3

Introduction to the Numerical Methods

This chapter provides an overview of the numerical platforms that we use throughout the book. We provide some motivation for the two that we have chosen and describe how to start using the programs.

3.1 Introduction

The models that we will use in this book are primarily differential equations. This presents a challenge: How do we understand the models, use them to make useful predictions or provide insight into the biology without several additional years of mathematical training? Typical undergraduate sequences in calculus, linear algebra, differential equations and scientific computing are useful for understanding the nuances of models. Nonlinearities often put models outside the reach of an undergraduate differential equations class while scientific computing or numerical methods classes often focus on the underlying methods, rather than the applications.

In this chapter, we discuss the two main computer programs/languages that we will use throughout the course. The book will intentionally provide parallel numerical scripts in Python and Matlab. Both of these are able to visualize solutions, numerically approximate solutions to a range of equations, and can be used to automate tasks, like sampling, which will be important throughout the book. So why these two packages? Years ago students would have to learn Fortran to be able to solve equations numerically – adding several semesters of course work. Both of these languages are widely used and it is our philosophy that students should have some exposure to all of these languages and then focus on the one that is most comfortable. There are many other languages/platforms that are in use. Languages based

on C-like languages including R are widely used. There are openMPI languages that are able to call both C^{++} and Fortran libraries such as Julia.

This book is relatively agnostic about the language of choice. Matlab is well developed and has an excellent support system. Python is very flexible and has many workgroups that are designed for helping. They each have some drawbacks as well. Matlab has a reputation as a 'slow' language because it is interpreted. However, there are many ways to speed things up with more experience. Python has a specific culture that can sometimes be overly complicated and difficult to follow since it is so flexible and does not have a centralized structure. R is a relatively complex language to start with and has a steep learning curve. All of these platforms will run into mathematical problems that take an unreasonable amount of time to complete. There should be no examples in this book where this will happen but keep in mind there is no perfect language and many, many people studying how to run more and more complicated problems. The recommendation here is to work with codes in both Matlab and Python. Eventually, each person will find what is most comfortable.

For those that have never done any programming it might seem strange to use the word 'comfortable' – aren't all language uncomfortable? This is definitely true at the beginning. After a bit of work, there are nuances that make certain languages/programs easier for different people. For example, Matlab m-files do not read indentation. So you can type commands wherever you want. Python is indent-aware and will throw an error (i.e. break and not run) if the indents do not match. This means that the typical Python files look neater and more organized than a typical working m-file. At the same time, Matlab is more self-contained and does not need any extra libraries to be read in so an m-file written for 2018 Matlab will work on any machine with 2018 Matlab. In Python and R, libraries have to be loaded to 'teach' the computers Python what certain commands mean. This can occasionally cause issues when sharing files.

All of these are accessible and we will start with essentially no prior knowledge. This chapter is to provide a basic outline of how to access the platform (Matlab or Python), how to write a program, how to edit a program and how to run a program. We will consider a few best

practices in writing numerical codes in general and in each platform specifically.

One important note about the codes provided in this book. For many chapters we have opted for simpler, less efficient, 'home-built' code rather than implementing large scale packages. There are two reasons for this. First, it is much easier to get used to writing and editing scripts with simpler examples – not to mention developing a better understanding of the actual methods used. Second, with all packages there is an issue of upkeep and verification. One cannot assume that the provided code does not have bugs or errors considering companies such as Apple, Google and Microsoft constantly develop patches to fix codes that cost millions of person hours to develop. Similarly, codes developed for specific projects are not always updates which means it is possible to lose functionality permanently if you don't know how it was written. Some exceptions here are in the later chapters where the implementation is beyond what is expected by readers. The book as chosen a few, well-known, reasonably well-supported implementations; however these are by no means the only methods available. The reader should feel free to explore decide what is the cutting-edge implementation.

3.2 Best Practices in Coding

Writing codes is very much like writing in any language. You would like the writing to be clear, concise, accurate and readable. One extra consideration is the user interaction with most of these programs. When writing a poem, it is unusual for the author to expect that a reader will add to it or change specific lines or words. Good coding expects this and invites this. So we have to make sure that the files are organized without extraneous parts or things that don't work. In the codes developed in this book, we have made a conscious choice to include some commands that are not the most efficient to ensure they are clear.

