Numerical Packages for ODEs

adopted from Solving Differential Equations Using Simulink

Dr. R. L. Herman, UNCW

February 3, 2025

First Order ODEs in MATLAB

One can use MATLAB to obtain solutions and plots of solutions of differential equations.[1] This can be done either symbolically, using **dsolve**, or numerically, using numerical solvers like **ode45**. In this section we will provide examples of using these to solve first order differential equations. We will end with the code for drawing direction fields, which are useful for looking at the general behavior of solutions of first order equations without explicitly finding the solutions.

[1] Later we discuss other platforms such as GNU Octave, Python, and Malple. At some point others can be added such as Mathematica, SageMath and Julia.

Symbolic Solutions

The function **dsolve** obtains the symbolic solution and **ezplot** is used to quickly plot the symbolic solution. As an example, we apply **dsolve** to solve the

$$x' = 2\sin t - 4x, \quad x(0) = 0 \tag{1}$$

At the MATLAB prompt, type the following:

```
sol = dsolve('Dx=2*sin(t)-4*x','x(0)=0','t');
ezplot(sol,[0 10])
xlabel('t'),ylabel('x'), grid
```

The solution is given as

```
sol =


 (2*exp(-4*t))/17 - (2*17^(1/2)*cos(t + atan(4)))/17
```

Figure 1 shows the solution plot.

ODE45 and Other Solvers.

There are several ODE solvers in MATLAB, implementing Runge-Kutta and other numerical schemes. Examples of its use are in the differential equations textbook. For example, one can implement

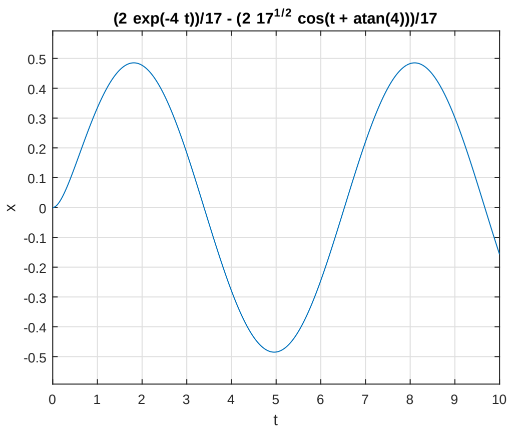

Figure 1: The solution of Equation (1) with $x(0) = 0$ found using MATLAB's **dsolve** command.

ode45 to solve the initial value problem

$$\frac{dy}{dt} = -\frac{yt}{\sqrt{2-y^2}}, \quad y(0) = 1,$$

using the following code:

```
[t y]=ode45('func',[0 5],1);
plot(t,y)
xlabel('t'),ylabel('y')
title('y(t) vs t')
```

One can define the function **func** in a file **func.m** such as

```
function f=func(t,y)
f=-t*y/sqrt(2-y.^2);
```

Running the above code produces Figure 2.

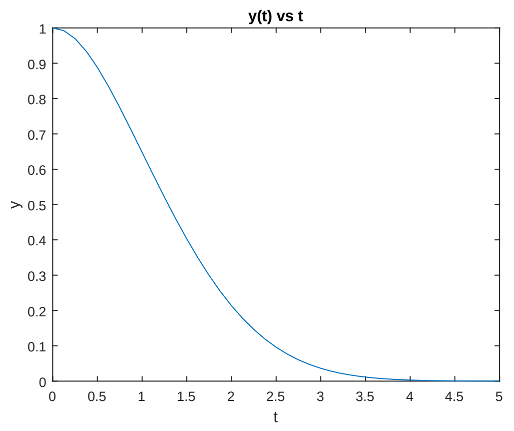

Figure 2: A plot of the solution of $\frac{dy}{dt} = -\frac{yt}{\sqrt{2-y^2}}, \quad y(0) = 1$, found using MATLAB's **ode45** command.

One can also use **ode45** to solve higher order differential equations. Second order differential equations are discussed in Section . See MATLAB help for other examples and other ODE solvers.

