1 Introduction to Matplotlib: 3D Plotting and Animations

Lab Objective: *3D plots and animations are useful in visualizing solutions to ODEs and PDEs found in many dynamics and control problems. In this lab we explore the functionality contained in the 3D plotting and animation libraries in Matplotlib.*

Introduction

Matplotlib is a Python library that contains tools for creating plots in multiple dimensions. The library contains important classes that are needed to create plots. The most important objects to understand in this lab are figure objects, axes objects, and line objects. These three objects are created using the following code.

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()                  # Create figure object.
>>> ax = fig.add_subplot(111)           # Create axes object.
>>> line2d, = plt.plot([],[])           # Create empty 2D Line object
>>> line3d, = plt.plot([],[],[])        # Create empty 3D line object
```

Recall that `plt.figure()` creates a `matplotlib.figure.Figure` object, which is the window that is displayed when `plt.show()` is called. 3D plotting and animation both require explicitly defining the `Figure` object, as shown above. This allows for the object to be updated and modified, as will be explained later in the lab.

`Figure` objects contain `matplotlib.axes._subplots.AxesSubplot` objects, called *axes*. Axes are spaces to plot on, and are created by the `add_subplot()` method of a `Figure` object. Figures can have multiple axes.

Calling `plt.plot()` returns a list of line objects. For example, supposing `x1`, `y1`, `x2`, and `y2` are arrays containing data for two separate curves, then calling `plt.plot(x1, y1, x2, y2)` will return a list with two elements. Each element of the list is a `matplotlib.lines.Line2D` object. If the axes is three-dimensional, then the returned list will contain `matplotlib.lines.Line3D` objects. Because this function call returns a list, if only one line is plotted, adding a trailing comma to the variable name will assign the name to the first element of the returned list. You can alternatively reference the zero index of the returned list, but using a trailing comma is standard.

Animation Background

The animation library in Matplotlib contains a class called `FuncAnimation`. We will use this class throughout this lab. `FuncAnimation` requires a user-defined *update* function that controls the plot for each frame of the animation. This grants the user wide flexibility and control of the resulting animation. The following steps describe the process of creating a simple animated plot using the `FuncAnimation` class.

1. Compute all data to be plotted.
2. Explicitly define figure object.
3. Define line objects to be altered dynamically.
4. Create function to update line objects.
5. Create `FuncAnimation` object.
6. Display using `plt.show()`.

These steps will be explained by way of an example. The arrays `x` and `y` contain data giving the location of a particle moving in the plane. To visualize this motion, one could animate the particle as well as display the trajectory that the particle has traveled. For this animation, two separate `Line2D` objects must be created on an axes object. The first, `particle` will be for the position of the particle itself, and the second, `traj` will be for the trajectory that the particle has traveled. Note that these objects are created with empty lists of data. The update function will be used to dynamically set the data to be plotted in these line objects.

```
>>> import matplotlib.animation as animation
>>> import numpy as np
>>> t = np.linspace(0,2*np.pi,100)
>>> x = np.sin(t)
>>> y = t**2
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.set_xlim((-1.1,1.1))
>>> ax.set_ylim((0,40))
>>> particle, = plt.plot([],[], marker='o', color='r')
>>> traj, = plt.plot([],[], color='r', alpha=0.5)
```

The update function must be defined a specific way in order to interact properly with the `matplotlib.animation.FuncAnimation` object. The update function must accept the current frame index as its first input parameter and it must return a list or tuple of line objects. The current frame index is used to access the data to be plotted in the current frame. Both 2D and 3D line objects have the built-in method `.set_data()`. This function takes in two one-dimensional arrays representing x and y values to plot. This allows a single line object to display different data for each frame. Inside the update function, `.set_data()` is called on the line objects with the relevant data as inputs.

```
>>> def update(i):
>>>     particle.set_data(x[i],y[i])
>>>     traj.set_data(x[:i+1],y[:i+1])
>>>     return particle,traj
```

Next, the `FuncAnimation` object is created. The argument `frames` specifies the iterable representing the frame indices. If `frames` is an integer, it is treated as the iterable `range(frames)`. After the `FuncAnimation` object is created, `plt.show()` displays the animation.

```
>>> ani = FuncAnimation(fig, update, frames=range(100), interval=25)
>>> plt.show()
```

The following table shows more parameters that can be passed into FuncAnimation.

Parameter	Description
`fargs` (tuple)	Additional arguements to pass update function
`interval` (float)	Delay between frames in milliseconds
`repeat` (bool)	Determines whether animation repeats (Default True)
`blit` (bool)	Determines whether blitting is used (Default False)

> **Note**
>
> When using FuncAnimation, it is essential that a reference is kept to the instance of the class. The animation is advanced by a timer and if a reference is not held for the object, `Python` will automatically garbage collect and the animation will stop.

Embedding Animations

Some systems may struggle with using `plt.show()` to display an animation. In this case, it may be easier to embed the animation using the HTML5 API. Jupyter notebooks use HTML to display their contents, so we can leverage this and use HTML5's video capabilities to insert video directly into a notebook.

To embed the video directly into a notebook using HTML5 you must use the `IPython.display` module. This module will be able to interpret an encoded HTML5 video, which `matplotlib.animation` can create. This method tends to be much more simple than rendering the animation to an `.mp4` file and then embedding that file into a notebook, as it does not require an outside encoder, and it tends to be encounter fewer bugs than using `plt.show()`. Here is a snippet you may reference to embed an animation using `IPyton.display`

```
# required import statements
from IPython.display import HTML
import matplotlib.pyplot as plt
from matplotlib import animation

# disable interactive mode
plt.ioff()
'''
Here we would insert whatever code needed to create the animation
such as instantiating the fig object and defining the update function
'''
# create animation
ani = animation.FuncAnimation(fig, update, frames=N, interval=i)
```

```
# render as html5 and embed
HTML(ani.to_html5_video())
```

Problem 1. Use the FuncAnimation class to animate the function $y = \sin(x + 0.1t)$ where $x \in [0, 2\pi]$, and t ranges from 0 to 100 seconds.

3D Plotting Introduction

3D plotting is very similar to 2D plotting. The main difference is that a set of 3D axes must be created within the figure object. A 3D axes object is created using the additional keyword argument `projection='3d'`, as shown below. Note that the Axes3D submodule must first be imported in order to create the 3D axis.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>>
>>> # Create figure object.
>>> fig = plt.figure()
>>>
>>> # Create 3D axis object using add_subplot().
>>> ax = fig.add_subplot(111, projection='3d')
```

3D Static Plotting

When the axes object is explicitly defined, plots are generated by calling the chosen plot function (such as `ax.plot()` on the axes object. Additional information on the use of axes objects can be found here: https://matplotlib.org/api/axes_api.html.

Problem 2. The orbits for Mercury, Venus, Earth, and Mars are stored in the file `orbits.npz`. The file contains four NumPy arrays: `mercury`, `venus`, `earth`, and `mars`. The first column of each array contains the x-coordinates, the second column contains the y-coordinates, and the third column contians the z-coordinates of each planet, all relative to the Sun, and expressed in AU (astronomical units, the average distance between Earth and the Sun, approximately 150 million kilometers).

Use `np.load('orbits.npz')` to load the data for the four planets' orbits. Create a 3D plot of the orbits, and compare your results with Figure 1.1.

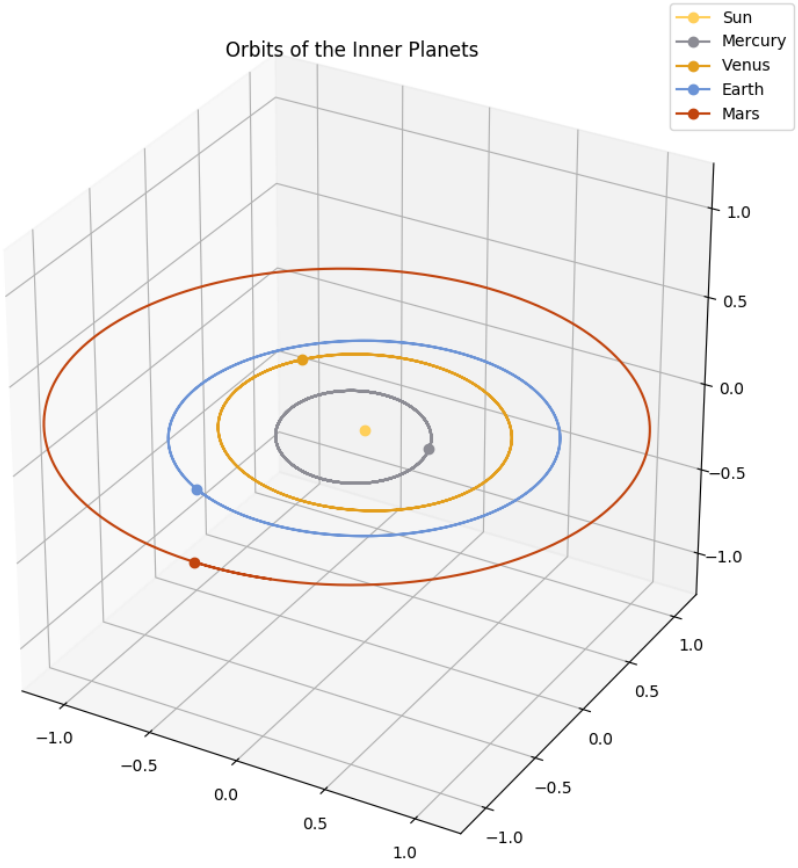


Figure 1.1: The solution to Problem 2.

3D Animations

The key difference between 2D and 3D animations is that the `.set_data()` method does not support setting the `z` values. Instead, set the `x` and `y` values with `.set_data()` as before, and then set the `z` values with `.set_3d_properties()`. The `.set_3d_properties()` function call is also made inside the update function.

Animation in 3D requires more careful consideration than in the 2D case. When `matplotlib` displays a 3D plot, it does so in an interactive figure that allows the user to change the camera angle and position. Since 3D rendering is more computationally expensive than 2D rendering, interactive views of 3D animations often have poor framerates and choppy rendering. The solution is to use the tools in the `matplotlib.animation` module to render the animation and then embed that rendered

animation within a Jupyter Notebook. This can be done in two ways: embedding the video directly using an HTML5 API, or encoding the animation to an `.mp4` video and embedding that video. Embedding the video directly tends to be easier, as `matplotlib` does not come with an `.mp4` video encoder, while it can encode video for HTML5.

Saving Animations

Saving the animation by encoding it to a `.mp4` file will allow you to display the video inline inside a Jupyter Notebook, or view it using any video player supporting the chosen filetype.

Unfortunately, Matplotlib does not come with a built-in video encoder. The `matplotlib.animation` module supports several third-party encoders. FFmpeg is a lightweight solution which can be obtained from: https://www.ffmpeg.org/download.html.

To prevent the animation from displaying while it is being rendered as video, use `plt.ioff()`. This turns off matplotlib's interactive mode until `plt.ion()` is called. After creating the animation object, use its `.save()` method with the desired filename to render and save the video. The following code is given for reference:

```
animation.writer = animation.writers['ffmpeg']
plt.ioff()    # Turn off interactive mode to hide rendering animations

# Code to create figure, axes, update function
ani = animation.FuncAnimation(fig,update,frames,interval)
ani.save('my_animation.mp4')
```

To display the `.mp4` video in a Jupyter Notebook, run the following code in a separate markdown cell:

```
<video src="planet_ani.mp4" controls>
```

Problem 3. Each row of the arrays in `orbits.npz` gives the position of the planets at evenly spaced time points. The arrays correspond to 1400 points in time over a 700 day period (beginning on 2018-5-30). Create a 3D animation of the planet orbits. Display lines for the trajectories of the orbits and points for the current positions of the planets at each point in time. Your `update()` function will need to return a list of Line3D objects, one for each orbit trajectory and one for each planet position marker. Embed your animated plot.

Surface Plotting

3D surface plotting is very similar to regular 3D plotting discussed earlier. The difference with surface plots is that they require first creating a *meshgrid* for X and Y. Meshgrids are created using the NumPy command `np.meshgrid(x, y)` where `x` and `y` are 1D arrays representing the x and y coordinates of the grid. This function creates 2D arrays `X` and `Y` that combined give cartesian cordinates for every point made from the `x` and `y` arrays.

Once a meshgrid is defined, a surface plot is generated by calling `ax.plot_surface(X, Y, Z)`, where `Z` is a 2D array of height values that is the same shape as `X` and `Y`.

Problem 4. Make a surface plot of the bivariate normal density function given by:

$$f(\mathbf{x}) = \frac{1}{\sqrt{det(2\pi\Sigma)}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right]$$

where $\mathbf{x} = [x, y]^T$, $\mu = [0, 0]^T$ is the mean vector, and

$$\Sigma = \begin{bmatrix} 1 & 3/5 \\ 3/5 & 2 \end{bmatrix}$$

is the covariance matrix. Compare your results with Figure 1.2.

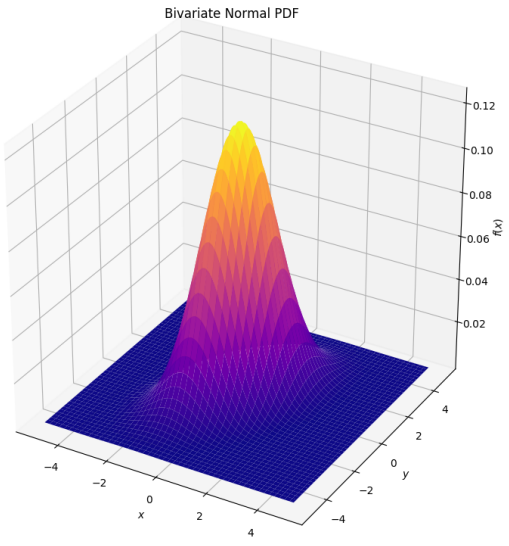


Figure 1.2: The solution to Problem 4.

Surface Animations

Animating a 3D surface is slightly different from animating a parametric curve in 3D. The object created by `.plot_surface()` does not have a `.set_data()` method. Instead, use `ax.clear()` to empty the axes at each frame, followed by a new call to `ax.plot_surface()`. Note that the axes limits must be reset after `ax.clear()` is called.

Problem 5. Use the data in `vibration.npz` to produce a surface animation of the solution to the wave equation for an elastic rectangular membrane. The file contains three NumPy arrays:

`X`, `Y`, `Z`. `X` and `Y` are meshgrids of shape `(300,200)` corresponding to 300 points in the y-direction and 200 points in the x-direction, giving a 2x3 rectangle with one corner at the origin. `Z` is of shape `(150,300,200)`, giving the height of the vibrating membrane at each (x,y) point for 150 values of time. In the language of partial differential equations, this is the solution to the following initial/boundary value problem:

$$u_{tt} = 6^2(u_{xx} + u_{yy})$$
$$(x, y) \in [0, 2] \times [0, 3], t \in [0, 5]$$
$$u(t, 0, y) = u(t, 2, y) = u(t, x, 0) = u(t, x, 3) = 0$$
$$u(0, x, y) = xy(2 - x)(3 - y)$$

1 Intro to IVP and BVP

Initial Value Problems

An initial value problem is a differential equation with a set of constraints at the initial point. An IVP may look something like this

$$\begin{aligned} y'' + y' + y &= f(t) \\ y(a) &= \alpha \\ y'(a) &= \beta \\ t &\in [a, b]. \end{aligned}$$

This problem gives a differential equation with initial conditions for y and y'.

Formulating and solving initial value problems is an important tool when solving many types of problems. One simple example of an IVP would be a differential equation modeling the path of a ball thrown in the air where the initial position ($y(a)$) and velocity ($y'(a)$) are known. These problems can be tricky to solve by hand. Luckily, SciPy has great tools that help us solve initial value problems for most systems of first order ODEs. We will be using `solve_ivp` from `scipy.integrate`.

Consider the following example

$$y'' + 3y = sin(t), \quad y(0) = -\pi/2, \quad y'(0) = \pi, \quad t \in [0, 5]$$

We begin by changing this second order ODE into a first order ODE system.

$$\text{Let } y_1 = y \text{ and } y_2 = y' \text{ so that,}$$
$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ \sin t - 3y_1 \end{bmatrix}.$$

This formulation allows us to use `solve_ivp`. We need three code elements in order to implement `solve_ivp`:

1. the ODE function:
   This function defines the ODE system by returning an array containing our first order system of ODES.

2. the time domain:
   This is a tuple giving the interval of integration.

3. the initial conditions:
   This is an array containing the initial conditions starting with the "zeroeth" derivative constraint, followed by the first derivative constraint, and so on if there are other constraints of higher order derivatives.

The following code is for the example given above.

```
from scipy.integrate import solve_ivp
import numpy as np

# element 1: the ODE function
def ode(t, y):
    '''defines the ode system'''
    return np.array([y[1],np.sin(t)-3*y[0]])

# element 2: the time domain
t_span = (0,5)

# element 3: the initial conditions
y0 = np.array([-np.pi /2, np.pi])

# solve the system
# max_step is an optional parameter that controls maximum step size and
# a smaller value will result in a smoother graph
sol = solve_iv
```

Then we can plot the solution with the following code

```
from matplotlib import pyplot as plt

plt.plot(sol.t,sol.y[0])
plt.xlabel('t')
plt.ylabel('y(t)')
plt.show()
```

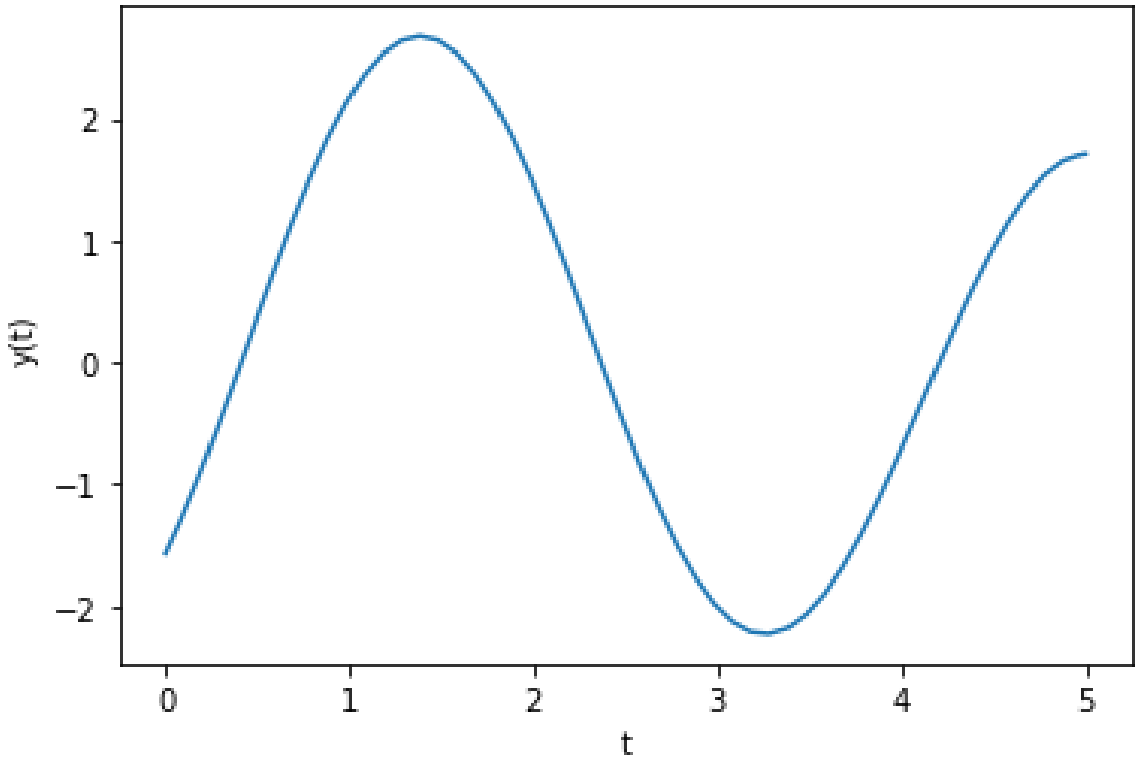


Figure 1.1: The solution to the above example

Problem 1. Use `solve_ivp` to solve for y in the equation $y'' - y = sin(t)$ with initial conditions $y(0) = -\frac{1}{2}$, $y'(0) = 0$ and plot your solution on the interval $[0, 5]$. Compare this to the analytic solution $y = -\frac{1}{2}(e^{-t} + sin(t))$.

Note: Using `max_step` $= 0.1$ with give you the smoother graph seen in figure 1.2.

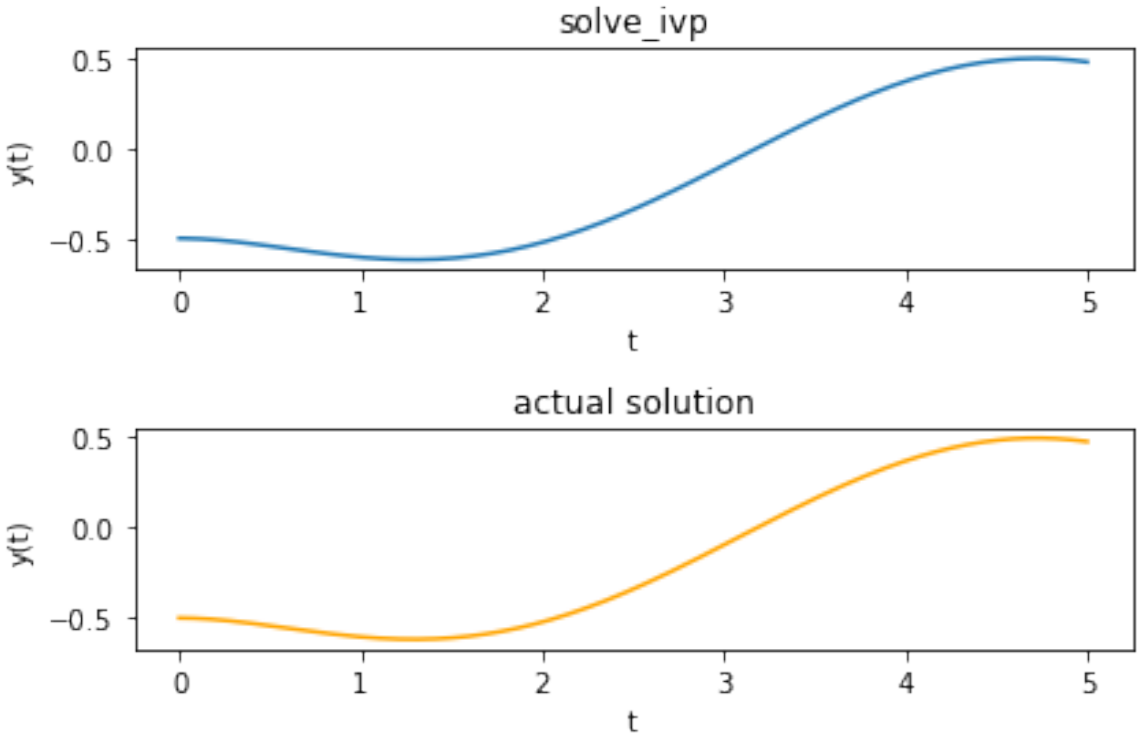


Figure 1.2: The solution to problem 1

Boundary Value Problems

A boundary value problem is a differential equation with a set of constraints. It is similar to initial value problems, but may give end constraints as well as initial constraints. A boundary value problem may look something like this

$$\begin{aligned} y'' + y' + y &= f(t) \\ y(a) &= \alpha \\ y(b) &= \beta \\ t &\in [a, b], \end{aligned}$$

where we have both right and left hand boundary conditions on y. One simple example of an IVP would be a differential equation modeling the path of a ball thrown in the air where the initial position ($y(a)$) and final position ($y(b)$).

SciPy has great tools that help us solve boundary value problems. We will be using `solve_bvp` from `scipy.integrate`. Consider the following example:

$$y'' + 9y = cos(t), \quad y'(0) = 5, \quad y(\pi) = -\frac{5}{3}. \tag{1.1}$$

We begin by changing this second order ODE into a first order ODE system.

$$\text{Let } y_1 = y \text{ and } y_2 = y' \text{ so that,}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ \cos t - 9y_1 \end{bmatrix}.$$

This formulation allows us to use `solve_bvp`. It is important to notice that there are several key differences between `solve_ivp` and `solve_bvp`. We need four code elements in order to implement `solve_bvp`:

1. the ODE function:
   This is the same function we used in `solve_ivp`.

2. the boundary condition function:
   Instead of just having a tuple containing our initial values, we now must use a function that returns an array of the residuals of the boundary conditions. We pass in 2 arrays: `ya`, representing the initial values, and `yb`, representing the final values. The ith entry of those arrays represents the boundary condition at the ith derivative. `ya[0]-x` would indicate that we know `y(a)=x`, `ya[1]-x` would indicate that we know `y`(a)=x`, `yb[0]-x` would indicate that we know `y(b)=x`, or `yb[1]-x` would indicate that we know `y`(b)=x`,

3. the time domain:
   Instead of a tuple giving the interval of integration, we now must pass in a linspace from the starting time to the ending time, containing the desired number of points (we now must choose the number).

4. the initial guess:
   As we no longer know all of the initial values, we now must make an educated guess. This is an array of shape (`n`,`t_steps`) where `n` is the shape of the output of the ODE function and `t_steps` is the chosen number of steps in our time domain linspace.

```
from scipy.integrate import solve_bvp
import numpy as np

# element 1: the ODE function
def ode(t,y):
    ''' define the ode system '''
    return np.array([y[1], np.cos(t) - 9*y[0]])
# element 2: the boundary condition function
def bc(ya,yb):
    ''' define the boundary conditions '''
    # ya are the initial values
    # yb are the final values
    # each entry of the return array will be set to zero
    return np.array([ya[1] - 5, yb[0] + 5/3])

# element 3: the time domain.
t_steps = 100
t = np.linspace(0,np.pi,t_steps)

# element 4: the initial guess.
y0 = np.ones((2,t_steps))

# Solve the system.
sol = solve_bvp(ode, bc, t, y0)
```

The syntax for plotting the function is also slightly different:

```
import matplotlib.pyplot as plt

# here we plot sol.x instead of sol.t
plt.plot(sol.x, sol.y[0])
plt.xlabel('t')
plt.ylabel('y(t)')
plt.show()
```

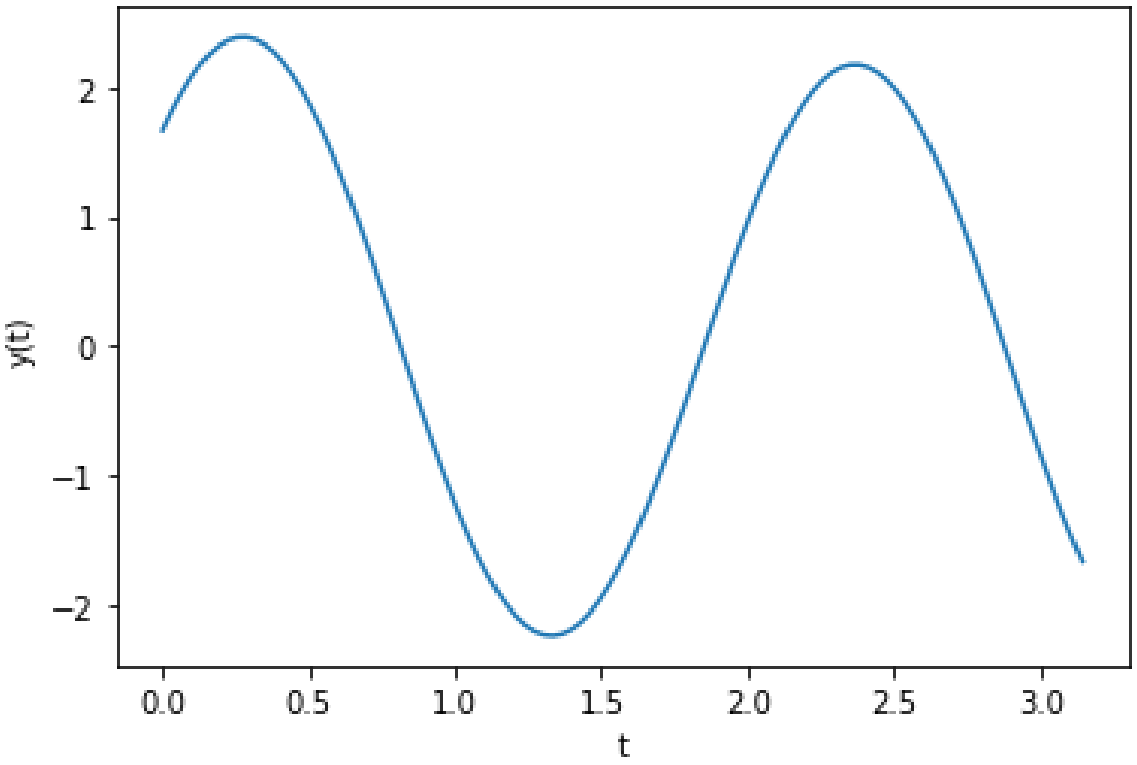


Figure 1.3: The solution to the above problem

Problem 2. Use `solve_bvp` to solve for y in the equation $y'' - y = sin(t)$ with initial conditions $y(0) = -\frac{1}{2}$, $y'(0) = 0$ and plot your solution on the interval $[0, 5]$. Compare this to the analytic solution $y = -\frac{1}{2}(e^{-t} + sin(t))$.

Note: Using `max_step` $= 0.1$ with give you the smoother graph seen in figure 1.4.

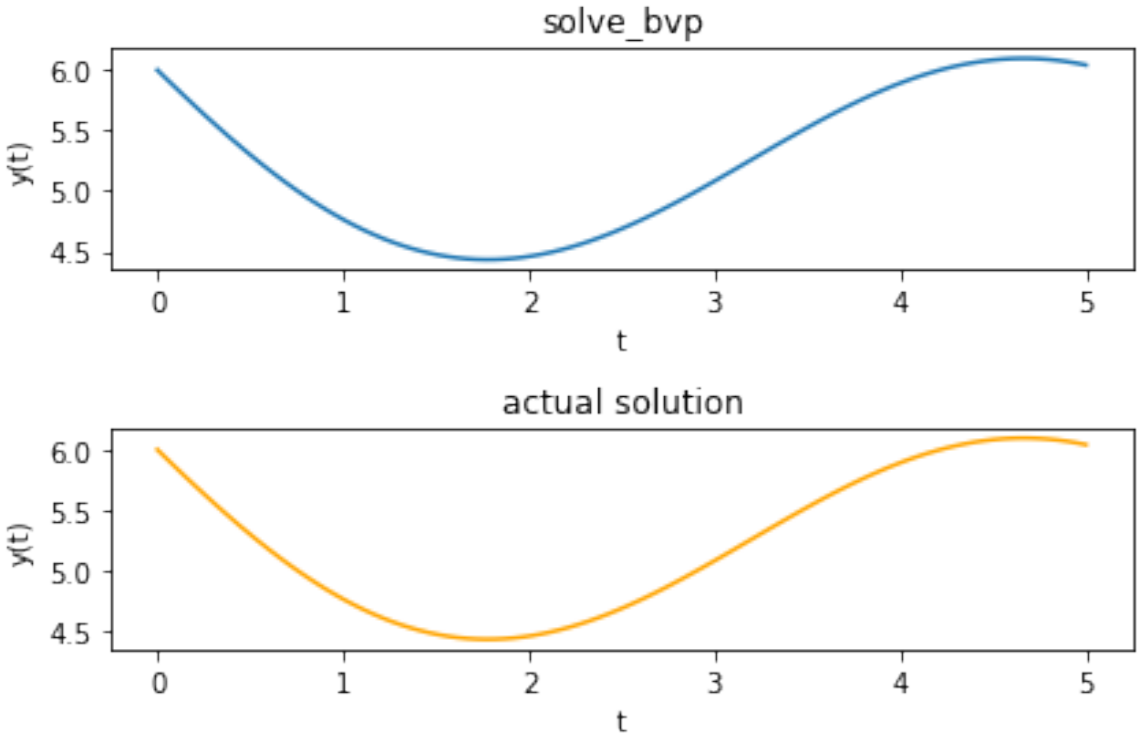


Figure 1.4: The solution to problem 2.

One other useful function of `solve_bvp`: `sol.sol` creates a callable function, which is the estimation of the boundary value problem. You can utilize it by plugging in any int or linspace (`sol.sol(np.linspace)`, `sol.sol(int)`, `sol.sol(list)`), like a normal lambda function.

The Pitfalls of `solve_bvp`

One of the common issues with `solve_bvp` is choosing a guess for the initial value. Often, small changes in the guess can cause large changes in the final approximation. The next problem demonstrates the huge difference that can be made between an initial guess of 10 and an initial guess of 9.99

Problem 3. Use `solve_bvp` to solve for y in the equation $y'' = (1 - y') * 10y$ with boundary conditions $y(0) = -1$ and $y(1) = \frac{3}{2}$ and plot your solution on the interval $[0, 1]$. Use an initial guess of 10. Compare this to the the same solution using an initial guess of 9.99.

The solution is found in figure 1.5.

