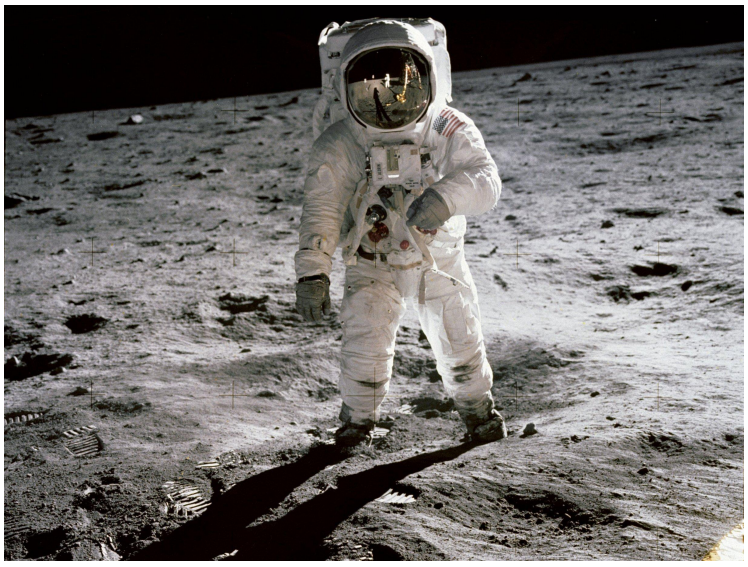
Labs for Foundations of Applied Mathematics

Volume 4
Modeling with Dynamics and Control

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker
Brigham Young University

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

K. Baldwin
Brigham Young University

J. Bejarano
Brigham Young University

J. Bennett
Brigham Young University

A. Berry
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

S. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

M. Cutler
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

O. Escobar Rodriguez
Brigham Young University

M. Fabiano
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
Brigham Young University

R. Fuhriman
Brigham Young University

T. Gledhill
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

S. Halverson
Brigham Young University

E. Hannesson
Brigham Young University

S. Harding
Brigham Young University

K. Harmer
Brigham Young University

J. Henderson
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

B. Hepner
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
Brigham Young University

J. Larsen
Brigham Young University

J. Larsen
Brigham Young University

J. Leete
Brigham Young University

Q. Leishman
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
Brigham Young University

M. Matsushita
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

E. Mercer
Brigham Young University

D. Miller
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

J. Murphey
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

C. Noorda
Brigham Young University

A. Oldroyd
Brigham Young University

J. Oliphant
Brigham Young University

A. Oveson
Brigham Young University

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

H. Ringer
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

K. Sandall
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

N. Schill
Brigham Young University

N. Sill
Brigham Young University

D. Smith
Brigham Young University

J. Smith
Brigham Young University

P. Smith
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

B. Trendler
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

R. Wonnacott
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics Volume 4: Modeling with Dynamics and Control* by Humpherys, Jarvis and Whitehead. The labs focus on numerical methods for solving ordinary and partial differential equations, including applications to optimal control problems. The reader should be familiar with Python [VD10] and its NumPy [Oli06, ADH+01, Oli07] and Matplotlib [Hun07] packages before attempting these labs. See the Python Essentials manual for introductions to these topics.

Contents

Part I

Labs

1 Animations and 3D Plotting in Matplotlib

Lab Objective: *Animations and 3D plots 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.*

Animation Basics

The Matplotlib library has a module `matplotlib.animation` that has many powerful options In particular, it contains a class called `FuncAnimation` that we will use throughout this lab. This class allows us to create animations in a very flexible way. `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 an animated plot using the `FuncAnimation` class:

1. Create a figure object.
2. Create line objects to be altered dynamically.
3. Choose how to parameterize the frames.
4. Create a function to update line objects.
5. Create a `FuncAnimation` object.
6. Display the animation (several methods exist to do so).

Let's work through an example of this. Consider the function

$$f(x,t) = \frac{1}{1+t}e^{-x^2/(1+t)^2}$$

modeling the diffusion of heat through a rod. Suppose that we want to animate how the temperature changes in the segment of a rod for $x \in [-4,4]$ as time evolves from $t = 0$ to $t = 5$. In this case, it will be easiest to not precompute the data. This has a higher risk of making the plot stutter if we just use `plt.show()`, but other methods of showing the plot (discussed below) can avoid this issue.

First, we'll set up ranges for x and t:

```
import numpy as np

xs = np.linspace(-4, 4, 150)
ts = np.linspace(0, 5, 251)
```

Next, we will explicitly create the figure and axis objects, as well a line object that we will update. Most Matplotlib functions actually return the objects that they manipulate, so this mostly just requires saving the output of the functions we call:

```
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation

# Create a figure and axis object
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
plt.xlim([-2, 2])
plt.ylim([0, 1.1])
```

Note

Many functions for customizing plots (e.g. setting titles, axis ranges, and labels) have different function names when called directly on an axes object. However, calling from `plt` can still be used for these as long as you create subplots one-at-a-time. More detailed information on the functions available to use with axes objects can be found here: `https://matplotlib.org/stable/api/axes_api.html`.

When we have explicitly defined an axes object, plots are generated by calling the chosen plot function on the axes object (for example, `ax.plot(...)`). The syntax is otherwise identical to calling directly from `plt`:

```
# Create an empty line object
# plt.plot actually returns a list of objects; the trailing comma
#   extracts the value from the list
line, = ax.plot([], [], "r-")
```

If we want more lines or also points or scatterplots, we create those here as well.

Next, we need to create the update function. The `FuncAnimation` object will call this function to create each frame of the animation. The value passed to this function will be a value from our frames list. Since we aren't precomputing the data we will plot, it will be most convenient to use `ts` as the frame values. In other cases, it can be more useful to use a `range()` instead. So, every time our update function is called, it will be passed the current value of t. We will use the `set_data()` method to update the line object we created earlier:

```
def update(t):
    line.set_data(xs, np.exp(- xs**2 / (1+t)**2) / (1+t))
```

If we had multiple line objects, we would call `.set_data()` on each of them inside of `update()`.

Finally, we will create and display the animation:

```
ani = FuncAnimation(fig, update, frames=ts, interval=20)
plt.show()
```

The `interval` parameter specifies how many milliseconds should be between each frame (as an integer). Usually we want to choose it so that it takes the animation one second to go from $t = 0$ to $t = 1$. If time goes from t_0 to t_f and we are using n_t points in our linspace, the right number of milliseconds per frame is given by the following formula:

$$\text{Interval} = \left\lceil \frac{1000(t_f - t_0)}{n_t - 1} \right\rceil.$$

Some additional parameters to `FuncAnimation` are given in the table below.

Note

Using `plt.show()` does not work on all platforms (e.g. VSCode). For alternate methods to show plots, refer to the Saving Animations section below and to the Additional Materials section.

Parameter	Description
`fargs` (tuple)	Additional arguments to pass update function
`repeat` (bool)	Determines whether animation repeats (Default: True.)
`blit` (bool)	Determines whether blitting is used. (Default: False.) Blitting means that `FuncAnimation` will attempt to only update parts of the plot that were actually changed, which may make your animation run faster. If this is enabled, your `update` function needs to return a list of all of the line objects that were updated.

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.

Achtung!

If you display an animation in a Jupyter notebook using `plt.show()`, it will not persist after closing, stopping, and reopening the notebook. What this means is that when you open the notebook later, there will be a static image in place of your animation. This is particularly problematic for turning in an animation to be graded. Instead of using `plt.show()`, save the animation to a file and embed it in your notebook (described below).

Saving Animations

The simplest way to save an animation is to encode it to a `.mp4` file, which 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 that is relatively easy to install:

- On Linux, run `sudo apt-get install ffmpeg` in the terminal, or the equivalent command for your package manager.
- On Mac, run `brew install ffmpeg` in the terminal.
- On Windows:
  - FFmpeg is easiest to install if you have Windows Subsystem for Linux (WSL) installed. In that case, you can run the Linux installation command in your WSL terminal.
  - Otherwise, you can download it manually from `https://github.com/BtbN/FFmpeg-Builds/releases`;[1] choose one of the versions there marked for Windows. If you use this option, you must also manually add its file location to your PATH environment variable. **If this option does not work easily,** it will be easier to just install WSL and use the other option. Consult *Getting Started* for instructions on how to do this.

To check if you have FFmpeg installed correctly, open your terminal and run the following command:

```
$ ffmpeg
```

It should print out a (rather long) help page.

When available, FFmpeg is generally chosen as the default by Matplotlib. If you get errors, however, you may need to manually specify to Matplotlib to use it:

```
animation.writer = animation.writers["ffmpeg"]
```

If Matplotlib does not recognize FFmpeg after it is installed, you may also need to restart your Python instance.

Now we proceed to actually saving the animation. Create the animation object as normal,

```
# Code to create figure, axes, and update function goes here
# ...
ani = animation.FuncAnimation(fig, update, frames=frames, interval=interval)
```

[1] This is one of the sources for the release version endorsed by the FFmpeg website `https://www.ffmpeg.org/download.html`.

then, in a seperate code cell, use its `.save()` method with the desired filename to render and save the video.

```
ani.save("my_animation.mp4")
```

Finally, to display the `.mp4` video in a Jupyter Notebook, place the following HTML code in a separate **markdown** cell (with the filename changed as appropriate):

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

The video will remain embedded in the Jupyter Notebook as long as the `.mp4` file is found in the same directory as it.

> NOTE
>
> In Jupyter Notebook, to insert a markdown cell, first insert a new cell, then in the dropdown menu at the top, change the type from Code to Markdown.
>
> If you update the video file, you will need to refresh the markdown cell for the embedded video to update. To do this, open the markdown cell in text mode again, and then return it to display mode.
>
> Remember to push the video files of your animation with the rest of your lab!

Problem 1. Use the FuncAnimation class to animate the function $y = \sin(x + 3t)$ where $x \in [0, 2\pi]$, and t ranges from 0 to 10 seconds. Embed your animation into the notebook.

Hint: For the `frames` argument, use a linspace from 0 to 10.

3D Plotting Basics

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"`:

```
>>> # Create figure object.
>>> fig = plt.figure()
>>>
>>> # Create 3D axis object using add_subplot().
>>> ax = fig.add_subplot(111, projection="3d")
```

3D axes objects can also be created using `plt.subplot`; however, many 3D plotting functions only work if you have the axes object, so it is better to use `fig.add_subplot`.

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. The `.npz` filetype loads as (essentially) a dictionary of arrays; this file has four keys: `"mercury"`, `"venus"`, `"earth"`, and `"mars"`. Create a 3D plot of the planet orbits, the starting positions of each planet as a point, and the position of the sun as a point, and compare your results with Figure 1.1. Make sure to include a legend.

As you work through the next few problems, it may be helpful to use a for loop and/or dictionaries to plot each of the planets.

Hint: The z range of the data is very narrow. Set the z range of the plot manually with `ax.set_zlim3d()`.

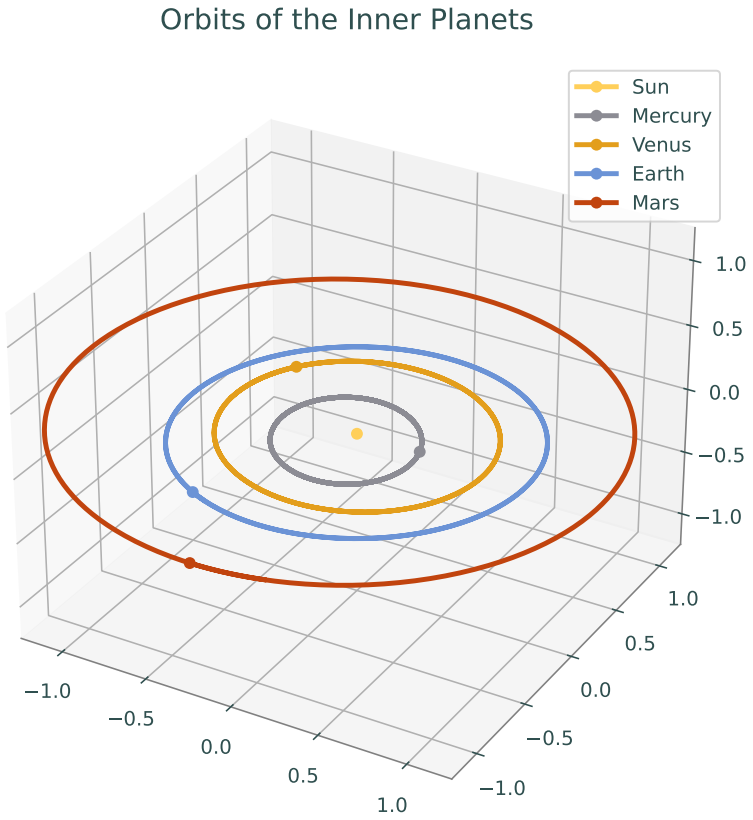


Figure 1.1: The solution to Problem 2.

3D Animations

The key difference between 2D and 3D animations is that the `.set_data()` method does not support setting the `z` values. Instead, use the `.set_data_3d()` method, which can be used to update `x`, `y`, and `z` all together. Note that in order to use this method, the line object must be a 3D line object. An empty one can be created as follows:

```
line, = ax.plot([], [], [])
```

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. This is what calling `plt.plot()` attempts to do; instead, it is much better to either render the animation to a file and then embed the file as discussed above (recommended), or to use the HTML5 API to embed it as discussed in Additional Materials (runs faster but often has issues after closing the notebook).

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 sun and current positions of the planets at each point in time. The lines displayed at each frame should only be the part the planet has traveled so far; see Figure 1.2 for an example of what this should look like. Include a legend, and embed your animated plot.

Hint: For the `frames` argument, use `range(1400)`. The parameter your `update()` function will receive will be the *index* of time for the frame, rather than the actual value of time. This will be useful for slicing arrays.

ACHTUNG!

The method `.set_data_3d()` expects a sequence of data for each dimension. Keep this in mind as you create your update functions for the points in Problem 3

Surface Plotting

3D surface plotting is very similar to regular 3D plotting discussed earlier. With surface plots, however, we must first create a *meshgrid* for X and Y. The process is identical to how to plot heatmaps and contour plots.

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

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

where $\mathbf{x} = [x, y]^\mathsf{T}$, $\mu = [0, 0]^\mathsf{T}$ 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 1.3.

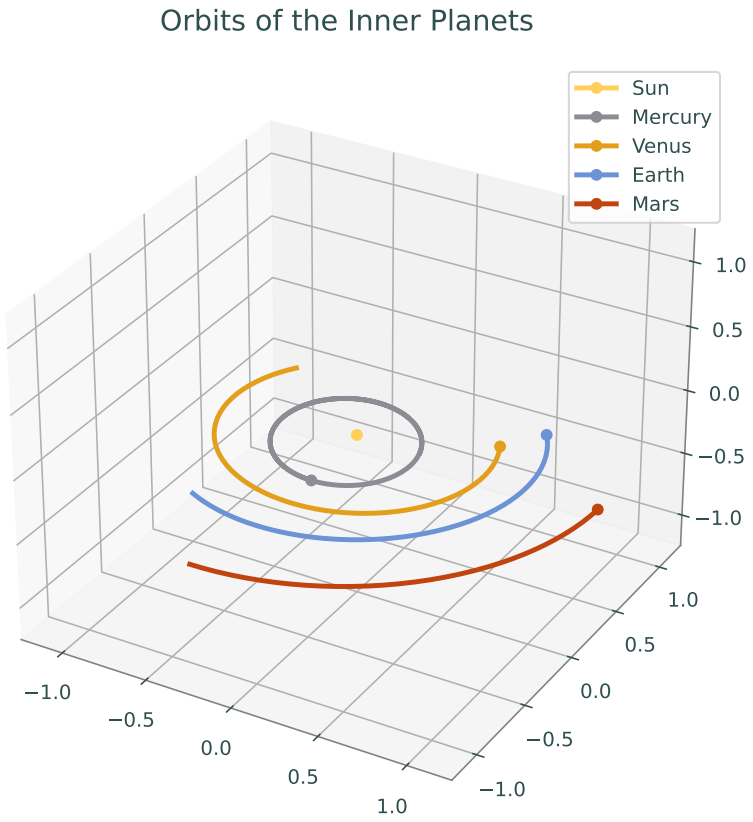


Figure 1.2: Example of what your Problem 3 animation should look like mid-animation.

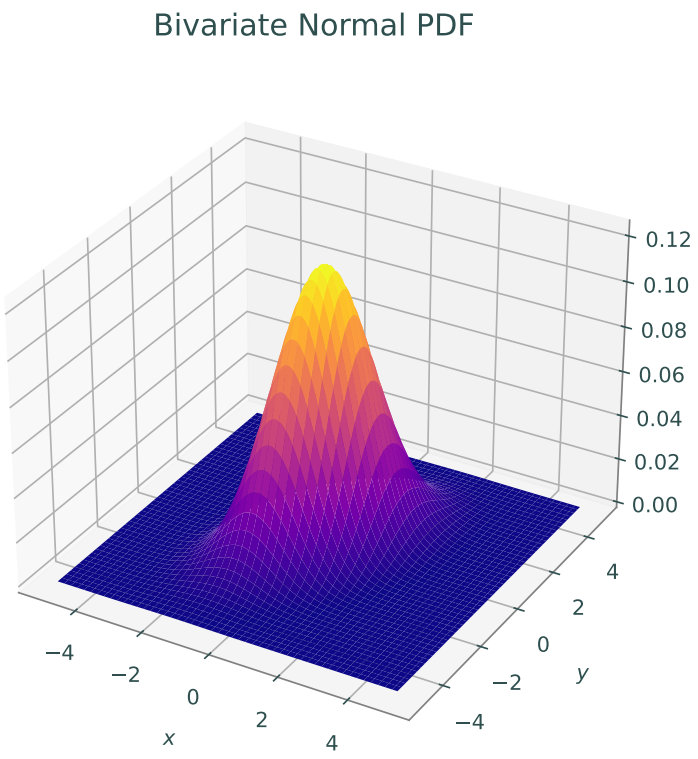


Figure 1.3: The solution to Problem 4.

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 axis limits must be reset after `ax.clear()` is called; otherwise, the limits will change every frame of the animation and it will look rather strange.

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. Embed your animation into the notebook.

In the language of partial differential equations, this is the solution to the following initial/boundary value problem:

$$u_{tt} = 6^2(u_{xx} + u_{yy})$$

$$(x, y) \in [0, 2] \times [0, 3], t \in [0, 5]$$

$$u(t, 0, y) = u(t, 2, y) = u(t, x, 0) = u(t, x, 3) = 0$$

$$u(0, x, y) = xy(2 - x)(3 - y)$$

Achtung!

Remember to push your video files with the rest of the lab! Otherwise, your grader will not be able to view your animations, even if they are correctly embedded in your notebook.

When running `git add`, you will need to explicitly specify the video files' filenames the first time you add them: `git add animation1.mp4 animation2.mp4 [...]`

Additional Material

Directly Embedding Animations

While saving animations to a file has the advantage that the animation will always persist if the notebook is closed and reopened, it tends to be much slower than directly embedding the animation in the notebook. Directly embedding can thus be useful in the process of creating an animation by allowing faster experimentation.

After creating an animation, calling `plt.show()` will attempt to embed it; however, many systems may struggle to display an animation in this way. When this is the 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 can be created by `matplotlib.animation`. This method tends to be much more simple than rendering the animation to an `.mp4` file and then embedding that file into a notebook, and it tends to encounter fewer bugs than using `plt.show()`. However, the animation generally does not persist if the notebook is closed and reopened, and this generally still requires FFmpeg to be installed. Here is a snippet you may reference to embed an animation using `IPython.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, interval)
# render as html5 and embed
HTML(ani.to_html5_video())
```

Achtung!

Note that animations that are embedded in the notebook using `HTML()` ***do not always*** persist if the notebook is closed and reopened. They sometimes do, but are very inconsistent. While this method can be useful for more quickly testing animations, ***do not*** use this method for embedding the final animations of your finished lab, as the person grading them will not be able to view your animations.

2 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 use `solve_ivp`:

1. The ODE function:
   This function defines the right-hand side of the ODE system, and returns an array containing its values for 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 of each coordinate of the ODE. In our example, these are the value of the "zeroeth" derivative, followed by the first derivative, and so on if there are higher order derivatives.

The following code sets up and solves the IVP in the above example:

```
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_ivp(ode, t_span, y0, max_step=0.1)

# as an alternative, the parameter t_eval can be used to evaluate the function
# at specific points; this can also be used to get a smooth graph
sol = solve_ivp(ode, t_span, y0, t_eval=np.linspace(0, 5, 150))
```

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()
```

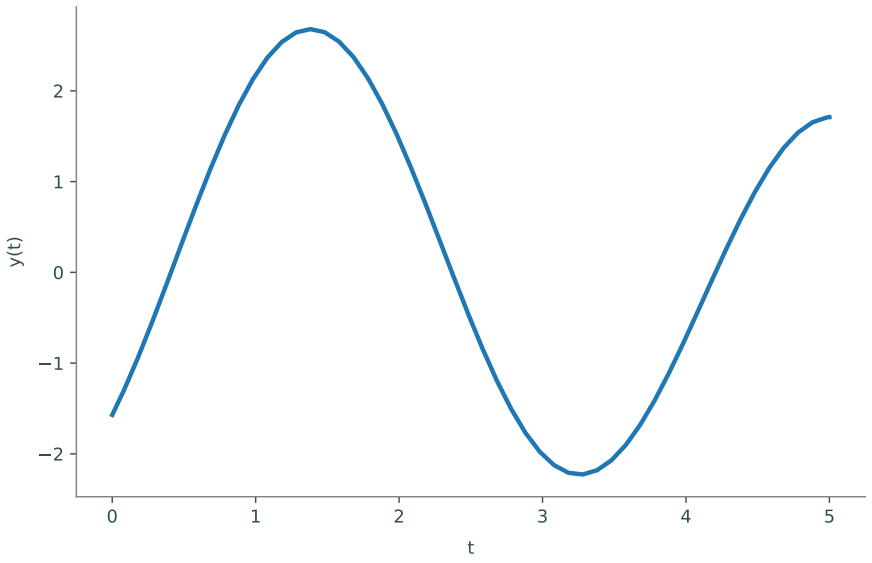


Figure 2.1: The solution to the above example

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 2.2.

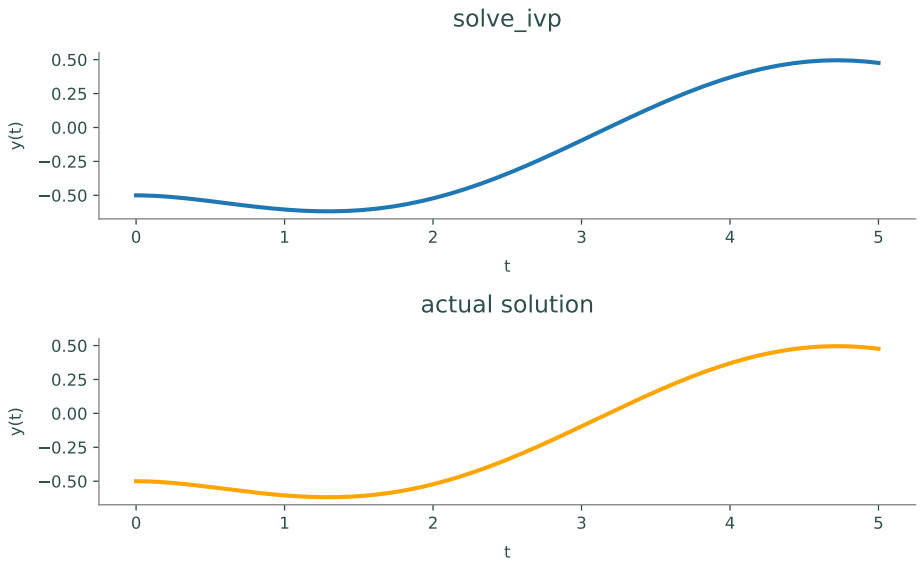


Figure 2.2: The solution to Problem 1

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 a BVP would be a differential equation modeling the path of a ball thrown in the air where the initial position ($y(a)$) and final position ($y(b)$) are known. Note that like an IVP problem, a BVP problem has two boundary conditions.

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

$$y'' + 9y = \cos(t), \quad y'(0) = 5, \quad y(\pi) = -\frac{5}{3}. \tag{2.1}$$

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 use `solve_bvp`:

1. The ODE function:
   This is essentially the same function we used in `solve_ivp`.[1]

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 coordinate of the ODE. Returning `ya[0]-x` would indicate that we know $y_1(a) = x$, `ya[1]-x` would indicate that we know $y_2(a) = x$, `yb[0]-x` would indicate that we know $y_1(b) = x$, and `yb[1]-x` would indicate that we know $y_2(b) = x$.

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). As part of its algorithm, `solve_bvp` will add additional points to the mesh to attempt to reduce the error of the approximation, so it is not generally necessary to pass in a very fine mesh. This also means that the mesh of the returned solution will generally not be the same as the one you pass in here.

[1] There is a technical difference between how the two methods call the ODE function. Unlike `solve_ivp`, `solve_bvp` calls the function on *all* of the time steps all at once, so `t` will be an array and `y` will be a (n, T) array where n is the dimension of the ODE and T is the number of timesteps. For most applications, this leads to no difference in how you code the ODE function, as can be seen in the examples; however, for some applications, such as piecewise ODE functions, this fact must be taken into consideration.

4. The initial guess:
   As we no longer know all of the initial values, we now must make a (hopefully 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()
```

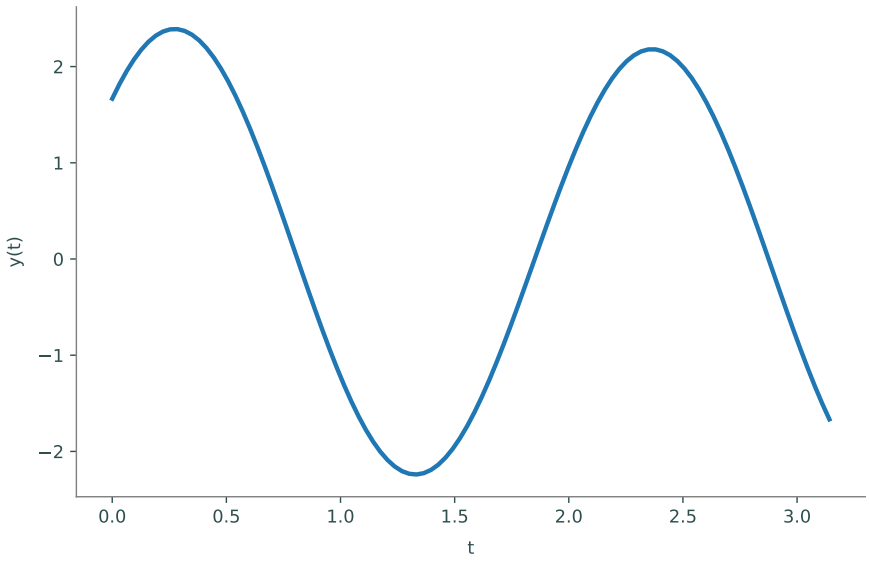


Figure 2.3: The solution to the above boundary value problem

Problem 2. Use `solve_bvp` to solve for y in the equation $y'' + y' = -\frac{1}{4}e^{-t/2} + \sin(t) - \cos(t)$ with boundary conditions $y(0) = 6$, $y'(5) = -0.324705$ and plot your solution on the interval $[0, 5]$. Compare this to the analytic solution $y = e^{-t/2} - \sin(t) + 5$.

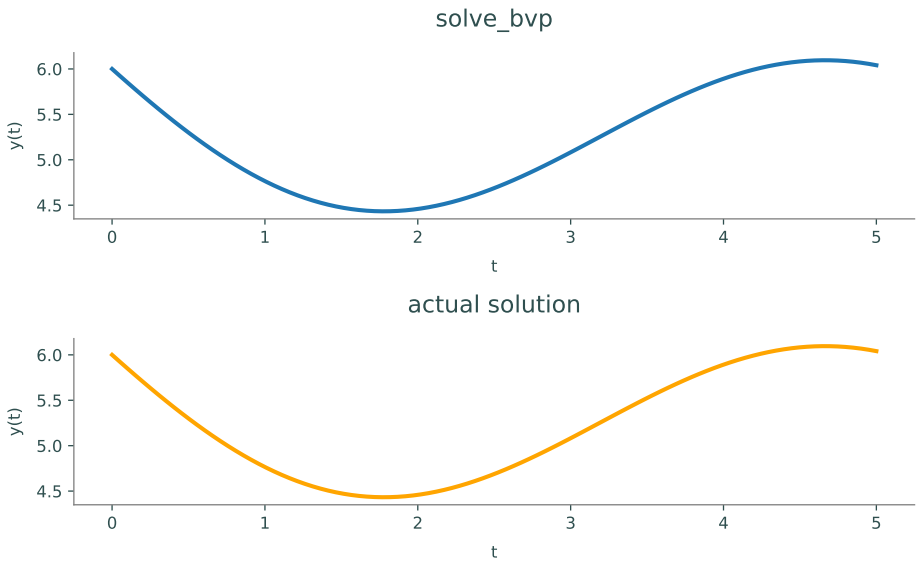


Figure 2.4: The solution to problem 2.

One other useful functionality of `solve_bvp`: `sol.sol` is a callable function, which is the estimation of the boundary value problem. You can plug in any value or numpy array (`sol.sol(np.linspace)`, `sol.sol(float)`, `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 reason for this is that the algorithm used by `solve_bvp` is essentially a version of Newton's method set up to approximate the boundary value problem, and thus can be sensitive to the initial guess. The next problem demonstrates the huge difference that can be made between a constant initial guess of 10 and a constant 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. For both of your initial guesses, use 50 steps in t.

The solution is found in Figure 2.5.

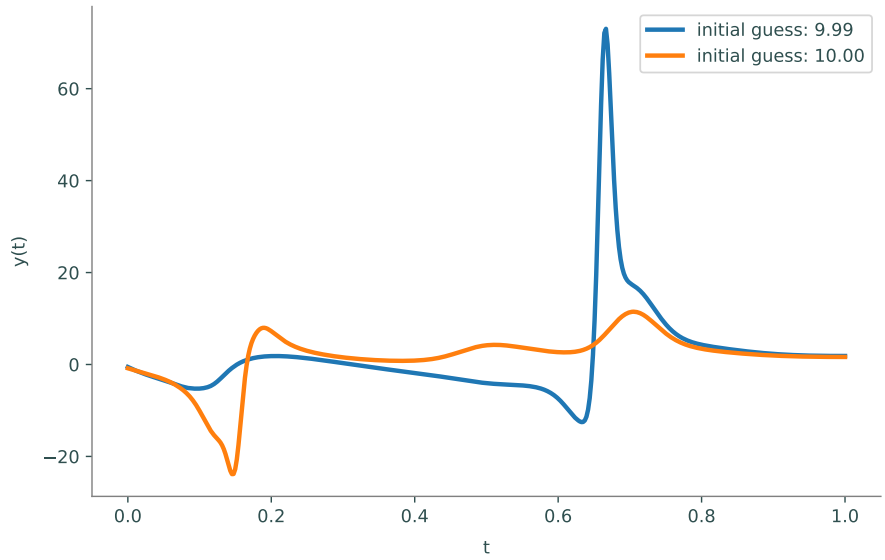


Figure 2.5: The solution to problem 3.

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 differences 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

$$\frac{dx}{dt} = ax + yz \tag{2.2}$$
$$\frac{dy}{dt} = bx + cy - xz \tag{2.3}$$
$$\frac{dz}{dt} = -z - xy \tag{2.4}$$

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 the Four-Wing attractor.

Problem 4. Use `solve_ivp` to solve the Four-Wing Attractor as described in equations (2.2), (2.3), and (2.4) 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.

Examples of solutions are given in Figure 2.6.

Hint: Because the attractor lies mostly within the box $[-2, 2]^2$, it is best to have the initial conditions also within this box. Also, you may need the time scale to run from about about 0 to 400 to visualize the full attractor.

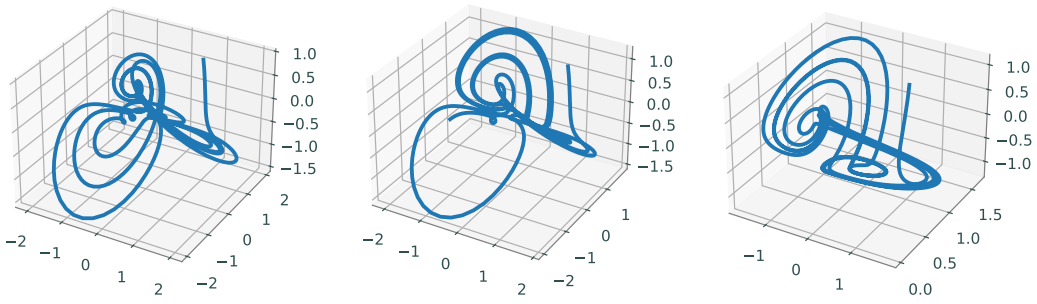


Figure 2.6: Possible solutions to problem 4.

BVPs with Free Parameters

Often when solving a BVP, there are extra variables that we wish to solve for in addition to the solution. These are called unknown or free parameters. A free parameter refers to any undetermined constant or parameter within the problem that remains free until boundary conditions or additional constraints provide a way to fix its value. We will encounter various scenarios in later labs when a method to solve for such parameters will be useful.

Fortunately, the function `solve_bvp` also supports solving for free parameters in a boundary value problem. The syntax is very similar to what we did above, except that our functions will have an additional argument that is a list of all of the free parameters. For example, suppose we have the following BVP with free parameter d, that is, in addition to the BVP, we are also solving for an unknown parameter d:

$$\begin{aligned} x'(t) &= (1+t)y(t) - x(t) \\ y'(x) &= d\sin(x(t)) \\ x(0) = 1,\ x(1) &= 0,\ y(1) = 2 \end{aligned}$$

Note that we need one additional boundary condition for each free parameter; in this case, since we have two variables and one free parameter, we need three boundary conditions. We set up the ODE and boundary condition functions as follows:

```
# The ODE function
def ode(t, y, p):
    ''' Defines the ODE system '''
    return np.array([
        (1+t)* y[1] - y[0],
        p[0] * np.sin(y[0])
    ])

# The boundary condition function
def bcs(ya, yb, p):
    ''' Defines the boundary conditions '''
    return np.array([
        ya[0] - 1,
        yb[0] - 0,
        yb[1] - 2
    ])
```

Note that both of these functions accept an additional argument `p`, which is a list of the free parameters in the problem. In this case, we only have one parameter, so `p=[d]`. Using `solve_bvp` to get the solution is similar to before, except that we must also pass in a guess for the free parameters with the argument `p`:

```
# Guess of the solution values
t = np.linspace(0, 1, 50)
y_guess = np.ones((2, 50))
p_guess = [1]

# Solve
sol = solve_bvp(ode, bcs, t, y_guess, p=p_guess)
```

The solution can be plotted as before, and the value of the free parameters for the solution can be found with `sol.p`:

```
plt.plot(sol.x, sol.y[0], label="$x(t)$")
plt.plot(sol.x, sol.y[1], label="$y(t)$")
```

```
plt.legend()
plt.xlabel('t')
plt.ylabel("y(t)")
plt.title(f'$d = {sol.p[0]:.4f}$')
plt.show()
```

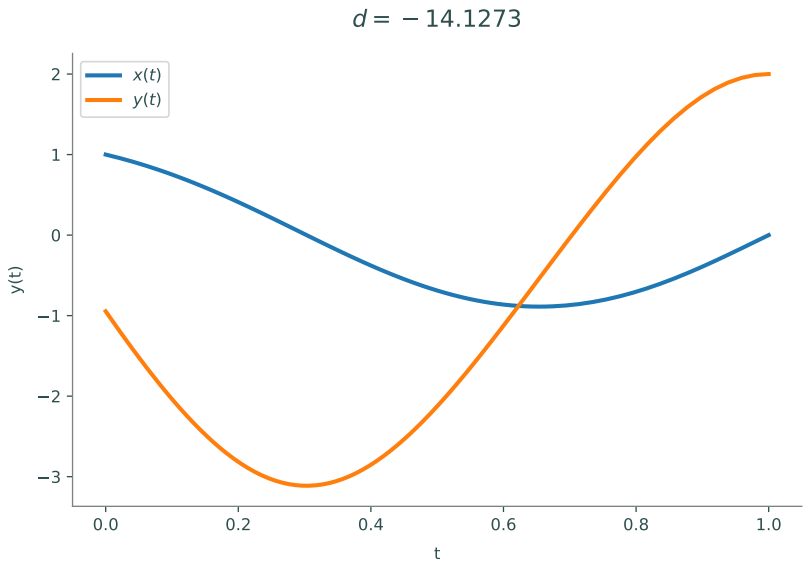


Figure 2.7: The solution to the above example

Free parameters occur in eigenvalue problems for differential operators. The general form of these is

$$\begin{aligned} Dy &= \lambda y \\ y(a) = &y(b) = 0 \end{aligned}$$

where D is some differential operator and λ is an unknown scalar. These differential eigenvalue problems are very analogous to the finite-dimensional case of matrix eigenvalue problems, except that instead of trying to find eigenvectors, we now try to find *eigenfunctions*. As in the matrix case, we are not interested in the trivial solution $y = 0$. Furthermore, if we have an eigenfunction y, any multiple ky is also an eigenfunction. To solve both of these issues, we will stipulate that $y'(a) = 1$; this guarantees the solution is not identically zero, and also makes it unique for a given value of λ.

Sturm-Liouville problems are an important category of these eigenvalue problems. These have the special form

$$\begin{aligned} (py')' + qy &= \lambda ry \\ y(a) = y(b) &= 0 \end{aligned}$$

where $p(t), q(t), r(t)$ are known functions and λ is an unknown scalar. Sturm-Liouville problems and their extensions are important theoretically, and their solutions have some very nice properties. They also have applications in PDEs and occur in areas such as physics and quantum mechanics.

Problem 5. An important problem in quantum mechanics is to find steady-state solutions of the Schrödinger equation. These functions are solutions to the *time-independent Schrödinger equation*. This equation is a differential equation for the wave function ψ, with one free parameter E. In one dimension, this equation is

$$-\frac{\hbar^2}{2m}\psi''(x) + U(x)\psi(x) = E\psi(x) \tag{2.5}$$

where U is a known function describing the potential energy. Note that this is in fact a Sturm-Liouville problem. If a function ψ and scalar E satisfy this equation, they describe an allowed steady state of a particle in the system. The value of E is the energy of the particle in that state, and the values of ψ are related to the probability of the particle being in any given location. For simplicity, we will let $\frac{\hbar^2}{2m} = 1$.[a]

Write a function that uses `solve_bvp` to find ψ and E that are solutions to (2.5) for the potential $U(x) = x^2$ and with boundary conditions $\psi(-1) = \psi(1) = 0, \psi'(-1) = 1$. By varying your initial guess for E, use your function to find solutions for several different values of E, and plot them together. Label the solutions with the values of E that you found.

[a]Making constants equal to one in this way is acutally done quite frequently in particle physics, by choosing the units we are using appropriately.

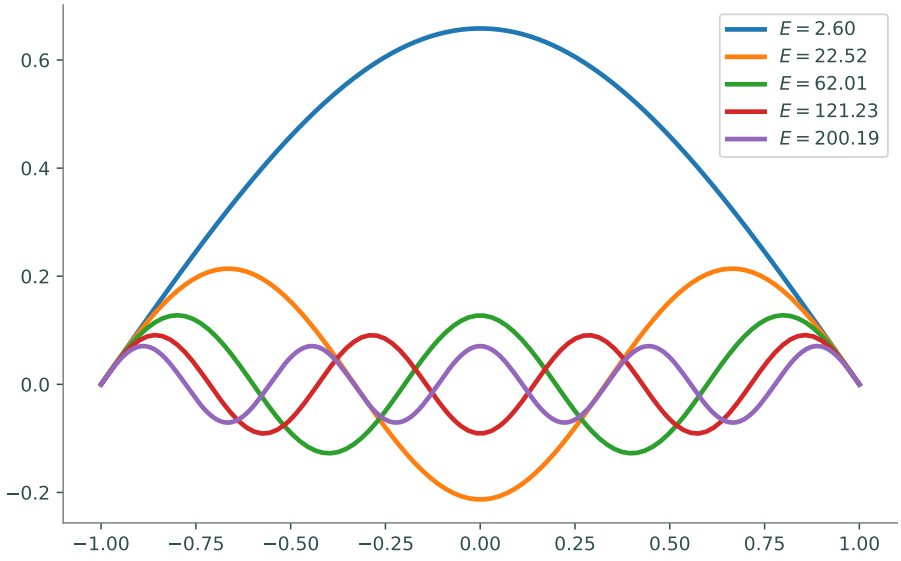


Figure 2.8: A possible solution to problem 5.

3 Modelling the Spread of an Epidemic: SIR Models

Numerical Solvers

We often rely on numerical solvers to numerically integrate ordinary differential equations (ODEs), especially for complicated and high-dimensional systems with no symbolic solution. 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`.

As we discussed in the previous lab, Intro to IVPs and BVPs, `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:

$$\begin{aligned} \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}' &= \begin{bmatrix} y_2(t) \\ \sin(t) - 5y_2(t) - y_1(t) \end{bmatrix} \\ y_1(0) = 0, \quad y_2(0) &= 1, \quad t \in [0, 3\pi] \end{aligned} \tag{3.1}$$

```
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, t_eval=np.linspace(t0, tf, 150))

# 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()
```

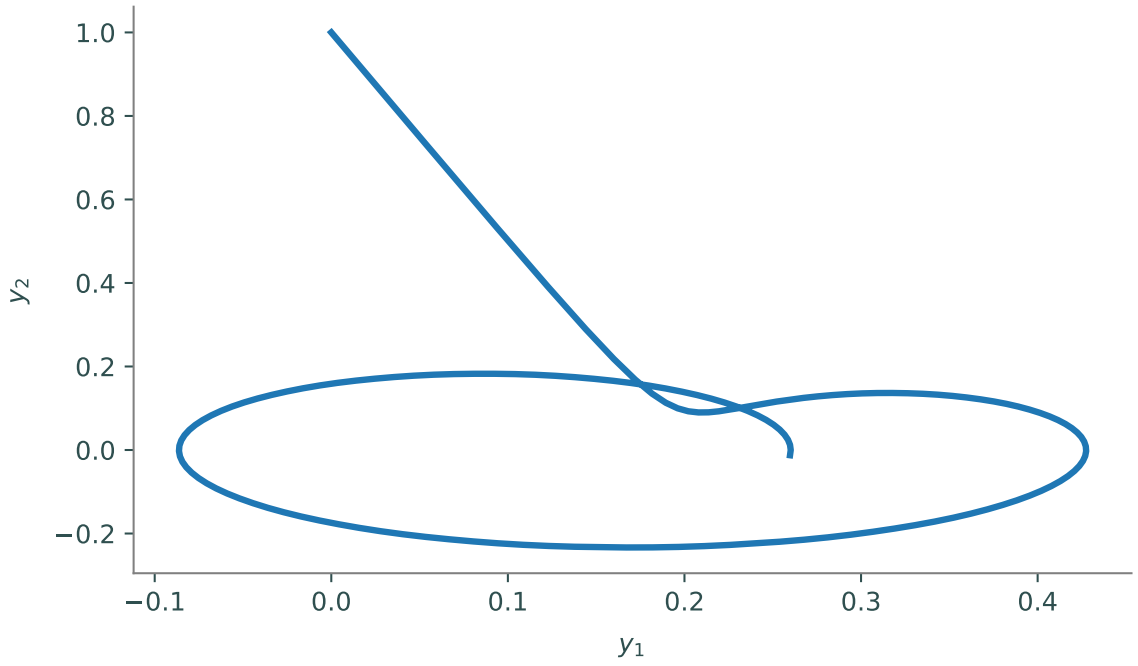


Figure 3.1: Solution to the example given by Equation (3.1)

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

$$\frac{dS}{dt} = -\beta IS, \tag{3.2}$$
$$\frac{dI}{dt} = \beta IS - \gamma I, \tag{3.3}$$
$$\frac{dR}{dt} = \gamma I. \tag{3.4}$$

Problem 1. Suppose that, in a city of approximately three million, five people have recently entered the city carrying a certain disease. Each infected individual 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` over a time period of fifty days, and plot your results. Compare your plot to Figure 1. 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_eval` argument of `solve_ivp` to specify the points in time that you want the solution's value at. This parameter accepts a linspace of time values. Specify enough points that your graph is smooth.

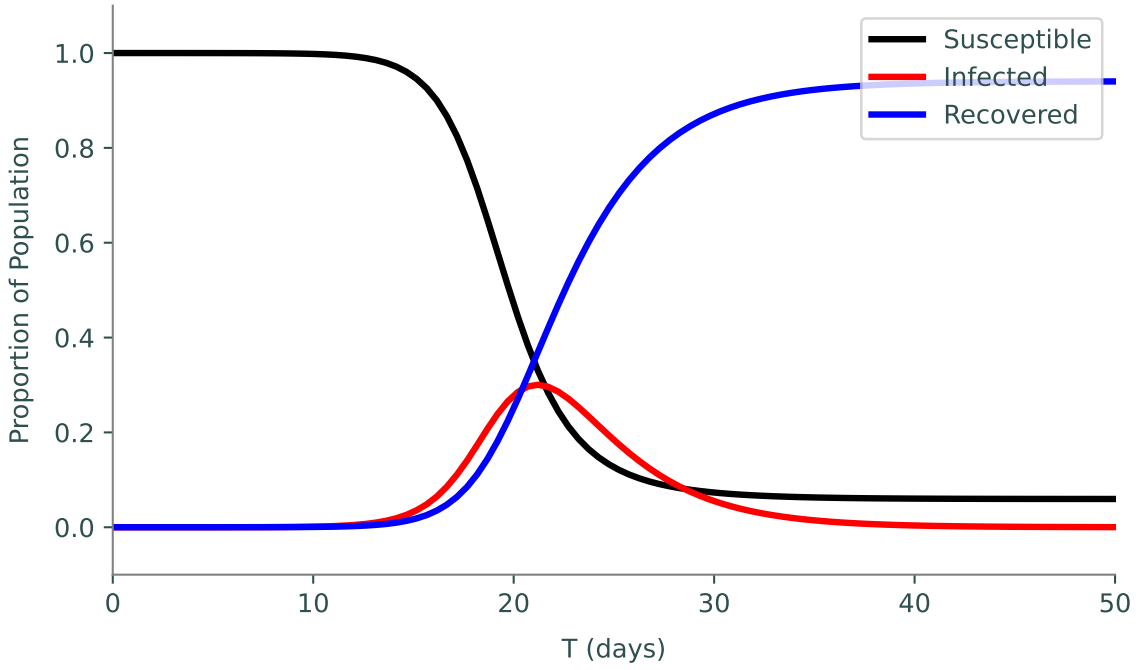


Figure 3.2: Solution to Problem 1

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 spreads the virus to 2.5 people on average over the whole time that they are sick.

Plot your results for each day and compare to Figure 3.3. Also answer the following questions:

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?

Hint: Recall that β is the average number of contacts an infected person makes per day that could spread the disease, and γ is the reciprocal of the average length of time spent in the infectious phase.

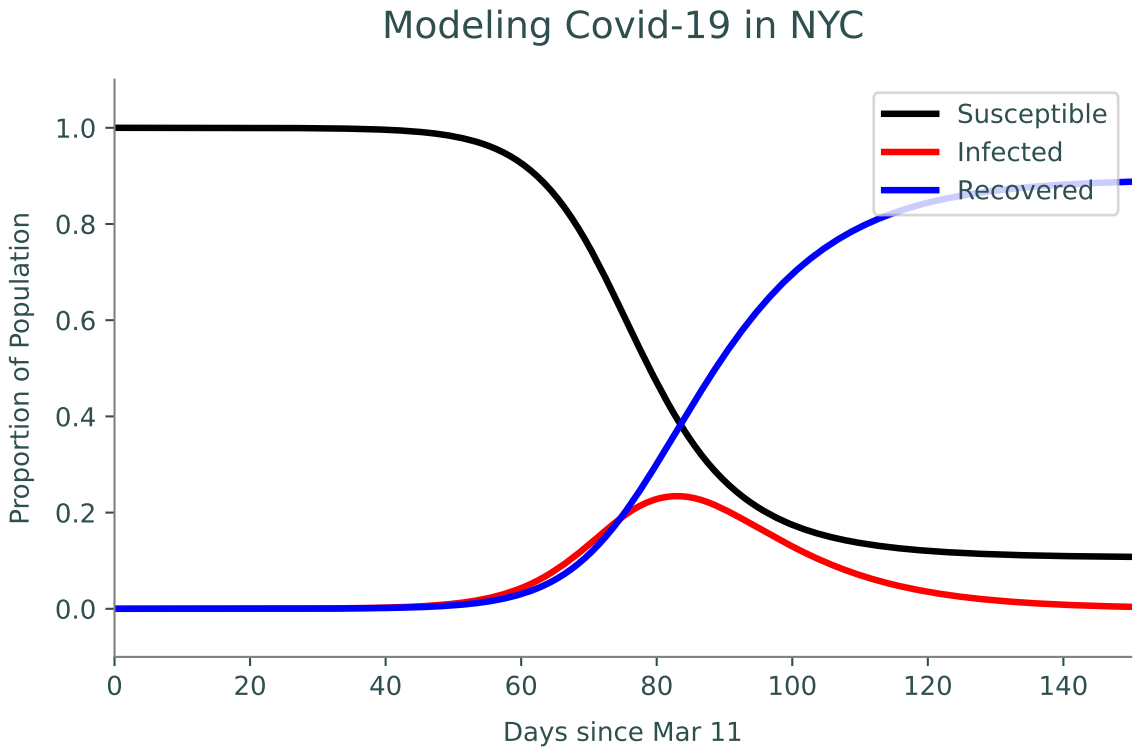


Figure 3.3: Solution to Problem 2.

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.

$$\frac{dS}{dt} = fR + \mu(1-S) - \beta IS, \tag{3.5}$$
$$\frac{dI}{dt} = \beta IS - (\gamma+\mu)I, \tag{3.6}$$
$$\frac{dR}{dt} = -fR + \gamma I - \mu R. \tag{3.7}$$

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. Suppose the average person is 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 rate $f = 1/50$. 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 let $\mu = 0.0001$.

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 3.4.

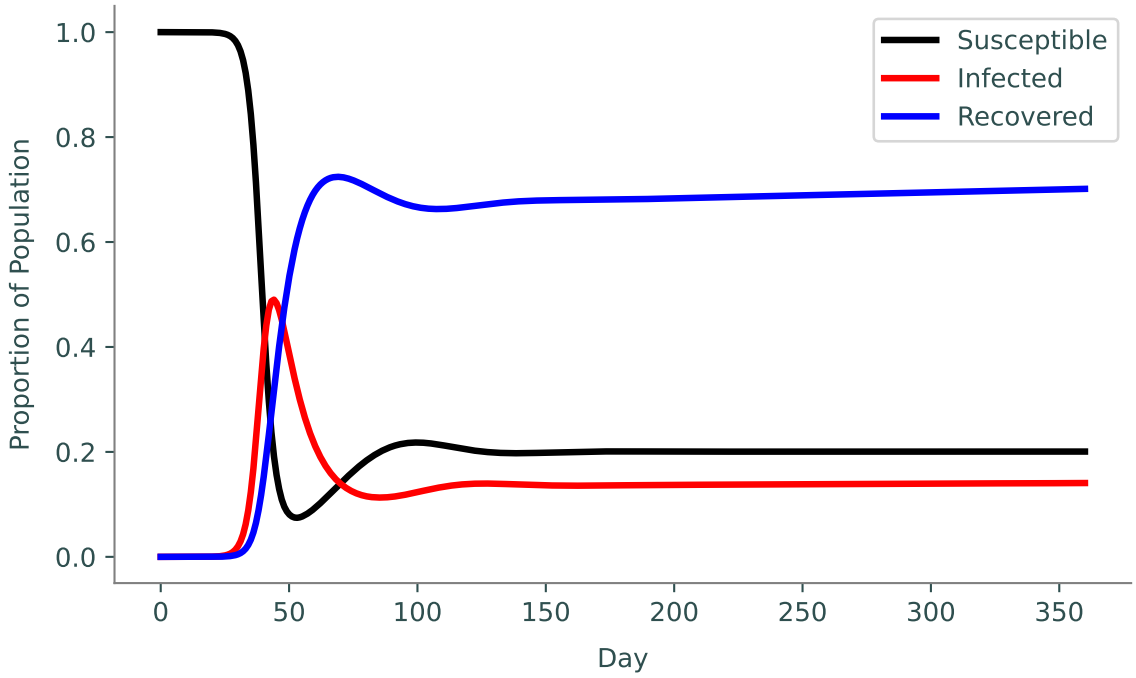


Figure 3.4: Solution to Problem 3.

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), which results in the following SIR type model:

$$\frac{dS}{dt} = -\frac{\alpha}{t} I, \tag{3.8}$$

$$\frac{dI}{dt} = \left(\frac{\alpha}{t} - \frac{1}{T_G}\right) I, \tag{3.9}$$

$$\frac{dR}{dt} = \frac{1}{T_G} I. \tag{3.10}$$

The values α and T_G are the model parameters. The product 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. We will use `scipy.optimize.minimize` to find the parameters that minimize the error between the model and the actual data.

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

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

We will create a function that calculates the error of the model against the data, and pass it into `scipy.optimize.minimize`. This function will accept an array containing all of the parameters, and return a floating-point value. The function `scipy.optimize.minimize` will then return an `OptimizeResult` object, which contains the optimal parameters.