Direction Fields

One can produce direction fields in MATLAB. For the differential equation

$$\frac{dy}{dx} = f(x, y),$$

we note that $f(x, y)$ is the slope of the solution curve passing through the point in the xy=plane. Thus, the direction field is a collection of tangent vectors at points (x, y) indication the slope, $f(x, y)$, at that point.

A sample code for drawing direction fields in MATLAB is given by

```
dy=1-y;
dx=ones(size(dy));
quiver(x,y,dx,dy)
axis([0,2,0,1.5])
xlabel('x')
ylabel('y')
```

The mesh command sets up the xy-grid. In this case x is in $[0, 2]$ and y is in $[0, 1.5]$. In each case the grid spacing is 0.1.

We let dy = 1-y and dx =1. Thus,

$$\frac{dy}{dx} = \frac{1-y}{1} = 1 - y.$$

The **quiver** command produces a vector (dx,dy) at (x,y). The slope of each vector isdy/dx. The other commands label the axes and provides a window with xmin=0, xmax=2, ymin=0, ymax=1.5. The result of using the above code is shown in Figure 3.

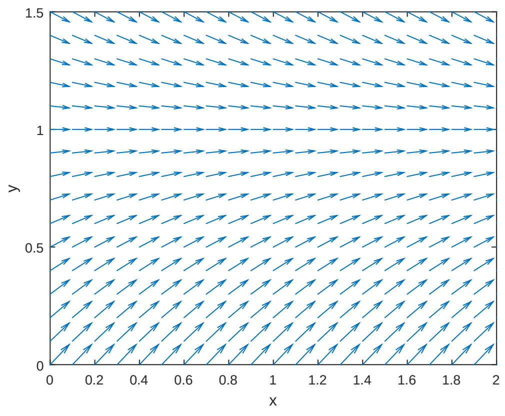

Figure 3: A direction field produced using MATLAB's **quiver** function for $y' = 1 - y$.

One can add solution, or integral, curves to the direction field for different initial conditions to further aid in seeing the connection between direction fields and integral curves. One needs to add to the direction field code the following lines:

```
hold on
[t,y] = ode45(@(t,y) 1-y, [0 2], .5);
plot(t,y,'k','LineWidth',2)
[t,y] = ode45(@(t,y) 1-y, [0 2], 1.5);
plot(t,y,'k','LineWidth',2)
hold off
```

Here the function $f(t,y) = 1 - y$ is entered this time using MATLAB's anonymous function, **@(t,y) 1-y**. Before plotting, the **hold** command is invoked to allow plotting several plots on the same figure. The result is shown in Figure 4

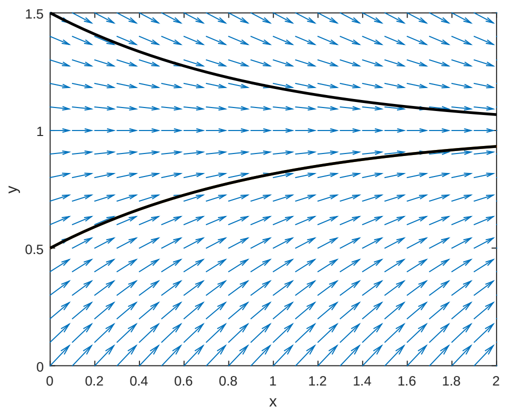

Figure 4: A direction field produced using MATLAB's **quiver** function for $y' = 1 - y$ with solution curves added.