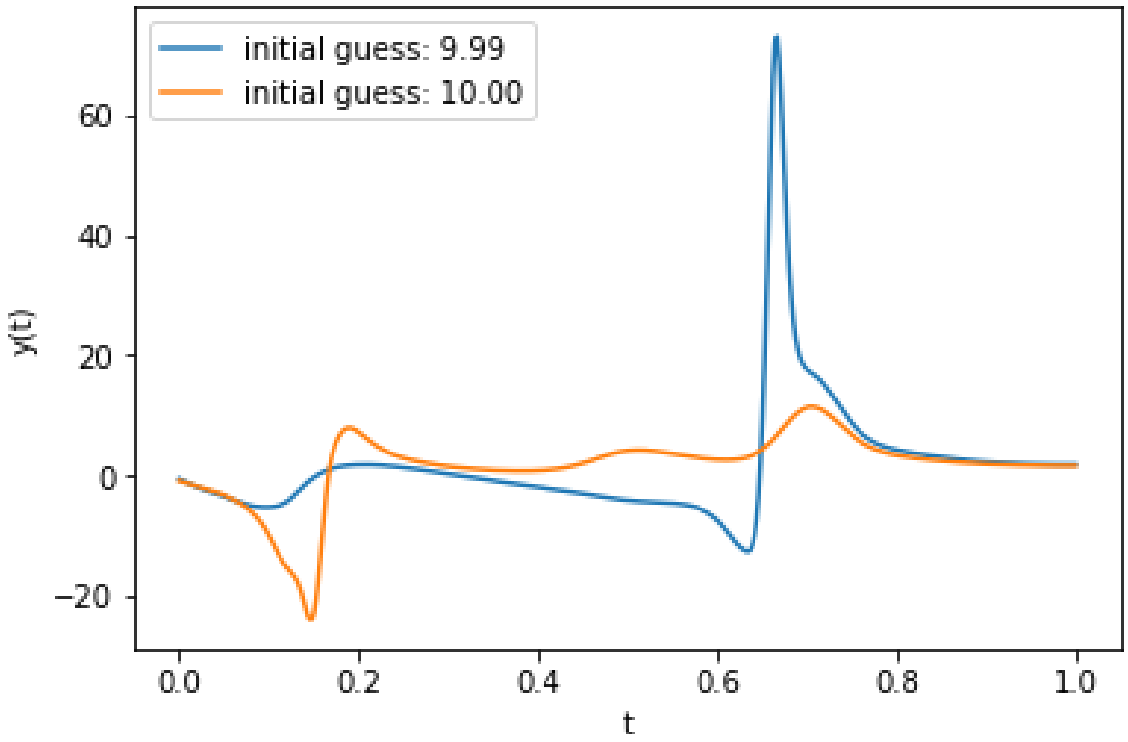


Figure 1.5: The solution to problem 3.

`solve_ivp` and Strange Attractors

In the growing field of dynamical systems, an attractor is a set of states toward which a system tends to evolve. Strange Attractors are special in that they showcase complex behavior in a simple set of equations. A minute change in the initial values can cause massive discrepancies in the outcome. The most famous of these is the Lorenz Attractor, introduced by Edward Lorenz in 1963. Later on, we dedicate a full lab to the study of the Lorenz Attractor, but today we focus on modeling the Four-Wing attractor. This is a system of first order ODEs defined by the following set of equations

$$\frac{dx}{dt} = ax + yz \tag{1.2}$$
$$\frac{dy}{dt} = bx + cy - xz \tag{1.3}$$
$$\frac{dz}{dt} = -z - xy \tag{1.4}$$

given some constants a, b, and c. As we mentioned earlier, `solve_ivp` and `solve_bvp` can be used to solve and model systems of first order ODEs. We will now use `solve_ivp` to model a Four-Wing attractor.

Problem 4. Use `solve_ivp` to solve the Four-Wing Attractor as described in equations (1.2), (1.3), and (1.4) where $a = 0.2$, $b = 0.01$, and $c = -0.4$. Try this with 3 different initial values

and plot (in three dimensions) the 3 corresponding graphs.

Possible solutions are seen in Figure 1.6.

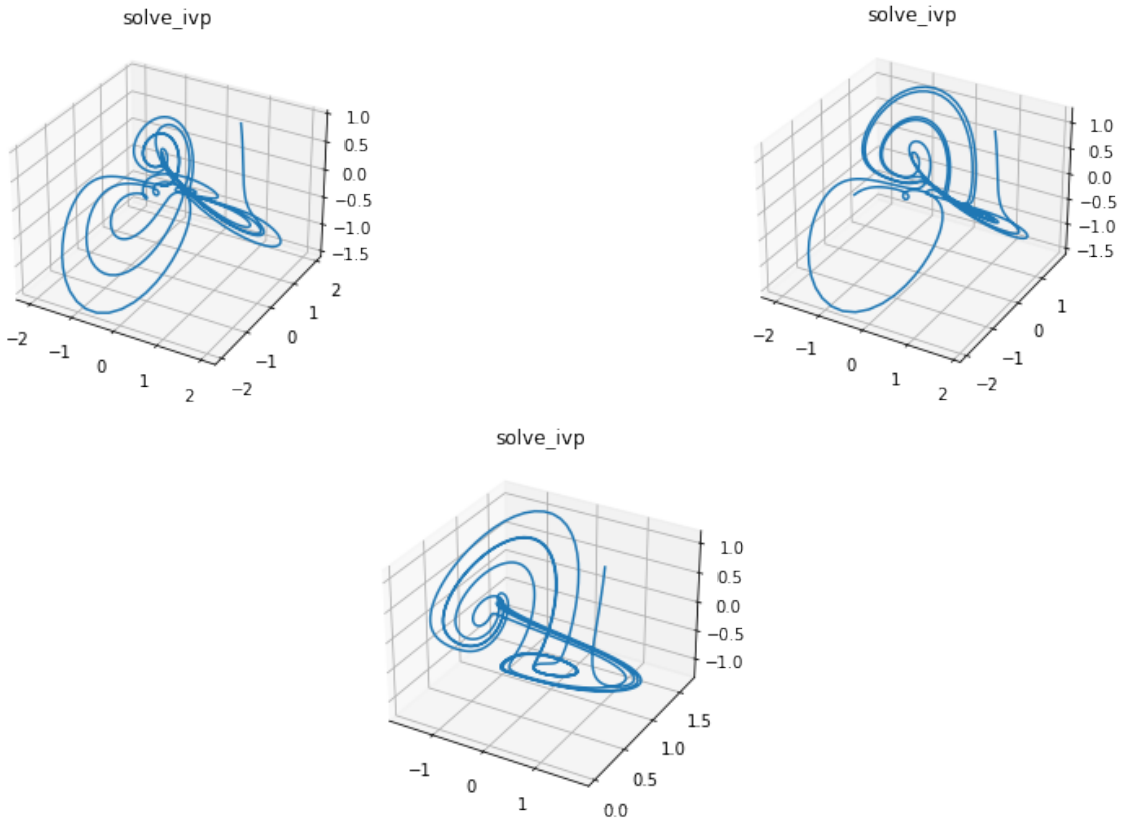


Figure 1.6: Possible solutions to problem 4.

1 Modelling the spread of an epidemic: SIR models

Numerical Solvers

We often rely on numerical solvers to numerically integrate ordinary differential equations, ODEs. Because of the complexity of many ODE systems, these numerical solvers allow us to solve complex ODE systems that may not be solvable symbolically, or are high dimensional. In this lab we will be using `solve_ivp`, which is a part of `scipy.integrate`, to solve ODE systems related to epidemic models. You can read the documentation for `solve_ivp` at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`.

`solve_ivp` takes the ODE as a function, a tuple containing the start and end time, and an array with the initial conditions as arguments, and returns a bunch object containing the solution and other information. We can solve the following ODE system with the following code.

$$\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}' = \begin{bmatrix} y_2(t) \\ \sin(t) - 5y_2(t) - y_1(t) \end{bmatrix} \qquad (1.1)$$
$$y_1(0) = 0, \quad y_2(0) = 1, \quad t \in [0, 3\pi]$$

```
import numpy as np
from scipy.integrate import solve_ivp

# define the ode system as given in the problem
def ode(t,y):
    return np.array([y[1], np.sin(t) - 5*y[1] - y[0]])

# define the t0 and tf parameters
t0 = 0
tf = 3*np.pi

# define the initial conditions
y0 = np.array([0,1])

# solve the system
sol = solve_ivp(ode, (t0,tf), y0)

# Plot the system
```

```
import matplotlib.pyplot as plt

# plot y_1 against y_2
plt.plot(sol.y[0],sol.y[1])
plt.xlabel('$y_1$')
plt.ylabel('$y_2$')
plt.show()
```

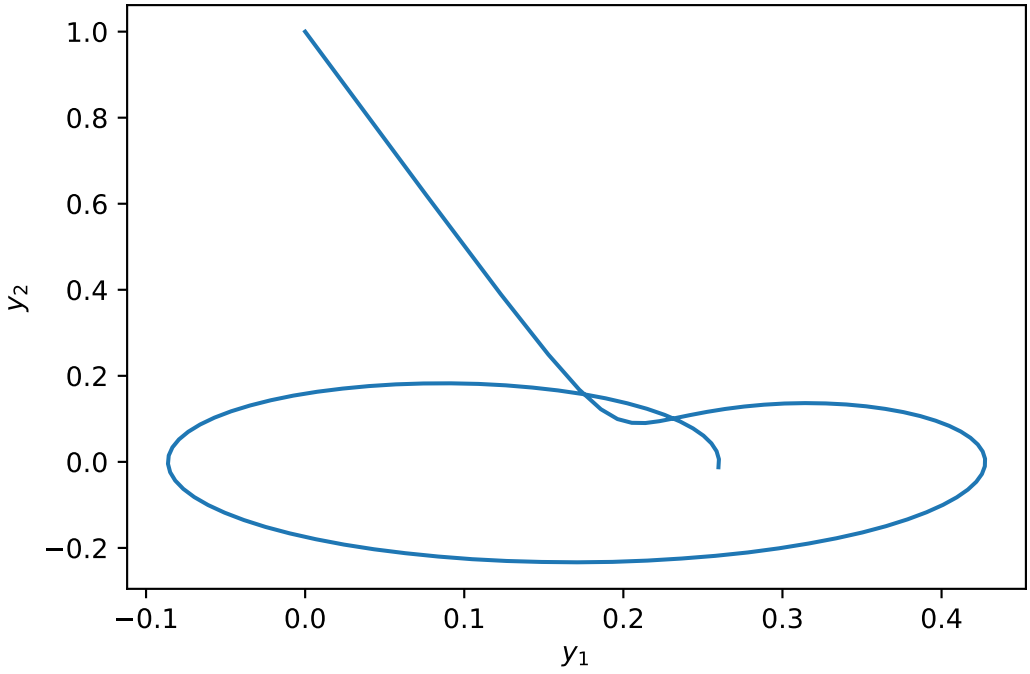


Figure 1.1: Solution to (1.1)

The SIR Model

The SIR model describes the spread of an epidemic through a large population. It does this by describing the movement of the population through three phases of the disease: those individuals who are *susceptible*, those who are *infectious*, and those who have been *removed* from the disease. Those individuals in the removed class have either died, or have recovered from the disease and are now immune to it. If the outbreak occurs over a short period of time, we may reasonably assume that the total population is fixed, so that $S'(t) + I'(t) + R'(t) = 0$. We may also assume that $S(t) + I(t) + R(t) = 1$, so that $S(t)$ represents the *fraction* of the population that is susceptible, etc.

Individuals may move from one class to another as described by the flow

$$S \to I \to R.$$

Let us consider the transition rate between S and I. Let β represent the average number of contacts made per unit time period (one day perhaps) that could spread the disease. The proportion of these contacts that are with a susceptible individual is $S(t)$. Thus, one infectious individual will on average infect $\beta S(t)$ others per day. Let N represent the total population size. Then we obtain the differential equation

$$\frac{d}{dt}(S(t)N) = -\beta S(t)(I(t)N)$$

Now consider the transition rate between I and R. We assume that there is a fixed proportion γ of the infectious group who will recover on a given day, so that

$$\frac{d}{dt}R(t) = \gamma I(t).$$

Note that γ is the reciprocal of the average length of time spent in the infectious phase.

Since the derivatives sum to 0, we have $I'(t) = -S'(t) - R'(t)$, so the differential equations are given by

$$\frac{dS}{dt} = -\beta IS, \tag{1.2}$$
$$\frac{dI}{dt} = \beta IS - \gamma I, \tag{1.3}$$
$$\frac{dR}{dt} = \gamma I. \tag{1.4}$$

Problem 1. Suppose that, in a city of approximately three million, five people who have just become infectious have recently entered the city carrying a certain disease. Each of those individuals has one contact each day that could spread the disease, and an average of three days is spent in the infectious state. Find the solution of the corresponding SIR equations using solve_ivp for fifty days, where each time period is half a day, and plot your results. Use the percentages of each state, not the actual number of people in the state.

At the peak of the infection, how many in the city will still be able to work (assume for simplicity that those who are in the infectious state either cannot go to work or are unproductive, etc.)?

Hint: Use the t-values paramter in solve_ivp to pass in an array of t-values.

Compare your plot to Figure 1.

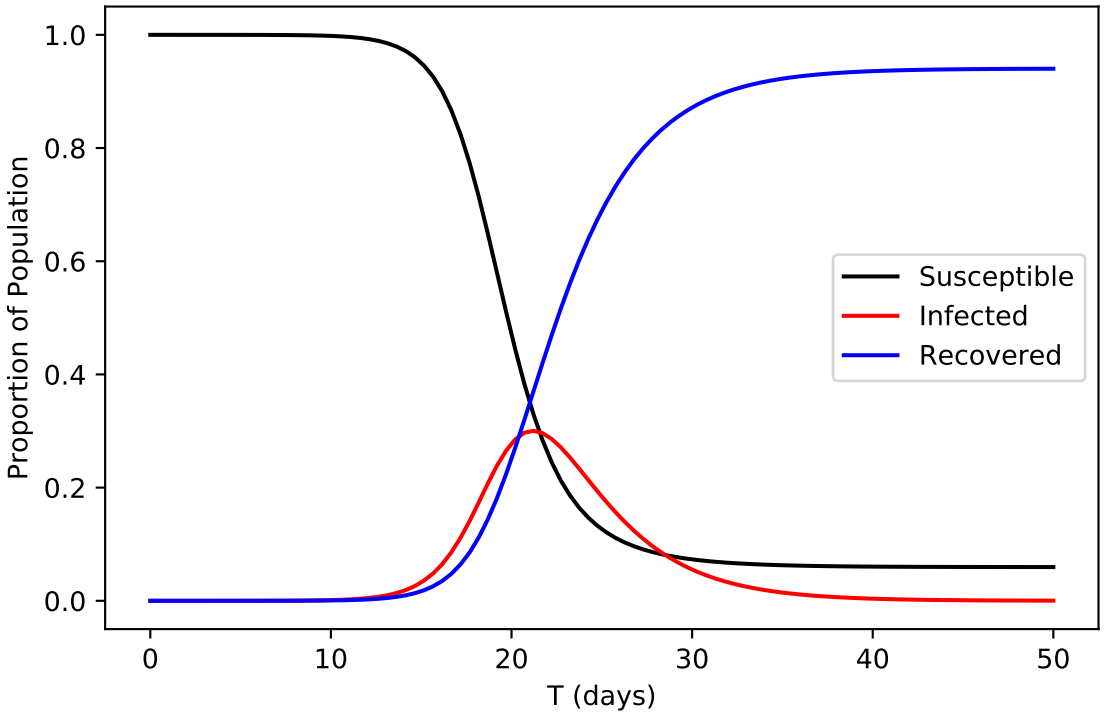


Figure 1.2: Solution to Problem (1)

SIR is an effective model for epidemic spread under certain assumptions. For example, we assume that the network is what's called "fully mixed." This implies that no group of members of a network are more likely to encounter each other than any other group. Because of this assumption, we should not use SIR to model networks we know to be poorly mixed. In fact, we should be clear in stating that almost no network is truly fully mixed; however this model is still effective for networks that are reasonably well mixed. In the next problem we will be using SIR to model data from the recent COVID-19 outbreak. To adhere to the "reasonably well mixed" criteria, we will be using only data from one county at a time.

Problem 2. On March 11, 2020, New York City had 52 confirmed cases of COVID-19. On that day New York started its lock-down measures. Using the following information, model what the spread of the virus could have been, using `solve_ivp()`, if New York did not implement any measures to curb the spread of the virus over the next 150 days:

- There are approximately 8.399 million people in New York city.
- The average case of COVID-19 lasts for 10 days.
- Each infected person can spread the virus to 2.5 people.

Plot your results for each day and compare to Figure 1.3.

1. At the projected peak, how many concurrent active cases are there?
2. Assuming that about 5% of COVID-19 cases require hospitalization, and using the fact that there are about 58,000 hospital beds in NYC, how many beds over capacity will the hospitals in NYC be at the projected peak?

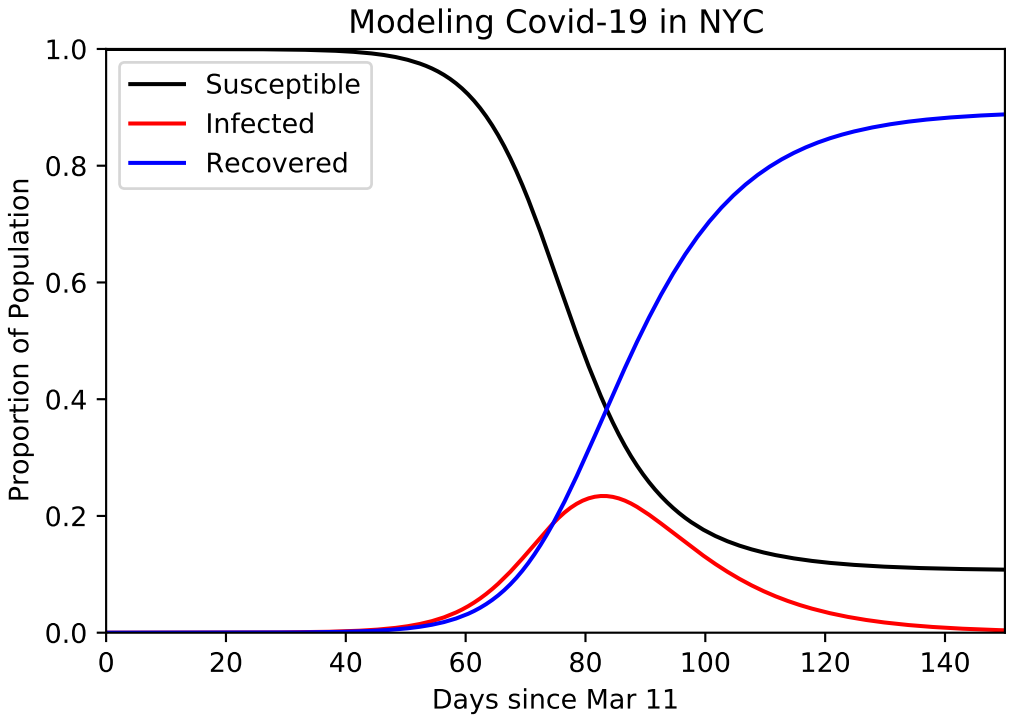


Figure 1.3: Solution to Problem (2).

Variations on the SIR Model

The SIS model is a common variation of the SIR model. SIS Models describe diseases where individuals who have recovered from the disease do not gain any lasting immunity. There are only two compartments in this model: those who are *susceptible*, and those who are *infectious*. Here, f is the rate of becoming susceptible again.

The basic equations are given by

$$\frac{dS}{dt} = -\beta IS + fI,$$
$$\frac{dI}{dt} = \beta IS - fI$$

Another alteration we can make to the SIR model is to add a birth and death rate. In the equations below we are assuming that the natural death rate together with the death rate caused by

the disease is equal to the birth rate. This model is given by

$$\begin{aligned}\frac{dS}{dt} &= \mu(1-S) - \beta IS,\\ \frac{dI}{dt} &= \beta IS - (\gamma+\mu)I,\\ \frac{dR}{dt} &= \gamma I - \mu R\end{aligned}$$

where μ represents the death rate and equal birth rate, noting that any new person born is born into the susceptible population.

If we combine the last two variations we made on the SIR model we come to this formulation, which is an SIRS model. This SIRS model allows the transfer of individuals from the recovered/removed class to the susceptible class and includes modeling of the birth and death rates.

$$\frac{dS}{dt} = fR + \mu(1-S) - \beta IS, \tag{1.5}$$
$$\frac{dI}{dt} = \beta IS - (\gamma+\mu)I, \tag{1.6}$$
$$\frac{dR}{dt} = -fR + \gamma I - \mu R. \tag{1.7}$$

Problem 3. There are 7 billion people in the world. Influenza, or the flu, is one of those viruses that everyone can be susceptible to, even after recovering. The flu virus is able to change in order to evade our immune system, and we become susceptible once more, although technically it is now a different strain. Suppose the virus originates with 1000 people in Texas after Hurricane Harvey flooded Houston, and stagnant water allowed the virus to proliferate. According to WebMD, once you get the virus, adults are contagious up to a week and kids up to 2 weeks. For this lab, suppose you are contagious for 10 days before recovering. Also suppose that on average someone makes one contact every two days that could spread the flu. Since we can catch a new strain of the flu, suppose that a recovered individual becomes susceptible again with probability $f = 1/50$. The flu is also known to be deadly, killing hundreds of thousands every year on top of the normal death rate. To assure a steady population, let the birth rate balance out the death rate, and in particular let $\mu = .0001$.

Using the SIRS model above, plot the proportion of population that is Susceptible, Infected, and Recovered over a one-year span (365 days). Compare your plot to Figure 1.4.

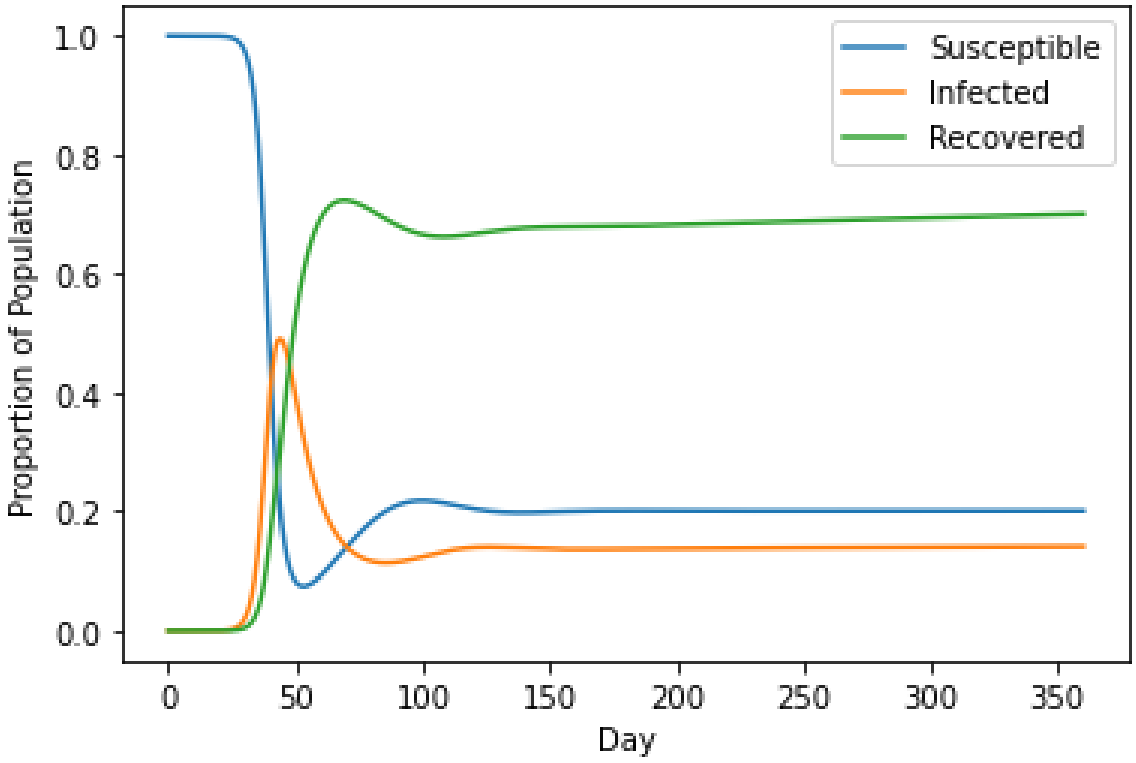


Figure 1.4: Solution to Problem (3).

Modeling COVID-19 with Social Distancing

Social distancing upsets the main assumption that is made when trying to model epidemic spread using SIR models. During the periods of lockdown instituted by governments, the interaction networks between people in a city or county were disrupted to the point that standard SIR models were no longer effective at modeling the spread of COVID-19. A paper released in May of 2020 presented some alternative models for COVID-19 that have some success in modeling its spread during periods of social distancing.

This model claims that the growth of $I(t)$ is polynomial with exponential decay (PGED). So we get the following form

$$I(t) \approx Bt^{\alpha}e^{-t/T_G},$$

which results in the following SIR type model

$$\frac{dS}{dt} = -\frac{\alpha}{t}I, \tag{1.8}$$

$$\frac{dI}{dt} = \left(\frac{\alpha}{t} - \frac{1}{T_G}\right)I, \tag{1.9}$$

$$\frac{dR}{dt} = \frac{1}{T_G}I, \tag{1.10}$$

where α and T_G are simply model parameters. In this model αT_G can be interpreted as the time of epidemic peak.

Fitting Models

Model fitting can be a frustrating task if we only use our intuition and guess and check. Thankfully, SciPy's `optimize` library has tools we can use to make these problems a lot easier. Many of the functions in this library are designed to take an arbitrary function and find whatever input makes the output close to zero. Our job is to create a function that outputs zero at the right values.

Suppose we have some data that we believe to follow a cubic trend with the following model

$$\alpha x^3 + \beta(x^2 + 2x) + \delta.$$

In order to fit the data to this model we can use `scipy.optimize.minimize` and create a function that will output zero when the correct parameters are input. `scipy.optimize.minimize` will then return an `OptimizeResult` object, which contains the optimal paramters.

```
# import the minimizer function
from scipy.optimize import minimize

# load the data and get the x and y values
data = np.load('to_fit.npy')
xs = data[:,0]
ys = data[:,1]

# define the function we want to minimize
def fun(params):
    # unpack the parameters
    a,b,d = params

    # get the model output based on the parameters
    out = a*xs**3 + b*(xs**2 + 2*xs) + d

    # find the difference between out and the data
    diff = out - ys

    # must return a float
    return np.linalg.norm(diff)

# make a guess for the parameters
p0 = (1,1,1)

# find the best parameters for this model
minimize(fun,p0)
```

Problem 4. Fit the PGED model to the COVID-19 data provided in `new_york_cases.npy`. Plot your results against $1 - S(t)$.

Hint: Set $t_0 = 1$ as the PEGD model requires to divide by t, so we must have $t \neq 0$.

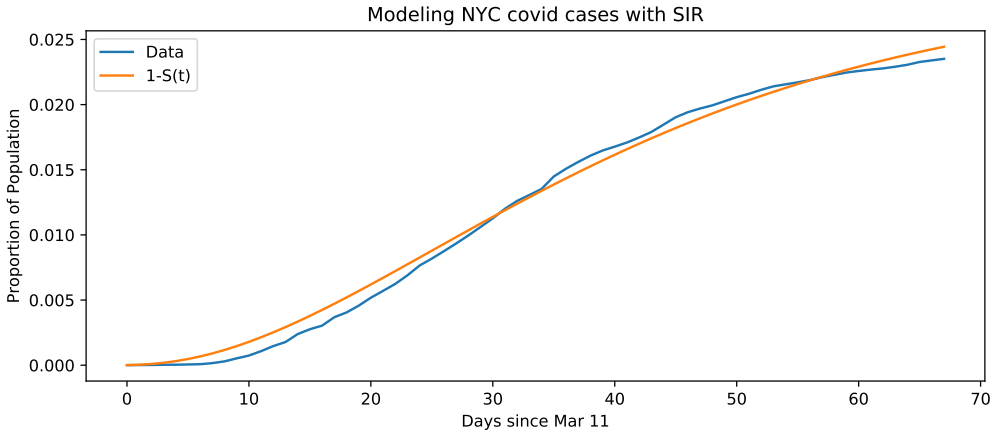


Figure 1.5: Solution to (4)

Boundary Value Problems

The next exercise uses a variation of the SIR model called an SEIR model to describe the spread of measles[1]. This new model adds another compartment, called the *exposed* or *latency* phase. It assumes that the rate at which measles is contracted depends on the season, i.e. the rate is periodic. That allows us to formulate the yearly occurrence rate for measles as a boundary value problem. The boundary value problem looks like

$$\begin{bmatrix} S \\ E \\ I \end{bmatrix}' = \begin{bmatrix} \mu - \beta(t)SI \\ \beta(t)SI - E/\lambda \\ E/\lambda - I/\eta \end{bmatrix}, \tag{1.11}$$

$$\begin{aligned} S(0) &= S(1), \\ E(0) &= E(1), \\ I(0) &= I(1) \end{aligned} \tag{1.12}$$

Parameters μ and λ represent the birth rate of the population and the latency period of measles, respectively. η represents the infectious period before an individual moves from the infectious class to the recovered class. After recovery an individual remains immune, which is why $R(t)$ is not included in the system. The set up of this problem is not normal since we are excluding $R(t)$, but it results in a nice graph.

To solve this problem we will use a full-featured BVP solver that is available in `SciPy`. The code below demonstrates how to use `solve_bvp` to solve the BVP

$$\varepsilon y'' + yy' - y = 0, \quad y(-1) = 1, \quad y(1) = -1/3, \quad \varepsilon = .1 \tag{1.13}$$

Look at figure 1.6 for the solution.

[1]Numerical Solution of Boundary Value Problems for Ordinary Differential Equations, by Aescher, Mattheij, and Russell

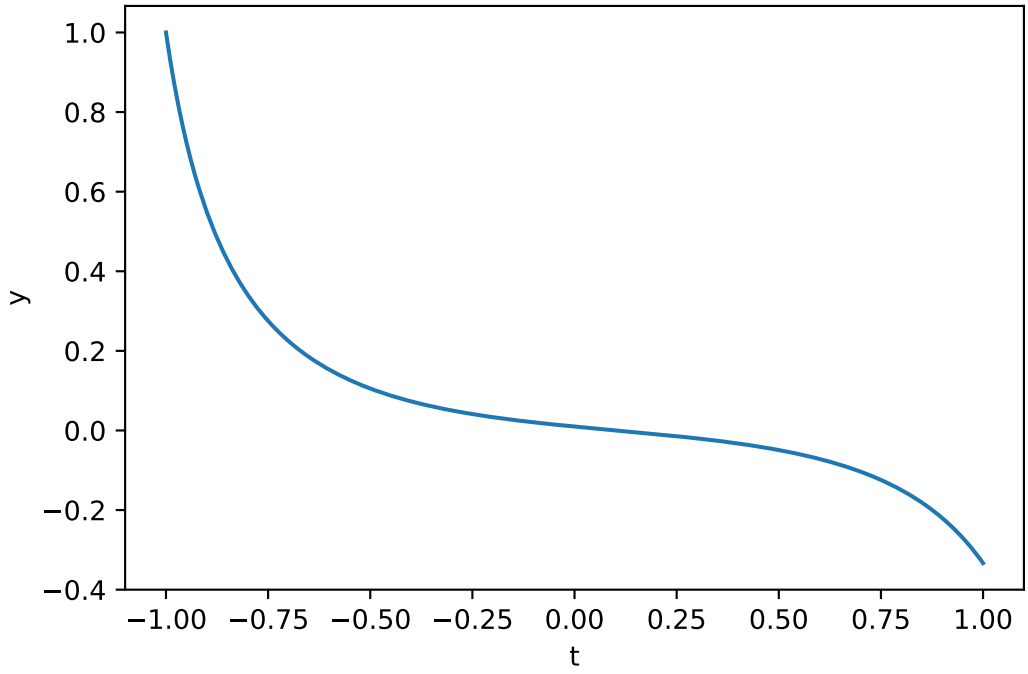


Figure 1.6: Solution to Equation (1.13)

The BVP solver expects you to pass it the boundary conditions as a callable function that computes the difference between a guess at the boundary conditions and the desired boundary conditions. When we use the BVP solver, we will tell it how many constraints there should be on each side of the domain so it knows how many entries to expect. In this case, we have one boundary condition on either side. These constraints are expected to evaluate to 0 when the boundary condition is satisfied.