3.2.1 Folder structure

If you open your computer and the desktop is full of files, images, and assorted notes, I would encourage you to start learning file structures. In this book, there will be many different files used. Some will be used for many applications, some will only be used for a single model. Your life will be immeasurably easier if you learn to sort these so you can find them easily and not accidentally delete or overwrite files you are working on. The simplest structure, that has some redundancy, but is ordered in a sensible manner might be a folder for the specific course. Within that folder, you have folders for each chapter. Within each chapter folder you have folders for Matlab and Python and an assignment folder. Within each of the numerical folders you have folders for tasks in the assignment (for example Question 1). There is a high likelihood of redundancy with this since Question 1 might use the same script as Question 2. However, there is much less chance that you accidentally over-write your code answering Question 1 when you start working on Question 2. Typically, for long-lived projects, there are specific times when the file structure is revised. But this should be done as a completely separate task than building the files. It is our expectation that most students using this book will not need to revise their file structure.

One disadvantage to this is that when running scripts, there is the need to change folders. Fortunately in all of our examples, the play button will query this and automatically change folders for you. But it is something to be aware of when sharing codes with other people.

3.2.2 Naming Conventions

All names should be explanatory. This includes folder names, file names, variable names, function names, and code-block names. If you have a file in a folder called `temp1`, it might be junk or it might be the final that you never got around to moving to the correct location. Likewise, a file called `stuff.py` requires a person to open it to see what it is. It is very confusing if you open a file that is supposed to plot a function $f(x,t)$ and x is never used and there is some parameter called g that has a value that is never explained. All of these names should be understandable with very little reader orientation.

Consider variable names like `gamma` for a parameter γ. File names like `pred_prey_chapter_1` are easy to understand.

3.2.3 Code Structure

All codes consist of blocks that are typically identified by some commenting convention. In Matlab it might look like,

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Solve the Differential Equation   %
% y' = A y                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

All codes need a header and a body. The header explains what the code is used for, when it was written/edited and any information needed to read and run the code. An example header in Python looks like,

```
"""
Created on Wed Jan 13 12:14:28 2021
Data fitting routine for microbiome data.
Gives current best estimate for parameters
and is the set that SA_microbiome_exc uses
@author: cogan
"""
```

Directly after the header in all our codes will clean our current environment. Computers have a tendency to remember things unless they are explicitly told to forget them. We advise clearing the local and global environment and closing all open figure windows. For Matlab, this can be done using the commands,

```
clear
close all
```

In Python, we use,

```
from IPython import get_ipython
get_ipython().magic('reset -sf')
```

The body of a code often includes blocks that should be indicated. There are lots of different ways to do this, but we typically use blocked comments. The symbols # and % are used to start comments in Python/R and Matlab, respectively. So a block might start with

```
############################
# Visualize all variables  #
############################
```

in Python and R or

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Visualize all variables   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

in Matlab.

3.2.4 Comments

All code should have explicit comments that describe what is going on. These can be done in code blocks or directly after specific lines. These comments should be brief, but self-contained and explain what the specific line or section is doing. You definitely should avoid commenting every line, but anything that reminds the reader what is going on will help remind you what is going on when you look at your codes in 3 months!

3.3 Getting The Programs Running

In this section we describe how to get all programs Matlab and Anaconda. Provide screen shots of what they look like. Define windows etc.

3.3.1 Python

There are many ways to start using Python. You can download the source files directly and use them from a command line – for example from a terminal window on a Mac or in a c drive on a windows

