numerical method to solve ODEs. Here, we use Euler's method to solve the given differential equation. The conceptual steps are as follows and the actual code is implemented in MATLAB.

Step-1: Write the differential equation as a set of (if applicable) first order equations in the form $\dot{x} = f(x,t)$:

$$\dot{x} = x + t$$

Step-2: Specify initial conditions, time interval for solution, and time step for advancing the solution.

$$\begin{aligned} x(0) &= 0 && \text{(given)} \\ t_{\text{span}} &= [0 \quad 2] && \text{(given)} \\ h(=\Delta t) &= 0.002 && \text{(our choice)} \end{aligned}$$

Step-3: Implement Euler's method for computing x_{n+1} given x_n, t_n and h, and advance the time instance to $t_{n+1} = t_n + h$.

$$\begin{aligned} x_{n+1} &= x_n + \dot{x}(x_n, t_n)h \\ t_{n+1} &= t_n + h \end{aligned}$$

Step-4: Store solution at each time step in an array and plot the solution as t vs x.

Here is the MATLAB code with comments that solves the given ODE with the given initial condition.

```
% SCRIPT to solve xdot = x+t with x(0)=0 for t up to 2 sec
%-------------------------------------------------------------
x0=0; t0=0;              % specify initial conditions
tf=2;                    % specify final time
x=x0;  t=t0;             % initialize arrays for x and t
h=0.002;                 % specify time step
nsteps = (tf-t0)/h;      % calculate number of time steps
for n = 1:nsteps         % compute solutions in a loop
  xdot = x(n) + t(n);    % calculate the derivate at (tn,xn)
  x(n+1) = x(n) + h*xdot;% calculate x(n+1)
  t(n+1) = t(n) + h;     % advance tn to t(n+1)
end
plot(t,x), xlabel('t'), ylabel('x')
```

The result of running this code in MATLAB is shown in fig. 3.15. Please note that this is a plain vanilla code that does not take advantage of several nice features of MATLAB. It is meant to illustrate how we can use Euler's method in a straightforward manner to solve a differential equation.

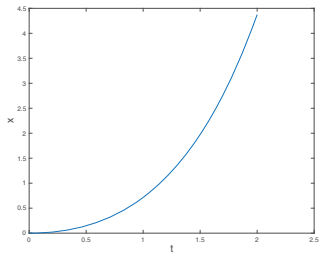

Figure 3.15: Numerical solution of $\dot{x} = x + t$, $x(0) = 0$ for $t \leq 2$ using Euler's method in MATLAB.

SAMPLE 3.8 Numerical solution of a set of ODEs: Solve the following set of two first order differential equation using Euler's method:

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= -\alpha x \end{aligned}$$

with the initial conditions $x(0) = 1$ and $y(0) = 0$. Write separate codes for implementing the solution algorithm (e.g., Euler's method) and the given differential equation, so that you can use the method of solution with any set of differential equations.

Solution Unlike the previous sample problem, we now have a set of two first order equations. We can put them in an array, say $\dot{z}$, and solve for z where x and y are treated as two separate elements of z, *i.e.*,

$$z = \begin{pmatrix} x \\ y \end{pmatrix} \quad \Rightarrow \quad \dot{z} \equiv \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} \dot{y} \\ -\alpha \dot{x} \end{pmatrix},$$

and the initial conditions for z can be written as,

$$z(0) = \begin{pmatrix} x(0) \\ y(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Let us write a MATLAB code where the equation is coded in a separate function and Euler's method is coded in another function assuming that we are solving for a set of first order equations.