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

epsilon, lbc, rbc = .1, 1, - 1/3

# The ode function takes the independent variable first
# It has return shape (n,)
def ode(x , y):
    return np.array([y[1] , (1/epsilon) * (y[0] - y[0] * y[1])])

# The return shape of bcs() is (n,)
def bcs(ya, yb):
    BCa = np.array([ya[0] - lbc])   # 1 Boundary condition on the left
    BCb = np.array([yb[0] - rbc])   # 1 Boundary condition on the right
    # The return values will be 0s when the boundary conditions are met exactly
    return np.hstack([BCa, BCb])
```

```
# The independent variable has size (m,) and goes from a to b with some step ↩
    size
X = np.linspace(-1, 1, 200)
# The y input must have shape (n,m) and includes our initial guess for the ↩
    boundaries
y = np.array([-1/3, -4/3]).reshape((-1,1))*np.ones((2, len(X)))

# There are multiple returns from solve_bvp(). We are interested in the y ↩
    values which can be found in the sol field.
solution = solve_bvp(ode, bcs, X, y)
# We are interested in only y, not y', which is found in the first row of sol.
y_plot = solution.sol(X)[0]

plt.plot(X, y_plot)
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```

Problem 5. Consider equations (1.11) and (1.12). Let the periodic function for our measles case be $\beta(t) = \beta_0(1 + \beta_1 \cos 2\pi t)$. Use parameters $\beta_1 = 1$, $\beta_0 = 1575$, $\eta = 0.01$, $\lambda = .0279$, and $\mu = .02$. Note: in this case, time is measured in years, so run the solution over the interval $[0, 1]$ to show a one-year cycle. The boundary conditions in (1.12) are just saying that the year will begin and end in the same state.

One issue that we encounter with this problem is that we have 6 boundary conditions but we only have 3 free variables. The 6 boundary counditions are the initial and final conditions of S, E, and I. `solve_bvp` only allows as many boundary conditions as there are free variables, so what we can do is include "dummy" variables in the ODE. This allows more boundary conditions in the BVP solver, while not changing the ODE system that we are solving. To deal with this, let $C(t) = [C_1(t), C_2(t), C_3(t)]$, and add the equation

$$C'(t) = 0$$

to the system of ODEs given above (for a total of 6 equations) resulting in this final 6 variable system

$$\begin{bmatrix} S(t) \\ E(t) \\ I(t) \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}' = \begin{bmatrix} \mu - \beta(t)SI \\ \beta(t)SI - E/\lambda \\ E/\lambda - I/\eta \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

We can then apply all 6 of the boundary conditions that we need. The boundary conditions

can be separated using the following trick:

$$\begin{pmatrix} C_1(0) \\ C_2(0) \\ C_3(0) \end{pmatrix} = \begin{pmatrix} S(0) \\ E(0) \\ I(0) \end{pmatrix}, \quad \begin{pmatrix} C_1(1) \\ C_2(1) \\ C_3(1) \end{pmatrix} = \begin{pmatrix} S(1) \\ E(1) \\ I(1) \end{pmatrix}.$$

Now C_1, C_2, C_3 become the 4th, 5th, and 6th rows of your solution matrix, so the 3 boundary conditions for the left are obtained by subtracting the last three entries of $y(0)$ from the first three entries, giving you $ya[0:3] - ya[3:]$. Similarly, your right boundary conditions will look like $yb[0:3] - yb[3:]$.

When you code your boundary conditions, note that `solve_bvp` changes the initial conditions to force all the entries in the return of `bcs()` to be zero. You can use the initial conditions from Fig. 1.7 as your initial guess (which will be an array of 6 elements). Remember that the initial infected proportion is small, not 0.

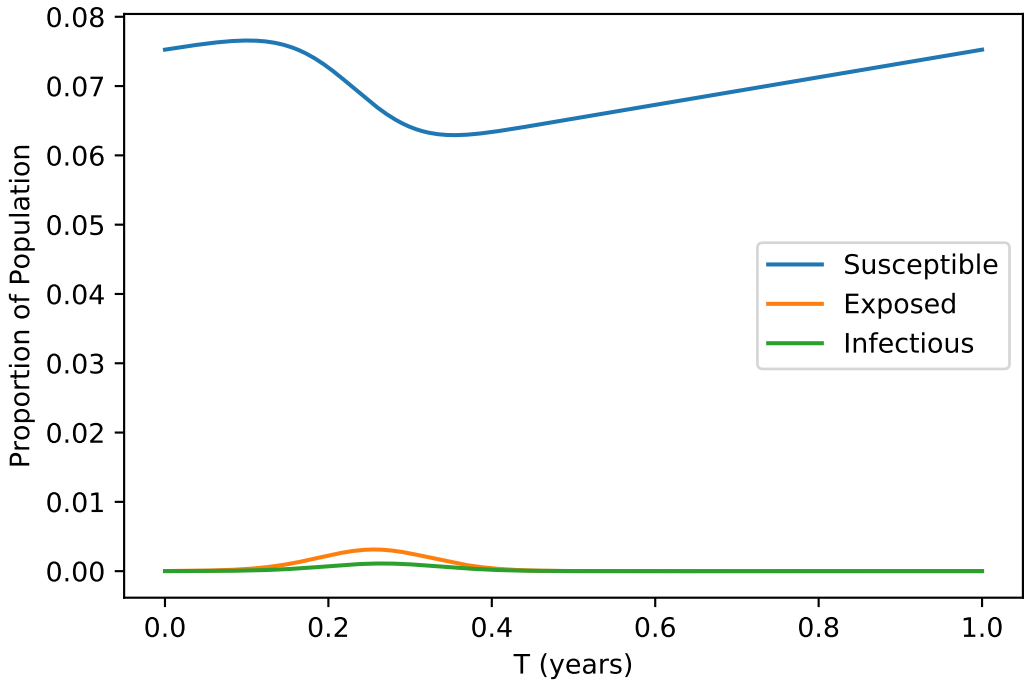


Figure 1.7: Solution to Problem (5)

1 Numerical Methods for Initial Value Problems; Harmonic Oscillators

Lab Objective: *Implement several basic numerical methods for initial value problems (IVPs) and use them to study harmonic oscillators.*

Methods for Initial Value Problems

Consider the *initial value problem* (IVP)

$$\begin{aligned} \mathbf{x}'(t) &= f(\mathbf{x}(t), t), \quad t_0 \leq t \leq t_f \\ \mathbf{x}(t_0) &= \mathbf{x}_0, \end{aligned} \tag{1.1}$$

where f is a suitably continuous function. A solution of (1.1) is a continuously differentiable, and possibly vector-valued, function $\mathbf{x}(t) = [x_1(t), \ldots, x_m(t)]^\mathsf{T}$, whose derivative $\mathbf{x}'(t)$ equals $f(\mathbf{x}(t), t)$ for all $t \in [t_0, t_f]$, and for which the *initial value* $\mathbf{x}(t_0)$ equals $\mathbf{x}_0$.

Under the right conditions, namely that f is uniformly Lipschitz continuous in $\mathbf{x}(t)$ near $\mathbf{x}_0$ and continuous in t near t_0, (1.1) is well-known to have a unique solution. However, for many IVPs, it is difficult, if not impossible, to find a closed-form, analytic expression for $\mathbf{x}(t)$. In these cases, numerical methods can be used to instead *approximate* $\mathbf{x}(t)$.

As an example, consider the initial value problem

$$\begin{aligned} x'(t) &= \sin(x(t)), \\ x(0) &= x_0. \end{aligned} \tag{1.2}$$

The solution $x(t)$ is defined implicitly by

$$t = \ln \left| \frac{\cos(x_0) + \cot(x_0)}{\csc(x(t)) + \cot(x(t))} \right|.$$

This equation cannot be solved for $x(t)$, so it is difficult to understand what solutions to (1.2) look like. Since $sin(n\pi) = 0$, there are constant solutions $x_n(t) = n\pi$, $n \in \mathbb{Z}$. Using a numerical IVP solver, solutions for different values of x_0 can be approximated. Figure 1.1 shows several of these approximate solutions, along with some of the constant, or *equilibrium*, solutions.

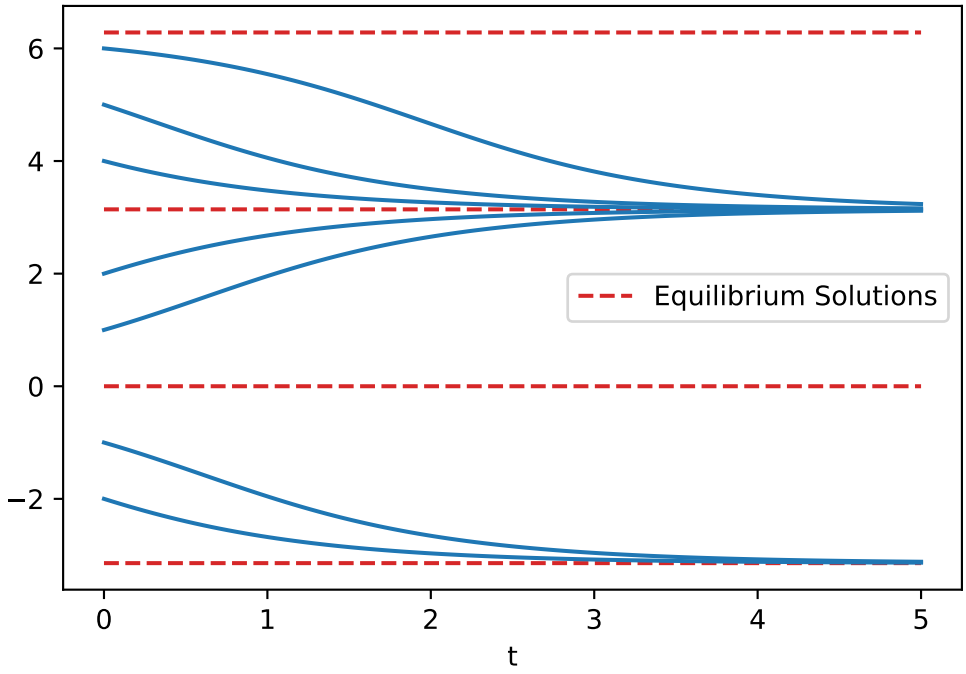


Figure 1.1: Several solutions of (1.2), using `scipy.integrate.odeint`.

Numerical Methods

For the numerical methods that follow, the key idea is to seek an approximation for the values of $\mathbf{x}(t)$ only on a finite set of values $t_0 < t_1 < \ldots < t_{n-1} < t_n \ (= t_f)$. In other words, these methods try to solve for $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ such that $\mathbf{x}_i \approx \mathbf{x}(t_i)$.

Euler's Method

For simplicity, assume that each of the n subintervals $[t_{i-1}, t_i]$ has equal length $h = (t_f - t_0)/n$. h is called the *step size*. Assuming $\mathbf{x}(t)$ is twice-differentiable, for each component function $x_j(t)$ of $\mathbf{x}(t)$ and for each i, Taylor's Theorem says that

$$x_j(t_{i+1}) = x_j(t_i) + hx_j'(t_i) + \frac{h^2}{2}x_j''(c) \text{ for some } c \in [t_i, t_{i+1}].$$

The quantity $\frac{h^2}{2}x_j''(c)$ is negligible when h is sufficiently small, and thus $x_j(t_{i+1}) \approx x_j(t_i) + hx_j'(t_i)$. Therefore, bringing the component functions of $\mathbf{x}(t)$ back together gives

$$\begin{aligned}\mathbf{x}(t_{i+1}) &\approx \mathbf{x}(t_i) + h\mathbf{x}'(t_i),\\ &\approx \mathbf{x}(t_i) + hf(\mathbf{x}(t_i), t_i).\end{aligned}$$

This approximation leads to the *Euler method*: Starting with $\mathbf{x}_0 = \mathbf{x}(t_0)$, $\mathbf{x}_{i+1} = \mathbf{x}_i + hf(\mathbf{x}_i, t_i)$ for $i = 0, 1, \ldots, n-1$. Euler's method can be understood as starting with the point at $\mathbf{x}_0$, then

calculating the derivative of $\mathbf{x}(t)$ at t_0 using $f(\mathbf{x}_0, t_0)$, followed by taking a step in the direction of the derivative scaled by h. Set that new point as $\mathbf{x}_1$ and continue.

It is important to consider how the choice of step size h affects the accuracy of the approximation. Note that at each step of the algorithm, the *local truncation error*, which comes from neglecting the $x_j''(c)$ term in the Taylor expansion, is proportional to h^2. The error $||\mathbf{x}(t_i) - \mathbf{x}_i||$ at the *ith* step comes from $i = \frac{t_i - t_0}{h}$ steps, which is proportional to h^{-1}, each contributing h^2 error. Thus the *global truncation error* is proportional to h. Therefore, the Euler method is called a *first-order method*, or a $\mathcal{O}(h)$ method. This means that as h gets small, the approximation of $\mathbf{x}(t)$ improves in two ways. First, $\mathbf{x}(t)$ is approximated at more values of t (more information about the solution), and second, the accuracy of the approximation at any t_i is improved proportional to h (better information about the solution).

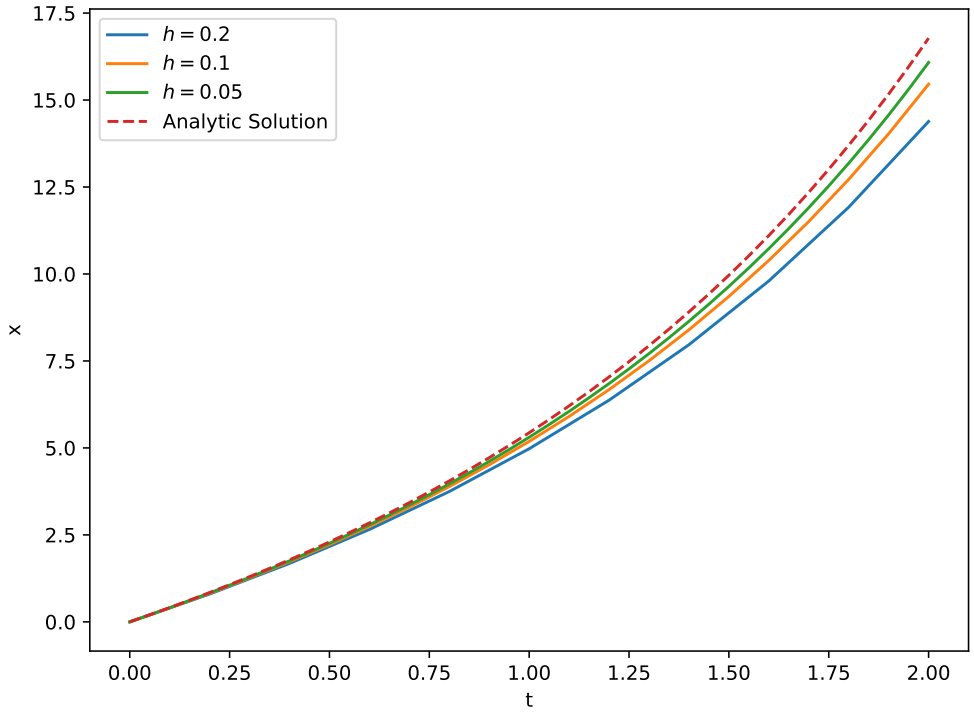


Figure 1.2: The solution of (1.3), alongside several approximations using Euler's method.

Problem 1. Write a function which implements Euler's method for an IVP of the form (1.1). Test your function on the IVP:

$$\begin{aligned} x'(t) &= x(t) - 2t + 4, \quad 0 \leq t \leq 2, \\ x(0) &= 0, \end{aligned} \tag{1.3}$$

where the analytic solution is $x(t) = -2 + 2t + 2e^t$. Use the Euler method to numerically approximate the solution with step sizes $h = 0.2, 0.1$, and 0.05. Plot the true solution alongside the three approximations, and compare your results with Figure 1.2.

Midpoint Method

The midpoint method is very similar to Euler's method. For small h, use the approximation

$$\mathbf{x}(t_{i+1}) \approx \mathbf{x}(t_i) + hf(\mathbf{x}(t_i) + \frac{h}{2}f(\mathbf{x}(t_i), t_i), t_i + \frac{h}{2},).$$

In this approximation, first set $\hat{\mathbf{x}}_i = \mathbf{x}_i + \frac{h}{2}f(\mathbf{x}_i, t_i)$, which is an Euler method step of size $h/2$. Then evaluate $f(\hat{\mathbf{x}}_i, t_i + \frac{h}{2})$, which is a more accurate approximation to the derivative $\mathbf{x}'(t)$ in the interval $[t_i, t_{i+1}]$. Finally, a step is taken in that direction, scaled by h. It can be shown that the local truncation error for the midpoint method is $\mathcal{O}(h^3)$, giving global truncation error of $\mathcal{O}(h^2)$. This is a significant improvement over the Euler method. However, it comes at the cost of additional evaluations of f and a handful of extra floating point operations on the side. This tradeoff will be considered later in the lab.

Runge-Kutta Methods

The Euler method and the midpoint method belong to a family called *Runge-Kutta methods*. There are many Runge-Kutta methods with varying orders of accuracy. Methods of order four or higher are most commonly used. A fourth-order Runge-Kutta method (RK4) iterates as follows:

$$\begin{aligned}
K_1 &= f(\mathbf{x}_i, t_i),\\
K_2 &= f(\mathbf{x}_i + \frac{h}{2}K_1, t_i + \frac{h}{2}),\\
K_3 &= f(\mathbf{x}_i + \frac{h}{2}K_2, t_i + \frac{h}{2}),\\
K_4 &= f(\mathbf{x}_i + hK_3, t_{i+1}),\\
\mathbf{x}_{i+1} &= \mathbf{x}_i + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4).
\end{aligned}$$

Runge-Kutta methods can be understood as a generalization of quadrature methods for approximating integrals, where the integrand is evaluated at specific points, and then the resulting values are combined in a weighted sum. For example, consider a differential equation

$$x'(t) = f(t)$$

Since the function f has no x dependence, this is a simple integration problem. In this case, Euler's method corresponds to the left-hand rule, the midpoint method becomes the midpoint rule, and RK4 reduces to Simpson's rule.

Advantages of Higher-Order Methods

It can be useful to visualize the order of accuracy of a numerical method. A method of order p has relative error of the form

$$E(h) = Ch^p$$

taking the logarithm of both sides yields

$$log(E(h)) = p \cdot log(h) + log(C)$$

Therefore, on a log-log plot against h, $E(h)$ is a line with slope p and intercept $log(C)$.

Problem 2. Write functions that implement the midpoint and fourth-order Runge-Kutta methods. Use the Euler, Midpoint, and RK4 methods to approximate the value of the solution for the IVP (1.3) from Problem 1 for step sizes of $h = 0.2$, 0.1, 0.05, 0.025, and 0.0125.

Plot the following graphs

- The true solution alongside the approximation obtained from each method when $h = 0.2$.
- A log-log plot (use `plt.loglog`) of the relative error $|x(2) - x_n|/|x(2)|$ as a function of h for each approximation.

Compare your second plot with Figure 1.3.

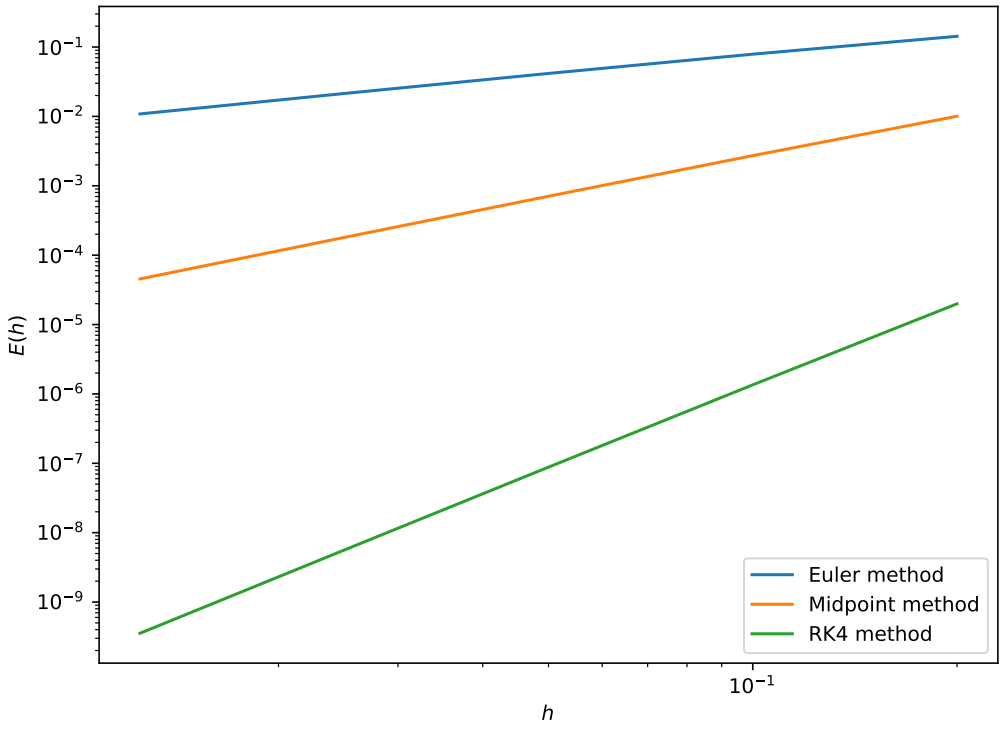


Figure 1.3: Loglog plot of the relative error in approximating $x(2)$, using step sizes $h = 0.2$, 0.1, 0.05, 0.025, and 0.0125. The slope of each line demonstrates the first, second, and fourth order convergence of the Euler, Midpoint, and RK4 methods, respectively.

The Euler, midpoint, and RK4 methods help illustrate the potential trade-off between order of accuracy and computational expense. To increase the order of accuracy, more evaluations of f must be performed at each step. It is possible that this trade-off could make higher-order methods undesirable, as (in theory) one could use a lower-order method with a smaller step size h. However, this is not generally the case. Assuming efficiency is measured in terms of the number of f-evaluations required to reach a certain threshold of accuracy, higher-order methods turn out to be much more efficient. For example, consider the IVP

$$\begin{aligned} x'(t) &= x(t)\cos(t), \quad t \in [0, 8], \\ x(0) &= 1. \end{aligned} \tag{1.4}$$

Figure 1.4 illustrates the comparative efficiency of the Euler, Midpoint, and RK4 methods applied to (1.4). The higher-order RK4 method requires fewer f-evaluations to reach the same level of relative error as the lower-order methods. As h becomes small, which corresponds to increasing functional evaluations, each method reaches a point where the relative error $|x(8) - x_n|/|x(8)|$ stops improving. This occurs when h is so small that floating point round-off error overwhelms local truncation error. Notice that the higher-order methods are able to reach a better level of relative error before this phenomena occurs.

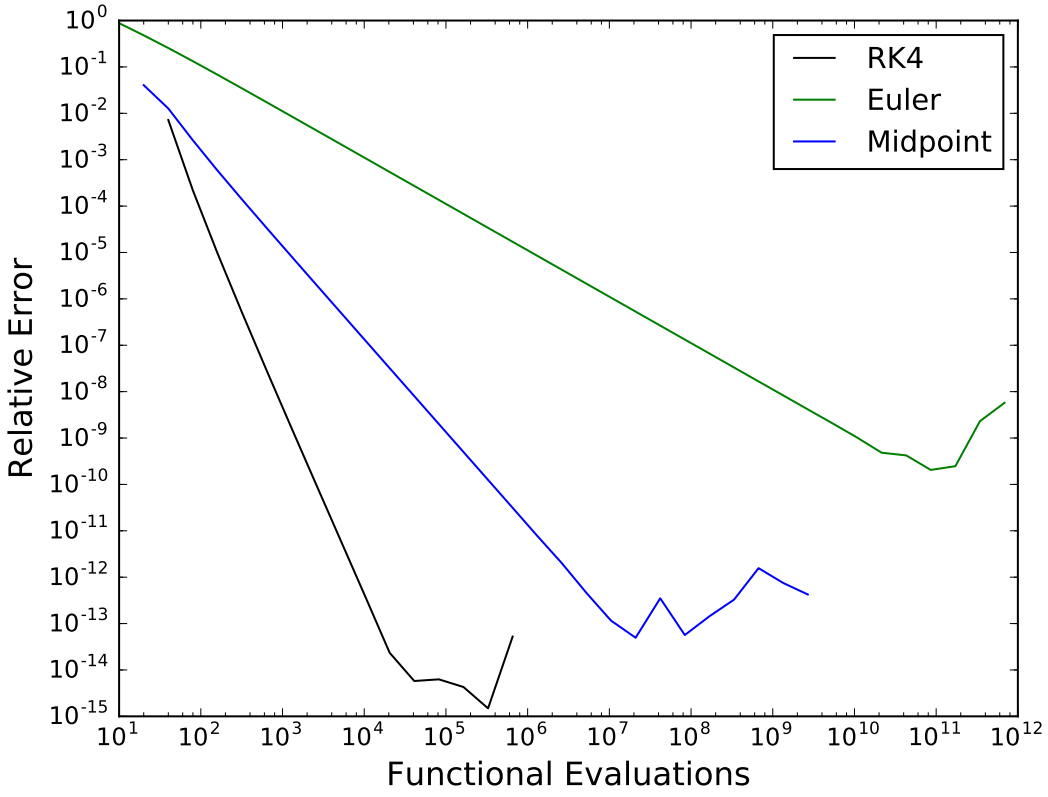


Figure 1.4: The relative error in computing the solution of (1.4) at $x = 8$ versus the number of times the right-hand side of (1.4) must be evaluated.

Harmonic Oscillators and Resonance

Harmonic oscillators are common in classical mechanics. A few examples include the pendulum (with small displacement), spring-mass systems, and the flow of electric current through various types of circuits. A harmonic oscillator $y(t)$[1] is a solution to an initial value problem of the form

$$\begin{aligned} my'' + \gamma y' + ky &= f(t), \\ y(0) = y_0, \quad y'(0) &= y_0'. \end{aligned}$$

Here, m represents the mass on the end of a spring, γ represents the effect of damping on the motion, k is the spring constant, and $f(t)$ is the external force applied.

Simple harmonic oscillators

A simple harmonic oscillator is a harmonic oscillator that is not damped, $\gamma = 0$, and is free, $f = 0$, rather than forced, $f \neq 0$. A simple harmonic oscillator can described by the IVP

$$\begin{aligned} my'' + ky &= 0, \\ y(0) = y_0, \quad y'(0) &= y_0'. \end{aligned}$$

The solution of this IVP is $y = c_1 \cos(\omega_0 t) + c_2 \sin(\omega_0 t)$, where $\omega_0 = \sqrt{k/m}$ is the natural frequency of the oscillator and c_1 and c_2 are determined by applying the initial conditions.

To solve this IVP using a Runge-Kutta method, it must be written in the form

$$\mathbf{x}'(t) = f(\mathbf{x}(t), t)$$

This can be done by setting $x_1 = y$ and $x_2 = y'$. Then we have

$$\mathbf{x}' = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}' = \begin{bmatrix} x_2 \\ \frac{-k}{m} x_1 \end{bmatrix}$$

Therefore

$$f(\mathbf{x}(t), t) = \begin{bmatrix} x_2 \\ \frac{-k}{m} x_1 \end{bmatrix}$$

Problem 3. Use the RK4 method to solve the simple harmonic oscillator satisfying

$$\begin{aligned} my'' + ky &= 0, \quad 0 \leq t \leq 20, \\ y(0) = 2, \quad y'(0) &= -1, \end{aligned} \tag{1.5}$$

for $m = 1$ and $k = 1$.

Plot your numerical approximation of $y(t)$. Compare this with the numerical approximation when $m = 3$ and $k = 1$. Consider: Why does the difference in solutions make sense physically?

[1] It is customary to write y instead of $y(t)$ when it is unambiguous that y denotes the dependent variable.

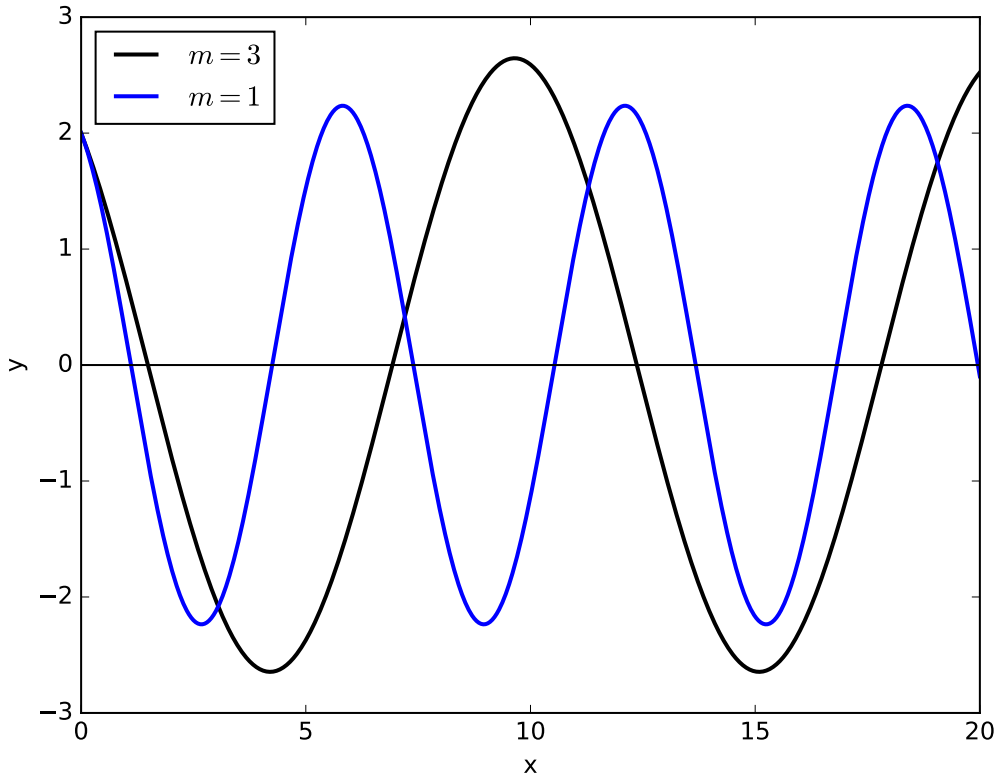


Figure 1.5: Solutions of (1.5) for several values of m.

Damped free harmonic oscillators

A damped free harmonic oscillator $y(t)$ satisfies the IVP

$$my'' + \gamma y' + ky = 0,$$
$$y(0) = y_0, \quad y'(0) = y_0'.$$

The roots of the characteristic equation are

$$r_1, r_2 = \frac{-\gamma \pm \sqrt{\gamma^2 - 4km}}{2m}.$$

Note that the real parts of r_1 and r_2 are always negative, and so any solution $y(t)$ will decay over time due to a dissipation of the system energy. There are several cases to consider for the general solution of this equation:

1. If $\gamma^2 > 4km$, then the general solution is $y(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t}$. Here the system is said to be *overdamped*. Notice from the general solution that there is no oscillation in this case.

2. If $\gamma^2 = 4km$, then the general solution is $y(t) = c_1 e^{\gamma t/2m} + c_2 t e^{\gamma t/2m}$. Here the system is said to be *critically damped*.

3. If $\gamma^2 < 4km$, then the general solution is

$$\begin{aligned} y(t) &= e^{-\gamma t/2m}\left[c_1\cos(\mu t) + c_2\sin(\mu t)\right], \\ &= Re^{-\gamma t/2m}\sin(\mu t + \delta), \end{aligned}$$

where R and δ are fixed, and $\mu = \sqrt{4km - \gamma^2}/2m$. This system does oscillate.

Problem 4. Use the RK4 method to solve for the damped free harmonic oscillator satisfying

$$\begin{aligned} &y'' + \gamma y' + y = 0, \quad 0 \le t \le 20, \\ &y(0) = 1, \quad y'(0) = -1. \end{aligned}$$

For $\gamma = 1/2$, and $\gamma = 1$, simultaneously plot your numerical approximations of y.

Forced harmonic oscillators without damping

Consider the systems described by the differential equation

$$my''(t) + ky(t) = F(t). \tag{1.6}$$

In many instances, the external force $F(t)$ is periodic, so assume that $F(t) = F_0\cos(\omega t)$. If $\omega_0 = \sqrt{k/m} \neq \omega$, then the general solution of 1.6 is given by

$$y(t) = c_1\cos(\omega_0 t) + c_2\sin(\omega_0 t) + \frac{F_0}{m(\omega_0^2 - \omega^2)}\cos(\omega t).$$

If $\omega_0 = \omega$, then the general solution is

$$y(t) = c_1\cos(\omega_0 t) + c_2\sin(\omega_0 t) + \frac{F_0}{2m\omega_0}t\sin(\omega_0 t).$$

When $\omega_0 = \omega$, the solution contains a term that grows arbitrarily large as $t \to \infty$. If we included damping, then the solution would be bounded but large for small γ and ω close to ω_0.

Consider a physical spring-mass system. Equation 1.6 holds only for small oscillations; this is where Hooke's law is applicable. However, the fact that the equation predicts large oscillations suggests the spring-mass system could fall apart as a result of the external force. This mechanical resonance has been known to cause failure of bridges, buildings, and airplanes.

Problem 5. Use the RK4 method to solve the damped and forced harmonic oscillator satisfying

$$\begin{aligned} &2y'' + \gamma y' + 2y = 2\cos(\omega t), \quad 0 \le t \le 40, \\ &y(0) = 2, \quad y'(0) = -1. \end{aligned} \tag{1.7}$$

For the following values of γ and ω, plot your numerical approximations of $y(t)$: $(\gamma, \omega) = (0.5, 1.5)$, $(0.1, 1.1)$, and $(0, 1)$. Compare your results with Figure1.7.

