2021 ASEE ANNUAL CONFERENCE
Virtual Meeting | July 26–29, 2021 | Pacific Daylight Time
Paper ID #32990

A Study of Differential Equation Solver Suites and Real-world Applications Using Python, Maple, and Matlab

Dr. Mohammad Rafiq Muqri, DeVry University, Ontario, CA

Dr. Mohammad R. Muqri is a Professor in College of Engineering and Information Sciences at DeVry University. He received his M.S.E.E. degree from University of Tennessee, Knoxville. His research interests include modeling and simulations, algorithmic computing, data analytics, analog and digital signal processing.

A study of Differential equation solver suites and real world applications using Python, Maple and Matlab

Introduction

The objective of this paper is to teach students how to solve ODEs using Python and Matlab's ODE solver programming tools. This teaching module will thus prepare our beginning junior electronics, computer, and bioengineering students before they encounter sensor/signal conditioning, processing, and other topics that they may delve into for their capstone senior project. Matlab also presents several tools for modeling linear systems.

This paper will explain how this learning and teaching module is instrumental for progressive learning of students; the paper will also demonstrate how the numerical and integral algorithms are derived and computed through leverage of the python data structures. As a result, there will be a discussion concerning the comparison of python and Matlab programming as well as students' feedback. The result of this new approach is expected to strengthen the capacity and quality of our undergraduate degree programs and enhance overall student learning and satisfaction.

Matlab a programming language developed by MathWorks, ,. It started out as a matrix programming language where linear algebra programming was simple. It can be run both under interactive sessions and as a batch job.

Maple is a symbolic and numeric computing environment as well as a multiparadigm programming language. It covers several areas of technical computing, such as symbolic mathematics, numerical analysis, data processing, visualization, and others. Maple has a toolbox known as MapleSim which adds functionality for multidomain physical modeling and code generation.

There are two basic types of boundary condition categories for ODEs – initial value problems and two-point boundary value problems. Initial value problems are simpler to solve because you only have to integrate the ODE one time. The solution of a two- point boundary value problem usually involves iterating between the values at the beginning and end of the range of integration. Runge-Kutta schemes are among the most commonly used techniques to solve initial-value problem ODEs.

Maple can manipulate mathematical expressions and find symbolic solutions to certain problems, such as those arising from ordinary and partial differential equations.
Depending on the application in as to how you define a problem, for different people with their peculiar background, Matlab, Maple and Python may end up being 'best' for that application. Actually, often, one will find that a mixture of a symbolic package, and a numeric package (or library), with a little glue programming, will be best. This is because for advanced applications, one probably really wants to do

1. symbolic model manipulation
2. symbolic model simplification
3. numeric model simulation
4. code generation (for efficiency)

Some experts have reported that the premise of Matlab is numerical computing. Depending on the application, say if one just wants to numerically compute eigenvalues, inverses, or numerically solve differential equations then probably Python is the way to go, because one can easily learn Python and make use of libraries like Numpy and SciPy with rich numerical computing tools and abundant community support and best of all it is free.

Introduction to ODEs

Differential equations are used to model a wide range of physical processes; technology students will use them in chemistry, biophysics, mechanics, thermodynamics, electronics, and almost every other scientific and engineering discipline. An ODE is used to express the rate of change of one quantity with respect to another. One defining characteristic of an ODE is that its derivatives are a function of one independent variable. The order of a differential equation is defined as the order of the highest derivative appearing in the equation and ODE can be of any order. A general form of a first-order ODE can be written in the form

$dy/dt + p(t)y + q(t) + s = 0$

where p(t) and q(t) are functions of t.

This equation can be rewritten as shown below $d/dt(y) + y\,p(t) = -q(t) - s$ where s is zero.

A classical integrating factor method can be used for solving this linear differential equation of first order. The integrating factor is $e^{\int p\,dt}$.

Euler Method

Graphical methods produce plots of solutions to first order differential equations of the form y’ = f(x,y), where the derivative appears on the left side of the equation. If an initial condition of the form y(x0) = y0 is also specified, then the only solution curve of interest is y’ = f(x,y) the one that passes through the intial point (x0,y0). For the first-order initial-value problem the popular graphical method also known as Euler method can be used that satisfies the formula given below yn+1 = yn + hf(xn ,yn) which can also be written as yn+1 = yn + h(y’n), where the approximate solution at xn is designated by y(xn), or simply yn. The true solution at xn will be denoted by either Y(xn) or Yn. Note that once yn is known, equation y’ = f(x,y) can be used to obtain yn’ as yn’ = f(xn ,yn) (1.0)

Modified Euler’s Method:

This is a simple predictor-corrector method that uses Euler’s method as the predictor and then uses the average value of y’ at both the left and right end points of the interval [xn, xn+1] (n= 0,1,2, …) as the slope of the line element approximation to the solution over that interval. The resuting equations are:

predictor: $y_{n+1} = y_n + h(y'_n)$ corrector:

$y_{n+1} = y_n + h/2$…...

For notational convenience, we designate the predicted value of y_{n+1} by py_{n+1}.

Since $y'_n = f(x_n, y_n)$, it then follows that

$$py'_{n+1} = f(x_{n+1}, y_{n+1}) \qquad (1.1)$$

and the modified Euler method becomes

predictor: $py_{n+1} = y_n + h(y'_n)$

corrector: $y_{n+1} = y_n + h/2(py'_{n+1} + y'_n)$ (1.2)

Example 1

Use the modified Euler's method to solve $y' - y + x = 0$; $y(0) = 2$ on the interval[0,1] with $h= 0.1$

Using Matlab dsolve function we get

```
>> dsolve('Dy = y - x', 'y(0)= 2','x')
```

the solution $y = x + \exp(x) + 1$

Euler's modified Numerical Method

Here $f(x,y) = y - x$, and $y_0 = 2$. From equation (1.0), we have $y_0 = f(0,2) = 2 - 0 = 2$

Using equations (1.1) and (1.2), we compute

For $n = 0$, $x_1 = 0.1$ $py_1 = y_0 + h\, y_0' = 2 + .1(2)$

$= 2.2$ $py_1' = f(x_1, py_1) = f(0.1, 2.2) = 2.2 - 0.1 =$

2.1 $y_1 = y_0 + h/2(py_1' + y_0') = 2 + 0.05(2.1 +$

$2) = 2.205$ y_1'