```
# 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 calculate_error(params):
    # Unpack the parameters
    a, b, d = params

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

    # Find the difference between out and the data
    diff = model_prediction - ys

    # Calculate the error
    return np.linalg.norm(diff)

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

# Find the best parameters for this model
result = minimize(calculate_error, p0)

# Get the minimizer
print(result.x)
```

Problem 4. The file `new_york_cases.npy` contains daily case counts for COVID-19 beginning on March 11, 2020. These counts are the total number of people who have been sick at any time up to that point; that is, the sum of the number of people currently infected and the number of people who have recovered, corresponding to $I(t) + R(t)$.

Convert the counts from the file to proportions of New York's population (recall that the total population is 8.399 million people). Fit the PGED model to the COVID-19 data by using `scipy.optimize.minimize` to find values of α and T_G that minimize the difference between the observed proportions and the model's prediction for $I(t)+R(t)$. Unlike the example above, in this problem our model is the system of ODEs (3.8), (3.9), and (3.10) rather than an explicit formula. So, to find `model_output` in the function you pass into `scipy.optimize.minimize`, you will need to use `solve_ivp` to solve the system of ODEs every time the function is called.

Plot the actual data alongside the values of $I(t) + R(t)$ predicted by your model. Print the values of α and T_G you found.

Hint: Set $t_0 = 1$; the PEGD model requires dividing by t, so we must have $t \neq 0$. To pass the values of α and T_G into your ODE function, you can use the argument `args=(alpha, T_G)` inside `solve_ivp`. Use the `t_eval` argument to get the ODE solution values at the correct times.

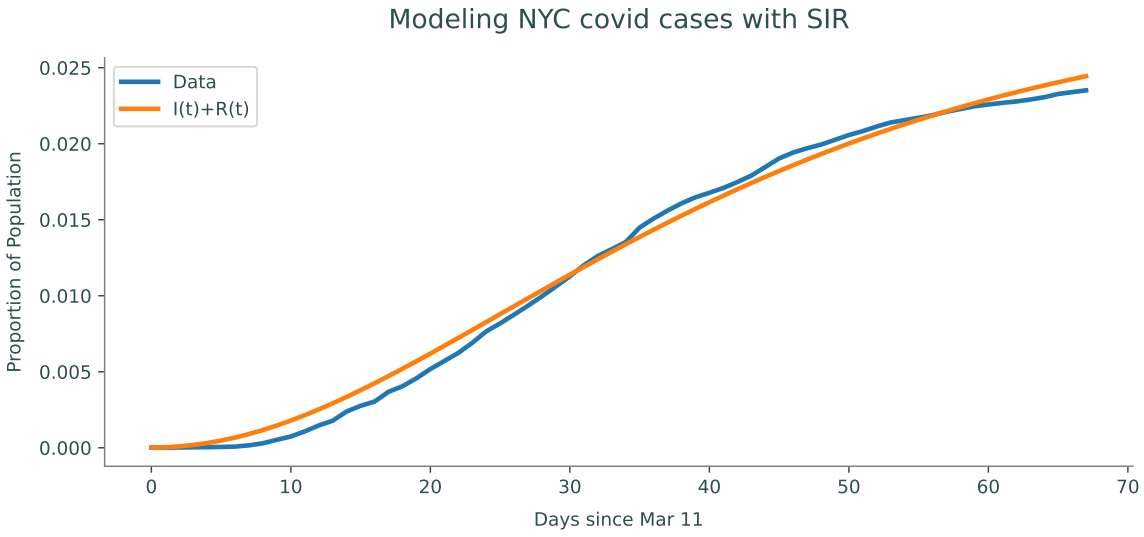


Figure 3.5: Solution to Problem 4

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

$$\begin{bmatrix} S \\ E \\ I \end{bmatrix}' = \begin{bmatrix} \mu - \beta(t)SI \\ \beta(t)SI - E/\lambda \\ E/\lambda - I/\eta \end{bmatrix}, \tag{3.11}$$

$$\begin{aligned} S(0) &= S(1), \\ E(0) &= E(1), \\ I(0) &= I(1) \end{aligned} \tag{3.12}$$

The parameters μ and λ represent the birth rate of the population and the latency period of measles, respectively. The parameter η represents the length of 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 the BVP solver available in SciPy. As a refresher, the code below demonstrates how to use `solve_bvp` to solve the BVP

$$\varepsilon y'' + yy' - y = 0, \quad y(-1) = 1, \quad y(1) = -1/3, \quad \varepsilon = 0.1. \tag{3.13}$$

See Figure 3.6 for the solution.

The BVP solver requires a callable function for the boundary conditions. This function needs to compute the difference between the value of the current guess at the boundary conditions and the desired boundary conditions. In this case, we have one boundary condition on either side. These constraints will evaluate to 0 precisely when the boundary condition is satisfied.

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

epsilon, lbc, rbc = 0.1, 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):
```

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

```
    # The return values will be 0s when the boundary conditions are met exactly
    return np.array([
        ya[0] - lbc, # One boundary condition on the left
         yb[0] - rbc, # One boundary condition on the right
    ])

# The independent variable has size (m,) and goes from a to b
t = np.linspace(-1, 1, 200)
# The initial guess for y must have shape (n,m)
y_guess = np.array([-1/3, -4/3]).reshape((-1, 1))*np.ones((2, len(t)))

# Solve the BVP
solution = solve_bvp(ode, bcs, t, y_guess)
# The returned object has multiple objects. We will use sol, which is a
#     callable function of the solution. We are interested in only y,
#     which is the first row.
y_plot = solution.sol(t)[0]

# You can also instead use sol.x and sol.y. Note that these will not
#     be the same lengths as your initial guesses.

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

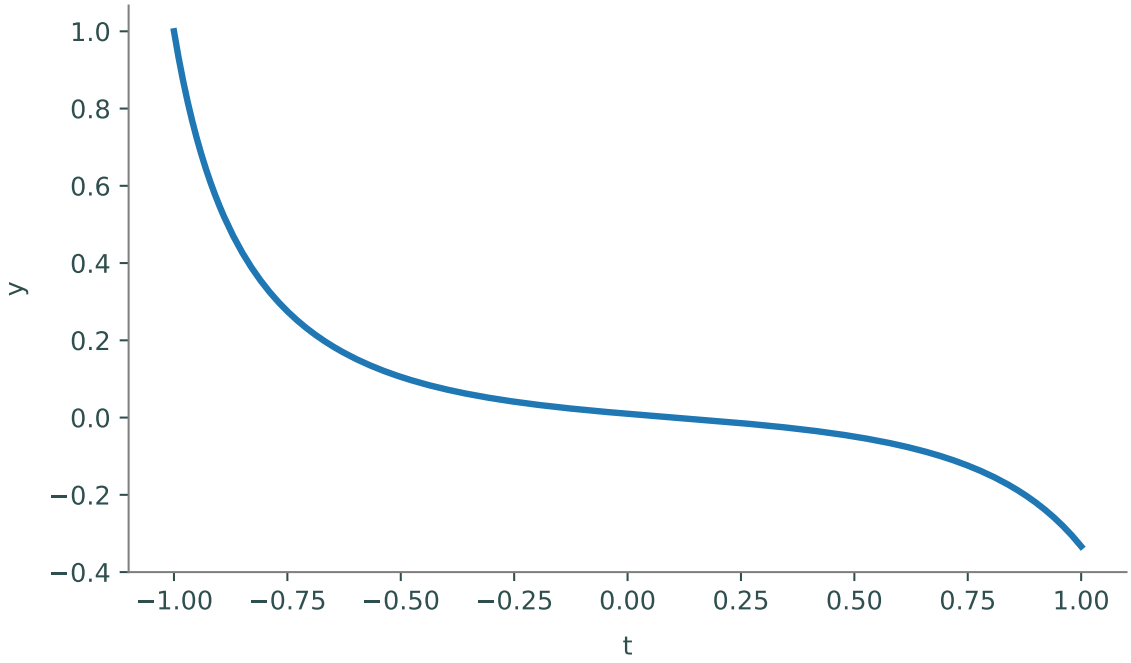


Figure 3.6: Solution to the example given by Equation (3.13)

Problem 5. In this problem we will solve the BVP given by equations (3.11) and (3.12).

Let the periodic function for our measles case be $\beta(t) = \beta_0(1+\beta_1 \cos 2\pi t)$. Use parameters $\beta_1 = 1$, $\beta_0 = 1575$, $\eta = 0.01$, $\lambda = 0.0279$, and $\mu = 0.02$. With these parameter values, time is measured in years, so run the solution over the interval $[0, 1]$ to show a one-year cycle. The boundary conditions in (3.12) are just saying that the year will begin and end in the same state.

Create functions for the ODE and for the boundary conditions. Solve the BVP with the given parameters over a period of one year, and plot the values of S, E, and I. Compare your results with Figure 3.7.

Hint: Use the initial conditions from Figure 3.7 as your initial guess. Remember that the initial infected proportion is small, not 0.

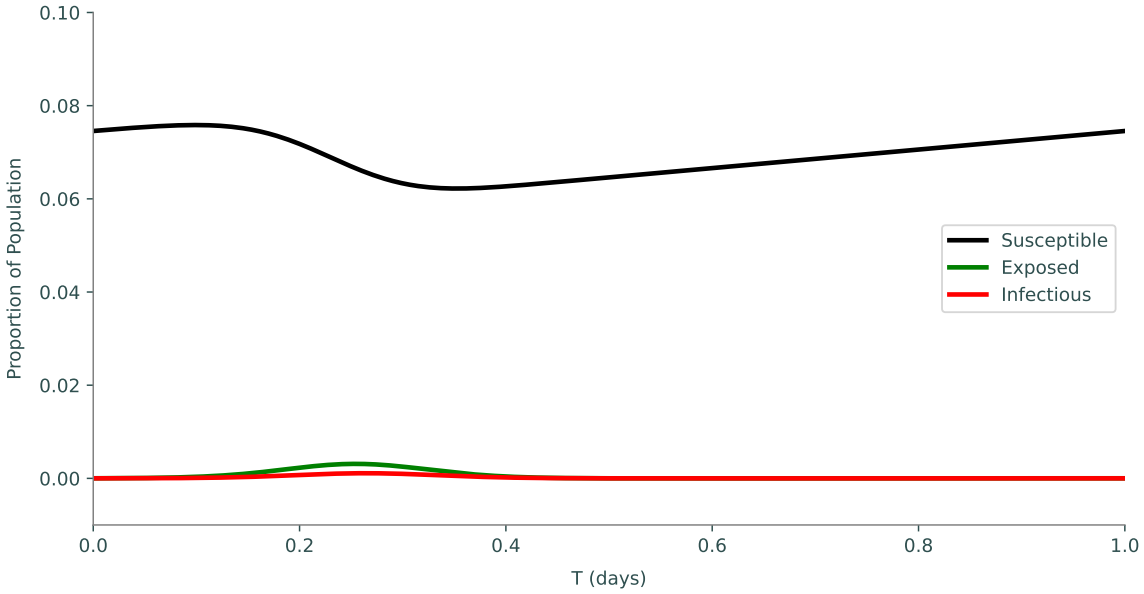


Figure 3.7: Solution to Problem 5

4 Numerical Methods for Initial Value Problems

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)

$$\begin{aligned} \mathbf{x}'(t) &= f(\mathbf{x}(t), t), \quad t_0 \leq t \leq t_f \\ \mathbf{x}(t_0) &= \mathbf{x}_0, \end{aligned} \tag{4.1}$$

where f is a suitably continuous function. A solution of (4.1) 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, (4.1) 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

$$\begin{aligned} x'(t) &= \sin(x(t)), \\ x(0) &= x_0. \end{aligned} \tag{4.2}$$

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 (4.2) 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 4.1 shows several of these approximate solutions, along with some of the constant, or *equilibrium*, solutions.

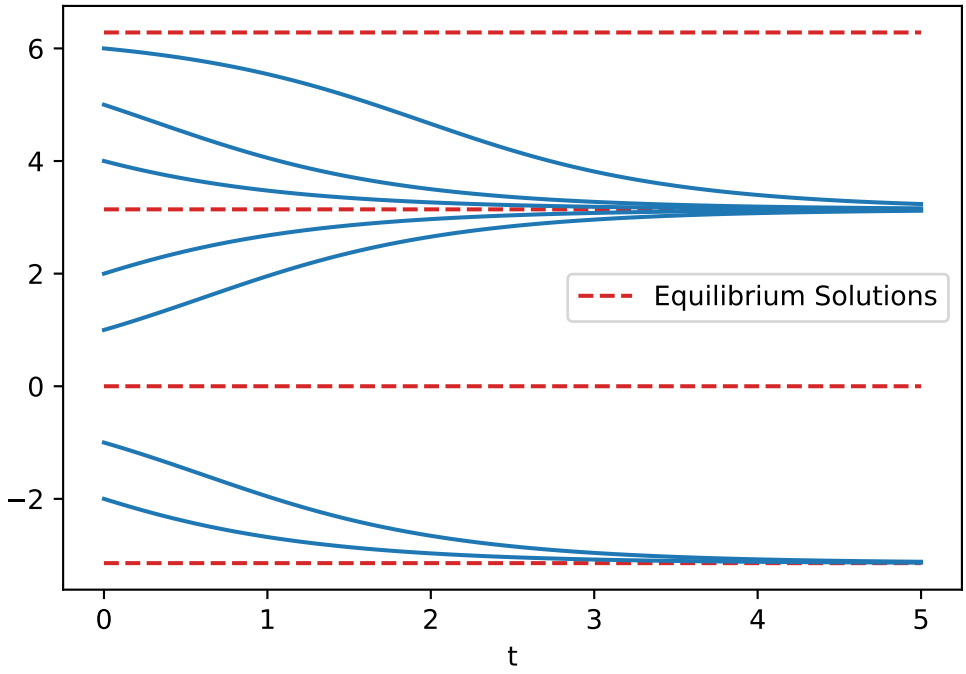


Figure 4.1: Several solutions of (4.2), using `scipy.integrate.odeint`.

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).

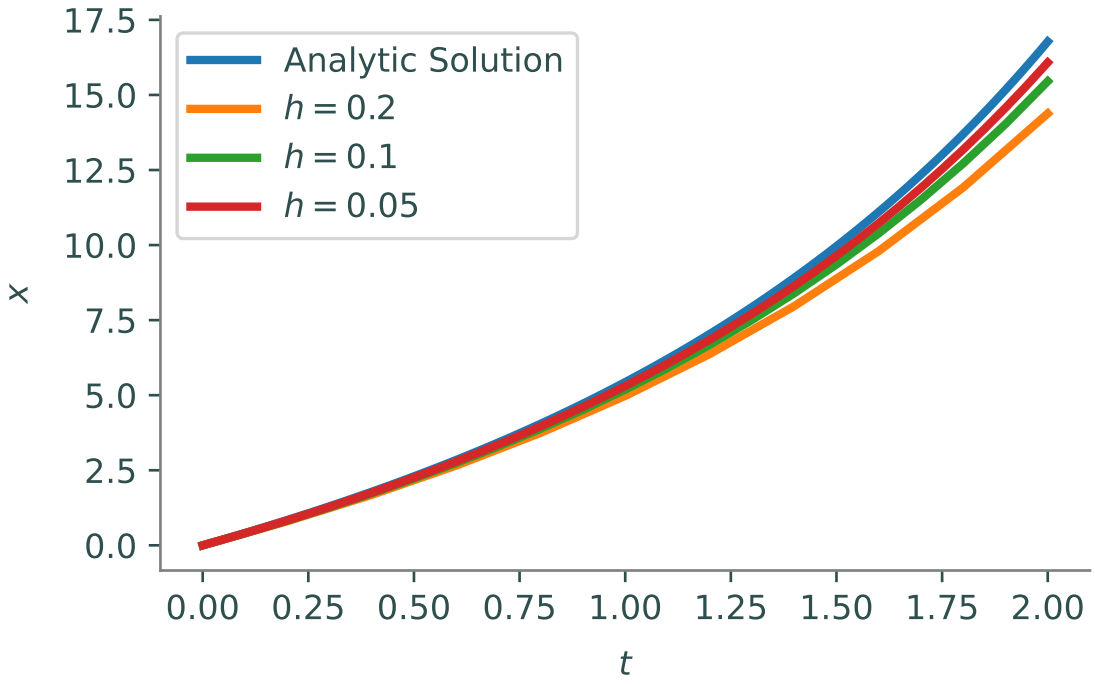


Figure 4.2: The solution of (4.3), alongside several approximations using Euler's method.

Problem 1. Write a function which implements Euler's method for an IVP of the form (4.1). Test your function on the IVP:

$$\begin{aligned} x'(t) &= x(t) - 2t + 4, \quad 0 \le t \le 2, \\ x(0) &= 0, \end{aligned} \tag{4.3}$$

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 4.2.

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 (4.3) 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 4.3.

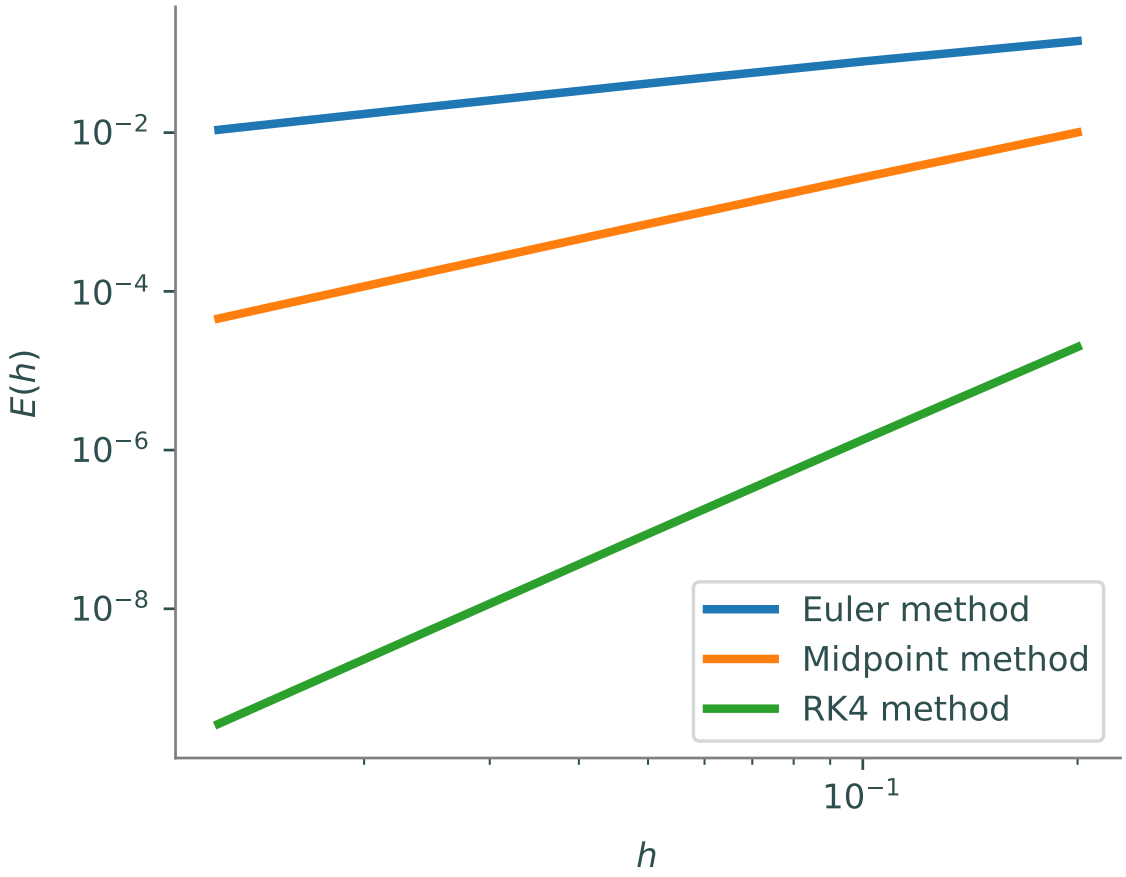


Figure 4.3: Loglog plot of the relative error in approximating $x(2)$, using step sizes $h = 0.2$, 0.1, 0.05, 0.025, and 0.0125. The slope of each line demonstrates the first, second, and fourth order convergence of the Euler, Midpoint, and RK4 methods, respectively.

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

$$\begin{aligned} x'(t) &= x(t)\cos(t), \quad t \in [0, 8], \\ x(0) &= 1. \end{aligned} \tag{4.4}$$

Figure 4.4 illustrates the comparative efficiency of the Euler, Midpoint, and RK4 methods applied to (4.4). 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.

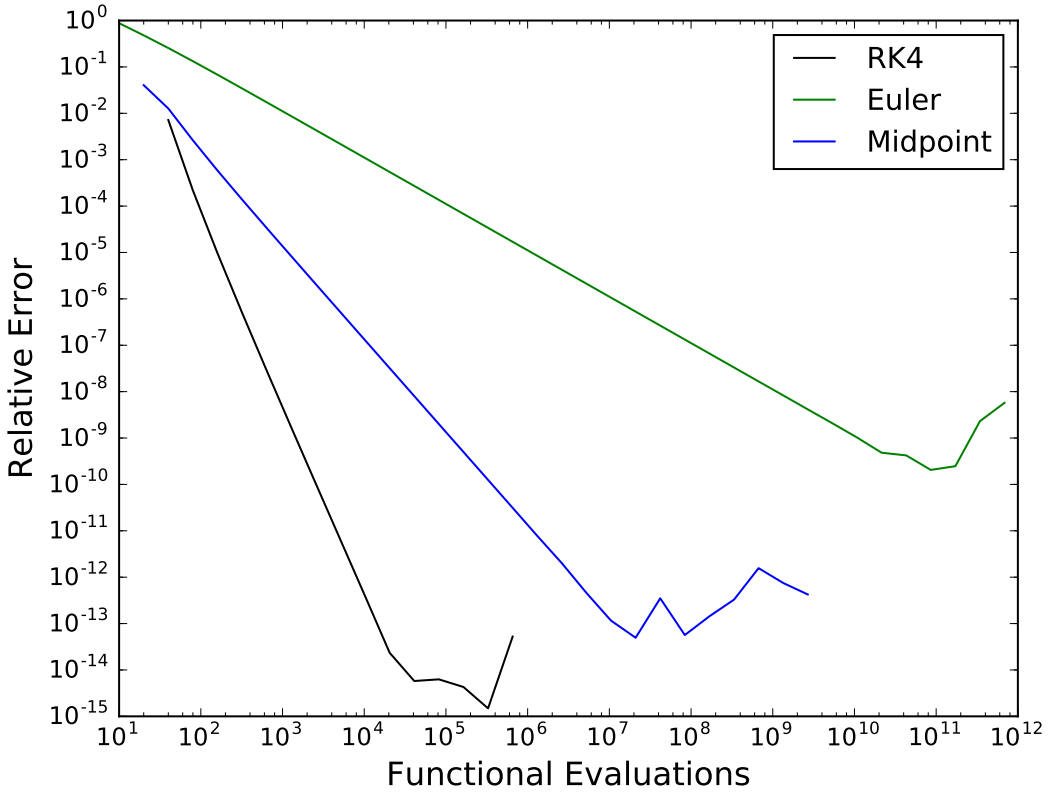


Figure 4.4: The relative error in computing the solution of (4.4) at $x = 8$ versus the number of times the right-hand side of (4.4) must be evaluated.

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

$$\begin{aligned} my'' + ky &= 0, \quad 0 \leq t \leq 20, \\ y(0) = 2, \quad y'(0) &= -1, \end{aligned} \tag{4.5}$$

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.

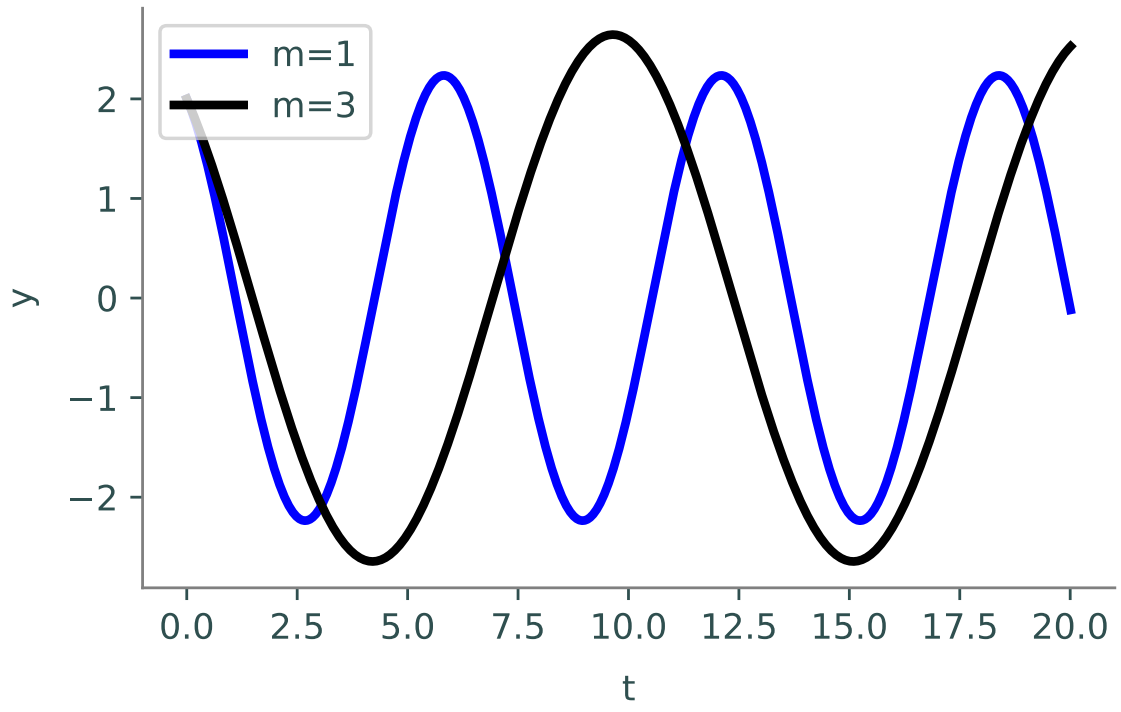


Figure 4.5: Solutions of (4.5) for several values of m.

Damped free harmonic oscillators

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

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

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

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

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

$$my''(t) + ky(t) = F(t). \tag{4.6}$$

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 4.6 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 4.6 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

$$\begin{aligned} &2y'' + \gamma y' + 2y = 2\cos(\omega t), \quad 0 \le t \le 40, \\ &y(0) = 2, \quad y'(0) = -1. \end{aligned} \tag{4.7}$$

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 Figure 4.6.

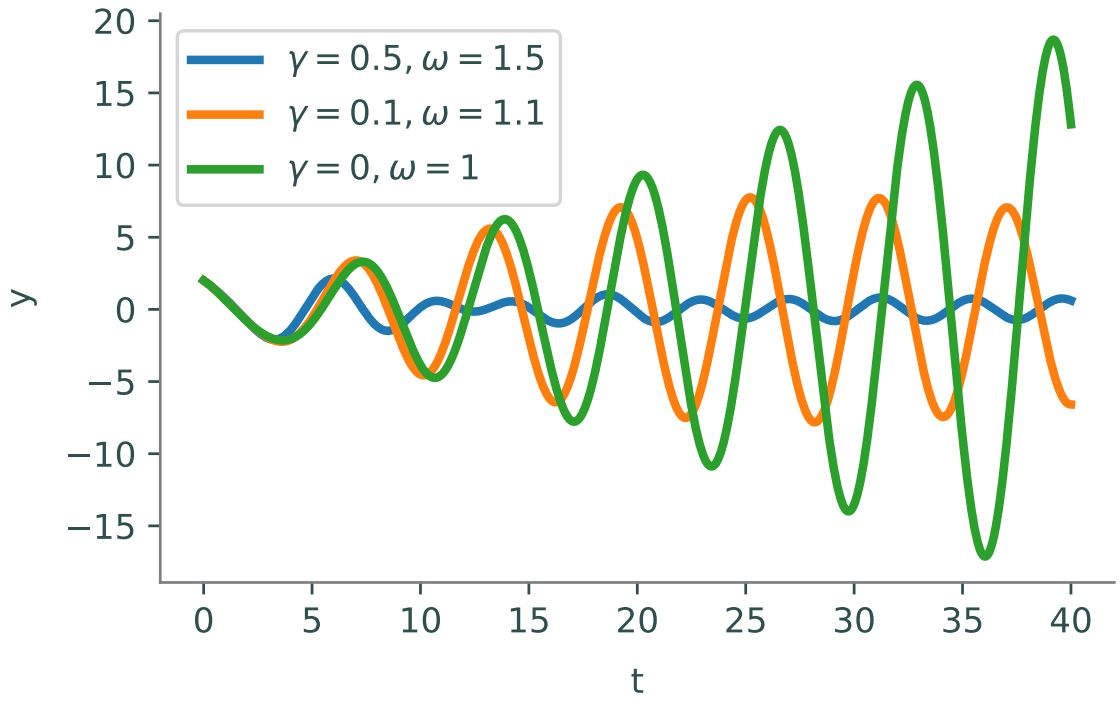


Figure 4.6: Solutions of (4.7) for several values of ω and γ.

5 Predator-Prey Models

Lab Objective: *We apply methods for solving initial value problems to analyzing several dynamical systems between predator and prey populations.*

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 $w(t) = 0$ we have $dr/dt = \alpha r(t)$ for some $\alpha > 0$.
- In the absence of rabbits, the wolves die out. Thus, when $r(t) = 0$ we have $dw/dt = -\delta w(t)$ for some $\delta > 0$.
- 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:

$$\begin{aligned} \frac{dr}{dt} &= \alpha r - \beta rw = r(\alpha - \beta w) \\ \frac{dw}{dt} &= -\delta w + \gamma rw = w(-\delta + \gamma r) \end{aligned} \tag{5.1}$$

Problem 1. Define the function `predator_prey()` that accepts the current time t, the current $r(t)$ and $w(t)$ values as a 1d array y, and the parameters α, β, δ, and γ, and returns the right hand side of (5.1) as a tuple.

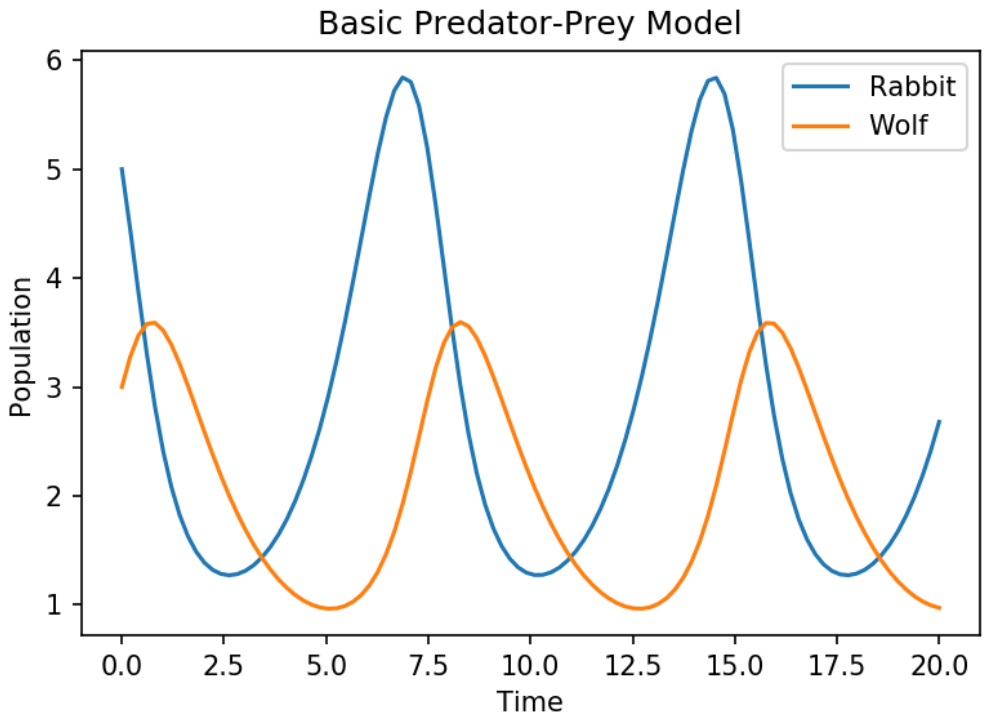


Figure 5.1: The solution to Problem 2 of the system found in (5.1)

Problem 2. Use `solve_ivp` and your function from Problem 1 to solve (5.1) with initial conditions $(r_0, w_0) = (5, 3)$ and time ranging from 0 to 20 years. Use $\alpha = 1.0$, $\beta = 0.5$, $\delta = 0.75$, and $\gamma = 0.25$ as your growth parameters. Display the resulting rabbit and wolf populations over time on the same plot. Your graph should match the graph in Figure 5.1.

To pass parameters into your ODE function, use the `args` argument of `solve_ivp`. For example:

```
# Pass the arguments into solve_ivp here:
t = np.linspace(t0, tf, t_steps)
solve_ivp(predator_prey, t_span, y0, t_eval=t,
                args=(alpha, beta, gamma, delta))
```

The populations are stored as rows in the attribute `y` of the output of `solve_ivp`.

Variations on the Predator-Prey

The Lotka-Volterra model

The representation of the predator-prey relationship found in (5.1) 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

$$\begin{aligned}\frac{dU}{d\bar{t}} &= U(1 - V),\\ \frac{dV}{d\bar{t}} &= \eta V(U - 1).\end{aligned} \tag{5.2}$$

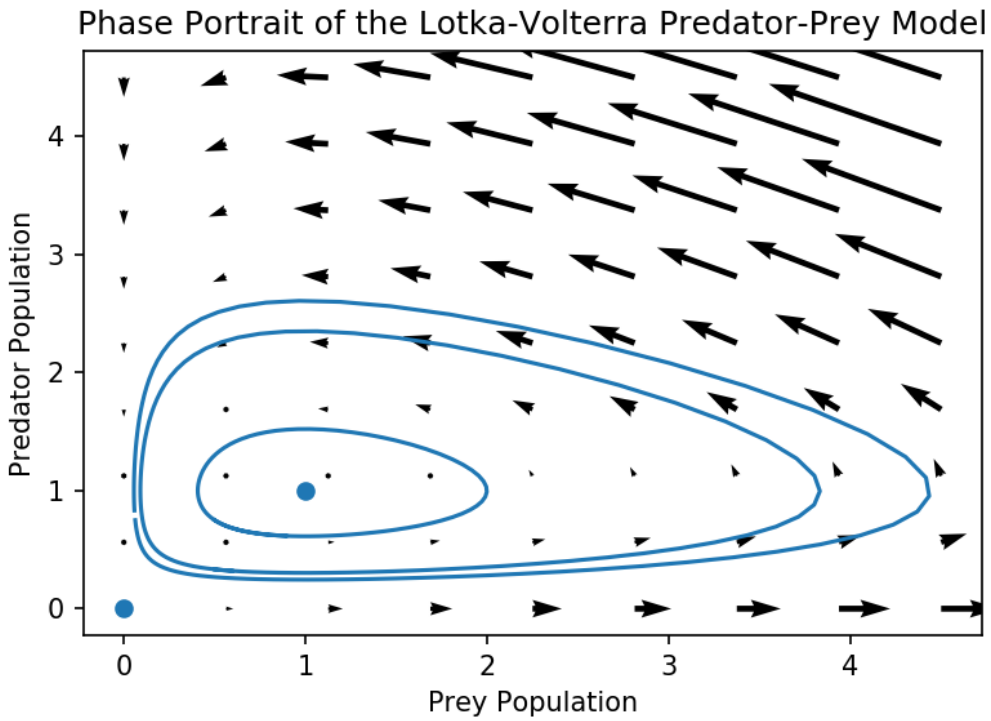


Figure 5.2: The phase portrait for the nondimensionalized Lotka-Volterra predator-prey equations with parameters $\eta = 1/3$.

Problem 3. Similar to Problem 1, define a function `lotka_volterra()` that takes in the current time t, the current predator and prey populations as a 1d array y, and the growth parameter η, and returns the right hand side of the nondimensional Lotka-Volterra system (5.2).

Plot the phase portrait and several solutions of (5.2) for $\eta = 1/3$. Using `solve_ivp`, solve (5.2) 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. Also plot the equilibria $(0, 0)$ and $(1, 1)$ as points.

The following three lines of code can be used to plot the phase portrait of (5.2):