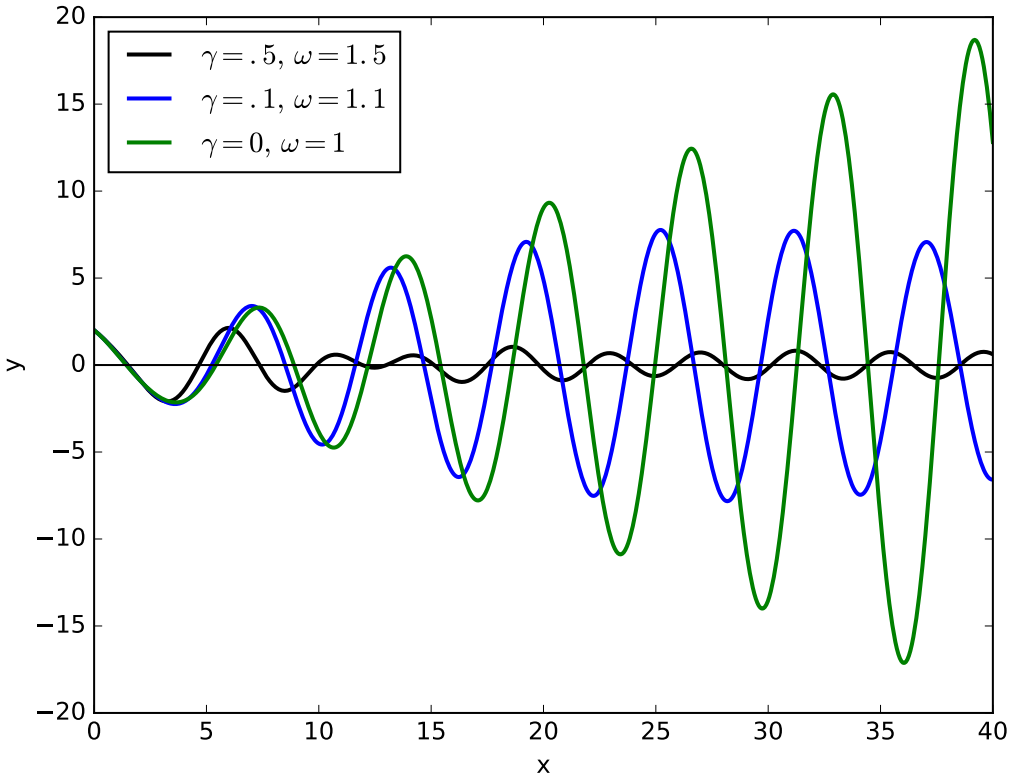


Figure 1.6: Solutions of (1.7) for several values of ω and γ.

1 Predator-Prey and Weight Change Models

Lab Objective: *We introduce built-in methods for solving Initial Value Problems and apply the methods to two dynamical systems. The first system looks at the relationship between a predator and its prey. The second model is a weight change model based on thermodynamics and kinematics.*

Predator-Prey Model

ODEs are commonly used to model relationships between predator and prey populations. For example, consider the populations of wolves, the predator, and rabbits, the prey, in Yellowstone National Park. Let $r(t)$ and $w(t)$ represent the rabbit and wolf populations respectively at time t, measured in years. We will make a few assumptions to simplify our model:

- In the absence of wolves, the rabbit population grows at a positive rate proportional to the current population. Thus when $w(t) = 0$, $dr/dt = \alpha r(t)$, where $\alpha > 0$.

- In the absence of rabbits, the wolves die out. Thus when $r(t) = 0$, $dw/dt = -\delta w(t)$, where $\delta > 0$.

- The number of encounters between rabbits and wolves is proportional to the product of their populations. The wolf population grows proportional to the number of encounters by $\beta r(t)w(t)$ (where $\beta > 0$), and the rabbit population decreases proportional to the number of encounters by $-\gamma r(t)w(t)$ (where $\gamma > 0$).

This leads to the following system of ODEs:

$$\begin{aligned} \frac{dr}{dt} &= \alpha r - \beta rw = r(\alpha - \beta w) \\ \frac{dw}{dt} &= -\delta w + \gamma rw = w(-\delta + \gamma r) \end{aligned} \tag{1.1}$$

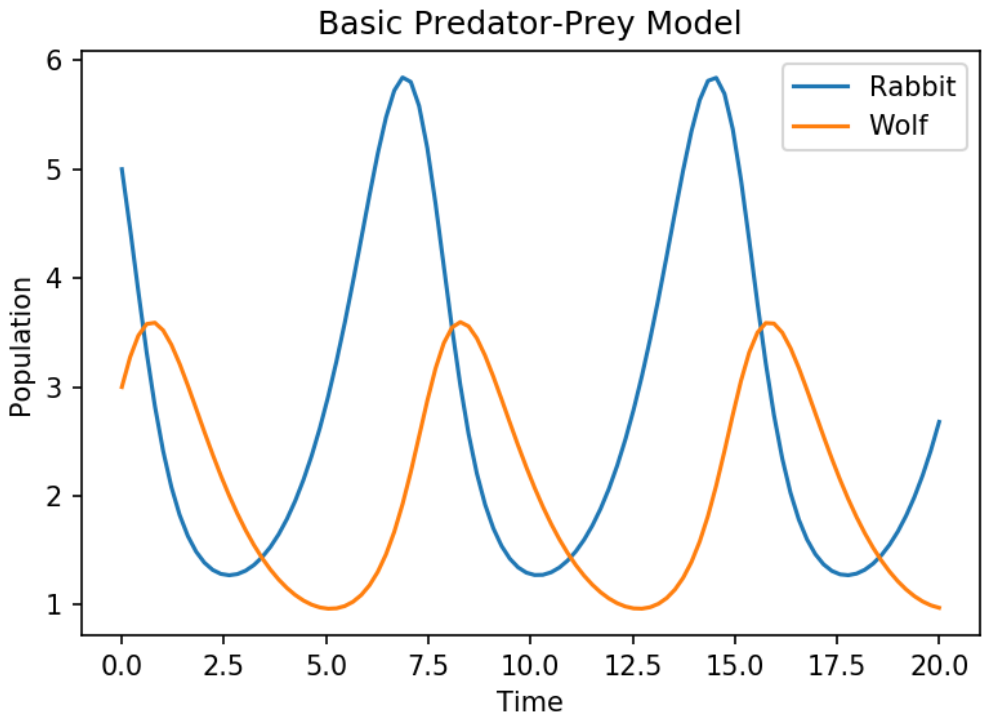


Figure 1.1: The solution to the system found in (1.1)

Problem 1. Define the function `predator_prey()` that accepts the current $r(t)$ and $w(t)$ values as a 1d array y, and the current time t, and returns the right hand side of (1.1) as an ndarray. Use $\alpha = 1.0$, $\beta = 0.5$, $\delta = 0.75$, and $\gamma = 0.25$ as your growth parameters.
Hint: you will want to use `solve_ivp`.

Problem 2. Use `solve_ivp` to solve (1.1) with initial conditions $(r_0, w_0) = (5, 3)$ and time ranging from 0 to 20 years. Display the resulting rabbit and wolf populations over time (stored as rows in the attribute `y` of the output of `solve_ivp`) on the same plot. Your graph should match the graph in figure 1.1.

Variations on the Predator-Prey

The Lotka-Volterra model

The representation of the predator-prey relationship found in (1.1) is called the Lotka-Volterra predator-prey model and is typically given by

$$\begin{aligned}\frac{du}{dt} &= \alpha u - \beta uv,\\ \frac{dv}{dt} &= -\delta v + \gamma uv.\end{aligned}$$

where u and v represent the prey and predator populations, respectively. Here α, β, δ, and γ are the same as before but now for an arbitrary prey and predator.

The equlibria (fixed points) of a system occur when the derivatives are zero. In this example, that occurs at $(u, v) = (0, 0)$ and $(u, v) = (\frac{c}{d}, \frac{a}{b})$. Visualizing the phase portrait helps to give more insight into the dynamics of a system. We will do this by first nondimensionalzing our system to reduce the number of parameters. Let $U = \frac{\gamma}{\delta}u$, $V = \frac{\beta}{\alpha}v$, $\bar{t} = \alpha t$, and $\eta = \frac{\gamma}{\alpha}$. Substituting into the original ODEs we obtain the nondimensional system of equations

$$\begin{aligned}\frac{dU}{d\bar{t}} &= U(1 - V),\\ \frac{dV}{d\bar{t}} &= \eta V(U - 1).\end{aligned} \tag{1.2}$$

Problem 3. Similar to problem 1, define the function `Lotka_volterra()` that takes in the current predator and prey populations as a 1d array y and the current time as a float t and returns the right hand side of the system (1.2) with $\eta = 1/3$.

The following three lines of code plot the phase portrait of (1.2). For more documentation on quiver plots see the documentation.

```
Y1, Y2 = np.meshgrid(np.linspace(0, 4.5, 25), np.linspace(0, 4.5, 25))
dU, dV = Lotka_Volterra(0, (Y1, Y2))
Q = plt.quiver(Y1[::3, ::3], Y2[::3, ::3], U[::3, ::3], V[::3, ::3])
```

Using `solve_ivp`, solve (1.2) with three different initial conditions $y_0 = (1/2, 1/3)$, $y_0 = (1/2, 3/4)$, and $y_0 = (1/16, 3/4)$ and time domain $t = [0, 13]$. Plot these three solutions on the same graph as the phase portrait and the equilibria $(0, 0)$ and $(1, 1)$.

Since your solutions are being plotted with the phase portrait, plot the two populations against each other (instead of both individually against time). Your plot should match 1.2.

The Logistic model

Notice that the Lotka-Volterra equations predict prey populations will grow exponentially in the absence of predators. The logistic predator-prey equations change this dynamic by adding a carrying

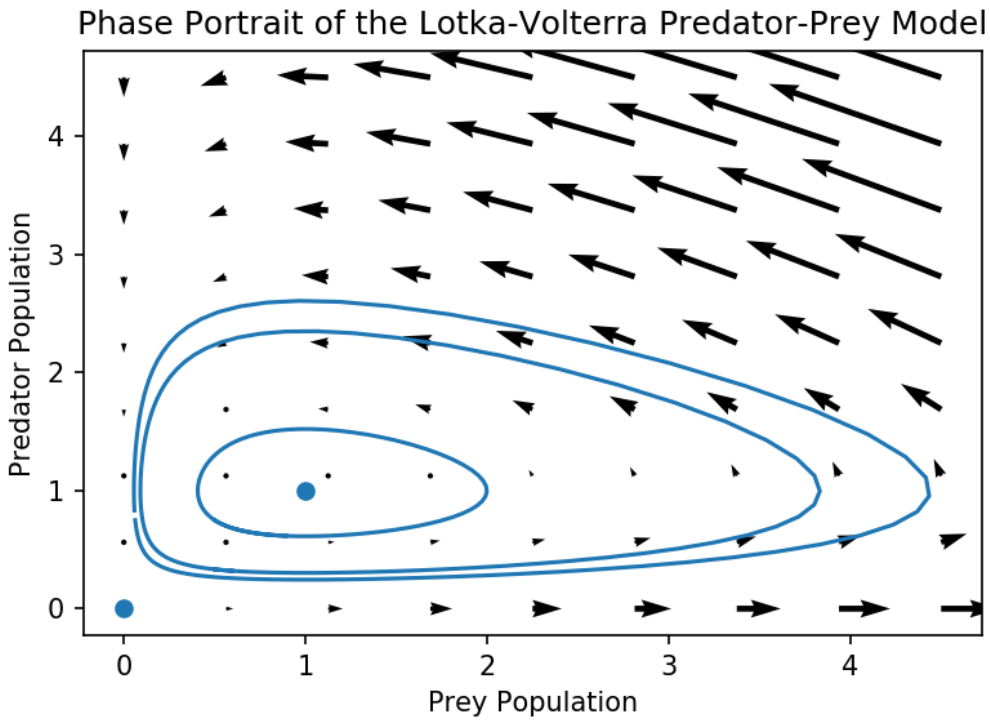


Figure 1.2: The phase portrait for the nondimensionalized Lotka-Volterra predator-prey equations with parameters $\eta = 1/3$.

capacity K to the prey population:

$$\begin{aligned} \frac{du}{dt} &= \alpha u\left(1 - \frac{u}{K}\right) - \beta uv, \\ \frac{dv}{dt} &= -\delta v + \gamma uv. \end{aligned}$$

We can again do dimensional analysis on this system to simplify parameters. Let $U = \frac{u}{K}$, $V = \frac{\beta}{\alpha}v$, $\bar{t} = \alpha t$, $\eta = \frac{\gamma K}{\alpha}$, and $\rho = \frac{\delta}{\gamma K}$. Then the nondimensional logistic equations are

$$\begin{aligned} \frac{dU}{d\bar{t}} &= U(1 - U - V), \\ \frac{dV}{d\bar{t}} &= \eta V(U - \rho). \end{aligned} \tag{1.3}$$

Problem 4. Define a new function `Logistic_Model()` that takes in the current predator and prey populations y and the current time t and returns the right hand side of (1.3) as a tuple. Use `solve_ivp` to compute solutions (U, V) of (1.3) for initial conditions $(1/3, 1/3)$ and $(1/2, 1/5)$ with $(t0, tf) = (0, 13)$. Do this for parameter values η, $\rho = 1$, 0.3 and also for values η, $\rho = 1$, 1.1.

Create a phase portrait for the logistic equations using both sets of parameter values. Plot

the direction field, all equilibrium points, and both solution orbits on the same plot for each set of parameter values.

A Weight Change Model

The main idea behind weight change is simple. If a person takes in more energy than they expend, they gain weight. If they take in less than they expend, they lose weight. Let *energy balance* EB be the difference between *energy intake* EI and *energy expenditure* EE, so that

$$EB = EI - EE.$$

If the balance is positive, weight is gained and similarly if the balance is negative, weight is lost.

A person's body weight at a time t can be expressed as the sum of the weight of their fat tissue $F(t)$ and the weight of their lean tissue $L(t)$; that is, $BW(t) = F(t) + L(t)$. Using this, the change in body weight can be expressed as the following system of ODEs:

$$\begin{aligned} \frac{dF}{dt} &= \frac{(1-p(t))EB(t)}{\rho_F}, \\ \frac{dL}{dt} &= \frac{p(t)EB(t)}{\rho_L}, \end{aligned} \tag{1.4}$$

where $(1-p(t))$ and $p(t)$ represent the proportion of the energy balance $(EB(t))$ that results in a change in the quantity of fatty or lean tissue, respectively. The constants ρ_F and ρ_L represent the energy density of fatty and lean tissue, approximated as $\rho_F = 9400$ kcal/kg and $\rho_L = 1800$ kcal/kg.

To solve this system, we first need to express $p(t)$ and $EB(t)$ in terms of F and L. These functions will also depend on physical activity level, PAL, and energy intake, EI, which vary among individuals.

We will find an expression for $p(t)$ using Forbes' Law[1] which states that

$$\frac{dF}{dL} = \frac{F}{10.4}.$$

Notice

$$\frac{F}{10.4} = \frac{dF}{dL} = \frac{dF/dt}{dL/dt} = \frac{\dfrac{(1-p(t))EB(t)}{\rho_F}}{\dfrac{p(t)EB(t)}{\rho_L}} = \frac{\rho_L}{\rho_F}\frac{1-p(t)}{p(t)}.$$

Solving for $p(t)$ gives Forbes' equation

$$p(t) = \frac{C}{C+F(t)} \quad \text{where} \quad C = 10.4\frac{\rho_L}{\rho_F}. \tag{1.5}$$

We will now find an expression for $EB(t)$. Recall $EB(t) = EI - EE$. We will use the following expression for energy expenditure (EE) to define $EB(t)$.

$$EE = PAL \times RMR \tag{1.6}$$

where PAL is physical activity level (as previously mentioned) and RMR is resting metabolic rate. Physical activity level can be determined using the table above.

[1] *Lean body mass-body fat interrelationships in humans*, Forbes, G.B.; *Nutrition reviews*, pgs 225-231, 1987.

1.40–1.69	People who are sedentary and do not exercise regularly, spend most of their time sitting, standing, with little body displacement
1.70–1.99	People who are active, with frequent body displacement throughout the day or who exercise frequently
2.00–2.40	People who engage regularly in strenuous work or exercise for several hours each day

Table 1.1: This is a rough guide for physical activity level (PAL).

We will use the following equation for computing RMR,

$$RMR = K + \gamma_F F(t) + \gamma_L L(t) + \eta_F \frac{dF}{dt} + \eta_L \frac{dL}{dt} + \beta_{at} EI, \tag{1.7}$$

where $\gamma_F = 3.2$ kcal/kg/d, $\gamma_L = 22$ kcal/kg/d, $\eta_F = 180$ kcal/kg, and $\eta_L = 230$ kcal/kg [2] [3]. Further, we let $\beta_{at} = 0.14$ denote the coefficient for adaptive thermogenesis. Finally, we remark that the constant K can be tuned to an individual's body type directly through RMR and fat measurement, and is assumed to remain constant over time.

Thus, since the input EI is assumed to be known, we can use (1.6), (1.7) and (1.5) to write (1.4) in terms of F and L, thus allowing us to close the system of ODEs.

Specifically, we have

$$\begin{aligned}
RMR = \frac{EE}{PAL} &= K + \gamma_F F(t) + \gamma_L L(t) + \eta_F \frac{dF}{dt} + \eta_L \frac{dL}{dt} + \beta_{at} EI \\
\frac{1}{PAL}\left(EE - EI + EI\right) &= K + \gamma_F F(t) + \gamma_L L(t) \\
&\quad + \left(\frac{\eta_F}{\rho_F}(1 - p(t)) + \frac{\eta_L}{\rho_L} p(t)\right) EB(t) + \beta_{at} EI. \\
\left(\frac{1}{PAL} - \beta_{at}\right) EI &= K + \gamma_F F(t) + \gamma_L L(t) \\
&\quad + \left(\frac{\eta_F}{\rho_F}(1 - p(t)) + \frac{\eta_L}{\rho_L} p(t) + \frac{1}{PAL}\right) EB(t).
\end{aligned}$$

Solving for $EB(t)$ in the last equation yields

$$EB(t) = \frac{\left(\frac{1}{PAL} - \beta_{at}\right) EI - K - \gamma_F F(t) - \gamma_L L(t)}{\frac{\eta_F}{\rho_F}(1 - p(t)) + \frac{\eta_L}{\rho_L} p(t) + \frac{1}{PAL}}. \tag{1.8}$$

In equilibrium ($EB = 0$), this gives us

$$K = \left(\frac{1}{PAL} - \beta_{at}\right) EI - \gamma_F F - \gamma_L L. \tag{1.9}$$

Thus, for a subject who has maintained the same weight for a while, one can determine K by using (1.9), if they know their average caloric intake and amount of fat (assume $L = BW - F$).

[2] *Modeling weight-loss maintenance to help prevent body weight regain*; Hall, K.D. and Jordan, P.N.; *The American journal of clinical nutrition*, pg 1495, 2008

[3] *Quantification of the effect of energy imbalance on bodyweight*; Hall, K.D. et al.; *The Lancet*, pgs 826-837, 2011

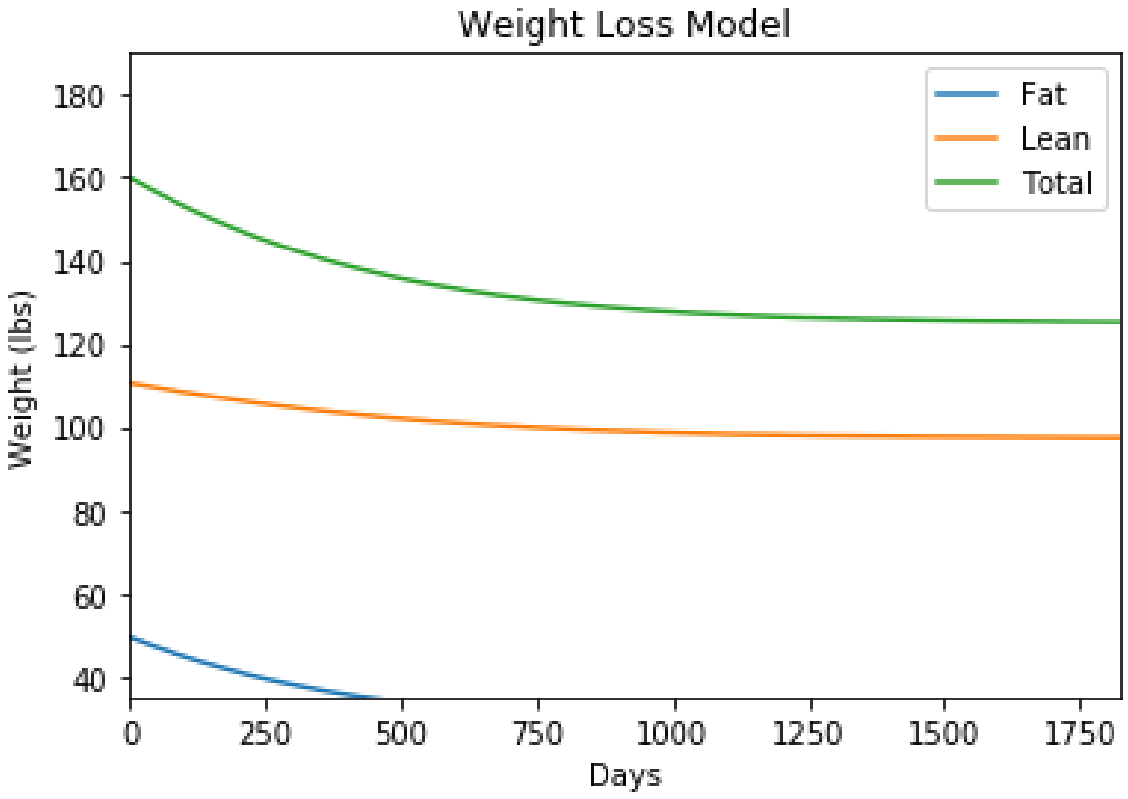


Figure 1.3: The solution of the weight change model for problem 6.

Problem 5. Write a function `forbes()` which takes as input `F`, the weight of fat tissue at a given time (i.e. the function $F(t)$ evaluated at a certain time), and returns Forbe's equation given in (1.5). Also write the function `energy_balance()` which takes as input `F`, `L`, `PAL`, and `EI` and returns the energy balance as given in (1.8). In `energy_balance()` we also have that `F` is the fat tissue weight at a given time, and `L` is the lean tissue weight at a given time.

Using `forbes()` and `energy_balance()`, define the function `weight_odesystem()` which takes as input the current time as a float t and the current fat and lean weights as an array y and returns the right hand side of (1.4) as a tuple.

Use $\rho_F = 9400$, $\rho_L = 1800$, $\gamma_F = 3.2$, $\gamma_L = 22$, $\eta_F = 180$, $\eta_L = 230$, $K = 0$ and $\beta_{AT} = 0.14$.

Hint: The functions `forbes()` and `energy_balance()` are not time dependent in the same way equations (1.5) and (1.8) are. The time dependent portions of these functions, $F(t)$ and $L(t)$, are determined by what will be input from the `y` argument of `weight_odesystem()`.

Problem 6. Consider the initial value problem corresponding to (1.4). The following function returns the fat mass of an individual based on body weight (kg), age (years), height (meters), and sex. Use this function to define initial conditions F_0 and L_0 for the IVP above: $F_0 =$

$fat\_mass(args^*)$, $L_0 = BW - F_0$.

```
def fat_mass(BW, age, H, sex):
    BMI = BW / H**2.
    if sex == 'male':
        return BW * (-103.91 + 37.31 * log(BMI) + 0.14 * age) / 100
    else:
        return BW * (-102.01 + 39.96 * log(BMI) + 0.14 * age) / 100
```

Suppose a 38 year old female, standing 5'8" and weighing 160 lbs, reduces her intake from 2143 to 2025 calories/day, and increases her physical activity from little to no exercise (PAL=1.4) to exercising to 2-3 days per week (PAL=1.5).

Use (1.9) and the original intake and phyical activity levels to compute K for this system. Then use `solve_ivp` to solve the IVP. Graph the solution curve for this single-stage weightloss intervention over a period of 5 years. Your plot should match figure 1.3.

Note the provided code requires quantities in metric units (kilograms, meters, days) while our graph is converted to units of pounds and days. Use the conversions 1 lb = 2.204 kg, 1 ft = 0.305 m, and 1 yr = 365 days.

Problem 7. Modify the preceding problem to handle a two stage weightloss intervention: Suppose for the first 16 weeks intake is reduced from 2143 to 1600 calories/day and physical activity is increased from little to no exercise (PAL=1.4) to an hour of exercise 5 days per week (PAL=1.7). The following 16 weeks intake is increased from 1600 to 2025 calories/day, and exercise is limited to only 2-3 days per week (PAL=1.5).

You will need to recompute F_0, and L_0 at the end of the first 16 weeks, but K will stay the same. Find and graph the solution curve over the 32 week period.

1 Lorenz Equations

Lab Objective: *Investigate the behavior of a system that exhibits chaotic behavior. Demonstrate methods for visualizing the evolution of a system.*

Chaos is everywhere. It can crop up in unexpected places and in remarkably simple systems, and a great deal of work has been done to describe the behavior of chaotic systems. One primary characteristic of chaos is that small changes in initial conditions result in large changes over time in the solution curves.

The Lorenz System

One of the earlier examples of chaotic behavior was discovered by Edward Lorenz. In 1963, while working to study atmospheric dynamics, he derived the simple system of equations

$$\begin{aligned}\frac{\partial x}{\partial t} &= \sigma\,(y - x)\\ \frac{\partial y}{\partial t} &= \rho x - y - xz\\ \frac{\partial z}{\partial t} &= xy - \beta z\end{aligned}$$

where σ, ρ, and β are all constants. After deriving these equations, he plotted the solutions and observed some unexpected behavior. For appropriately chosen values of σ, ρ, and β, the solutions did not tend toward any steady fixed points, nor did the system permit any stable cycles. The solutions did not tend off toward infinity either. With further work, he began the study of what was called a *strange attractor*. This system, though relatively simple, exhibits chaotic behavior.

Problem 1. Write a function that implements the Lorenz equations. Let $\sigma = 10$, $\rho = 28$, $\beta = \frac{8}{3}$. Make a 3D plot of a solution to the Lorenz equations, where the initial conditions x_0, y_0, z_0 are each drawn randomly from a uniform distribution on $[-15, 15]$. As usual, use `scipy.integrate.odeint` to compute an approximate solution. Compare your results with Figure 1.1.

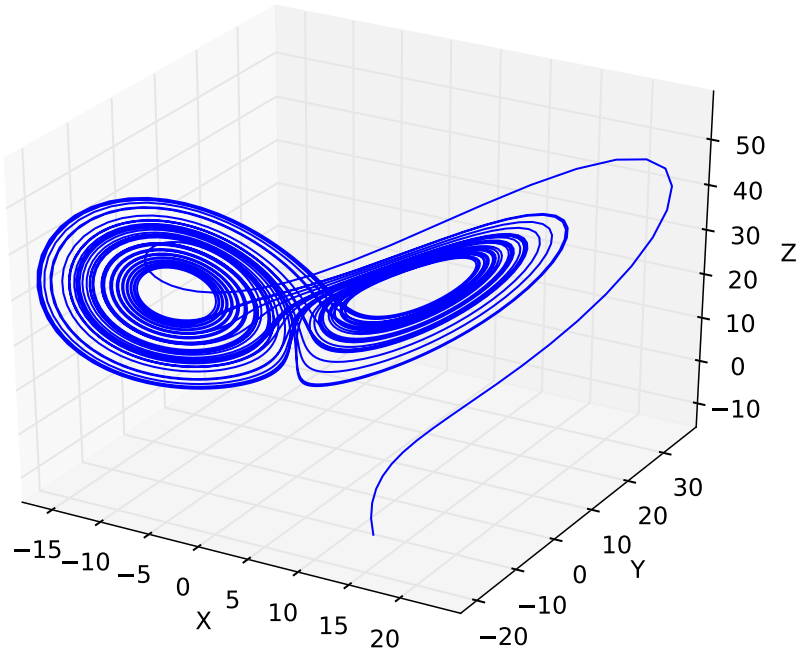


Figure 1.1: Approximate solution to the Lorenz equation with random initial conditions

Basin of Attraction

Notice in the first problem that the solution tended to a 'nice' region. This region is a basin of attraction, and the set of numerical values towards which a system will converge to is an **attractor**. Consider what happens when we change up the initial conditions.

Problem 2. To better visualize the Lorenz attractor, produce a single 3D plot displaying three solutions to the Lorenz equations, each with random initial conditions (as before). Compare your results with Figure 1.2.

Chaos

Chaotic systems exhibit high sensitivity to initial conditions. This means that a small difference in initial conditions may result in solutions that diverge significantly from each other. However, chaotic systems are not *random*. According to Lorenz, chaos is "when the present determines the future, but the approximate present does not approximately determine the future."

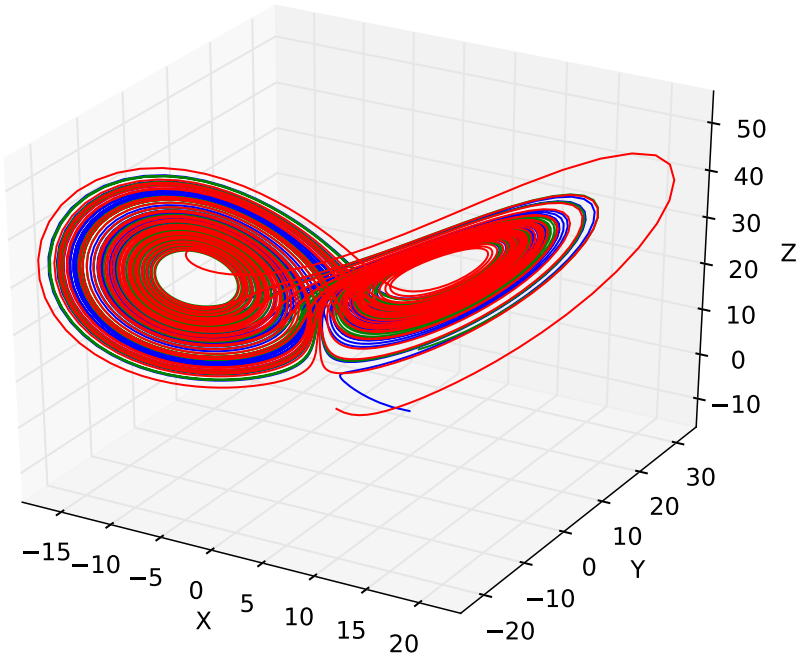


Figure 1.2: Multiple solutions to the Lorenz equation with random initial conditions

Problem 3. Use `matplotlib.animation.FuncAnimation` to produce a 3D animation of two solutions to the Lorenz equations with nearly identical initial conditions. To make the initial conditions, draw x_0, y_0, z_0 as before, and then make a second initial condition by adding a small perturbation to the first (Hint: try using `np.random.randn(3)*(1e-10)` for the perturbation). Note that it may take several seconds before the separation between the two solutions will be noticeable.

The animation should display a point marker as well as the past trajectory curve for each solution. Save your animation as `lorenz_animation.mp4`.

In a chaotic system, round-off error implicit in a numerical method can also cause divergent solutions. For example, using a Runge-Kutta method with two different values for the stepsize h on identical initial conditions will still result in approximations that differ in a chaotic fashion.

Problem 4. The `odeint` function allows user to specify error tolerances (similar to setting a value of h in a Runge-Kutta method). Using a single initial condition, produce two approximations by using the `odeint` arguments (`atol=1e-15, rtol=1e-13`) for the first approximation

and (`atol=1e-12, rtol=1e-10`) for the second.

As in the previous problem, use `FuncAnimation` to animation both solutions. Save the animation as `lorenz_animation2.mp4`.

Lyapunov Exponents

The *Lyapunov exponent* of a dynamical system is one measure of how chaotic a system is. While there are more conditions for a system to be considered chaotic, one of the primary indicators of a chaotic system is *extreme sensitivity to initial conditions*. Strictly speaking, this is saying that a chaotic system is poorly conditioned. In a chaotic system, the sensitivity to changes in initial conditions depends expoentially on the time the system is allowed to evolve. If $\delta(t)$ represents the difference between two solution curves, when $\delta(t)$ is small, the following approximation holds.

$$\|\delta(t)\| \sim \|\delta(0)\|e^{\lambda t}$$

where λ is a constant called the Lyapunov exponent. In other words, $\log(\|\delta(t)\|)$ is approximately linear as a function of time, with slope λ. For the Lorenz system (and for the parameter values specified in this lab), experimentally it can be verified that $\lambda \approx .9$.

Problem 5. Estimate the Lyapunov exponent of the Lorenz equations by doing the following:

- Produce an initial condition that already lies in the attractor. This can be done by using a random "dummy" initial condition, approximating the resulting solution to the Lorenz system for a short time, and then using the endpoint of that solution (which is now in the attractor) as the desired initial condition.
- Produce a second initial condition by adding a small perturbation to the first (as before).
- For both initial conditions, use `odeint` to produce approximate solutions for $0 \leq t \leq 10$
- Compute $\|\delta(t)\|$ by taking the norm of the vector difference between the two solutions for each value of t.
- Use `scipy.stats.linregress` to calculate a best-fit line for $\log(\|\delta(t)\|)$ against t.
- The slope of the best-fit line is an approximation for the Lyapunov exponent λ

Produce a plot similar to Figure 1.3 by using `plt.semilogy`.