Second Order ODEs in MATLAB

We can also use **ode45** to solve second and higher order differential equations. The key is to rewrite the single differential equation as a system of first order equations. Consider the simple harmonic oscillator equation, $\ddot{x} + \omega^2 x = 0$. Defining $y_1 = x$ and $y_2 = \dot{x}$, and noting that

$$\ddot{x} + \omega^2 x = \dot{y}_2 + \omega^2 y_1,$$

we have

$$\begin{aligned} \dot{y}_1 &= y_2, \\ \dot{y}_2 &= -\omega^2 y_1. \end{aligned}$$

Furthermore, we can view this system in the form $\dot{\mathbf{y}} = \mathbf{y}$. In particular, we have

$$\frac{d}{dt}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ -\omega^2 y_2 \end{bmatrix}$$

Now, we can use **ode45**. We modify the code slightly from Chapter 1.

```
[t y]=ode45('func',[0 5],[1 0]);
```

Here [0 5] gives the time interval and [1 0] gives the initial conditions

$$y_1(0) = x(0) = 1, \quad y_2(0) = \dot{x}(0) = 1.$$

The function **func** is a set of commands saved to the file **func.m** for computing the righthand side of the system of differential equations. For the simple harmonic oscillator, we enter the function as

```
function dy=func(t,y)
omega=1.0;
dy(1,1) = y(2);
dy(2,1) = -omega^2*y(1);
```

There are a variety of ways to introduce the parameter ω. Here we simply defined it within the function. Furthermore, the output **dy** should be a column vector.

After running the solver, we then need to display the solution. The output should be a column vector with the position as the first element and the velocity as the second element. So, in order to plot the solution as a function of time, we can plot the first column of the solution, y(:,1), vs t:

```
plot(t,y(:,1))
xlabel('t'),ylabel('y')
title('y(t) vs t')
```

The resulting solution is shown in Figure 5.

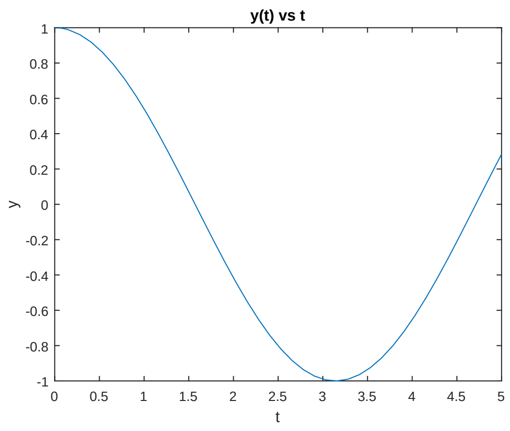

Figure 5: Solution plot for the simple harmonic oscillator.

We can also do a phase plot of velocity vs position. In this case, one can plot the second column, y(:,2), vs the first column, y(:,1):

```
 plot(y(:,1),y(:,2))
 xlabel('y'),ylabel('v')
 title('v(t) vs y(t)')
```

The resulting solution is shown in Figure 6.

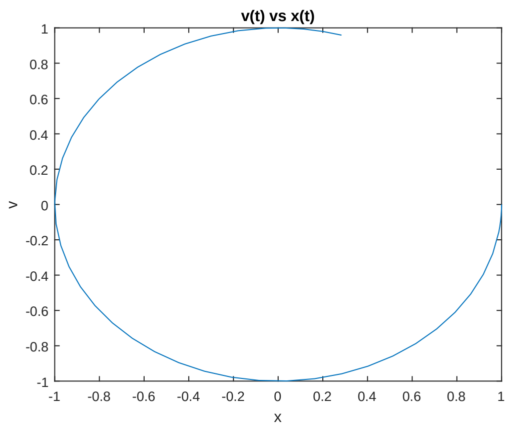

Figure 6: Phase plot for the simple harmonic oscillator.

Finally, we can plot a direction field using a quiver plot and add solution curves using **ode45**. The direction field is given for $\omega = 1$ by dx=y and dy=-x.

```
clear
[x,y]=meshgrid(-2:.2:2,-2:.2:2);
dx=y;
dy=-x;
quiver(x,y,dx,dy)
axis([-2,2,-2,2])
xlabel('x')
ylabel('y')

hold on
[t y]=ode45('func',[0 6.28],[1 0]);
plot(y(:,1),y(:,2))
hold off
```

The resulting plot is given in Figure 7.