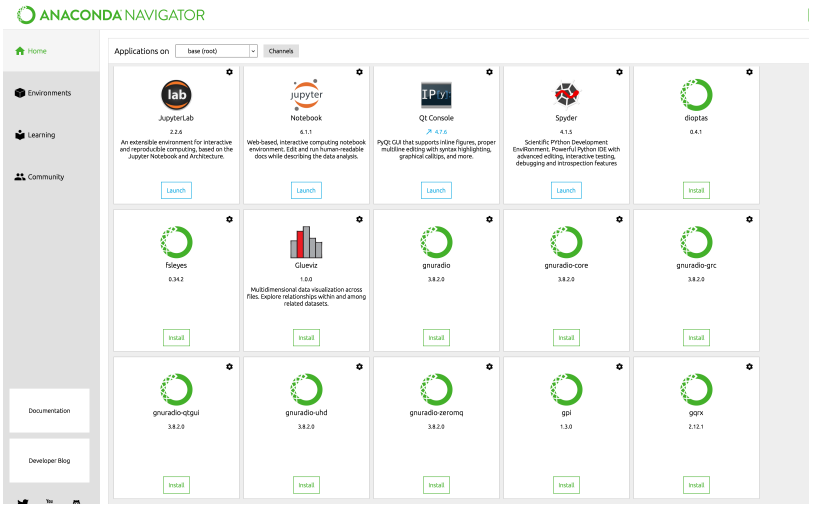

Figure 3.1: Screenshot of the Anaconda Navigator

computer. There are also many packages that have been developed to combine an editor where you can write the programs along with a graphical interface. We like the Anaconda package as it is easy to keep updated and well documented. The main package can be found at `https://www.anaconda.com`. When you download and install this, you will be able to open the main window that has many, many options for programs (see Figure 3.1). From here we use the Spyder environment to write and run our Python codes (see Figure 3.2).

Spyder has a lot of buttons that help run and debug programs. These include the green 'play' button that runs all the lines scripts and 'play/-pause' that moves line-by-line. This also gives the path for the programs and other information that helps with navigation etc.

Anaconda also has an environment designed for interactive Jupyter notebooks. This is a nice way to organize blocks of text and codes in one place.

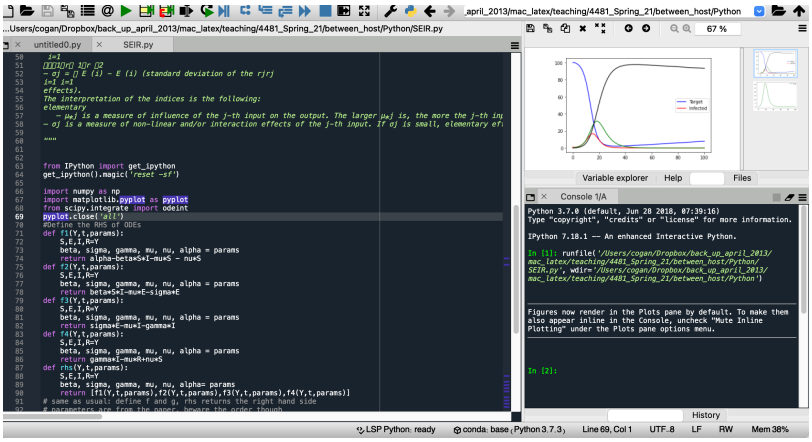

Figure 3.2: Screenshot of the Spyder environment

3.3.2 Matlab

3.4 Initial Programs

Matlab and Python treat variables and inputs quite differently. We will provide a starter code for each language that illustrates some of the differences.

In all these platforms we will need to have parameters. These are symbols that we provide specific values for. These are relatively obvious, typing `alpha=1` in all these languages then specifies this value in future commands. Some differences can be seen when running programs. Python does not display anything unless instructed to. Matlab displays unless instructed not to with a semicolon (;) at the end of a line.

Variables are typically represented as vectors or arrays. In Matlab, everything is an array. So if `alpha =1`, Matlab interprets this as a 1×1 array. Just as usual, there are dependent and independent variables. Independent variables require definition while the dependent variables are usually the output of specific commands. To define an independent variable t that lives in the interval $[a,b]$, we will subdivide this interval into n points. The commands to do this for a specific number of points (say $n = 100$) on a specific interval (say $[-\pi,\pi]$)

```
n=100
t=[-pi:2*pi/(n-1): pi]
```

You can check the size of this by typing `size(t)`.

In Python, there is a bit more work. Python is used for a wide variety of applications and often the arrays are not numerical. For example, Python is perfectly happy with an array that is a list of names. To make sure Python has access to the numerical commands we will use, we will add two commands to the header files,

```
import numpy as np
from matplotlib import pyplot
```

To make an array for t similar to Matlab, we can type,

```
t=np.arange(-np.pi,np.pi,1/99)
```

So what is this doing? the `np.*` indicates that * should be interpreted using numpy. If you just type pi, python defaults to thinking of this as letters rather than a fundamental constant. Whereas `np.pi` tells Python that we really mean the constant $\pi = 3.1415....$

If you want to make a dependent variable, it is not terribly different than typical functional notation used in Calculus. In Matlab you could type

```
f = sin(t),
```

for example. But if you want to multiply variables you have to keep in mind that Matlab treats everything as an array. So multiplication really means either dot or cross-product. If you want to use the function $f(t) = t^2$, you have to tell Matlab you don't mean the inner product of the vector t with itself. This is done using a special syntax,

```
f = t.^2,
```

where the $.^2$ tells Matlab you mean to square each element of the array, t.