Hint: Remember that the best-fit line you calculated corresponds to a best-fit exponential for $\|\delta(t)\|$. If `a` and `b` are the slope and intercept of the best-fit line, the best-fit exponential can be plotted using `plt.semilogy(t,np.exp(a*t+b))`.

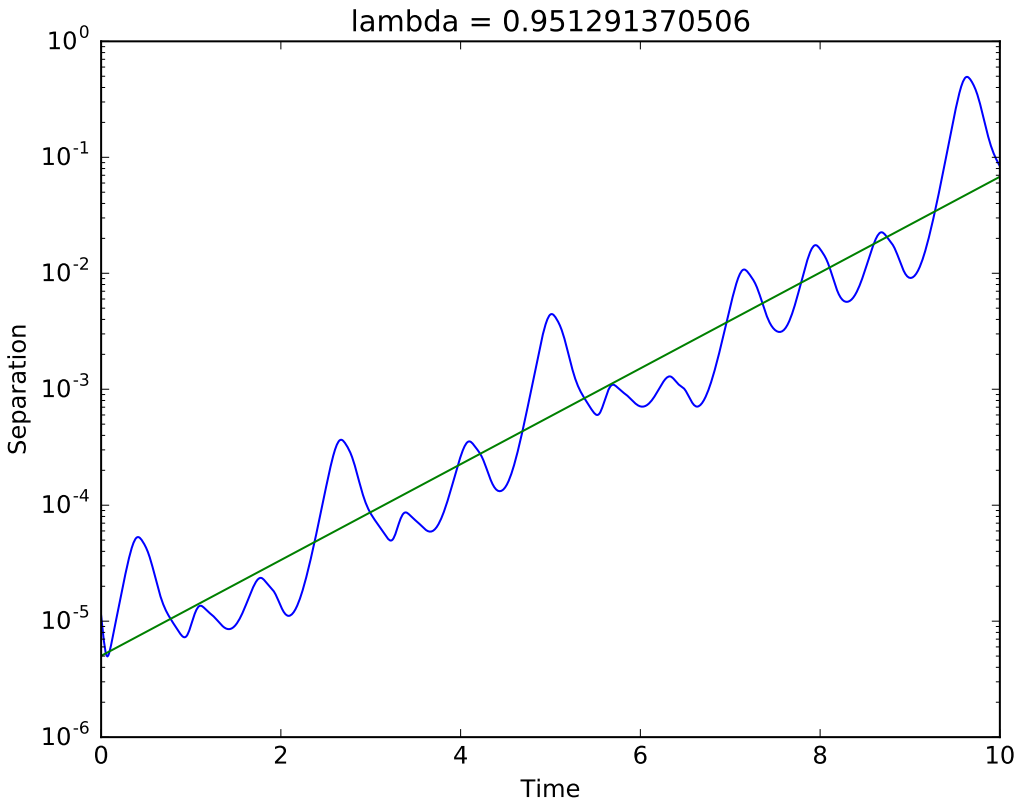


Figure 1.3: A semilog plot of the separation between two solutions to the Lorenz equations together with a fitted line that gives a rough estimate of the Lyapunov exponent of the system.

1 Bifurcations and Hysteresis

Recall that any ordinary differential equation can be written as a first order system of ODEs,

$$\dot{x} = F(x), \quad \dot{x} := \frac{d}{dt}x(t). \tag{1.1}$$

Many interesting applications and physical phenomena can be modeled using ODEs. Given a mathematical model of the form (1.1), it is important to understand geometrically how its solutions behave. This information can then be conveyed in a phase portrait, a graph describing solutions of (1.1) with differential initial conditions. The first step in constructing a phase portrait is to find the equilibrium solutions of the equation, i.e., the zeros of $F(x)$, and to determine their stability.

It is often the case that the mathematical model we study depends on some parameter or set of parameters λ. Thus the ODE becomes

$$\dot{x} = F(x, \lambda). \tag{1.2}$$

The parameter λ can then be tuned to better fit the physical application. As λ varies, the equilibrium solutions and other geometric features of (1.2) may suddenly change. A value of λ where the phase portrait changes is called a *bifurcation point*; the study of how these changes occur is called *bifurcation theory*. The parameter values and corresponding equilibrium solutions are often graphed together in a bifurcation diagram.

As an example, consider the scalar differential equation

$$\dot{x} = x^2 + \lambda. \tag{1.3}$$

For $\lambda > 0$ equation (1.3) has no equilibrium solutions. At $\lambda = 0$ the equilibrium point $x = 0$ appears, and for $\lambda < 0$ it splits into two equilibrium points. For this system, a bifurcation occurs at $\lambda = 0$. This is an example of a saddle-node bifurcation. The bifurcation diagram is shown in Figure 1.1

Suppose that $F(x_0, \lambda_0) = 0$. We use a method called natural embedding to find zeros (x, λ) of F for nearby values of λ. Specifically, we step forward in λ by letting $\lambda_1 = \lambda_0 + \triangle\lambda$, and use Newton's method to find the value x_1 that satisfies $F(x_1, \lambda_1) = 0$. This method works well except when λ is near a bifurcation point λ^*.

The following code implements the natural embedding algorithm, and then uses that algorithm to find the curves in the bifurcation diagram for (1.3). Notice that this algorithm needs a good initial guess for x_0 to get started.

```
import numpy as np
```

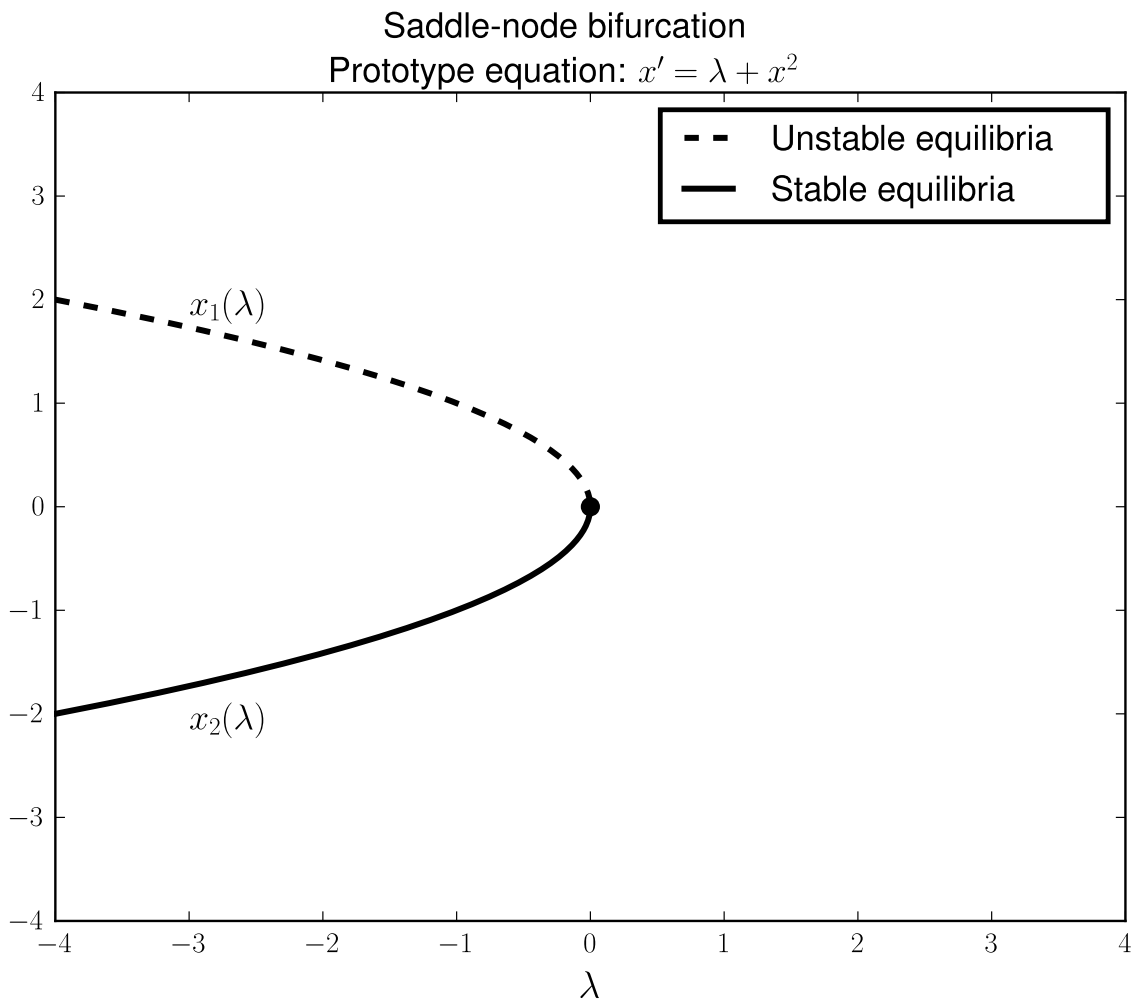


Figure 1.1: Bifurcation diagram for the equation $\dot{x} = \lambda + x^2$.

```
import matplotlib.pyplot as plt
from scipy.optimize import newton

def EmbeddingAlg(param_list, guess, F):
    X = []
    for param in param_list:
        try:
            # Solve for x_value making F(x_value, param) = 0.
            x_value = newton(F, guess, fprime=None, args=(param,), tol=1E-7, ↩
                maxiter=50)
            # Record the solution and update guess for the next iteration.
            X.append(x_value)
            guess = x_value
        except RuntimeError:
            # If Newton's method fails, return a truncated list of parameters
```

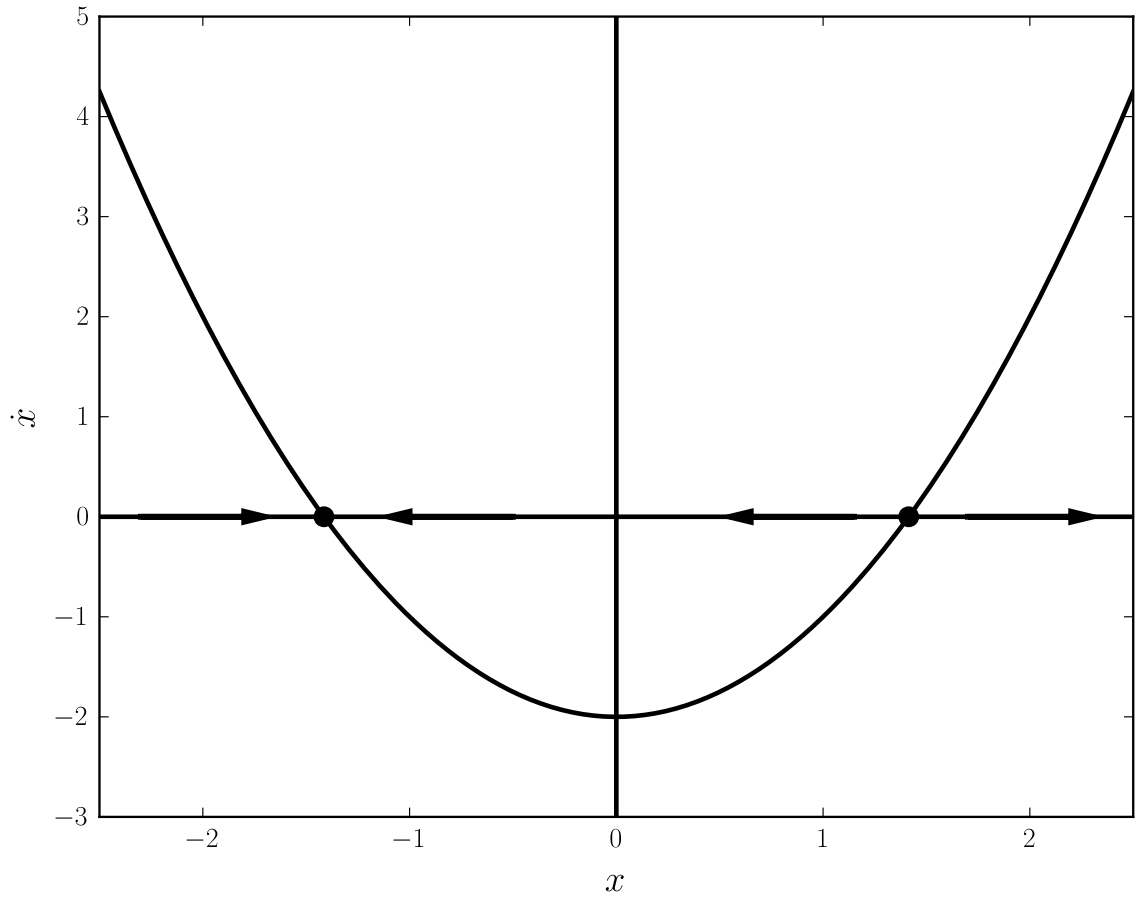


Figure 1.2: Phase Portrait for the equation $\dot{x} = -2 + x^2$.

```
            # with the corresponding x values.
            return param_list[:len(X)], X
    # Return the list of parameters and the corresponding x values.
    return param_list, X

def F(x, lmbda):
    return x**2 + lmbda

# Top curve shown in the bifurcation diagram
C1, X1 = EmbeddingAlg(np.linspace(-5, 0, 200), np.sqrt(5), F)
# The bottom curve
C2, X2 = EmbeddingAlg(np.linspace(-5, 0, 200), -np.sqrt(5), F)
```

Problem 1. Use the natural embedding algorithm to create a bifurcation diagram for the

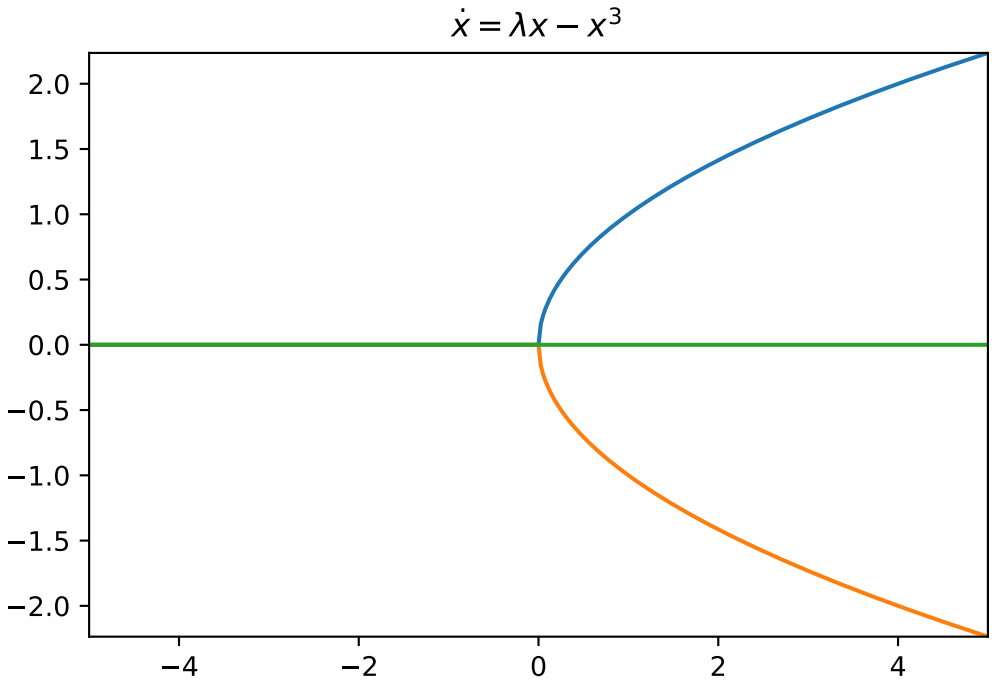


Figure 1.3: Bifurcation diagram for the equation $\dot{x} = \lambda x - x^3$.

differential equation

$$\dot{x} = \lambda x - x^3.$$

This type of bifurcation is called a pitchfork bifurcation (you should see a pitchfork in your diagram).

Hints: Essentially this amounts to running the same code as the example, but with different parameters and function calls so that you are tracing through the right curves for this problem. To make this first problem work, you will want to have your 'linspace' run from high to low instead of from low to high. There will be three different lines in this image. See Figure 1.3.

Problem 2. Create bifurcation diagrams for the differential equation

$$\dot{x} = \eta + \lambda x - x^3,$$

where $\eta = -1, -.2, .2$ and 1. Notice that when $\eta = 0$ you can see the pitchfork bifurcation of the previous problem. There should be four different images, one for each value of η. Each image will be built of 3 pieces. See Figure 1.4.

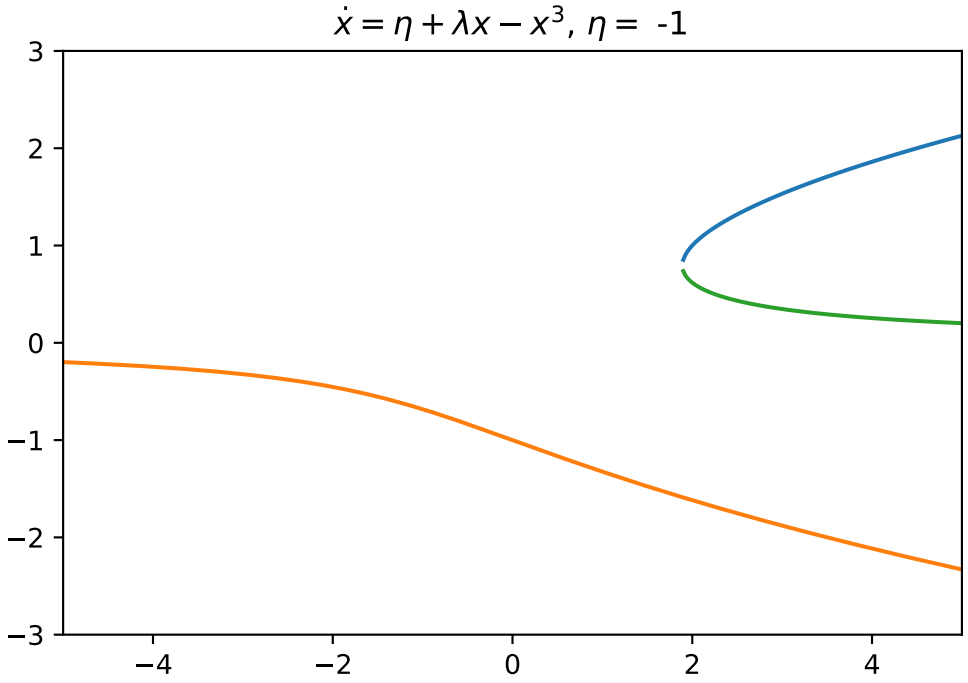


Figure 1.4: Bifurcation diagram for the equation $\dot{x} = \eta + \lambda x - x^3$ with $\eta = -1$.

Hysteresis

The following ODE exhibits an interesting bifurcation phenomenon called hysteresis:

$$x' = \lambda + x - x^3.$$

This system has a bifurcation diagram containing what is known as a hysteresis loop, shown in Figure 1.5. In the hysteresis loop, when the parameter λ moves beyond the bifurcation point the equilibrium solution makes a sudden jump to the other stable branch. When this occurs the system cannot reach its previous equilibrium by simply rewinding the parameter slightly. The next section discusses a model with a hysteresis loop.

Budworm Population Dynamics

Here we study a mathematical model describing the population dynamics of an insect called the spruce budworm. In eastern Canada, an outbreak in the budworm population can destroy most of the trees in a forest of balsam fir trees in about 4 years. The mathematical model is given by

$$\dot{N} = RN\left(1 - \frac{N}{K}\right) - p(N). \tag{1.4}$$

This model was studied by Ludwig et al (1978), and is described well in Strogatz's text *Nonlinear Dynamics and Chaos*. Here $N(t)$ represents the budworm population at time t, R is the growth rate of the budworm population and K represents the carrying capacity of the environment. We

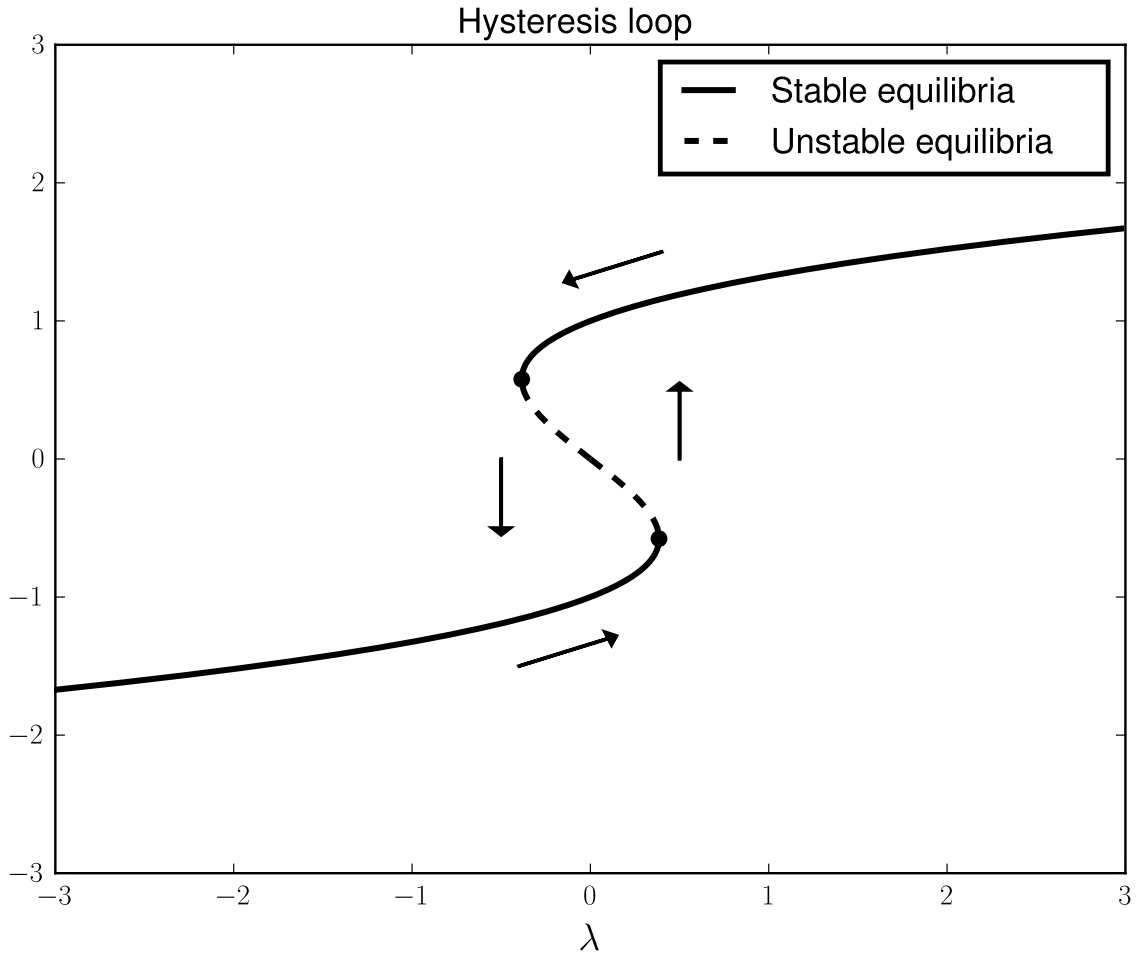


Figure 1.5: Bifurcation diagram for the ODE $x' = \lambda + x - x^3$.

could interpret K to represent the amount of food available to the budworms. $p(N)$ represents the death rate of budworms due to predators (birds); we assume specifically that $p(N)$ has the form $P(N) = \frac{BN^2}{A^2+N^2}$.

Before studying the equilibrium points of (1.4) it is important to reduce the number of parameters in the system by nondimensionalizing. Thus, we make the coordinate change $x = N/A$, $\tau = Bt/A$, $r = RA/B$, and $k = K/A$, obtaining finally the system

$$\frac{dx}{d\tau} = rx(1 - x/k) - \frac{x^2}{1 + x^2}. \tag{1.5}$$

Note that $x = 0$ is always an equilibrium solution. To find other equilibrium solutions we study the equation $r(1-x/k)-x/(1+x^2) = 0$. Fix $r = .56$, and consider Figure (1.6) ($k = 8$ in the figure).

Problem 3 (Budworm Population). Reproduce the bifurcation diagram for the differential

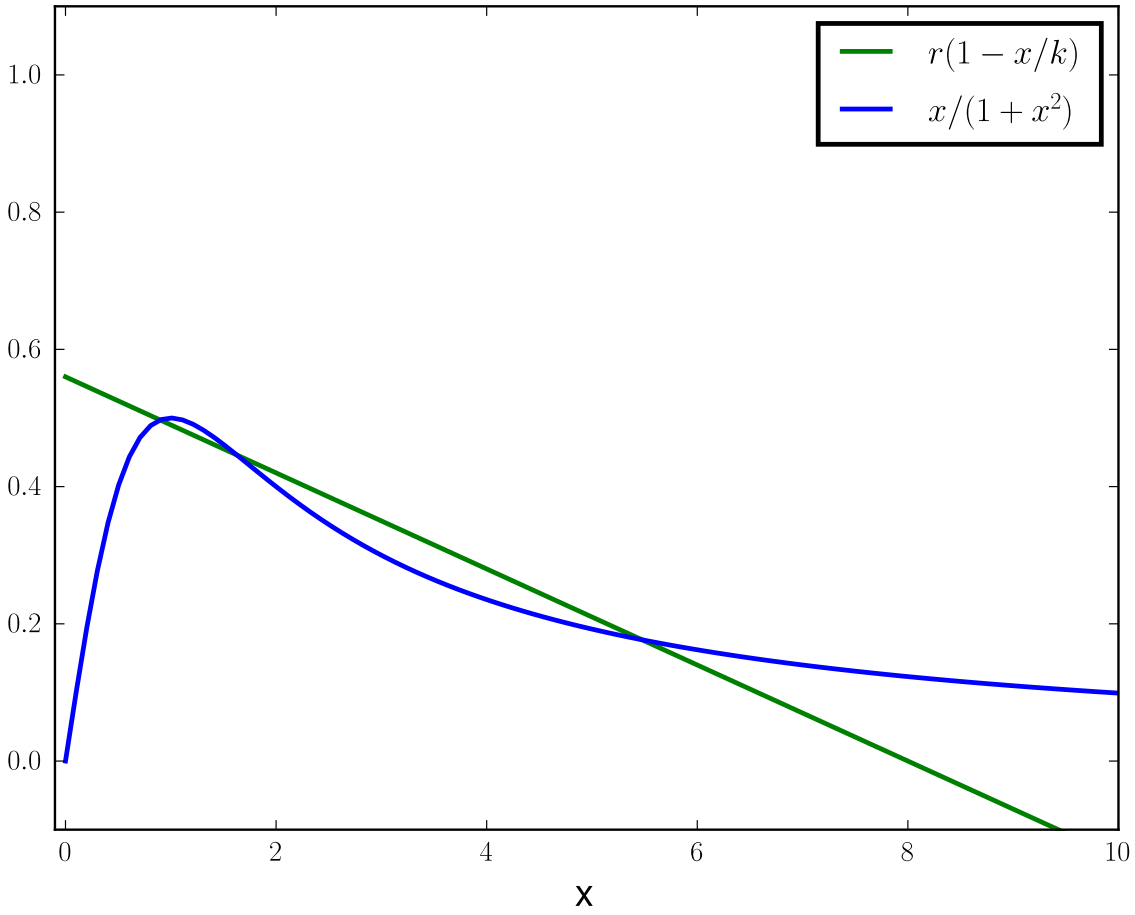


Figure 1.6: Graphical demonstration of nonzero equilibrium solutions for the budworm population (here $r = .56,\ k = 8$); equilibrium solutions occur where the curves cross. As k increases, the line $y = r(1 - x/k)$ gets more shallow and the number of solutions goes from one to three and then back to one.

equation

$$\frac{dx}{d\tau} = rx(1 - x/k) - \frac{x^2}{1 + x^2},$$

where $r = 0.56$.

Hint: Find a value for k that you know is in the middle of the plot (i.e. where there are three possible solutions), then use the code above to expand along each contour till you obtain the desired curve. Now find the proper initial guesses that give you the right bifurcation curve. The final plot will look like the one in Figure 1.7, but you will have to run the embedding algorithm 4-6 times to get every part of the plot.

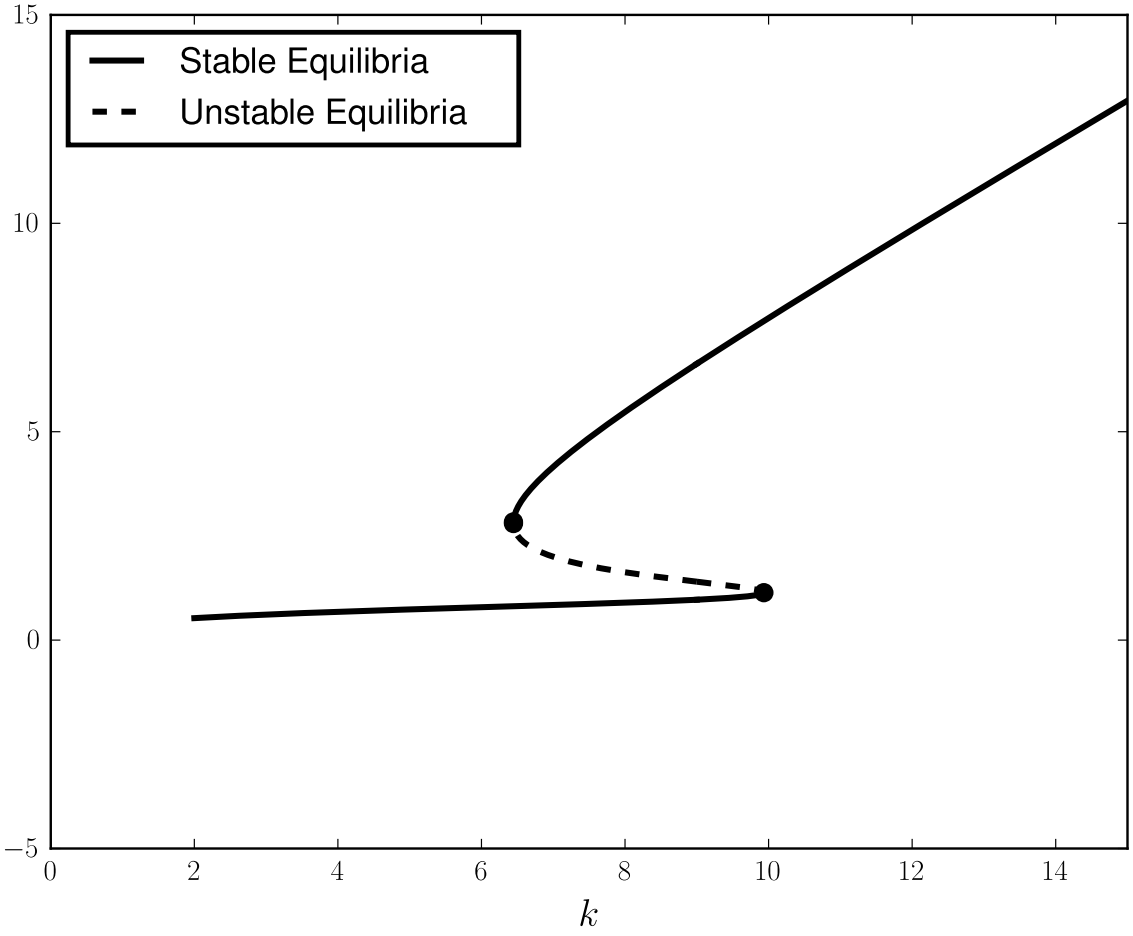


Figure 1.7: Bifurcation diagram for the budworm population model. The parameter r is fixed at 0.56. The lower stable branch is known as the refuge level of the bundworm population, while the upper stable branch is known as the outbreak level. Once the budworm population reaches an outbreak level, the available food (foliage of the balsam fir trees) in the system must be reduced drastically to jump back down to refuge level. Thus many of the balsam fir trees die before the budworm population returns to refuge level.

1 The Finite Difference Method

Lab Objective: *The finite difference method provides a solid foundation for solving partial differential equations. Understanding and applying finite difference is key to understanding numerical solutions to PDEs.*

A **finite difference** for a function $f(x)$ is an expression of the form $f(x+s) - f(x+t)$. Finite differences can give a good approximation of derivatives.

Suppose we have a function $u(x)$, defined on an interval $[a, b]$. Let $a = x_0, x_1, \dots x_{n-1}, x_n = b$ be a grid of $n+1$ evenly spaced points, with $x_{i+1} - x_i = h$, where $h = (b-a)/n$.