- **Coding the differential equation:** Let us write a function that takes an array variable z, time t, and additional problem parameter p (*e.g.*, α in our equation), as the input, computes $\dot{z}$ inside the function and puts it out as the output (see schematic diagram in fig. 3.16). Here is the MATLAB code:

```
function [ zdot ] = MyDIffEqn(z,t)
% MyDiffEqn takes the value of array z at time t
%  and outputs the derivative zdot

alpha = 1;          % specify problem parameters
x = z(1); y = z(2); % unpack z
xdot = y            % code the given equation
ydot = -alpha*x
zdot = [xdot; ydot] % pack the derivatives in zdot
end                 % end of function
```

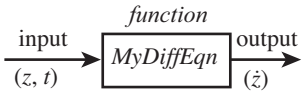

Figure 3.16: Schematic of coding a differential equation as a MATLAB function

- **Coding Euler's method:** We already coded Euler's method in the previous sample. We just need to convert it into an independent function which takes an array z_n at time t_n, calls the coded differential equation function (here, *MyDiffEqn* to compute $\dot{z}_n$ and advances the solution to t_{n+1} by computing z_{n+1}. Here is the MATLAB code:

```
function [zn1, tn1] = EulerForArray(zn, tn, h);
% EulerForArray computes the soltution zn1 at time
% tn1 of a set of differential equations  given the values
% of zn at time tn where tn1 = tn+h. The time step h must
% be specified by the user

zdot = MyDiffEqn(zn,tn);
zn1 = zn + h*zdot;
tn1 = tn + h;
end
```

- **Putting it all together:** We now have these two functions. *EulerForArray* already calls the function *MyDiffEqn*. We only need to write a master code that sets up the entire solution by specifying initial conditions $z(0)$, time step h, and time span t_{span} as we did in the previous sample.

```
% ODESolver: Script file that solves the differential equation
% coded in MyDiffEqn.m using Euler's method coded
% in EulerForArray.m with specified initial conditions
% and over the specified time interval

t0 = 0; tf = 10;          %specify time span [t0, tf]
z0 = [1; 0];              %specify initial conditions
h = 0.01;                 %specify time step
nsteps = (tf-t0)/h;       %calculate number of time steps

t = t0; z = z0';          %initialize output arrays t and z
tn = t; zn = z0;          %initialize tn and zn
for k=1:nsteps            %start a loop for computing z
    [zn1,tn1] = EulerForArray(zn,tn,h); %compute zn1 and tn1
            z = [z; zn1'];            %append zn1 in z
            t = [t; tn1];             %append tn1 in t
            zn = zn1; tn = tn1;       % reset zn and tn
end
x = z(:,1);               %unpack x from z (1st column)
y = z(:,2);               %unpack y from z (2nd column)
subplot(2,1,1), plot(t,x), xlabel('t'), ylabel('x')
subplot(2,1,2), plot(t,y), xlabel('t'), ylabel('y')
```

The two plots obtained from executing the script file are shown in fig. 3.17. The given differential equation represents a harmonic oscillator $\ddot{x} + \alpha x = 0$ and hence the solution for x and $\dot{x}(= y)$ are sinusoidal functions as expected.

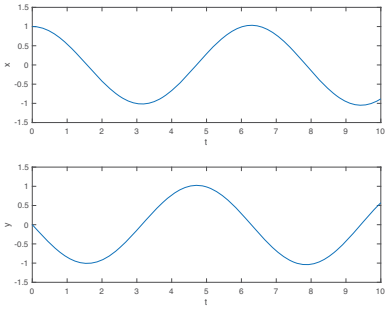

Figure 3.17: Plot of x vs t and y vs t as obtained from the numerical solution

Comments:

- The most important thing to realize here is that z is an array of independent variables and $\dot{z}$ is an array of first derivative of the variables in z. If we had a set of n first order equations, we could pack them all in the same $\dot{z}$. Having all equations in an array is very convenient for solving them using numerical solution algorithms. Note that Euler's method works the same way on an array as it does on a single equation.
- The codes presented here are written for clarity, not for minimizing code length or maximizing efficiency. We could, for example, condense all lines of code in *MyDiffEqns* to just two lines:

```
alpha = 1;
zdot = [z(2); -alpha*z(1)];
```

  Similarly, we could reduce the number of lines of code in the final script file *ODESolver* by combining different lines. The code then, admittedly, becomes a bit cryptic.
- Specifying direct numerical values to variables inside a function is usually not a good idea (it is called *hard coding*). Such values should be passed through input variables. For example, we specified the value of `alpha` in *MyDiffEqns* and values of time step `h` in *EulerForArray*. It would be much nicer to pass their values as input to the function so that if we want to solve the differential equations with various values of such parameters, we do not have to edit these functions every time.