```
Y1, Y2 = np.meshgrid(np.linspace(0, 4.5, 9), np.linspace(0, 4.5, 9))
dU, dV = lotka_volterra(0, (Y1, Y2), eta)
Q = plt.quiver(Y1, Y2, U, V)
```

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 Figure 5.2.

The Logistic model

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

$$\begin{aligned} \frac{dU}{d\bar{t}} &= U(1 - U - V), \\ \frac{dV}{d\bar{t}} &= \eta V(U - \rho). \end{aligned} \tag{5.3}$$

Problem 4. Define a new function `logistic_model()` that takes in the current time t, the current predator and prey populations y, the parameters η and ρ, and returns the right hand side of the nondimensional logistic model (5.3) as a tuple.

Use `solve_ivp` to compute solutions (U, V) of (5.3) for initial conditions $(1/3, 1/3)$ and $(1/2, 1/5)$ with $(t_0, t_f) = (0, 13)$. Do this for parameter values $\eta = 1$, $\rho = 0.3$ and also for values $\eta = 1$, $\rho = 1.1$.

Create a phase portrait for the logistic equations for each set of parameter values. Plot the direction field, all equilibrium points, and both solution orbits on the same plot for each set of parameter values.

Hint: there are three equilibrium points for each set of parameter values. However, you only need to plot the ones with nonnegative values of U and V, as these are the only ones that correspond to physically-meaningful solutions.

Competition between Prey

One interesting extension we can consider is what happens if we have multiple species of prey competing for the resources of their environment as well as are hunted by predators. Supppose we now wish to model the populations of rabbits (prey), elk (also prey), and wolves in Yellowstone. For simplicity, we will assume that the two prey populations share strictly the same resources, and can support up to either K_1 rabbits or K_2 elk. Let u be the population of rabbits, v the population of elk, and w the population of wolves. Expanding on the logistic model, we can model their populations as follows:

$$\begin{aligned}
\frac{du}{dt} &= \alpha_1 u\left(1 - \frac{u}{K_1} - \frac{v}{K_2}\right) - \beta_1 wu \\
\frac{dv}{dt} &= \alpha_2 v\left(1 - \frac{u}{K_1} - \frac{v}{K_2}\right) - \beta_2 wv \\
\frac{dw}{dt} &= -\delta w + \gamma_1 wu + \gamma_2 wv.
\end{aligned}$$

We again perform dimensional analysis to reduce the number of parameters. Let $U = \frac{u}{K_1}$, $V = \frac{v}{K_2}$, $W = \frac{\beta_1}{\alpha_1} w$, and $\bar{t} = \alpha_1 t$, and define new parameters as $\eta = \frac{\gamma_1 K_1}{\alpha_1}$, $\rho = \frac{\delta}{\gamma_1 K_1}$, $\xi = \frac{\gamma_2 K_2}{\gamma_1 K_1}$, $\alpha = \frac{\alpha_2}{\alpha_1}$, and $\beta = \frac{\beta_2}{\beta_1}$. Then, the nondimensional equations for two prey species are

$$\begin{aligned}
\frac{dU}{d\bar{t}} &= U(1 - U - V - W), \\
\frac{dV}{d\bar{t}} &= \alpha V(1 - U - V) - \beta VW, \\
\frac{dW}{d\bar{t}} &= \eta W(U + \xi V - \rho).
\end{aligned} \tag{5.4}$$

Problem 5. Define a new function `two_prey_species()` that takes in the current time t, the current prey and predator populations y, the parameters α, β, η, ξ, and ρ, and returns the right-hand-side of the nondimensional two prey species system (5.4) as a tuple.

Use `solve_ivp` to compute solutions (U, V, W) of (5.4) using the three initial condition $(1/3, 1/3, 1/3)$, $(1/2, 1/3, 1/5)$, and $(1, 1/10, 1/2)$, with $(t_0, t_f) = (0, 25)$. Use parameter values $\eta = 1$, $\rho = 0.3$, $\xi = 0.5$, $\alpha = 0.2$, $\beta = 0.1$. Plot the numerical solutions for the populations as functions against time.

Do the dynamics predicted by this model seem realistic? Write (in a markdown cell) your answer and reasoning behind it.

6 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 characterization 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

$$\begin{aligned}\frac{dx}{dt} &= \sigma\,(y - x)\\ \frac{dy}{dt} &= \rho x - y - xz\\ \frac{dz}{dt} &= xy - \beta z\end{aligned}$$

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. This began the study of what was called a *strange attractor*. Although relatively simple, this system 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 $[-15, 15]$ and for t in the range $[0, 25]$. As usual, use `scipy.integrate.solve_ivp` to compute an approximate solution. Compare your results with Figure 6.1.

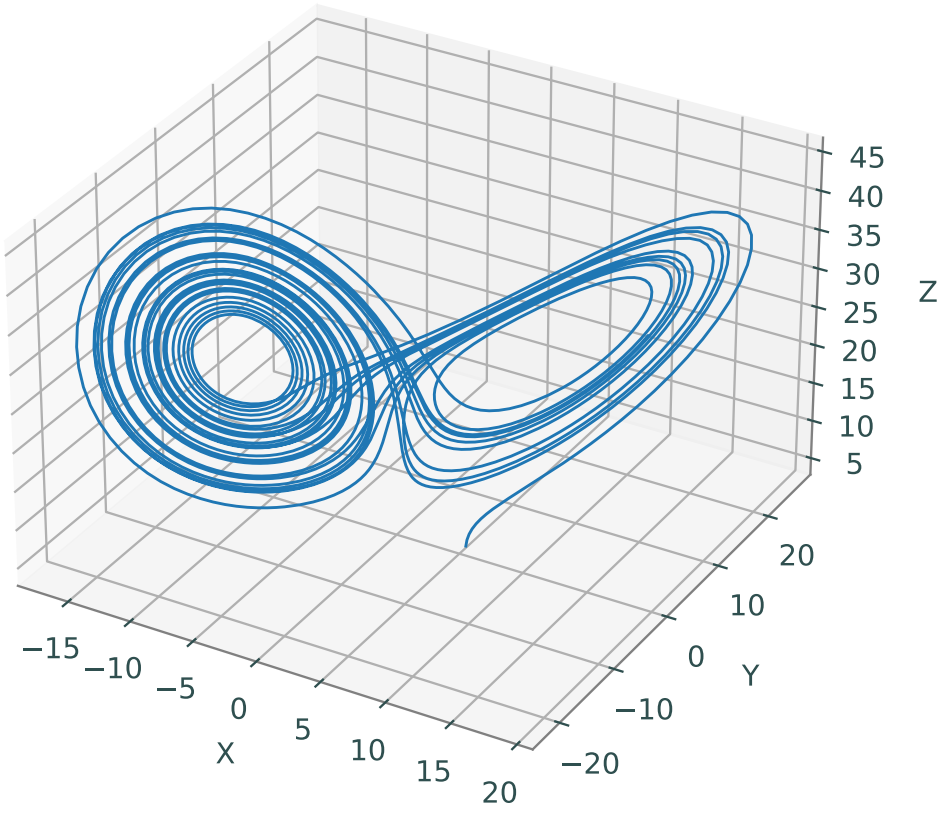


Figure 6.1: Approximate solution to the Lorenz equation with random initial conditions

Basin of Attraction

Notice in the first problem that the solution tended to a certain region, called an *attractor*. The *basin of attraction* of an attractor is the set of initial conditions that tend towards the attractor. We will investigate the basin of attraction of the Lorenz system by changing 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 6.2.

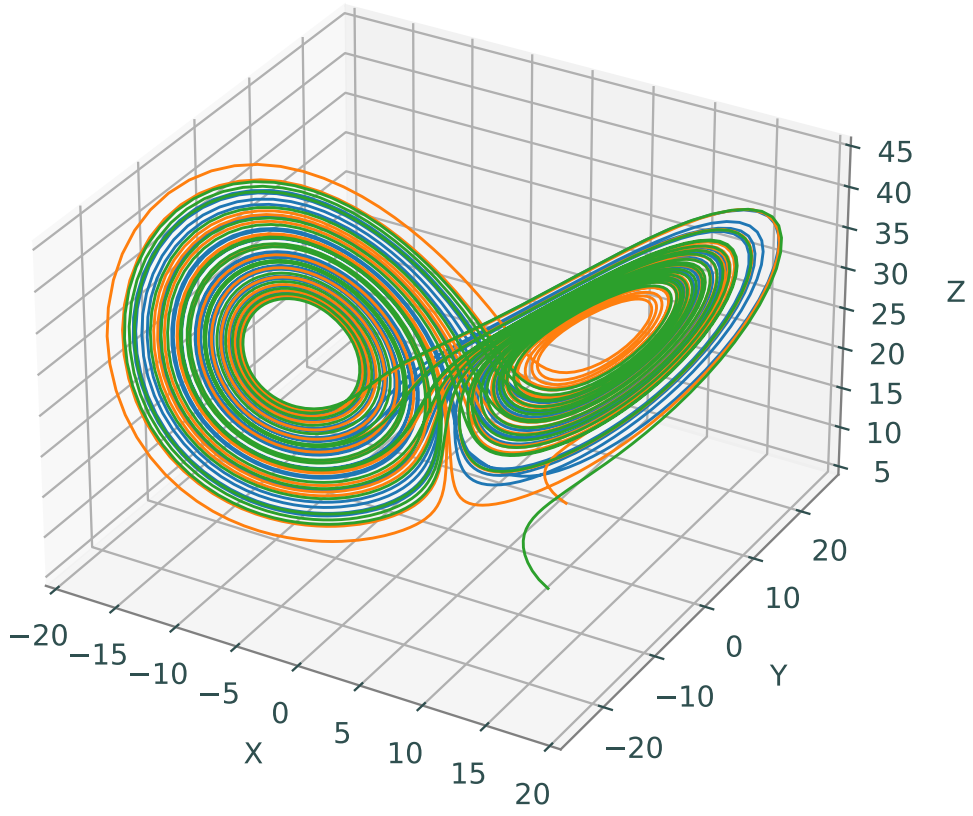


Figure 6.2: Multiple solutions to the Lorenz equation with random initial conditions

Chaos

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

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 (try using `np.random.randn(3)*(1e-8)` for the perturbation). Note that it may take several seconds before the separation between the two solutions will be noticeable. Evaluate on `t_span=(0,50)` and use the `t_eval` argument of `solve_ivp`, with `t_eval=np.linspace(0,50,3000)`. Using this argument causes `solve_ivp` to return the solution's values at the points you pass in.

The animation should display a point marker as well as the past trajectory curve for each solution. Save your animation as `lorenz_animation1.mp4` and embed it into the notebook.

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. Even differences due to small numerical errors can cause solutions of chaotic systems to diverge from each other. The `solve_ivp` 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 `solve_ivp` 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` and embed it into the notebook. Use the same `t_span` and `t_eval` arguments as in problem 3.

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.

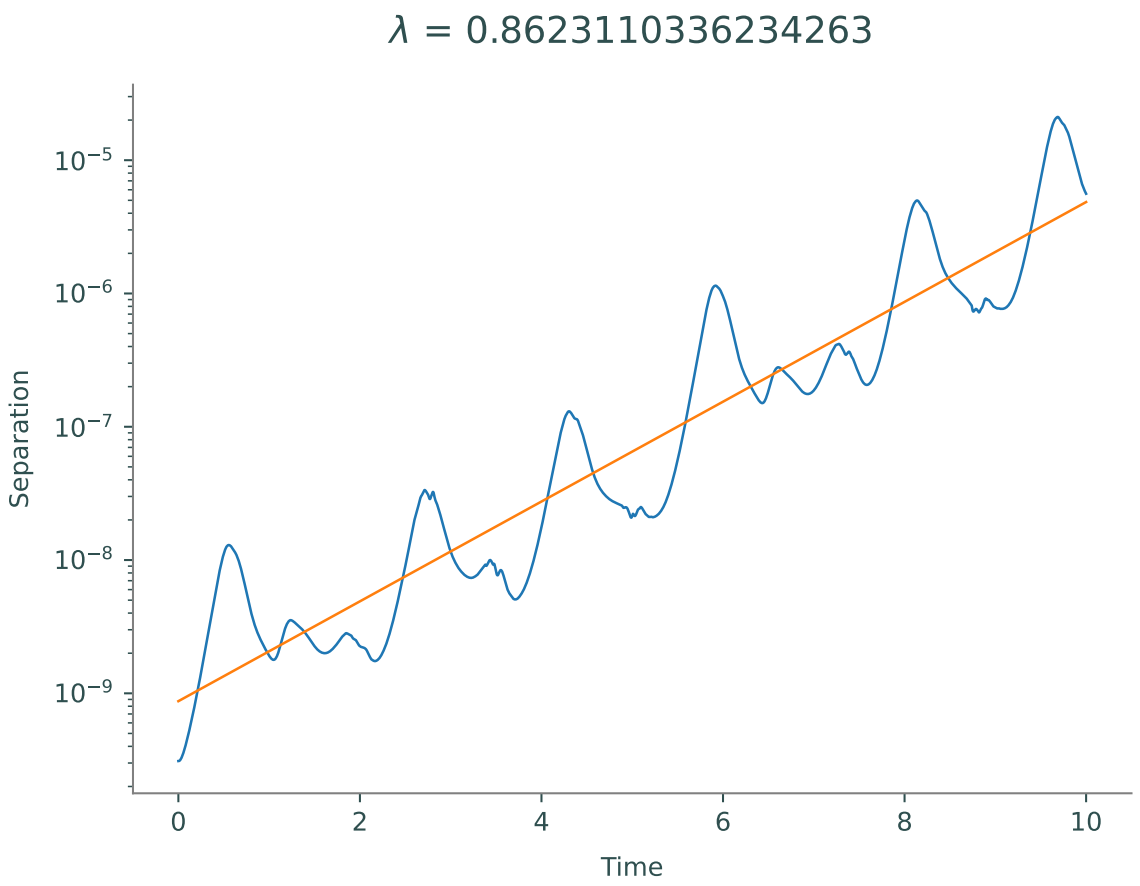


Figure 6.3: A semilog plot of the separation between two solutions to the Lorenz equations together with a fitted line that gives an estimate of the Lyapunov exponent of the system.

- Produce a second initial condition by adding a small perturbation to the first (as before).
- For both initial conditions, use `solve_ivp` 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.

Print your approximation of λ, and produce a plot similar to Figure 6.3 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))`. Use the `t_eval` argument of `solve_ivp` (as in Problem 3) for both solutions to the Lorenz equation so that you can compute $\|\delta(t)\|$ for the same time steps.

7 Bifurcations and Hysteresis

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

$$x' = F(x), \quad x' \coloneqq \frac{d}{dt}x(t). \tag{7.1}$$

Many interesting applications and physical phenomena can be modeled using ODEs. Given a mathematical model of the form (7.1), 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 (7.1) 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

$$x' = F(x, \lambda). \tag{7.2}$$

The parameter λ can then be tuned to better fit the physical application. As λ varies, the equilibrium solutions and other geometric features of (7.2) 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

$$x' \quad = \quad x^2 + \lambda. \tag{7.3}$$

For $\lambda > 0$ equation (7.3) 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 7.1

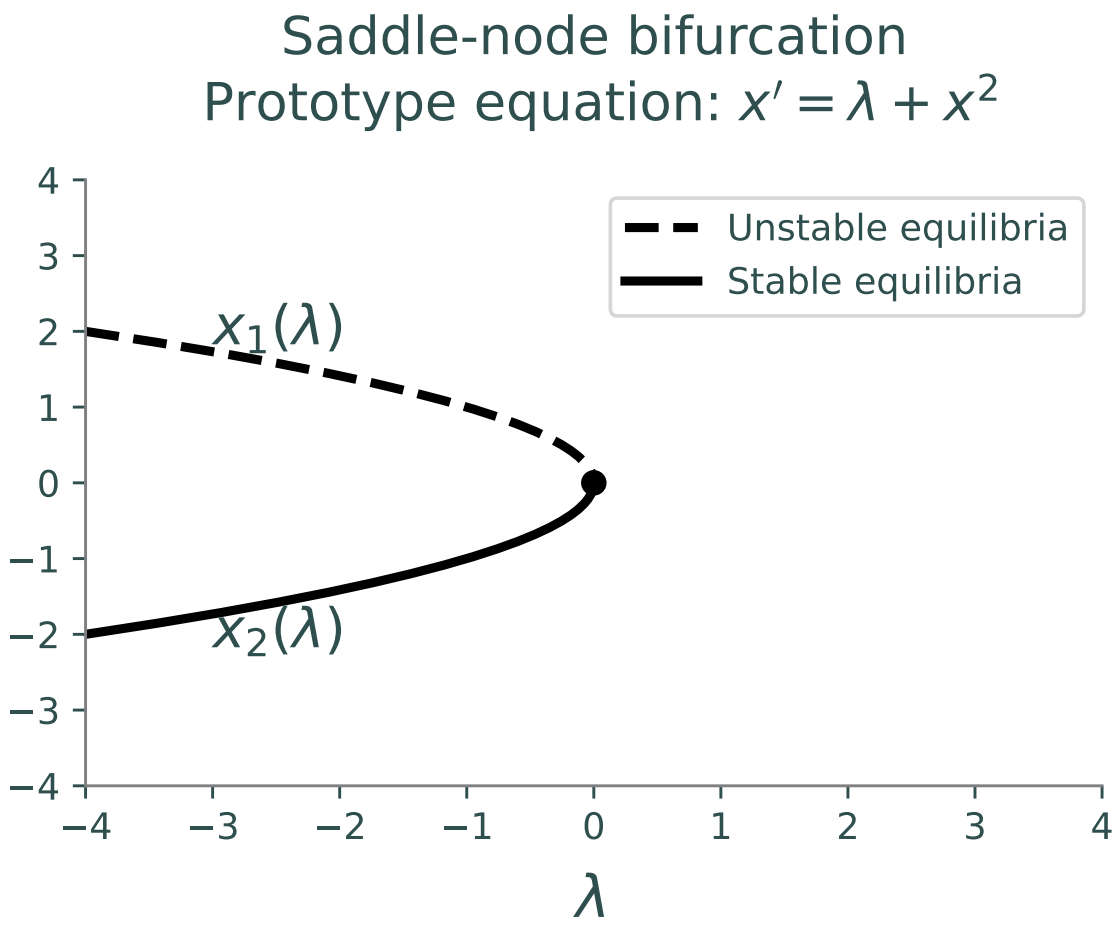


Figure 7.1: Bifurcation diagram for the equation $x' = \lambda + x^2$.

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 (7.3). Notice that this algorithm needs a good initial guess for x_0 to get started.

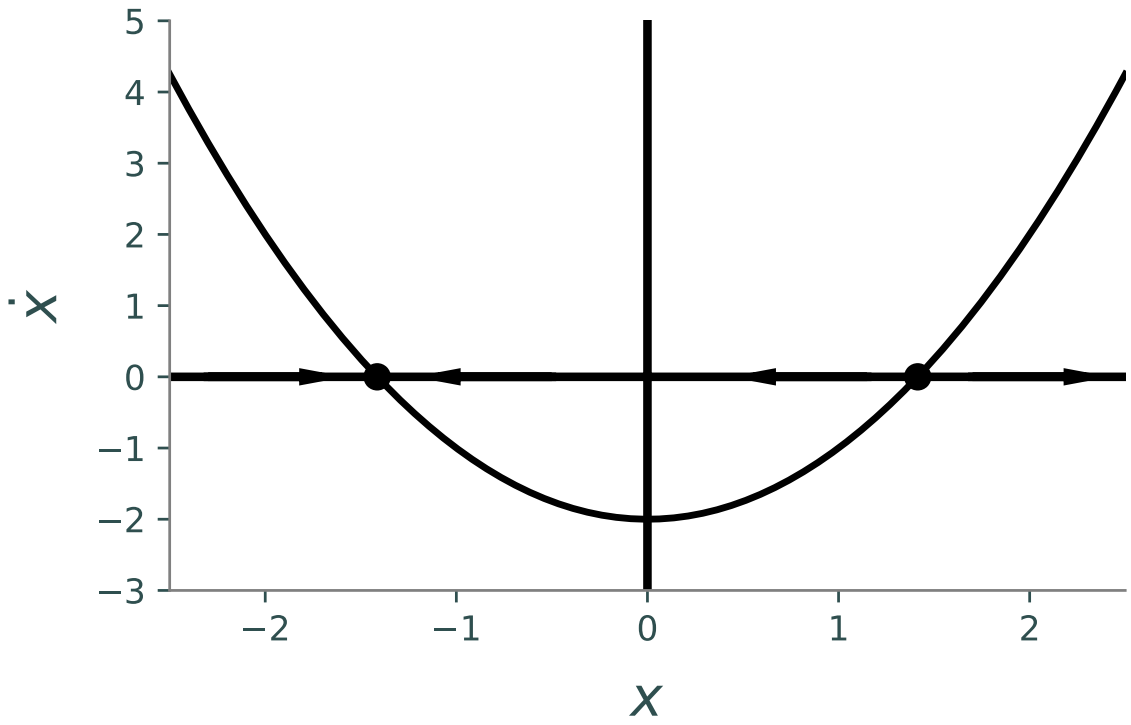


Figure 7.2: Phase Portrait for the equation $x' = -2 + x^2$.

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

def embedding_alg(lams, guess, F):
    xs = list()
    for lam in lams:
        try:
            # Solve for x_value making F(x_value, param) = 0.
            x_value = newton(F, guess, args=(param,), tol=1e-7)
            # Record the solution and update guess for the next iteration.
            xs.append(x_value)
            guess = x_value
        except RuntimeError:
            # If Newton's method fails, return a truncated list of parameters
            # with the corresponding x values.
            return lams[: len(xs)], xs
    # Return the list of parameters and the corresponding x values.
    return lams, xs

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

```
# Top curve shown in the bifurcation diagram
lams1, xs1 = embedding_alg(np.linspace(-5, 0, 200), np.sqrt(5), F)
# The bottom curve
lams2, xs2 = embedding_alg(np.linspace(-5, 0, 200), -np.sqrt(5), F)
```

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

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

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 all of which must be produced using the `embedding_alg` function. Any hard coding will result in an automatic 0. See Figure 7.3.

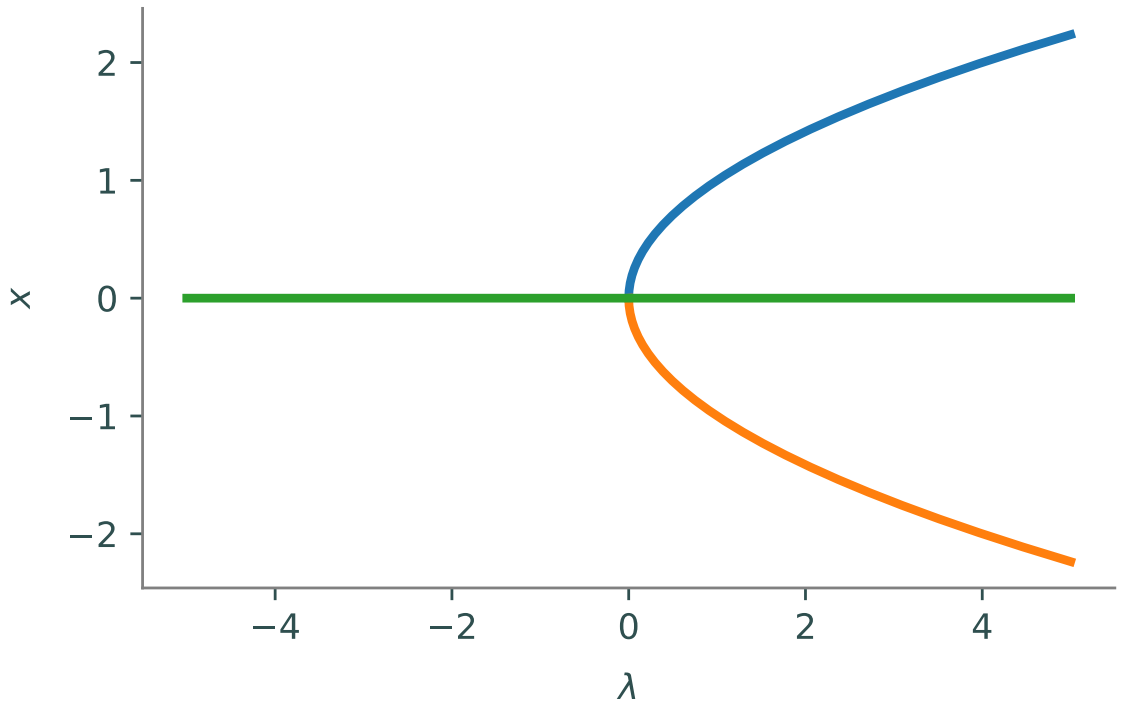


Figure 7.3: Bifurcation diagram for the equation $x' = \lambda x - x^3$.

Problem 2. Another useful tool for analyzing a bifurcation diagram can be to plot the trajectory of the solutions, given different parameters and initial conditions, such that the parameters chosen are from each partition of the bifurcation diagram. For example, the points

$$(\lambda, x_0) \in \left\{ \left(\frac{1}{2}, \frac{1}{2}\right), \left(\frac{1}{2}, \frac{-1}{2}\right), \left(\frac{-1}{2}, \frac{1}{2}\right), \left(\frac{-1}{2}, \frac{-1}{2}\right) \right\}$$

all lie in different parts of the bifurcation diagram in Figure 7.3. We can pick a λ and x_0 and find the trajectory of that points, given the ODE

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

and the initial condition $x(0) = x_0$. Use the four parameter value and initial condition combinations above to solve the ODE

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

Use `solve_ivp` to plot on the time interval $0 \leq t \leq 20$. Be sure to include a legend. See Figure 7.4.

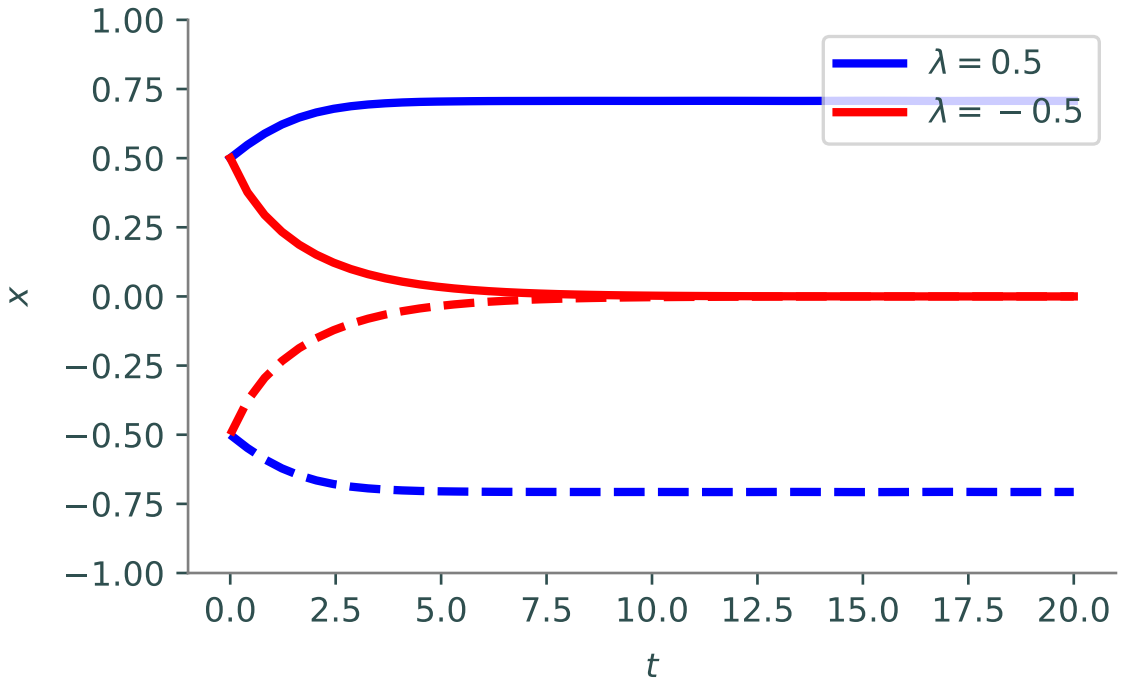


Figure 7.4: Possible trajectories given different x_0's and λ's for the equation $x' = \lambda x - x^3$.

Hysteresis

The following ODE exhibits an interesting bifurcation phenomenon called hysteresis:

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

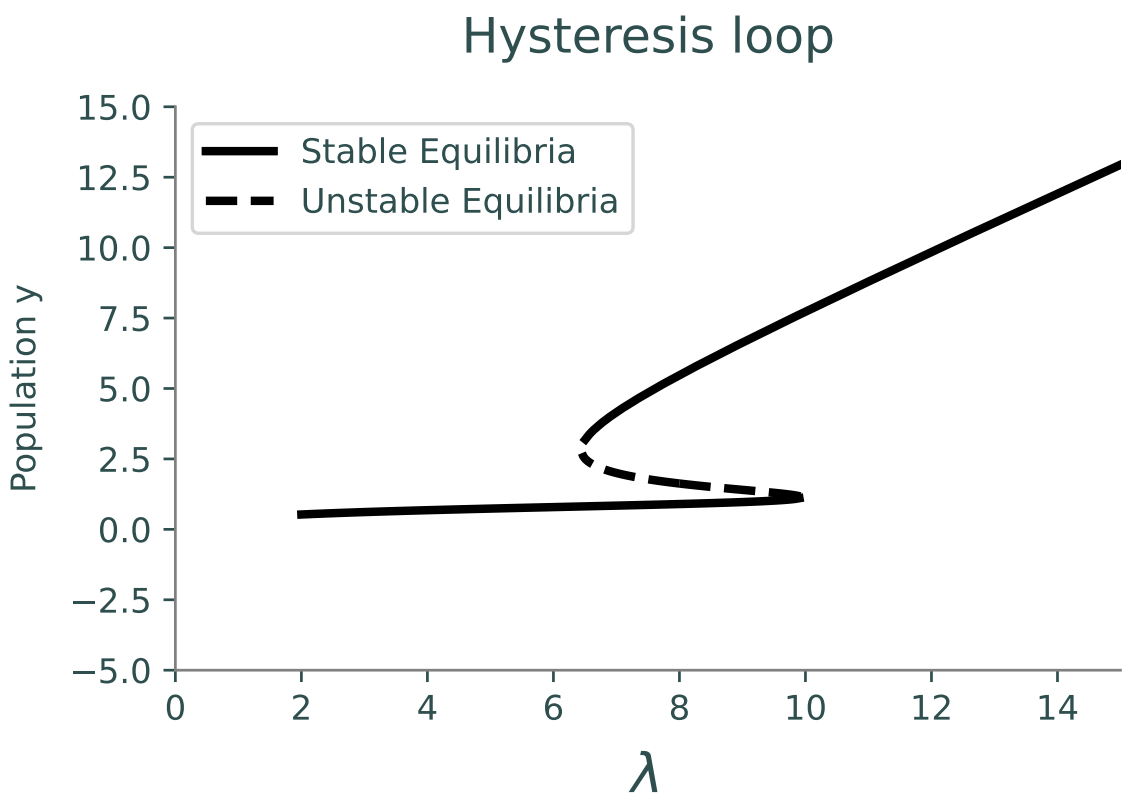


Figure 7.5: Bifurcation diagram for the ODE $x' = \lambda + x - x^3$.

This system has a bifurcation diagram containing what is known as a hysteresis loop, shown in Figure 7.5. 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

$$N' = RN\left(1 - \frac{N}{K}\right) - p(N). \tag{7.4}$$

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 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 $P(N) = \frac{BN^2}{A^2+N^2}$.

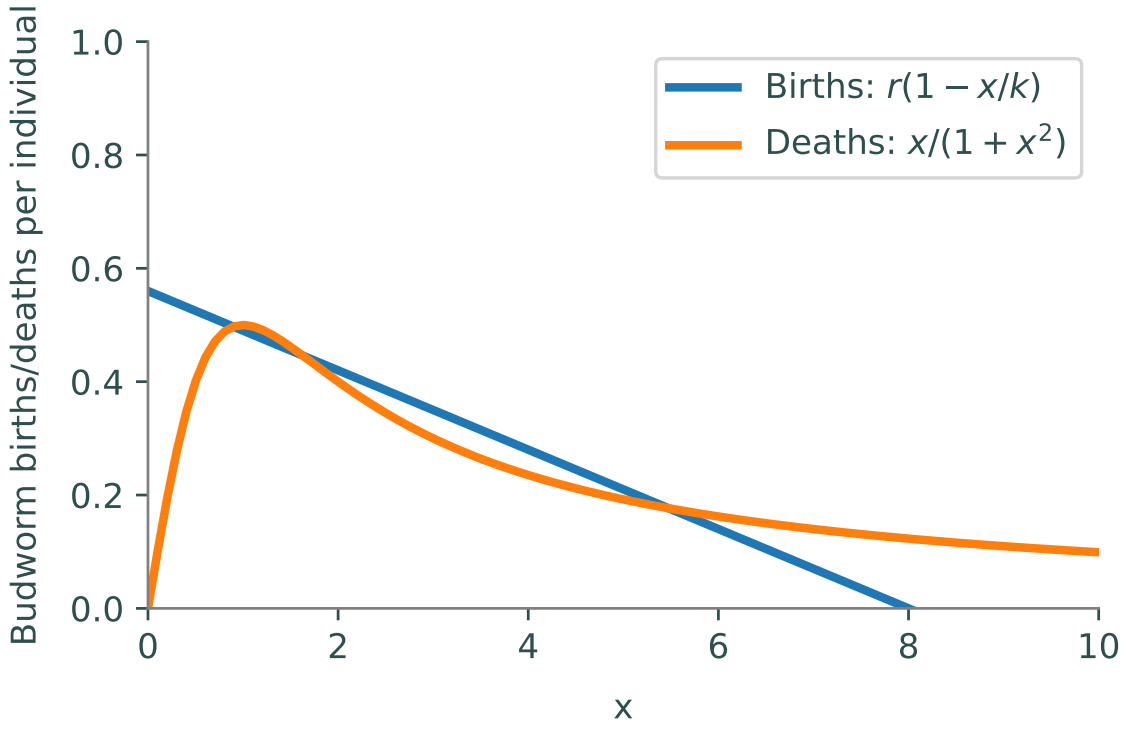


Figure 7.6: Graphical demonstration of nonzero equilibrium solutions for the budworm population (here $r = .56,\ k = 8$); equilibrium solutions occur where the curves cross. As k increases, the line $y = r(1 - x/k)$ gets more shallow and the number of solutions goes from one to three and then back to one.

Before studying the equilibrium points of (7.4) 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

$$\frac{dx}{d\tau} = rx(1 - x/k) - \frac{x^2}{1 + x^2}. \tag{7.5}$$

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 (7.6) ($k = 8$ in the figure).

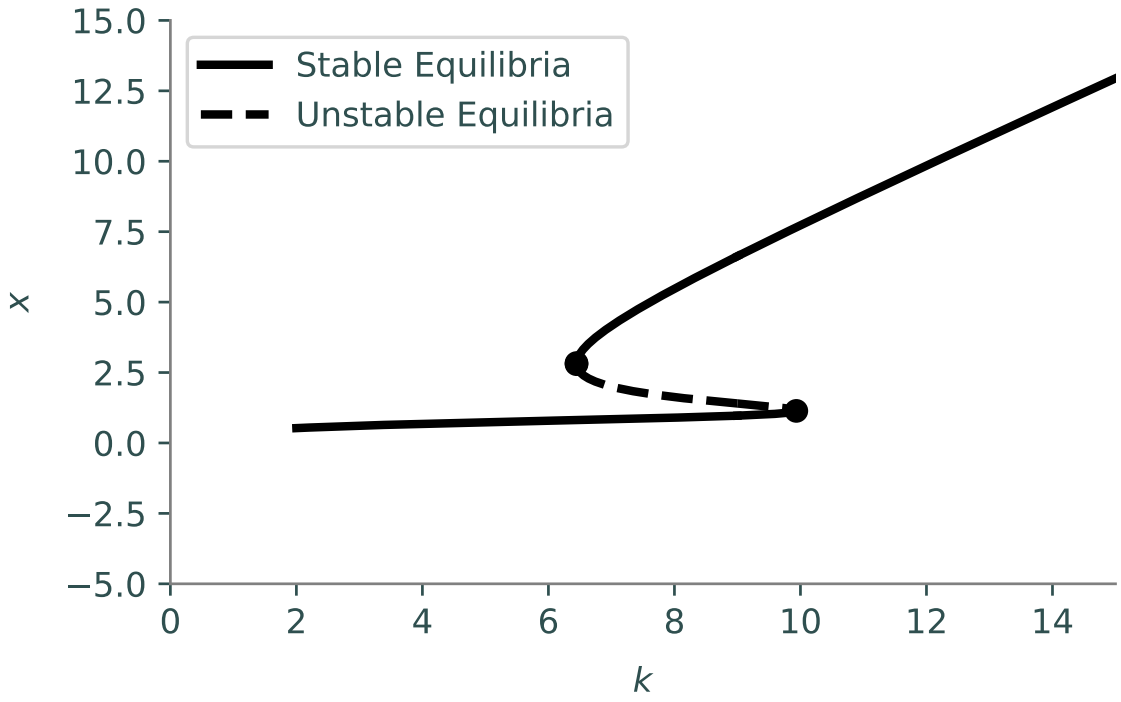


Figure 7.7: Bifurcation diagram for the budworm population model. The parameter r is fixed at 0.56. The lower stable branch is known as the refuge level of the bundworm population, while the upper stable branch is known as the outbreak level. Once the budworm population reaches an outbreak level, the 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.

Problem 3. Reproduce the bifurcation diagram (7.5) for the differential equation

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

where $r = 0.56$. Be sure to include a legend.

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 from the previous problems 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 7.7, but you will have to run the embedding algorithm 4-6 times to get every part of the plot. In order to make a black dashed line, add `'--k'` as the third argument in plt.plot() and use `'-k'` as the third argument in plt.plot() to get the solid black line.

Problem 4. Assume a time-dependent carrying capacity, defined by

$$k(t) = \begin{cases} 8 & t \in [0, 60) \\ 12 & t \in [60, 150) \\ 8 & t \in [150, 220) \\ 6 & t \in [220, 300) \end{cases}$$

and assume $r = 0.56$ and $x(0) = x_0 = 0.3$. Plot the state-space diagram for the differential equation

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

using `solve_ivp` to solve the differential equation. Also plot the carrying capacity as a function of time on the same axes. Be sure to include a legend.

Notice that returning the parameter k to 8 does not make the budworm population decrease back to the value it approached before increasing k to 12.

Two-Dimensional State Space

The *Hopf bifurcation* is one type of bifurcation that only occurs in a two- or higher-dimensional space.[1] Examples include railway vehicle stability, in which a vehicle's motion transitions from stable to unstable as speed increases. There are two types of Hopf bifurcations: *supercritical* and *subcritical*. In polar coordinates, a simple supercritical system can take the form

$$\begin{aligned} r' &= \mu r - r^3 \\ \theta' &= \omega + br^2. \end{aligned} \tag{7.6}$$

We will be examining the effect of μ on the stability of the origin. We restrict our analysis with the convention that $r \geq 0$. At the origin ($r = 0$), we have $r' = 0$. This suggests that the origin is an equilibrium, and a conversion to Cartesian coordinates confirms this (see the "Additional Material" section).

For $r > 0$, $\mu \leq 0$ implies that $r' < 0$, so the origin is in fact *asymptotically stable*. However, something interesting happens when μ becomes positive. In that case, we find that for $r < \sqrt{\mu}$ we have $r' > 0$, so the origin is unstable. On the other hand, for $r > \sqrt{\mu}$ we have $r' < 0$. This means that all trajectories not on the circle $S = \{r : r = \sqrt{\mu}\}$ tend toward S. Moreover, on S we have $r' = 0$, so trajectories on the circle will stay on it.

It turns out S is a special case of a *stable* limit cycle, defined by the property that nearby trajectories approach the cycle as time goes to infinity. (The circle S is special in that it attracts trajectories globally, rather than just in a neighborhood.)

[1] `https://www.biodyn.ro/course/literatura/Nonlinear_Dynamics_and_Chaos_2018_Steven_H._Strogatz.pdf`

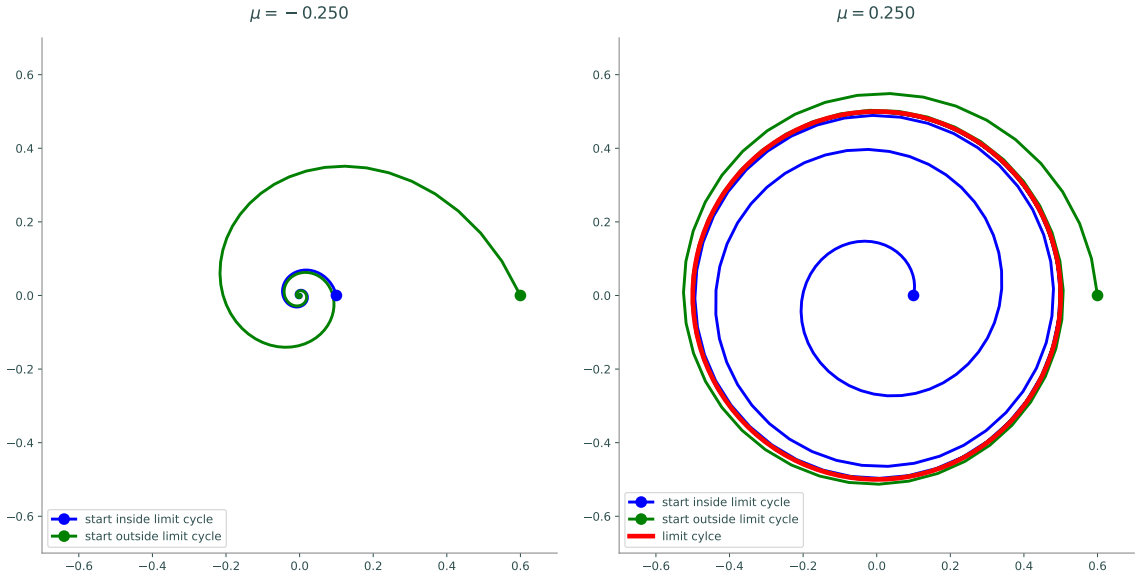


Figure 7.8: The solution to Problem 5 at $\mu = -0.25$ and at $\mu = 0.25$. The points mark the initial positions of the two trajectories.

Problem 5. Create an animation showing how two trajectories of the system (7.6) change as the parameter μ changes from -0.25 to 0.25. Use 251 steps in your μ-linspace. Use the two inital starting points $(r_0, \theta_0) = (0.1, 0)$ and $(0.6, 0)$. Set $\omega = b = 1$. Using `solve_ivp`, let the two trajectories evolve from $t_0 = 0$ to $t_f = 16\pi$. Also plot the limit cycle (when $\mu > 0$), recalling from the above that it occurs at $r = \sqrt{\mu}$. Be sure to include a legend. Compare with Figure 7.8. Embed your animation in your notebook, and remember to push the animation as well.

Hint: To fit the animation in the frame, it may be useful to set the plot view limits to $(-0.7, 0.7)$ (e.g., with `ax.set_xlim`).

Additional Material

Conversion of the Hopf Bifurcation System to Cartesian Coordinates

We can convert the polar system (7.6) to Cartesian coordinates using $x = r\cos\theta$, $y = r\sin\theta$, and $x^2 + y^2 = r^2$:

$$\begin{aligned}
x' &= r'\cos\theta - r\theta'\sin\theta \\
&= \left(\mu r - r^3\right)\cos\theta - r\left(\omega + br^2\right)\sin\theta \\
&= \left(\mu - r^2\right) r\cos\theta - \left(\omega + br^2\right) r\sin\theta \\
&= \left(\mu - \left(x^2 + y^2\right)\right) x - \left(\omega + b\left(x^2 + y^2\right)\right) y \\
&= \mu x - x^3 - xy^2 - \omega y - bx^2 y - by^3 \\
y' &= r'\sin\theta + r\theta'\cos\theta \\
&= \left(\mu r - r^3\right)\sin\theta + r\left(\omega + br^2\right)\cos\theta \\
&= \left(\mu - r^2\right) r\sin\theta + \left(\omega + br^2\right) r\cos\theta \\
&= \left(\mu - \left(x^2 + y^2\right)\right) y + \left(\omega + b\left(x^2 + y^2\right)\right) x \\
&= \mu y - x^2 y - y^3 + \omega x + bx^3 + bxy^2
\end{aligned}$$

8 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

$$u(x+h) = u(x) + u'(x)h + \frac{1}{2}u''(x)h^2 + \frac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \tag{8.1}$$

$$u(x-h) = u(x) - u'(x)h + \frac{1}{2}u''(x)h^2 - \frac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \tag{8.2}$$

Subtracting (8.2) from (8.1) 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:

$$\frac{1}{2h}\underbrace{\begin{bmatrix} -1 & 0 & 1 & & & \\ & -1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 0 & 1 \end{bmatrix}}_{(n-1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-1}) \\ u(x_n) \end{bmatrix}}_{(n+1)\times 1} \approx \underbrace{\begin{bmatrix} u'(x_1) \\ u'(x_2) \\ \vdots \\ u'(x_{n-2}) \\ u'(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1}. \tag{8.3}$$

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

$$\frac{1}{2h}\underbrace{\begin{bmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} u(x_1) \\ u(x_2) \\ \vdots \\ u(x_{n-2}) \\ u(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1} + \underbrace{\begin{bmatrix} -u(x_0)/(2h) \\ 0 \\ \vdots \\ 0 \\ u(x_n)/(2h) \end{bmatrix}}_{(n-1)\times 1} \approx \underbrace{\begin{bmatrix} u'(x_1) \\ u'(x_2) \\ \vdots \\ u'(x_{n-2}) \\ u'(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1}. \tag{8.4}$$

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 (8.1) and (8.2) 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

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \end{bmatrix}}_{(n-1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-1}) \\ u(x_n) \end{bmatrix}}_{(n+1)\times 1} \approx \underbrace{\begin{bmatrix} u''(x_1) \\ u''(x_2) \\ \vdots \\ u''(x_{n-2}) \\ u''(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1}. \tag{8.5}$$

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

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} u(x_1) \\ u(x_2) \\ \vdots \\ u(x_{n-2}) \\ u(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1} + \underbrace{\begin{bmatrix} u(x_0)/h^2 \\ 0 \\ \vdots \\ 0 \\ u(x_n)/h^2 \end{bmatrix}}_{(n-1)\times 1} = \underbrace{\begin{bmatrix} u''(x_1) \\ u''(x_2) \\ \vdots \\ u''(x_{n-2}) \\ u''(x_{n-1}) \end{bmatrix}}_{(n-1)\times 1} \tag{8.6}$$

Each of these matrices consists mostly of zeros; that is, they are *sparse*. So make sure to use `scipy.sparse` when constructing these matrices and those that follow in this lab.[1]

Problem 1. Let $u(x) = \sin((x+\pi)^2 - 1)$. Use (8.3) - (8.6) to approximate $\frac{1}{2}u'' - u'$ at the grid points where $a = 0$, $b = 1$, and $n = 10$. Graph the result.

Hint: You may find `scipy.sparse.diags` useful.

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:

$$\begin{aligned} &\varepsilon u''(x) - u(x)' = f(x), \quad x \in (0,1), \\ &u(0) = \alpha, \quad u(1) = \beta. \end{aligned} \tag{8.7}$$

Equation (8.7) 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

$$D_h U_i = \varepsilon\frac{1}{h^2}\left(U_{i+1} - 2U_i + U_{i-1}\right) - \frac{1}{2h}\left(U_{i+1} - U_{i-1}\right). \tag{8.8}$$

Thus we discretize equation (8.7) using the equations

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

[1] See the Volume 1 lab "Linear Systems" for a refresher on sparse matrices.

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:

$$\begin{aligned} \frac{1}{h^2}\underbrace{\begin{bmatrix} -2\varepsilon & (\varepsilon - h/2) & 0 & \dots & 0 \\ (\varepsilon + h/2) & -2\varepsilon & (\varepsilon - h/2) & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & (\varepsilon + h/2) & -2\varepsilon & (\varepsilon - h/2) \\ 0 & \dots & & (\varepsilon + h/2) & -2\varepsilon \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_{n-2} \\ U_{n-1} \end{bmatrix}}_{(n-1)\times 1} \\ = \underbrace{\begin{bmatrix} f(x_1) - \alpha(\varepsilon + h/2)/h^2 \\ f(x_2) \\ \vdots \\ f(x_{n-2}) \\ f(x_{n-1}) - \beta(\varepsilon - h/2)/h^2 \end{bmatrix}}_{(n-1)\times 1}. \end{aligned} \tag{8.9}$$

Problem 2. Use equation (8.9) to solve the singularly perturbed BVP (8.7) 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.

Hint: Use `scipy.sparse.linalg.spsolve` to solve for U.

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, $h_1 > h_2 > \dots > h_m$. 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

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

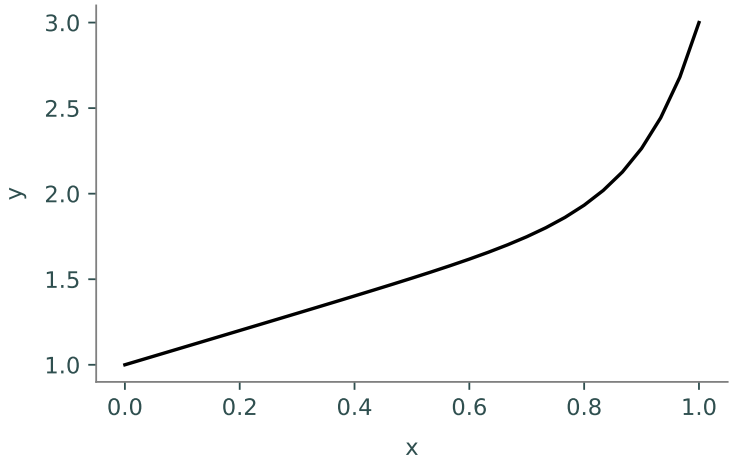


Figure 8.1: The solution to Problem 2. The solution gets steeper near $x = 1$ as ε gets small.

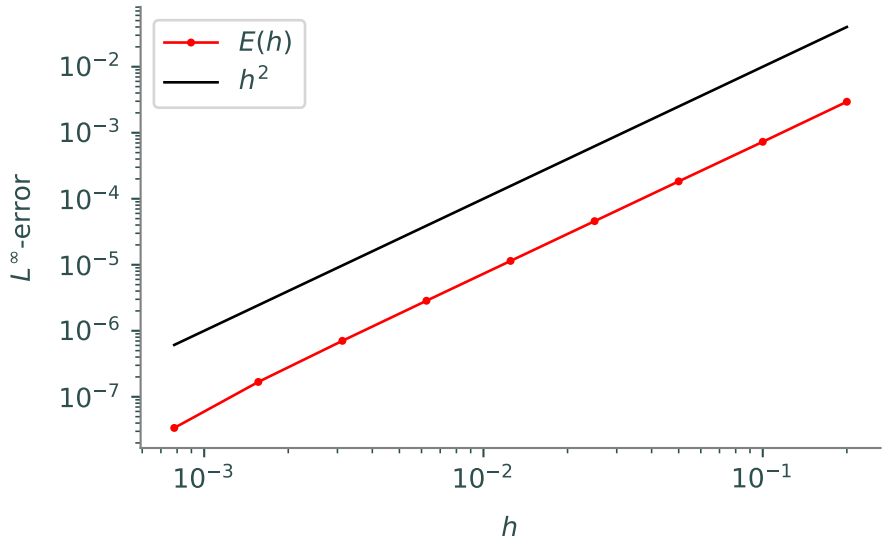


Figure 8.2: Demonstration of second-order convergence for the finite difference approximation (8.8) of the BVP given in (8.7) with $\varepsilon = .5$.

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 (8.7). 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 8.2, except with $\varepsilon = 0.1$. Also plot h^2 on the same plot, as in Figure 8.2. Remember to use sparse matrices.

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)]))
```

Consider a similar, but somewhat generalized ODE of the form in (8.7),

$$\begin{aligned} &a_1(x)y''(x) + a_2(x)y'(x) + a_3(x)y(x) = f(x), \quad x \in (a, b), \\ &y(a) = \alpha, \quad y(b) = \beta. \end{aligned}$$

The functional coefficients a_1, a_2, a_3 can be treated similarly to constant coefficients, but with each row corresponding to a gridpoint in x (but not including the endpoints). Applying the finite difference approximations gives the following system of equations:

$$\begin{aligned} &\frac{1}{h^2}\left(\left(-2a_1(x_1) + h^2 a_3(x_1)\right) U_1 + \left(a_1(x_1) + \frac{h}{2}a_2(x_1)\right) U_2\right) \\ &\qquad = f(x_1) - \alpha\left(\frac{a_1(x_1)}{h^2} - \frac{a_2(x_1)}{2h}\right) \\ &\frac{1}{h^2}\left(\left(a_1(x_2) - \frac{h}{2}a_2(x_2)\right) U_1 + \left(-2a_1(x_2) + h^2 a_3(x_2)\right) U_2 + \left(a_1(x_2) + \frac{h}{2}a_2(x_2)\right) U_3\right) \\ &\qquad = f(x_2) \\ &\qquad \vdots \\ &\frac{1}{h^2}\left(\left(a_1(x_k) - \frac{h}{2}a_2(x_k)\right) U_{k-1} + \left(-2a_1(x_k) + h^2 a_3(x_k)\right) U_k + \left(a_1(x_k) + \frac{h}{2}a_2(x_k)\right) U_{k+1}\right) \\ &\qquad = f(x_k) \end{aligned}$$

$$\vdots$$

$$\frac{1}{h^2}\left(\left(a_1(x_{n-1}) - \frac{h}{2}a_2(x_{n-1})\right)U_{n-2} + \left(-2a_1(x_{n-1}) + h^2 a_3(x_{n-1})\right)U_{n-1}\right)$$
$$= f(x_{n-1}) - \beta\left(\frac{a_1(x_{n-1})}{h^2} + \frac{a_2(x_{n-1})}{2h}\right)$$

This can (and should) be put into matrix form to be solved. It was not done here because it requires more page space than is available.

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 (8.8) correctly. Remember to use sparse matrices.

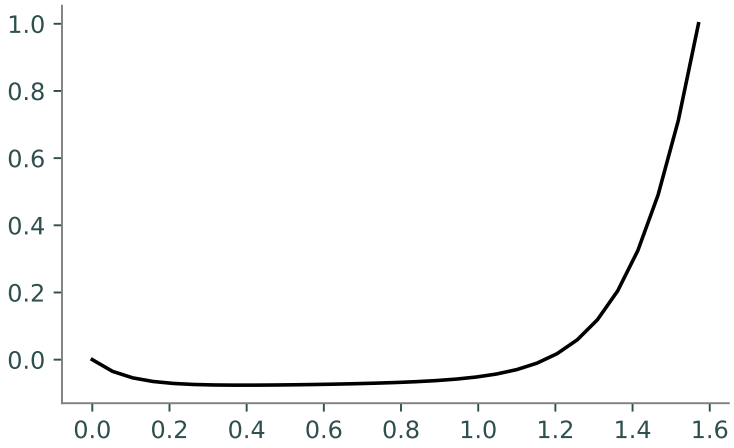


Figure 8.3: The solution to Problem 4.

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.

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.

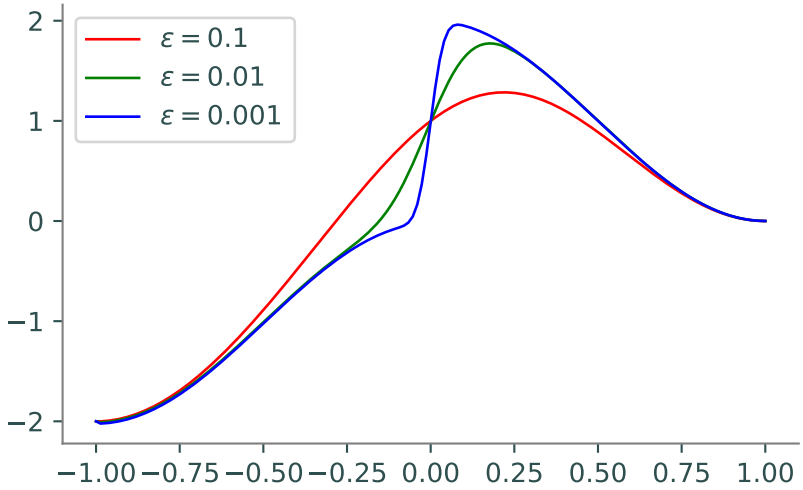


Figure 8.4: The solution to Problem 5.