$= f(x_1, y_1) = f(0.1, 2.205) = 2.205 - 0.1 = 2.105$

It can be shown that for $n = 1$, $x_2 = 0.2$

$py_2 = y_1 + h\, y_1' = 2.4155$ $py_2' =$

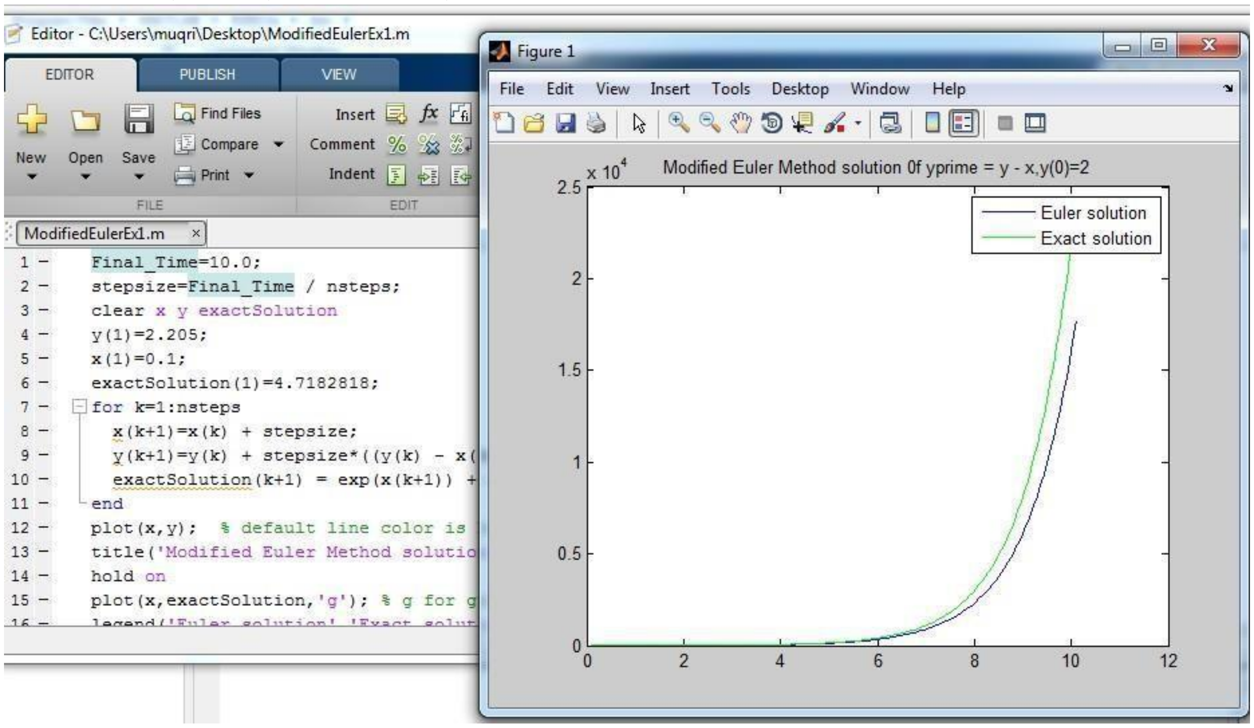
$f(x_2, py_2) = f(0.2, 2.4155) = 2.2155$ $y_2 = y_1 + h/2(py_2' + y_1') =$

2.421025 $y_2' = f(x_2, y_2) = f(0.2, 2.421025) = 2.221025$ and so on

Instead of computing x_n, y_n, and true solution etc. at different points, the following Matlab script can be used to obtain the Modified Euler solution as depicted in Figure below.

Code-Run

```
% nsteps = 150
Final_Time=10.0;
stepsize=Final_Time / nsteps;
clear x y exactSolution
y(1)=2.205; x(1)=0.1;
exactSolution(1)=4.7182818; for
k=1:
nsteps x(k+1)=x(k) + stepsize;
  y(k+1)=y(k) + stepsize*((y(k) - x(k)));
exactSolution(k+1) = exp(x(k+1)) + x(k+1) + 1;
end
plot(x,y);  % default line color is blue
title(‘Modified Euler Method solution of yprime = y – x, y(0) = 2’)
hold on
plot(x,exactSolution,'g'); % g for green
line legend('Euler solution','Exact solution') hold
off  error=norm(y-
exactSolution)/norm(exactSolution);
```



```
Final_Time=10.0;
stepsize=Final_Time / nsteps;
clear x y exactSolution
y(1)=2.205;
x(1)=0.1;
exactSolution(1)=4.7182818;
for k=1:nsteps
  x(k+1)=x(k) + stepsize;
  y(k+1)=y(k) + stepsize*((y(k) - x(
  exactSolution(k+1) = exp(x(k+1)) +
end
plot(x,y);  % default line color is
title('Modified Euler Method solutio
hold on
plot(x,exactSolution,'g'); % g for g
legend('Euler solution','Exact solut
```

Solve Differential Equation with ODEINT

Differential equations are solved in Python with the Scipy.integrate package using function **ODEINT**. Another Python package that solves differential equations is GEKKO. ODEINT requires three inputs:

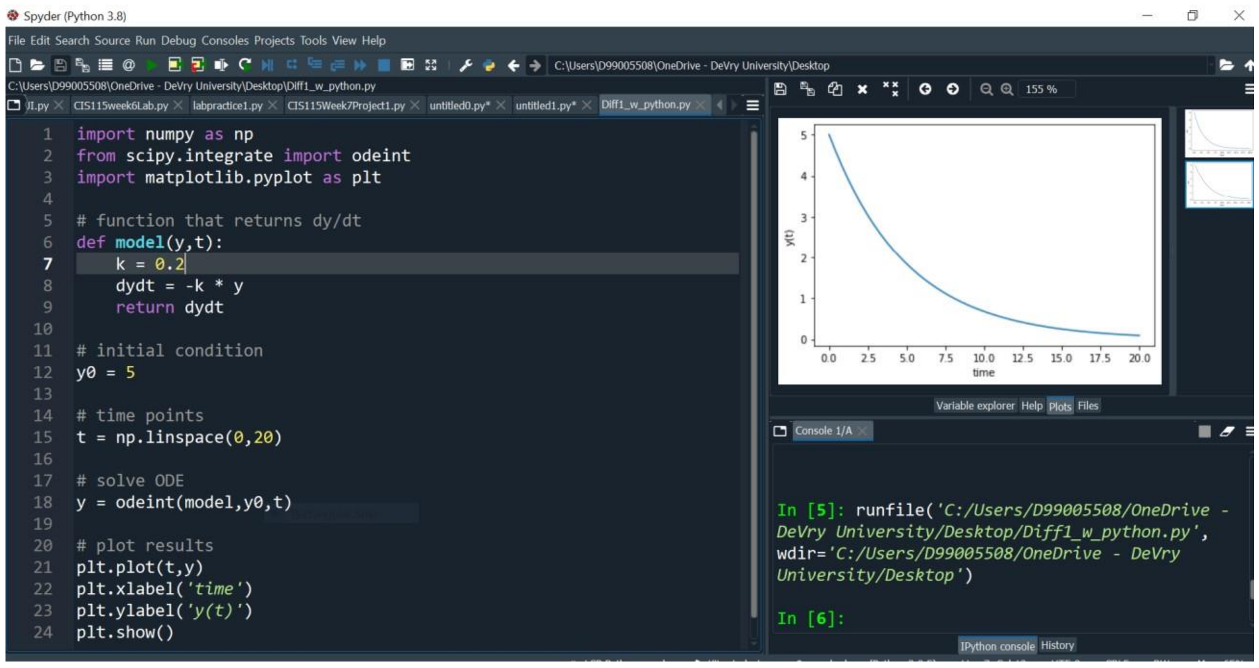
y = odeint(model, yo, t)

1. **model**: Function name that returns derivative values at requested y and t values as dydt = model(y,t)
2. **y0**: Initial conditions of the differential states
3. **t**: Time points at which the solution should be reported. Additional internal points are often calculated to maintain accuracy of the solution but are not reported. An example of using *ODEINT* is with the following differential equation with

parameter $k=0.3$, the initial condition $y_0=5$ and the following differential equation.

$$dy(t)/dt=-ky(t)$$

The Python code first imports the needed Numpy, Scipy, and Matplotlib packages. The model, initial conditions, and time points are defined as inputs to *ODEINT* to numerically calculate *y(t)*.