SAMPLE 3.9 Using canned programs for solving ODEs: Find the solution of the following nonlinear pendulum equation over the time interval $0 \leq t \leq 20$ using any numerical ODE solver:

$$\ddot{\theta} + \sin\theta = 0; \quad \theta(0) = \frac{19\pi}{20}, \quad \dot{\theta}(0) = 0.$$

Solution Most mathematical software packages offer canned programs for solving ODEs. Here we will use MATLAB's most popular and generally robust ODE solver, `ode45`, which is an implementation of fourth and fifth order Runge-Kutta algorithm. Since we are going to use a canned program (MATLAB calls them *functions*), we will have to follow the syntax specified by the program for input and output variables. The built-in function `ode45` requires the differential equations to be coded in the following format:

```
[zdot] = myODEs( t, z )
```

output — function name — input

Figure 3.18: The ODEs need to be coded in a *function* with a specified syntax for the input and output so that the MATLAB's built-in solvers can use them properly.

Here the function, *myODEs*, contains the ODEs of interest as a set of first order ODES cast in the form of an array where the output array `zdot` represents the time derivative of the input array `z` and the input instant `t`. In this sample, we show how to form and code these arrays. However, before we proceed to write the function, we need to cast the given second order ODE as a set of first order ODEs. Following the discussion in the text, we do it by introducing another variable, say, α, where $\alpha = \dot{\theta}$. Then, we can write the given pendulum equation as a set of first order equations in θ and α as follows:

$$\begin{aligned} \dot{\theta} &= \alpha \\ \dot{\alpha} &= -\sin\theta. \end{aligned}$$

The initial conditions, now in terms of the new variables, are $\theta(0) = \frac{19\pi}{20}$ and $\alpha(0) = 0$.

For coding these ODEs, we need to put them in a convenient array form. Here the two first order ODEs are in variables θ and α. So, let $z_1 \equiv \theta$ and $z_2 \equiv \alpha$, where z_1 and z_2 are the elements of input array $\mathbf{z}$. Thus we have,

$$\mathbf{z} \equiv \begin{pmatrix} \dot{z}_1 \\ \dot{z}_2 \end{pmatrix} = \begin{pmatrix} z_2 \\ -\sin(z_1) \end{pmatrix}$$

and the initial conditions are

$$\mathbf{z(0)} \equiv \begin{pmatrix} z_1(0) \\ z_2(0) \end{pmatrix} = \begin{pmatrix} \frac{19\pi}{20} \\ 0 \end{pmatrix}.$$

Now we are ready to code the given differential equations in a function with the required input and output syntax:

```
function [zdot] = myODE(t,z);
% Call syntax:  [zdot] = myODE(t, z);
%  Inputs are:  t = time
%               z = [z(1); z(2)] = [theta; alpha]
%   Output is:  zdot = [z(2); -sin(z(1))]
```

```
% unpack z for clarity
z1 = z(1);    z2 = z(2);
% compute derivatives
z1dot = z2;
z2dot = -sin(z1);

% now pack derivatives in the output array zdot
zdot = [z1dot;  z2dot];
```

The next step is to use MATLAB's built-in ODE solver, `ode45`, to get the solution of the given differential equation. We need to specify the time span and the initial conditions for `ode45` to work. Time span is already given in the problem: $0 \leq t \leq 20$; so `tspan = [0  20]`, and the initial condition is `z0 = [19*pi/20; 0]`. Here is the script file containing the commands that run `ode45` and show the solution in the form of a plot of θ vs t over the specified time domain.

```
tspan = [0 20];       % specify time span
z0 = [19*pi/20; 0];   % specify initial conditions
[t, z] = ode45(@myODE, tspan, z0);     % run ode45

theta = z(:,1);       % unpack solution array z;
                      % theta is column 1 of z
thetadot = z(:,2);    % thetadot is column 2 of z

figure(1)             % open a figure window
plot(t,theta)         % plot theta as a function of t
xlabel('Time (t)'), ylabel('theta')    % label axes

figure(2)             % open another figure window
plot(t,thetadot)      % plot thetadot as a function of t
xlabel('Time (t)'), ylabel('thetadot') % label axes
```