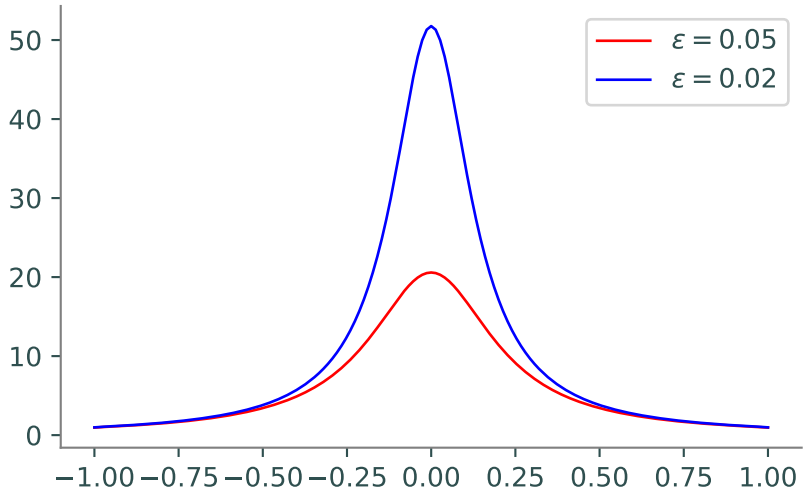


Figure 8.5: The solution to Problem 6.

9 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 traveling 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

$$u_{tt} = s^2 \nabla^2 u \tag{9.1}$$

where $\nabla^2 = \nabla \cdot \nabla$ is the Laplace operator. 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 traveling to the right with constant speed $|s|$, and the other is a signal traveling to the left with speed $|s|$.

Given two homogeneous Dirichlet boundary conditions (for the second-order spatial derivative) and two sets of initial conditions (because the second-order time derivative), the wave equation takes the form

$$\begin{aligned} u_{tt} &= s^2 u_{xx}, \quad 0 < x < l, \quad t > 0, \\ u(0,t) &= u(l,t) = 0, \\ u(x,0) &= f(x), \\ u_t(x,0) &= g(x). \end{aligned} \tag{9.2}$$

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. Note that the superscipt index m denotes the time index and runs from 0 to M, while the subscript j denotes the spatial index and runs from 0 to J. Recall that the centered approximations in space and time are

$$\begin{aligned} D_{tt}U_j^m &= \frac{U_j^{m+1} - 2U_j^m + U_j^{m-1}}{(\Delta t)^2}, \\ D_{xx}U_j^m &= \frac{U_{j+1}^m - 2U_j^m + U_{j-1}^m}{(\Delta x)^2}. \end{aligned} \tag{9.3}$$

The resulting method is given by

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

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

$$U^{m+1} = AU^m - U^{m-1} \tag{9.4}$$

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

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

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

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

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

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

or if f is not readily differentiable,

$$U_j^1 = U_j^0 + g(x_j)\Delta t + \frac{\lambda^2}{2}(U_{j-1}^0 - 2U_j^0 + U_{j+1}^0). \tag{9.5}$$

This method is conditionally stable; the CFL condition (a necessary condition for convergence of PDEs) is that $\lambda \leq 1$.

Notice that the matrix A is sparse. As such, you'll want to use sparse matrices and routines from `scipy.sparse`.

Problem 1. Define a function `solve_wave` to numerically approximate solutions to equations of the form (9.2). Your function should accept the following as parameters:

- a spatial domain (x_0, x_f),
- a time domain (t_0, t_f),
- a number of spatial intervals J,
- a number of time intervals M,
- the parameter s,
- an initial condition $f(x) = u(x, 0)$, and
- an initial condition $g(x) = u_t(x, 0)$.

Compute and return the numerical approximation U as an array using equations (9.4) and (9.5).

Hint: Remember to use `scipy.sparse` when defining A, though U may be returned as a dense matrix (that is, a NumPy array). You may find `scipy.sparse.diags` useful.

Problem 2. 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 $t \in [0, 0.5]$. 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 $u(x,t) = \sin(2\pi x)\cos(2\pi t)$ graphically. This function is known as a standing wave. See Figure 9.1.

Hint: Consider writing a function to create the animation, as you'll need to create similar animations for the next several problems as well.

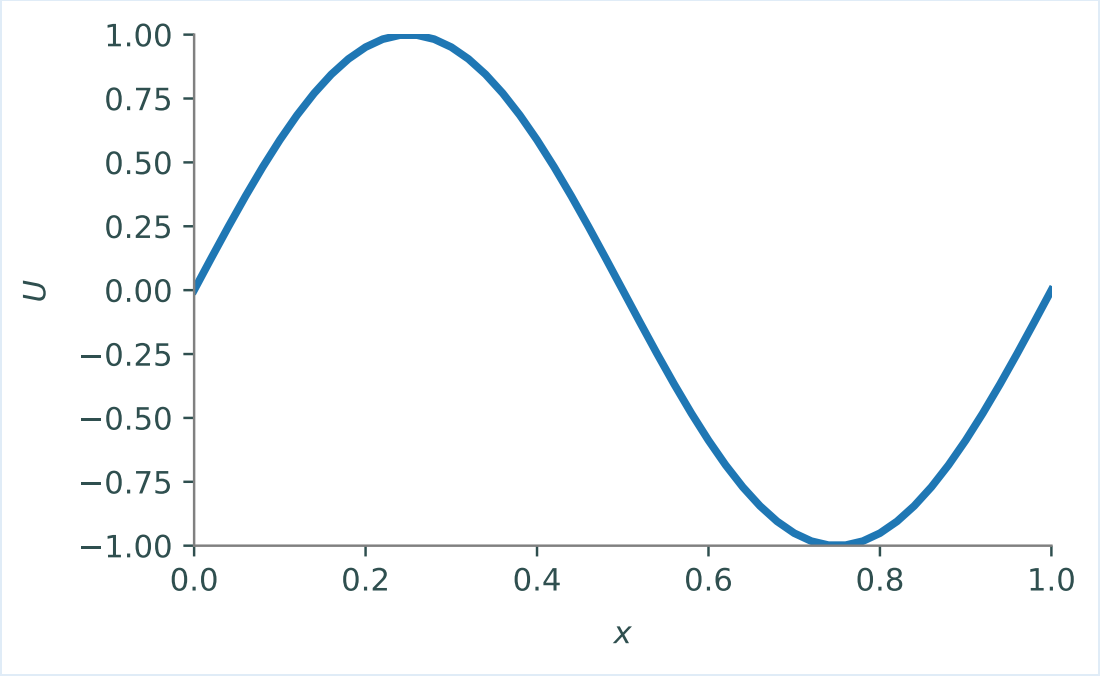


Figure 9.1: $u(x, t = 0)$.

Problem 3. Consider the initial boundary value problem

$$\begin{aligned} u_{tt} &= u_{xx}, \\ u(0,t) &= u(1,t) = 0, \\ u(x,0) &= 0.2e^{-m^2\left(x-\frac{1}{2}\right)^2} \\ u_t(x,0) &= 0.4m^2\left(x-\frac{1}{2}\right)e^{-m^2\left(x-\frac{1}{2}\right)^2}. \end{aligned}$$

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 Figure 9.2.

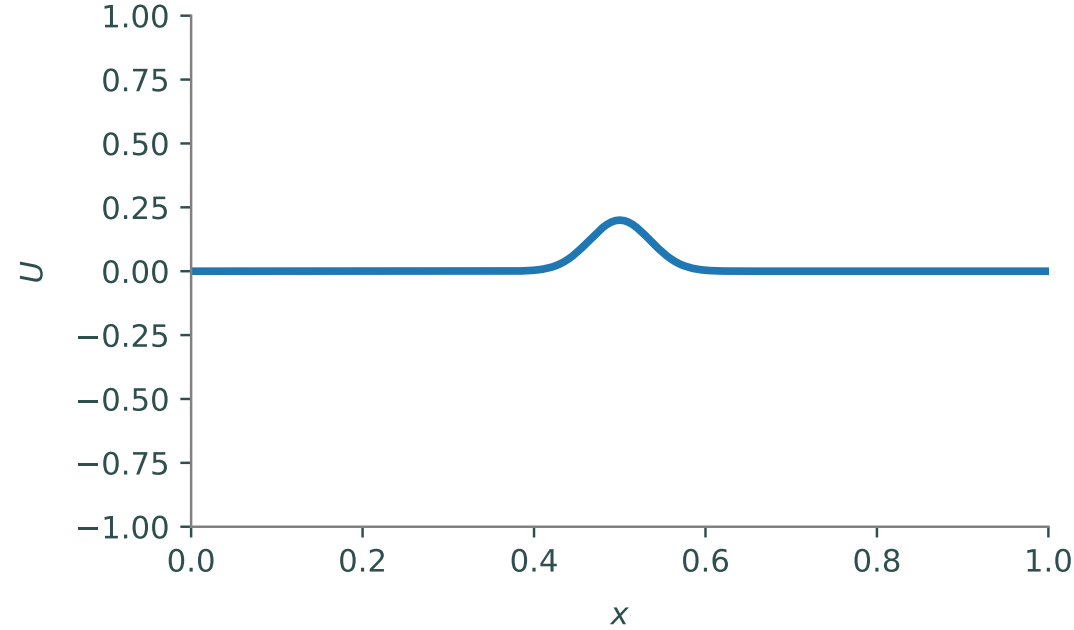


Figure 9.2: $u(x, t = 0)$.

Problem 4. Consider the initial boundary value problem

$$\begin{aligned} u_{tt} &= u_{xx}, \\ u(0,t) &= u(1,t) = 0, \\ u(x,0) &= 0.2e^{-m^2(x-1/2)^2} \\ u_t(x,0) &= 0. \end{aligned}$$

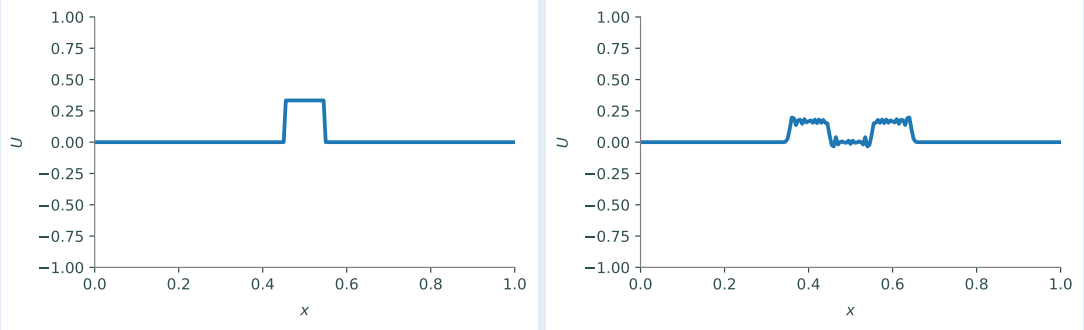
The initial condition separates into two smaller, slower-moving pulses, one traveling to the right and the other to the left. This solution models, for example, a plucked guitar string

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 traveling waves, one traveling to the left and the other to the right, as described earlier.

Problem 5. 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 9.3.



(a) $u(x, t = 0)$. (b) $u(x, t = 0.1)$.

Figure 9.3: The graphs for Problem 5 at various times t.

Traveling 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

$$u_t = G(u, u_x, u_{xx}, \ldots). \tag{9.6}$$

An interesting question to ask is whether (9.6) has traveling wave solutions: is there a signal or wave profile $f(x)$, so that $u(x,t) = f(x - st)$ is a solution of (9.6) that carries the signal at a constant speed s? These traveling waves are often significant physically. For example, in a PDE modeling insect population dynamics a traveling wave could represent a swarm of locusts; in a PDE describing a combustion process a traveling wave could represent an explosion or detonation.

Burgers' equation

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

$$u_t + \left(\frac{u^2}{2}\right)_x = \nu u_{xx}, \tag{9.7}$$

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 traveling wave solution $u(x,t) = \hat{u}(x-st)$ for Burgers' equation. We transform (9.7) into the moving frame $(x,t) \to (\bar{x},\bar{t}) = (x-st,t)$. In this frame (9.7) becomes

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

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

$$-su_{\bar{x}} + \left(\frac{u^2}{2}\right)_{\bar{x}} = \nu u_{\bar{x}\bar{x}}. \tag{9.8}$$

From here on we will drop the bar notation for simplicity. We seek a traveling 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 traveling 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 traveling wave must be $s = \frac{u_- + u_+}{2}$.

Usually at this point, the traveling wave must be numerically solved using the profile ODE ((9.8) for Burgers equation). However, the profile ODE for Burgers' is simple enough that it is possible to obtain an analytic solution. The traveling wave is given by

$$\hat{u}(x) = s - a\tanh\left(\frac{ax}{2\nu} + \delta\right) \tag{9.9}$$

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

Stability of traveling waves

Suppose that an evolutionary PDE

$$u_t = G(u, u_x, u_{xx}, \ldots). \tag{9.10}$$

has a traveling 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 traveling wave show up in real life? This question is the start of the mathematical study of stability of traveling waves.

We begin by translating (9.10) 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 traveling 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 traveling 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 traveling wave is a nontrivial task.

Visualizing stability of the traveling wave solution of Burgers' equation

The traveling 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

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

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

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

$$\begin{aligned} U_j^{m+1} - U_j^m = {} & K_1\left[(s - U_j^{m+1})(U_{j+1}^{m+1} - U_{j-1}^{m+1}) + (s - U_j^m)(U_{j+1}^m - U_{j-1}^m)\right] \\ & + K_2\left[(U_{j+1}^{m+1} - 2U_j^{m+1} + U_{j-1}^{m+1}) + (U_{j+1}^m - 2U_j^m + U_{j-1}^m)\right] \end{aligned} \tag{9.11}$$

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

Unlike the previous problems, this equation is implicit in U^{m+1}. That is, we can't solve each iteration for the next step U^{m+1} explicitly. Instead, we'll need to use a root-finding algorithm like Newton's method.

Problem 6. Numerically solve the initial value problem

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

for $t \in [0,1]$, where $\hat{u}$ is given in (9.9) with $\nu = 1$ and $\delta = 0$. Let the perturbation $v(x)$ be given by

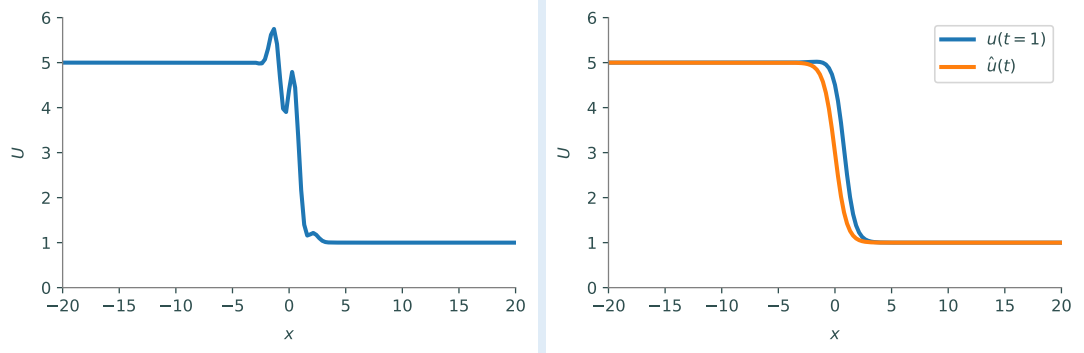
$$v(x) = 3.5(\sin(3x) + 1)\frac{1}{\sqrt{2\pi}}\exp(-x^2/2).$$

Approximate the x domain, $(-\infty, \infty)$, numerically by the finite interval $[-20, 20]$, and fix $u(-20) = u_-$, $u(20) = u_+$. Let $u_- = 5$, $u_+ = 1$ which makes $s = 3$. Use 150 intervals in space and 350 steps in time. Animate your solution U. Also include in your animation the original traveling wave $\hat{u}$. You should see the solution converge to a translate of $\hat{u}$. See Figure 9.4. For your root-finding algorithm, use `scipy.optimize.fsolve`.

Hint: To solve for each U^{m+1}, define a function `cranknicolson` that accepts a guess for U^{m+1} and returns the expression obtained from moving all the terms in (9.11) to one side (call the expression $\tilde{U}$). At each iteration, use `fsolve` to find the value of U^{m+1} that makes $\tilde{U} = \mathbf{0}$—that is, it solves (9.11). You can use U^m as a good guess.

We still need to enforce the boundary conditions. Before returning $\tilde{U}$, set the value of, for example, the first value $\tilde{U}_0$ to be $U_0^{m+1} - u_-$. This will ensure `fsolve` finds U^{m+1} such that $U_0^{m+1} = u_-$.

Also note that `fsolve` accepts an additional parameter `args` that it passes to the function whose root it is finding. You might consider setting up `cranknicolson` to accept the previous value U^m.



(a) $u(x, t = 0)$. (b) $u(x, t = 1)$ vs $\hat{u}$.

Figure 9.4: The graphs for Problem 6.

Two-Dimensional Wave

Consider the two-dimensional wave equation:

$$u_{tt} = \alpha^2 \nabla^2 u = \alpha^2(u_{xx} + u_{yy}) \tag{9.12}$$

$$u(x, y, t) = 0, \quad (x, y) \in \partial([x_0, x_f] \times [y_0, y_f]) \tag{9.13}$$

$$\begin{aligned} u(x, y, 0) &= f(x, y) \\ u_t(x, y, 0) &= g(x, y) \end{aligned} \tag{9.14}$$

for some real, non-negative α.

As with the one-dimensional wave equation, we'll use the second-order centered difference approximations (as in (9.3)) for each of the second derivatives. Let $U_{i,j}^m$ be the numerical approximation for $u(x_i, y_j, t_m)$. We can approximate (9.12) with

$$\frac{U_{i,j}^{m+1} - 2U_{i,j}^m + U_{i,j}^{m-1}}{(\Delta t)^2} = \alpha^2 \left[\frac{U_{i+1,j}^m - 2U_{i,j}^m + U_{i-1,j}^m}{(\Delta x)^2} + \frac{U_{i,j+1}^m - 2U_{i,j}^m + U_{i,j-1}^m}{(\Delta y)^2} \right].$$

Assuming the same step size in each spatial dimension $h = \Delta x = \Delta y$, we can rearrange:

$$\begin{aligned} U_{i,j}^{m+1} &= \frac{\alpha^2(\Delta t)^2}{h^2}(U_{i+1,j}^m - 2U_{i,j}^m + U_{i-1,j}^m + U_{i,j+1}^m - 2U_{i,j}^m + U_{i,j-1}^m) + 2U_{i,j}^m - U_{i,j}^{m-1} \\ &= \lambda \left(U_{i+1,j}^m + U_{i-1,j}^m + U_{i,j+1}^m + U_{i,j-1}^m - 4U_{i,j}^m \right) + 2U_{i,j}^m - U_{i,j}^{m-1} \end{aligned} \tag{9.15}$$

where $\lambda = \frac{\alpha^2(\Delta t)^2}{h^2}$. Assume there are N subintervals in each spatial dimension so that there are $N+1$ points, $x_0 < \ldots < x_N$ and $y_0 < \ldots < y_N$. Because of our homogeneous Dirichlet boundary conditions (9.13), we have $U_{i,j}^m = 0$ for $i = 0, N$ or $j = 0, N$, so we can just compute $U_{i,j}^{m+1}$ for $1 \le i \le N-1$ and $1 \le j \le N-1$, and ignore $U_{i,j}^m$ terms on the boundary. We'll flatten each U^m and write this scheme as the matrix equation

$$U^{m+1} = AU^m - U^{m-1} \tag{9.16}$$

where

$$A = \begin{bmatrix} T & \Lambda & & \\ \Lambda & T & \Lambda & \\ & \ddots & \ddots & \Lambda \\ & & \Lambda & T \end{bmatrix}$$

$$\Lambda = \lambda I$$

$$T = \begin{bmatrix} 2-4\lambda & \lambda & & & & \\ \lambda & 2-4\lambda & \lambda & & & \\ & \ddots & \ddots & \ddots & & \\ & & \lambda & 2-4\lambda & \lambda & \\ & & & \lambda & 2-4\lambda & \end{bmatrix}.$$

We have that T and Λ are both $(N-1) \times (N-1)$, and each U^m has length $(N-1)^2$. Note that we may flatten each U^m either by columns or by rows due to the spatial symmetry of (9.15).

However, for a given time step our iterative matrix algorithm (9.16) requires two previous time steps, so for the first time step we'll need a different method. We'll start with a Taylor approximation centered at $t = 0$ and then use (9.12) and (9.14):

$$\begin{aligned} u(x, y, \Delta t) &= u(x, y, 0) + \Delta t\, u_t(x, y, 0) + \frac{(\Delta t)^2}{2} u_{tt}(x, y, 0) + \mathcal{O}\left((\Delta t)^3\right) \\ &= u(x, y, 0) + \Delta t\, g(x, y) + \frac{(\Delta t)^2}{2} \alpha^2 \nabla^2 u(x, y, 0) + \mathcal{O}\left((\Delta t)^3\right). \end{aligned}$$

We then expand the last term using second-order centered second-derivative approximations as in (9.3):

$$\begin{aligned}
\frac{(\Delta t)^2}{2}\alpha^2\nabla^2 u(x,y,0) &= \frac{\alpha^2(\Delta t)^2}{2}\big(u_{xx}(x,y,0)+u_{yy}(x,y,0)\big)\\
&= \frac{\alpha^2(\Delta t)^2}{2}\bigg(\frac{u(x+h,y,0)-2u(x,y,0)+u(x-h,y,0)}{h^2}\\
&\qquad + \frac{u(x,y+h,0)-2u(x,y,0)+u(x,y-h,0)}{h^2} + \mathcal{O}\left((\Delta t)^2\right)\bigg)\\
&= \frac{\alpha^2(\Delta t)^2}{2h^2}\bigg(u(x+h,y,0)+u(x-h,y,0)\\
&\qquad + u(x,y+h,0)+u(x,y-h,0)-4u(x,y,0)\bigg) + \mathcal{O}\left((\Delta t)^4\right).
\end{aligned}$$

Using our $U^m_{i,j}$ notation, $u(x,y,\Delta t)$ becomes

$$U^1_{i,j} = U^0_{i,j} + \Delta t\, g(x,y) + \frac{\lambda}{2}\left(U^0_{i+1,j}+U^0_{i-1,j}+U^0_{i,j+1}+U^0_{i,j-1}-4U^0_{i,j}\right) + \mathcal{O}\left((\Delta t)^3\right).$$

Problem 7. Solve the 2D wave equation (9.12). Use $N = 200$ spatial subintervals and $M = 500$ time subintervals. Use a spatial domain $(x,y) \in [-10,10]\times[-10,10]$ and a time domain $t \in (t_0,t_f) = (0,40)$. Set $f(x,y) = 3\frac{1}{2\pi}\exp\left(-\frac{1}{2}\left(x^2+y^2\right)\right)$, and set $g(x,y) = 0$. Finally, use $\alpha = 0.8$. Animate the result, and compare with Figure 9.5. Remember to use sparse matrices.

Hint: You may find useful the functions `scipy.sparse.diags`, `scipy.sparse.block_diag`, and `<sparse_matrix>.setdiag`. Also note that $f(x,y)$ is equal to 3 times the probability density function of the standard two-dimensional normal distribution.

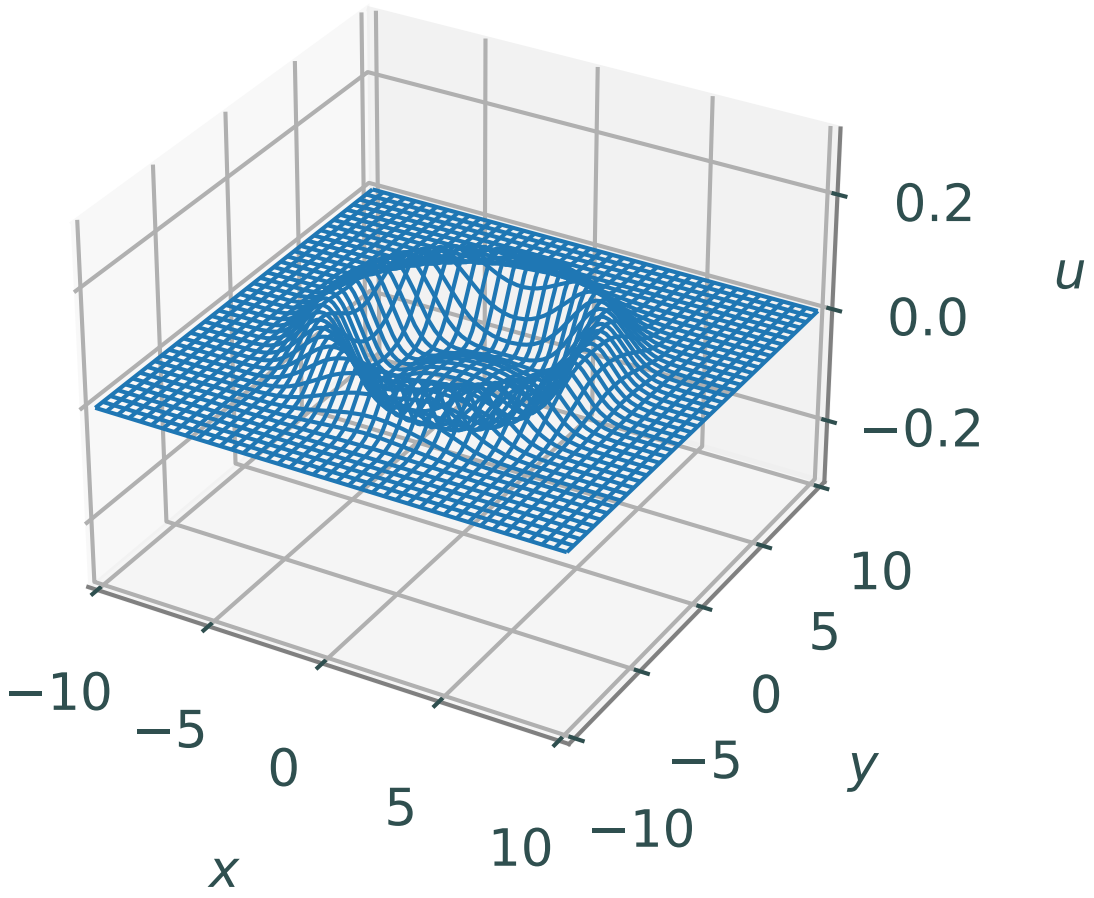


Figure 9.5: The solution to Problem 7 (the 2D wave equation) $u(x, y, t = 6)$.

10 Conservation Laws and Heat Flow

Many physical phenomena have a conservation law associated with them. For instance, matter, energy, and momentum are all conserved quantities. The *fundamental conservation law* states that the rate of change of the total quantity in the system is equal to the rate that the quantity enters the system plus the rate at which the quantity is produced by sources inside the system. While this is a *global* property, we can use it to obtain a *local* differential equation that the concentration of the quantity must obey everywhere in the system. Because of this, conservation laws are very important in modeling a wide variety of phenomena.

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 $\vec{n}(\vec{x})$ represent the normal direction to V at $\vec{x} \in \partial V$, and let $\vec{J}(\vec{x}, t)$ be the flux vector for the quantity, so that $\vec{J}(\vec{x}, t) \cdot \vec{n}(\vec{x})\, dA$ 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

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

and the rate at which the quantity enters V is

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

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

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

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

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

Putting these together yields

$$\int_V u_{\vec{x}}(\vec{x}, t)\, d\vec{x} = \int_V \left(-\nabla \cdot \vec{J} + g(\vec{x}, t, u)\right)\, d\vec{x}$$

Since this holds for all nice subsets $V \subset \Omega$ with V arbitrarily small, the integrands must be equal everywhere, and 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 operator and $\nabla \cdot \vec{J} = \frac{\partial J_1}{\partial x_1} + \cdots + \frac{\partial J_n}{\partial x_n}$

Constitutive Relations

So far, our conservation law consists of 2 unknowns (u and J) but only 1 equation. To this equation we need to add other equations, called *constitutive relations*, which are used to fully determine the system.

For example, suppose we wish to model 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

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

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

$$u_t - \nu \nabla^2 u(\vec{x}, t) = g(\vec{x}, t, u)$$

where ∇^2 is the Laplace operator:

$$\nabla^2 u(\vec{x}, t) = \frac{\partial^2 u}{\partial x_1^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2}.$$

The function g represents heat sources and sinks within the region.

Numerically modeling heat flow

Consider the heat flow equation in one dimension together with an appropriate initial condition $u(x, 0) = f(x)$, homogeneous Dirichlet boundary conditions, and $g(x, t, u) = 0$:

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

We will create an approximation U_j^m to $u(x_j, t_m)$ on the grid $x_j = a + j\Delta x$, $t_m = m\Delta t$, where j and m are indices, $j = 0, \ldots, J$ and $m = 0, \ldots, M$. Thus U_j^m denotes the approximate value of u at the j-th grid point and the m-th time step.

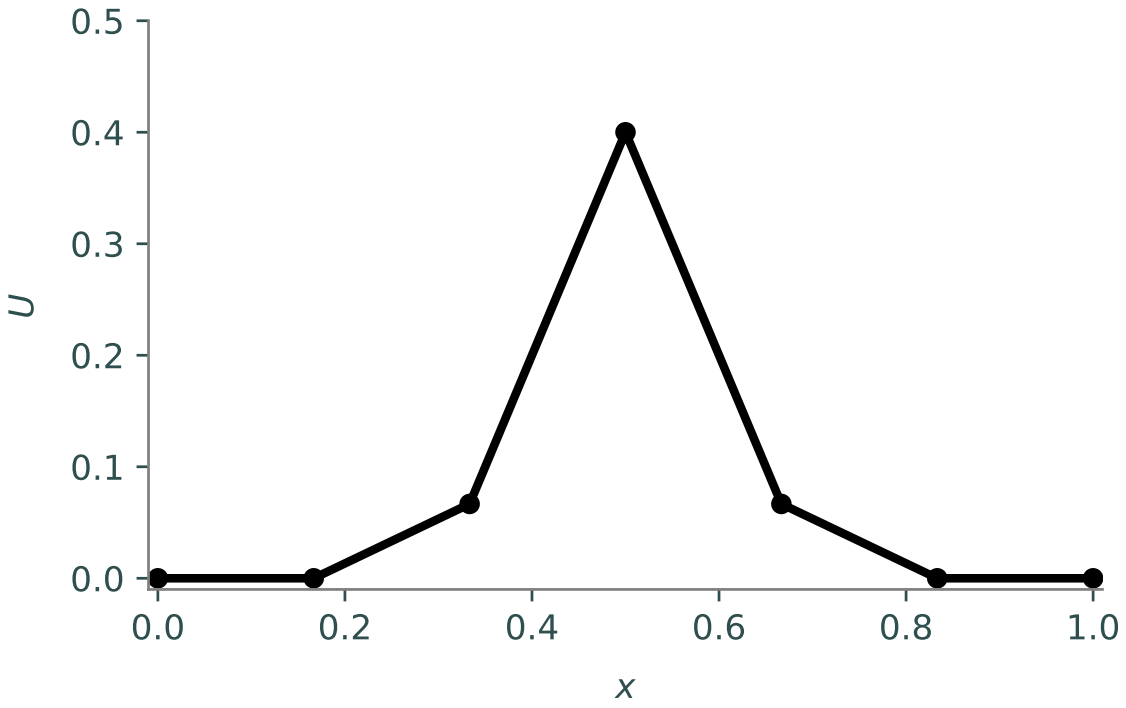


Figure 10.1: The graph of U^0, the approximation to the solution $u(x, t = 0)$ for Problem 1.

As before, we will use the finite difference method to create this approximation. Recall that by using Taylor's theorem, we have the first-order forward difference approximation

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

and the second-order centered difference approximation

$$u_{xx}(x_j,t_m) = \frac{u(x_j+\Delta x,t_m) - 2u(x_j,t_m) - u(x_j-\Delta x,t_m)}{(\Delta x)^2} + \mathcal{O}\left((\Delta x)^2\right).$$

Applying these difference approximations give us the $\mathcal{O}\left((\Delta x)^2 + \Delta t\right)$ explicit method

$$\frac{U_j^{m+1} - U_j^m}{\Delta t} = \nu \frac{U_{j+1}^m - 2U_j^m + U_{j-1}^m}{(\Delta x)^2}, \tag{10.1}$$

$$U_j^{m+1} = U_j^m + \frac{\nu \Delta t}{(\Delta x)^2}(U_{j+1}^m - 2U_j^m + U_{j-1}^m). \tag{10.2}$$

This method can be written in matrix form as

$$U^{m+1} = AU^m, \tag{10.3}$$

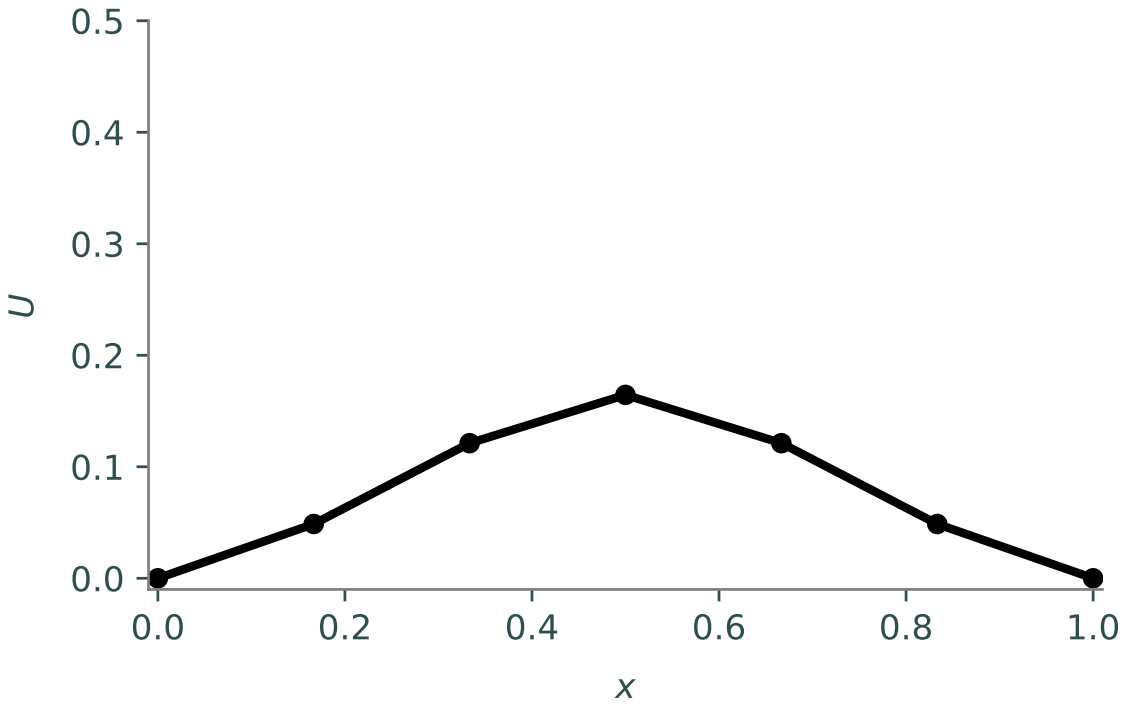


Figure 10.2: The graph of U^4, the approximation to the solution $u(x, t = 0.4)$ for Problem 1.

where A is the $(J+1) \times (J+1)$ tridiagonal matrix given by

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

where $\lambda = \nu \Delta t/(\Delta x)^2$, n is the number of spatial subintervals, and U^m represents the approximation at time t_m. We can initialize this method using the initial condition given in our problem, which tells us that $U_j^0 = f(x_j)$.

To account for our constant 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. Note that the first and last rows of the matrix representation of the differencing scheme are the same as the first and last rows of the identity matrix. This has the effect of keeping the boundary points the same as in the previous step, and thus the same as in the initial condition.

Problem 1. Consider the initial/boundary value problem

$$\begin{aligned} &u_t = 0.05u_{xx}, \quad x \in [0,1], \quad t \in [0,1] \\ &u(0,t) = 0, \quad u(1,t) = 0, \\ &u(x,0) = 2\max\{0.2 - |x - 0.5|, 0\}. \end{aligned} \tag{10.4}$$

Using (10.3), approximate the solution $u(x,t)$ by taking $J = 6$ subintervals in the x dimension and $M = 10$ subintervals in time. Plot the solution at the times $t = 0, t = 0.4$, and $t = 1$. The graphs for U^0 and U^4 are given in Figures 10.1 and 10.2. Be sure to use `scipy.sparse` when defining the matrix A.

Hint: `sparse.diags` may be useful.

Problem 2. Solve the initial/boundary value problem

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12,12], \quad t \in [0,1], \\ &u(-12,t) = 0, \quad u(12,t) = 0, \\ &u(x,0) = \max\{1 - x^2, 0\} \end{aligned} \tag{10.5}$$

using the first-order explicit method (10.3). Use $J = 140$ subintervals in the x dimension and $M = 70$ subintervals in time. The initial and final states are shown in Figure 10.3. Animate your results.

Explicit methods usually have a stability condition, called a CFL condition (for Courant–Friedrichs–Lewy). For method (10.3) the CFL condition that must be satisfied is that

$$\lambda = \frac{\nu \Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

Repeat your computations using $J = 140$ subintervals in the x dimension and $M = 66$ subintervals in time. Animate the results. For these values, the CFL condition is broken; you should be able to clearly see the result of this instability in the approximation U^{66}.

Implicit Methods

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

$$\begin{aligned} u_t(x_j, t_{m+1/2}) &= \frac{u(x_j, t_{m+1}) - u(x_j, t_m)}{\Delta t} + \mathcal{O}\left((\Delta t)^2\right), \\ u_{xx}(x_j, t_{m+1/2}) &= \frac{u_{xx}(x_j, t_{m+1}) + u_{xx}(x_j, t_m)}{2} + \mathcal{O}\left((\Delta x)^2\right). \end{aligned}$$

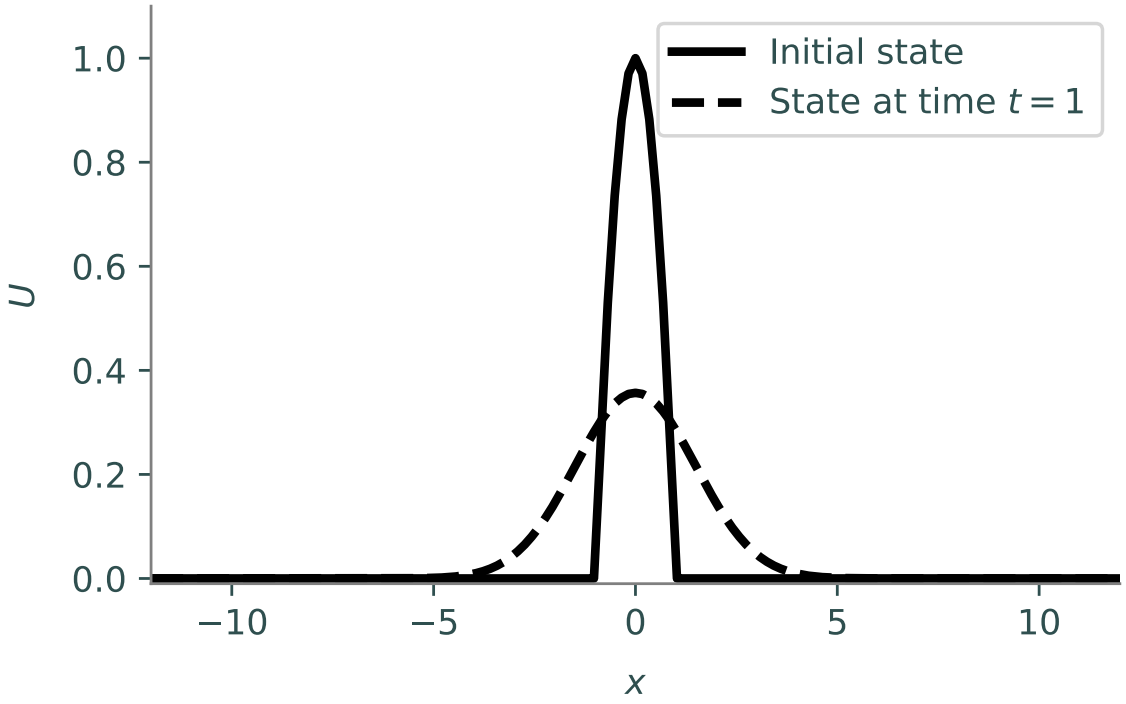


Figure 10.3: The initial and final states for equation Problem 2.

The first equation is a finite difference approximation for u_t, and the second is a midpoint approximation applied to u_{xx}. Then for the equation $u_t = \nu u_{xx}$, these approximations give the relation

$$\begin{aligned} \frac{U_j^{m+1} - U_j^m}{\Delta t} &= \frac{\nu}{2}\left(\frac{U_{j+1}^m - 2U_j^m + U_{j-1}^m}{(\Delta x)^2} + \frac{U_{j+1}^{m+1} - 2U_j^{m+1} + U_{j-1}^{m+1}}{(\Delta x)^2}\right), \\ U_j^{m+1} &= U_j^m + \frac{\nu \Delta t}{2(\Delta x)^2}\left(U_{j+1}^m - 2U_j^m + U_{j-1}^m + U_{j+1}^{m+1} - 2U_j^{m+1} + U_{j-1}^{m+1}\right). \end{aligned} \tag{10.6}$$

This method can be written in matrix form as

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

where A and B are tridiagonal matrices given by

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

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

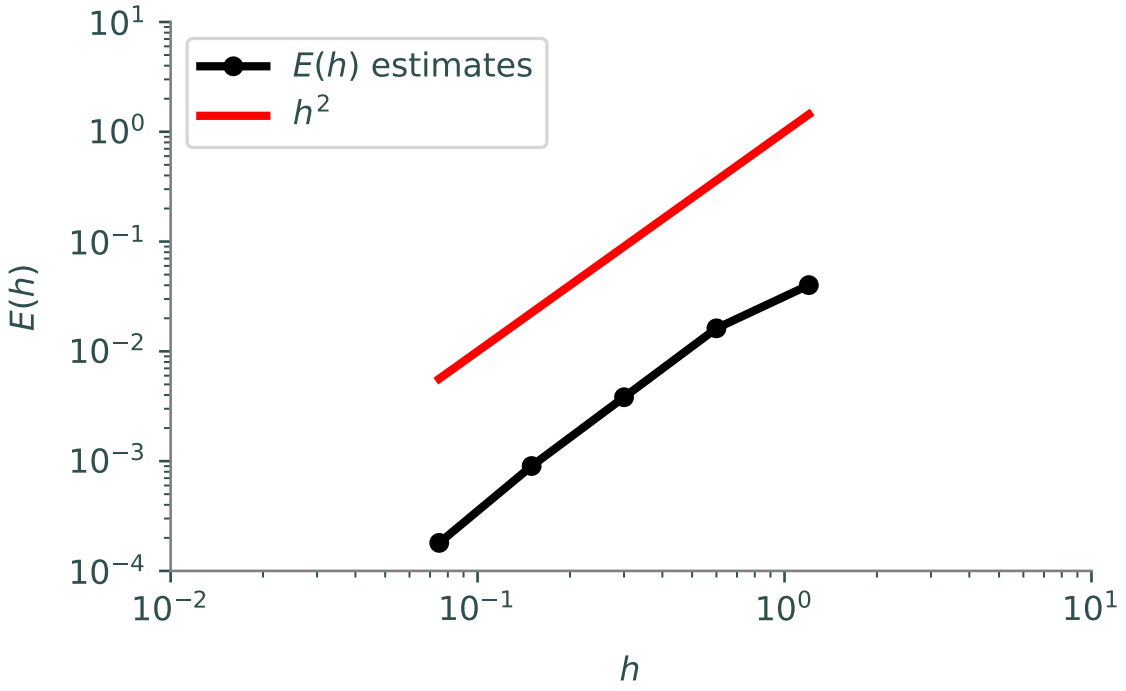


Figure 10.4: $E(h)$ represents the (approximate) maximum error in the numerical solution U to Problem 3 at time $t = 1$, using a spatial step size of h.

where $\lambda = \nu\Delta t/\left(2(\Delta x)^2\right)$, and U^m represents the approximation at time t_m. Note that here we have defined λ differently than we did before!

Accuracy of Numerical Approximations

How do we know if a numerical approximation is reasonable? One way to determine this is to compute solutions for various spatial step sizes h and see if the solutions are converging to something, which we hope to be the true solution. 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 $h \to 0$ (that is, for $h > 0$ small enough).

So, we will compute the approximation y_k for each step size h_k, $h_1 > h_2 > \ldots > h_q$. We will think of y_q as the true solution. Then the error of the approximation for step size $h_k, k < q$, is

$$E(h_k) = \max(|y_k - y_q|) \approx Ch_k^p,$$
$$\log(E(h_k)) = \log(C) + p\log(h_k).$$

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

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12, 12], \quad t \in [0, 1], \\ &u(-12, t) = 0, \quad u(12, t) = 0, \\ &u(x, 0) = \max\{1 - x^2, 0\}. \end{aligned} \tag{10.7}$$

Note that this is an implicit linear scheme; hence, the most efficient way to find U^{m+1} is to create the matrix B as a sparse matrix and use `scipy.sparse.linalg.spsolve`.

Demonstrate that the numerical approximation at $t = 1$ converges. Do this by computing U at $t = 1$ using $20, 40, 80, 160, 320$, and 640 intervals for both time and space (i.e., $J = M$). Reproduce the log–log plot shown in Figure 10.4. The slope of the line there shows the order of convergence.

To measure the error, let h be the spatial step size Δx. Use the solution with the smallest h (largest number of intervals) as if it were the exact solution, then compare each solution only at the x-values that are represented in the solution with the largest h (smallest number of intervals). Use the maximum absolute difference as the value of the error for each solution.