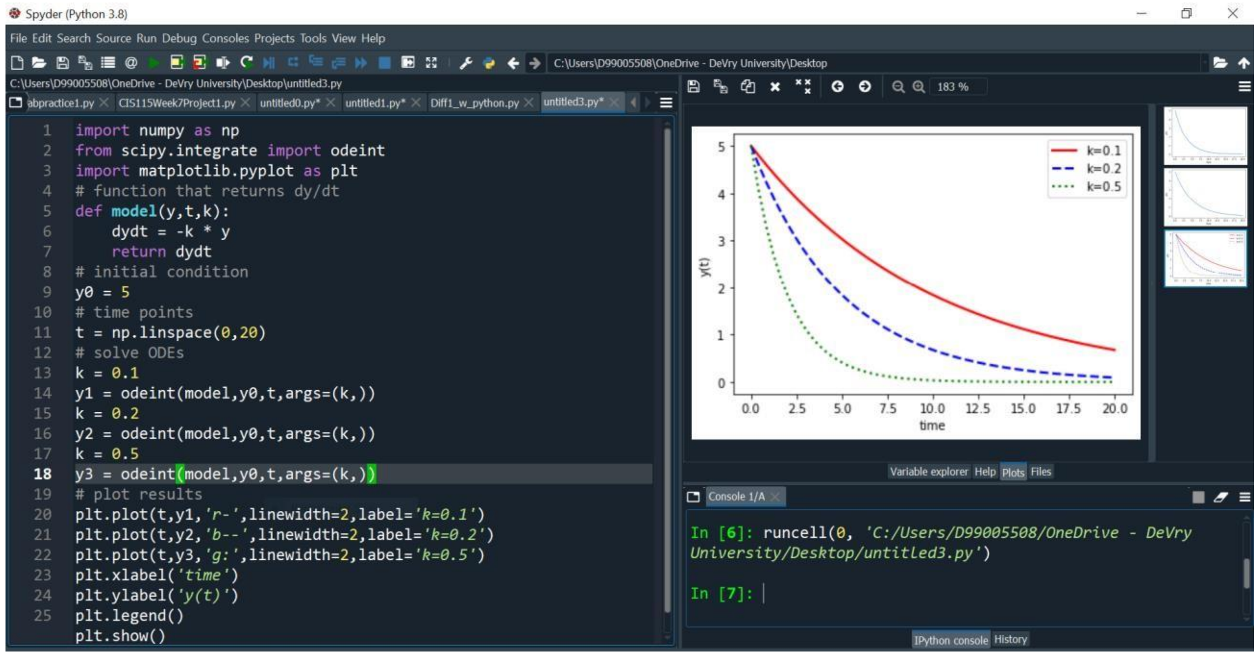

Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\D99005508\OneDrive - DeVry University\Desktop
C:\Users\D99005508\OneDrive - DeVry University\Desktop\Diff1_w_python.py
l.py CIS115week6Lab.py labpractice1.py CIS115Week7Project1.py untitled0.py* untitled1.py* Diff1_w_python.py
155 %
y(t)
time
Variable explorer Help Plots Files
Console 1/A
IPython console History

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

# function that returns dy/dt
def model(y,t):
    k = 0.2
    dydt = -k * y
    return dydt

# initial condition
y0 = 5

# time points
t = np.linspace(0,20)

# solve ODE
y = odeint(model,y0,t)

# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

```
In [5]: runfile('C:/Users/D99005508/OneDrive - DeVry University/Desktop/Diff1_w_python.py', wdir='C:/Users/D99005508/OneDrive - DeVry University/Desktop')

In [6]:
```

Figure 1. Python script to solve first order ODE using ODEINT

An optional fourth input is *args* that allows additional information to be passed into the *model* function. The *args* input is a tuple sequence of values. The argument *k* is now an input to the *model* function by including an addition argument.


Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\D99005508\OneDrive - DeVry University\Desktop
C:\Users\D99005508\OneDrive - DeVry University\Desktop\untitled3.py
abpractice1.py
CIS115Week7Project1.py
untitled0.py*
untitled1.py*
Diff1_w_python.py
untitled3.py*
183 %
k=0.1
k=0.2
k=0.5
y(t)
time
Variable explorer Help Plots Files
Console 1/A
IPython console History

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t,k):
    dydt = -k * y
    return dydt
# initial condition
y0 = 5
# time points
t = np.linspace(0,20)
# solve ODEs
k = 0.1
y1 = odeint(model,y0,t,args=(k,))
k = 0.2
y2 = odeint(model,y0,t,args=(k,))
k = 0.5
y3 = odeint(model,y0,t,args=(k,))
# plot results
plt.plot(t,y1,'r-',linewidth=2,label='k=0.1')
plt.plot(t,y2,'b--',linewidth=2,label='k=0.2')
plt.plot(t,y3,'g:',linewidth=2,label='k=0.5')
plt.xlabel('time')
plt.ylabel('y(t)')
plt.legend()
plt.show()
```

```
In [6]: runcell(0, 'C:/Users/D99005508/OneDrive - DeVry
University/Desktop/untitled3.py')

In [7]:
```

Figure 2. Revised Python script employing tuple sequence of values using ODEINT

Simulation Prediction of HIV Spread

The human immunodeficiency virus (HIV) infection spreads and can develop into acquired immunodeficiency syndrome (AIDS). AIDS can lead to immune system failure and eventual inability to defend the body against infection or cancer. Without treatment with antiretroviral drugs, survival time after infection with HIV is about 9 to 11 years, depending on a number of factors. Antiretroviral drugs such as TDF (tenofovir), either 3TC (lamivudine) or FTC (emtricitabine), and EFV (efavirenz) are recommended by the World Health Organization as soon as HIV infection is diagnosed. This simulation predicts the spread of HIV infection in a body with an initial infection.

The spread of HIV in a patient is approximated with balance equations on (H)ealthy, (I)nfected, and (V)irus population counts.

Objective: Simulate temperature response and compare to data from the TCLab A

simplified dynamic model of the Temperature Control Lab (TCLab) is the following:

$\tau p\ dT/dt = (Ta - T) + Kp\ Q$

τp is the time constant in seconds. Simulation is run at ambient 23 degree Celsius, and a gain of Kp= 0.8. With the temperature initially at ambient temperature (Ta), simulate the change in temperature over the 5 minutes when heater *Q* is adjusted to 50%. Use values of τ=120 sec, Kp=0.8^{o}C/%, and Ta=23^{o}C. Compare the simulated temperature response to data from the TCLab. Add a simulation prediction to the script below to compare with the TCLab data.

Student Exercices:

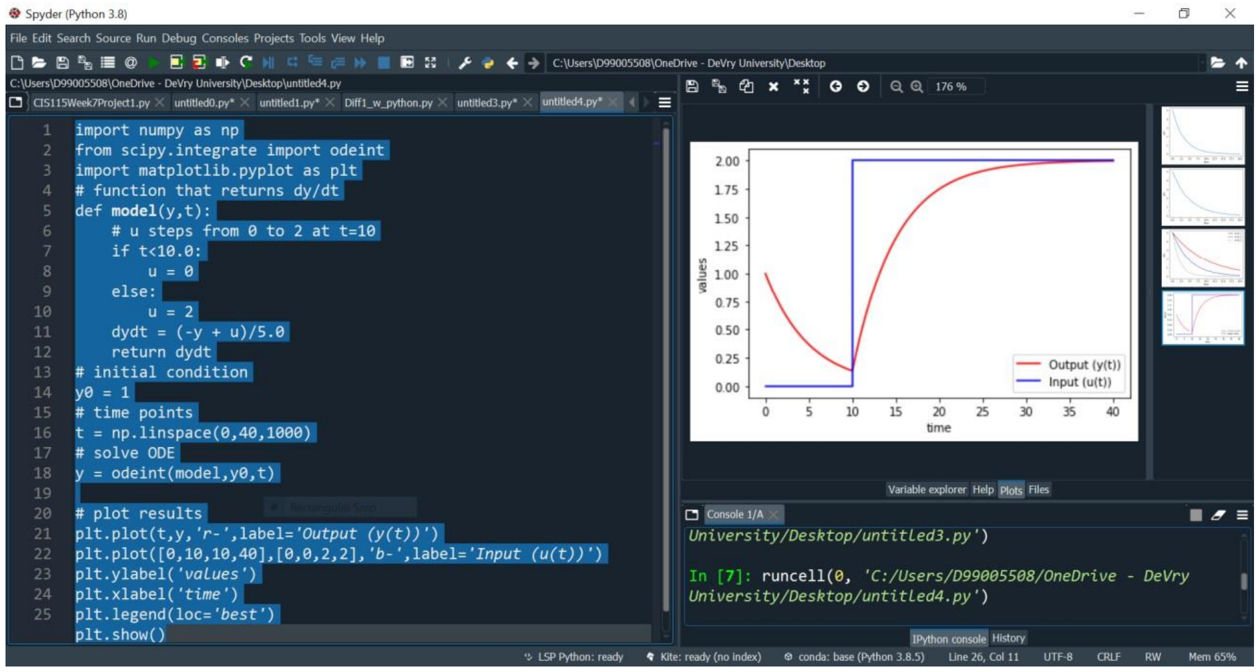
Find a numerical solution to the following differential equations with the associated initial conditions. Expand the requested time horizon until the solution reaches a steady state. Show a plot of the states (*x(t)* and/or *y(t)*). Report the final value of each state as $t \to \infty$

StudentExercise 1: $dy/dt + y(t) = 1$, $y(0) = 1$

StudentExercise 2: $5dy/dt + y(t) = u(t)$, $y(1) = 1$; u steps from 0 to 2 from t = 10

StudentExercise 3: Solve for x(t) and y(t) and show that the solutions are equivalent.
$dx(t)/dt=3\exp(-t)$

$dy(t)/dt=3-y(t)$,
where $x(0)=0$, $y(0)=0$
StudentExercise 3: is solved for the reference:



```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t):
    # u steps from 0 to 2 at t=10
    if t<10.0:
        u = 0
    else:
        u = 2
    dydt = (-y + u)/5.0
    return dydt
# initial condition
y0 = 1
# time points
t = np.linspace(0,40,1000)
# solve ODE
y = odeint(model,y0,t)

# plot results
plt.plot(t,y,'r-',label='Output (y(t))')
plt.plot([0,10,10,40],[0,0,2,2],'b-',label='Input (u(t))')
plt.ylabel('values')
plt.xlabel('time')
plt.legend(loc='best')
plt.show()
```

```
University/Desktop/untitled3.py')

In [7]: runcell(0, 'C:/Users/D99005508/OneDrive - DeVry
University/Desktop/untitled4.py')
```

Figure 3.

Simulation of Covid-19 Pandemic

Background

The COVID-19 epidemic reached the United States in early 2020, rapidly spreading across several states by March. To mitigate disease spread, states issued stay-at-home orders, closed schools and businesses, and put in place mask mandates. In major cities like New York City (NYC) and Chicago, the first wave ended in June. In the winter, a second wave broke out in both cities. Understanding why initial waves end and subsequent waves begin is key to being able to predict future epidemic dynamics.

Here's where modeling can help. But classical epidemiological models were developed almost 100 years ago. While these models are mathematically robust, they don't perfectly capture reality. One of their flaws is failing to account for the structure of person-to-person contact networks, which serve as channels for the spread of infectious diseases.

The python code snippets for analyzing the COVID-19 pandemic are explored and discussed further in Appendix A. The purpose of these notes is to introduce engineering technology students to quantitative modeling of infectious disease dynamics, and help them to modeling with ordinary differential equations. In this paper, covid-19 dynamics are modeled using a standard SEIR (Susceptible-Exposed Infected-Removed) model of disease spread, represented as a system of ordinary differential equations where the number of agents is large and there are no exogenous stochastic shocks.

In continuous-time stochastic processes, there is an important distinction between randomness that leads to continuous paths vs. those which have (almost surely right-continuous) jumps in their paths. The most tractable of these includes the theory of Levy Processes.

According to probability theory, a Levy process is a stochastic process with independent, stationary increments which represents the motion of a point whose successive displacements are random such that the displacements in pairwise disjoint time intervals are independent, and displacements in different time intervals of the same length have identical probability distributions. A Levy process therefore may be viewed as the continuous-time analog of a random walk.
Among the appealing features of Levy Processes is that they fit well into the sorts of Markov modeling techniques that economists tend to use in discrete time, and usually fulfill the measurability required for calculating expected present discounted values.

Unlike in discrete-time, where a modeler has license to be creative, the rules of continuous-time stochastic processes are much stricter. One can show that a Levy process's noise can be decomposed into two portions:

1. Weiner Processes (aka Brownian Motion) which leads to a diffusion equation, and is the only continuous-time Levy process with continuous paths
2. Poisson Processes with an arrival rate of jumps in the variable.

Modeling Shocks in COVID 19 with Stochastic Differential Equations

To discuss the basic SIR/SID model let us demonstrate another common compartmentalized model which modifies the previous model by removing the exposed state, and more carefully manages the death state, D.

The states are now: susceptible (S), infected (I), resistant (R), or dead (D). Please note that unlike the previous SEIR model, the R state is only for those recovered, alive, and currently resistant. As before, we start by assuming those have recovered have acquired immunity. Later, we could consider transitions from R to S if resistance is not permanent due to virus mutation, etc.

Transition Rates
Terminology and development of the model.
$\beta(t)$ is called the *transmission rate* or *effective contact rate* (the rate at which individuals bump into others and expose them to the virus)
γ is called the *resolution rate* (the rate at which infected people recover or die)
$\delta(t)\in[0,1]$ is the *death probability*

As before, we re-parameterize as $R0(t): =\beta(t)/\gamma$, where R0 has previous interpretation.

Jumping directly to the equations in s,i,r,d already normalized by N,

dsdidrdd=−γR0sidt=(γR0si−γi)dt=(1−δ)γidt=δγidt(1)(1)ds=−γR0sidtdi=(γR0si−γi)dtdr=(1−δ)γidt
dd=δγidt

Note that the notation has changed to heuristically put the dt on the right hand side, which will be used when adding the stochastic shocks.
A standard Monte Carlo simulation, a software program samples a random value from each input distribution and runs the model using those values. After repeating the process a number of times (typically 100 to 10,000), it estimates probability distributions for the uncertain outputs of the model from the random sample of output values. The larger the sample size, the more accurate the estimation of the output distributions. Microsoft Excel and other spreadsheets do not support Monte Carlo simulation directly. There are a number of software products that are add-ins to Excel that let you perform Monte Carlo simulation.

The best known are Oracle Crystal Ball and Palisade Software’s @Risk, which are neat products. What is different about Analytica is that Lumina designed Analytica from its inception to perform Monte Carlo simulation (and LHS methods), so, probabilistic analysis is kind of fully integrated into the product from the start. This gives Analytica’s Monte Carlo features certain advantages over spreadsheet add-ins, in terms of ease of use and speed of computation. You can define any variable, or any cell in an array, as a discrete or continuous distribution. You can view the probability distribution for any resulting variable as a set of probability bands (selected percentiles), as a probability density function, cumulative distribution function, or even view the underlying random sample.

Matlab has quite a few different ODE solvers, including ode23 which uses simultaneously second and third order Runge-Kutta formulas to make estimates of the error, and calculate the time step size. Since the second and third order Runge-Kutta methods require less steps, ode23 is considered less expensive in terms of computation demands than ode45, but it is also lower order.
These few examples demonstrate how students can be introduced not only to classic differential equations and numerical methods for solving differential equations, but also to basic concepts of object-based programming with Python. Given time, it can be proved that this group of students who are currently taking the junior-level signal and systems course have a far better understanding than the group of students who were never exposed to this teaching module.

Student Feedback

We administered an anonymous questionnaire to obtain feedback from the students in relation to the choice of using python and Matlab for solving ODEs. We have provided some selected questions from the questionnaire and their respective responses that apply particularly to the choice of programming language and lessons learned by early exposure to this module. Note that all the students' comments were encouraging and positive; there were almost no negative responses obtained.

- *Do you feel the learning module reinforced class material in a good, bad, or indifferent way?*

Sample Responses:

- Excellent. The python programs have greatly enhanced the overall class information.
- In an indifferent way. I like the computing with python but it was slightly cumbersome, may be because I am more comfortable with Matlab.
- Why python, why not C++? I was able to retrofit them using C++, although I like C#'s GUI development.