Similarly in Python

```
f=np.sin(t),
```

gives $\sin(t)$.

`f=t**2,`

gives $f = t^2$. The $**2$ is Python syntax for 'raise to the second power'.

Another useful thing is to be able to visualize variables. We will focus on plotting in one dimension , but keep in mind we might want to visualize two or three dimensional functions, scatterplots, barplots, etc. All these programs can do this and we will introduce the syntax as we need it. The basic plotting syntax for Matlab is `plot(t,f,options)`. We will leave the options for now, but typing `help plot` in Matlab will give a nice overview of things that can be done. Similarly in Python and R the plotting commands we use are `pyplot.plot(t,f,options)` and `plot(t,f)`, respectively. Note that Python has multiple plotting libraries, but matplotlib is the one that we use and denote it `pyplot`.

The final piece that is important is solving differential equations numerically. This is an incredibly important part of mathematics and there has been tremendous progress in this in the last 60 years. Really, even Newton compiled approximate solutions to differential equations so really, this topic is several hundreds of years old. We are not going to be able to cover everything. Most of the models that we are using are nice enough that we will not run into many issues that the standard differential equation solvers cannot handle. But it is absolutely important to understand that there are many assumptions that go into solving differential equations numerically. It is not difficult to find simple equations that will break all of these methods. This is another reason to have multiple ways to implement numerical methods since the comparison can provide some confidence in the output. But keep in mind, the output from Matlab or Python can be wrong and sometimes in very serious ways.

So how do we solve differential equations? We need a few things in either Matlab or Python. Most of the general solvers are built to solve equations in the form,

$$\begin{aligned} \frac{d\mathbf{y}}{dt} &= f(\mathbf{y},t), \\ \mathbf{y}(0) &= \mathbf{y}_0. \end{aligned}$$

So we need a place to store the right-hand-side. We need the initial

condition. We also need to tell the programs where to stop. The standard functions in Matlab and Python will take discrete steps in time and provide estimates for the value of **y** at these times. The step-sizes are normally controlled by the internal programs used in the functions called to solve the differential equations. This is to help take large steps if the solution is not changing much and refine the time steps if the function varies a lot. Sometimes this can cause issues – if the solution changes extremely quickly, the step size can get small enough that nothing actually changes. These equations are called *stiff* and special methods need to be used. Also, because the step size may change, it may be difficult to compare solutions quantitatively since the t-values may not overlap. This is usually corrected by interpolation – but there are fancier ways.

3.4.1 Differential Equations in Python

There are many, many ways to solve differential equations in Python. We use the package `scipy` that has an integration package and a very flexible solver `solve_ivp`.

A basic script for solving a scalar differential equation only takes a few lines,

```
"""
Created on Wed Dec 22 21:11:32 2021
Basic ODE Script
@author: cogan
"""
from IPython import get_ipython
get_ipython().magic('reset -sf')
import numpy as np
from matplotlib import pyplot
# import plotting library
from scipy.integrate import solve_ivp

def Rhs(t,Y,params):
   r, K = params
   y = Y
   return r*y*(K-y)
```