You are used to seeing the derivative $u'(x)$, which can written in centered-difference form as:

$$u'(x) = \lim_{h \to \infty} \frac{u(x+h) - u(x-h)}{2h}.$$

Suppose we are interested in knowing the value of the derivative at the points $\{x_i\}$. Even if we don't have a formula for $u'(x)$, we can approximate it using finite differences. We first write the Taylor polynomial expansion of $u(x+h)$ and $u(x-h)$ centered at x. This gives

$$u(x+h) = u(x) + u'(x)h + \frac{1}{2}u''(x)h^2 + \frac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \tag{1.1}$$

$$u(x-h) = u(x) - u'(x)h + \frac{1}{2}u''(x)h^2 - \frac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \tag{1.2}$$

Subtracting (1.2) from (1.1) and rearranging gives

$$u'(x) = \frac{u(x+h) - u(x-h)}{2h} + \mathcal{O}(h^2).$$

In terms of our grid points $\{x_i\}$, we have:

$$u'(x_i) \approx \frac{u(x_i+h) - u(x_i-h)}{2h} = \frac{u(x_{i+1}) - u(x_{i-1})}{2h}.$$

We won't worry about the derivative at the endpoints, $u'(x_0)$ and $u'(x_n)$. This allows us to approximate the values $\{u'(x_i)\}$ as the solution to a system of equations:

$$\frac{1}{2h}\underbrace{\begin{bmatrix} -1 & 0 & 1 & & & \\ & -1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 0 & 1 \end{bmatrix}}_{(n-1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-1}) \\ u(x_n) \end{bmatrix}}_{(n+1)\times 1} \approx \underbrace{\begin{bmatrix} u'(x_1) \\ u'(x_2) \\ \vdots \\ u'(x_{n-2}) \\ u'(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1}. \tag{1.3}$$

This can be rewritten with a $(n-1)\times(n-1)$ tridiagonal matrix instead:

$$\frac{1}{2h}\underbrace{\begin{bmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} u(x_1) \\ u(x_2) \\ \vdots \\ u(x_{n-2}) \\ u(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1} + \underbrace{\begin{bmatrix} -u(x_0)/(2h) \\ 0 \\ \vdots \\ 0 \\ u(x_n)/(2h) \end{bmatrix}}_{(n-1)\times 1} \approx \underbrace{\begin{bmatrix} u'(x_1) \\ u'(x_2) \\ \vdots \\ u'(x_{n-2}) \\ u'(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1}. \tag{1.4}$$

Next, we will consider the approximation for $u''(x)$. If we let

$$u'(x) \approx \frac{u(x+\frac{h}{2}) - u(x-\frac{h}{2})}{h}$$

then

$$u''(x) \approx \frac{u'(x+\frac{h}{2}) - u'(x-\frac{h}{2})}{h} \approx \frac{\frac{u((x+\frac{h}{2})+\frac{h}{2}) - u((x+\frac{h}{2})-\frac{h}{2})}{h} - \frac{u((x-\frac{h}{2})+\frac{h}{2}) - u((x-\frac{h}{2})-\frac{h}{2})}{h}}{h}$$
$$= \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

You can achieve the same result by again consider the Taylor polynomial expansion and adding (1.1) and (1.2) and rearranging. Thus

$$u''(x_i) \approx \frac{u(x_i+h) - 2u(x_i) + u(x_i-h)}{h^2} = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2}$$

Again ignoring the second derivative at the endpoints, this can be written in matrix form as

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \end{bmatrix}}_{(n-1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-1}) \\ u(x_n) \end{bmatrix}}_{(n+1)\times 1} \approx \underbrace{\begin{bmatrix} u''(x_1) \\ u''(x_2) \\ \vdots \\ u''(x_{n-2}) \\ u''(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1}. \tag{1.5}$$

This can also be written with a $(n-1)\times(n-1)$ tridiagonal matrix:

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} u(x_1) \\ u(x_2) \\ \vdots \\ u(x_{n-2}) \\ u(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1} + \underbrace{\begin{bmatrix} u(x_0)/h^2 \\ 0 \\ \vdots \\ 0 \\ u(x_n)/h^2 \end{bmatrix}}_{(n-1)\times 1} = \underbrace{\begin{bmatrix} u''(x_1) \\ u''(x_2) \\ \vdots \\ u''(x_{n-2}) \\ u''(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1} \tag{1.6}$$

Problem 1. Let $u(x) = \sin((x+\pi)^2 - 1)$. Use (1.3) - (1.6) to approximate $\frac{1}{2}u'' - u'$ at the grid points where $a = 0$, $b = 1$, and $n = 10$. Graph the result.

The previous equations are not only useful for approximating derivatives, but they can be also used to solve differential equations. Suppose that instead of knowing the function $u(x)$, we know that $\frac{1}{2}u'' - u' = f$, where the function $f(x)$ is given. How do we solve for $u(x)$?

Finite Difference Methods

Numerical methods for differential equations seek to approximate the exact solution $u(x)$ at some finite collection of points in the domain of the problem. Instead of analytically solving the original differential equation, defined over an infinite-dimensional function space, they use a well-chosen finite system of algebraic equations to approximate the original problem.

Consider the following differential equation:

$$\begin{aligned} &\varepsilon u''(x) - u(x)' = f(x), \quad x \in (0,1), \\ &u(0) = \alpha, \quad u(1) = \beta. \end{aligned} \tag{1.7}$$

Equation (1.7) can be written $Du = f$, where $D = \varepsilon \frac{d^2}{dx^2} - \frac{d}{dx}$ is a differential operator defined on the infinite-dimensional space of functions that are twice continuously differentiable on $[0,1]$ and satisfy $u(0) = \alpha$, $u(1) = \beta$.

We look for an approximate solution $\{U_i\}$, where

$$U_i \approx u(x_i)$$

on an evenly spaced grid of points, $a = x_0, x_1, \ldots, x_n = b,$. Our finite difference method will replace the differential operator $D = \varepsilon \frac{d^2}{dx^2} - \frac{d}{dx}$, (which is defined on an infinite-dimensional space), with finite difference operators (defined on a finite dimensional space). To do this, we replace derivative terms in the differential equation with appropriate difference expressions.

Recalling that

$$\begin{aligned} \frac{d^2}{dx^2}u(x_i) &= \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2} + \mathcal{O}(h^2), \\ \frac{d}{dx}u(x_i) &= \frac{u(x_{i+1}) - u(x_{i-1})}{2h} + \mathcal{O}(h^2). \end{aligned}$$

we define the finite difference operator D_h by

$$D_h U_i = \varepsilon \frac{1}{h^2}\left(U_{i+1} - 2U_i + U_{i-1}\right) - \frac{1}{2h}\left(U_{i+1} - U_{i-1}\right). \tag{1.8}$$

Thus we discretize equation (1.7) using the equations

$$\frac{\varepsilon}{h^2}(U_{i+1} - 2U_i + U_{i-1}) - \frac{1}{2h}\left(U_{i+1} - U_{i-1}\right) = f(x_i), \quad i = 1, \ldots, n-1,$$

along with boundary conditions $U_0 = \alpha$, $U_n = \beta$.

This gives $n+1$ equations and $n+1$ unknowns, and can be written in matrix form as

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} h^2 & 0 & 0 & \dots & 0 \\ (\varepsilon+h/2) & -2\varepsilon & (\varepsilon-h/2) & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & (\varepsilon+h/2) & -2\varepsilon & (\varepsilon-h/2) \\ 0 & \dots & & 0 & h^2 \end{bmatrix}}_{(n+1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_{n-1} \\ U_n \end{bmatrix}}_{(n+1)\times 1} = \underbrace{\begin{bmatrix} \alpha \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \\ \beta \end{bmatrix}}_{(n+1)\times 1}.$$

As before, we can remove two equations to modify the system to obtain an $(n-1)\times(n-1)$ tridiagonal system:

$$\begin{aligned} \frac{1}{h^2}\underbrace{\begin{bmatrix} -2\varepsilon & (\varepsilon-h/2) & 0 & \dots & 0 \\ (\varepsilon+h/2) & -2\varepsilon & (\varepsilon-h/2) & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & (\varepsilon+h/2) & -2\varepsilon & (\varepsilon-h/2) \\ 0 & \dots & & (\varepsilon+h/2) & -2\varepsilon \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_{n-2} \\ U_{n-1} \end{bmatrix}}_{(n-1)\times 1} \\ = \underbrace{\begin{bmatrix} f(x_1)-\alpha(\varepsilon+h/2)/h^2 \\ f(x_2) \\ \vdots \\ f(x_{n-2}) \\ f(x_{n-1})-\beta(\varepsilon-h/2)/h^2 \end{bmatrix}}_{(n-1)\times 1}. \end{aligned} \tag{1.9}$$

Problem 2. Use equation (1.9) to solve the singularly perturbed BVP (1.7) on the interval $[0,1]$ with $\varepsilon = 1/10$, $f(x) = -1$, $\alpha = 1$, and $\beta = 3$ on a grid with $n = 30$ subintervals. Graph the solution. This BVP is called singularly perturbed because of the location of the parameter ε. For $\varepsilon = 0$ the ODE has a drastically different character - it then becomes first order, and can no longer support two boundary conditions.

A heuristic test for convergence

The finite differences used above are second order approximations of the first and second derivatives of a function. It seems reasonable to expect that the numerical solution would converge at a rate of about $\mathcal{O}(h^2)$. How can we check that a numerical approximation is reasonable?

Suppose a finite difference method is $\mathcal{O}(h^p)$ accurate. This means that the error $E(h) \approx Ch^p$ for some constant C as $h \to 0$ (in other words, for $h > 0$ small enough).

So compute the approximation y_k for each stepsize h_k, $h_1 > h_2 > \ldots > h_m$. y_m should be the most accurate approximation, and will be thought of as the true solution. Then the error of the approximation for stepsize $h_k, k < m$, is

$$\begin{aligned} E(h_k) &= \max(|y_k - y_m|) \approx Ch_k^p, \\ \log(E(h_k)) &= \log(C) + p\log(h_k). \end{aligned}$$

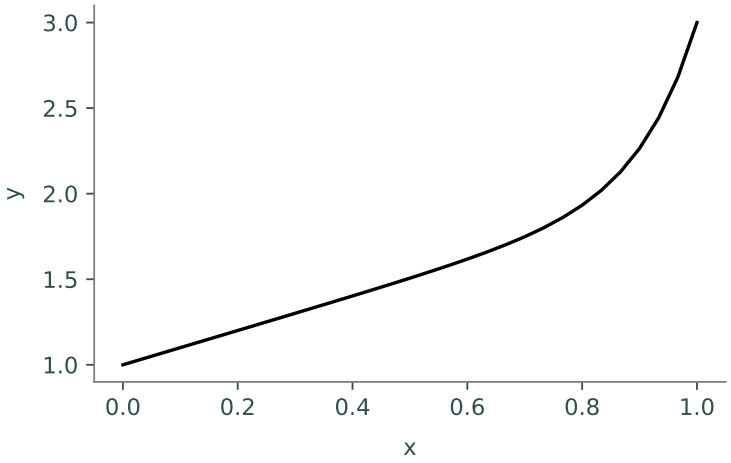


Figure 1.1: The solution to Problem 2. The solution gets steeper near $x = 1$ as ε gets small.

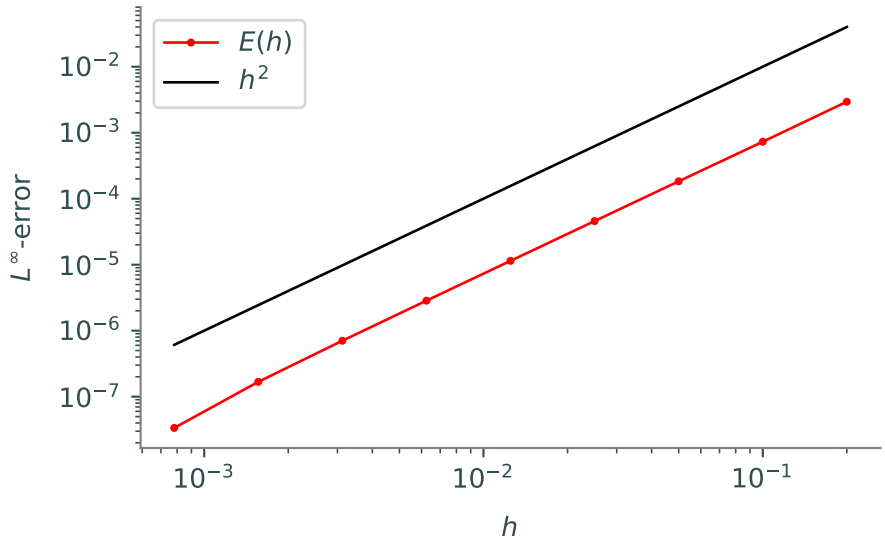


Figure 1.2: Demonstration of second order convergence for the finite difference approximation (1.8) of the BVP given in (1.7) with $\varepsilon = .5$.

Thus on a log-log plot of $E(h)$ vs. h, these values should be on a straight line with slope p when h is small enough to start getting convergence. We should note that demonstrating second-order convergence does NOT imply that the numerical approximation is converging to the correct solution.

Problem 3. Implement a function `singular_bvp` to compute the finite difference solution to 1.7. Using $n = 5 \times 2^0, 5 \times 2^1, \ldots, 5 \times 2^9$ subintervals, compute 10 approximate solutions. Use these to visualize the $\mathcal{O}(h^2)$ convergence of the finite difference method from Problem 2 by producing a loglog plot of error against subinterval count; this will be similar to Figure 1.2,

except with $\varepsilon = 0.1$.

To produce the plot, treat the approximation with $n = 5 \times 2^9$ subintervals as the "true solution", and measure the error for the other approximations against it. Note that, since the ratios of numbers of subintervals between approximations are multiples of 2, we can compute the L_∞ error for the $n = 5 \times 2^j$ approximation by using the `step` argument in the array slicing syntax:

```
# best approximation; the vector has length 5*2^9+1
sol_best = singular_bvp(eps,alpha,beta,f,5*(2**9))

# approximation with 5*(2^j) intervals; the vector has length 5*2^j+1
sol_approx = singular_bvp(eps,alpha,beta,f,5*(2**j))

# approximation error; slicing results in a vector of length 5*2^j+1,
#    which allows it to be compared
error = np.max(np.abs(sol_approx - sol_best[::2**(9-j)]))
```

Problem 4. Extend your finite difference code to the case of a general second order linear BVP with boundary conditions:

$$a_1(x)y''(x) + a_2(x)y'(x) + a_3(x)y(x) = f(x), \quad x \in (a, b),$$
$$y(a) = \alpha, \quad y(b) = \beta.$$

Use your code to solve the boundary value problem

$$\varepsilon y'' - 4(\pi - x^2)y = \cos x,$$
$$y(0) = 0, \quad y(\pi/2) = 1,$$

for $\varepsilon = 0.1$ on a grid with $n = 30$ subintervals. Be sure to modify the finite difference operator D_h in (1.8) correctly.

The next few problems will help you test your finite difference code.

Problem 5. Numerically solve the boundary value problem

$$\varepsilon y''(x) + xy'(x) = -\varepsilon\pi^2 \cos(\pi x) - \pi x \sin(\pi x),$$
$$y(-1) = -2, \quad y(1) = 0,$$

for $\varepsilon = 0.1, 0.01$, and 0.001. Use a grid with $n = 150$ subintervals. Plot your solutions.

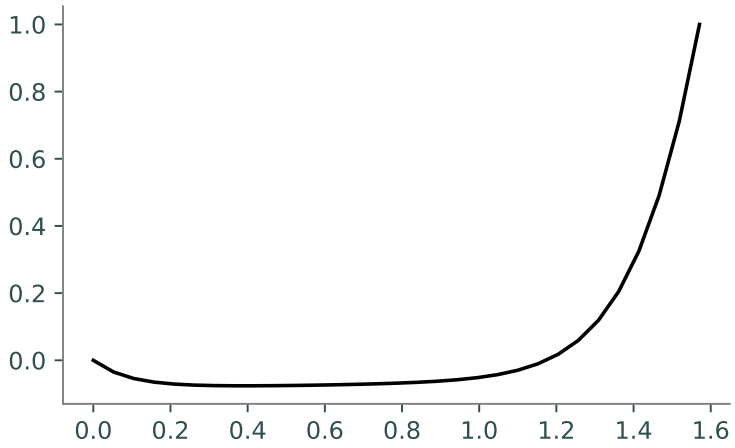


Figure 1.3: The solution to Problem 4.

Problem 6. Numerically solve the boundary value problem

$$(\varepsilon + x^2)y''(x) + 4xy'(x) + 2y(x) = 0,$$
$$y(-1) = 1/(1+\varepsilon), \quad y(1) = 1/(1+\varepsilon),$$

for $\varepsilon = 0.05, 0.02$. Use a grid with $n = 150$ subintervals. Plot your solutions.

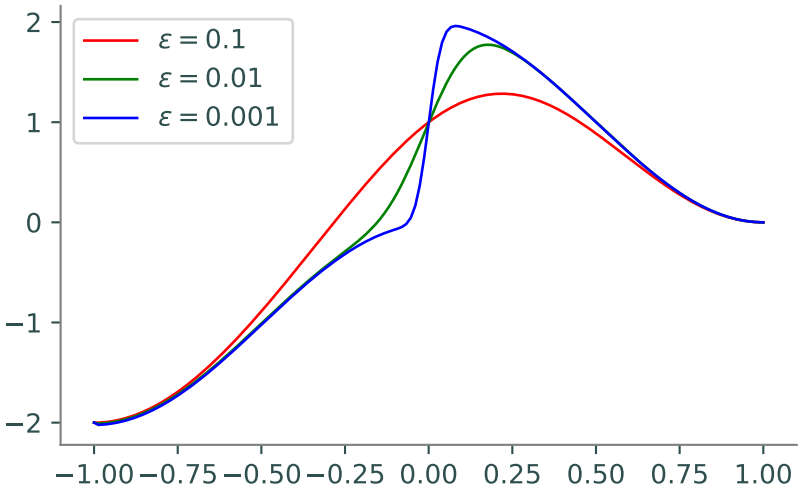


Figure 1.4: The solution to Problem 5.

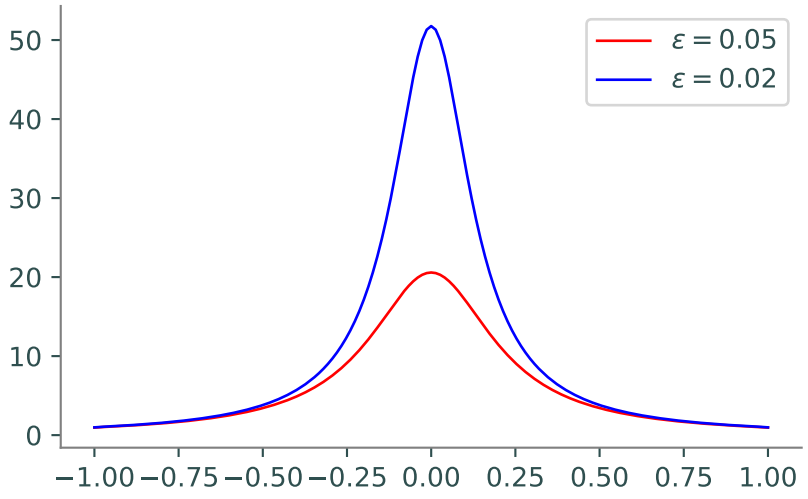


Figure 1.5: The solution to Problem 6.

1 Wave Phenomena

Advection Equation

The advection equation (or transport equation) is given by $u_t + su_x = 0$, where s is a nonzero constant. Consider the Cauchy problem

$$\begin{aligned} &u_t + su_x = 0, \quad -\infty < x < \infty, \\ &u(x,0) = f(x). \end{aligned}$$

The function $f(x)$ may be thought of as an initial wave or signal. The general solution of this initial boundary value problem is $u(x,t) = f(x-st)$ (check this!). The solution $u(x,t)$ is a travelling wave that takes the signal $f(x)$ and moves it along at a constant speed s - to the right if $s > 0$, and to the left if $s < 0$.

Wave Equation

Many different wave phenomena can be described using a hyperbolic PDE called the wave equation. These wave phenomena occur in fields such as electromagnetics, fluid dynamics, and acoustics. This equation is given by

$$u_{tt} = s^2 \triangle u. \tag{1.1}$$

The 1D equation can be derived in the context of many physical models; a common derivation describes the motion of a string vibrating in a plane. Another nice derivation uses Hooke's law from the theory of elasticity.

After making the change of variables $(\xi, \eta) = (x - st, x + st)$ and using the chain rule, we find that the 1D wave equation $u_{tt} = s^2 u_{xx}$ is equivalent to $u_{\xi\eta} = 0$. The general solution of this last equation is

$$u(\xi, \eta) = F(\xi) + G(\eta)$$

for some scalar functions F and G. In (x,t) coordinates the solution is

$$u(x,t) = F(x - st) + G(x + st)$$

Thus the general solution of the wave equation is the sum of two parts: one is a signal travelling to the right with constant speed $|s|$, and the other is a signal travelling to the left with speed $|s|$.

The wave equation is usually seen in the context of an initial boundary value problem. This takes the form

$$\begin{aligned} u_{tt} &= s^2 u_{xx}, \quad 0 < x < l, \quad t > 0, \\ u(0,t) &= u(l,t) = 0, \\ u(x,0) &= f(x), \\ u_t(x,0) &= g(x). \end{aligned}$$

Numerical solution of the wave equation

We look to approximate $u(x,t)$ on a grid of points $(x_j, t_m)_{j=0,m=0}^{J,M}$. Denote the approximation to $u(x_j,t_m)$ by U_j^m. Recall that the centered approximations in space and time are

$$\begin{aligned} D_{tt}U_j^m &= \frac{U_j^{m+1} - 2U_j^m + U_j^{m-1}}{(\triangle t)^2}, \\ D_{xx}U_j^m &= \frac{U_{j+1}^m - 2U_j^m + U_{j-1}^m}{(\triangle x)^2}. \end{aligned}$$

The resulting method is given by

$$\begin{aligned} \frac{U_j^{m+1} - 2U_j^m + U_j^{m-1}}{(\triangle t)^2} &= s^2 \frac{U_{j+1}^m - 2U_j^m + U_{j-1}^m}{(\triangle x)^2}, \\ U_j^{m+1} &= -U_j^{m-1} + 2(1-\lambda^2)U_j^m + \lambda^2(U_{j+1}^m + U_{j-1}^m), \end{aligned}$$

where $\lambda = s(\triangle t)/(\triangle x)$. This method may be written in matrix form as

$$U^{m+1} = AU^m - U^{m-1}$$

where

$$A = \begin{bmatrix} 2(1-\lambda^2) & \lambda^2 & & & \\ \lambda^2 & 2(1-\lambda^2) & \lambda^2 & & \\ \ddots & \ddots & \ddots & & \\ & \lambda^2 & 2(1-\lambda^2) & \lambda^2 & \\ & & \lambda^2 & 2(1-\lambda^2) & \end{bmatrix}$$

and

$$U^m = \begin{bmatrix} U_1^m \\ U_2^m \\ \vdots \\ U_{J-1}^m \end{bmatrix}$$

In the matrix equation above, we have already used the boundary conditions to determine that $U_0^m = U_J^m = 0$ at each time t_m. Note that, to obtain the approximation U_j^{m+1} of $u(x_j, t_{m+1})$, the method uses the value of the approximation at *the previous two time steps*. We can find the solution for the first two time steps by using the initial conditions. Using the initial conditions directly gives an approximation at $t = t_0 = 0$:

$$U_j^0 = f(x_j), \quad 1 \le j \le J-1$$

To obtain an approximation at the second time step, we consider the Taylor expansion

$$u(x_j, t_1) = u(x_j, 0) + u_t(x_j, 0)\triangle t + u_{tt}(x_j, 0)\frac{\triangle t^2}{2} + u_{ttt}(x_j, t_1^*)\frac{\triangle t^3}{6}.$$

Recalling that the solution $u(x, t)$ satisfies the wave equation, we substitute in expressions from our initial conditions:

$$u(x_j, t_1) = u(x_j, 0) + g(x_j)\triangle t + s^2 f''(x_j)\frac{\triangle t^2}{2} + u_{ttt}(x_j, t_1^*)\frac{\triangle t^3}{6}.$$

Ignoring the third order term, we obtain a second order approximation for the second time step:

$$U_j^1 = U_j^0 + g(x_j)\triangle t + s^2 f''(x_j)\frac{\triangle t^2}{2}, \quad 1 \le j \le J-1$$

or if f is not readily differentiable,

$$U_j^1 = U_j^0 + g(x_j)\triangle t + \frac{\lambda^2}{2}(U_{j-1}^0 - 2U_j^0 + U_{j+1}^0)$$

This method is conditionally stable; the CFL condition is that $\lambda \le 1$.

Problem 1. Consider the initial boundary value problem

$$\begin{aligned} u_{tt} &= u_{xx}, \\ u(0,t) &= u(1,t) = 0, \\ u(x,0) &= \sin(2\pi x), \\ u_t(x,0) &= 0. \end{aligned}$$

Numerically approximate the solution $u(x, t)$ for $t \in [0, .5]$. Use $J = 50$ subintervals in the x dimension and $M = 50$ subintervals in the t dimension. Animate the results. Compare your results with the analytic solution $u(x, t) = \sin(2\pi x)\cos(2\pi t)$. This function is known as a standing wave. See Figure 1.1.

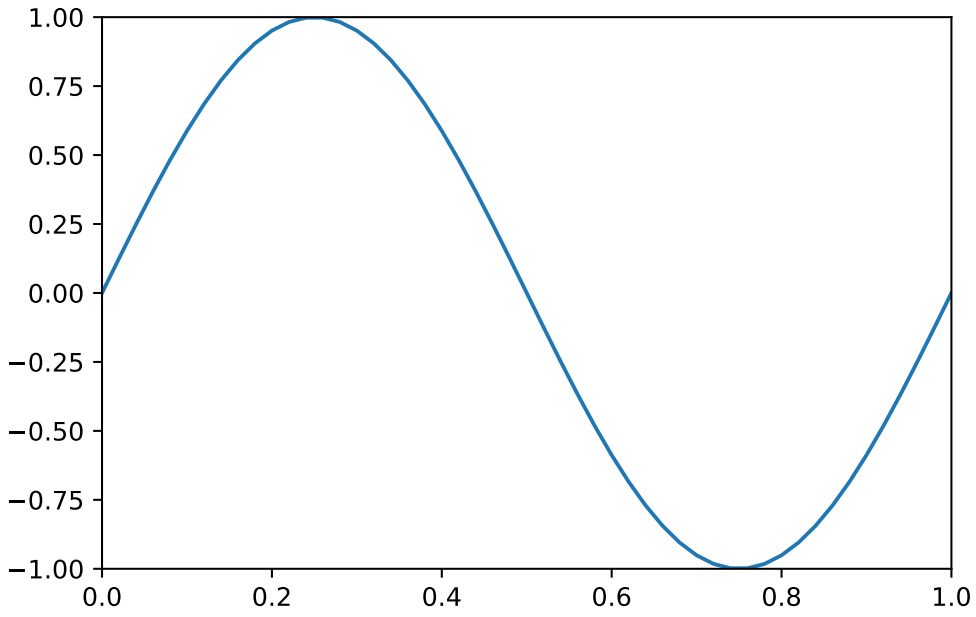


Figure 1.1: $u(x, t = 0)$.

Problem 2. Consider the initial boundary value problem

$$\begin{aligned} u_{tt} &= u_{xx}, \\ u(0,t) &= u(1,t) = 0, \\ u(x,0) &= .2e^{-m^2(x-1/2)^2} \\ u_t(x,0) &= .4m^2(x-1/2)e^{-m^2(x-1/2)^2}. \end{aligned}$$

The solution of this problem is a Gaussian pulse. It travels to the right at a constant speed. This solution models, for example, a wave pulse in a stretched string. Note that the fixed boundary conditions reflect the pulse back when it meets the boundary.

Numerically approximate the solution $u(x,t)$ for $t \in [0,1]$. Set $m = 20$. Use 200 subintervals in space and 220 in time, and animate your results. Then use 200 subintervals in space and 180 in time, and animate your results. Note that the stability condition is not satisfied for the second mesh. See 1.2.

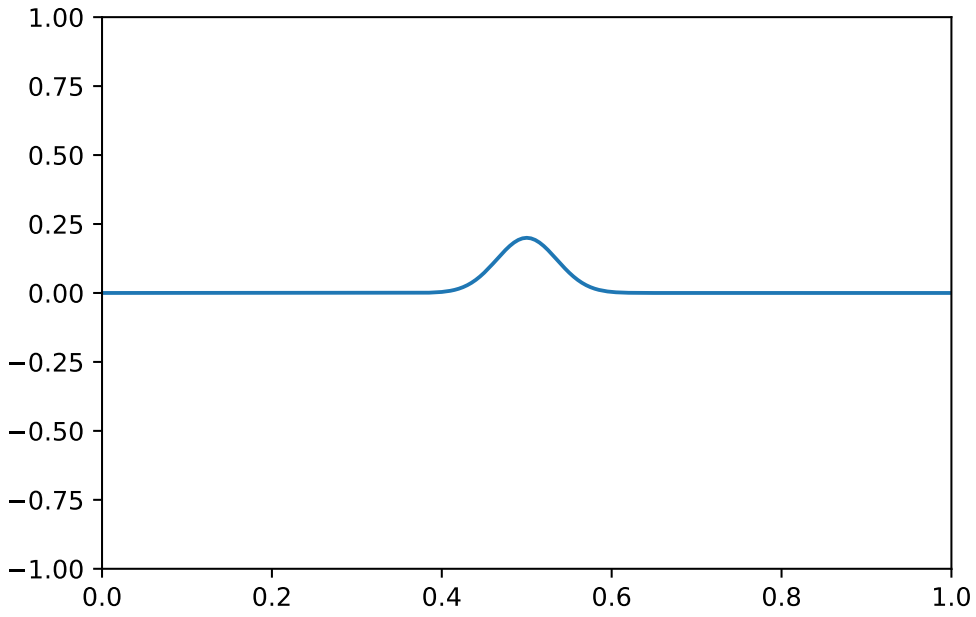


Figure 1.2: $u(x, t = 0)$.

Problem 3. Consider the initial boundary value problem

$$\begin{aligned} u_{tt} &= u_{xx}, \\ u(0,t) &= u(1,t) = 0, \\ u(x,0) &= .2e^{-m^2(x-1/2)^2} \\ u_t(x,0) &= 0. \end{aligned}$$

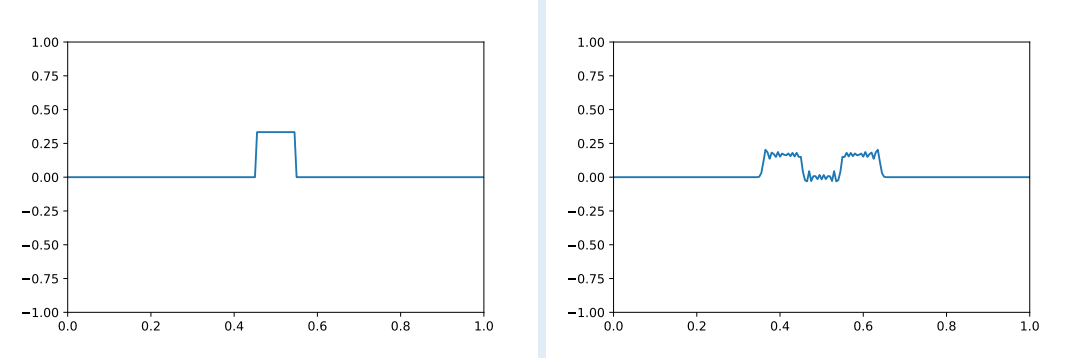
The initial condition separates into two smaller, slower-moving pulses, one travelling to the right and the other to the left. This solution models, for example, a plucked guitar string

Numerically approximate the solution $u(x,t)$ for $t \in [0,2]$. Set $m = 20$. Use 200 subintervals in space and 440 in time, and animate your results. It is rather easy to see that the solution to this problem is the sum of two travelling waves, one travelling to the left and the other to the right, as described earlier.

Problem 4. Consider the initial boundary value problem

$$\begin{aligned} u_{tt} &= u_{xx}, \\ u(0,t) &= u(1,t) = 0, \\ u(x,0) &= \begin{cases} 1/3 & \text{if } 5/11 < x < 6/11, \\ 0 & \text{otherwise} \end{cases} \\ u_t(x,0) &= 0. \end{aligned}$$

Numerically approximate the solution $u(x,t)$ for $t \in [0,2]$. Use 200 subintervals in space and 440 in time, and animate your results. Even though the method is second order and stable for this discretization, since the initial condition is discontinuous there are large dispersive errors. See Figure 1.3.



(a) $u(x, t = 0)$. (b) $u(x, t = .1)$.

Figure 1.3: The graphs for Problem 4 at various times t.

Travelling Wave Solutions of an Evolution Equation

Recall that the advection (transport) equation with initial conditions, given by

$$\begin{aligned} u_t + s u_x &= 0, \quad -\infty < x < \infty, \\ u(x,0) &= f(x), \end{aligned}$$

has as its general solution $u(x,t) = f(x - st)$. Consider a general evolutionary PDE of the form

$$u_t = G(u, u_x, u_{xx}, \ldots) \tag{1.2}$$

An interesting question to ask is whether (1.2) has travelling wave solutions: is there a signal or wave profile $f(x)$, so that $u(x,t) = f(x - st)$ is a solution of (1.2) that carries the signal at a constant speed s? These travelling waves are often significant physically. For example, in a PDE modeling insect population dynamics a travelling wave could represent a swarm of locusts; in a PDE describing a combustion process a travelling wave could represent an explosion or detonation.