- These lab modules really set the pace for the class and enhance the experience.

- *Has the teaching module been straight forward and easy to understand?*

  Do you have suggestions for improvement?

Sample Responses:

- HIV model was Pretty straight forward, but covid-19 stochastic process based model was confusing.
- Add a section on covid-19 Markov's chain based Analysis using the Matlab Statistics Toolbox

 Did the Python software in conjunction with Matlab aid in understanding of the lab exercises?

Sample Responses:

- MatLab was very easy to use.

- Yes it did. The first few labs were OK, but Covid-19 pandemic simulation probably is difficult because I had to brush up on my python lists, tuples and file input/output for retrieving files.

- Yes. The use of python to solve ODEs was initially quite friendly and gave a fresh different outlook, but probably not the best when we delved into covid-19 ODEs.

 What other aspects do you feel would have been useful in this teaching module?

Sample Responses:

- When to use Runge-Kutta and when not to use it.

- Few more examples explaining use of ODE45, ODE23, GEKKO ODEs.

In summary, we can see from the above responses that students enjoyed this teaching module overall. Their responses to the choice of programming language were fairly positive, but more mixed as expected. The incorporation of many demonstrations and interactive lab experiences with python and Matlab into this teaching module proved to be effective.
I was fortunate to monitor and discuss the experiences of these students who took the senior project capstone class with me last year. As kind of expected they were very positive with the outcome and obtained an enhanced understanding of even application of second order differential equations which they attributed to their early exposure to python as well as the reinforcement of the usage of Matlab functions to solve ODEs.

Covid-19 Challenges and Discussion

Several approaches already exist for modeling the effect of heterogeneity on epidemic dynamics, but they typically assume heterogeneity remains constant over time. In this paper we assume to incorporate time variations in individual social activity into existing epidemiological models.

We also need to consider another important parameter the immunity factor, which tells us how much the reproduction number drops as susceptible individuals are removed from the population. The reproduction number indicates how transmissible an infectious disease is. Specifically, the quantity refers to how many people one infected person will in turn infect.

There is also a temporary state of immunity which may arise because population heterogeneity is not permanent. Afterall people change their social behavior over time. We have observed that individuals who self-isolated during the first wave--staying home, not having visitors over, ordering groceries online--subsequently start relaxing their behaviors, because any quick increase in social activity means additional exposure risk.

Basic epidemiological models we have used only have one characteristic time, called the generation interval or incubation period. It actually refers to the time when someone can infect another person after becoming infected himself. For COVID-19, it's roughly five days. But that's only one timescale. There are other timescales over which people change their social behavior.
As such a complicated, multidimensional model is needed to describe each group of people with different susceptibilities to disease. which is way beyond the scope of this paper. Although we tend to oversimplify by compressing this model into only three equations, developing a single parameter to capture biological and social sources of heterogeneity.

In conclusion, it can be stated that with proper guidance, monitoring, and diligent care, engineering technology students can be exposed earlier to ODEs, python data structures and the basics of Matlab, and Simulink. This will go a long way in motivating them, eliminating their fear, improving their understanding, and enhancing their quality of education.

Bibliography

1. Muqri, M., Shakib, J., *A Taste of Java-Discrete and Fast Fourier Transforms,* American Society for Engineering Education, AC 2011-451.
2. Shakib, J., Muqri, M., *Leveraging the Power of Java in the Enterprise*, American Society for Engineering Education, AC 2010-1701.
3. Hambley, Alan R., *Electrical Engineering Principles and Applications,* Prentice Hall, 2011.
4. Blinowska, K., Durka, P., *The application of wavelet transforms and matching pursuit to the time- varying EEG signals,* in Intelligent Engineering Systems Through Artificial Neural Networks, Editors, Dagli & Fernandez, volume 4, pp. 535-540, ASME Press, New York, l994.
5. Murach, J., Urban, M., *Python Programming*, Mike Murach & assciates, Inc, 2016.
6. Lee Wei-Ming., - *Python Machine Learning* – Willey, 2019.
7. Hambley, Alan R., *Electrical Engineering Principles and Applications*, Prentice Hall, 2011.

Appendix A: Python Classes ODE and ODE Solver

Stochastic Processes

Monte Carlo methods have three characteristics:

1. Random sample generation
2. Known input distribution
3. Numerical experiments

The direct output of the Monte Carlo simulation method is the generation of random sampling. Other performance or statistical outputs are indirect methods which depend on the applications. There are many different numerical experiments that can be done, probability distribution is one of them. Probability distribution identifies either the probability of each value of an unidentified random variable (when the variable is discrete), or the probability of the value falling within a particular interval (when the variable is continuous). That is equivalent to saying that for random variables X with the distribution in question, Pr[X = a] = 0 for all real numbers a. That is, the probability that X attains the value a is zero, for any number a. If the distribution of X is continuous, then X is called a continuous random variable. Normal distribution, continuous uniform distribution, beta distribution, and Gamma distribution are well known absolutely continuous distributions.

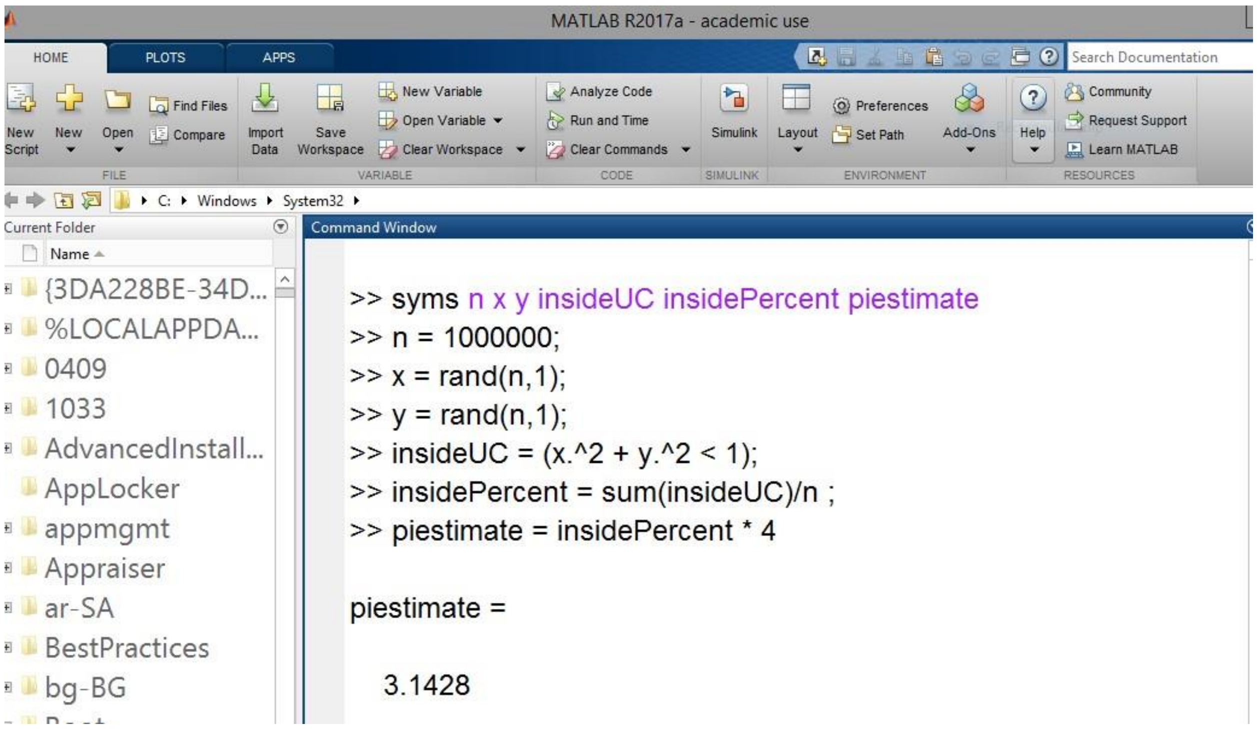
A standard Monte Carlo simulation, a software program samples a random value from each input distribution and runs the model using those values. After repeating the process a number of times (typically 100 to 10,000), it estimates probability distributions for the uncertain outputs of the model from the random sample of output values. The larger the sample size, the more accurate the estimation of the output distributions. Microsoft Excel and other spreadsheets do not support Monte Carlo simulation directly. There are a number of software products that are add-ins to Excel that let you perform Monte Carlo simulation. These are explained in detail in Appendix A.

Simple Monte Carlo Estimation Examples

Monte Carlo methods is a class of numerical methods that relies on random sampling. If you had a circle and a square where the length of a side of the square was the same as the diameter of the circle, the ratio of the area of the circle to the area of the square would be $\pi/4$. So, if you put this circle inside the square and select many random points inside the square, the number of points inside the circle divided by the number of points inside the square and the circle would be approximately $\pi/4$

Consider the following Monte Carlo method which computes the value of π:

1. Uniformly scatter some points over a unit square [0,1]×[0,1].
2. For each point, determine whether it lies inside the unit circle.
3. The percentage of points inside the unit circle is an estimate of the ratio of the area inside the circle and the area of the square,which is π/4. Multiply the percentage by 4 to estimate π. The Matlab script depicted below in Figure 1 performs the Monte Carlo computation:



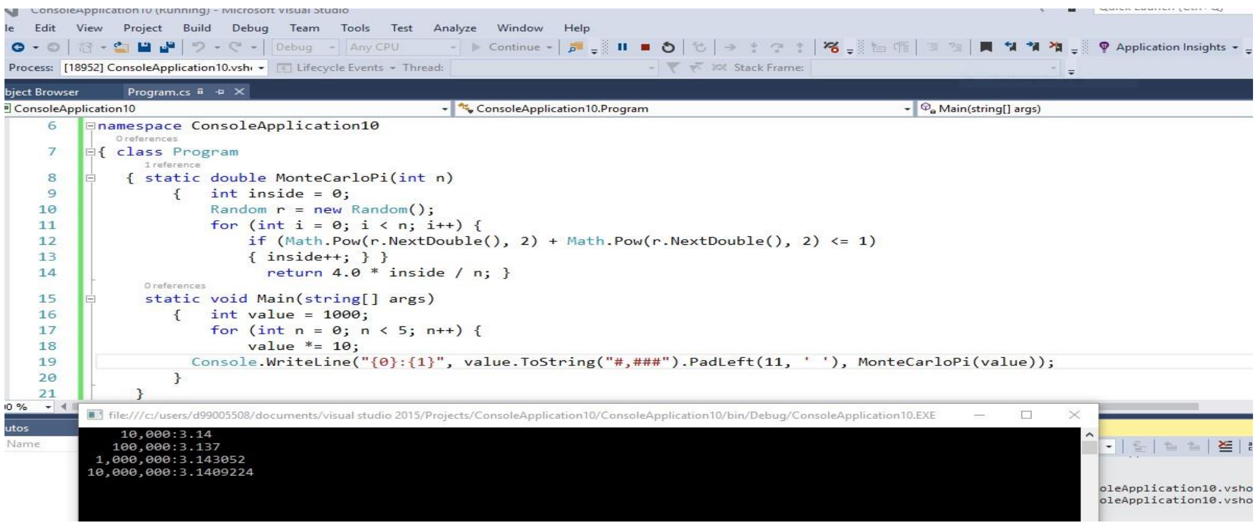
```
>> syms n x y insideUC insidePercent piestimate
>> n = 1000000;
>> x = rand(n,1);
>> y = rand(n,1);
>> insideUC = (x.^2 + y.^2 < 1);
>> insidePercent = sum(insideUC)/n ;
>> piestimate = insidePercent * 4