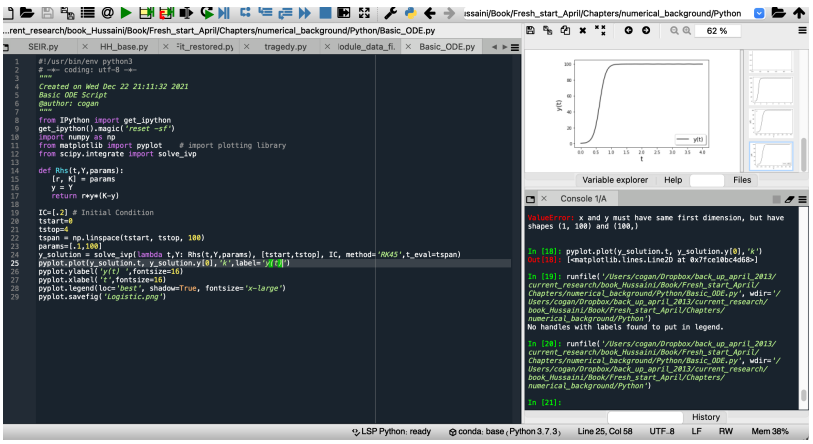

Figure 3.3: Screenshot of the Spyder environment after running the basic differential equation solver.

```
IC=.2 # Initial Condition
tstart=0
tstop=4
tspan = np.linspace(tstart, tstop, 100)
params=[.1,100]
y_solution = solve_ivp(lambda t,Y: Rhs(t,Y,params),
     [tstart,tstop], IC, method='RK45',t_eval=tspan)
pyplot.plot(y_solution.t,
y_solution.y[0],'k',label='y(t)')
pyplot.ylabel('y(t) ',fontsize=16)
pyplot.xlabel('t',fontsize=16)
pyplot.savefig('Logistic.png')
```

A screenshot of the Spyder window shows the result of pushing play is shown in Figure 3.3.

3.4.2 Differential Equations in Matlab

Numerical solutions to differential equations in Matlab have similar requirements as in Python. We have to define the right-hand-sides, the initial conditions and the time interval. Matlab uses a set of packages (often referred to as the ODE-suite). There are several options includ-

ing `ODE45`, `ODE23s` and others. The most general one is `ODE45` and it works for most things that we will need. There are several ways to implement the right-hand-sides, but we will use a method similar in structure to Python.

The only difference is that Matlab uses two different types of functions. It is typical to include the functions in Python at the top of the script. Often codes in Matlab are separated into different m-files. One m-file describes the right-hand-side function and the other defines parameters, initial conditions and other inputs, solves the equations and plots them. A sample code for a scalar ODE with the right-hand-side function m-file, in Matlab is reproduced below. First the file `rhs.m` is defined,

```
function f=rhs(t,Y,params)
%this is the content of the right-hand-side.
%It takes
%inputs t, Y, and params and outputs f
r=params(1);
k=params(2);
y=Y;
f=r*y*(k-y);
```

The differential equation is solve using the m-file, `Logistic_example.m`,

```
%basic Scalar ODE using a m-file call
clear
close all

%define parameters
params=[.2,1];

t_start = 0;
t_stop = 100;
y_0=.1;
[t,y]=ode45(@(t,Y) rhs(t,Y,params), ...
       [t_start t_stop], y_0);
```

```
plot(t,y)
xlabel('Time')
ylabel('y(t)')
```

Note that both `Logistic_example.m` and `rhs.m` have to be in the same folder, where Matlab is currently running.

There are examples in other places where one defines an anonymous function that sits in the `traj_logistic.m` file. This is similar to how Python looks, although the functions for Python are introduced at the top of the script and at the bottom in Matlab.

3.5 Problems

Problems 3.1 *Complete all the problems in Matlab.*

(a) Use Matlab to calculate $5^{(3/2)} + e^{(.1)}$

(b) For loops: Using the syntax:

```
for i=1:.01:10
do something
end
```

Create an $m-file$ *that makes a vector* $x = [0, .1^2, .2^2, ...]$.

There are at least two ways to build arrays in Matlab using for loops. One is to 'append' to an existing array:

```
a=[]; % creates a memory location for a
for i= 1:.1:1
a=[a, i^2] % appends i^2 to a
end
```

A second way is to enter values into an existing array:

```
a=[]; % creates a memory location for a
for i= 1:10
a(i) = i^2 % puts i^2 into the ith location
end
```

Save the m-file as

`array_example.m`

(c) *There are two ways of defining functions. You can use an m-file or do it 'in-line'. I like to use m-file or function calls. The syntax is*

```
function f = fun1(x)
f = ***
```

This takes inputs of x and outputs $f(x) = ***$.

Use this to make a function $\frac{\cos(x)}{(1-x)}$.

(d) *Graphing the syntax for a basic plot is* `plot(x,f(x))`,

where x is a given array and f *is a defined function. Use this to plot* $f(x) = \frac{\cos(x)}{(1-x)}$, *where x is the array built in the first problem.*

(e) *Differential equations The syntax for solving an ODE,*

$$\begin{aligned} \frac{dx}{dt} &= f(t,x), \\ x(0) &= x_0 \end{aligned}$$

requires a few things: a function defining f, *an initial condition* x_0 *and the call for the ODE solver.*

An example code starts by defining an m-file that you save as

`fun1.m`

containing

```
function f=fun1(t,y)
f=-t*y/sqrt(2-y^2);
```

You also need an initial condition, say $x(0) = x_0 = 1$. Then an interval to solve the equation on, say $t = [0,5]$.