GNU Octave

MUCH OF MATLAB'S FUNCTIONALITY CAN BE USED IN GNU OCTAVE. However, a simple solution of a differential equation is not the same. Instead GNU Octave uses the Fortan **lsode** routine. The

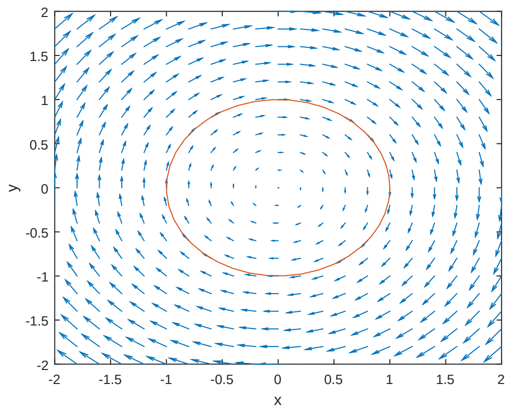

Figure 7: Phase plot for the simple harmonic oscillator.

main code below gives what is needed to solve the system

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ -cy \end{bmatrix}.$$

```
global c
c=1;

y=lsode("oscf",[1,0],(tau=linspace(0,5,100))');

figure(1);
plot(tau,y(:,1));
xlabel('t')
ylabel('x(t)')

figure(2);
plot(y(:,1),y(:,2));
xlabel('x(t)')
ylabel('y(t)')
```

The function called by the **lsode** routine, **oscf**, looks similar to MATLAB code. However, one needs to take care in the syntax and ordering of the input variables. The output from this code is shown in Figure 8.

```
function ydot=oscf(y,tau);
global c

ydot(1)=y(2);
ydot(2)=-c*y(1);
```

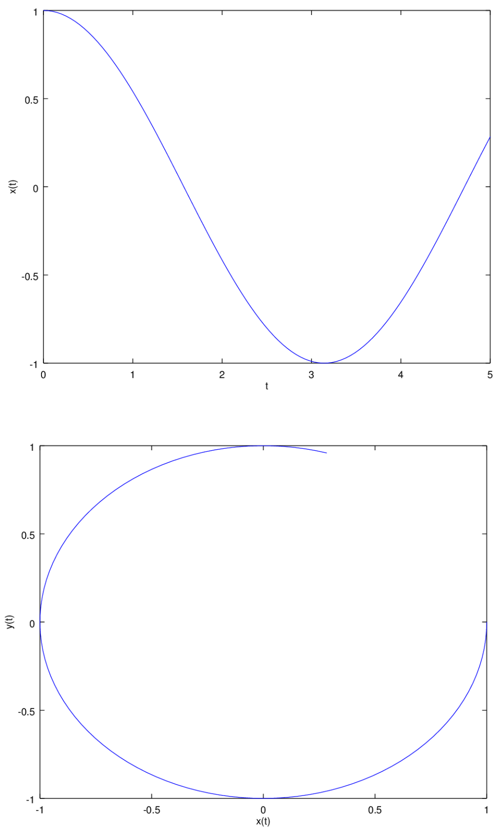

Figure 8: Numerical solution of the simple harmonic oscillator using GNU Octave's **lsode** routine. In these plots are the position and velocity vs times plots and a phase plot.

Python Implementation

One can also solve ordinary differential equations using Python. One can use the **odeint** routine from **scipy.inegrate**. This uses a variable step routine based on the Fortan **lsoda** routine. The below code solves a simple harmonic oscillator equation and produces the plot in Figure 9.[2]