Burgers' equation

We will examine the process of studying travelling wave solutions using Burgers' equation, a nonlinear PDE from gas dynamics. It is given by

$$u_t + \left(\frac{u^2}{2}\right)_x = \nu u_{xx}, \tag{1.3}$$

where u and ν represent the velocity and viscosity of the gas, respectively. It models both the process of transport with the nonlinear advection term $(u^2/2)_x = uu_x$, as well as diffusion due to the viscosity of the gas (νu_{xx}).

Let us look for a travelling wave solution $u(x,t) = \hat{u}(x-st)$ for Burgers equation. We transform (1.3) into the moving frame $(x,t) \to (\bar{x},\bar{t}) = (x-st,t)$. In this frame (1.3) becomes

$$u_{\bar{t}} - su_{\bar{x}} + \left(\frac{u^2}{2}\right)_{\bar{x}} = \nu u_{\bar{x}\bar{x}} \tag{1.4}$$

This new frame of reference corresponds to an observer moving along with the wave, so that the wave appears stationary as the observer studies it. Thus, $\hat{u}_{\bar{t}} = 0$, so that the wave profile $\hat{u}$ satisfies the ordinary differential equation

$$-su_{\bar{x}} + \left(\frac{u^2}{2}\right)_{\bar{x}} = \nu u_{\bar{x}\bar{x}}. \tag{1.5}$$

From here on we will drop the bar notation for simplicity. We seek a travelling wave solution with asymptotically constant boundary conditions; that is, $\lim_{x\to\pm\infty} \hat{u}(x) = u_\pm$

both exist, and $\lim_{x\to\pm\infty} \hat{u}'(x) = 0$. We will suppose that $u_- > u_+ > 0$.

Note that to this point we still don't know the speed of the travelling wave. Integrating both sides of this differential equation, and then taking the limit as $x \to +\infty$, we obtain

$$\begin{aligned}
-s\int_{-\infty}^{x} u' + \int_{-\infty}^{x}\left(\frac{u^2}{2}\right)' &= \nu\int_{-\infty}^{x} u'', \\
-s(u(x)-u_-) + \frac{u^2(x)}{2} - \frac{u_-^2}{2} &= \nu(u'(x) - u'(-\infty)), \\
-s(u_+ - u_-) + \frac{u_+^2}{2} - \frac{u_-^2}{2} &= 0.
\end{aligned}$$

Thus given boundary conditions $u_\pm$ at $\pm\infty$, the speed of the travelling wave must be $s = \frac{u_- + u_+}{2}$.

Usually at this point, the travelling wave must be numerically solved using the profile ODE ((1.5) for Burgers equation). However, the profile ODE for Burgers is simple enough that it is possible to obtain an analytic solution. The travelling wave is given by

$$\hat{u}(x) = s - a\tanh\left(\frac{ax}{2\nu} + \delta\right)$$

where $a = (u_- - u_+)/2$ and δ is fixed real number. We get a family of solutions because any translation of a travelling wave solution is also a travelling wave solution.

Stability of travelling waves

Suppose that an evolutionary PDE

$$u_t = G(u, u_x, u_{xx}, \ldots). \tag{1.6}$$

has a travelling wave solution $u(x,t) = \hat{u}(x - st)$. An interesting question to consider is whether the mathematical solution, $\hat{u}$, has a physical analogue. In other words, does the travelling wave show up in real life? This question is the start of the mathematical study of stability of travelling waves.

We begin by translating (1.6) into the moving frame $(x,t) \to (\bar{x},\bar{t}) = (x - st, t)$. In this frame the PDE becomes

$$u_t - su_x = G(u, u_x, u_{xx}, \ldots).$$

In these coordinates the travelling wave is stationary. Thus, the solution of

$$\begin{aligned} u_t - su_x &= G(u, u_x, u_{xx}, \ldots), \\ u(x, t = 0) &= \hat{u}(x), \end{aligned}$$

is given by $u(x,t) = \hat{u}(x)$. We say that the travelling wave $\hat{u}$ is asymptotically orbitally stable if whenever $v(x)$ is a small perturbation of $\hat{u}(x)$, the general solution of

$$\begin{aligned} u_t - su_x &= G(u, u_x, u_{xx}, \ldots), \\ u(x, t = 0) &= v(x), \end{aligned}$$

converges to some translation of $\hat{u}$ as $t \to \infty$. Using this definition to prove stability of a travelling wave is a nontrivial task.

Visualizing stability of the travelling wave solution of Burgers' equation

The travelling wave solution of Burgers' equation is a stable wave. To view this numerically, we discretize the PDE

$$u_t - su_x + uu_x = u_{xx}$$

using the second order centered approximations

$$D_t U_j^{n+1/2} = \frac{U_j^{n+1} - U_j^n}{\triangle t}, \quad D_{xx} U_j^{n+1/2} = \frac{1}{2}\left(\frac{U_{j+1}^{n+1} - U_{j-1}^{n+1}}{2\triangle x} + \frac{U_{j+1}^n - U_{j-1}^n}{2\triangle x}\right),$$

$$D_{xx} U_j^{n+1/2} = \frac{1}{2}\left(\frac{U_{j+1}^{n+1} - U_j^{n+1} + U_{j-1}^{n+1}}{(\triangle x)^2} + \frac{U_{j+1}^n - U_j^n + U_{j-1}^n}{(\triangle x)^2}\right)$$

Substituting these expressions into the PDE we obtain a second-order, implicit Crank-Nicolson method

$$\begin{aligned} U_j^{n+1} - U_j^n = K_1\big[(s - U_j^{n+1})(U_{j+1}^{n+1} - U_{j-1}^{n+1}) + (s - U_j^n)(U_{j+1}^n - U_{j-1}^n)\big] \\ + K_2\big[(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}) + (U_{j+1}^n - 2U_j^n + U_{j-1}^n)\big], \end{aligned}$$

where $K_1 = \frac{\triangle t}{4\triangle x}$ and $K_2 = \frac{\triangle t}{2(\triangle x)^2}$.

Problem 5. Numerically solve the initial value problem

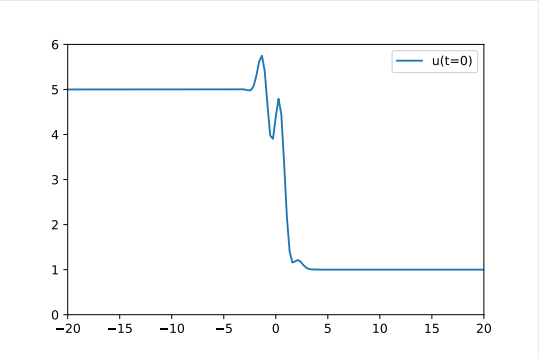
$$\begin{aligned} u_t - su_x + uu_x &= u_{xx}, \quad x \in (-\infty, \infty), \\ u(x, 0) &= v(x), \end{aligned}$$

for $t \in [0, 1]$. Let the perturbation $v(x)$ be given by

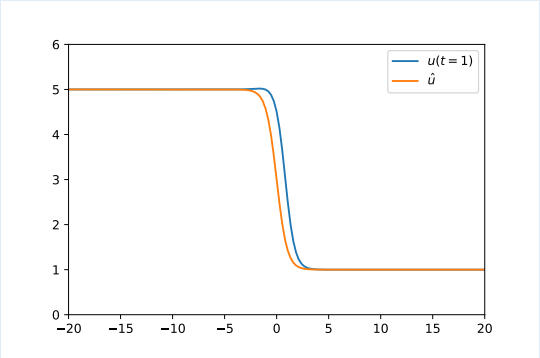
$$v(x) = 3.5(\sin(3x) + 1)\frac{1}{\sqrt{2\pi}} \exp(-x^2/2)$$

And let the initial condition be $u(x, 0) = \hat{u}(x) + v(x)$ Approximate the x domain,$(-\infty, \infty)$, numerically by the finite interval $[-20, 20]$, and fix $u(-20) = u_-$, $u(20) = u_+$. Let $u_- = 5$, $u_+ = 1$. Use 150 intervals in space and 350 steps in time. Animate your results. You should see the solution converge to a translate of the travelling wave $\hat{u}$. See Figure 1.4.

Hint: This difference scheme is no longer a linear equation. We have a nonlinear equation in U^{n+1}. We can still solve this function using Newton's method or some other similar solver. In this case, use `scipy.optimize.fsolve`.



(a) $u(x, t = 0)$.



(b) $u(x, t = 1)$ vs $\hat{u}$.

Figure 1.4: The graphs for Problem 5

1 Conservation laws and heat flow

A conservation law is a balance law, and corresponds to an equation that describes how a quantity is balanced in some system throughout a given process. (Consider how this is related to conservation laws in physics.) For example, suppose we are keeping track of some measurable quantity in a physical system (e.g. heat, water, etc). The fundamental conservation law then states that the rate of change of the total quantity in the system is equal to the rate of the quantity flowing into the system plus the rate at which the quantity is produced by sources inside the system.

Derivation of the Conservation equation in multiple dimensions

Suppose Ω is a region in $\mathbb{R}^n$, and $V \subset \Omega$ is bounded with a reasonably well-behaved boundary ∂V. Let $u(\vec{x}, t)$ represent the density (concentration) of some quantity throughout Ω. Let $\vec{n}(x)$ represent the normal direction to V at $x \in \partial V$, and let $\vec{J}(\vec{x}, t)$ be the flux vector for the quantity, so that $\vec{J}(\vec{x}, t) \cdot \vec{n}(x)\, dA$ represents the rate at which the quantity leaves V by crossing a boundary element with area dA. Note that the total amount of the quantity in V is

$$\int_V u(\vec{x}, t)\, dt,$$

and the rate at which the quantity enters V is

$$-\int_{\partial V} \vec{J}(\vec{x}, t) \cdot \vec{n}(x)\, dA.$$

We let the source term be given by $g(\vec{x}, t, u)$; we may interpret this to mean that the rate at which the quantity is produced in V is

$$\int_V g(\vec{x}, t, u)\, dt.$$

Then the integral form of the conservation law for u is expressed as

$$\frac{d}{dt}\int_V u(\vec{x}, t)\, d\vec{x} = -\int_{\partial V} \vec{J} \cdot \vec{n}\, dA + \int_V g(\vec{x}, t, u)\, d\vec{x}.$$

If u and J are sufficiently smooth functions, then we have

$$\frac{d}{dt}\int_V u\, d\vec{x} = \int_V u_t\, d\vec{x},$$

and

$$\int_{\partial V} \vec{J} \cdot \vec{n}\, dA = \int_V \nabla \cdot \vec{J}\, d\vec{x}.$$

Since this holds for all nice subsets $V \subset \Omega$ with V arbitrarily small, we obtain the differential form of the conservation law for u:

$$u_t + \nabla \cdot \vec{J} = g(\vec{x}, t, u),$$

where ∇ is the gradient function and $\nabla \cdot \vec{J} = \frac{\partial J_1}{\partial x_1} + \cdots + \frac{\partial J_n}{\partial x_n}$

Constitutive Relations

Currently our conservation law appears in the form

$$u_t + \nabla \cdot \vec{J} = g(\vec{x}, t, u).$$

Thus the conservation law consists of one equation and 2 unknowns (u and J). To this equation we add other equations, called constitutive relations, which are used to fully determine the system.

For example, suppose we wish to describe the flow of heat. Since heat flows from warmer regions to colder regions, and the rate of heat flow depends on the difference in temperature between regions, we usually assume that the flux vector $\vec{J}$ is given by

$$\vec{J}(x,t) = -\nu \nabla u(x,t),$$

where ν is a diffusion constant and $\nabla u(x,t) = [\partial_{x_1} u \dots \partial_{x_n} u]^T$. This constitutive relation is called Fick's law, and is the basic model for any diffusive process. Substituting into the conservation law we obtain

$$u_t - \nu \Delta u(x,t) = g(\vec{x}, t, u)$$

where Δ is the Laplacian operator, and $\Delta u(x,t) = \frac{\partial^2 u}{\partial x_1^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2}$. The function g represents heat sources/sinks within the region.

Numerically modeling heat flow

Consider the heat flow equation in one dimension together with appropriate initial conditions and homogeneous Dirichlet boundary conditions:

$$\begin{aligned} &u_t = \nu u_{xx}, \quad x \in [a,b], \quad t \in [0,T], \\ &u(a,t) = 0, \quad u(b,t) = 0, \\ &u(x,0) = f(x). \end{aligned}$$

We will look for an approximation U_i^j to $u(x_i, t_j)$ on the grid $x_i = a + hi$, $t_j = kj$, where h and k are small changes in x and t respectively and i and j are indices. Note that the index i ranges over different spacial grid points and the index j ranges over different time steps. We will denote the approximate value of u at the i'th grid point and the j'th time step as U_i^j.

A common method for modeling ordinary and partial differential equations is the finite difference method, so-named because equations containing derivatives are replaced with equations containing difference schemes. These difference schemes can often be found using Taylor's theorem. For example, the equation

$$u(x, t_j + k) = u(x, t_j) + u_t(x, t_j)k + \mathcal{O}(k^2)$$

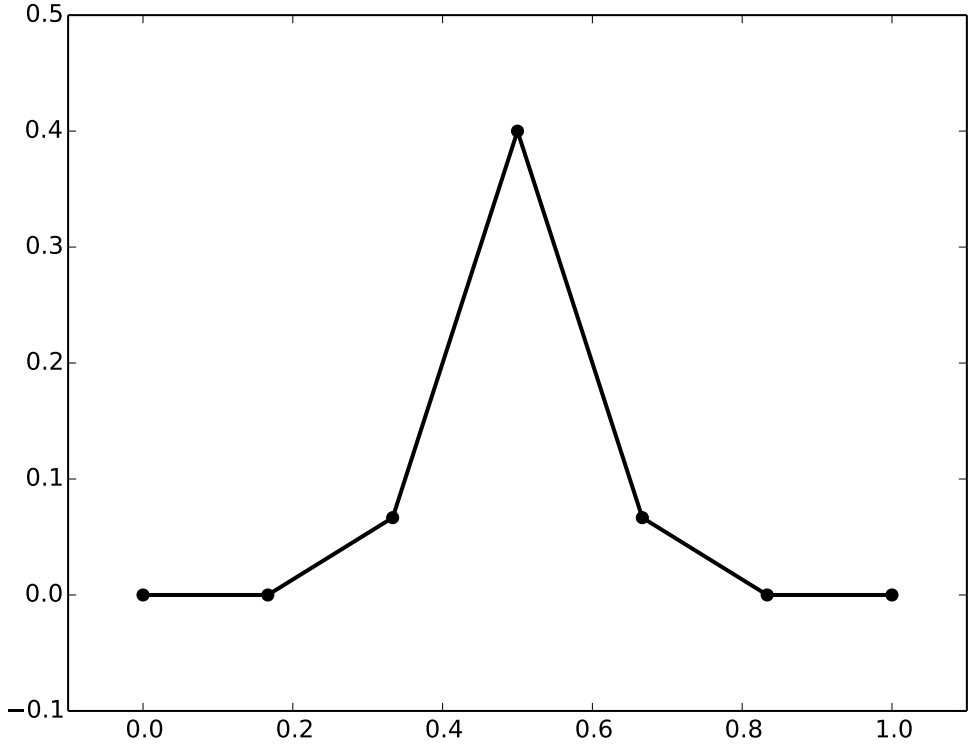


Figure 1.1: The graph of U^0, the approximation to the solution $u(x, t = 0)$ for Problem 1.

yields a first-order forward difference approximation to $u_t(x, t_j)$, namely,

$$u_t(x, t_j) = \frac{u(x, t_j + k) - u(x, t_j)}{k} + \mathcal{O}(k).$$

Similarly, by adding the equations

$$u(x_i + h, t) = u(x_i, t) + u_x(x_i, t)h + u_{xx}(x_i, t)\frac{h^2}{2} + u_{xxx}(x_i, t)h^3 + \mathcal{O}(h^4),$$
$$u(x_i - h, t) = u(x_i, t) + u_x(x_i, t)(-h) + u_{xx}(x_i, t)\frac{(-h)^2}{2} + u_{xxx}(x_i, t)(-h)^3 + \mathcal{O}(h^4),$$

we obtain a second-order centered difference approximation to $u_{xx}(x_i, t)$:

$$u_{xx}(x_i, t_j) = \frac{u(x_i + h, t_j) - 2u(x_i, t_j) - u(x_i - h, t_j)}{h^2} + \mathcal{O}(h^2).$$

These difference approximations give us the $\mathcal{O}(h^2 + k)$ explicit method

$$\begin{aligned} \frac{U_i^{j+1} - U_i^j}{k} &= \nu \frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2}, \\ U_i^{j+1} &= U_i^j + \frac{\nu k}{h^2}(U_{i+1}^j - 2U_i^j + U_{i-1}^j). \end{aligned} \tag{1.1}$$

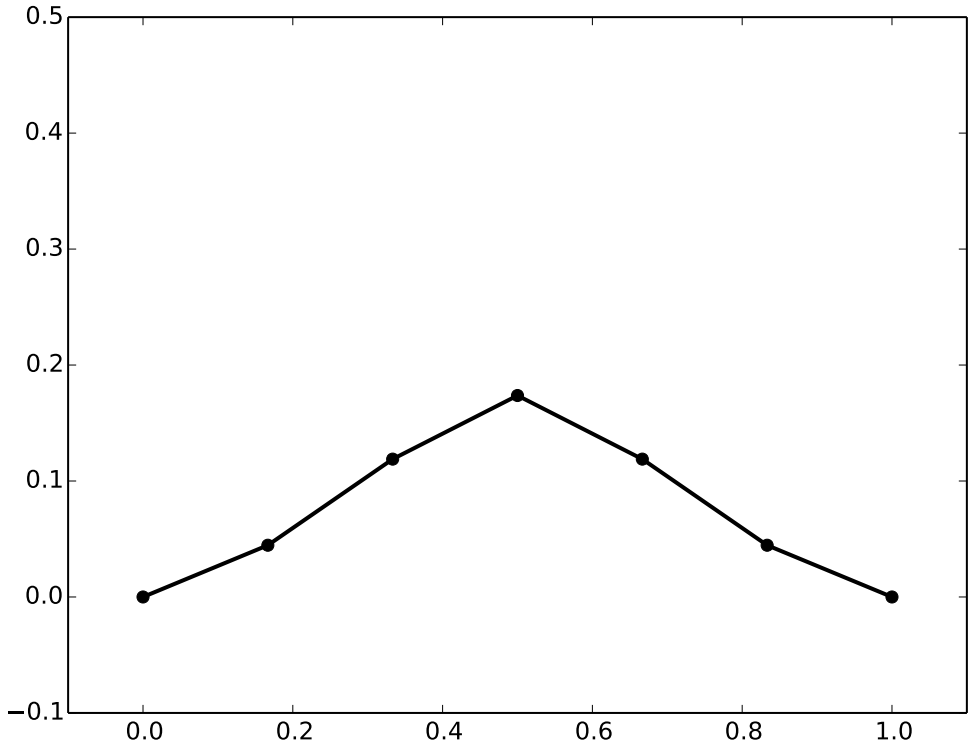


Figure 1.2: The graph of U^4, the approximation to the solution $u(x, t = .4)$ for Problem 1.

This method can be written in matrix form as

$$U^{j+1} = AU^j,$$

where A is the tridiagonal matrix given by

$$A = \begin{bmatrix} 1 & 0 & & & \\ \lambda & 1-2\lambda & \lambda & & \\ & \ddots & \ddots & \ddots & \\ & & \lambda & 1-2\lambda & \lambda \\ & & & 0 & 1 \end{bmatrix},$$

$\lambda = \nu k/h^2$, and U^j represents the approximation at time t_j. We can get this method started by using the initial condition given in our problem, so that $U_i^0 = f(x_i)$.

Note

Finite difference schemes, though they can be *represented* using matrix multiplication, should not be *implemented* using raw matrix multiplication. Using NumPy, it is best to vectorize the

difference scheme so that you do not have to loop over the spatial indices. If you are using a language with faster loops (like C, C++, Fortran, or Cython), it could work well to loop directly through the indices in both time and space.

To account for boundary conditions using this differencing scheme, simply set the boundary points to the appropriate values in the initial conditions, then avoid modifying them as you update for each time step. This would be the equivalent of replacing the first and last rows of the matrix representation of the differencing scheme with the first and last rows of the identity matrix.

Problem 1. Consider the initial/boundary value problem

$$\begin{aligned} &u_t = .05u_{xx}, \quad x \in [0,1], \quad t \in [0,1] \\ &u(0,t) = 0, \quad u(1,t) = 0, \\ &u(x,0) = 2\max\{.2 - |x - .5|, 0\}. \end{aligned} \tag{1.2}$$

Approximate the solution $u(x,t)$ at time $t = .4$ by taking 6 subintervals in the x dimension and 10 subintervals in time. The graphs for U^0 and U^4 are given in Figures 1.1 and 1.2.

Problem 2. Solve the initial/boundary value problem

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12,12], \quad t \in [0,1], \\ &u(-12,t) = 0, \quad u(12,t) = 0, \\ &u(x,0) = \max\{1 - x^2, 0\} \end{aligned} \tag{1.3}$$

using the first order explicit method 1.1. Use 140 subintervals in the x dimension and 70 subintervals in time. The initial and final states are shown in Figure 1.3. Animate your results.

Explicit methods usually have a stability condition, called a CFL condition (for Courant-Friedrichs-Lewy). For method 1.1 the CFL condition that must be satisfied is that

$$\lambda \leq \frac{1}{2}.$$

Repeat your computations using 140 subintervals in the x dimension and 66 subintervals in time. Animate the results. For these values the CFL condition is broken; you should easily see the result of this instability in the approximation U^{66}.

Implicit methods often have better stability properties than explicit methods. The Crank-Nicolson method, for example, is unconditionally stable and has order $\mathcal{O}(h^2 + k^2)$. To derive the Crank-Nicolson method, we use the following approximations:

$$\begin{aligned} u_t(x_i, t_{j+1/2}) &= \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{k} + \mathcal{O}(k^2), \\ u_{xx}(x_i, t_{j+1/2}) &= \frac{u_{xx}(x_i, t_{j+1}) + u_{xx}(x_i, t_j)}{2} + \mathcal{O}(k^2). \end{aligned}$$

The first equation is a Finite Difference approximation, and the second is a midpoint approximation.

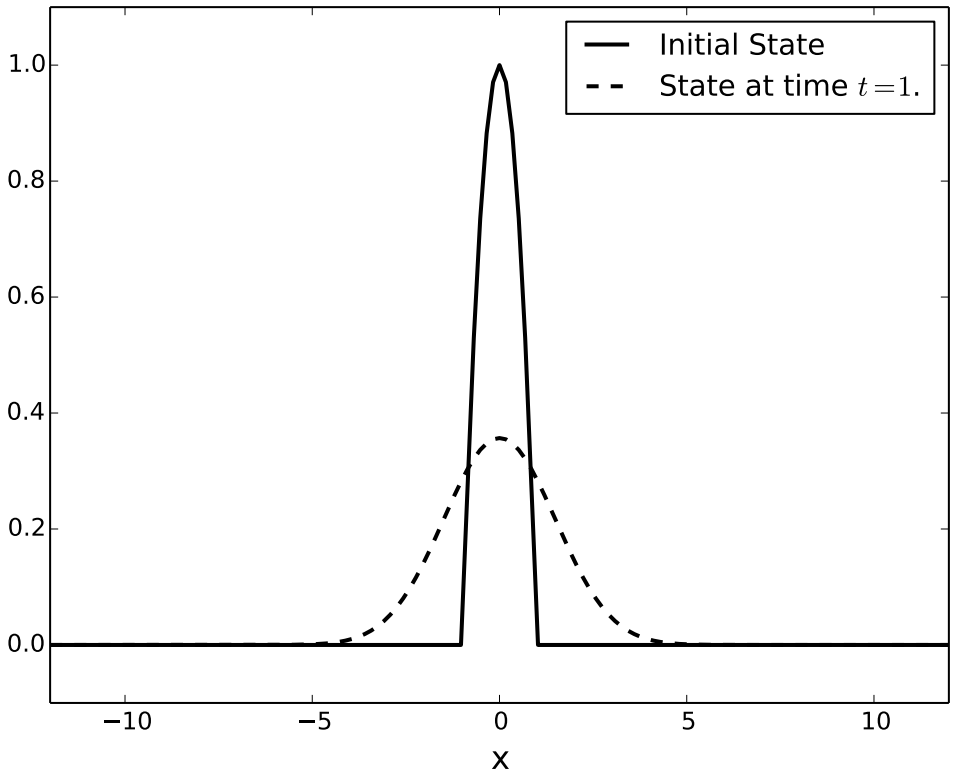


Figure 1.3: The initial and final states for equation Problem 2.

These approximations give the method

$$\begin{aligned} \frac{U_i^{j+1} - U_i^j}{k} &= \frac{1}{2}\left(\frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2} + \frac{U_{i+1}^{j+1} - 2U_i^{j+1} + U_{i-1}^{j+1}}{h^2}\right), \\ U_i^{j+1} &= U_i^j + \frac{k}{2h^2}\left(U_{i+1}^j - 2U_i^j + U_{i-1}^j + U_{i+1}^{j+1} - 2U_i^{j+1} + U_{i-1}^{j+1}\right). \end{aligned} \tag{1.4}$$

This method can be written in matrix form as

$$BU^{j+1} = AU^j,$$

where A and B are tridiagonal matrices given by

$$B = \begin{bmatrix} 1 & 0 & & & \\ -\lambda & 1+2\lambda & -\lambda & & \\ & \ddots & \ddots & \ddots & \\ & & -\lambda & 1+2\lambda & -\lambda \\ & & & 0 & 1 \end{bmatrix},$$

$$A = \begin{bmatrix} 1 & 0 & & & \\ \lambda & 1-2\lambda & \lambda & & \\ & \ddots & \ddots & \ddots & \\ & & \lambda & 1-2\lambda & \lambda \\ & & & 0 & 1 \end{bmatrix},$$

where $\lambda = \nu k/(2h^2)$, and U^j represents the approximation at time t_j. Note that here we have defined λ differently than we did before!

How do we know if a numerical approximation is reasonable? One way to determine this is to compute solutions for various step sizes h and see if the solutions are converging to something. To be more specific, suppose our finite difference method is $\mathcal{O}(h^p)$ accurate. This means that the error $E(h) \approx Ch^p$ for some constant C as $h \to 0$ (i.e., for $h > 0$ small enough).

So compute the approximation y_k for each stepsize h_k, $h_1 > h_2 > \ldots > h_m$. We will think of y_m as the true solution. Then the error of the approximation for stepsize $h_k, k < m$, is

$$\begin{aligned} E(h_k) &= \max(|y_k - y_m|) \approx Ch_k^p, \\ \log(E(h_k)) &= \log(C) + p\log(h_k). \end{aligned}$$

Thus on a log-log plot of $E(h)$ vs. h, these values should be on a straight line with slope p when h is small enough to start getting convergence.

Problem 3. Using the Crank Nicolson method, numerically approximate the solution $u(x,t)$ of the problem

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12, 12], \quad t \in [0, 1], \\ &u(-12, t) = 0, \quad u(12, t) = 0, \\ &u(x, 0) = \max\{1 - x^2, 0\}. \end{aligned} \tag{1.5}$$

Demonstrate that the numerical approximation at $t = 1$ converges to $u(x, t = 1)$. Do this by computing U at $t = 1$ using $20, 40, 80, 160, 320$, and 640 steps. Use the same number of steps in both time and space. Reproduce the loglog plot shown in Figure 1.4. The slope of the line there shows the proper rate of convergence.

To measure the error, use the solution with the smallest h (largest number of intervals) as if it were the exact solution, then sample each solution only at the x-values that are represented in the solution with the largest h (smallest number of intervals). Use the ∞-norm on the arrays of values at those points to measure the error.

Notice that, since the Crank-Nicolson method is unconditionally stable, there is no CFL condition and we can use the same number of intervals in time and space.

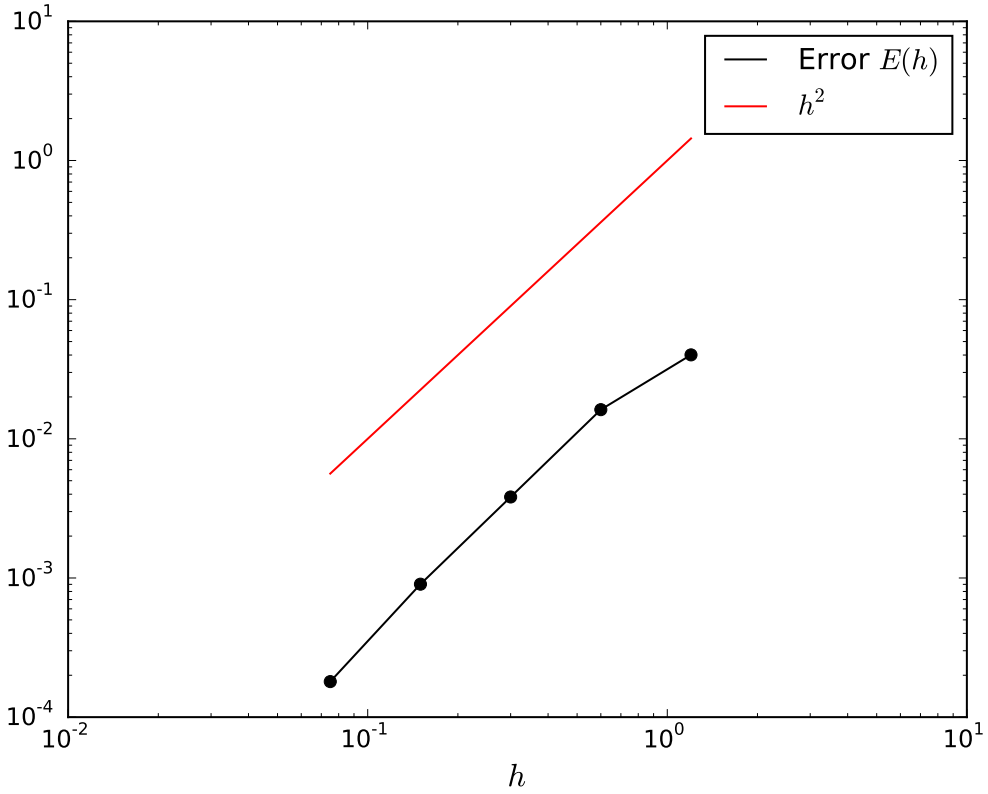


Figure 1.4: $E(h)$ represents the (approximate) maximum error in the numerical solution U to Problem 3 at time $t = 1$, using a stepsize of h.

1 Anisotropic Diffusion

Lab Objective: *Demonstrate the use of finite difference schemes in image analysis.*

A common task in image processing is to remove extra static from an image. This is most easily done by simply blurring the image, which can be accomplished by treating the image as a rectangular domain and applying the diffusion (heat) equation:

$$u_t = c\Delta u$$

where c is some diffusion constant and Δ is the Laplace operator. Unfortunately, this also blurs the boundary lines between distinct elements of the image.

A more general form of the diffusion equation in two dimensions is:

$$u_t = \nabla \cdot (c(x, y, t)\nabla u)$$

where c is a function representing the diffusion coefficient at each given point and time. In this case, $\nabla\cdot$ is the divergence operator and ∇ is the gradient.

To blur a picture uniformly, choose c to be a constant function. Since c controls how much diffusion is allowed at each point, it can be modified so that diffusion is minimized across edges in the image. In this way we attempt to limit diffusion near the boundaries between different features of the image, and allow smaller details of the image (such as static) to blur away. This method for image denoising is especially useful for denoising low quality images, and was first introduced by Pietro Perona and Jitendra Malik in 1987. It is known as Anisotropic Diffusion or Perona-Malik Diffusion.

A Finite Difference Scheme

Suppose we have some estimate E of the rate of change at a given point in an image. E will be largest at the boundaries in the image. We will then let $c(x, y, t) = g(E(x, y, t))$ where g is some function such that $g(0) = 1$ and $\lim_{x\to\infty} g(x) = 0$. Thus c will be small where E is large, so that little diffusion occurs near the boundaries of different portions of the image.

We will model this system using a finite differencing scheme with an array of values at a 2D grid of points, and iterate through time. Let $U^n_{l,m}$ be the discretized approximation of the function u, n be the index in time, l be the index along the x-axis, and m be the index along the y-axis.