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

11 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

$$\begin{aligned} U^{n+1}_{l,m} = U^n_{l,m} + \lambda\Big(& g(|U^n_{l-1,m} - U^n_{l,m}|)(U^n_{l-1,m} - U^n_{l,m}) \\ & + g(|U^n_{l+1,m} - U^n_{l,m}|)(U^n_{l+1,m} - U^n_{l,m}) \\ & + g(|U^n_{l,m-1} - U^n_{l,m}|)(U^n_{l,m-1} - U^n_{l,m}) \\ & + g(|U^n_{l,m+1} - U^n_{l,m}|)(U^n_{l,m+1} - U^n_{l,m})\Big), \end{aligned} \tag{11.1}$$

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.v3.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 `mode='F'`.

```
from matplotlib import cm, pyplot as plt
from imageio.v3 import imread

# To read in an image, convert it to grayscale, and rescale it.
picture = imread("coke_balloon.jpg", mode='F') * 1./255

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

Simplifying Calculations for Edges and Corners

You will notice that the algorithm given in (11.1) 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.

Coding special cases for the edges and corners is tedious and error-prone, however. 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 `l,m`, then our padded array `Y` has shape `l+2,m+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.

If we do this, we can simply apply (11.1) without having to make special cases for the edges and corners, since those previously undefined terms are each zero when using the padded matrix.

Problem 1. Complete the following function, by implementing the anisotropic diffusion algorithm found in 11.1 for black and white images. Use the padded array technique found in the Simplifying Calculations section.

In your function, use

$$g(x) = e^{-(x/\sigma)^2}$$

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

Hint: Use the `np.pad` function to implement the padded array technique.

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:

$$\begin{aligned} U^{n+1}_{l,m} = U^n_{l,m} + \lambda(&g(||U^n_{l-1,m} - U^n_{l,m}||)(U^n_{l-1,m} - U^n_{l,m}) \\ &+ g(||U^n_{l+1,m} - U^n_{l,m}||)(U^n_{l+1,m} - U^n_{l,m}) \\ &+ g(||U^n_{l,m-1} - U^n_{l,m}||)(U^n_{l,m-1} - U^n_{l,m}) \\ &+ g(||U^n_{l,m+1} - U^n_{l,m}||)(U^n_{l,m+1} - U^n_{l,m})) \end{aligned} \tag{11.2}$$

When implementing this scheme for colored images, use the 2-norm on 3-space, i.e

$$||x|| = \sqrt{x_1^2 + x_2^2 + x_3^2} \tag{11.3}$$

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 using Equation (11.2). 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_, g):
    """ Run the Anisotropic Diffusion differencing scheme
    on the array U of color 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 11.1: 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", mode='F')
x, y = image.shape
for i in range(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

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 11.1

As we can see, after 100 iterations, some of the boundaries between similar shades of grey have smeared unevenly. 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.

To do this, we will define an iterative scheme similar to (11.2), but with g having a different value at each pixel and for each timestep. Using the 2-norm, we first find

$$g^n_{l,m} = \text{mean of 2 smallest } \left(\|U^n_{l-1,m} - U^n_{l,m}\|, \|U^n_{l+1,m} - U^n_{l,m}\|, \|U^n_{l,m-1} - U^n_{l,m}\|, \|U^n_{l,m+1} - U^n_{l,m}\|\right). \tag{11.4}$$

The update process then becomes

$$\begin{aligned} U^{n+1}_{l,m} &= U^n_{l,m} + \lambda(g^n_{l,m}(U^n_{l-1,m} - U^n_{l,m}) \\ &\qquad + g^n_{l,m}(U^n_{l+1,m} - U^n_{l,m}) \\ &\qquad + g^n_{l,m}(U^n_{l,m-1} - U^n_{l,m}) \\ &\qquad + g^n_{l,m}(U^n_{l,m+1} - U^n_{l,m})) \\ &= U^n_{l,m} + \lambda g^n_{l,m}(U^n_{l-1,m} + U^n_{l+1,m} + U^n_{l,m-1} + U^n_{l,m+1} - 4U^n_{l,m}) \end{aligned} \tag{11.5}$$

To compute the values of $g^n_{l,m}$, it is helpful to construct `(l,m)`-shape arrays of the $\|U^n_{l\pm1,m\pm1} - U^n_{l,m}\|$, put them together into a `(4,l,m)`-shape array $[\|U^n_{l-1,m} - U^n_{l,m}\|, \|U^n_{l+1,m} - U^n_{l,m}\|, \|U^n_{l,m-1} - U^n_{l,m}\|, \|U^n_{l,m+1} - U^n_{l,m}\|]$, and then using `np.argsort` with the argument `axis=0`.

If the pixel values are scaled to be in $[0, 1]$ and $\lambda \leq \frac{1}{4}$, the method will be stable, as the scheme's tendency is to move points closer to the values of their neighbors. Below, we include an example of using a minimum-bias filter to diminish the noise in a color image.

Problem 4. 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") * 1./255
x,y,z = image.shape
for dim in range(z):
    for i in range(x*y//30):
        # Assign a random value to a random place
        image[randint(x), randint(y), dim] = (127 + randint(127)) /255.
```

Hint: You may need to reshape the array g so that it has dimension (l, m, 1). That way you can array broadcast it into U, which is (l,m,3).

Original image

Randomly changed 48,000 color values

50 iterations of a min-biased scheme

12 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 wish to study the distribution of heat in a fluid that is moving at some constant speed a. Let y denote the temperature of the fluid at any given location and time. The equation modeling this situation can be obtained from the differential form of the conservation law, where the flux is the sum of a diffusive term $-\varepsilon y_x$ and an advection (or transport) term ay:

$$J = ay - \varepsilon y_x$$

The one-dimension conservation law states that y must then obey the partial differential equation $y_t + J_x = f(x)$, where f represents heat sources in the system. Since $J_x = ay_x - \varepsilon y_{xx}$, we obtain the *advection-diffusion equation*

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

As time progresses, we expect the temperature of the fluid in the pipe to reach a steady state distribution, with $y_t = 0$. Once this steady state has been reached, the heat distribution y then satisfies the ODE

$$\varepsilon y'' - ay' = -f(x).$$

We consider the scenario of 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 $f(x) = 1$. Note that since this a second-order ODE, we need two boundary conditions. Suppose that the fluid is already at a known temperature $y = 2$ as it enters the pipe. This imposes the boundary condition $y(0) = 2$. Suppose further that 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 and thus influence the steady-state temperature. Putting this together leads to a well defined BVP:

$$\begin{aligned} &\varepsilon y'' - y' = -1, \quad 0 < x < 1, \\ &y(0) = 2, \quad y(1) = 4. \end{aligned} \tag{12.1}$$

The analytic solution for $\varepsilon = 0.1$ is shown in Figure 12.1.

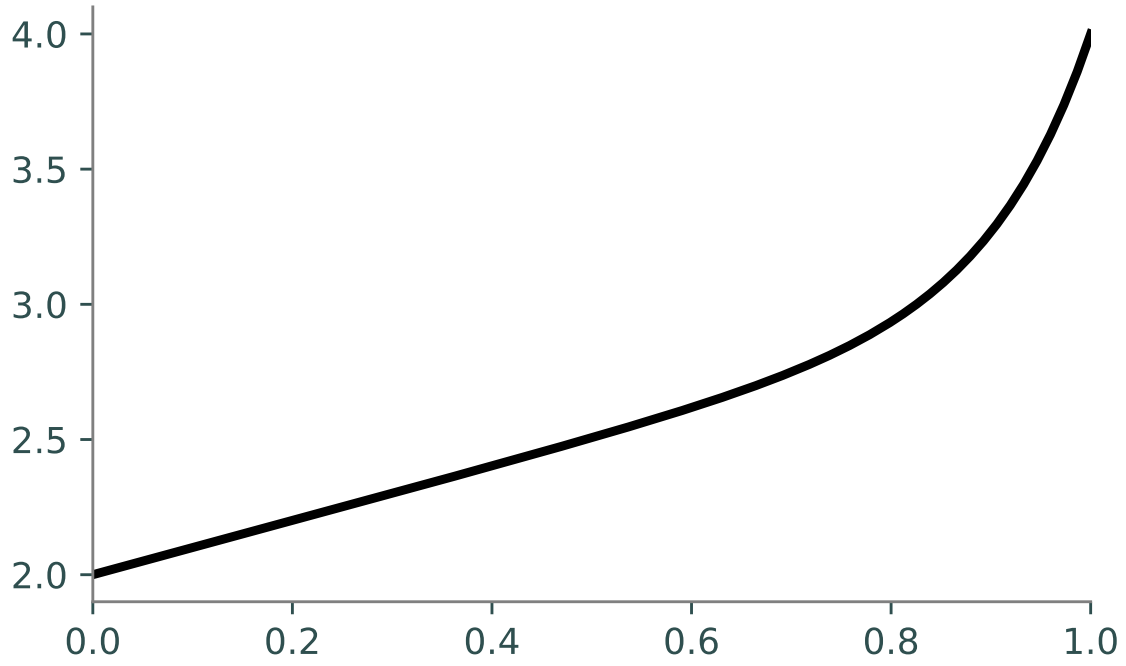


Figure 12.1: The analytic solution of (12.1) for $\varepsilon = 0.1$.

The Weak Formulation

Stepping back momentarily, consider the equation

$$\begin{aligned} &\varepsilon y'' - y' = -f, \quad 0 < x < 1, \\ &y(0) = \alpha, \quad y(1) = \beta. \end{aligned} \tag{12.2}$$

To approximate the solution y using the finite element method, we reframe the problem into one involving integrals, known as its *weak formulation*.

Let w be a smooth function on $[0, 1]$ satisfying $w(0) = w(1) = 0$. Multiplying (12.2) by w and integrating over $[0, 1]$ yields

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

where the second equality follows by integration by parts. For notational convenience, define the functionals a and l by

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

Then, any solution to (12.2) will also satisfy

$$a(y, w) = l(w) \tag{12.3}$$

This equation is the *weak formulation* of (12.2). Note that any solution to the original ODE is also a solution to the weak formulation. However, solutions to the weak formulation need not be solutions to the original ODE, as they may not even be differentiable everywhere. While this may seem like an undesirable property, it allows us to use a wider variety of functions to approximate the true solution.

Now, we choose some appropriate vector space V of functions, and consider the problem of finding a function $y \in V$ that satisfies the weak formulation (12.3) for all $w \in V_0 = \{w \in V | w(0) = w(1) = 0\}$. The *finite element method* consists of choosing V to be some set of piecewise polynomial functions. In this lab, we will consider the case of using piecewise linear functions.

The Finite Element Method

Let P_n be some partition of $[0,1]$, $0 = x_0 < x_1 < \ldots < x_n = 1$, and let V_n be the set of continuous linear piecewise functions v on $[0,1]$ such that v is linear on each subinterval $[x_j, x_{j+1}]$. These subintervals are the finite elements for which this method is named. Note that V_n has dimension $n+1$, since each of the continuous piecewise linear functions in V are uniquely determined by their values at the $n+1$ points $x_0, x_1 \ldots, x_n$. Let $V_{n,0}$ be the subspace of V_n of dimension $n-1$ whose elements are zero at the endpoints of $[0,1]$.

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 12.2 and 12.3. These hat functions form a basis for V_n. Note that the points $x_0, \ldots, x_n$ need not be evenly spaced, and the h_i do not need to be equal. This is in fact one of the major strengths of this approach, as it allows adapting the points in the partition to the problem, which can reduce the error in the approximation. When applied to PDEs, it also is a simple way to handle unusually-shaped domains.

We now can write our approximate solution for y and the arbitrary function w as a linear combination of these basis elements, which will enable us to solve the system numerically. In particular, we can write $\hat{y}(x) = \sum_{i=0}^{n} k_i \phi_i(x)$, where the k_i are to be determined.

To make things more concrete, consider the case of $n = 5$ with the partition $\mathrm{P}_5 = \{x_0, x_1, \ldots, x_5\}$. 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

$$a(\hat{y}, w) = l(w)$$

for all $w \in V_{5,0}$. This can be written equivalently as

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

since a and l are linear in w and $\phi_1, \phi_2, \phi_3, \phi_4$ form a basis for $V_{5,0}$. Since a is also linear in y, we further obtain

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

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

$$AK = \Phi, \tag{12.4}$$

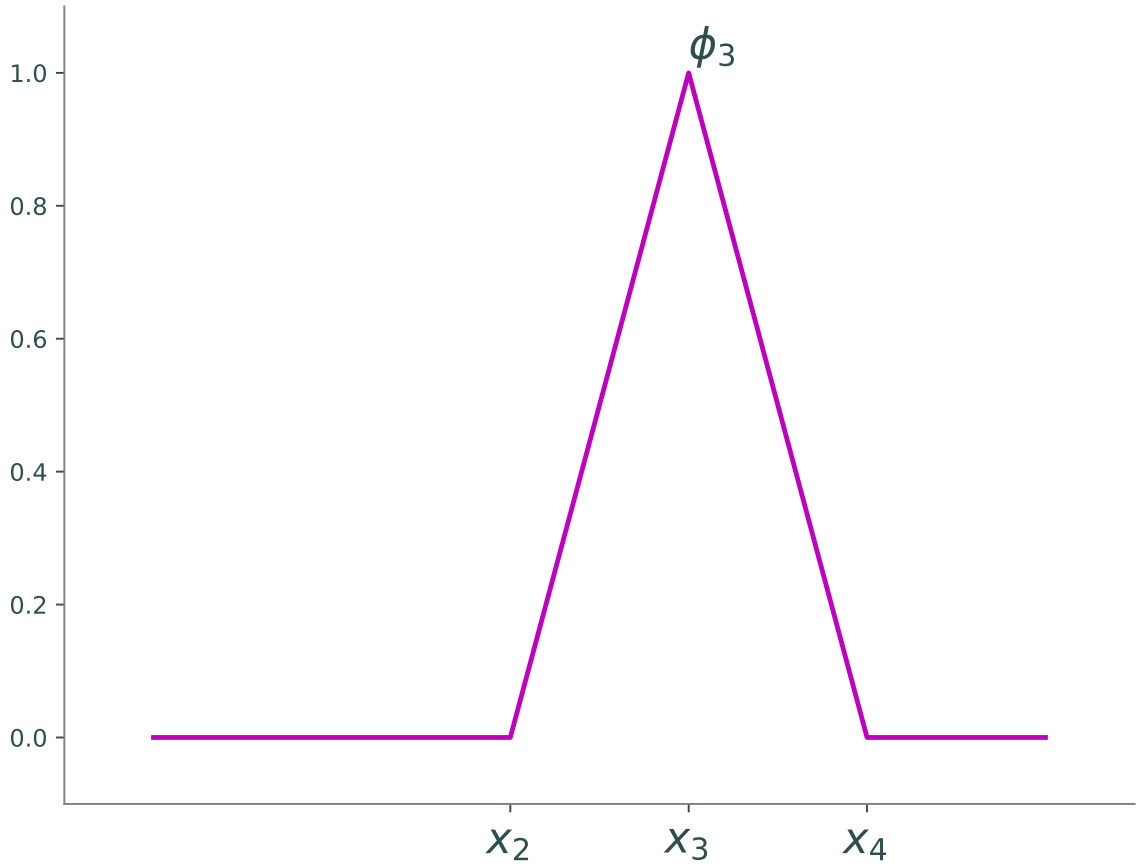


Figure 12.2: The basis function ϕ_3, when the x_i are evenly spaced.

where

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ a(\phi_0, \phi_1) & a(\phi_1, \phi_1) & a(\phi_2, \phi_1) & 0 & 0 & 0 \\ 0 & a(\phi_1, \phi_2) & a(\phi_2, \phi_2) & a(\phi_3, \phi_2) & 0 & 0 \\ 0 & 0 & a(\phi_2, \phi_3) & a(\phi_3, \phi_3) & a(\phi_4, \phi_3) & 0 \\ 0 & 0 & 0 & a(\phi_3, \phi_4) & a(\phi_4, \phi_4) & a(\phi_5, \phi_4) \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{12.5}$$

and

$$K = \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix}, \quad \Phi = \begin{bmatrix} \alpha \\ l(\phi_1) \\ l(\phi_2) \\ l(\phi_3) \\ l(\phi_4) \\ \beta \end{bmatrix}. \tag{12.6}$$

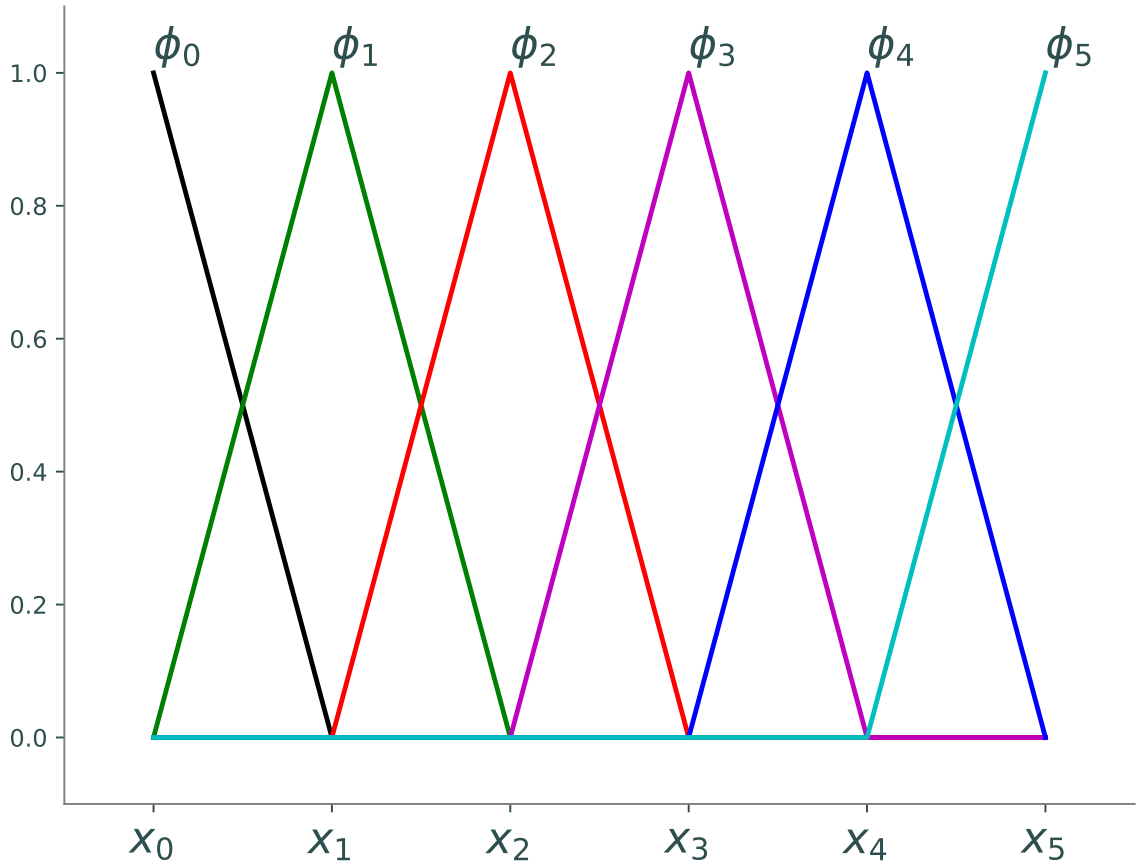


Figure 12.3: The six basis functions for V_5, when the x_i are evenly spaced.

Note that since $a(\phi_i, \phi_j) = 0$ for most values of i, j (in particular, when the hat functions do not have overlapping domains), the finite element method results in a sparse linear system. To compute the coefficients of (12.4) we begin by evaluating some integrals. Since

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

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

we obtain

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

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

which can be put together to obtain (for $f(x) = 1$)

$$a(\phi_i, \phi_j) = \begin{cases} \varepsilon/h_{i+1} + 1/2 & \text{if } j = i+1, \\ -\varepsilon/h_i - \varepsilon/h_{i+1} & \text{if } j = i, \\ \varepsilon/h_i - 1/2 & \text{if } j = i-1, \\ 0 & \text{otherwise,} \end{cases} \tag{12.7}$$

$$l(\phi_j) = -\frac{1}{2}(h_j + h_{j+1}). \tag{12.8}$$

Equation (12.4) 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 sparse matrices from `scipy.sparse`.[1]

If you have become completely lost in the math at this point, do not fear. To summarize, we need to solve 12.4 for K, where A is defined by 12.5 and Φ is defined in 12.6. The elements of A and Φ are defined in 12.7 and 12.8. The vector K is the approximated solution to the ODE given in 12.9 and can be plotted very simply using `plt.plot(x, k)`, where `x` and `k` are arrays of the x_i and k_i. Note that h_i is indexed from 1 to $N+1$, and that $h_i = x_i - x_{i-1}$. You should now have everything you need to know to tackle the problems below.

Problem 1. Use the finite element method to solve

$$\begin{aligned} &\varepsilon y'' - y' = -1, \\ &y(0) = \alpha, \quad y(1) = \beta, \end{aligned} \tag{12.9}$$

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

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

Hint: Make sure that your code does not assume that the grid points are evenly spaced. You may find `scipy.sparse.diags` useful.

[1] See the Volume 1 lab "Linear Systems" for a refresher on sparse matrices.

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 (12.9) 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 12.4. Specifically, use the grid points defined by

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

Be sure to include a legend with your plot.

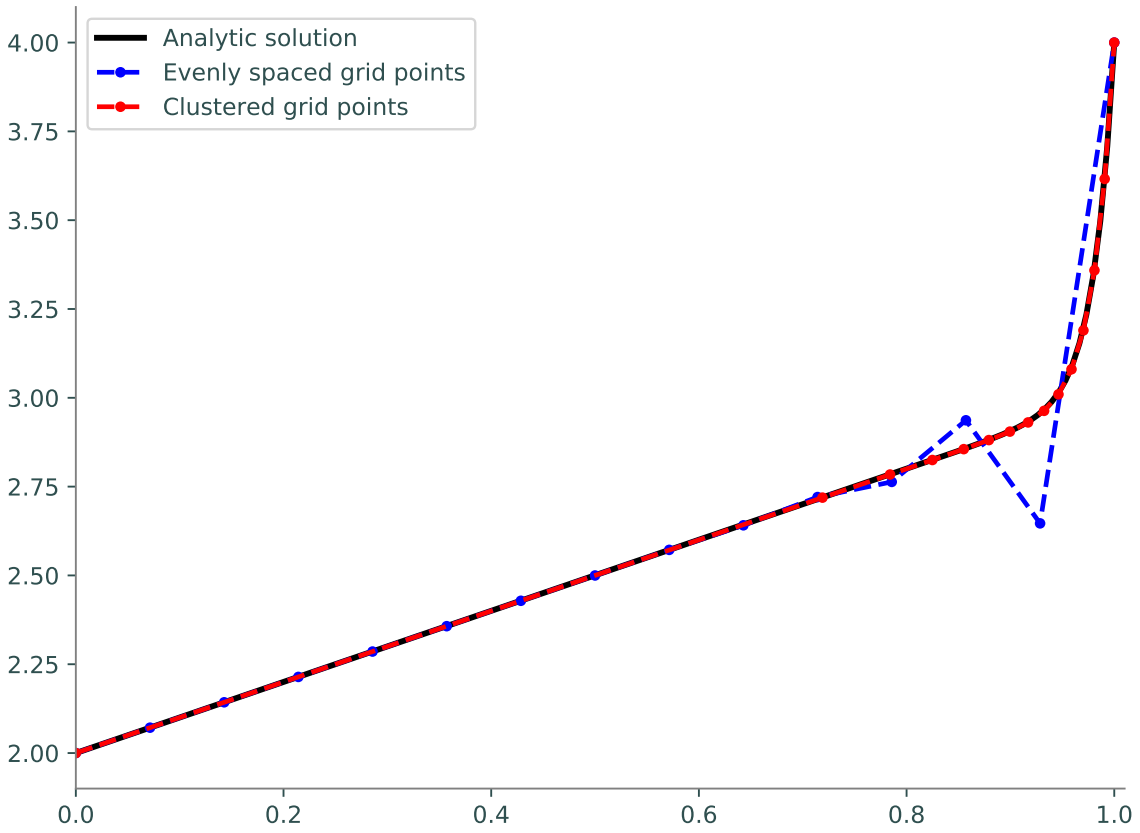


Figure 12.4: Two finite element approximations using 15 grid points, with different spacings.

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 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 (12.4) is exactly the same as the matrix for the second-order centered finite difference method.)

Solve (12.9) with the finite element method using $N = 2^i$ evenly-spaced finite elements, $i = 4, 5, \ldots, 21$. Remember to use sparse matrices, as this greatly reduces the memory and computation needed for the larger N. Compute the error as the maximum absolute value of the difference of the values of the approximate and true solutions at each of the x_i. Use a log-log plot to graph the error, and compare with Figure 12.5. Be sure to label your axes.

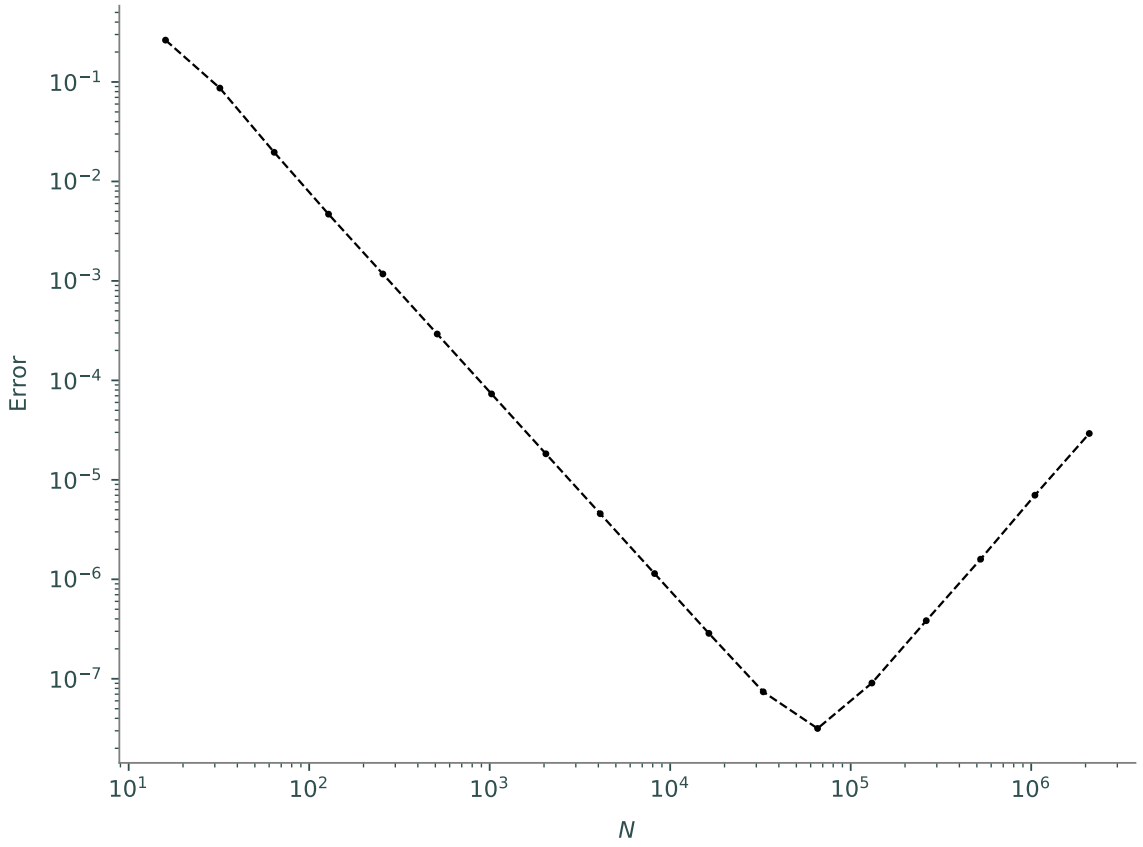


Figure 12.5: Error for the second-order finite element method, as the number of subintervals N grows. Round-off error eventually overwhelms the approximation.

13 Poisson's Equation

Suppose that we want to describe the distribution of heat throughout a region Ω. Let $h(x)$ represent the temperature on the boundary of Ω ($\partial\Omega$), and let $g(x)$ represent the initial heat distribution at time $t = 0$. If we let $f(x,t)$ represent any heat sources/sinks in Ω, then the flow of heat can be described by the boundary value problem (BVP)

$$\begin{aligned} u_t &= \triangle u + f(x,t), \quad x \in \Omega, \quad t > 0, \\ u(x,t) &= h(x), \quad x \in \partial\Omega, \\ u(x,0) &= g(x). \end{aligned} \tag{13.1}$$

When the source term f does not depend on time, there is often a steady-state heat distribution u_∞ that is approached as $t \to \infty$. This steady state u_∞ is a solution of the BVP

$$\begin{aligned} \triangle u + f(x) &= 0, \quad x \in \Omega, \\ u(x,t) &= h(x), \quad x \in \partial\Omega. \end{aligned} \tag{13.2}$$

This last partial differential equation, $\triangle u = -f$, is called Poisson's equation. This equation is satisfied by the steady-state solutions of many other evolutionary processes. Poisson's equation is often used in electrostatics, image processing, surface reconstruction, computational fluid dynamics, and other areas.

Poisson's equation in two dimensions

Consider Poisson's equation together with Dirichlet boundary conditions on a rectangular domain $R = [a,b] \times [c,d]$:

$$\begin{aligned} u_{xx} + u_{yy} &= f, \quad x \text{ in } R \subset \mathbb{R}^2, \\ u &= g, \quad x \text{ on } \partial R. \end{aligned} \tag{13.3}$$

Let $a = x_0, x_1, \ldots, x_n = b$ and $c = y_0, y_1, \ldots, y_n = d$ be evenly spaced grids. Furthermore, suppose that $b - a = d - c$, so the rectangular domain is also square. Thus we have a single stepsize h, where $h = x_{i+1} - x_i = y_{i+1} - y_i$

We look for an approximation $U_{i,j}$ on the grid $\{(x_i, y_j)\}_{i,j=0}^n$.

Recall that

$$\begin{aligned}\triangle u &= u_{xx}(x,y) + u_{yy}(x,y)\\ &= \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}\\ &\quad + \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2} + \mathcal{O}(h^2).\end{aligned}$$

We replace $\triangle$ with the finite difference operator $\triangle_h$, defined by

$$\triangle_h U_{ij} = \frac{U_{i+1,\,j} - 2U_{i,\,j} + U_{i-1,\,j}}{h^2} + \frac{U_{i,\,j+1} - 2U_{i,\,j} + U_{i,\,j-1}}{h^2}, \tag{13.4}$$

$$= \frac{1}{h^2}(U_{i-1,\,j} + U_{i+1,\,j} + U_{i,\,j-1} + U_{i,\,j+1} - 4U_{i,\,j}). \tag{13.5}$$

These equations are linear, so we can expect to write them in matrix form. However, since our unknown variables are doubly-indexed (for x_i and y_j), we first need to rewrite them as a 1-dimensional array. We can do this by "stacking" the columns of the 2-dimensional array. Let the vector of unknowns U be:

$$U = \begin{bmatrix} U^1 \\ U^2 \\ \vdots \\ U^{n-1} \end{bmatrix} \text{ where } U^j = \begin{bmatrix} U_{1,\,j} \\ U_{2,\,j} \\ \vdots \\ U_{n-1,\,j} \end{bmatrix} \text{ for each } j,\ 1 \le j \le n-1.$$

Then the set of equations

$$\triangle_h U_{ij} = f_{ij}, \quad i,j = 1,\ldots,n-1,$$

can be written in matrix form as

$$AU + p + q = f. \tag{13.6}$$

A is the $(n-1)\times(n-1)$ block tridiagonal matrix (of total size $(n-1)^2 \times (n-1)^2$) given by

$$\frac{1}{h^2}\begin{bmatrix} T & I & & & \\ I & T & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & T & I \\ & & & I & T \end{bmatrix} \tag{13.7}$$

where I is the $n-1 \times n-1$ identity matrix, and T is the $n-1 \times n-1$ tridiagonal matrix

$$\begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & 1 & -4 \end{bmatrix}.$$

The vectors p and q come from the boundary conditions of (13.3), and are given by

$$p = \begin{bmatrix} p^1 \\ \vdots \\ p^{n-1} \end{bmatrix}, \quad q = \begin{bmatrix} q^1 \\ \vdots \\ q^{n-1} \end{bmatrix},$$

where

$$p^j = \frac{1}{h^2}\begin{bmatrix} g(x_1, y_j) \\ 0 \\ \vdots \\ 0 \\ g(x_{n-1}, y_j) \end{bmatrix}, \quad 1 \le j \le n-1,$$

and

$$q^1 = \frac{1}{h^2}\begin{bmatrix} g(x_1, y_0) \\ g(x_2, y_0) \\ \vdots \\ g(x_{n-2}, y_0) \\ g(x_{n-1}, y_0) \end{bmatrix}, \quad q^{n-1} = \frac{1}{h^2}\begin{bmatrix} g(x_1, y_n) \\ g(x_2, y_n) \\ \vdots \\ g(x_{n-2}, y_n) \\ g(x_{n-1}, y_n) \end{bmatrix}, \quad q^j = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \quad 2 \le j \le n-2.$$

Note that both p and q have a total size of $(n-1)^2 \times 1$, and each p^j and q^j is $(n-1) \times 1$.

The vector f (which is the same size as p and q) comes from the source term of (13.3), and is given by

$$f = \begin{bmatrix} f^1 \\ \vdots \\ f^{n-1} \end{bmatrix}, \quad \text{where} \quad f^j = \begin{bmatrix} f(x_1, y_j) \\ f(x_2, y_j) \\ \vdots \\ f(x_{n-1}, y_j) \end{bmatrix}$$

Note that this linear system is very large (A has $(n-1)^4$ entries) and very sparse (most of the entries in A are zero). Thus we will should make use of sparse matrix routines (such as those in `scipy.sparse` and `scipy.sparse.linalg`) in order to reduce the time and memory used in setting up and solving the linear system.

Problem 1. Complete the function `poisson_square` by implementing the finite difference method (13.6), returning the approximate solution U as a an array. Use `scipy.sparse.linalg.spsolve` to solve the linear system. Use your function to solve the boundary value problem:

$$\begin{aligned} \Delta u &= 0, \quad x \in [0,1] \times [0,1], \\ u(x,y) &= x^3, \quad (x,y) \in \partial([0,1] \times [0,1]). \end{aligned} \tag{13.8}$$

Use $n = 100$ subintervals for both x and y. Plot the solution as a 3D surface.

Hint: `scipy.sparse.block_diag` and `scipy.sparse.diags` may be useful.

Poisson's equation and conservative forces

In physics Poisson's equation is used to describe the scalar potential of a conservative force. In general

$$\Delta V = -f$$

where V is the scalar potential of the force, or the potential energy a particle would have at that point, and f is a source term. Examples of conservative forces include Newton's Law of Gravity (where matter is the source term) and Coulomb's Law, which gives the force between two charge particles (where charge is the source term).

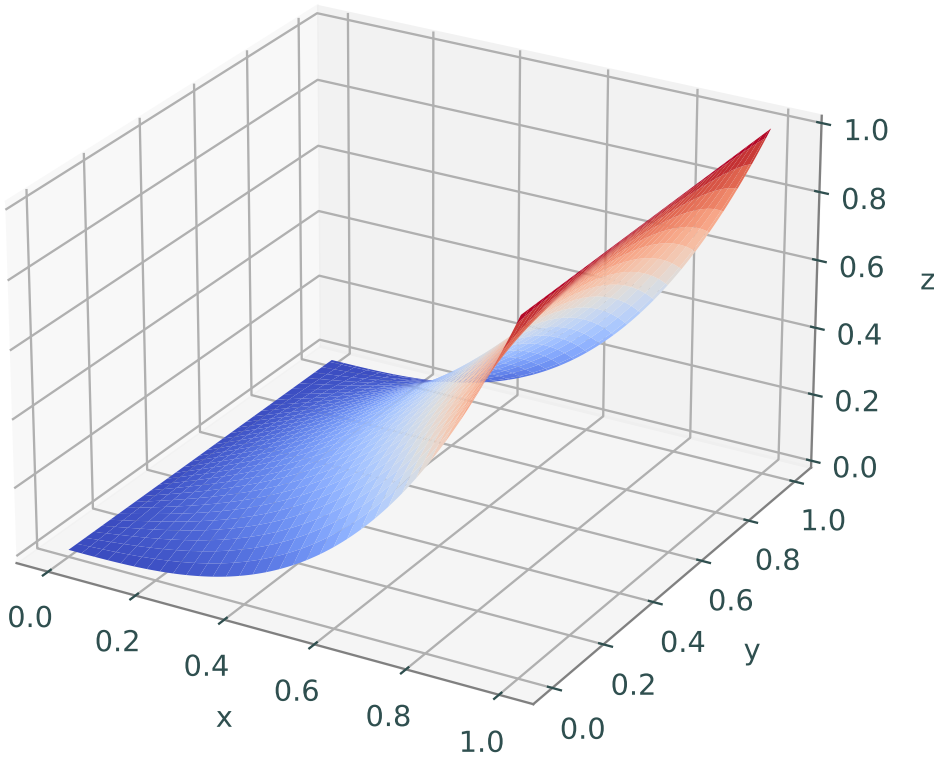


Figure 13.1: The solution of (13.8).

In electrostatics the electric potential is also known as the voltage, and is denoted by V. From Maxwell's equations it can be shown that that the voltage obeys Poisson's equation with the electric charge density (like a continuous cloud of electrons) being the source term:

$$\Delta V = -\frac{\rho}{\varepsilon_0},$$

where ρ is the charge density and ε_0 is the permissivity of free space, which is a constant that we'll leave as 1.

Usually a nonzero V at a point will cause a charged particle to move to a lower potential, changing ρ and the solution to V. However, in this analysis we'll assume that the charges are fixed in place.

Suppose we have 3 nested pipes. The outer pipe is attached to "ground," which usually we define to be $V = 0$, and the inner two have opposite relative charges. Physically the two inner pipes would function like a capacitor.

The following code will plot the charge distribution of this setup.

```
import matplotlib.colors as mcolors
def source(X, Y):
    """
    Takes arbitrary arrays of coordinates X and Y and returns an array of the ↩
        same shape
    representing the charge density of nested charged squares
    """
    src = np.zeros(X.shape)

    src[np.logical_or(np.logical_and(np.logical_or(abs(X - 1.5) < .1, abs(X + ↩
        1.5) < .1), abs(Y) < 1.6),
        np.logical_and(np.logical_or(abs(Y - 1.5) < .1, abs(Y + 1.5) < .1), abs↩
            (X) < 1.6))] = 1

    src[np.logical_or(np.logical_and(np.logical_or(abs(X - 0.9) < .1, abs(X + ↩
        0.9) < .1), abs(Y) <  1),
        np.logical_and(np.logical_or(abs(Y - 0.9) < .1, abs(Y + 0.9) < .1), abs↩
            (X) < 1))] = -1

    return src

# Generate a color dictionary for use with LinearSegmentedColormap
# that places red and blue at the min and max values of data
# and white when data is zero.
def genDict(data):
    zero = 1 / (1 - np.max(data) / np.min(data))
    cdict = {
        "red":   [(0, 1, 1), (zero, 1, 1), (1, 0, 0)],
        "green": [(0, 0, 0), (zero, 1, 1), (1, 0, 0)],
        "blue":  [(0, 0, 0), (zero, 1, 1), (1, 1, 1)]
    }
    return cdict

a1 = -2
b1 = 2
c1 = -2
d1 = 2
n = 100
X = np.linspace(a1, b1, n)
Y = np.linspace(c1, d1, n)
X, Y = np.meshgrid(X, Y)

plt.imshow(source(X, Y), cmap=mcolors.LinearSegmentedColormap("cmap", genDict(↩
    source(X, Y))))
```

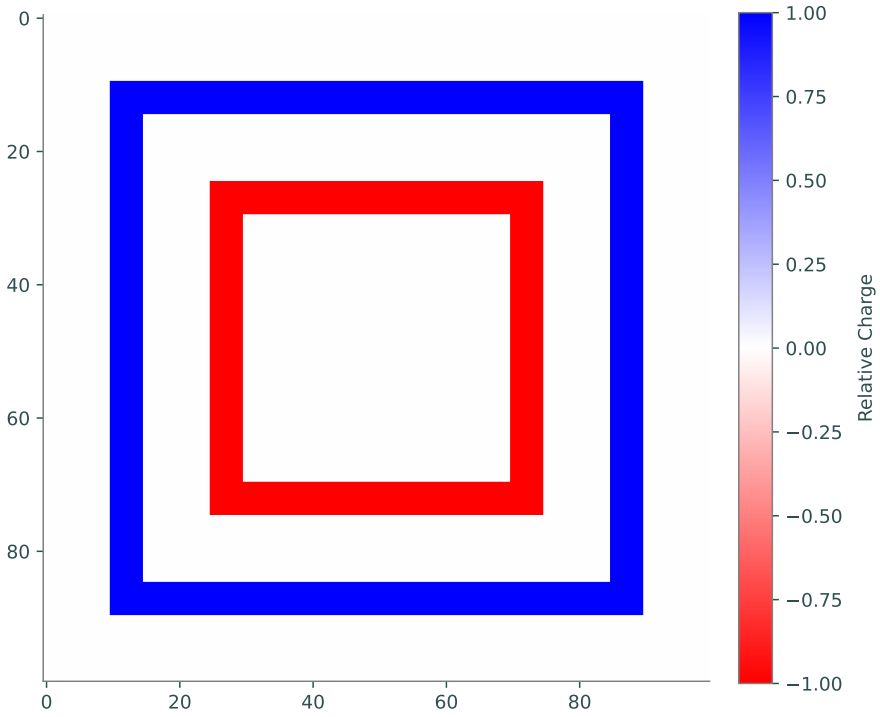


Figure 13.2: The charge density of the 3 nested pipes.

```
plt.colorbar(label="Relative Charge")
plt.show()
```

The function `genDict` scales the color values to be white when the charge density is zero. This is mostly to help visualize where there are neutrally charged zones by forcing them to be white. You may find it useful to also apply it when you solve for the electric potential.

With this definition of the charge density, we can solve Poisson's equation for the potential field.

Problem 2. Using the `poisson_square` function, solve

$$\begin{aligned} \Delta V &= -\rho(x,y), \quad x \in [-2,2] \times [-2,2], \\ u(x,y) &= 0, \quad (x,y) \in \partial([-2,2] \times [-2,2]). \end{aligned} \tag{13.9}$$

for the electric potential V. Use the source function defined above, such that $\rho(x,y) = \text{source}(x,y)$. Use $n = 100$ subintervals for x and y. Using the code provided above, plot your solution along with the `source` function. Compare your solution with Figure 13.3.

Poisson's Equation and Image Editing

The Poisson equation is also very useful for things outside of physics. For example, it can be used for image editing. We will use it to photoshop one image v onto another image u_0.

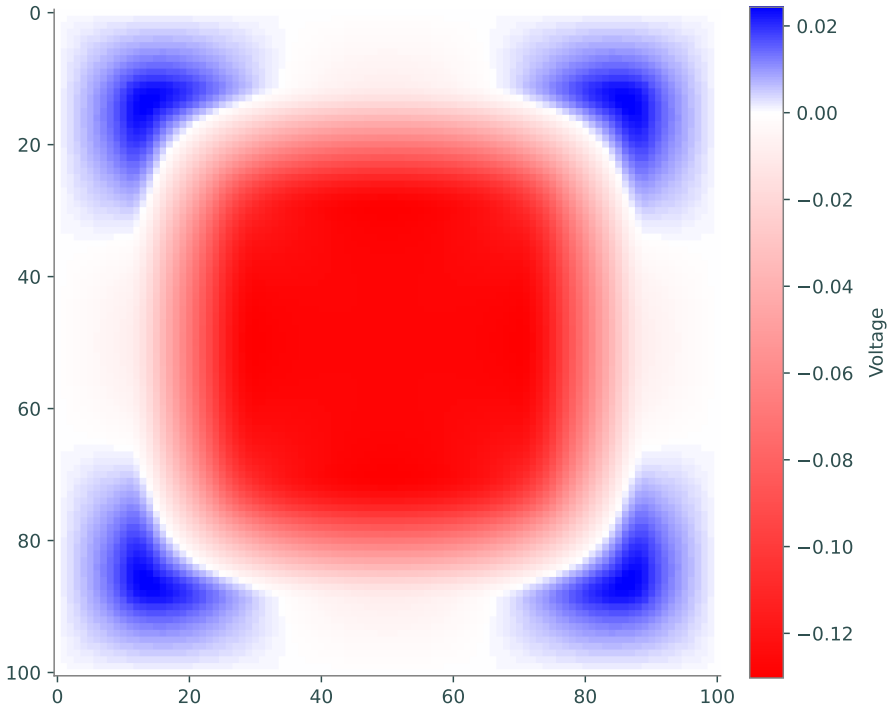


Figure 13.3: The electric potential of the 3 nested pipes.

Let $u_0 : S \to \mathbb{R}, S \subset \mathbb{R}^2$ be the original image, which is a 2D grid of values between 0 and 255. We want to insert a new image, $v : P \to \mathbb{R}$ into $P' \subset S$. Numerically, we will treat these as arrays where $v \in M_{m \times n}$, $u_0 \in M_{m' \times n'}, m' > m, n' > n$, and $u \in M_{m \times n}$ is the smoothed image inside of u_0. A visual representation can be seen in Figure 13.4.

Achtung!

Note that u_0 is defined on P' since $P' \subset S$.

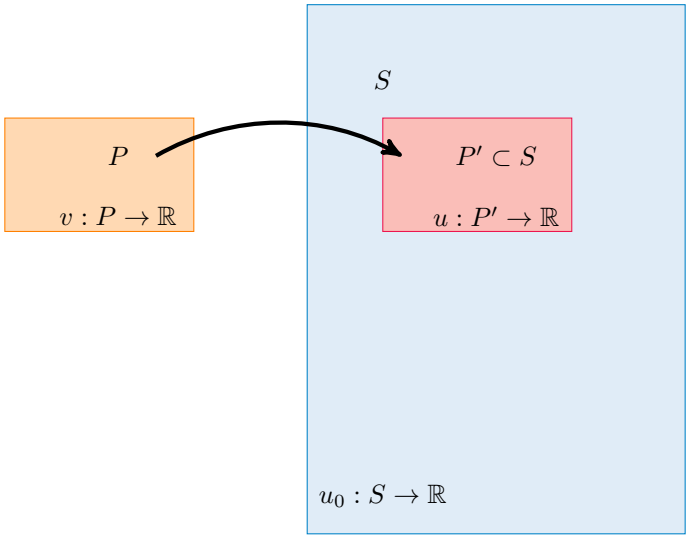


Figure 13.4: Photoshopping of v into the space defined by P' by solving for u.

We will use v to create the source function f in Equation (13.3).

$$f(x, y) = \triangle v(x, y). \tag{13.10}$$

We want to use the original picture u_0 to constrain the boundary of P', so let g from Equation (13.3) be defined as

$$g(x, y) = u_0(x, y), \quad (x, y) \in \partial P'. \tag{13.11}$$

Using these equations, we can blend an image v in very well. There are, of course, ways to make this better (but also more complicated), such as adding some of the original texture from u_0 to the final image u, but we'll leave those aside for now. If you're interested, see this paper for more information on using the Poisson equation for image editing.

Problem 3. Using the data file `dr_jarvis.jpg` as the source image v and `mount_rushmore.jpg` as the destination image u_0, put Dr. Jarvis' face on Mount Rushmore using Poisson's equation to blend it in.

We'll follow a similar process to what we did in Problem 1. Use equation (13.5) (letting $h = 1$) with equation (13.10) to calculate $f(x, y)$ from v. Consruct the matrices T and A. Then note that instead of flattening the source function $f(x, y)$ and the boundary conditions $g(x, y)$ into vectors f, p, and q to insert into the matrix equation $AU = f - p - q$ (see (13.6)), we can instead subtract the boundary condition matrix $g(x, y)$ from the source function matrix $f(x, y)$ first and then flatten. That is, construct a matrix

$$r(x, y) = \begin{cases} f(x, y), & (x, y) \in P' \\ f(x, y) - g(x, y), & (x, y) \in \partial P', \end{cases}$$

then flatten

$$r = \begin{bmatrix} r^1 \\ \vdots \\ r^{n-1} \end{bmatrix}, \quad r^j = \begin{bmatrix} r(x_1, y_j) \\ \vdots \\ r(x_{n-1}, y_j) \end{bmatrix},$$

and finally solve $AU = r$.

Hint: Consider the region P' of the original image to be the set of matrix indices included in `[x0, x0+w-1] × [y0, y0+w-1]`. Then the boundary $\partial P'$ is the border along the rows `x0-1` and `x0+w` and along the columns `y0-1` and `y0+w`.

Hint: You will only want to use part of the image from `dr_jarvis.jpg` to paste onto Mount Rushmore. The following code will help you import the image, select the appropriate part, and paste it in the correct place so that it looks like the image in Figure 13.5. Note that you will need to transpose the image again before displaying it.

```
source_im = np.mean(imageio.v3.imread("dr_jarvis.jpg"), axis=2).transpose↩
    () / 255
dest_im = np.mean(imageio.v3.imread("mount_rushmore.jpg"), axis=2).↩
    transpose() / 255

# Width of space (number of pixels) to replace in destination image
w = 130

# Position in destination image
x0 = 322
y0 = 215

# Position in source image
x0s = 60
y0s = 84

# Show original image
plt.imshow(dest_im.transpose(), cmap="gray")
plt.show()

# Source image with a buffer of 1 pixel for the finite difference method.
# The buffer will be excluded when inserting into the Mount Rushmore image↩
    .
# The "*0.58" will make it look better when displayed.
image = source_im[x0s-1 : x0s+w+1, y0s-1 : y0s+w+1] * 0.58

# Calculate f(x, y)...

# Calculate the solution U...

# Paste Dr. Jarvis into the original image
new_image = dest_im.copy()
new_image[x0:x0+w, y0:y0+w] = U.reshape(w,w)
```

```
plt.imshow(new_image.transpose(), cmap="gray")
plt.show()
```

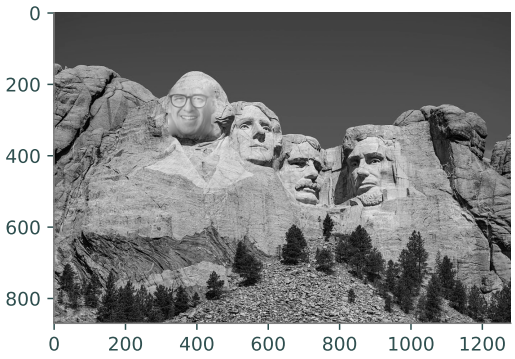


Figure 13.5: A Founding Father. Also the solution to Problem 3.

14 Spectral 1: Method of Mean Weighted Residuals

Lab Objective: *We introduce the method of mean weighted residuals (MWR) and use it to derive a pseudospectral method. This method will then be used to solve several boundary value problems.*

Consider a linear differential equation

$$Lu = f$$

defined on the interval $[-1, 1]$, together with associated boundary conditions. We will approximate the solution $u(x)$ by a linear combination of $N + 1$ basis functions ϕ_i, so that

$$u(x) \approx u_N(x) = \sum_{i=0}^{N} a_i \phi_i(x).$$

To determine appropriate constants a_i, we then minimize the residual function

$$R(x, u_N) = Lu_N - f.$$

Note that $R(x, u) = Lu - f = 0$ for the true solution $u(x)$.