piestimate =

    3.1428
```

Figure 4. Monte Carlo Matlab computation to estimate pi

This example represent a general procedure of Monte Carlo methods: First, the input random variables (*x* and *y*) are sampled. Second, for each sample, a calculation is performed to obtain the outputs (whether the point is inside or not). Due to the randomness in the inputs, the outputs are also random variables. Finally, the statistics of the output random variables (the percentage of points inside the circle) are computed, which estimates the output.
Students were then given the handout and advised to compute value of pi for each value of *n* = 100, *n* = 10000 and *n* = 1000000, run the script 3 times.

Ex.1: Compute and interpret as to how accurate is the estimated *pi* using:

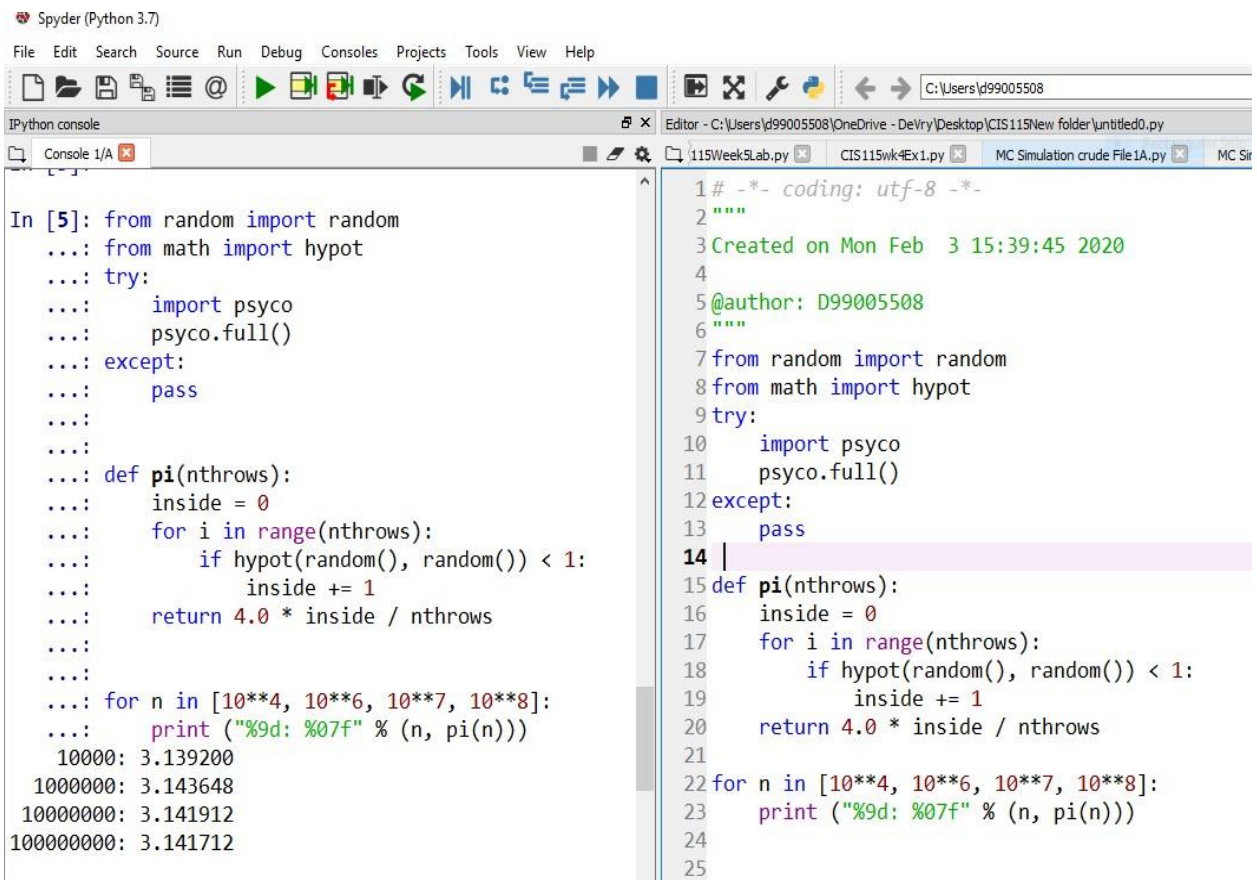
1. MatLab
2. Microsoft Visual Studio and C#
3. Python