[2] One can insert the following into an online compiler such as myCompiler.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Solve dv/dt = [y, - cx] for v = [x,y]
def odefn(v,t, c):
  x, y = v
  dvdt = [y, -c*x ]
  return dvdt

v0 = [1.0, 0.0]
```

```
t = np.arange(0.0, 10.0, 0.1)
c = 5;

sol = odeint(odefn, v0, t,args=(c,))

plt.plot(t, sol[:,0],'b')
plt.xlabel('Time␣(sec)')
plt.ylabel('Position')
plt.title('Position␣vs␣Time')
plt.show()
```

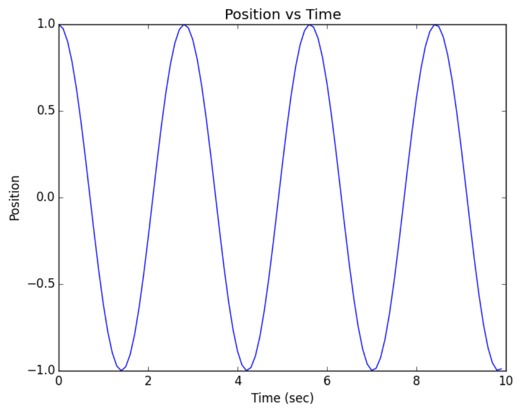

Figure 9: Numerical solution of the simple harmonic oscillator using Python's odeint.

If one wants to use something similar to the Runga-Kutta scheme, then the **ode** routine can be used with a specification of ode solver. The below code solves a simple harmonic oscillator equation and produces the plot in Figure 10.

```
from scipy import *
from scipy.integrate import ode
from pylab import *


# Solve dv/dt = [y, - cx] for v = [x,y]
def odefn(t,v, c):
   x, y = v
   dvdt = [y, -c*x ]
   return dvdt

v0 = [1.0, 0.0]

t0=0;
tf=10;
dt=0.1;
```

```
c = 5;

Y=[];
T=[];

r = ode(odefn).set_integrator('dopri5')
r.set_f_params(c).set_initial_value(v0,t0)

while r.successful() and r.t+dt < tf:
    r.integrate(r.t+dt)
    Y.append(r.y)
    T.append(r.t)

Y = array(Y)

subplot(2,1,1)
plot(T,Y)
plt.xlabel('Time␣(sec)')
plt.ylabel('Position')

subplot(2,1,2)
plot(Y[:,0],Y[:,1])
xlabel('Position')
ylabel('Velocity')
show()
```

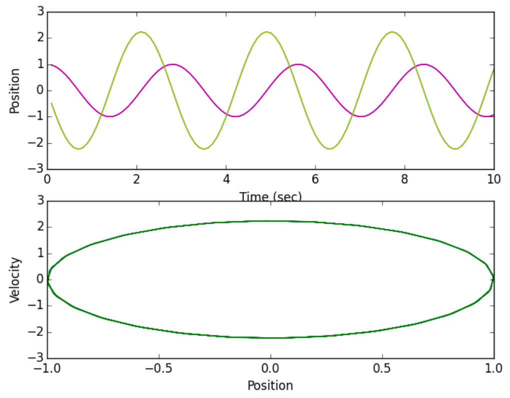

Figure 10: Numerical solution of the simple harmonic oscillator using Python's ode routine. In these plots are the position and velocity vs times plots and a phase plot.

Maple Implementation

MAPLE ALSO HAS BUILT-IN ROUTINES FOR SOLVING DIFFERENTIAL EQUATIONS. First, we consider the symbolic solutions of a differential equation. An example of a symbolic solution of a first order differential equation, $y' = 1 - y$ with $y(0) - 1.5$, is given by

```
> restart: with(plots):
> EQ:=diff(y(x),x)=1-y(x):
> dsolve({EQ,y(0)=1.5});
```

The resulting solution from Maple is

$$y(x) = 1 + \frac{1}{2}e^{-x}.$$

One can also plot direction fields for first order equations. An example is given below with the plot shown in Figure 11.