This general strategy is often called the method of mean weighted residuals (MWR method). The MWR method is a general framework that describes many other, more specific methods. These more specific methods come from differing approaches to minimizing the residual $R(x, u_N)$, and the choice of basis functions ϕ_i.

The Pseudospectral Method

The pseudospectral or collocation method is obtained from the MWR method by forcing the residual function $R(x, u_N)$ to equal zero at $N + 1$ points in $[-1, 1]$, called collocation points. When done correctly, the pseudospectral method gives high accuracy and converges rapidly.

Let the collocation points be the Gauss-Lobatto points, $x_i = \cos(\pi i/N)$, $i = 0, \dots, N$. The appropriate solution u_N may be represented with two equivalent forms. First, u_N can be described by the first $N+1$ coefficients $\{a_i\}_{i=0}^N$ of its expansion with a polynomial basis ϕ_i (e.g. the Chebyshev Polynomials). Second, since u_N is a polynomial of order N, it may also be uniquely described by its values at the collocation points, that is, the unknown values $\{u_N(x_i)\}_{i=0}^N$.

These equivalent forms satisfy

$$MA = F \tag{14.1}$$

and

$$LU = F \tag{14.2}$$

where

$$\begin{aligned} U_i &= u(x_i) \\ A_i &= a_i \\ F_i &= f(x_i) \\ L_{ij} &= LC_j(x)\Big|_{x=x_i} \\ M_{ij} &= L\phi_j(x)\Big|_{x=x_i} \end{aligned}$$

The functions C_j above are the cardinal functions, defined to be the polynomials of least degree satisfying

$$C_j(x_i) = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$$

Thus, u_N can also be expanded in the basis of cardinal functions

$$u_N(x) = \sum_{j=0}^{N} u_N(x_j) C_j(x),$$

which effectively becomes the Lagrange interpolating polynomials at the Gauss-Lobatto points. When $L = d/dx$, the matrix corresponding to equation (14.2) is given by

$$L_{ij} = \frac{dC_j}{dx}(x_i) = \begin{cases} (1 + 2N^2) \ / \ 6 & i = j = 0, \\ -(1 + 2N^2) \ / \ 6 & i = j = N, \\ -x_j \ / \ [2(1 - x_j^2)] & i = j, \ 0 < j < N, \\ (-1)^{i+j} \alpha_i \ / \ [\alpha_j (x_i - x_j)] & i \neq j. \end{cases}$$

where $\alpha_0 = \alpha_N = 2$, and $\alpha_j = 1$ otherwise.

This matrix is often called the differentiation matrix (D), and can be used to piece together the matrix L for more complicated differential operators. A stable, vectorized function to build the differentiation matrix is given below.

```
import numpy as np

def cheb(N):
    x = np.cos((np.pi/N)*np.linspace(0, N, N+1))
    x.shape = (N+1, 1)
    lin = np.linspace(0, N, N+1)
    lin.shape = (N+1, 1)

    c = np.ones((N+1, 1))
    c[0], c[-1] = 2., 2.
    c = c*(-1.)**lin
    X = x*np.ones(N+1) # Broadcast along 2nd dimension (columns)
```

```
    dX = X - X.T

    D = (c*(1./c).T)/(dX + np.eye(N+1))
    D = D - np.diag(np.sum(D.T, axis=0))
    x.shape = (N+1,)
    # Here we return the differentiation matrix and the Chebyshev points,
    # numbered from x_0 = -1 to x_N = 1
    return D[::-1, ::-1], x[::-1]
```

Problem 1. Use the differentiation matrix to numerically approximate the derivative of $u(x) = e^x \cos(6x)$ on a grid of N Chebyshev points where $N = 6, 8$, and 10. (Use the linear system $DU \approx U'$.) Then use barycentric interpolation (`scipy.interpolate.barycentric_interpolate`) to approximate u' on a grid of 100 evenly spaced points.

Graphically compare your approximation to the exact derivative. Note that this convergence would not be occurring if the collocation points were equally spaced.

To approximate $u''(x)$ on the grid $\{x_i\}$, we use

$$U'' \approx D^2 U.$$

The BVP

$$\begin{aligned} u'' &= f(x), \quad x \in [-1, 1], \\ u(-1) &= 0, \quad u(1) = 0, \end{aligned}$$

can be discretized by the linear system

$$D^2 U = F, \tag{14.3}$$

where $F = [f(x_0), \ldots, f(x_N)]^\mathsf{T}$. Since we have Dirichlet boundary conditions of 0, we can satisfy the boundary condition by forcing $U[0] = U[N] = 0$. This is done by replacing the first and last equations in (14.3) by the boundary conditions.

```
# The following code will force U[0] = U[N] = 0
D, x = cheb(N)    # For some N
D2 = np.dot(D, D)
D2[0, :], D2[-1, :] = 0, 0
D2[0, 0], D2[-1, -1] = 1, 1
F[0], F[-1] = 0, 0
```

Problem 2. Use the pseudospectral method to solve the boundary value problem

$$\begin{aligned} u'' &= e^{2x}, \quad x \in (-1, 1), \\ u(-1) &= 0, \quad u(1) = 0. \end{aligned}$$

Use $N = 8$ in the `cheb(N)` method and use barycentric interpolation to approximate u on 100 evenly spaced points. Compare your numerical solution with the exact solution,

$$u(x) = \frac{-\cosh(2) - \sinh(2)x + e^{2x}}{4}.$$

Problem 3. Use the pseudospectral method to solve the boundary value problem

$$u'' + u' = e^{3x}, \quad x \in (-1, 1),$$
$$u(-1) = 2, \quad u(1) = -1.$$

Use $N = 8$ in the `cheb(N)` method and use barycentric interpolation to approximate u on 100 evenly spaced points.

Hint: Apply the Dirichlet boundary conditions to the left-hand side of the equation only *after* adding the differentiation matrices $D + D^2$.

The previous exercise involved setting up and solving a linear system

$$AU = F,$$

where F is a vector whose entries are e^{3x} evaluated at the collocation points x_j, and U represents the approximation to the solution u at those points. However, whenever the ODE is nonlinear, the discretization becomes a nonlinear system of equations that must be solved using Newton's method. The next exercise contains a BVP whose ODE is nonlinear, with the additional complexity that the domain of the problem is not $[-1, 1]$.

Problem 4. Use the pseudospectral method to solve the boundary value problem

$$u'' = \lambda \sinh(\lambda u), \quad x \in (0, 1),$$
$$u(0) = 0, \quad u(1) = 1$$

for several values of λ: $\lambda = 4, 8, 12$. Begin by transforming this BVP onto the domain $-1 < x < 1$. Use $N = 20$ in the `cheb(N)` method and use barycentric interpolation to approximate u on 100 evenly spaced points.

Below is sample code for implementing Newton's Method

```
from scipy.optimize import root

N = 20
D, x = cheb(20)

def F(U):
    out = None  # Set up the equation you want the root of.
    # Make sure to set the boundaries correctly

    return out  # Newtons Method updates U until the output is all 0's.
```

```
guess = np.ones_like(x)    # Your guess is same size as the cheb(N) output
solution = root(F, guess).x
```

Hint: use an array of ones for your guess, as shown in the code above.

Minimizing the Area of a Surface of Revolution

A surface of revolution that minimizes its area is an example of a larger class of surfaces called minimal surfaces. A famous example of a minimal surface is a soap bubble. Soap bubbles minimize their surface area while containing a fixed volume of air. This behavior extends to merged bubbles, and a soap film whose boundary is a wire frame. Minimal surfaces have applications in molecular engineering and material science, and general relativity, where they describe the apparent horizon of a black hole.

Consider a function $y(x)$ defined on $[-1, 1]$ satisfying $y(-1) = a$, $y(1) = b$. The area of the surface obtained by revolving the graph of $y(x)$ about the x-axis is given by

$$T[y(x)] = \int_{-1}^{1} 2\pi y(x)\sqrt{1 + (y'(x))^2}\, dx.$$

To find the function $y(x)$ whose surface of revolution minimizes surface area, we must minimize the functional $T[y]$. This is a classical problem from a branch of mathematics called the calculus of variations. Standard derivatives allow us to find the minimum values of functions defined on $\mathbb{R}^n$, and where they occur. The calculus of variations allows us to find the minimum values of functions whose input are other functions.

From the calculus of variations we know that a necessary condition for $y(x)$ to minimize $T[y]$ is that the Euler-Lagrange equation must be satisfied:

$$L_y - \frac{d}{dx}L_{y'} = 0,$$

where $L(x, y, y') = 2\pi y\sqrt{1 + (y')^2}$. Simplifying the Euler-Lagrange equation for our problem results in the ODE

$$yy'' - (y')^2 - 1 = 0.$$

Discretizing this ODE using the pseudospectral method results in the (nonlinear) system of equations

$$Y \odot (D^2Y) - (DY) \odot (DY) - \mathbf{1} = 0,$$

where $\odot$ is the Hadamard product (denoting element-wise multiplication) and $\mathbf{1}$ is a vector of ones.

Problem 5. Find the function $y(x)$ that satisfies $y(-1) = 1$, $y(1) = 7$, and whose surface of revolution (about the x-axis) minimizes surface area. Compute the surface area, and plot the surface. Use $N = 50$ in the `cheb(N)` method and use barycentric interpolation to approximate u on 100 evenly spaced points. Your solution should look like Figure 14.1

Below is sample code for creating the 3D wireframe figure.

```
barycentric = None # This is the output of barycentric_interpolate() on ↩
    100 points
```

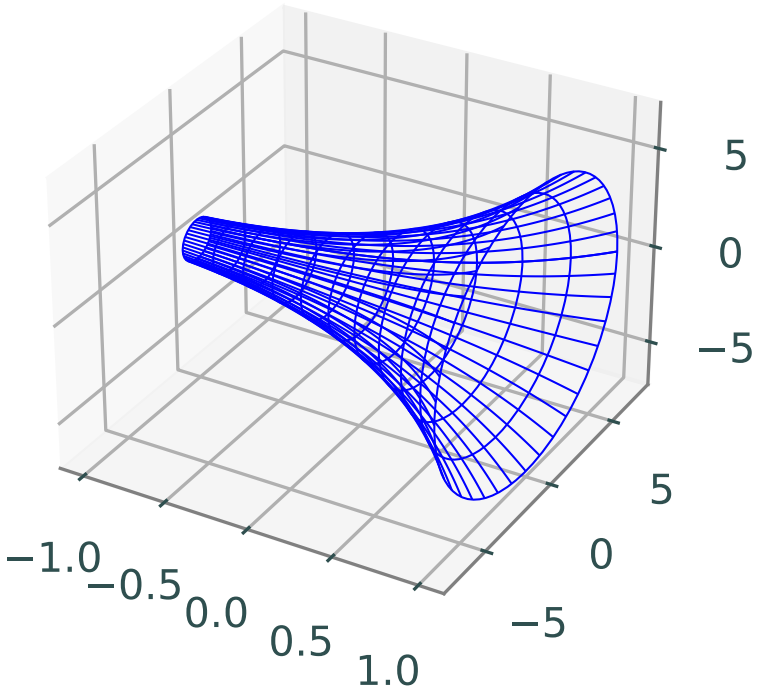


Figure 14.1: The minimal surface corresponding to Problem 5.

```
lin = np.linspace(-1, 1, 100)
theta = np.linspace(0, 2*np.pi, 401)
X, T = np.meshgrid(lin, theta)
Y, Z = barycentric*np.cos(T), barycentric*np.sin(T)
fig = plt.figure()
ax = fig.add_subplot(projection="3d")
ax.plot_wireframe(X, Y, Z, color='b', rstride=10, cstride=10, lw=0.5)
plt.show()
```

Hint: This problem is sensitive to initial conditions; use a vector of twos as the initial guess.

15 Spectral 2: A Pseudospectral Method for Periodic Functions

Lab Objective: *We look at a pseudospectral method with a Fourier basis, and numerically solve two PDEs using a pseudospectral discretization in space and a Runge-Kutta integration scheme in time.*

Let f be a periodic function on $[0, 2\pi]$. Let $x_1, \dots, x_N$ be N evenly spaced grid points on $[0, 2\pi]$. Since f is periodic on $[0, 2\pi]$, we can ignore the grid point $x_N = 2\pi$. We will further assume that N is even; similar formulas can be derived for N odd. Let $h = 2\pi/N$; then $\{x_0, \dots, x_{N-1}\} = \{0, h, 2h, \dots, 2\pi - h\}$.

The discrete Fourier transform (DFT) of f, denoted by $\hat{f}$ or $\mathcal{F}(f)$, is given by

$$\hat{f}(k) = h \sum_{j=0}^{N-1} e^{-ikx_j} f(x_j) \quad \text{where } k = -N/2 + 1, \dots, 0, 1, \dots, N/2.$$

The inverse DFT is then given by

$$f(x_j) = \frac{1}{2\pi} \sum_{k=-N/2}^{N/2} \frac{e^{ikx_j}}{c_k} \hat{f}(k), \quad j = 0, \dots, N-1, \tag{15.1}$$

where

$$c_k = \begin{cases} 2 & \text{if } k = -N/2 \text{ or } k = N/2, \\ 1 & \text{otherwise.} \end{cases} \tag{15.2}$$

The inverse DFT can then be used to define a natural interpolant (sometimes called a band-limited interpolant) by evaluating (15.1) at any x rather than x_j:

$$p(x) = \frac{1}{2\pi} \sum_{k=-N/2}^{N/2} e^{ikx} \hat{f}(k). \tag{15.3}$$

The interpolant for f' is then given by

$$p'(x) = \frac{1}{2\pi} \sum_{k=-N/2+1}^{N/2-1} ike^{ikx} \hat{f}(k). \tag{15.4}$$

Consider the function $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$. Using (15.4), the derivative u' may be approximated with the following code. [1] We note that although we only approximate u' at the Fourier grid points, (15.4) provides an analytic approximation of u' in the form of a trigonometric polynomial.

```
import numpy as np
from scipy.fftpack import fft, ifft
import matplotlib.pyplot as plt

N=24
x1 = (2.*np.pi/N)*np.arange(1, N+1)
f = np.sin(x1)**2.*np.cos(x1) + np.exp(2.*np.sin(x1+1))

# This array is reordered in Python to
# accomodate the ordering inside the fft function in scipy.
k = np.concatenate(( np.arange(0, N/2) ,
                     np.array([0])  , # Because hat{f}'(k) at k = N/2 is zero.
                     np.arange(-N/2+1, 0, 1)))

# Approximates the derivative using the pseudospectral method
f_hat = fft(f)
fp_hat = ((1j*k)*f_hat)
fp = np.real(ifft(fp_hat))

# Calculates the derivative analytically
x2 = np.linspace(0, 2*np.pi, 200)
derivative = (2.*np.sin(x2)*np.cos(x2)**2. -
                 np.sin(x2)**3. +
                 2*np.cos(x2+1)*np.exp(2*np.sin(x2+1))
                 )

plt.plot(x2, derivative, "-k", linewidth=2.)
plt.plot(x1, fp, "*b")
plt.savefig("spectral2_derivative.pdf")
plt.show()
```

[1] See *Spectral Methods in MATLAB* by Lloyd N. Trefethen. Another good reference is *Chebyshev and Fourier Spectral Methods* by John P. Boyd.

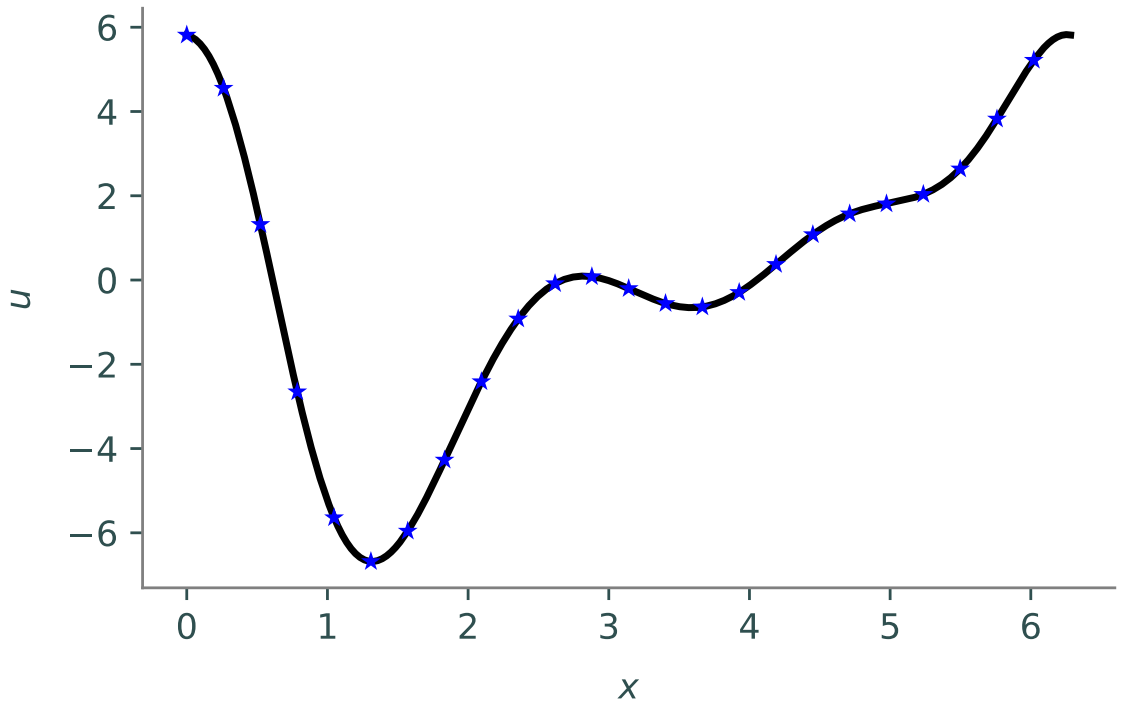


Figure 15.1: The derivative of $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$.

Problem 1. Consider again the function $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$. Create a function that approximates $\frac{1}{2}u'' - u'$ on the Fourier grid points for $N = 24$.

The advection equation

Recall that the advection equation is given by

$$u_t + cu_x = 0 \tag{15.5}$$

where c is the speed of the wave (the wave travels to the right for $c > 0$). We will consider the solution of the advection equation on the domain $[0, 2\pi]$ and assume periodic boundary conditions.

A common method for solving time-dependent PDEs is called the *method of lines*. To apply the method of lines to our problem, we use our Fourier grid points in $[0, 2\pi]$: given an even N, let $h = 2\pi/N$, so that $\{x_0, \dots, x_{N-1}\} = \{0, h, 2h, \dots, 2\pi - h\}$. By using these grid points we obtain the collection of equations

$$u_t(x_j, t) + cu_x(x_j, t) = 0, \quad t > 0, \quad j = 0, \dots N - 1. \tag{15.6}$$

Let $U(t)$ be the vector-valued function given by $U(t) = (u(x_j, t))_{j=0}^{N-1}$. Let $\mathcal{F}(U)(t)$ denote the discrete Fourier transform of $u(x, t)$ (in space), so that

$$\mathcal{F}(U)(t) = (\hat{u}(k, t))_{k=-N/2+1}^{N/2}.$$

Define $\mathcal{F}^{-1}$ similarly. Using the pseudospectral approximation in space leads to the system of ODEs

$$U_t + \vec{c}\mathcal{F}^{-1}\left(i\vec{k}\mathcal{F}(U)\right) = 0 \tag{15.7}$$

where $\vec{k}$ is a vector, and $\vec{k}\mathcal{F}(U)$ denotes element-wise multiplication. Similarly $\vec{c}$ could also be a vector, if the wave speed c is allowed to vary. We can then use an ODE solver for the time derivative and the pseudospectral method for spatial derivatives.

Problem 2. Using `solve_ivp`, solve the initial value problem

$$u_t + c(x)u_x = 0, \tag{15.8}$$

where $c(x) = 0.2 + \sin^2(x-1)$, and $u(x, t=0) = e^{-100(x-1)^2}$. Plot your numerical solution from $t = 0$ to $t = 8$ over 250 time steps and 200 x steps. Note that the initial data is nearly zero near $x = 0$ and 2π, and so we can use the pseudospectral method.[a] Use the following code to help graph. The solution can be seen in Figure 15.2.

```
t_steps = 250    # Time steps
x_steps = 200     # x steps

'''
Your code here to set things up
'''

sol = # use solve_ivp

X, Y = np.meshgrid(x_domain, t_domain)
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
surf = ax.plot_surface(X, Y, np.transpose(sol.y), cmap="coolwarm")
ax.set_zlim(0, 3)
ax.view_init(elev=40, azim=280, roll=0)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$T$")
ax.set_zlabel(r"$z$")
ax.set_box_aspect(aspect=None, zoom=0.8)
plt.show()
```

[a] This problem is solved in *Spectral Methods in MATLAB* using a leapfrog discretization in time.

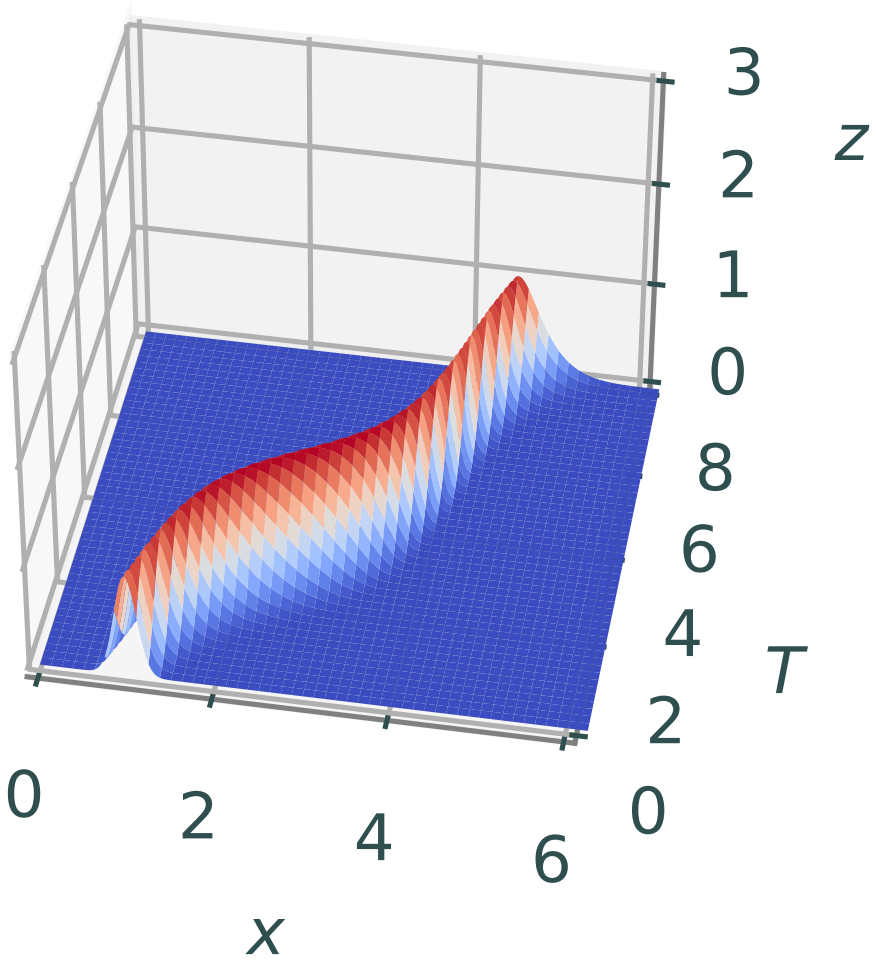


Figure 15.2: The solution of the variable speed advection equation; see Problem 2.

The wave equation

We can use the pseudospectral method to solve higher-order PDEs as well. A common example of a second-order hyperbolic PDE is the wave equation, which is often used in physics. The wave equation is given by

$$u_{tt} - cu_{xx} = 0,$$

where c is the wave speed. Unlike the advection equation, waves that encounter mediums of different wave speeds will reflect part of the wave back. We will see this in Problem 3. First, we must change this second-order method-of-lines ODE equation into a first-order equation. Using the *method of lines*, the wave equation can be written as a system of first-order ODEs,

$$\frac{\partial}{\partial t}\begin{bmatrix} u \\ u_t \end{bmatrix} = \begin{bmatrix} u_t \\ cu_{xx} \end{bmatrix}. \tag{15.9}$$

We can then use an ODE solver, such as `solve_ivp` to solve for the time derivatives, while we use the spectral method for the spatial derivatives.

Problem 3. Using `solve_ivp`, solve the initial value problem

$$u_{tt} - c(x)u_{xx} = 0, \tag{15.10}$$

where

$$c(x) = \begin{cases} 4 & 0 \le x < \pi \\ 1 & \pi \le x < 2\pi \end{cases},$$

with $u(x, t = 0) = 0.2e^{(-10(x-5)^2)}$ and $u_t(x, t = 0) = -4(x-5)e^{(-10(x-5)^2)}$. Plot your numerical solution from $t = 0$ to $t = 3$ over 150 time steps and 100 x steps. As in the previous problem, the initial data is nearly zero near $x = 0$ and 2π, and so we can use the pseudospectral method (the solution is approximately periodic because the boundaries both stay near zero). Use the code provided in the previous problem (but with different numbers of steps in the x and t directions) to help with plotting. The solution can be seen in Figure 15.3. Note that the wave speeds up at the barrier, and some of it is reflected.

Hints: The initial conditions for `solve_ivp` need to be in a one-dimensional array. Concatenate the initial conditions into a single $(100 + 100) \times 1$ array to start `solve_ivp`. You will need to make corresponding changes in the RHS derivative function (the first parameter of `solve_ivp`) by returning a $(100+100)\times 1$ array, where the first 100 elements are the derivative $\frac{\partial}{\partial t}u$, and the second 100 elements are the derivative $\frac{\partial}{\partial t}u_t$.

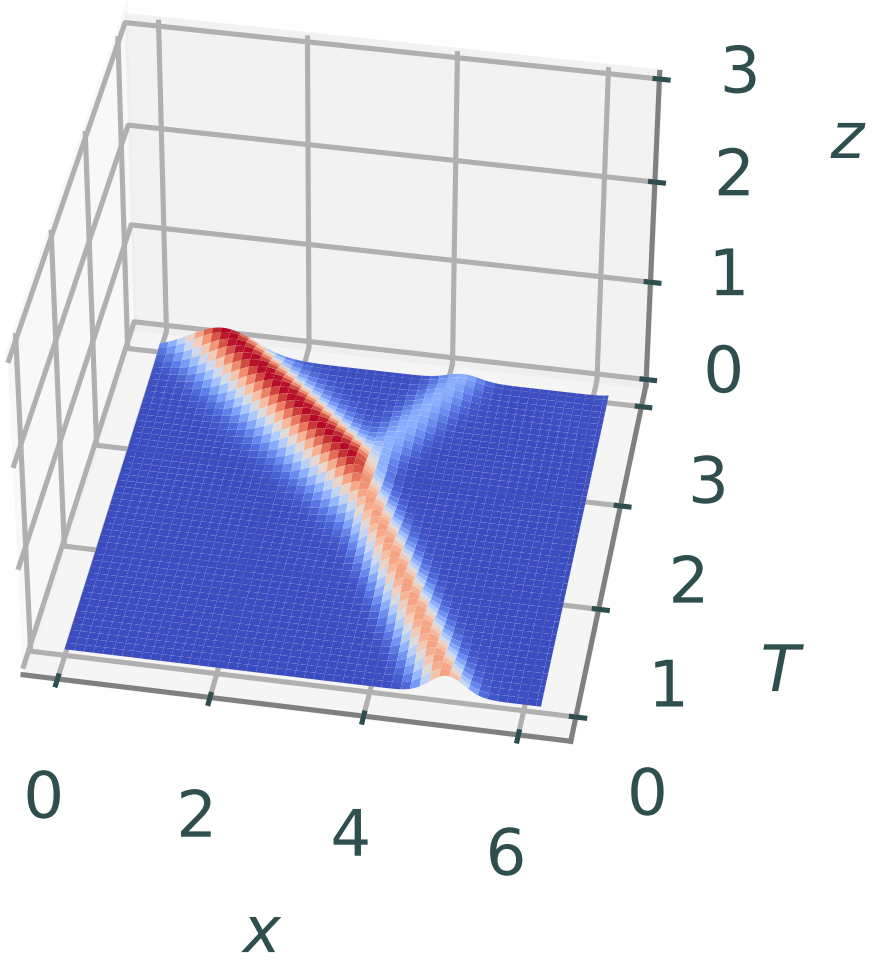


Figure 15.3: The solution of the wave equation for a pulse hitting a barrier at $x = \pi$; see Problem 3.

16 Inverse Problems

An important concept in mathematics is the idea of a well posed problem. The concept initially came from Jacques Hadamard. A mathematical problem is *well posed* if

1. a solution exists,
2. that solution is unique, and
3. the solution is continuously dependent on the data in the problem.

A problem that is not well posed is *ill posed*. Notice that a problem may be well posed, and yet still possess the property that small changes in the data result in larger changes in the solution; in this case the problem is said to be ill conditioned, and has a large condition number.

Note that for a physical phenomena, a well posed mathematical model would seem to be a necessary requirement! However, there are important examples of mathematical problems that are ill posed. For example, consider the process of differentiation. Given a function u together with its derivative u', let $\tilde{u}(t) = u(t) + \varepsilon \sin(\varepsilon^{-2} t)$ for some small $\varepsilon > 0$. Then note that

$$\|u - \tilde{u}\|_\infty = \varepsilon,$$

while

$$\|u' - \tilde{u}'\|_\infty = \varepsilon^{-1}.$$

Since a small change in the data leads to an arbitrarily large change in the output, differentiation is an ill posed problem. And we haven't even mentioned numerically approximating a derivative!

For an example of an ill posed problem from PDEs, consider the backwards heat equation with zero Dirichlet conditions:

$$\begin{aligned} &u_t = -u_{xx}, \quad (x,t) \in (0,L) \times (0,\infty), \\ &u(0,t) = u(L,t) = 0, \quad t \in (0,\infty), \\ &u(x,0) = f(x), \quad x \in (0,L). \end{aligned} \tag{16.1}$$

For the initial data $f(x) = 0$, the unique[1] solution is $u(x,t) = 0$. Given the initial data $f(x) = \frac{1}{n}\sin(\frac{n\pi x}{L})$, one can check that there is a unique solution $u(x,t) = \frac{1}{n}\sin(\frac{n\pi x}{L})\exp((\frac{n\pi}{L})^2 t)$. Thus, on a finite interval $[0,T]$, as $n \to \infty$ we see that a small difference in the initial data results in an arbitrarily large difference in the solution.

[1] See *Partial Differential Equations* by Lawrence C. Evans, chapter 2.3, for a proof of uniqueness.

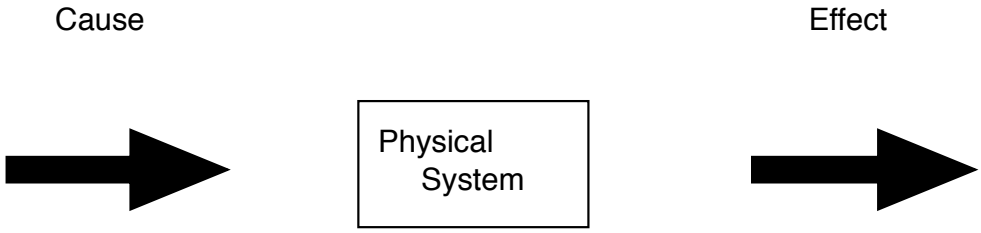


Figure 16.1: Cause and effect within a given physical system.

Inverse Problems

As implied by the name, inverse problems come in pairs. For example, differentiation and integration are inverse problems. The easier problem (in this case integration) is often called the direct problem. Historically, the direct problem is usually studied first.

Given a physical system, together with initial data (the "cause"), the direct problem will usually predict the future state of the physical system (the "effect"); see Figure 16.1. Inverse problems often turn this on its head—given the current state of a physical system at time T, what was the physical state at time $t = 0$?

Alternatively, suppose we measure the current state of the system, and we then measure the state at some future time. An important inverse problem is to determine an appropriate mathematical model that can describe the evolution of the system.

Another look at heat flow through a rod

Consider the following ordinary differential equation, together with natural boundary conditions at the ends of the interval[2]:

$$\begin{cases} -(au')' = f, & x \in (0,1), \\ a(0)u'(0) = c_0, & a(1)u'(1) = c_1. \end{cases} \tag{16.2}$$

This BVP can, for example, be used to describe the flow of heat through a rod. The boundary conditions would correspond to specifying the heat flux through the ends of the rod. The function $f(x)$ would then represent external heat sources along the rod, and $a(x)$ the thermal conductivity of the rod at each point.

[2]This example of an ill-posed problem is given in *Inverse Problems in the Mathematical Sciences* by Charles W Groetsch.

Typically, the thermal conductivity $a(x)$ would be specified, along with any heat sources $f(x)$, and the (direct) problem is to solve for the steady-state heat distribution $u(x)$. Here we shake things up a bit: suppose the heat sources f are given, and we can measure the heat distribution $u(x)$. Can we find the thermal conductivity of the rod? This is an example of a *parameter estimation problem.*

Let us consider a numerical method for solving (16.2) for the thermal conductivity $a(x)$. Subdivide $[0,1]$ into N equal subintervals, and let $x_j = jh$, $j = 0, \ldots, N$, where $h = 1/N$. Let $\phi_j(x)$ be the tent functions (used earlier in the finite element lab), given by

$$\phi_j(x) = \begin{cases} (x - x_{j-1})/h & x \in [x_{j-1}, x_j], \\ (x_{j+1} - x)/h & x \in [x_j, x_{j+1}], \\ 0 & \text{otherwise.} \end{cases}$$

We look for an approximation $a^h(x)$ that is a linear combination of tent functions. This will be of the form

$$a^h = \sum_{j=0}^{N} \alpha_j \phi_j, \quad \alpha_j = a(x_j). \tag{16.3}$$

The h in this equation indicates that each of the tent functions in the linear combination rely on $h = 1/N$, and that a different h or N will result in different tent functions, so a^h will be different. The second half of (16.3) says that a good choice of a^h is found by taking $\alpha_j = a(x_j)$. Integrating (16.2) from 0 to x, we obtain

$$\begin{aligned} \int_0^x -(au')'\, ds &= \int_0^x f(s)\, ds, \\ -\left[a(x)u'(x) - c_0\right] &= \int_0^x f(s)\, ds, \\ u'(x) &= \frac{c_0 - \int_0^x f(s)\, ds}{a(x)}. \end{aligned} \tag{16.4}$$

Thus for each x_j

$$\begin{aligned} u'(x_j) &= \frac{c_0 - \int_0^{x_j} f(s)\, ds}{a(x_j)}, \\ &= \frac{c_0 - \int_0^{x_j} f(s)\, ds}{\alpha_j}. \end{aligned}$$

The coefficients α_j in (16.3) can now be approximated as α_j^* by minimizing

$$\alpha_j^* = \arg\min_{\alpha_j} \left\{ \left(\frac{c_0 - \int_0^{x_j} f(s)\, ds}{\alpha_j} - u'(x_j) \right)^2 \right\}. \tag{16.5}$$

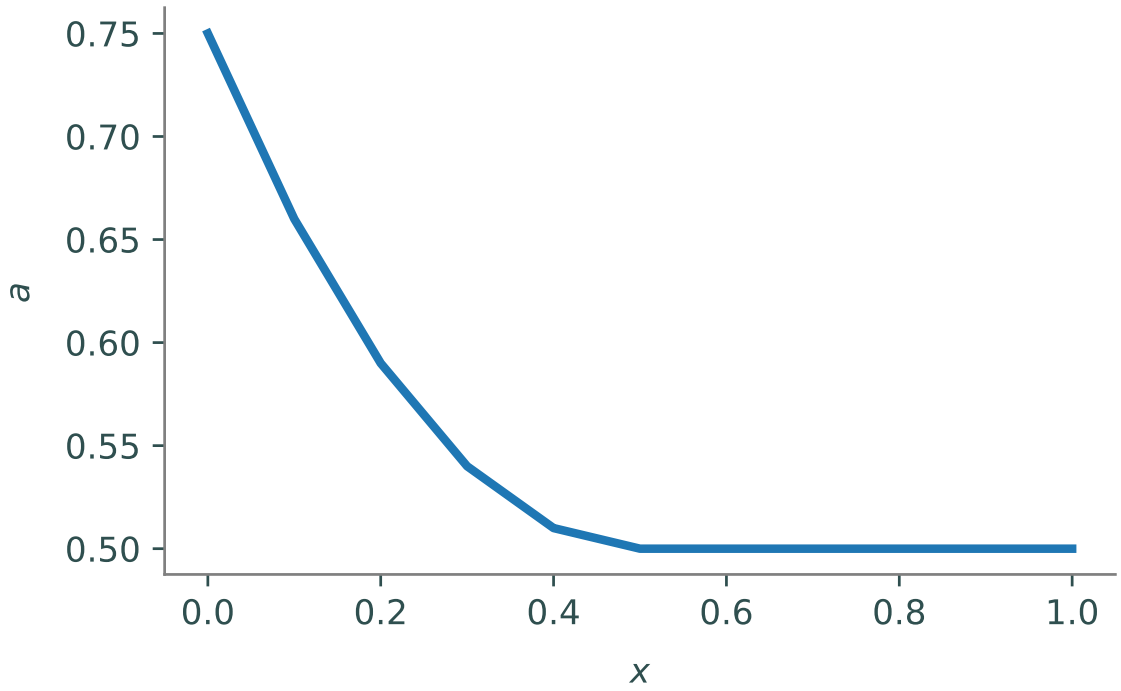


Figure 16.2: The solution $a(x)$ to Problem 1

Problem 1. Use (16.5) to solve (16.2) for $a(x)$ using the following conditions: $c_0 = 3/8$, $c_1 = 5/4$, $u(x) = x^2 + x/2 + 5/16$, $x_j = 0.1j$ for $j = 0, 1, \dots, 10$, and

$$f = \begin{cases} -6x^2 + 3x - 1 & x \leq 1/2, \\ -1 & 1/2 < x \leq 1, \end{cases}$$

Produce the plot shown in Figure 16.2.

Hint: use the `minimize` function in `scipy.optimize` and some initial guess to find the α_j. Alternatively, notice that (16.5) is simple enough that it can be solved explicitly for α_j.

Problem 2. Find the thermal conductivity function $a(x)$ satisfying

$$\begin{cases} -(au')' = -1, & x \in (0,1), \\ a(0)u'(0) = 1, & a(1)u'(1) = 2. \end{cases} \tag{16.6}$$

where $u(x) = x + 1 + \varepsilon \sin(\varepsilon^{-2}x)$. Using several values of $\varepsilon > 0.66049142$, plot the corresponding thermal conductivity $a(x)$ for x in `np.linspace(0,1,11)` to demonstrate that the problem is ill-posed, as in Figure 16.3. Be sure to add a legend to your plot.

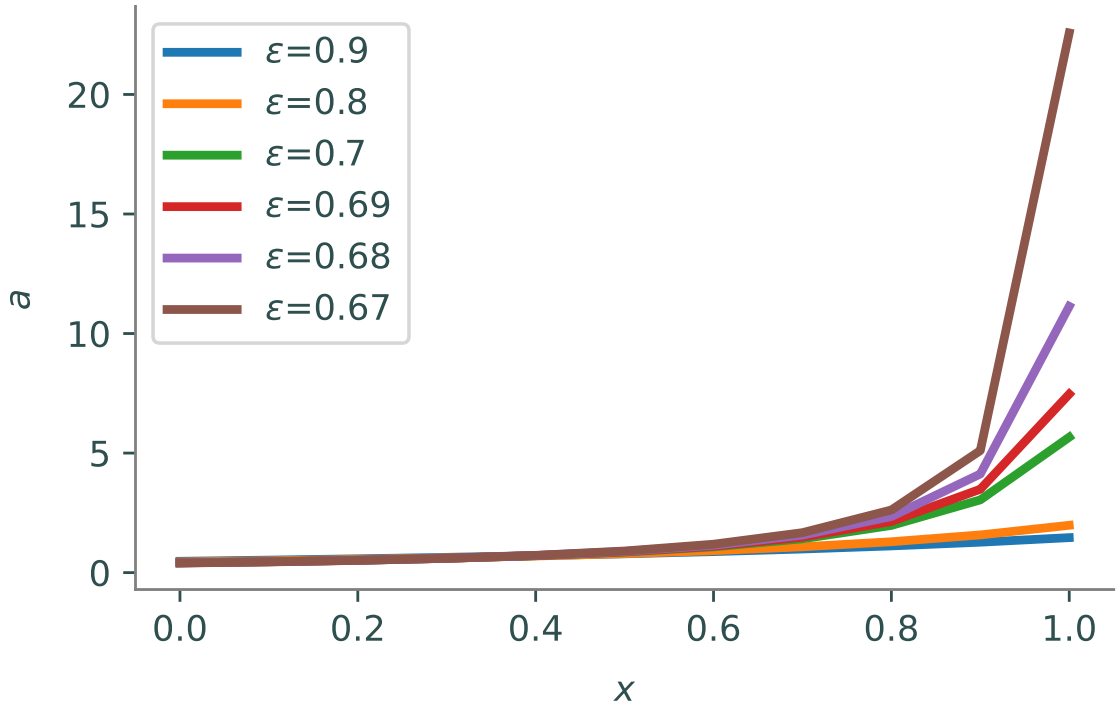


Figure 16.3: The thermal conductivity function $a(x)$ satisfying (16.6) for $\varepsilon = 0.8$.

Time-Dependent Inverse Problems

We can also expand our use of inverse problems from time-independent PDEs to time-dependent variables. In this section of the lab, we will do it with the nonlinear time-dependent diffusion equation. We will also introduce noise to the measurements, making this much more realistic.

Time-Dependent Heat Diffusion Update

The nonlinear time-dependent diffusion equation is

$$\begin{aligned}\frac{\partial}{\partial t}u(x,t) &= \frac{\partial}{\partial x}\left(\nu(x)\frac{\partial u(x,t)}{\partial x}\right)\\ &= \left(\frac{\partial \nu(x)}{\partial x}\right)\left(\frac{\partial u(x,t)}{\partial x}\right) + \nu(x)\left(\frac{\partial^2 u(x,t)}{\partial x^2}\right).\end{aligned} \tag{16.7}$$

Note that this form of the diffusion equation is slightly more general than the one in the heat diffusion lab, since the diffusion coefficient ν is not a constant.

If our spatial grid is indexed by $j \in \{0,\dots,J-1\}$ with step h, and our time grid is indexed by $m \in \{0,\dots,M-1\}$ with step k, then the time derivative can be approximated as

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

and the second-order centered difference approximation for the second spatial derivative is

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

The second-order centered difference approximations for the first spatial derivatives (of u and ν) are

$$u_x(x_j, t_m) = \frac{u(x_j + h, t_m) - u(x_j - h, t_m)}{2h} + \mathcal{O}(h^2)$$
$$\nu_x(x_j) = \frac{\nu(x_j + h) - \nu(x_j - h)}{2h} + \mathcal{O}(h^2).$$

Letting $\vec{\nu}$ be a vector of $\nu_j = \nu(x_j)$ evaluated at each grid point in space, the update iteration scheme then becomes

$$U_j^{m+1}(\vec{\nu}) = U_j^m + \frac{k}{h^2}\left[\nu_j(U_{j+1}^m - 2U_j^m + U_{j-1}^m) + \left(\frac{(\nu_{j+1} - \nu_{j-1})(U_{j+1}^m - U_{j-1}^m)}{4}\right)\right]. \tag{16.8}$$

where we omit (or suppress) the dependence of U_j^m, U_{j+1}^m, and U_{j-1}^m on $\vec{\nu}$ for brevity. Note that this update equation is the same as the one in the Heat Diffusion Lab, but with an added last term.

Inverse Problems with Noise

We want to find the heat diffusion coefficient $\vec{\nu}$ that minimizes the difference between the measured heat $\hat{U}$ and our approximation $U(\vec{\nu})$ which we compute using (16.8). That is, we want to find $\vec{\nu}$ such that the sum of squared errors

$$\text{SSE} = \sum_{j=0}^{J-1}\sum_{m=0}^{M-1}[U_j^m(\vec{\nu}) - \hat{U}_j^m]^2 \tag{16.9}$$

is minimized. Note that since $\hat{U}$ is a measurement of a true heat distribution $u^*(x, t)$, it is subject to noise. Also note that this is a least-squares problem (like Problem 1), but the solution is not trivial since each ν_j is constant in time, but multiple noisy measurements are taken at the same point x_j at different times t_m.

To solve this problem, we will choose a guess for our initial heat diffusion coefficient, solve for the heat flow using our guess, and then using a minimization method (such as BFGS), we will update our vector of parameters, $\vec{\nu}$. Here is some code to help get you started:

```
# This is an MxJ matrix of measured, noisy data.
U_hat = np.load("measured_heat.npy")
M, J = U_hat.shape

def sse(nu, U_hat):

    # Initialize U with the the first row (time) of `U_hat`, then loop
    # forward in time using `nu` and equation (8).

    return # Result of equation (9)

# This is a good guess. There are others that will work fine too.
guess = np.full(J, 2)

sol = minimize(sse, guess, args=U_hat, method="BFGS")
nu = sol.x
```

Problem 3. The file `measured_heat.npy` contains measurements over time ($t \in [0, 2]$) of the temperature along an insulated rod with a varying diffusion coefficient $\nu(x)$. The first row corresponds to the iniital state of the heat distribution in the rod. Use equation (16.8) and the code above to find an estimate $\vec{\nu}$ of $\nu(x)$ that minimizes the sum of squared errors (16.9). For $x \in [-10, 10]$, plot $\vec{\nu}$ along with the true $\nu(x)$, which is given by

$$\nu(x) = \frac{5}{1 + e^{-x}} + \frac{1}{2}.$$

Compare with Figure 16.4. Be sure to add a legend to your plot. Notice that the model does worse at the endpoints. This is partially because the endpoints in the PDE are fixed at zero with Dirichlet conditions, so the value of ν at the endpoints has little little effect on the evolution of the PDE.

Hint: Use array broadcasting when possible. Note that the array in `measured_heat.npy` is 21×11, so $M = 21$ and $J = 11$.

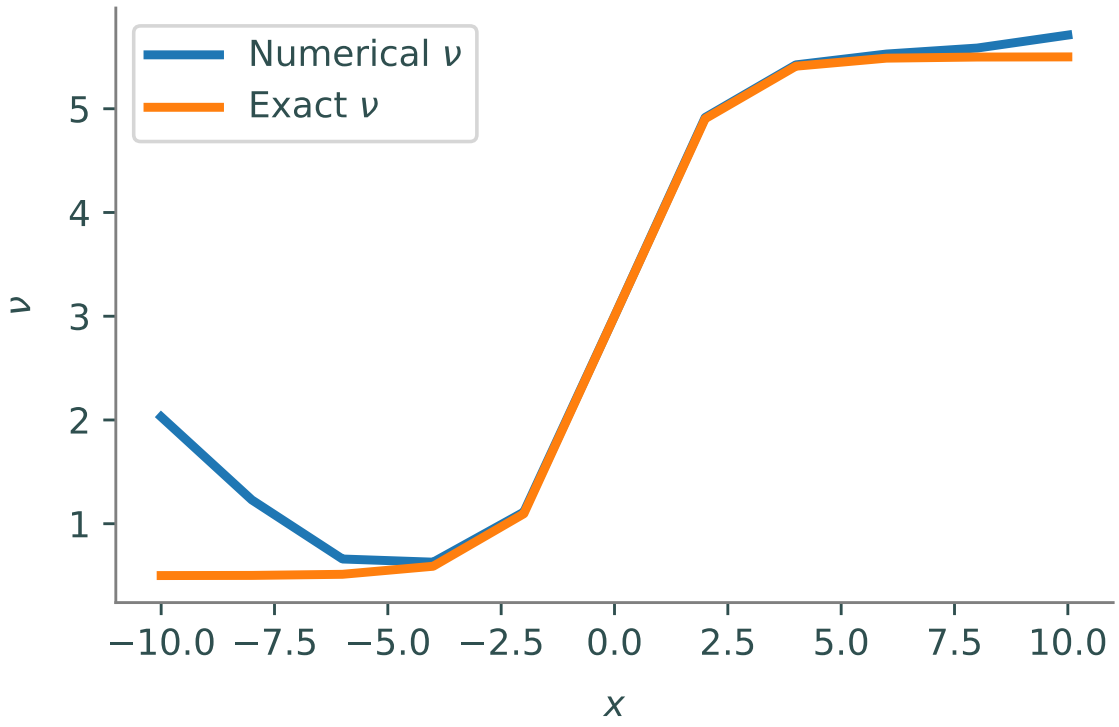


Figure 16.4: Solution to Problem 3

17 The Shooting Method for Boundary Value Problems

Consider a boundary value problem of the form

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \le x \le b, \\ y(a) &= \alpha, \quad y(b) = \beta. \end{aligned} \tag{17.1}$$

One natural way to approach this problem is to study the initial value problem (IVP) associated with this differential equation:

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \le x \le b, \\ y(a) &= \alpha, \quad y'(a) = s. \end{aligned} \tag{17.2}$$

The goal is to determine an appropriate value s so that the solution of the IVP (17.2) is also a solution of the BVP (17.1).