```
namespace ConsoleApplication10
{ class Program
  { static double MonteCarloPi(int n)
        {    int inside = 0;
             Random r = new Random();
             for (int i = 0; i < n; i++) {
                 if (Math.Pow(r.NextDouble(), 2) + Math.Pow(r.NextDouble(), 2) <= 1)
                 { inside++; } }
                   return 4.0 * inside / n; }
    static void Main(string[] args)
        {    int value = 1000;
             for (int n = 0; n < 5; n++) {
                 value *= 10;
           Console.WriteLine("{0}:{1}", value.ToString("#,###").PadLeft(11, ' '), MonteCarloPi(value));
        }
    }
```

```
     10,000:3.14
    100,000:3.137
  1,000,000:3.143052
 10,000,000:3.1409224
```

Figure 5. C# Program to compute the value of PI


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\d99005508
IPython console
Console 1/A
Editor - C:\Users\d99005508\OneDrive - DeVry\Desktop\CIS115New folder\untitled0.py
115Week5Lab.py
CIS115wk4Ex1.py
MC Simulation crude File1A.py

```
In [5]: from random import random
   ...: from math import hypot
   ...: try:
   ...:     import psyco
   ...:     psyco.full()
   ...: except:
   ...:     pass
   ...:
   ...:
   ...: def pi(nthrows):
   ...:     inside = 0
   ...:     for i in range(nthrows):
   ...:         if hypot(random(), random()) < 1:
   ...:             inside += 1
   ...:     return 4.0 * inside / nthrows
   ...:
   ...:
   ...: for n in [10**4, 10**6, 10**7, 10**8]:
   ...:     print ("%9d: %07f" % (n, pi(n)))
    10000: 3.139200
  1000000: 3.143648
 10000000: 3.141912
100000000: 3.141712
```

```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb  3 15:39:45 2020

@author: D99005508
"""
from random import random
from math import hypot
try:
    import psyco
    psyco.full()
except:
    pass

def pi(nthrows):
    inside = 0
    for i in range(nthrows):
        if hypot(random(), random()) < 1:
            inside += 1
    return 4.0 * inside / nthrows