```
> restart: with(DEtools):
> ode := diff(y(t),t) = 1-y(t):
> DEplot(ode,y(t),t=0..2,y=0..1.5,color=black);
```

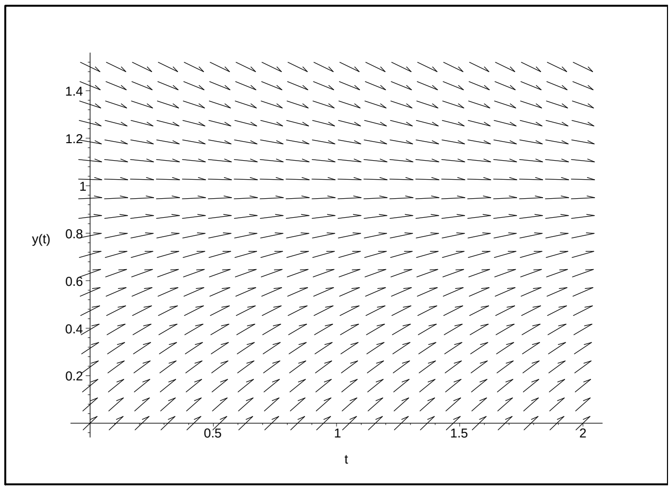

Figure 11: Maple direction field plot for first order differential equation.

In order to add solution curves, we specify initial conditions using the following lines as seen in Figure 12.

```
> ics:=[y(0)=0.5,y(0)=1.5]:
> DEplot(ode,yt),t=0..2,y=0..1.5,ics,arrows=medium,linecolor=black,color=black);
```

These routines can be used to obtain solutions of a system of differential equations.

```
> EQ:=diff(x(t),t)=y(t),diff(y(t),t)=-x(t):
> ICs:=x(0)=1,y(0)=0;
> dsolve([EQ, ICs]);
> plot(rhs(%[1]),t=0..5);
```

A phaseportrait with a direction field, as seen in Figure 13, is found using the lines

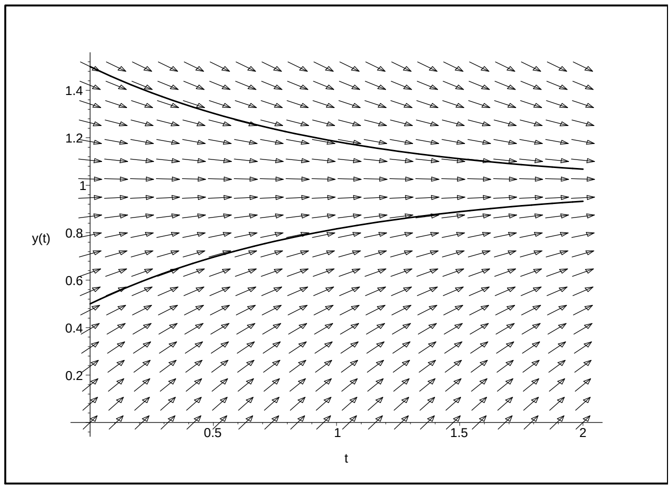

Figure 12: Maple direction field plot for first order differential equation with solution curves added.

```
> with(DEtools):
> DEplot( [EQ], [x(t),y(t)], t=0..5, x=-2..2, y=-2..2, [[x(0)=1,y(0)=0]],
  arrows=medium,linecolor=black,color=black,scaling=constrained);
```

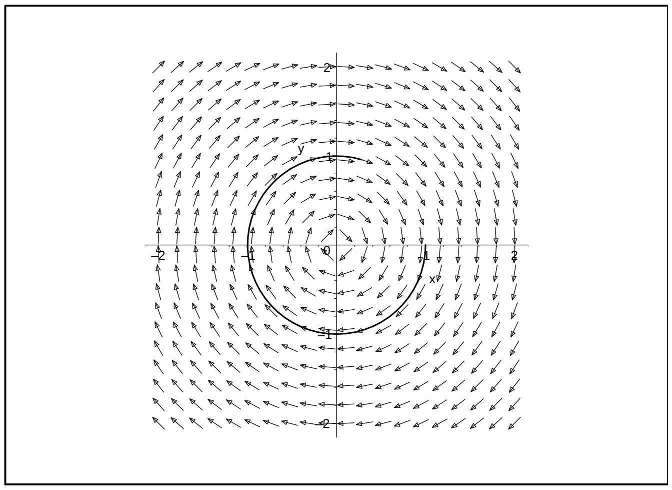

Figure 13: Maple system plot.