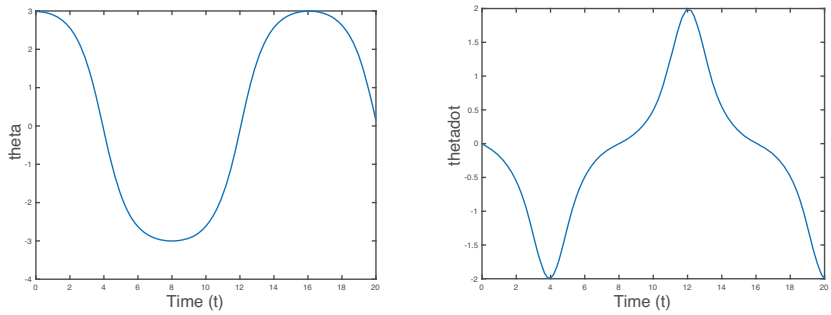

Figure 3.19: The solution obtained from MATLAB's ODE solver is plotted here: (a) plot of θ as a function of time t, and (b) plot of $\dot{\theta}$ as a function of time t.

3.2 ODEs, especially the common simple ODEs

3.2.1 Write as many different ODEs, and their general solutions, as you can.

3.2.1 Write as many different ODEs, and their general solutions, as you can.

Question 3.2.1:
Consider the differential equation $\frac{dy}{dx} = -2x$. Use Euler's method to approximate the value of y at $x = 2$, given that $y(0) = 3$. Use a step size of 0.5.

Solution:
Code:

```
dydt = @(x, y) -2 * x;

x0 = 0;
y0 = 3;
h = 0.5;

x_euler = x0:h:2;
n = numel(x_euler);
y_euler = zeros(1, n);
y_euler(1) = y0;

fprintf('Question 3.2.1:\n');
fprintf('Step\tx\t\t\t\ty\n');

for i = 2:n
    y_euler(i) = y_euler(i-1) + h * dydt(x_euler(i-1),
        y_euler(i-1));
    fprintf('%d\t%.2f\t\t\t%.4f\n', i-1, x_euler(i-1),
        y_euler(i-1));
end

fprintf('%d\t%.2f\t\t\t%.4f\n\n', n, x_euler(n),
   y_euler(n));

figure;
plot(x_euler, y_euler, '-o');
xlabel('x');
ylabel('y');
title('Question 3.2.1');
grid on;
hold on;
```

The approximate values of y at different steps are as follows:

Step	x	y
0	0.00	3.0000
1	0.50	3.0000
2	1.00	2.5000
3	1.50	1.5000
4	2.00	0.0000

The plot of the approximation using Euler's method is shown below:

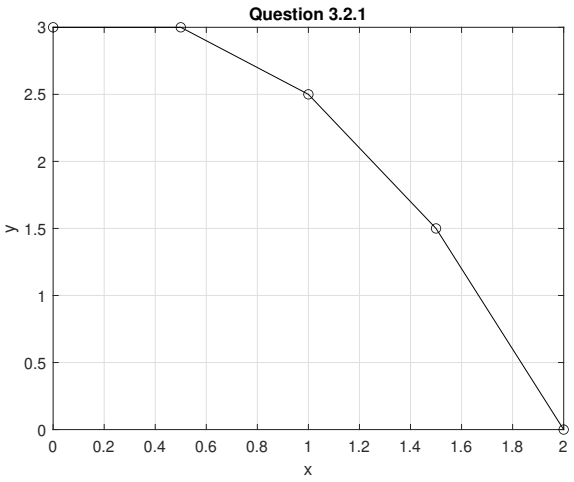

Figure 1: Graphical solution of 3.2.1