for n in [10**4, 10**6, 10**7, 10**8]:
    print ("%9d: %07f" % (n, pi(n)))
```

Figure 6. Python program to compute the value of PI using a function

Modeling Shocks in COVID 19 with Stochastic Differential Equations

Unlike in discrete-time, where a modeler has license to be creative, the rules of continuous-time stochastic processes are much stricter. One can show that a Levy process's noise can be decomposed into two portions:

1. Weiner Processes (aka Brownian Motion) which leads to a diffusion equation, and is the only continuous-time Levy process with continuous paths
2. Poisson Processes with an arrival rate of jumps in the variable.

To discuss the basic SIR/SID model let us demonstrate another common compartmentalized model which modifies the previous model by removing the exposed state, and more carefully manages the death state, D.

The states are now: susceptible (S), infected (I), resistant (R), or dead (D). Please note that unlike the previous SEIR model, the R state is only for those recovered, alive, and currently resistant. As before, we start by assuming those that have recovered have acquired immunity. Later, we could consider transitions from R to S if resistance is not permanent due to virus mutation, etc.

Transition Rates

Terminology and development of the model.

β(t) is called the *transmission rate* or *effective contact rate* (the rate at which individuals bump into others and expose them to the virus)

γ is called the *resolution rate* (the rate at which infected people recover or die)

δ(t)∈[0,1] is the *death probability*

As before, we re-parameterize as R0(t): =β(t)/γ, where R0 has previous interpretation.

Jumping directly to the equations in s,i,r,d already normalized by N,

dsdidrdd=−γR0sidt=(γR0si−γi)dt=(1−δ)γidt=δγidt(1)(1)ds=−γR0sidtdi=(γR0si−γi)dtdr=(1−δ)γidt
dd=δγidt

Note that the notation has changed to heuristically put the dt on the right hand side, which will be used when adding the stochastic shocks.

There are a number of software products that are add-ins to Excel that let you perform Monte Carlo simulation.

Next a model of the coronavirus (COVID-19) disease using simple rates of transmission is considered.

Let $N(t)$ be the total population of human. This population is divided into seven classes:
susceptible individuals $S(t)$,
exposed individuals $E(t)$,
asymptotically infected individuals $I_A(t)$,
symptomatic infected individuals $I_S(t)$,
quarantined individuals $Q(t)$,
and individuals that have recovered/remove from COVID-19 $R(t)$.
Based on this consideration,
the total population is $N(t) = S(t)+E(t)+Q(t)+I_A(t)+I_S(t)+R(t)$.

The natural human natality and mortality rates are denoted by Λ and μ respectively. Susceptible individuals (S) gets infected from enough contact with exposed individuals (E) at the rate of β or just move to quarantined class at the rate of τ. The exposed individuals (E) may move to quarantined (Q) class first or get infected without symptoms (asymptomatic) (I_A) or with symptoms (symptomatic) (I_S) at the rates of γ,σ and η respectively. Also quarantined individuals (Q) may be confirmed infected through a test with symptoms (I_S) or without symptoms (I_A) at the rates of υ and θ respectively. The asymptomatic infected individuals (I_A) may recover at the rate of r_1 and the symptomatic infected individuals (I_S) at the rate of r_2.

Each of these classes may decrease as a result of natural mortality μ, while the class of individuals infected with symptoms (I_S) may also decrease as a result of death from the disease at the rate of δ. In the infected class of individuals without symptoms (I_A), the death as a result of the disease is not considered. The possibility of reinfection after recovery has not been considered in this model.

Let us mathematical formulation to depict the spread of COVID-19. Using recursion and subsequent simplification stepsa system of nonlinear differential equations presented below:

$$dS(t)dtdE(t)dtdQ(t)dtdIA(t)dtdIS(t)dtdR(t)dt=\Lambda-(\tau+\mu)S(t)-\beta S(t)E(t),=\beta S(t)E(t)-(\gamma+\mu+\eta+\sigma)E(t),=\tau S(t)+\gamma E(t)-(\mu+\upsilon+\theta)Q(t),=\sigma E(t)+\theta Q(t)-(\mu+r1)IA(t),=\eta E(t)+\upsilon Q(t)-(\delta+\mu+r2)IS(t),=r1IA(t)+r2IS(t)-\mu R(t), \tag{2.1}$$

subject to the following initial conditions:

$$S(0)\geq 0,E(0)\geq 0,Q(0)\geq 0,IA(0)\geq 0,IS(0)\geq 0,R(0)\geq 0. \tag{2.2}$$

Introduction to SDEs
We start by extending our model to include randomness in R0(t)R0(t) and then the mortality rate δ(t)δ(t). The result is a system of Stochastic Differential Equations (SDEs). Shocks to Transmission Rates

As before, we assume that the basic reproduction number, R0(t)R0(t), follows a process with a reversion to a value R¯0(t)R¯0(t) which could conceivably be influenced by policy. The intuition is that even if the targeted R¯0(t)R¯0(t) was changed through social distancing/etc., lags in behavior and implementation would smooth out the transition, where ηη governs the speed of R0(t)R0(t) moves towards R¯0(t)R¯0(t).

Beyond changes in policy, randomness in R0(t)R0(t) may come from shocks to the β(t)β(t) process. For example,

- Misinformation on Facebook spreading non-uniformly.
- Large political rallies, elections, or protests.
- Deviations in the implementation and timing of lockdown policy between demographics, locations, or businesses within the system.
- Aggregate shocks in opening/closing industries.

To implement these sorts of randomness, a diffusion term can be added with an instantaneous volatility of σR0−−−√σR0.

- This equation is used in the Cox-Ingersoll-Ross and Heston models of interest rates and stochastic volatility.
- The scaling by the R0−−−√R0 ensure that the process stays weakly positive. The heuristic explanation is that the variance of the shocks converges to zero as R₀ goes to zero, enabling the upwards drift to dominate.

The notation for this SDE is then

dR0t=η(R¯0t−R0t)dt+σR0t−−−√dWt(2)(2)dR0t=η(R¯0t−R0t)dt+σR0tdWt

where WW is standard Brownian motion (i.e a Weiner Process).

Heuristically, if σ=0, divide this equation by dt and it nests the original ODE used in the previous lecture.

While we do not consider any calibration for the σ parameter, empirical studies such as Estimating and Simulating a SIRD Model of Covid -19 for Many Countries, States, and Cities demonstrate highly volatile R0(t)R0(t) estimates over time.

Even after lockdowns are first implemented, we see variation between 0.5 and 1.5. Since countries are made of interconnecting cities with such variable contact rates, a high σσ seems reasonable both intuitively and empirically.

Mortality Rates

Unlike the approach taken previously, one can tend to build up towards mortality rates which change over time.

Imperfect mixing of different demographic groups could lead to aggregate shocks in mortality (e.g. if a retirement home is afflicted vs. an elementary school). These sorts of relatively small changes might be best modeled as a continuous path process.

Let $\delta(t)$ be the mortality rate and in addition,

- Assume that the base mortality rate is $\bar{\delta}$, which acts as the mean of the process, reverting at rate θ. In more elaborate models, this could be time varying.
- The diffusion term has a volatility) $\sqrt{\xi\delta(1-\delta)}$.
- As the process gets closer to either $\delta=1$ or $\delta=0$, the volatility goes to 0, which acts as a force to allow the mean reversion to keep the process within the bounds
- Unlike the well-studied Cox-Ingersoll-Ross model, we make no claims on the long-run behavior of this process, but will be examining the behavior on a small timescale so this is not an issue.

Given this, the stochastic process for the mortality rate is,

$$d\delta_t=\theta(\bar{\delta}-\delta_t)dt+\xi\sqrt{(\delta_t(1-\delta_t)}dW_t \quad (3)$$

Where the W_t Brownian motion is independent from the previous process.

System of SDEs

The system (1) can be written in vector form $x:=[s,i,r,d,R_0,\delta]$ with parameter tuple $p:=(\gamma,\eta,\sigma,\theta,\xi,\bar{R}_0(\cdot),\bar{\delta})$ The general form of the SDE is.

$$dx_t=F(x_t,t;p)dt+G(x_t,t;p)dW$$

With the drift,

$$F(x,t;p):=\begin{bmatrix}-\gamma R_0 s i\\ \gamma R_0 s i-\gamma i\\ (1-\delta)\gamma i\\ \delta\gamma i\\ \eta(\bar{R}_0(t)-R_0)\\ \theta(\bar{\delta}-\delta)\end{bmatrix}$$

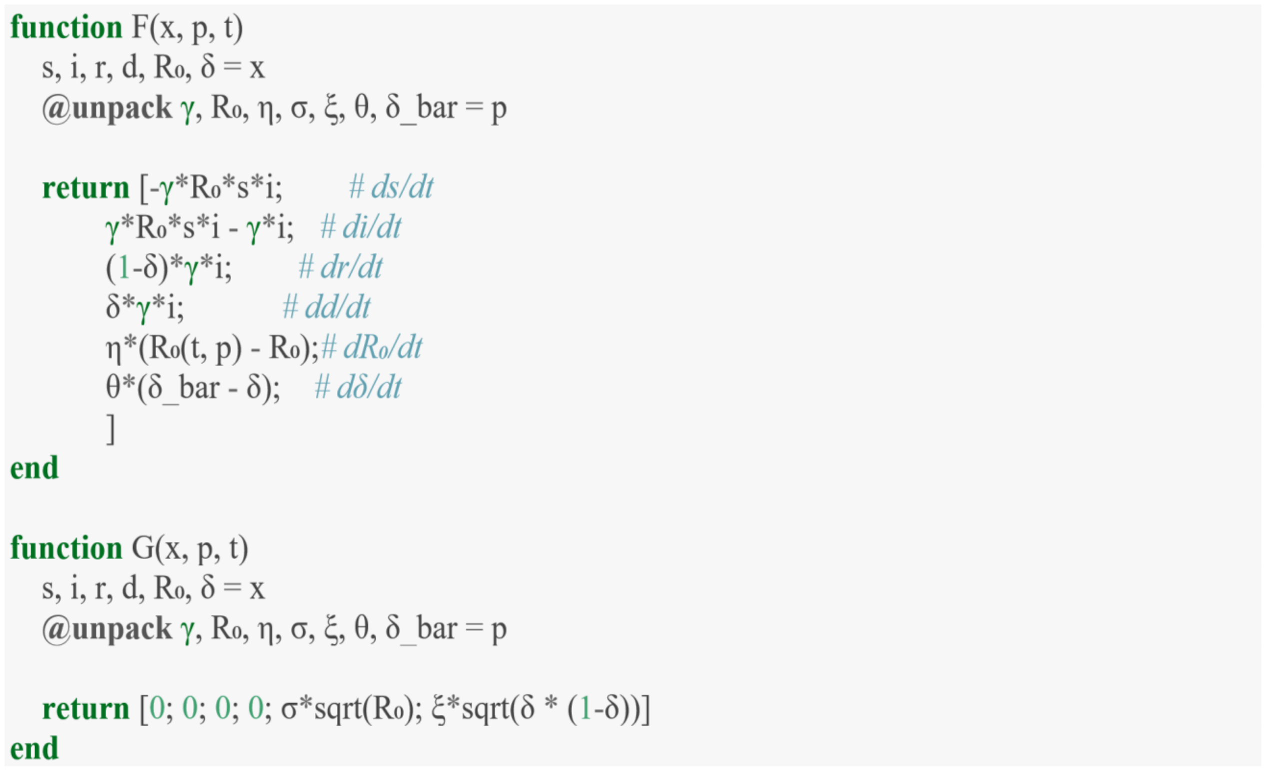
Here, it is convenient but not necessary for dW to have the same dimension as x. If so, then we can use a square matrix G(x,t;p) to associate the shocks with the appropriate x (e.g. diagonal noise, or using a covariance matrix).

As the two independent sources of Brownian motion only affect the dR0 and dδ terms (i.e. the 5th and 6th equations), define the covariance matrix as

G(x,t):=diag([0000σR0−−−√ξδ(1−δ)−−−−−−−−√])(5)(5)G(x,t):=diag([0000σR0ξδ(1−δ)])

Implementation

First, construct our FF from (4) and GG from (5)



```
function F(x, p, t)
    s, i, r, d, R₀, δ = x
    @unpack γ, R₀, η, σ, ξ, θ, δ_bar = p

    return [-γ*R₀*s*i;        # ds/dt
            γ*R₀*s*i - γ*i;   # di/dt
            (1-δ)*γ*i;        # dr/dt
            δ*γ*i;            # dd/dt
            η*(R₀(t, p) - R₀);# dR₀/dt
            θ*(δ_bar - δ);    # dδ/dt
            ]
end

function G(x, p, t)
    s, i, r, d, R₀, δ = x
    @unpack γ, R₀, η, σ, ξ, θ, δ_bar = p

    return [0; 0; 0; 0; σ*sqrt(R₀); ξ*sqrt(δ * (1-δ))]
end
```

G (generic function with 1 method)