Then in the command window, type

```
[t_out, y_out]=ode45('fun1',[0 5],1);
plot(t_out, y_out)
```

Adjust this to vary the initial condition and interval of solution.

Problems 3.2 *For Python, create a file,* `example.py`. *The header of this should begin with*

```
from IPython import get_ipython
get_ipython().magic('reset -sf')
####################
import numpy as np
from matplotlib import pyplot
# import plotting library
from scipy.integrate import solve_ivp
```

Use this file to edit all the parts of this assignment.

(a) Use Python to calculate $5^{(3/2)} + e^{(.1)}$

(b) For loops: Using the syntax:

```
for x in my_array :
    do something
```

create a .py-file that makes a vector $x = [0, .1^2, .2^2, ...]$.

(c) Functions I tend to use inline functions for Python, so in the same Python file, using the syntax,

```
def f(x):
    f=***
    return f
```

This takes inputs of x and outputs $f(x) = ***$.

Use this to make a function $\frac{\cos(x)}{(1-x)}$.

(d) Graphing the syntax for a basic plot is

```
pyplot.plot(t,f)
```

where x is a given array and f *is a defined function. Use this to plot* $f(x) = \frac{\cos(x)}{(1-x)}$, *where x is the array built in the first problem.*

(e) Differential equations The syntax for solving an ODE,

$$\begin{aligned} \frac{dx}{dt} &= f(t,x), \\ x(0) &= x_0 \end{aligned}$$

requires a few things: a function defining f, *an initial condition* x_0 *and the call for the ODE solver.*

An example code starts by defining an py-file that you save as

```
fun1.py
```

containing

```
def fun1(t,y):
 f=-t*y/sqrt(2-y^2);
 return f
```

You also need an intitial condition, say $x(0) = x_0 = 1$. *Then an interval to solve the equation on, say* $t = [0,5]$.

```
import numpy as np
from matplotlib import pyplot
 # import plotting library
from scipy.integrate import odeint

# Define a function which
#calculates the derivative\index{derivative}
def Rhs(t, Y,params):
    y=Y
    [a,b]=params
    return a*y-b*t

xs = np.linspace(0,5,100)
params=[-1,.2]
y0 = 1.0  # the initial condition
ys =solve_ivp(lambda t,Y: Rhs(t,Y,params),
      [tstart,tstop], IC,
      method='RK45',t_eval=tspan)
# Plot the numerical solution
pyplot.plot(xs,ys)
```

Adjust this to vary the initial condition and interval of solution.

3.6 Appendix: Sample Scripts

3.6.1 Python

Logistic_example.py:

```
"""
Created on Wed Dec 22 21:11:32 2021
Basic ODE Script
@author: cogan
"""
from IPython import get_ipython
get_ipython().magic('reset -sf')
```

```
import numpy as np
from matplotlib import pyplot
# import plotting library
from scipy.integrate import solve_ivp
#Define the differential equation
def rhs(t,Y,params):
    r, K = params
    y=Y
    return [r*y*(K-y)]
 # Initial Condition
IC=.2
#Intial time
tstart=0
#Final time
tstop=4
tspan = np.linspace(tstart, tstop, 100)
#Parameters
params=[.1,100]
y_solution = solve_ivp(lambda t,Y:
rhs(t,Y,params), [tstart,tstop], [IC],
method='RK45',t_eval=tspan)
pyplot.plot(y_solution.t,
                 y_solution.y[0],
                 'k',label='y(t)')
pyplot.ylabel('y(t) ',fontsize=16)
pyplot.xlabel('t',fontsize=16)
#pyplot.savefig('Logistic.png')
```

3.6.2 Matlab

Logistic_example.m:

```
%basic Scalar ODE
clear
close all

%define parameters
params=[.1,100];
```

```
%Define the time interval
t_start = 0;
t_stop = 4;
%Initial Condition
IC=.2;
%Solving the ODE
[t,y]=ode45(@(t,Y) rhs(t,Y,params), ...
[t_start t_stop], IC);
%plotting the ODE
plot(t,y,LineWidth=2)
xlabel('Time')
ylabel('Y')
%Define the right-hand-side function
function f=rhs(t,Y,params)
r=params(1);
k=params(2);
y=Y;
f=r*y*(k-y);
end
```