The Laplace operator can be approximated with the finite difference scheme

$$\Delta u = u_{xx} + u_{yy} \approx \frac{U^n_{l-1,m} - 2U^n_{l,m} + U^n_{l+1,m}}{(\Delta x)^2} + \frac{U^n_{l,m-1} - 2U^n_{l,m} + U^n_{l,m+1}}{(\Delta y)^2}.$$

A good metric to use with images is to let the distance between each pixel be equal to one, so $\Delta x = \Delta y = 1$. Rearranging terms, we obtain

$$\Delta u \approx (U^n_{l-1,m} - U^n_{l,m}) + (U^n_{l+1,m} - U^n_{l,m}) + (U^n_{l,m-1} - U^n_{l,m}) + (U^n_{l,m+1} - U^n_{l,m}).$$

Again, since we are working with images and not some time based problem, we can without loss of generality let $\Delta t = 1$, so we obtain the finite difference scheme

$$U^{n+1}_{l,m} = U^n_{l,m} + (U^n_{l-1,m} - U^n_{l,m}) + (U^n_{l+1,m} - U^n_{l,m}) + (U^n_{l,m-1} - U^n_{l,m}) + (U^n_{l,m+1} - U^n_{l,m}).$$

We will now limit the diffusion near the edges of objects by making the modification

$$\begin{aligned} U^{n+1}_{l,m} = U^n_{l,m} + \lambda\Big(& g(|U^n_{l-1,m} - U^n_{l,m}|)(U^n_{l-1,m} - U^n_{l,m}) \\ & + g(|U^n_{l+1,m} - U^n_{l,m}|)(U^n_{l+1,m} - U^n_{l,m}) \\ & + g(|U^n_{l,m-1} - U^n_{l,m}|)(U^n_{l,m-1} - U^n_{l,m}) \\ & + g(|U^n_{l,m+1} - U^n_{l,m}|)(U^n_{l,m+1} - U^n_{l,m})\Big), \end{aligned} \tag{1.1}$$

where $\lambda \leq \frac{1}{4}$ is the stability condition.

In this difference scheme, each term is affected most by nearby terms that are most similar to it, so less diffusion will happen anywhere there is a sharp difference between pixels. This scheme also has the useful property that it does not increase or decrease the total brightness of the image. Intuitively, this is because the effect of each point on its neighbors is exactly the opposite effect its neighbors have on it.

Two commonly used functions for g are $g(x) = e^{-\left(\frac{x}{\sigma}\right)^2}$ and $g(x) = \frac{1}{1+\left(\frac{x}{\sigma}\right)^2}$. The parameter σ allows us to control how much diffusion decreases across boundaries, with larger σ values allowing more diffusion. Note that $g(0) = 1$ and $\lim_{x\to\infty} g(x) = 0$ for both functions. In this lab we use $g(x) = e^{-\left(\frac{x}{\sigma}\right)^2}$.

It is worth noting that this particular difference scheme is *not* an accurate finite difference scheme for the version of the diffusion equation we discussed before, but it *does* accomplish the same thing in the same way. As it turns out, this particular scheme is the solution to a slightly different diffusion PDE, but can still be used the same way.

For this lab's examples we read in the image using the `imageio.imread` function, and normalized it so that the colors are represented as floating point values between 0 and 1. An image can converted to black and white when it is read by including the argument `as_gray=True`.

```
from matplotlib import cm, pyplot as plt
from imageio import imread

# To read in an image, convert it to grayscale, and rescale it.
picture = imread('balloon.png', as_gray=True) * 1./255

# To display the picture as grayscale
plt.imshow(picture, cmap=cm.gray)
plt.show()
```

Simplifying Calculations

You will notice that the algorithm given in 1.1 does not describe what to do for the edges and corners of U^{n+1}. In these cases we will simply eliminate the undefined terms in the algorithm. For example, the top edge equation becomes

$$\begin{aligned} U_{l,m}^{n+1} = U_{l,m}^{n} + \lambda (& g(|U_{l+1,m}^{n} - U_{l,m}^{n}|)(U_{l+1,m}^{n} - U_{l,m}^{n}) \\ & + g(|U_{l,m+1}^{n} - U_{l,m}^{n}|)(U_{l,m+1}^{n} - U_{l,m}^{n})) \\ & + g(|U_{l,m-1}^{n} - U_{l,m}^{n}|)(U_{l,m-1}^{n} - U_{l,m}^{n})), \end{aligned}$$

and top left corner equation becomes

$$\begin{aligned} U_{l,m}^{n+1} = U_{l,m}^{n} + \lambda (& g(|U_{l+1,m}^{n} - U_{l,m}^{n}|)(U_{l+1,m}^{n} - U_{l,m}^{n}) \\ & + g(|U_{l,m+1}^{n} - U_{l,m}^{n}|)(U_{l,m+1}^{n} - U_{l,m}^{n})). \end{aligned}$$

Essentially we are only using the terms of the difference scheme that are actually defined.

To help facilitate this we can create a larger "padded" matrix that will make these calculations easy to do. This padded matrix will have an extra row on the top and bottom, and an extra column on either side of the original matrix. These extra rows and columns will duplicate the outer edge of the original matrix.

So if our original array `X` has shape `m,n`, then our padded array `Y` has shape `m+2,n+2`. The top edge of `Y` will be defined so that `Y[0,1:-1] == X[0,:]` is true, and the rest of the edges of `Y` follow the same pattern.

Notice that this allows us to simply implement the algorithm found in 1.1 without having to make special cases for the edges and corners, since those previously undefined terms become zero when using the padded matrix.

Problem 1. Complete the following function, by implementing the anisotropic diffusion algorithm found in 1.1 for black and white images. Use the padded array technique found in the Simplifying Calculations section.

In your function, use

$$g(x) = e^{-\left(\frac{x}{\sigma}\right)^2}$$

```
def anisdiff_bw(U, N, lambda_, g):
    """ Run the Anisotropic Diffusion differencing scheme
    on the array U of grayscale values for an image.
    Perform N iterations, use the function g
    to limit diffusion across boundaries in the image.
    Operate on U inplace to optimize performance. """
    pass
```

Run the function on `balloon.jpg`. Show the original image and the diffused image for $\sigma = .1$, $\lambda = .25$, $N = 5, 20, 100$.

original image

5 iterations with $\sigma = .1$ and $\lambda = .25$

20 iterations

100 iterations

Color Schemes

Colored images can be processed in a similar manner. Instead of being represented as a two-dimensional array, colored images are represented as three dimensional arrays. The third dimension is used to store the intensities of each of the standard 3 colors. This diffusion process can be carried out in the exact same way, on each of the arrays of intensities for each color, but instead of detecting edges just in one color, we need to detect edges in any color, so instead of using something of the form $g(|U^n_{l+1,m} - U^n_{l,m}|)$ as before, we will now use something of the form $g(||U^n_{l+1,m} - U^n_{l,m}||)$, where $U^n_{l+1,m}$ and $U^n_{l,m}$ are vectors now instead of scalars. The difference scheme can be treated as an equation on vectors in 3-space and now reads:

$$\begin{aligned} U^{n+1}_{l,m} = U^n_{l,m} + \lambda(& g(||U^n_{l-1,m} - U^n_{l,m}||)(U^n_{l-1,m} - U^n_{l,m}) \\ & + g(||U^n_{l+1,m} - U^n_{l,m}||)(U^n_{l+1,m} - U^n_{l,m}) \\ & + g(||U^n_{l,m-1} - U^n_{l,m}||)(U^n_{l,m-1} - U^n_{l,m}) \\ & + g(||U^n_{l,m+1} - U^n_{l,m}||)(U^n_{l,m+1} - U^n_{l,m})) \end{aligned}$$

When implementing this scheme for colored images, use the 2-norm on 3-space, i.e $||x|| = \sqrt{x_1^2 + x_2^2 + x_3^2}$ where x_1, x_2, and x_3 are the different coordinates of x.

Problem 2. Complete the following function to process a colored image. You may modify your code from the previous problem. Measure the difference between pixels using the 2-norm. Use the corresponding vector versions of the boundary conditions given in Problem 1.

```
def anisdiff_color(U, N, lambda_, sigma):
    """ Run the Anisotropic Diffusion differencing scheme
    on the array U of grayscale values for an image.
    Perform N iterations, use the function g = e^{-x^2/sigma^2}
    to limit diffusion across boundaries in the image.
    Operate on U inplace to optimize performance. """
    pass
```

Run the function on `balloons_color.jpg`. Show the original image and the diffused image for $\sigma = .1$, $\lambda = .25$, $N = 5, 20, 100$.

Hint: If you have an $m \times n \times 3$ matrix representing the RGB differences of each pixel, then to find a matrix representing the norm of the differences, you can use the following code. This code squares each value and sums along the last axis, and takes the square root. In order to keep the dimension size of the matrix and aid in broadcasting, you must use `keepdims=True`.

```
# x is mxnx3 matrix of pixel color values
norm = np.sqrt(np.sum(x**2, axis=2, keepdims=True))
```

original image

after 50 iterations

Figure 1.1: Smearing of similar colors when using an anisotropic diffusion filter.

Noisy Images

Problem 3. Use the following code to add noise to your grayscale image.

```
from numpy.random import randint

image = imread('balloon.jpg', as_gray=True)
x, y = image.shape
for i in xrange(x*y//100):
    image[randint(x),randint(y)] = 127 + randint(127)
```

Run `anisdiff_bw()` on the noisy image with $\sigma = .1$, $\lambda = .25$, $N = 20$. Display the original image and the noisy image. Explain why anisotropic diffusion does not smooth out the noise.

Hint: Don't forget to rescale.

Minimum Bias (Optional)

This sort of anisotropic diffusion can be very effective, but, depending on the image, it may also smear out edges that do not have large differences between them. An example of this limitation can be seen in Figure 1.1

As we can see, after 100 iterations, some of the boundaries between similar shades of grey have smeared unevenly. You may still have to look closely to see it. This can be counteracted somewhat by further decreasing the σ value, but if we have random noise throughout the image, this will not remove it. If we have random static in the image, we can remove this using a modified version of the filter. Instead of measuring the rate of change in the picture in each direction, we change each point according to whether or not any of its adjacent points have roughly the same value it has. This is called a minimum-biased filter. This sort of trick is especially good for removing isolated pixels that are different from those around them. A very simple way to do this is by taking the average of the two smallest differences between each pixel and its eight neighbors and using that in place of g in the difference scheme above. Along the boundaries, we do not have 8 neighbors for each pixel, but we can get by by just using the pixels we have and eliminating the other terms in the difference scheme, just as we did before. This will make it so that points that neighbor points of similar value will not be changed, while points that do not match their surroundings will be faded to become more like the points surrounding them. This does not have the same symmetrical diffusion as the other scheme, i.e. if one pixel changes, it does not necessarily change its neighboring pixels by the same amount. As long as you leave $\lambda \leq \frac{1}{4}$ and you have scaled the pixels to have floating point values between 0 and 1, the scheme will still remain within its minimum and maximum bounds, since the tendency is always to move points closer to the values of their neighbors. To demonstrate the action of such a filter, we make changes to random pixels in the color version of the same photo and use both filters to remove the noise we have added. Below, we include an example where we have added noise to the color version of that same picture, then used a minimum-biased filter to diminish the noise and the original filter to smooth what remains.

***Problem 4.** (Optional)

Implement the minimum-biased finite difference scheme described above. Add noise to `balloons_color.jpg` using the provided code below, and clean it using your implementation. Show the original image, the noised image, and the cleaned image.

```
image = imread('balloons_color.jpg')
x,y,z = image.shape
for dim in xrange(z):
    for i in xrange(x*y//100):
        # Assign a random value to a random place
        image[randint(x),randint(y),dim] = 127 + randint(127)
```

Hint: Don't forget to rescale.

original image

randomly changed 100000 color values

300 iterations of a min-biased scheme

after 8 additional iterations of the first filter with $\lambda = .25$ and $\sigma = .04$.

1 The Finite Element method

Lab Objective: *The finite element method is commonly used for numerically solving partial differential equations. We introduce the finite element method via a simple BVP describing the steady state distribution of heat in a pipe as fluid flows through.*

Advection-Diffusion of Heat in a Fluid

We begin with the heat equation

$$y_t = \varepsilon y_{xx} + f(x)$$

where $f(x)$ represents any heat sources in the system, and εy_{xx} models the diffusion of heat. We wish to study the distribution of heat in a fluid that is moving at some constant speed a. This can be modelled by adding an advection or transport term to the heat equation, giving us

$$y_t + ay_x = \varepsilon y_{xx} + f(x).$$

We consider a fluid flowing through a pipe from $x = 0$ to $x = 1$ with speed $a = 1$, and as it travels it is warmed at a constant rate $f = 1$. We will impose the condition that $y = 2$ at $x = 0$, so that the fluid is already at a constant temperature as it enters the pipe.

These conditions yield

$$\begin{aligned} &y_t + y_x = \varepsilon y_{xx} + 1, \quad 0 < x < 1, \\ &y(0) = 2 \end{aligned}$$

As time increases we expect the temperature of the fluid in the pipe to reach a steady state distribution, with $y_t = 0$. The heat distribution then satisfies

$$\begin{aligned} &\varepsilon y'' - y' = -1, \quad 0 < x < 1, \\ &y(0) = 2. \end{aligned}$$

This problem is not fully defined, since it has only one boundary condition. Suppose a device is installed on the end of the pipe that nearly instantaneously brings the heat of the water up to $y = 4$. Physically we expect this extra heat that is introduced at $x = 1$ to diffuse backward through the water in the pipe. This leads to a well defined BVP,

$$\begin{aligned} &\varepsilon y'' - y' = -1, \quad 0 < x < 1, \\ &y(0) = 2, \quad y(1) = 4. \end{aligned} \tag{1.1}$$

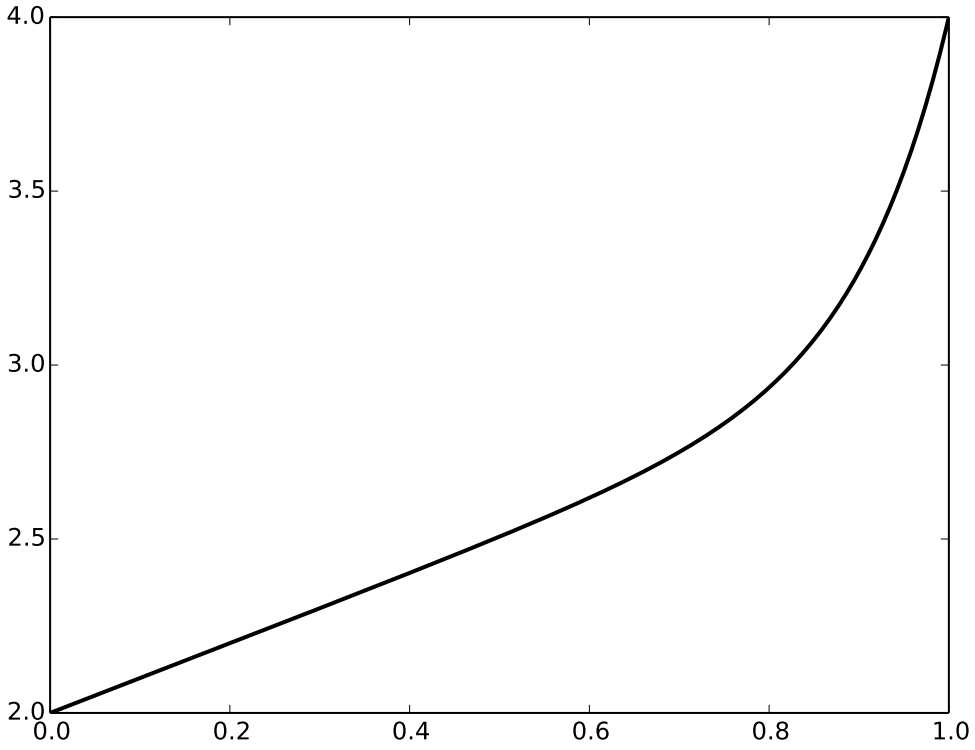


Figure 1.1: The solution of (1.1) for $\varepsilon = .1$.

The Weak Formulation

Consider the equation

$$\begin{aligned} &\varepsilon y'' - y' = f, \quad 0 < x < 1, \\ &y(0) = \alpha, \quad y(1) = \beta. \end{aligned} \tag{1.2}$$

To find the solution y using the finite element method, we reframe the problem and look at what is known as its weak formulation.

Let w be a smooth function on $[0, 1]$ satisfying $w(0) = w(1) = 0$. Multiplying (1.2) by w and integrating over $[0, 1]$ yields

$$\begin{aligned} \int_0^1 fw &= \int_0^1 \varepsilon y''w - y'w, \\ &= \int_0^1 -\varepsilon y'w' - y'w. \end{aligned}$$

Define a bilinear function a and a linear function l by

$$a(y,w) = \int_0^1 -\varepsilon y'w' - y'w,$$
$$l(w) = \int_0^1 fw.$$

Rather than trying to solve (1.2), we instead consider the problem of finding a function y such that

$$a(y,w) = l(w), \quad \forall\, w \in V_0, \tag{1.3}$$

where V is some appropriate vector space that is expected to allow us to approximate the solution y, and $V_0 = \{w \in V | w(0) = w(1) = 0\}$. (For example, we could consider the space of functions that are piecewise linear with vertices at a fixed set of points. This example is discussed further below.) This equation is called the weak formulation of (1.2).

Let P_n be some partition of $[0,1]$, $0 = x_0 < x_1 < \ldots < x_n = 1$, and let V_n be the finite-dimensional vector space of continuous functions v on $[0,1]$ where v is linear on each subinterval $[x_j, x_{j+1}]$. These subintervals are the finite elements for which this method is named. V_n has dimension $n+1$, since there are $n+1$ degrees of freedom for continuous piecewise linear functions in V. Let V_{n0} be the subspace of V_n of dimension $n-1$ whose elements are zero at the endpoints of $[0,1]$, and let $\triangle x_n = \max_{0 \le j \le n-1} |x_{j+1} - x_j|$.

Let $\{\mathrm{P}_n\}$ be a sequence of partitions that are refinements of each other, such that $\triangle x_n \to 0$ as $n \to \infty$. Then in particular $V_1 \subset V_2 \subset \ldots \subset V_n \ldots \subset V$. For each partition P_n we can look for an approximation $y_n \in V_n$ for the true solution y; if this is done correctly then $y_n \to y$ as $n \to \infty$.

The Numerical Method

Consider a partition $\mathrm{P}_5 = \{x_0, x_1, \ldots, x_5\}$. We will define some basis functions ϕ_i, $i = 0, \ldots, 5$ for the corresponding vector space V_5. Let the ϕ_i be the hat functions

$$\phi_i(x) = \begin{cases} (x - x_{i-1})/h_i & \text{if } x \in [x_{i-1}, x_i] \\ (x_{i+1} - x)/h_{i+1} & \text{if } x \in [x_i, x_{i+1}] \\ 0 & \text{otherwise} \end{cases}$$

where $h_i = x_i - x_{i-1}$; see Figures 1.2 and 1.3.

We look for an approximation $\hat{y} = \sum_{i=0}^5 k_i \phi_i \in V_5$ of the true solution y; to do this we must determine appropriate values for the constants k_i. We impose the condition on $\hat{y}$ that

$$a(\hat{y}, w) = l(w) \quad \forall\, w \in V_5.$$

Equivalently, we require that

$$a\left(\sum_{i=0}^5 k_i \phi_i, \phi_j\right) = l(\phi_j) \quad \text{for } j = 1,2,3,4,$$

since $\phi_1, \phi_2, \phi_3, \phi_4$ form a basis for V_5.

Since a is bilinear, we obtain

$$\sum_{i=0}^5 k_i a(\phi_i, \phi_j) = l(\phi_j) \quad \text{for } j = 1,2,3,4.$$

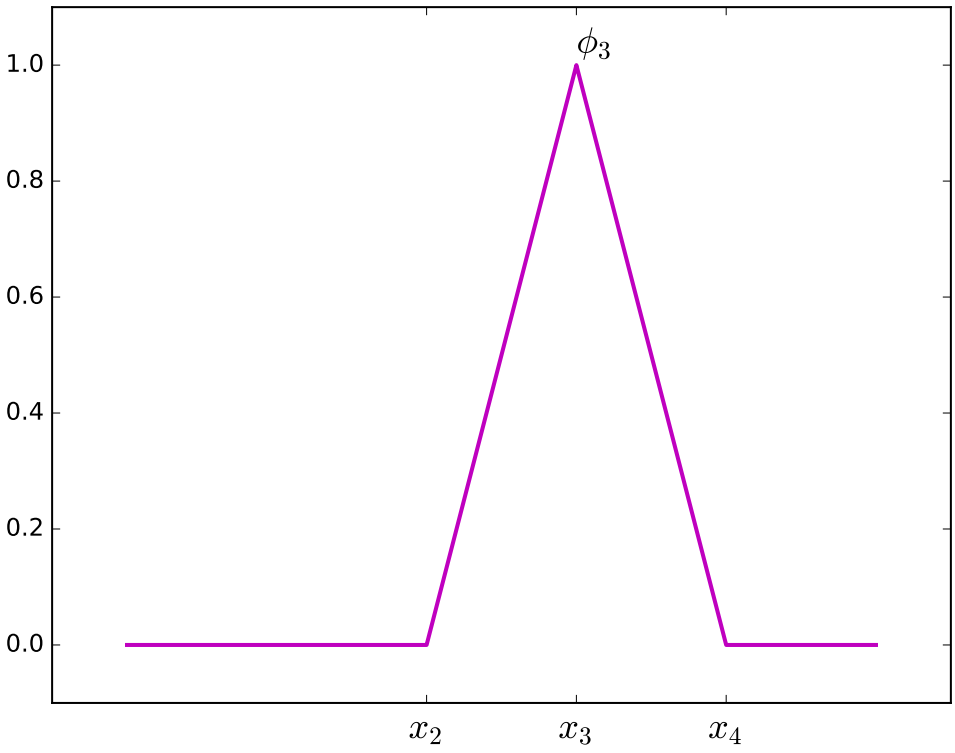


Figure 1.2: The basis function ϕ_3.

To satisfy the boundary conditions, we also require that $k_0 = \alpha$, $k_5 = \beta$. These equations can be written in matrix form as

$$AK = \Phi, \tag{1.4}$$

where

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ a(\phi_0,\phi_1) & a(\phi_1,\phi_1) & a(\phi_2,\phi_1) & 0 & 0 & 0 \\ 0 & a(\phi_1,\phi_2) & a(\phi_2,\phi_2) & a(\phi_3,\phi_2) & 0 & 0 \\ 0 & 0 & a(\phi_2,\phi_3) & a(\phi_3,\phi_3) & a(\phi_4,\phi_3) & 0 \\ 0 & 0 & 0 & a(\phi_3,\phi_4) & a(\phi_4,\phi_4) & a(\phi_5,\phi_4) \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$K = \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix}, \quad \Phi = \begin{bmatrix} \alpha \\ l(\phi_1) \\ l(\phi_2) \\ l(\phi_3) \\ l(\phi_4) \\ \beta \end{bmatrix}.$$

Note that $a(\phi_i, \phi_j) = 0$ for most values of i, j (that is, when the hat functions do not have overlapping domains). Thus the finite element method results in a sparse linear system. To compute

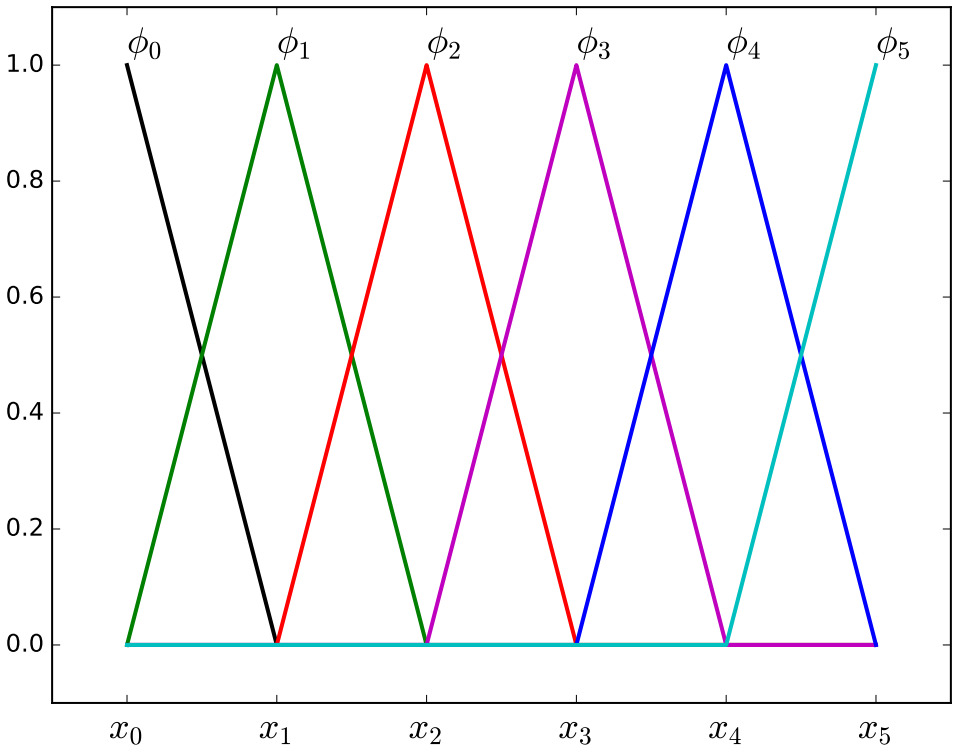


Figure 1.3: Basis functions for V_5.

the coefficients of (1.4) we begin by evaluating some integrals. Since

$$\phi_i'(x) = \begin{cases} 1/h_i & \text{for } x_{i-1} < x < x_i, \\ -1/h_{i+1} & \text{for } x_i < x < x_{i+1}, \\ 0 & \text{otherwise,} \end{cases}$$

we obtain

$$\int_0^1 \phi_i'\phi_j' = \begin{cases} -1/h_{i+1} & \text{if } j = i+1, \\ 1/h_i + 1/h_{i+1} & \text{if } j = i, \\ 0 & \text{otherwise,} \end{cases}$$

$$\int_0^1 \phi_i'\phi_j = \begin{cases} -1/2 & \text{if } j = i+1, \\ 1/2 & \text{if } j = i-1, \\ 0 & \text{otherwise,} \end{cases}$$

$$a(\phi_i, \phi_j) = \begin{cases} \varepsilon/h_{i+1} + 1/2 & \text{if } j = i+1, \\ -\varepsilon/h_i - \varepsilon/h_{i+1} & \text{if } j = i, \\ \varepsilon/h_i - 1/2 & \text{if } j = i-1, \\ 0 & \text{otherwise,} \end{cases}$$

$$l(\phi_j) = -(1/2)(h_j + h_{j+1}).$$

Equation (1.4) may now be solved using any standard linear solver. To handle the large number of elements required for Problem 3, you will want to use the tridiagonal algorithm provided in several of the earlier labs or the banded matrix solver included in `scipy.linalg`.

Problem 1. Use the finite element method to solve

$$\begin{aligned} &\varepsilon y'' - y' = -1, \\ &y(0) = \alpha, \quad y(1) = \beta, \end{aligned} \tag{1.5}$$

where $\alpha = 2, \beta = 4$, and $\varepsilon = 0.02$. Use $N = 100$ finite elements (101 grid points). Compare your solution with the analytic solution

$$y(x) = \alpha + x + (\beta - \alpha - 1)\frac{e^{x/\varepsilon} - 1}{e^{1/\varepsilon} - 1}.$$

Problem 2. One of the strengths of the finite element method is the ability to generate grids that better suit the problem. The solution of (1.5) changes most rapidly near $x = 1$. Compare the numerical solution when the grid points are unevenly spaced versus when the grid points are clustered in the area of greatest change; see Figure 1.4. Specifically, use the grid points defined by

```
even_grid = np.linspace(0,1,15)
clustered_grid = np.linspace(0,1,15)**(1./8)
```

Problem 3. Higher order methods promise faster convergence, but typically require more work to code. So why do we use them when a low order method will converge just as well, albeit with more grid points? The answer concerns the roundoff error associated with floating point

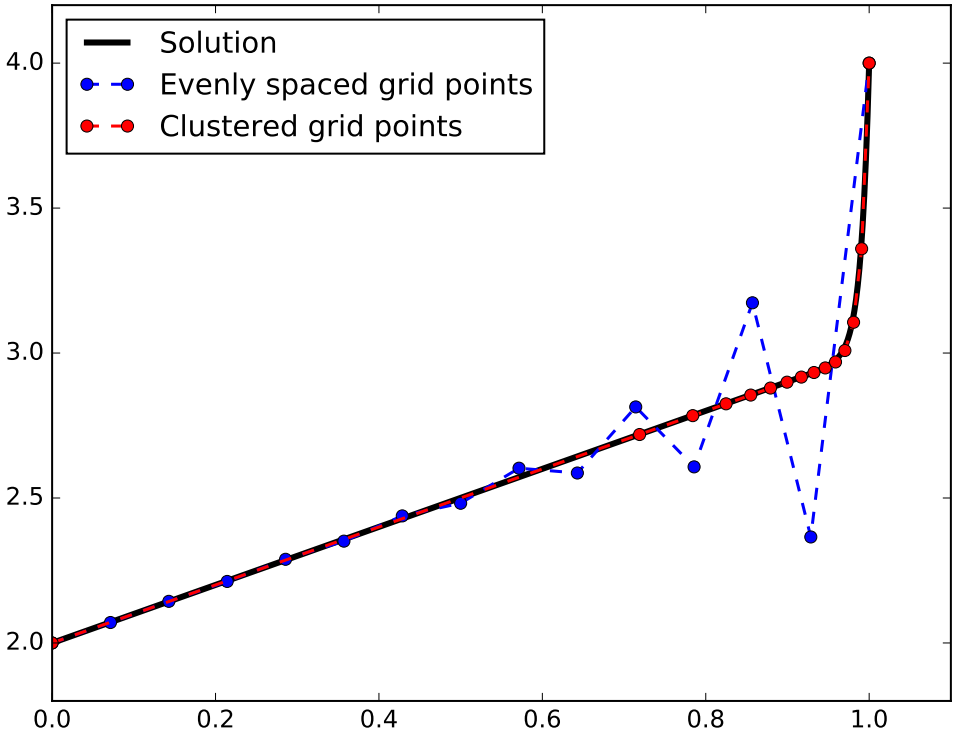


Figure 1.4: We plot two finite element approximations using 15 grid points.

arithmetic. Low order methods generally require more floating point operations, so roundoff error has a much greater effect.

The finite element method introduced here is a second order method, even though the approximate solution is piecewise linear. (To see this, note that if the grid points are evenly spaced, the matrix A in (1.4) is exactly the same as the matrix for the second order centered finite difference method.)

Solve (1.5) with the finite element method using $N = 2^i$ finite elements, $i = 4, 5, \ldots, 21$. Use a log-log plot to graph the error; see Figure 1.5.

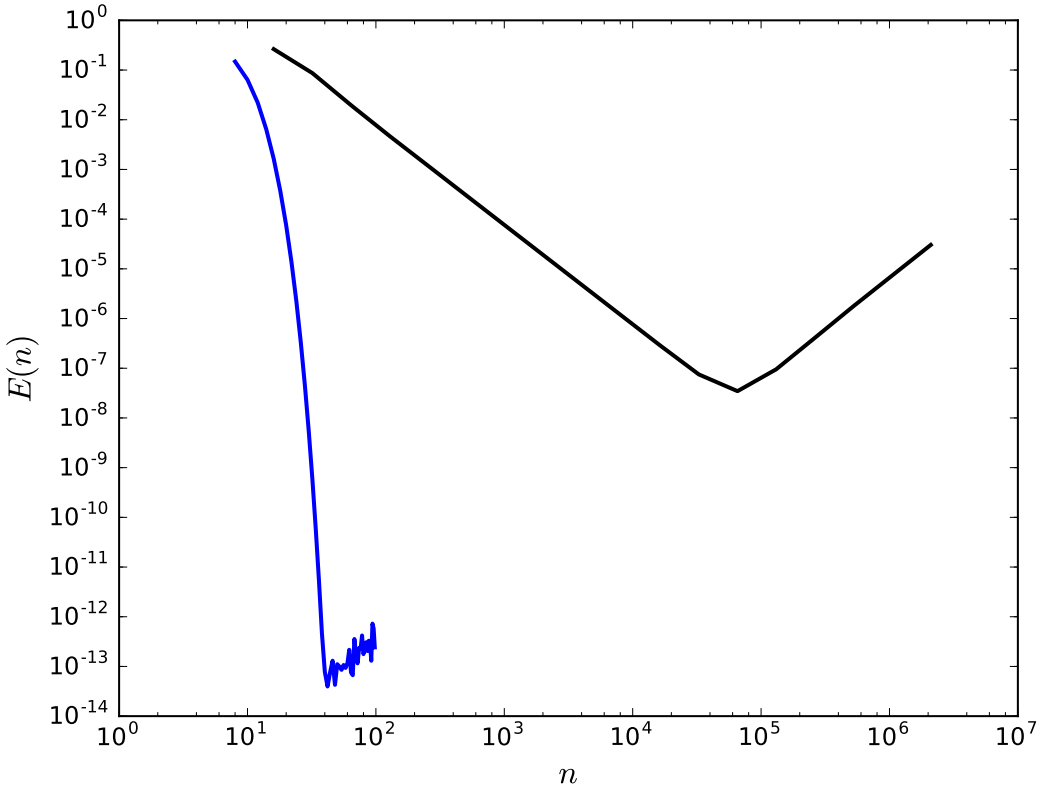


Figure 1.5: Error for the second order finite element method, as the number of subintervals n grows. Round-off error eventually overwhelms the approximation.