Note that we can consider the value $y(x)$ for a given initial condition $y'(a) = s$ as a function of both x and s. Let $y(x, s)$ be the solution of (17.2). The initial value conditions are then

$$y(a, s) = \alpha, \quad \frac{\partial y}{\partial x}(a, s) = s. \tag{17.3}$$

We wish to find a value of s so that $y(b, s) = \beta$. Consider the function $h(s) = y(b, s) - \beta$; this function is called the *residual function*. If $h(s) = 0$, then $y(b, s) = \beta$ and the boundary condition is satisfied, so zeros of the function h correspond to initial conditions that are solutions to the BVP (17.1). Applying Newton's method to the function $h(s)$, we obtain the iterative method

$$\begin{aligned} s_{n+1} &= s_n - \frac{h(s_n)}{h'(s_n)}, \\ &= s_n - \frac{y(b, s_n) - \beta}{\frac{\partial}{\partial s} y(b, s)\big|_{s_n}}, \quad n = 0, 1, \ldots. \end{aligned}$$

Provided our initial guess s_0 is sufficiently good, this will converge to a value of s such that the initial value problem is also a solution to the boundary value problem. Notice that finding $y(b, s_n)$ requires solving the initial value problem using RK4 or some other method.

We recall that Newton's method generally requires a good initial guess s_0. A plausible initial guess for this setup would be the average rate of change of the solution across the entire interval, which gives $s_0 = (\beta - \alpha)/(b - a)$. If this initial guess is insufficient, it may be refined by manually inspecting the solution $y(x, s_0)$ of the initial value problem.

Using Newton's method requires us to evaluate or approximate the function $h'(s_n)$. This term may be approximated with a finite difference $h'(s_n) \approx \frac{h(s_n)-h(s_{n-1})}{s_n-s_{n-1}}$, giving us the iterative method

$$\begin{aligned} s_{n+1} &= s_n - h(s_n)\frac{(s_n - s_{n-1})}{h(s_n) - h(s_{n-1})} \\ &= s_n - (y(b, s_n) - \beta)\frac{s_n - s_{n-1}}{y(b, s_n) - y(b, s_{n-1})}, \quad n = 1, 2, \dots \end{aligned}$$

This variation of Newton's method is called the *secant method*, and requires two initial values instead of one. The secant method generally does not converge as quickly as standard Newton's method, but it avoids needing to compute the actual derivative of $h(s)$.

As an example, consider the boundary value problem

$$\begin{aligned} y'' &= -4y - 9\sin(x), \ x \in [0, 3\pi/4], \\ y(0) &= 1, \\ y(3\pi/4) &= -\frac{1+3\sqrt{2}}{2}. \end{aligned} \tag{17.4}$$

This has the exact solution

$$y(x) = \cos(2x) + \frac{1}{2}\sin(2x) - 3\sin(x).$$

The following code implements the secant method to solve (17.4) numerically. We use `scipy.integrate.solve_ivp` to solve the initial value problems.

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

# Secant method
def secant_method(h, s0, s1, max_iter=100, tol=1e-8):
    """
    Finds a root of h(s)=0 using the secant method with the
    initial guesses s0, s1.
    """
    for i in range(max_iter):
        # Get the residuals
        h0 = h(s0)
        h1 = h(s1)
        # Update
        s2 = s1 - h1 * (s1 - s0)/(h1 - h0)
        s0, s1 = s1, s2

        # Check convergence
        if abs(h1) < tol:
            return s2

    print("Secant method did not converge")
    return s2
```

```
# Define the ODE right-hand side
def ode(x, y):
    return np.array([
        y[1],
        -4*y[0]-9*np.sin(x)
    ])

# Endpoint values
a = 0
b = 3/4 * np.pi
alpha = 1
beta =  - (1+3*np.sqrt(2))/2

# Define a residual function
def residual(s):
    # Find the right endpoint
    sol = solve_ivp(ode, (a, b), [alpha, s])
    yb = sol.y[0,-1]
    return yb - beta

# Find the right value of s using the secant method
s = secant_method(residual, (beta-alpha)/2, -1)

# Compute and plot the solution
x = np.linspace(0,3*np.pi/4,100)
y = solve_ivp(ode, (a, b), (alpha, s), t_eval=x).y[0]

plt.plot(x, y)
plt.show()
```

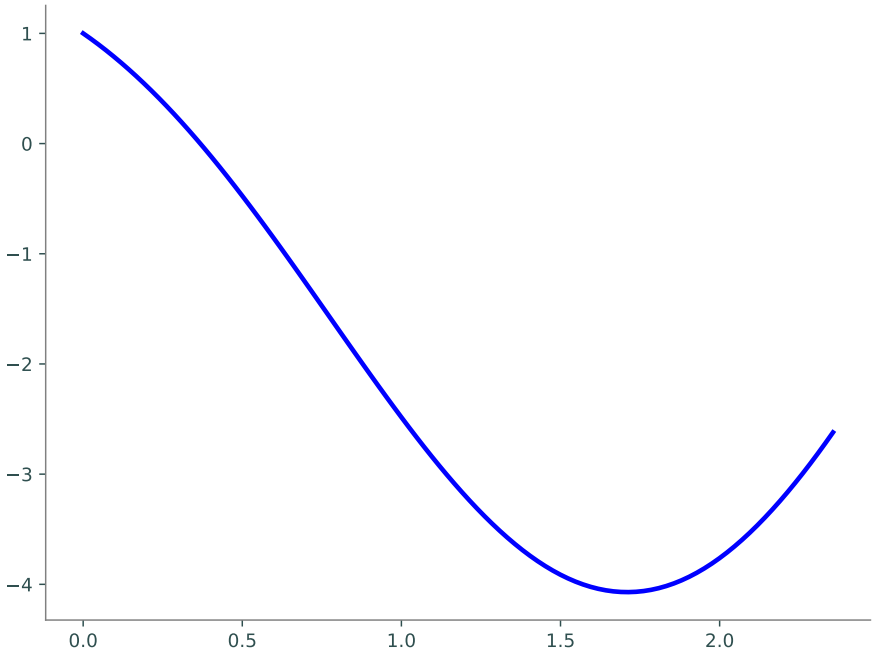


Figure 17.1: The solution to the BVP (17.4) from the above example.

Problem 1. Appropriately defined initial value problems will usually have a unique solution. Boundary value problems are not so straightforward; they may have no solution or they may have several, and you may have to determine which solutions are physically interesting.

Use the secant method to solve the following BVP:[a]

$$y'' = -e^{y-1}, \quad x \in [0,1],$$
$$y(0) = y(1) = 1.$$

This BVP has two solutions. Using the secant method, find both numerical solutions and print their initial slopes. Plot the solutions and compare with Figure 17.2. What initial values s_0, s_1 did you use to find them?

[a]This example is from *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations* by Ascher, Mattheij, and Russell, page 89.

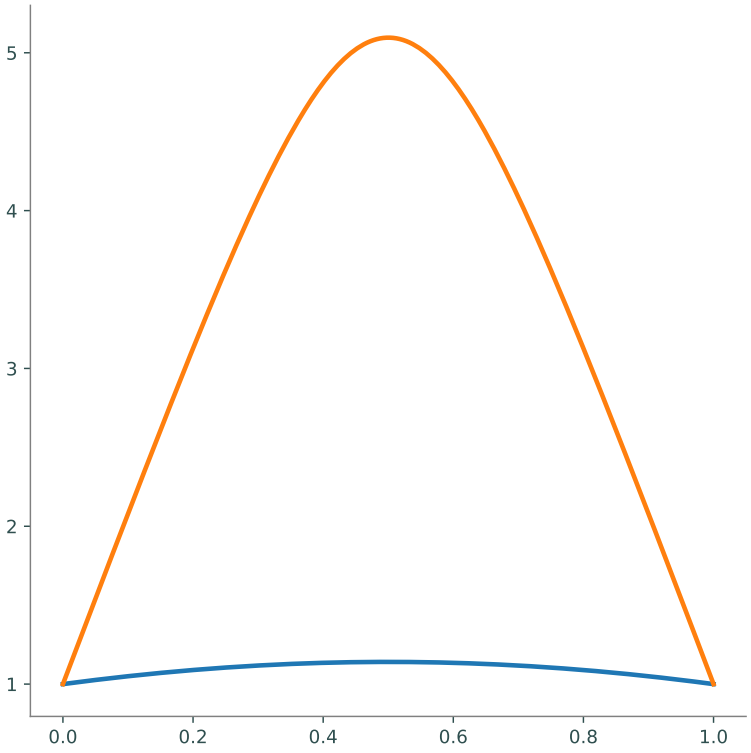


Figure 17.2: Both solutions to the boundary value problem given in Problem 1.

Instead of using the secant method, let us consider how to solve for $h'(s) = \frac{\partial}{\partial s}y(b,s)$ for problems of the form given in (17.1). For typical systems of ODEs, the solution $y(x,s)$ is smooth enough that it can be differentiated with respect to x and s in any order.[1] Let $z(x,s) = \frac{\partial}{\partial s}y(x,s)$, and note that $h'(s) = z(b,s)$. Using the chain rule, we obtain

$$\begin{aligned} z'' = \frac{\partial}{\partial s}y''(x,s) &= \frac{\partial f}{\partial y}(x, y(x,s), y'(x,s)) \cdot \frac{dy}{ds}(x,s), \\ &+ \frac{\partial f}{\partial y'}(x, y(x,s), y'(x,s)) \cdot \frac{\partial y'}{\partial s}(x,s), \end{aligned}$$

Using the initial conditions associated with $y(x,s)$ and noting that $z(x,s) = \frac{\partial}{\partial s}y(x,s)$ and $z'(x,s) = \frac{\partial}{\partial s}y'(x,s)$, we obtain the following initial value problem for $z(x,s)$:

$$\begin{aligned} z'' &= z\frac{\partial f}{\partial y}(x,y,y') + z'\frac{\partial f}{\partial y'}(x,y,y'), \quad a \le x \le b, \\ z(a,s) &= 0,\ z'(a,s) = 1. \end{aligned}$$

[1] This is guaranteed to be the case if the right hand side of the ODE is C^1, as in all of the examples here, as this guarantees both partial derivatives of y are continuous.

To use Newton's method, the IVPs for y and z must be solved simultaneously. The iterative method then becomes

$$\begin{aligned} s_{n+1} &= s_n - \frac{h(s)}{h'(s)} \\ &= s_n - \frac{y(b, s_n) - \beta}{z(b, s_n)}, \ n = 0, 1, \dots \end{aligned} \tag{17.5}$$

We will run through an example to demonstrate this method. Let

$$\begin{aligned} y'' &= 3 + \frac{2y}{x^2}, \ x \in [1, e], \\ y(1) &= 6, \\ y(e) &= e^2 + \frac{6}{e}, \end{aligned}$$

and let $s = y'(1)$. Then

$$f = y'' = 3 + \frac{2y}{x^2},$$

and

$$\begin{aligned} h(s) &= y(b, s) = y(e, s), \\ h'(s) &= \frac{\partial}{\partial s} y(e, s). \end{aligned}$$

We then solve iteratively for s using Newton's method, starting with an initial guess s_0. With each iteration, we need to solve the initial value problem for y and z, given an s_n, using the first order system defined by

$$\begin{bmatrix} y \\ y' \\ z \\ z' \end{bmatrix}' = \begin{bmatrix} y' \\ 3 + \frac{2y}{x^2} \\ z' \\ z\frac{\partial f}{\partial y}(x, y, y') + z'\frac{\partial f}{\partial y'}(x, y, y') \end{bmatrix} = \begin{bmatrix} y' \\ 3 + \frac{2y}{x^2} \\ z' \\ \frac{2z}{x^2} \end{bmatrix},$$

$$\begin{aligned} z(1, s_n) = 0, \quad z'(1, s_n) = 1, \\ y(1, s_n) = 6, \quad y'(1, s_n) = s_n. \end{aligned}$$

We then use the solutions for $y(x, s_n)$ and $z(x, s_n)$ to find s_{n+1}, using equation (17.5), and iterate.

Problem 2. Use Newton's method to solve the BVP

$$\begin{aligned} y'' &= 3 + \frac{2y}{x^2}, \ x \in [1, e], \\ y(1) &= 6, \\ y(e) &= e^2 + \frac{6}{e}. \end{aligned}$$

Plot your solution, and compare with Figure 17.3. What is an appropriate initial guess?

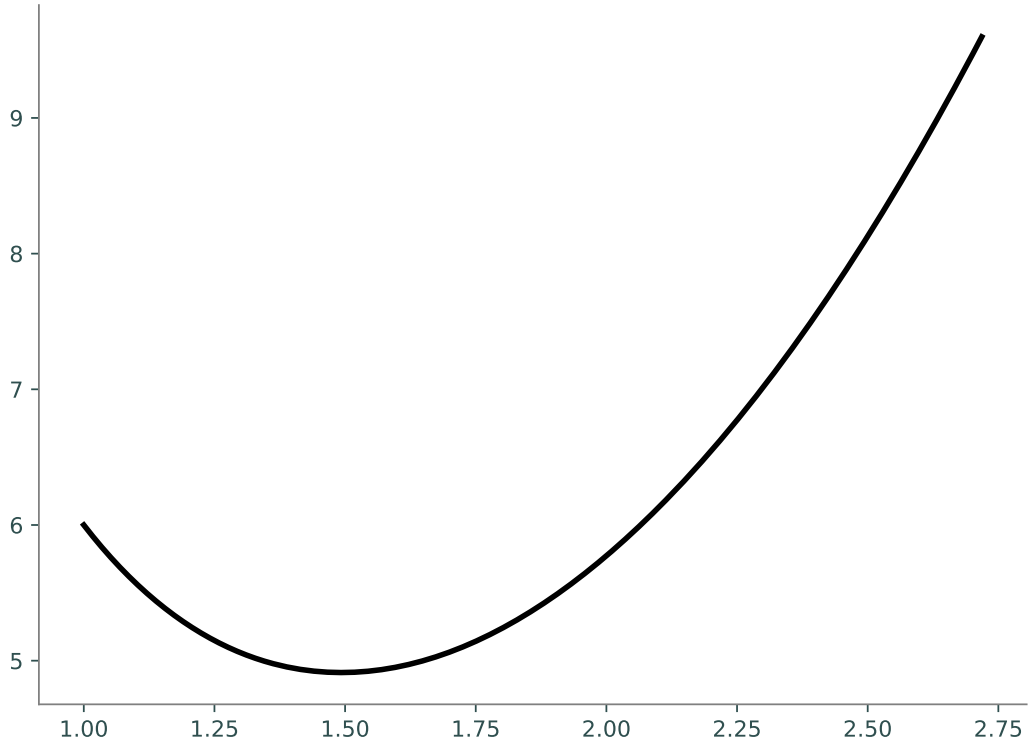


Figure 17.3: The solution of $y'' = 3 + 2y/x^2$, satisfying the boundary conditions $y(1) = 6$, $y(e) = e^2 + 6/e$, given in Problem 2.

The Cannon Problem

Consider the problem of aiming a projectile at a given target. Here we will construct a differential equation that describes the path of the projectile and takes into account air resistance. We will then use the shooting method to determine the angle at which the projectile should be launched.

Let t denote time, and the coordinates of the projectile be given by $\mathbf{r}(t) = (x(t), y(t))$. If $\theta(t)$ represents the angle of the velocity vector from the positive x-axis and $v(t) = \|\mathbf{v}(t)\|$ represents the speed of the projectile, then we have

$$\dot{x} = v\cos\theta,$$
$$\dot{y} = v\sin\theta.$$

Note that each of x, y, θ, and v are functions of t, so the dot denotes $\frac{d}{dt}$. The tangent vector to the path traced by the projectile is the unit vector in the direction of the projectile's velocity, so $\mathbf{T}(t) = (\cos\theta, \sin\theta)$. The unit normal vector $\mathbf{N}(t)$ is given by $\mathbf{N}(t) = (-\sin\theta, \cos\theta)$. Thus the relationship between basis vectors $\mathbf{i}, \mathbf{j}$, and $\mathbf{T}(\mathbf{t}), \mathbf{N}(t)$ is given by

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ \mathbf{j} \end{bmatrix} = \begin{bmatrix} \mathbf{T}(\mathbf{t}) \\ \mathbf{N}(\mathbf{t}) \end{bmatrix}$$

Let F_g represent the force on the projectile due to gravity, and F_d represent the force on the projectile due to air resistance. (We assume the air is still.) From Newton's law we have

$$m\dot{\mathbf{v}} = F_g + F_d.$$

The drag equation from fluid dynamics says that the force on the projectile due to air resistance is $kv^2 = (1/2)\rho c_D A v^2$, where ρ is the mass density of air (about 1.225 kg/m^3), v is the speed of the projectile, and A is its cross-sectional area. The drag coefficient c_D is a dimensionless quantity that changes with respect to the shape of the object. (If we assume our projectile is spherical with a diameter of .2 m, then its drag coefficient $c_D \approx 0.47$, its cross-sectional area is $\pi/100$ m^2, and we obtain $k \approx 0.009$.)

Thus the total force on the shell is

$$\begin{aligned} m\dot{\mathbf{v}} &= -mg\mathbf{j} - kv^2\mathbf{T}, \\ &= -mg(\mathbf{T}\sin\theta + \mathbf{N}\cos\theta) - kv^2\mathbf{T}, \\ &= (-mg\sin\theta - kv^2)\mathbf{T} - mg\mathbf{N}\cos\theta. \end{aligned} \tag{17.6}$$

From the identity $\mathbf{v} = (\dot{x}, \dot{y}) = (v\cos\theta, v\sin\theta)$ we have

$$\begin{aligned} m\dot{\mathbf{v}} &= m(\dot{v}\cos\theta - v\dot{\theta}\sin\theta, \dot{v}\sin\theta + v\dot{\theta}\cos\theta) \\ &= m(\dot{v}\cos\theta - v\dot{\theta}\sin\theta)(\cos\theta\mathbf{T} - \sin\theta\mathbf{N}) \\ &\quad + m(\dot{v}\sin\theta + v\dot{\theta}\cos\theta)(\mathbf{T}\sin\theta + \mathbf{N}\cos\theta), \\ &= m(\mathbf{T}\dot{v} + \mathbf{N}v\dot{\theta}). \end{aligned} \tag{17.7}$$

From equations (17.6) and (17.7) we have

$$\begin{aligned} m\dot{v} &= -mg\sin\theta - kv^2, \\ mv\dot{\theta} &= -mg\cos\theta. \end{aligned}$$

Thus we have the system of differential equations

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{v} &= -g\sin\theta - kv^2/m, \\ \dot{\theta} &= -g\cos\theta/v. \end{aligned}$$

We can actually write this problem to be independent of t, which will make solving it simpler, since we do not know the final time of impact. If we assume that t is an smooth invertible function of x (that is, $t = t(x)$), then we obtain

$$\begin{aligned} \frac{dy}{dx} &= \frac{dy}{dt}\frac{dt}{dx}, \\ &= \frac{dy}{dt}\frac{1}{\frac{dx}{dt}}, \\ &= \frac{v\sin\theta}{v\cos\theta} = \tan\theta. \end{aligned}$$

We find $\frac{dv}{dx}$ and $\frac{d\theta}{dx}$ in a similar manner. Thus our system of differential equations becomes

$$\begin{aligned} \frac{dy}{dx} &= \tan\theta, \\ \frac{dv}{dx} &= -\frac{g\sin\theta + \mu v^2}{v\cos\theta}, \\ \frac{d\theta}{dx} &= -\frac{g}{v^2}, \end{aligned} \tag{17.8}$$

where $\mu = k/m$. We can now consider y, v, and θ to be functions of x, and x to be the independent variable.

Problem 3. Suppose we have a cannon that fires a projectile at a velocity of 45 m/s, and the projectile has a mass of about 60 kg, so that $\mu = .0003$. At what angle $\theta(0)$ should it be fired to land at a distance of 195 m? Use the secant method to find initial values for θ that give solutions to the following BVP:

$$\begin{aligned} \frac{dy}{dx} &= \tan\theta, \\ \frac{dv}{dx} &= -\frac{g\sin\theta + \mu v^2}{v\cos\theta}, \\ \frac{d\theta}{dx} &= -\frac{g}{v^2}, \\ y(0) &= y(195) = 0, \\ v(0) &= 45 \text{ m/s} \end{aligned} \tag{17.9}$$

($g = 9.8067$ m/s^2.)

There are four angles $\theta(0)$ that produce solutions for this BVP when $\mu = 0.0003$. However, only two of the angles are physically meaningful for this problem as they lie in $(0, \pi/2)$, while the others lie in $(\pi, 3\pi/2)$ and correspond to the projectile moving from right to left. Find and plot the two solutions whose angles lie in $(0, \pi/2)$. Also find the two solutions when $\mu = 0$ (no air resistance), and compare. Graphs of the solutions are given in Figure 17.5.

Keep in mind that the unknown initial condition is $\theta(0)$, not $y'(0)$. What is the appropriate residual function $h(t)$ to apply the secant method to?

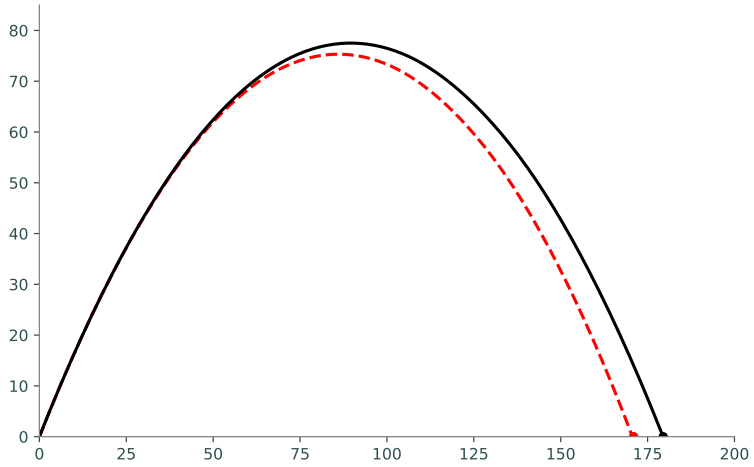


Figure 17.4: Two solutions of the system of equations (17.8), both with initial conditions $y(0) = 0$ m, $v(0) = 45$ m/s, and $\theta(0) = \pi/3$. The black curve is the trajectory of a projectile with no air resistance ($\mu = 0$). The red curve describes the trajectory of a more realistic projectile ($\mu = .0003$).

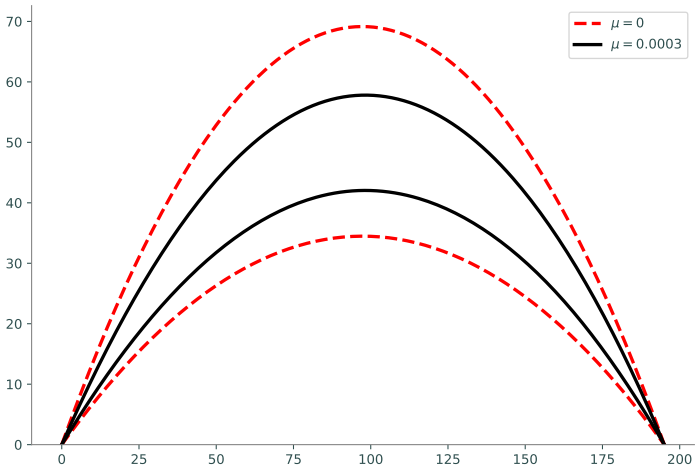


Figure 17.5: The two solutions of the boundary value problem (17.9) when the air resistance is described by the parameter $\mu = 0.0003$, and the two solutions with no air resistance ($\mu = 0$).

18 Total Variation and Image Processing

Lab Objective: *Minimizing an energy functional is equivalent to solving the resulting Euler-Lagrange equations. We introduce the method of steepest descent to solve these equations, and apply this technique to a denoising problem in image processing.*

The Gradient Descent method

Consider an energy functional $J[u]$, defined over a collection of admissible functions $u : \Omega \subset \mathbb{R}^n \to \mathbb{R}$, with the form

$$J[u] = \int_\Omega L(x, u, \nabla u)\, dx$$

where $L = L(x, u, \nabla u)$ is a function $\mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$. A standard result from the calculus of variations states that a minimizing function u^* satisfies the Euler-Lagrange equation

$$L_u - \sum_{i=1}^{n} \frac{\partial L_{u_{x_i}}}{\partial x_i} = L_u - \nabla \cdot L_{\nabla u} = L_u - \operatorname{div}(L_{\nabla u}) = 0. \tag{18.1}$$

where $L_{\nabla u} = \nabla' L = [L_{x_1}, \dots, L_{x_n}]^\intercal$.

This equation is typically an elliptic PDE, possessing boundary conditions associated with restrictions on the class of admissible functions u. To more easily compute (18.1), we consider a related parabolic PDE,

$$\begin{aligned} u_t &= -(L_u - \operatorname{div} L_{\nabla u}), \quad t > 0, \\ u(x, 0) &= u_0(x), \quad t = 0. \end{aligned} \tag{18.2}$$

A steady state solution of (18.2) does not depend on time, and thus solves the Euler-Lagrange equation. It is often easier to evolve an initial guess using (18.2), and stop whenever its steady state is well-approximated, than to solve (18.1) directly.

Consider the energy functional

$$J[u] = \int_\Omega \|\nabla u\|^2\, dx.$$

The minimizing function u^* satisfies the Euler-Lagrange equation

$$-\operatorname{div} \nabla u = -\Delta u = 0.$$

The gradient descent flow is the well-known heat equation

$$u_t = \Delta u.$$

The Euler-Lagrange equation could equivalently be described as $\Delta u = 0$, leading to the PDE $u_t = -\Delta u$. Since the backward heat equation is ill-posed, it would not be helpful in a search for the steady-state.

Let us take the time to make (18.2) more rigorous. We recall that

$$\begin{aligned}\delta J(u;h) &= \frac{d}{dt} J(u+\varepsilon h)\Big|_{\varepsilon=0}, \\ &= \int_\Omega (L_u(u) - \operatorname{div} L_{\nabla u}(u))h\,dx, \\ &= \langle L_u(u) - \operatorname{div} L_{\nabla u}(u), h\rangle_{L^2(\Omega)},\end{aligned}$$

for each u and each admissible perturbation h. Then using the Cauchy-Schwarz inequality,

$$|\delta J(u;h)| \le \|L_y(u) - \operatorname{div} L_{\nabla u}(u)\| \cdot \|h\|$$

with equality iff $h = \alpha(L_u(u) - \operatorname{div} L_{\nabla u}(u))$ for some $\alpha \in \mathbb{R}$. This implies that the "direction" $h = L_u(u) - \operatorname{div} L_{\nabla u}(u)$ is the direction of steepest ascent and maximizes $\delta J(u;h)$. Similarly,

$$h = -(L_u(u) - \operatorname{div} L_{\nabla u}(u))$$

points in the direction of steepest descent, and the flow described by (18.2) tends to move toward a state of lesser energy.

Minimizing the area of a surface of revolution

The area of the surface obtained by revolving a curve $u(x)$ about the x-axis is

$$A[u] = \int_a^b 2\pi u\sqrt{1+(u')^2}\,dx.$$

To minimize the functional A over the collection of smooth curves with fixed end points $u(a) = u_a$, $u(b) = u_b$, we use the Euler-Lagrange equation

$$\begin{aligned}0 &= 1 - u\frac{u''}{1+(u')^2}, \\ &= 1 + (u')^2 - uu'',\end{aligned} \tag{18.3}$$

with the gradient descent flow given by

$$\begin{aligned}&u_t = -1 - (u')^2 + uu'', \quad t > 0,\ x \in (a,b), \\ &u(x,0) = g(x), \quad t = 0, \\ &u(a,t) = u_a, \quad u(b,t) = u_b.\end{aligned} \tag{18.4}$$

Numerical Implementation

We will construct a numerical solution of (18.4) using the conditions $u(-1) = 1$, $u(1) = 7$. A simple solution can be found by using a second-order order discretization in space with a simple forward Euler step in time. We create the grid and set our end states below.

```
import numpy as np

a, b = -1, 1.
alpha, beta = 1., 7.
####  Define variables x_steps, final_T, time_steps  ####
delta_t, delta_x = final_T/time_steps, (b-a)/x_steps
x0 = np.linspace(a, b, x_steps+1)
```

Most numerical schemes have a stability condition that must be satisfied. Our discretization requires that $\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$. We continue by checking that this condition is satisfied, and use the straight line connecting the end points as initial data.

```
# Check a stability condition for this numerical method
if delta_t/delta_x**2. > .5:
    print("stability condition fails")

u = np.empty((2,x_steps+1))
u[0]  = (beta - alpha)/(b-a)*(x0-a) + alpha
u[1] = (beta - alpha)/(b-a)*(x0-a) + alpha
```

Finally, we define the right hand side of our difference scheme, and time step until the scheme converges.

```
def rhs(y):
    # Approximate first and second derivatives to second order accuracy.
    yp = (np.roll(y, -1) - np.roll(y, 1))/(2.*delta_x)
    ypp = (np.roll(y, -1) - 2.*y + np.roll(y, 1))/delta_x**2.
    # Find approximation for the next time step, using a first order Euler step
    y[1:-1] -= delta_t*(1. + yp[1:-1]**2. - 1.*y[1:-1]*ypp[1:-1])
    return y


# Time step until successive iterations are close
iteration = 0
while iteration < time_steps:
    u[1] = rhs(u[1])
    if norm(np.abs((u[0] - u[1]))) < 1e-5: break
    u[0] = u[1]
    iteration+=1

print("Difference in iterations is ", norm(np.abs((u[0] - u[1]))))
print("Final time = ", iteration*delta_t)
```

Problem 1. Using 20 x steps, 250 time steps, $a = -1$, $b = 1$, $\alpha = 1$, $\beta = 7$, and a final time of 0.2, plot the solution that minimizes (18.4). It should match figure 18.1.

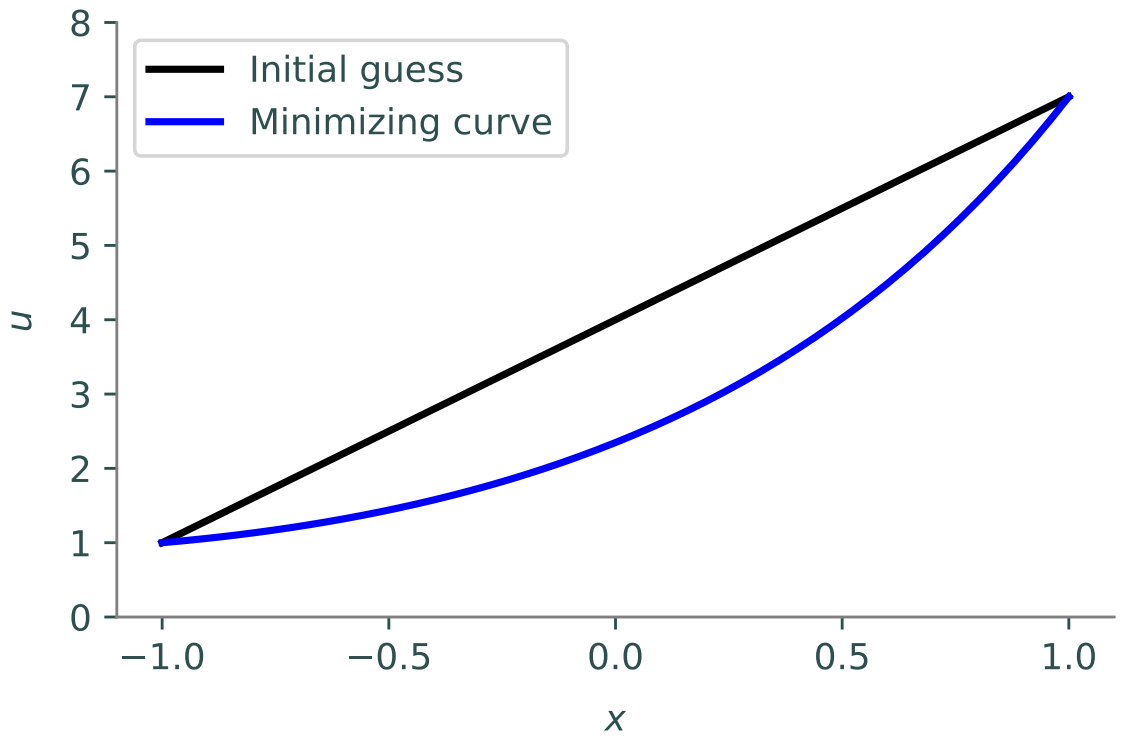


Figure 18.1: The solution of (18.3), found using the gradient descent flow (18.4).

Image Processing: Denoising

A greyscale image can be represented by a scalar-valued function $u : \Omega \to \mathbb{R}$, $\Omega \subset \mathbb{R}^2$. The following code reads an image into an array of floating point numbers, adds some noise, and saves the noisy image.

```
from numpy.random import randint, uniform, randn
import matplotlib.pyplot as plt
from matplotlib import cm
from imageio.v3 import imread, imwrite

imagename = "balloons_color.jpg"
changed_pixels=40000
# Read the image file imagename into an array of numbers, IM
# Multiply by 1. / 255 to change the values so that they are floating point
# numbers ranging from 0 to 1.
IM = imread(imagename, mode='F') * (1. / 255)
IM_x, IM_y = IM.shape

for lost in range(changed_pixels):
    x_, y_ = randint(1, IM_x-2), randint(1, IM_y-2)
    val =  .1*randn() + .5
    IM[x_, y_] = max( min(val, 1.), 0.)
```

```
imwrite("noised_"+imagename, IM)
```

A color image can be represented by three functions u_1, u_2, and u_3. In this lab we will work with black and white images, but total variation techniques can easily be used on more general images.

A simple approach to image processing

Here is a first attempt at denoising: given a noisy image f, we look for a denoised image u minimizing the energy functional

$$J[u] = \int_\Omega L(x, u, \nabla u)\, dx, \tag{18.5}$$

where

$$\begin{aligned} L(x, u, \nabla u) &= \frac{1}{2}(u - f)^2 + \frac{\lambda}{2}|\nabla u|^2, \\ &= \frac{1}{2}(u - f)^2 + \frac{\lambda}{2}(u_x^2 + u_y^2)^2. \end{aligned}$$

This energy functional penalizes 1) images that are too different from the original noisy image, and 2) images that have large derivatives. The minimizing denoised image u will balance these two different costs.

Solving for the original denoised image u is a difficult inverse problem-some information is irretrievably lost when noise is introduced. However, a priori information can be used to guess at the structure of the original image. For example, here λ represents our best guess on how much noise was added to the image, and is known as a regularization parameter in inverse problem theory.

The Euler-Lagrange equation corresponding to (18.5) is

$$\begin{aligned} L_u - \text{div } L_{\nabla u} &= (u - f) - \lambda \Delta u, \\ &= 0. \end{aligned}$$

and the gradient descent flow is

$$\begin{aligned} u_t &= -(u - f - \lambda \Delta u), \\ u(x, 0) &= f(x). \end{aligned} \tag{18.6}$$

Let u_{ij}^n represent our approximation to $u(x_i, y_j)$ at time t_n. We will approximate u_t with a forward Euler difference, and Δu with centered differences:

$$\begin{aligned} u_t &\approx \frac{u_{ij}^{n+1} - u_{ij}^n}{\Delta t}, \\ u_{xx} &\approx \frac{u_{i+1,j}^n - 2u_{ij}^n + u_{i-1,j}^n}{\Delta x^2}, \\ u_{yy} &\approx \frac{u_{i,j+1}^n - 2u_{ij}^n + u_{i,j-1}^n}{\Delta y^2}. \end{aligned}$$

Original image

Image with white noise

Figure 18.2: Noise.

Problem 2. Using $\Delta t = 1e{-}3$, $\lambda = 40$, $\Delta x = 1$, and $\Delta y = 1$, implement the numerical scheme mentioned above to obtain a solution u. (So $\Omega = [0, n_x] \times [0, n_y]$, where n_x and n_y represent the number of pixels in the x and y dimensions, respectively.) Take 250 steps in time. Plot the image with noise as well as the smoothed image. Compare your results with the first half of Figure 18.3.

Hint: Use the function `np.roll` to compute the spatial derivatives. For example, the second derivative can be approximated at interior grid points using

```
u_xx = np.roll(u, -1, axis=1) - 2*u + np.roll(u, 1, axis=1)
```

Image Processing: Total Variation Method

We represent an image by a function $u : [0, 1] \times [0, 1] \to \mathbb{R}$. A C^1 function $u : \Omega \to \mathbb{R}$ has bounded total variation on Ω ($BV(\Omega)$) if $\int_\Omega |\nabla u| < \infty$; u is said to have total variation $\int_\Omega |\nabla u|$. Intuitively, the total variation of an image u increases when noise is added.

Initial diffusion-based approach

Total variation based approach

Figure 18.3: The solutions of (18.6) and (18.11), found using a first order Euler step in time and centered differences in space.

The total variation approach was originally introduced by Ruding, Osher, and Fatemi[1]. It was formulated as follows: given a noisy image f, we look to find a denoised image u minimizing

$$\int_\Omega |\nabla u(x)|\, dx \tag{18.7}$$

subject to the constraints

$$\int_\Omega u(x)\, dx = \int_\Omega f(x)\, dx, \tag{18.8}$$

$$\int_\Omega |u(x) - f(x)|^2\, dx = \sigma|\Omega|. \tag{18.9}$$

Intuitively, (18.7) penalizes fast variations in f - this functional together with the constraint (18.8) has a constant minimum of $u = \frac{1}{|\Omega|}\int_\Omega u(x)\, dx$. This is obviously not what we want, so we add a constraint (18.9) specifying how far $u(x)$ is required to differ from the noisy image f. More precisely, (18.8) specifies that the noise in the image has zero mean, and (18.9) requires that a variable σ be chosen a priori to represent the standard deviation of the noise.

Chambolle and Lions proved that the model introduced by Rudin, Osher, and Fatemi can be formulated equivalently as

$$F[u] = \min_{u \in BV(\Omega)} \int_\Omega |\nabla u| + \frac{\lambda}{2}(u - f)^2\, dx, \tag{18.10}$$

[1]L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms", *Physica D.*, 1992.

where $\lambda > 0$ is a fixed regularization parameter[2]. Notice how this functional differs from (18.5): $\int_\Omega |\nabla u|$ instead of $\int_\Omega |\nabla u|^2$. This turns out to cause a huge difference in the result. Mathematically, there is a nice way to extend F and the class of functions with bounded total variation to functions that are discontinuous across hyperplanes. The term $\int |\nabla|$ tends to preserve edges/boundaries of objects in an image.

The gradient descent flow is given by

$$\begin{aligned} u_t &= -\lambda(u-f) + \frac{u_{xx}u_y^2 + u_{yy}u_x^2 - 2u_x u_y u_{xy}}{(u_x^2+u_y^2)^{3/2}}, \\ u(x,0) &= f(x). \end{aligned} \tag{18.11}$$

Notice the singularity that occurs in the flow when $|\nabla u| = 0$. Numerically we will replace $|\nabla u|^3$ in the denominator with $(\varepsilon + |\nabla u|^2)^{3/2}$, to remove the singularity.

Problem 3. Using $\Delta t = 1e-3, \lambda = 1, \Delta x = 1$, and $\Delta y = 1$, implement the numerical scheme mentioned above to obtain a solution u. Take 200 steps in time. Display both the diffusion-based and total variaton images of the balloon. Compare your results with Figure 18.3. How small should ε be?

Hint: To compute the spatial derivatives, consider the following:

```
u_x = (np.roll(u, -1, axis=1) -  np.roll(u, 1, axis=1))/2
u_xx = np.roll(u, -1, axis=1) - 2*u + np.roll(u, 1, axis=1)
u_xy = (np.roll(u_x, -1, axis=0) - np.roll(u_x, 1, axis=0))/2.
```

[2]A. Chambelle and P.-L. Lions, "Image recovery via total variation minimization and related problems", *Numer. Math.*, 1997.

19 Transit Time Crossing a River

Lab Objective: *This lab discusses a classical calculus of variations problem: how is a river to be crossed in the shortest possible time? We will look at a numerical solution using the pseudospectral method.*

Suppose a boat is to be rowed across a river, from a point A on one side of a river ($x = -1$), to a point B on the other side ($x = 1$). Assuming the boat moves at a constant speed 1 relative to the current, how must the boat be steered to minimize the time required to cross the river?

Let us consider a typical trajectory for the boat as it crosses the river. If T is the time required to cross the river, then the position s of the boat at time t is

$$\begin{aligned} s(t) &= (x(t), y(t)), \quad t \in [0, T], \\ s'(t) &= (x'(t), y'(t)) \\ &= (\cos\theta(x(t)), \sin\theta(x(t))) + (0, c(x(t))). \end{aligned}$$

Here $(\cos\theta(x), \sin\theta(x))$ represents the motion of the boat due to the rower, and $(0, c(x))$ is the motion of the boat due to the current.

We can relate the angle at which the boat is steered to the graph of its trajectory by noting that

$$\begin{aligned} y'(x) &= \frac{y'(t)}{x'(t)}, \\ &= \frac{\sin\theta(x) + c(x)}{\cos\theta(x)}, \\ &= c(x)\sec\theta(x) + \tan\theta(x). \end{aligned} \tag{19.1}$$

The time T required to cross the river is given by

$$\begin{aligned} T &= \int_{-1}^{1} t'(x)\,dx, \\ &= \int_{-1}^{1} \frac{1}{x'(t)}\,dx \\ &= \int_{-1}^{1} \sec\theta(x)\,dx. \end{aligned} \tag{19.2}$$

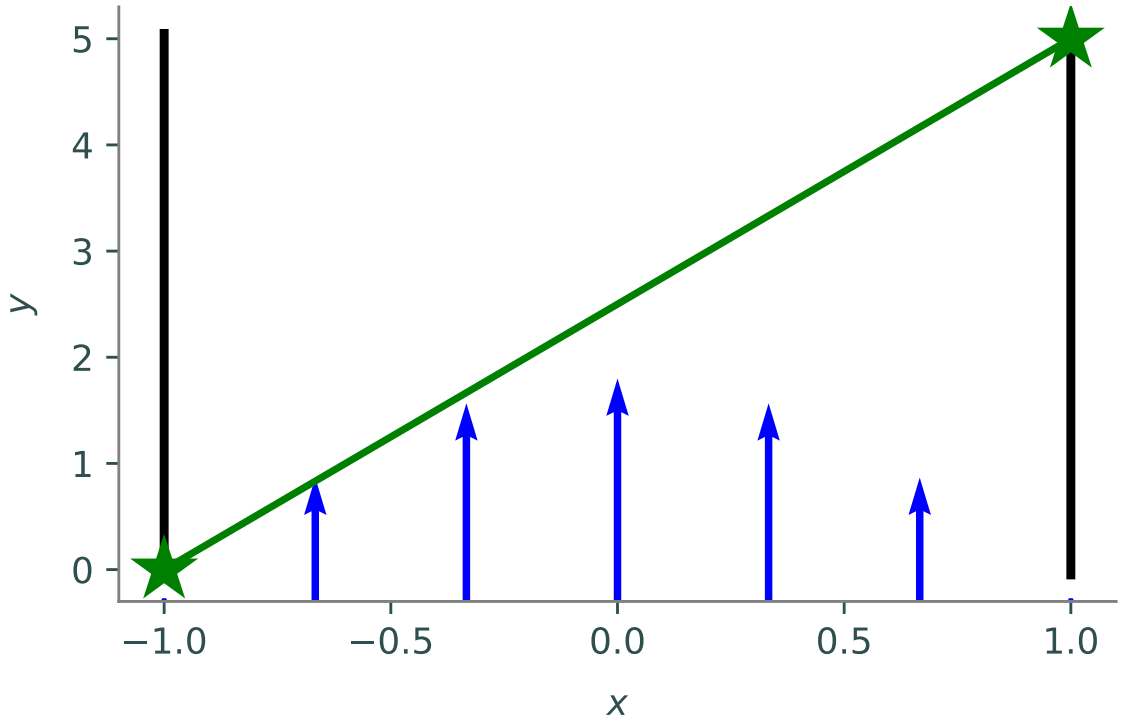


Figure 19.1: The river's current, along with a possible trajectory for the boat.

We would like to find an expression for the total time T required to cross the river from A to B, in terms of the graph of the boat's trajectory. To derive the functional $T[y]$, we note that

$$\begin{aligned} T[y] &= \int_{-1}^{1} \sec\theta(x)dx, \\ &= \int_{-1}^{1} \frac{1}{1-c(x)^2}\left(c(x)\tan\theta(x) + \sec\theta(x) - c(x)^2\sec\theta(x) - c(x)\tan\theta(x)\right)dx, \\ &= \int_{-1}^{1} \frac{1}{1-c(x)^2}\left(c(x)\tan\theta(x) + \sec\theta(x) - c(x)y'(x)\right)dx. \end{aligned}$$

Since

$$\begin{aligned} c(x)\tan\theta(x) + \sec\theta(x) &= \sqrt{1-c(x)^2 + (c(x)\sec\theta(x) + \tan\theta(x))^2}, \\ &= \sqrt{1-c(x)^2 + (y'(x))^2}, \end{aligned}$$

we obtain at last

$$T[y] = \int_{-1}^{1} \left[\alpha(x)\sqrt{1+(\alpha(x)y'(x))^2} - \alpha^2(x)c(x)y'(x)\right]dx, \tag{19.3}$$

where $\alpha(x) = (1-c^2(x))^{-1/2}$.

Problem 1. Assume that the current is given by $c(x) = -\frac{7}{10}(x^2 - 1)$. (This function assumes, for example, that the current is faster near the center of the river.) Write two python functions. The first should accept as arguments a function y, its derivative y', and an x-value, and return $L(x, y(x), y'(x))$ (where $T[y] = \int_{-1}^{1} L(x, y(x), y'(x))$). The second should use the first function to compute and return $T[y]$ for a given path $y(x)$.
Hint: The integration for $T[y]$ can be done use an approximation method such as the midpoint method or can be done using the `quad` function from `scipy.integrate`.

Problem 2. Let $y(x)$ be the straight-line path between $A = (-1, 0)$ and $B = (1, 5)$. Numerically calculate $T[y]$ to get an upper bound on the minimum time required to cross from A to B. Using (19.2), find a lower bound on the minimum time required to cross.
Hint: If $G = \int f(x)dx$ and we want to minimize G, try minimizing $f(x)$.

We look for the path $y(x)$ that minimizes the time required for the boat to cross the river, so that the function T is minimized. From the calculus of variations we know that a smooth path $y(x)$ minimizes T only if the Euler-Lagrange equation is satisfied. Recall that the Euler-Lagrange equation is

$$L_y - \frac{d}{dx} L_{y'} = 0.$$

Since $L_y = 0$, we see that the shortest time trajectory satisfies

$$\frac{d}{dx} L_{y'} = \frac{d}{dx} \left(\alpha^3(x) y'(x) (1 + (\alpha(x) y'(x))^2)^{-1/2} - \alpha^2(x) c(x) \right) = 0. \tag{19.4}$$

Problem 3. Numerically solve the Euler-Lagrange equation (19.4), using $c(x) = -\frac{7}{10}(x^2 - 1)$ and $\alpha(x) = (1 - c^2(x))^{-1/2}$, and $y(-1) = 0$, $y(1) = 5$. Plot y on $x \in [-1, 1]$.

Hint: Since this boundary value problem is defined over the domain $[-1, 1]$, it is easy to solve using the pseudospectral method. Begin by replacing each $\frac{d}{dx}$ with the pseudospectral differentiation matrix D. Then impose the boundary conditions and solve implicitly using `fsolve` from `scipy.optimize.root`. See the last two problems of Spectral 1 for a reminder on how to do this.

Problem 4. Plot the angle at which the boat should be pointed at each x-coordinate.

Hint: Use Equation (19.1); see Figure 19.3. Note that the angle the boat should be steered is *not* described by the tangent vector to the trajectory. Consider using `scipy.optimize.root` or `scipy.interpolate.barycentric_interpolate`

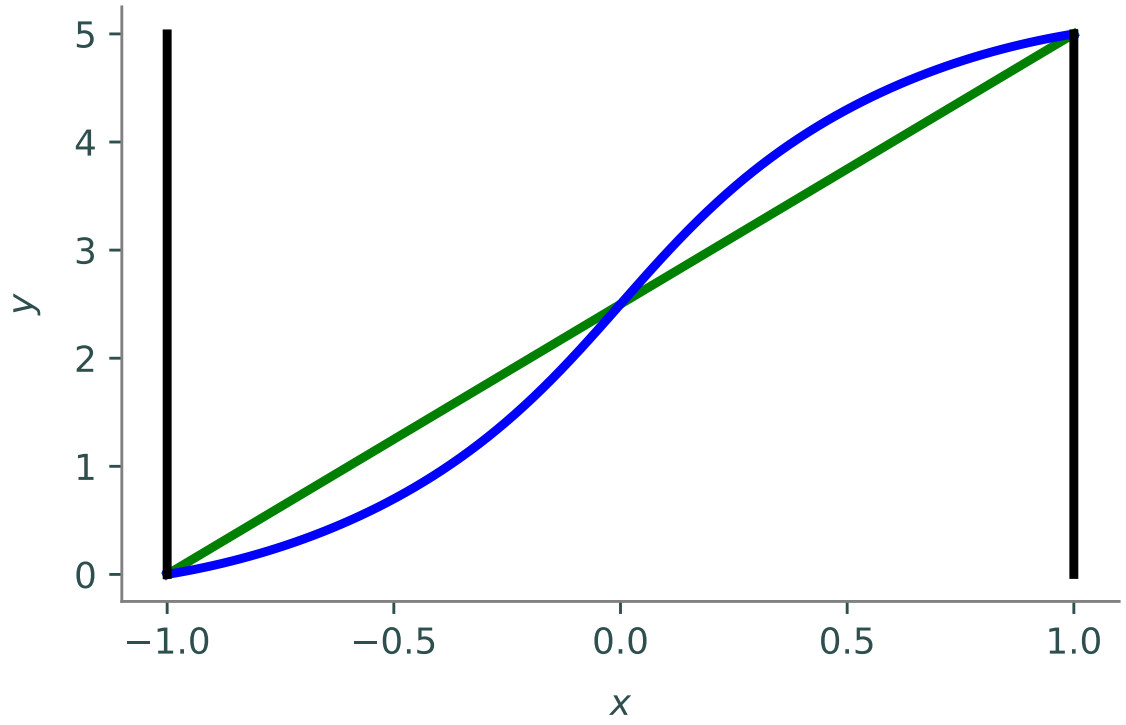


Figure 19.2: Numerical computation of the trajectory with the shortest transit time.

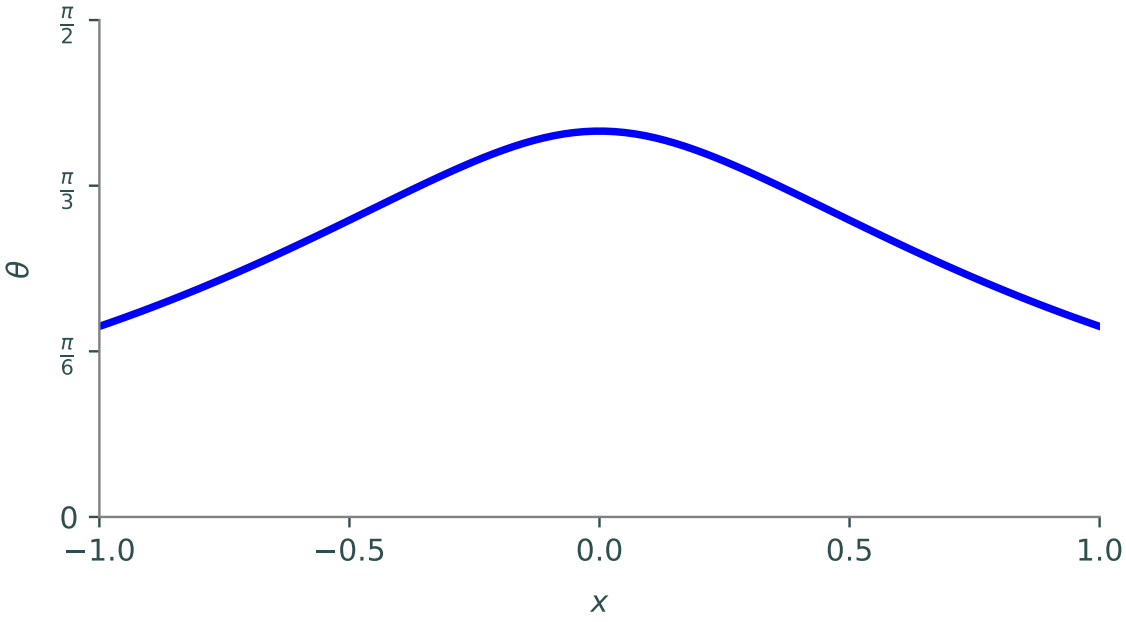


Figure 19.3: The optimal angle to steer the boat.

20 HIV Treatment Using Optimal Control

Introduction

Viruses cause many common illnesses in society today, including influenza, the common cold, and COVID-19. Viruses are obligate parasites, meaning that they must infect a host in order to replicate. After entering a host cell, viruses hijack host machinery to replicate their genome and translate their proteins. After this process, the new virus particles are assembled and lyse (break apart) the host cell to find a new host.

Mammalian immune systems are composed of two interconnected systems: the innate immune system and the adaptive immune system. While both branches of the immune system can combat viruses, the adaptive immune system is especially suited to recognize and neutralize viral infections. A major part of the adaptive immune response is helper T cells, as these cells moderate and regulate all other facets of the immune response. Helper T cells are most characterized by the presence of a receptor called CD4, which helps the cell recognize infections.

One of the most devastating viral illnesses today is acquired immunodeficiency syndrome (AIDS), caused by the human immunodeficiency virus (HIV). HIV specifically targets and replicates in helper T cells, rendering them nonfunctional and killing them. By taking out the most important regulator of the immune system, HIV makes it difficult for the body to fight infection, so sicknesses that would normally be trivial for the body to manage, such as the common cold, yeast infections, and pneumonia, become deadly.

Currently, there is no cure for HIV, and vaccines are difficult to develop. Treatments that curb the replication of HIV and help maintain healthy helper T cell population levels are available, but they are expensive and must be taken for the rest of a patient's life. Optimizing the dosage is essential to maximize the drug's effect while minimizing the cost and negative side-effects of long-term usage. In this lab, we will use optimal control to find the optimum dosage of a two-drug combination to fight HIV. In this lab we will use optimal control to find the optimal dosage of a two-drug combination[1].

Derivation of Control

We begin by defining some variables. Let

- T represent the concentration of $CD4^+$ T cells,

[1] *SHORT COURSES ON THE MATHEMATICS OF BIOLOGICAL COMPLEXITY*, Web. 15 Apr. 2015 http://www.math.utk.edu/ lenhart/smb2003.v2.html.

- V the concentration of HIV particles,
- s_1 and s_2 the production of T cells by various processes,
- B_1 and B_2 the half saturation constants (like crowd control in the blood stream and plasma),
- μ the death rate of uninfected T cells,
- k the rate of infection of T cells,
- c the death rate of the virus,
- g the input rate of some external viral source, and
- u_1 and u_2 the control variables, corresponding to the amount of drugs that introduce new T cells or kill the virus, respectively.[2]

Next we write the state system, the equations that describe the changes in T cells and viruses:[3]

$$\begin{aligned} \frac{dT(t)}{dt} &= s_1 - \frac{s_2 V(t)}{B_1 + V(t)} - \mu T(t) - kV(t)T(t) + u_1(t)T(t), \quad T(0) = T_0, \\ \frac{dV(t)}{dt} &= \frac{gV(t)}{B_2 + V(t)}\big(1 - u_2(t)\big) - cV(t)T(t), \qquad\qquad V(0) = V_0. \end{aligned} \tag{20.1}$$

The term $s_1 - \frac{s_2 V}{B_1+V}$ is the source/proliferation of unaffected T cells, μT the natural loss of T cells, kVT the loss of T cells by infection, $\frac{gV}{B_2+V}$ the viral contribution to plasma, and cVT the viral loss.

We now seek to maximize the functional

$$J(u_1, u_2) = \int_0^{t_f} [T - (A_1 u_1^2 + A_2 u_2^2)]\, dt.$$

This functional considers (1) the benefit of T cells, and (2) the systematic costs of drug treatments. The constants A_1 and A_2 represent scalars to adjust the size of terms coming from u_1^2 and u_2^2 respectively. We seek an optimal control u_1^*, u_2^* satisfying

$$J(u_1^*, u_2^*) = \max_{(u_1,u_2)\in U} J(u_1, u_2) = \min_{(u_1,u_2)\in U} -J(u_1, u_2),$$

where $U = \{(u_1, u_2) : a_i \le u_i(t) \le b_i \text{ for } t \in [0, t_f], i = 1, 2\}$.

Optimality System

The Hamiltonian is defined as:

$$H = \vec{\lambda} \cdot \vec{f} - L$$

$$H = \lambda_1 \left[s_1 - \frac{s_2 V}{B_1 + V} - \mu T - kVT + u_1 T \right] + \lambda_2 \left[\frac{g(1 - u_2)V}{B_2 + V} - cVT \right] + \left[T - (A_1 u_1^2 + A_2 u_2^2) \right].$$

[2] 'Immunotherapy of HIV-1 Infection', Kirschner, D. and Webb, G. F., Journal of Biological Systems, 6(1), 71-83 (1998)

[3] 'Optimal Control of an HIV Immunology Model', H.R Joshi

Note that the costate is represented with λ instead of p. The costate evolution equations are:

$$\begin{aligned} \lambda_1^{'} &= -\frac{\partial H}{\partial T} = -1 + \lambda_1[\mu + kV - u_1] + \lambda_2 cV, & \lambda_1(t_f) &= 0, \\ \lambda_2^{'} &= -\frac{\partial H}{\partial V} = \lambda_1 \left[\frac{B_1 s_2}{(B_1+V)^2} + kT\right] - \lambda_2 \left[\frac{B_2 g(1-u_2)}{(B_2+V)^2} - cT\right], & \lambda_2(t_f) &= 0. \end{aligned} \tag{20.2}$$

Using Pontryagin's maximum principle to find the control, we have

$$\frac{\partial H}{\partial u_1} = -2A_1 u_1(t) + \lambda_1 T(t) = 0 \qquad \frac{\partial H}{\partial u_2} = -2A_2 u_2(t) + \lambda_2 \left[\frac{-gV(t)}{B_2 + V(t)}\right] = 0$$

which gives (provided these are within the bounds of the controls)

$$\begin{aligned} u_1^*(t) &= \frac{1}{2A_1}\left[\lambda_1 T(t)\right], \\ u_2^*(t) &= \frac{-1}{2A_2}\left[\lambda_2 \frac{gV(t)}{B_2 + V(t)}\right]. \end{aligned}$$

Taking into account the bounds on the controls, we have

$$\begin{aligned} u_1^*(t) &= \min\left\{\max\{a_1, \frac{1}{2A_1}(\lambda_1 T(t))\}, b_1\right\}, \\ u_2^*(t) &= \min\left\{\max\{a_2, \frac{-\lambda_2}{2A_2}\frac{gV(t)}{B_2+V(t)}\}, b_2\right\}. \end{aligned} \tag{20.3}$$

Creating a Numerical Solver

In the preceding derivation, the states, costates, and controls all depend on each other, so we can't just solve the system once, not even numerically. Instead we must iteratively solve for the control u. In each iteration we solve our state equations and our costate equations numerically, then use those to find our new control. Lastly, we check to see if our control has converged. To solve each set of differential equations, we will use `solve_ivp`. However, our state equations depend on u, our costate equations depend on our state equations, and both depend on a lot of constants. Therefore, we will define `state_equations` and `costate_equations` to accept these as additional arguments, and we will give these additional arguments to `solve_ivp` with the `args` keyword.

We'll use the following constants throughout this lab:

```
# Constants used in equations
a_1, a_2 = 0, 0
b_1, b_2 = 0.02, 0.9
s_1, s_2 = 2, 1.5
mu = 0.002
k = 0.000025
g = 30
c = 0.007
B_1, B_2 = 14, 1
A_1, A_2 = 250000, 75

constants = a_1, a_2, b_1, b_2, s_1, s_2, mu, k, g, c, B_1, B_2, A_1, A_2
```

```
# Other constants
T0, V0 = 400, 3
n = 2000
t_f = 50
```

Problem 1. Create a function that defines the state equations as in (20.1) and returns both equations in a single array. The function should be able to be passed into `solve_ivp`.

```
def state_equations(t, y, u_interpolation, constants):
    '''
    Parameters
    ----------
    t : float
        the time
    y : ndarray (2,)
        the T cell concentration and the virus concentration at time t
    u_interpolation : CubicSpline
        the values of the control u_interpolation(t) = [u1(t), u2(t)]
    constants : a_1, a_2, b_1, b_2, s_1, s_2, mu, k, g, c, B_1, B_2, A_1, ↩
        A_2

    Returns
    -------
    y_dot : ndarray (2,)
        the derivative of the T cell concentration and the virus ↩
            concentration at time t
    '''

    a_1, a_2, b_1, b_2, s_1, s_2, mu, k, g, c, B_1, B_2, A_1, A_2 = ↩
        constants
```

You may use the following code to check that your function implements the state equations correctly:

```
u = lambda _: np.full(2, 1/2)
state = np.ones(2)

state_equations(0, state, u, constants)
# This should result in [2.397975, 7.493].
```

Problem 2. Create a function that defines the costate equations as in (20.2) and returns both equations in a single array. The function should be able to be passed into the `solve_ivp`.

```
def costate_equations(t, y, u_interpolation, state_solution, constants):
    '''
    Parameters
    ----------
    t : float
        the time
    y : ndarray (2,)
        the lambda values at time t
    u_interpolation : CubicSpline
        the values of the control u_interpolation(t) = [u1(t), u2(t)]
    state_solution : result of solve_ivp on state_equations with
        dense_output=True, i.e., state_solution.sol(t) = [T(t), V(t)]
    constants : a_1, a_2, b_1, b_2, s_1, s_2, mu, k, g, c, B_1, B_2, A_1, ↩
        A_2

    Returns
    --------------
    y_dot : ndarray (2,)
        the derivative of lambda at time t
    '''

    a_1, a_2, b_1, b_2, s_1, s_2, mu, k, g, c, B_1, B_2, A_1, A_2 = ↩
        constants
```

You may use the following code to check that your function implements the costate equations correctly:

```
u = lambda _: np.full(2, 1/2)
costate = np.ones(2)
class test_state_solution(): sol = lambda self, _: np.ones(2)

costate_equations(0, costate, u, test_state_solution(), constants)
# This should result in [-1.490975, -3.64964167].
```

Finally, we can put these together to create our solver.

Problem 3. Create and run a numerical solver for the HIV two drug model using the code below. Use (20.3) to solve for u_1^* and u_2^*.

Note that while the state equations have initial conditions, the costate equations have end conditions. Fortunately `solve_ivp` can handle this by reversing the start and end time arguments and then making sure to index the results backwards. For example, you might use `<solve_ivp_result>.y[:, ::-1]`.

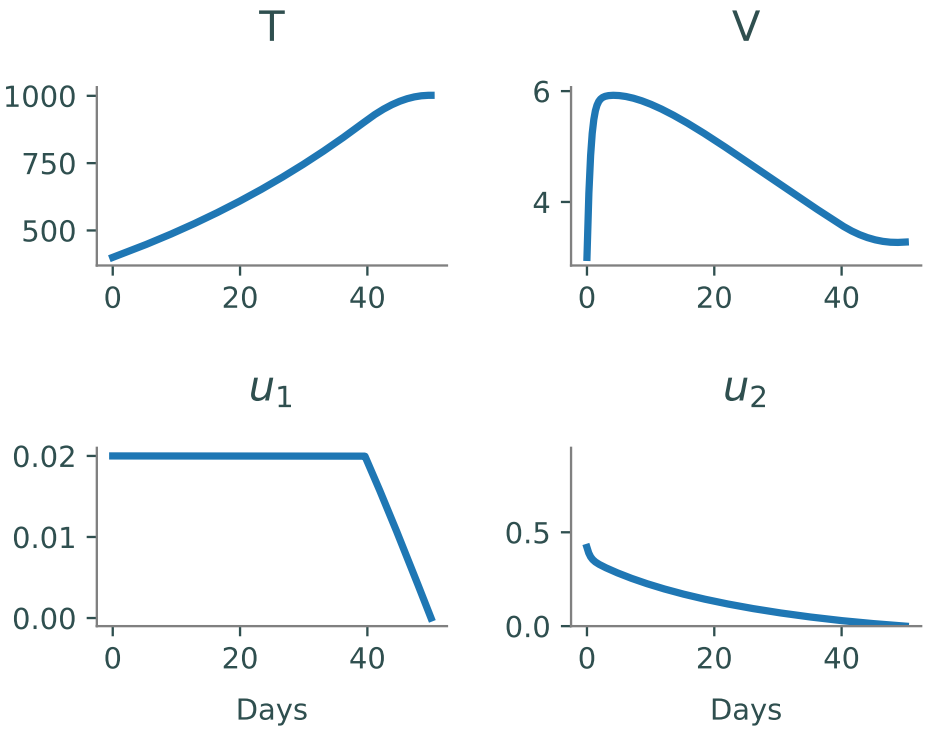


Figure 20.1: The solution to Problem 3.

When using `solve_ivp`, specify `max_step=0.5` to help with convergence, and make sure you're using the Runge–Kutta algorithm (`method="RK45"`). Also set `dense_output=True` so that you can call `<solve_ivp_result>.sol(t)` at arbitrary values of t in your state and costate equations. Finally, use `np.linspace(0, t_f, n)` for the `CubicSpline` interpolation of u and to evaluate `state_solution` and `costate_solution` when solving for the next u_1 and u_2.

```
# Initialize state, costate, and u.
state0 = np.array([T0, V0])
costate0 = np.zeros(2)

u = np.zeros((2, n))
u[0], u[1] = b_1, b_2

max_step = 0.5

epsilon = 0.001
test = epsilon + 1

tls = np.linspace(0, t_f, n)
while(test > epsilon):
    oldu = u.copy()
    # u_interpolation = CubicSpline(...)
```

```
    # Solve the state equations forward in time.
    # state_solution = solve_ivp(...)

    # Solve the costate equations backward in time.
    # costate_solution = solve_ivp(...)

    # Solve for u1 and u2.

    # Update control u with u1 and u2.

    # Test for convergence
    test = abs(oldu - u).sum()
```

Your solutions should match Figure 20.1.

Hint: To ensure the controls are within bounds, consider using `np.minimum` and `np.maximum`, or `np.clip`. Also, when generating a `CubicSpline` interpolation of the control `u`, you may need to specify an `axis` argument.

Patients usually take several different classes of drugs at a time to prevent HIV from replicating and progressing into AIDS. Reverse transcriptase inhibitors prevent the HIV genome from inserting itself into the host genome. These prevent helper T cell death by lowering the number of HIV particles in the body. Protease inhibitors prevent the activation of HIV proteins that are needed for replication. Fusion inhibitors can be taken early in the course of HIV infection and prevent the entry of HIV into helper T cells. There are many unique drugs in each class, all with known and unknown interactions and side effects. Physicians rotate through drugs to help their patients have a positive outcome and to prevent the virus from becoming resistant to any one drug.

21 Solitons

Lab Objective: *Use a pseudospectral method to study solitons, the traveling wave solutions of the Korteweg-de Vries equation.*

The Korteweg-de Vries (KdV) equation is a partial differential equation given by

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0.$$

that describes shallow water waves.

The KdV equation possesses traveling wave solutions called *solitons*. These traveling waves have the form

$$u(x,t) = 3s\,\mathrm{sech}^2\left(\frac{\sqrt{s}}{2}(x - st - a)\right),$$

where s is the speed of the wave. Solitons were first studied by John Scott Russell in 1834, in the Union Canal in Scotland. When a canal boat suddenly stopped, the water piled up in front of the boat continued moving down the canal in the shape of a pulse.

Note that there is a soliton solution for each wave speed s, and that the amplitude and speed of the soliton determine each other. Solitons are nonlinearly stable (bumped waves return to their previous shape), and they maintain their energy as they travel. Two interacting solitons will also both maintain their shapes after crossing paths.

Numerical solution

Consider the KdV equation on $[-\pi, \pi]$, together with an appropriate initial condition:

$$\begin{aligned} u_t &= -\frac{1}{2}\left(u^2\right)_x - u_{xxx}, \\ u(x,0) &= u_0(x). \end{aligned}$$

This form of the equation is slightly more convenient for the approach we will take. We will suppose the initial condition is equal at the two endpoints; that is, $u_0(-\pi) = u_0(\pi)$. This will allow us to use the pseudospectral method to find a numerical approximation for the solution $u(x,t)$.

As a reminder, the pseudospectral method involves writing the solution at each point in time using a set of basis functions, complex exponentials being the most common, and using this representation to convert the PDE into an ODE. Specifically, we can write any solution $u(x,t)$ as

$$u(x,t) = \sum_{k=-\infty}^{\infty} y_k(t)e^{ikx}.$$

Recall that k is known as the wave number. Note that all time-dependence of the solution is contained in the coefficients. We can only compute this to some finite precision, so we will choose some n and truncate the series as

$$u(x,t) = \sum_{k=-n}^{n} y_k(t)e^{ikx}.$$

The objective is to obtain an ordinary differential equation for the coefficients $y_k(t)$. We now plug it into the PDE:

$$\begin{aligned}
\frac{\partial}{\partial t}\sum_{k=-n}^{n} y_k(t)e^{ikx} &= -\frac{1}{2}\frac{\partial}{\partial x}\left(\sum_{k=-n}^{n} y_k(t)e^{ikx}\right)^2 - \frac{\partial^3}{\partial x^3}\sum_{k=-n}^{n} y_k(t)e^{ikx} \\
\sum_{k=-n}^{n} y_k'(t)e^{ikx} &= -\frac{1}{2}\frac{\partial}{\partial x}\left(\sum_{k=-n}^{n} y_k(t)e^{ikx}\right)^2 + \sum_{k=-n}^{n} ik^3 y_k(t)e^{ikx}
\end{aligned}$$

For this particular PDE, this leads to an apparent problem: the u^2 term will be difficult and computationally costly to differentiate. However, we can get around this difficulty using the fast Fourier transform.

Divide $[-\pi,\pi]$ into $2n+1$ intervals of equal width $\frac{2\pi}{2n+1}$, and let $-\pi = x_{-n}, x_{-n+1}, \dots, x_n, x_{n+1} = \pi$ be the $2n+2$ evenly-spaced gridpoints. For any function f on that interval with Fourier series $f(x) = \sum_{k=-\infty}^{\infty} a_k e^{ikx}$, we can use the discrete Fourier transform on the values $f(x_{-n}), \dots, f(x_{n+1})$ at the gridpoints to quickly get the Fourier coefficients $a_{-n}, \dots, a_n$. The inverse Fourier transform can be used to get the function values at the grid points from the Fourier coefficients. Both of these operations are very efficient, having complexity $O(n \log n)$. This sets up our strategy.

At each time t, we can use the inverse Fourier transform to compute the values of $u(x_m, t)$ for $m = -n, \dots, n+1$. Then, we apply the Fourier transform to u^2 to get its Fourier coefficients. We will denote these as w_k, so

$$u^2(x,t) = \sum_{k=-n}^{n} w_k(t)e^{ikx}.$$

Then,

$$\frac{\partial}{\partial x}u^2(x,t) = \sum_{k=-n}^{n} ikw_k(t)e^{ikx},$$

so the KdV equation can be written as

$$\sum_{k=-n}^{n} y_k'(t)e^{ikx} = \sum_{k=-n}^{n}\left(-\frac{1}{2}ikw_k(t) + ik^3 y_k(t)\right)e^{ikx}$$

Equating terms in the Fourier series yields the ordinary system of differential equations

$$y_k' = -\frac{1}{2}ikw_k + ik^3 y_k, \quad k = -n, \dots, n.$$

We can write this in a vectorized form as

$$\mathbf{y}' = -\frac{1}{2} i\mathbf{k} \odot \mathcal{F}(\mathcal{F}^{-1}(\mathbf{y})^2) + i\mathbf{k}^3 \odot \mathbf{y} \tag{21.1}$$

where $\mathcal{F}$ denotes the discrete Fourier transform. In this equation, all of the multiplication and exponentiation is componentwise.

To obtain the initial condition for the y_k, we can simply use the discrete Fourier transform again:

$$\mathbf{y}(0) = \mathcal{F}(u_0(x_{-n}), \ldots, u_0(x_{n+1}))$$

To compute the fast Fourier and inverse fast Fourier transforms numerically, we will use the `scipy.fft` module, which has functions `fft` for the fast Fourier transform and `ifft` for the inverse fast Fourier transform. These functions use an order for the coefficients that is slightly nonintuitive: the coefficients for $k \geq 0$ are all listed first, followed by the coefficients for $k < 0$. The vector of wavenumbers can be created as follows:

```
k = np.concatenate([
        np.arange(0, n+1),
        np.arange(-n-1, 0)
])
```

We are now prepared to numerically solve the KdV equation.

Problem 1. Write a function that accepts the time value t (which won't be used here, but will be useful later) the vector $\mathbf{y} = (y_0, y_1, \ldots, y_n, y_{-n-1}, \ldots, y_{-1})$ and the vector $\mathbf{k}$ of wavenumbers and returns $\mathbf{y}'$ as given in (21.1).

To numerically solve this ODE, we'll use `solve_ivp`. For this lab we want to specify a fixed time step size to use with the RK4 algorithm, but `scipy`'s implementation uses an adaptive method to control the time step. So we've defined an RK4 implementation that can be passed into `solve_ivp` with the `method` argument, and we provide the time step `dt` through the call to `solve_ivp`.

```
from scipy.integrate import solve_ivp, OdeSolver
from scipy.integrate._ivp.common import warn_extraneous

class RK4(OdeSolver):
    def __init__(self, fun, t0, y0, t_bound, dt, vectorized, **extraneous):
        super().__init__(fun, t0, y0, t_bound, vectorized, support_complex=True↩
            )

        self.dt = dt

        # t-linspace
        self.tls = np.arange(0, t_bound + dt, dt)

        self.idx = iter(range(1, len(self.tls)))

        warn_extraneous(extraneous)
```

```
    def _step_impl(self):
        self.y_old = self.y

        i = next(self.idx)
        t = self.tls[i]
        y = self.y
        f = self.fun
        dt = self.dt

        # RK4 algorithm
        K1 = f(t, y)
        K2 = f(t + dt / 2, y + dt * K1 / 2)
        K3 = f(t + dt / 2, y + dt * K2 / 2)
        K4 = f(t + dt, y + dt * K3)
        y_new = y + (dt / 6) * (K1 + 2 * K2 + 2 * K3 + K4)

        self.t = t
        self.y = y_new
        return True, None

    def _dense_output_impl(self):
        return lambda xs: np.interp(
            xs, (self.t_old, self.t), (self.y_old, self.y)
        )

# `args` is passed to the ODE function you defined in Problem 1.
# `dt` is passed to `RK4`.
sol = solve_ivp(..., args=(k,), method=RK4, dt=dt)
```

Once we have solved for the coefficients $\mathbf{y}(t)$, we need to convert them back into function values $u(x,t)$ in order to visualize the solution. This is accomplished by using the `ifft` function on the coefficient values at each time step. However, this function is designed to work with complex numbers, and returns a complex-valued array. Due to numerical error, even though our ODE solution is real-valued, there may be small imaginary components to the result; use `np.real` on the result to discard these.

Problem 2. Write a function that accepts an initial condition u_0, a final time `T`, the timestep `dt`, an integer n for the number of coefficients to use, and another integer `skip`. Numerically solve for the coefficients $\mathbf{y}(t)$ of a solution to the KdV equation.

Next, convert the Fourier coefficients back into function values at the gridpoints using the inverse Fourier transform. However, only do this for every `skip`-th timestep; we will be using far more timesteps than we want to plot. Return the gridpoints, the timesteps, and the solution $u(x,t)$.

Once we have the function values, we can plot them as a surface as follows:

```
fig = plt.figure()
```

```
ax = fig.add_subplot(1,1,1, projection="3d")

T, X = np.meshgrid(t, x, indexing="ij")
ax.plot_surface(T, X, u, cmap="coolwarm", rstride=1, cstride=1)
```

Problem 3. Numerically solve the KdV equation on $[-\pi, \pi]$ with initial conditions

$$u(x, t = 0) = 3s \operatorname{sech}^2\left(\frac{\sqrt{s}}{2}(x + a)\right),$$

where $s = 25^2$, $a = 2$. Solve on the time domain $[0, 0.0075]$, and use $n = 127$. Compare with Figure 21.1; to get a similar viewpoint, use the following:

```
ax.view_init(elev=45, azim=-45)
ax.set_zlim(0, 4000)
ax.invert_xaxis()
```

How small of a timestep did you need to use for the numerical integration to be stable? (Hint: it's smaller than 10^{-5}.) If your solution becomes full of NaN values, you are most likely using too large of a timestep.

Hint: `numpy` does not have a `sech` function; use `1/cosh(x)` to compute it instead.

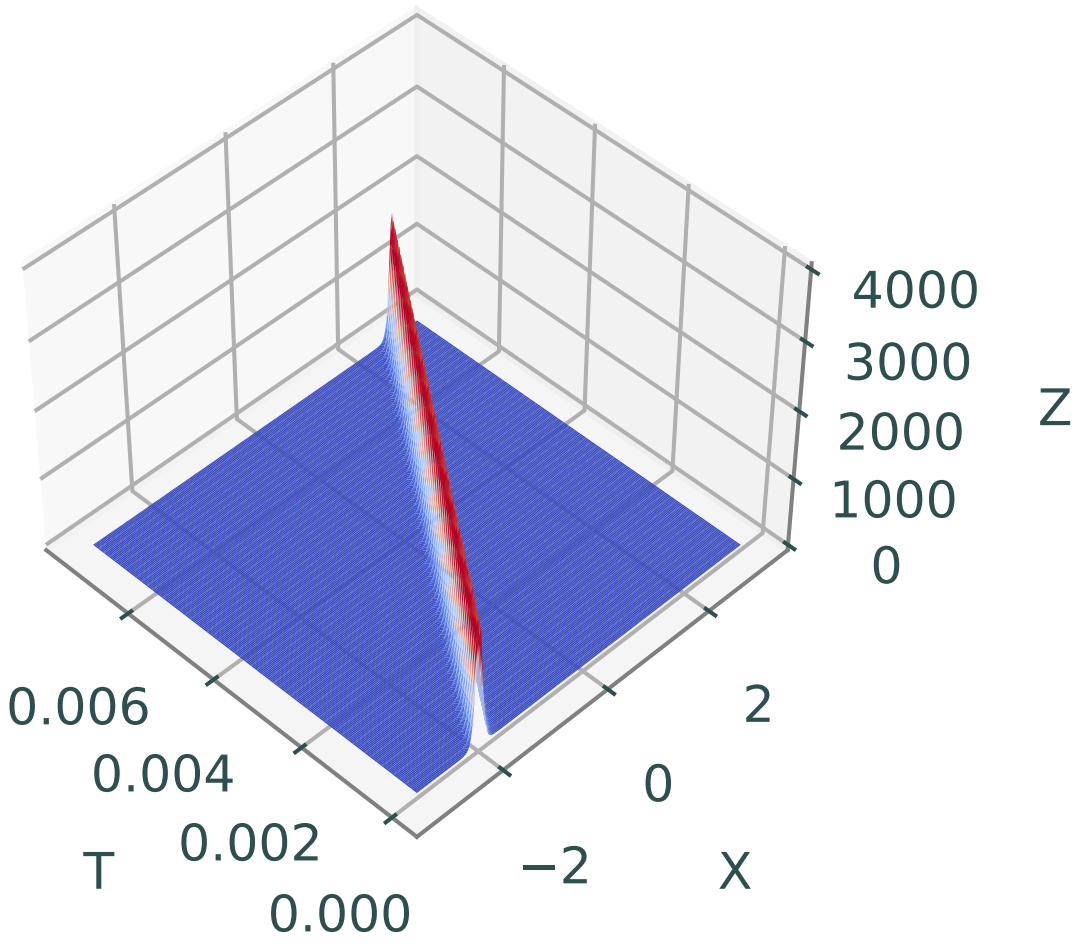


Figure 21.1: The solution to Problem 3.

Problem 4. Numerically solve the KdV equation on $[-\pi, \pi]$. This time we define the initial condition to be the superposition of two solitons:

$$u(x, t=0) = 3s_1 \operatorname{sech}^2\left(\frac{\sqrt{s_1}}{2}(x+a_1)\right) + 3s_2 \operatorname{sech}^2\left(\frac{\sqrt{s_2}}{2}(x+a_2)\right),$$

where $s_1 = 25^2$, $a_1 = 2$, and $s_2 = 16^2$, $a_2 = 1$.[a] Solve on the time domain $[0, 0.0075]$. The solution is shown in Figure 21.2.

[a] This problem is from *Spectral Methods in MATLAB*, by Trefethen.

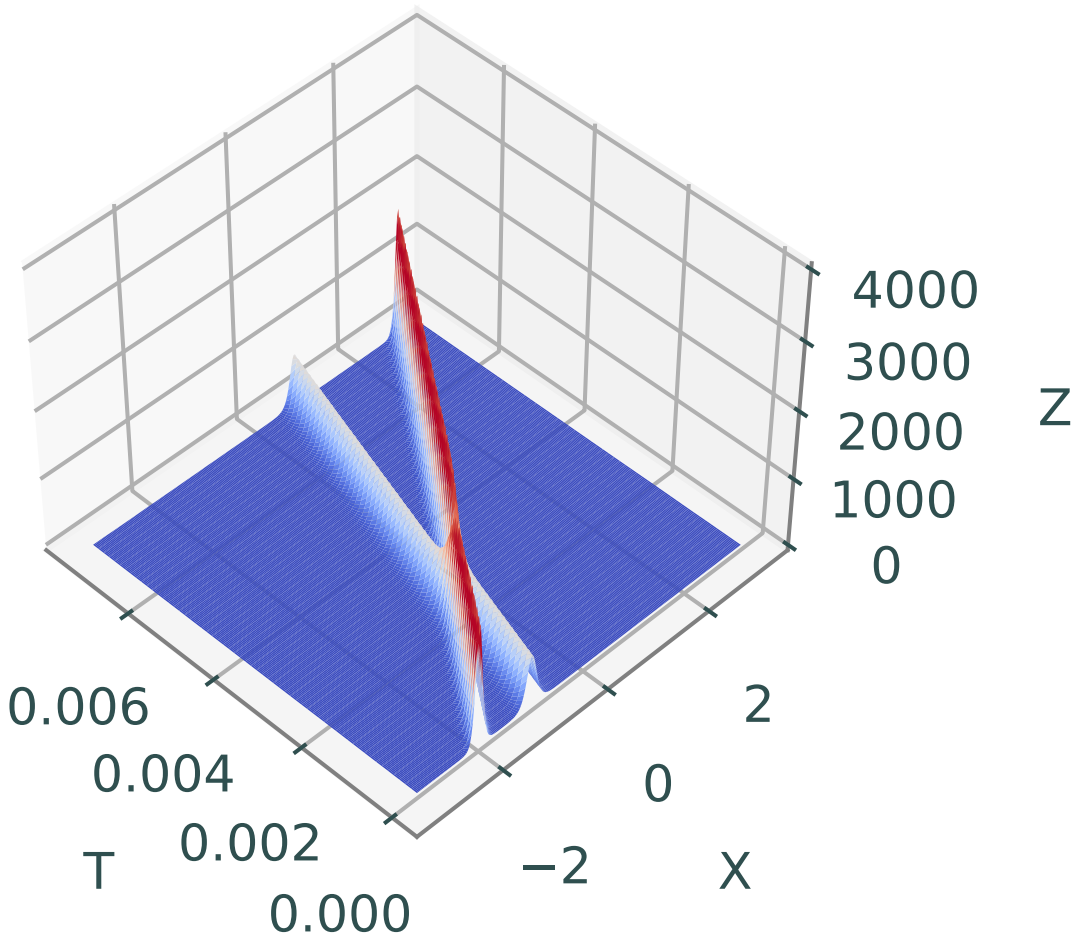


Figure 21.2: The solution to Problem 4.

Problem 5. Consider again equation (21.1). The linear term in this equation is $i\mathbf{k}^3\mathbf{y}$. This term contributes much of the exponential growth in the ODE, and contributes to how short the time step must be to ensure numerical stability. Make the substitution $z_k(t) = e^{-ik^3t}y_k(t)$ and find a similar ODE for $\mathbf{z}$. This essentially allows the exponential growth to be scaled out (it's solved for analytically, replacing it with rotation in the complex plane). Use the resulting equation to solve the previous problem. How much larger of a timestep can you use while this method remains stable?

22 Obstacle Avoidance

Lab Objective: *Solve boundary value problems that arise when using Pontryagin's Maximum principle.*

Pontryagin's Maximum Principle

Now that we understand how to solve boundary value problems, we can apply this to solve optimal control problems. Pontryagin's Maximum Principle is a very common way to formulate control problems as BVPs.

Fixed Time, Fixed Endpoint

We will begin with the more simple fixed time horizon problems. Fixed time horizon problems are commonly reformulated as boundary value problems, and we can apply what we have already learned about solving BVPs to make these problems easier to solve. We introduce fixed time horizon problems with a cost functional of the following form

$$J(u) = \int_{t_0}^{t_f} L(t, s(t), u(t))dt + K(t_f, s_f), \tag{22.1}$$

where t_0 and t_f are fixed. In this functional, $L(t, s(t), u(t))$ represents the cost of a certain path determined by the control u, and $K(t_f, s_f)$ is the terminal cost. We also have that

$$\dot{s} = f(t, s, u),\ s_0 = s(t_0),\ s_f = s(t_f). \tag{22.2}$$

In these equations t is time, s is the state variable, and u is the control variable. The maximum principle also uses the Hamiltonian equation

$$H(t, s, u, p) = \langle p, f(t, s, u)\rangle - L(t, s, u), \tag{22.3}$$

where p is a newly introduced variable called the costate. This Hamiltonian is then used to define an ODE system. This first equation defines a costate ODE system

$$\dot{p}^* = -H_s(t, s^*, u^*, p^*), \tag{22.4}$$

where a variable marked with an asterisk is the optimal choice of that variable, meaning that equation 22.4 is only true for the optimal state s^*, costate p^*, and control u^* functions. This next equation will allow us to solve for the control in terms of the state and costate

$$0 = H_u(t, s^*, u^*, p^*), \quad \forall t \in [t_0, t_f]. \tag{22.5}$$

The combination of these equations will allow us to create a BVP that will solve for the optimal control u^* and the associated states s^*. Our ODE comes from 22.2, 22.4, and 22.5, and the boundary values will come from our initial and final conditions on s.

A Specific Example

Let

$$J(u) = \int_0^{30} x^2 + \frac{2\pi}{5}u^2 dt,$$
$$\dot{s} = \begin{bmatrix} x' \\ u \end{bmatrix}, \text{ and } x'' = u.$$

Then

$$\begin{aligned} H(t, s, u, p) &= \langle p, f(t, s, u) \rangle - L(t, s, u) \\ &= \boldsymbol{p} \cdot \dot{\boldsymbol{s}} - x^2 - \frac{2\pi}{5}u^2 \\ &= \boldsymbol{p} \cdot \begin{bmatrix} x' \\ u \end{bmatrix} - x^2 - \frac{2\pi}{5}u^2 \\ &= p_1 x' + p_2 u - x^2 - \frac{2\pi}{5}u^2. \end{aligned}$$

We now need to find

$$H_{\boldsymbol{s}} = \begin{bmatrix} H_{s_1} \\ H_{s_2} \end{bmatrix} = \begin{bmatrix} H_x \\ H_{x'} \end{bmatrix} = \begin{bmatrix} -2x \\ p_1 \end{bmatrix},$$

and we see that

$$\boldsymbol{p}' = \left(\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} \right)' = -H_{\boldsymbol{s}} = \begin{bmatrix} 2x \\ -p_1 \end{bmatrix}.$$

Also, we know that $H_u = 0$ at the optimal solution, so

$$H_u = p_2 - \frac{4\pi}{5}u = 0 \Rightarrow u = \frac{5}{4\pi}p_2.$$

Thus, we have that

$$\left(\begin{bmatrix} x \\ x' \\ p_1 \\ p_2 \end{bmatrix} \right)' = \begin{bmatrix} x' \\ u \\ 2x \\ -p_1 \end{bmatrix} = \begin{bmatrix} x' \\ \frac{5}{4\pi}p_2 \\ 2x \\ -p_1 \end{bmatrix}. \tag{22.6}$$

You will now implement this in problem 1.

Problem 1. Given the following cost functional and boundary conditions, use the system of ODEs found in 22.6 to solve for and plot the optimal path (position as a function of time, $x(t)$) and acceleration (control as a function of time, $u(t) = \ddot{x}(t)$).

$$J(u) = \int_0^{30} x^2 + \frac{2\pi}{5} u^2 dt$$

$$s(t) = \begin{bmatrix} x(t) \\ x'(t) \end{bmatrix}, s(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad s(30) = \begin{bmatrix} 16 \\ 10 \end{bmatrix}$$

Plot your solutions for the optimal $x(t)$ (position) and $u(t)$ (acceleration) .

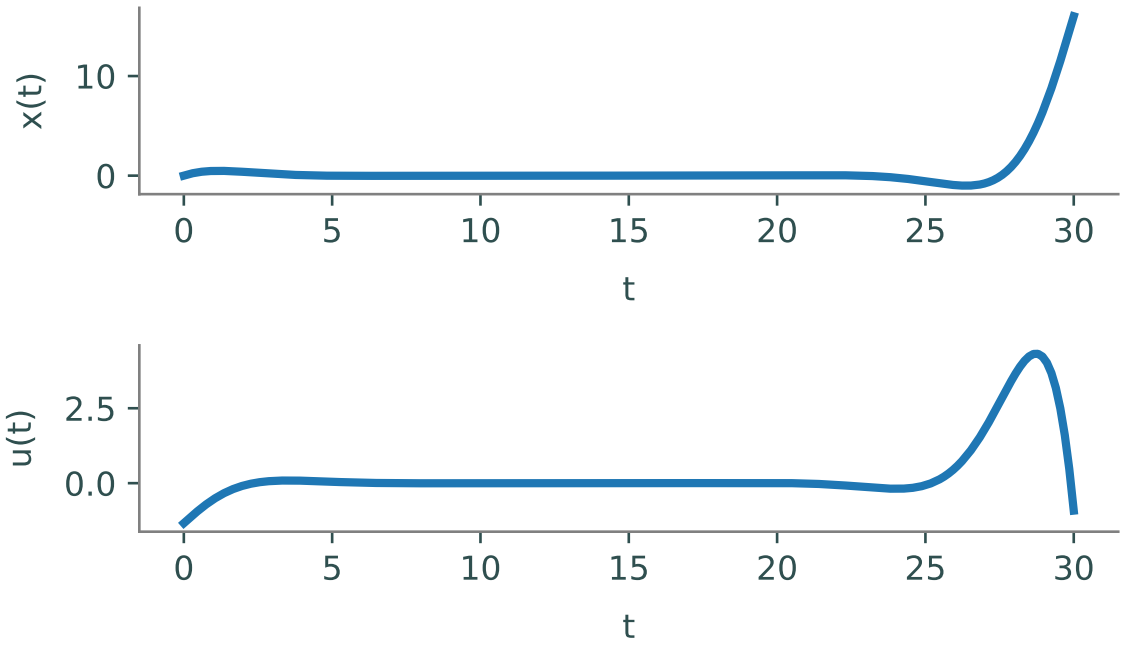


Figure 22.1: Solution to problem 1

Avoiding Collision

We now expand upon the technique learned above by adding an obstacle in our path. One area of application that relies heavily on optimal control is autonomous driving. A common problem in autonomous driving is the avoidance of obstacles. In this section we will outline a naïve solution to obstacle avoidance with a fixed time horizon.

First we can begin by defining our state variable s. We will want to understand the position and velocity at a given time so we will define the following state variable

$$s(t) = \begin{bmatrix} x(t) \\ y(t) \\ \dot{x}(t) \\ \dot{y}(t) \end{bmatrix} = \begin{bmatrix} s_1(t) \\ s_2(t) \\ s_3(t) \\ s_4(t) \end{bmatrix}, \tag{22.7}$$

which allows us to track those states in $\mathbb{R}^2$.

We can then establish the ODE defined in equation 22.2 by examining $\dot{s}(t)$

$$\dot{s}(t) = \begin{bmatrix} \dot{s_1}(t) \\ \dot{s_2}(t) \\ \dot{s_3}(t) \\ \dot{s_4}(t) \end{bmatrix} = \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \ddot{x}(t) \\ \ddot{y}(t) \end{bmatrix},$$

and if we define our control u_1 and u_2 to be acceleration in the x and y directions respectively, then we have

$$\dot{s}(t) = f(t, s, u) = \begin{bmatrix} s_3(t) \\ s_4(t) \\ u_1(t) \\ u_2(t) \end{bmatrix}. \tag{22.8}$$

Next we will define an obstacle. Since we are using integration to define cost, a reasonable way to model an obstacle in this problem would be to use a function. It would be helpful if this function is malleable, allowing us to reposition and resize it, based on the needs of the specific situation. This function also needs to have a large, preferably positive, value in a concentrated location, and it needs to vanish relatively quickly. A decent selection could be a function based on an ellipse, such as this function

$$C(x, y) = \frac{W_1}{((x - c_x)^2/r_x + (y - c_y)^2/r_y)^\lambda + 1}. \tag{22.9}$$

With the function 22.9, we can manipulate the center by changing c_x and c_y, and we can control the size by changing r_x and r_y. Changing the constant W_1 allows us to change the relative penalty of occupying the same location as the obstacle, and a reasonable value for λ will control the vanishing rate. We will also include a term in the cost functional that weights against high acceleration. This will allow us to model the real world more accurately, though the term we will be using is not a perfect representation of real world acceleration limitations. Our cost functional is the following

$$J(\boldsymbol{u}) = \int_{t_0}^{t_f} 1 + C(x(t), y(t)) + W_2 \left|\boldsymbol{u}(t)\right|^2 dt, \tag{22.10}$$

where $W_2 > 0$ defines the relative penalty of high acceleration. This functional will penalize passing near the obstacle and high levels of acceleration.

With the cost functional defined, we can now create the Hamiltonian and the rest of our BVP. We get the following Hamiltonian

$$H(t, p, s, u) = p_1 s_3 + p_2 s_4 + p_3 u_1 + p_4 u_2 - \left(1 + C(x, y) + W_2 \left(u_1(t)^2 + u_2(t)^2\right)\right), \tag{22.11}$$

which gives the following costate ODE by equation 22.4

$$\dot{p} = \begin{bmatrix} \dot{p}_1 \\ \dot{p}_2 \\ \dot{p}_3 \\ \dot{p}_4 \end{bmatrix} = \begin{bmatrix} C_x(x, y) \\ C_y(x, y) \\ -p_1 \\ -p_2 \end{bmatrix}. \tag{22.12}$$

Since we're given $H_u = 0$ in equation 22.5, then we also have the following relations

$$\begin{aligned} u_1(t) &= \frac{1}{2W_2} p_3(t) \\ u_2(t) &= \frac{1}{2W_2} p_4(t). \end{aligned} \tag{22.13}$$

Using Initial Guesses with `solve_bvp`

When solving boundary value problems in Python, we need to supply `solve_bvp` with an initial guess `y0` of the solution `y`. Note that this initial guess `y0` differs from the initial guess provided to `solve_ivp` and many other iterative solvers. Rather than providing `solve_bvp` with initial conditions, we are providing a guess for the entire solution of the ODE (i.e. at every time step). For many applications, providing an initial guess of ones is sufficient for `solve_bvp` to find the correct solution. However, for obstacle avoidance, solutions are often unstable enough that it becomes useful to provide `solve_bvp` with a better guess at the final solution. Note that our inital guess need not satisfy the boundary conditions. Also note that `solve_bvp` does not necessarily find a globally optimal solution. In a lot of cases, the solution it finds will be a local minimum, but for many practical applications (and for the purposes of this lab) local minima are sufficient.

An easy way to create an initial guess is to approximate a solution using line segments. For example, suppose we have an obstacle centered at $(4, 1)$ and we use the ODE found in 22.8 and 22.12 with an initial condition of $(6, 1.5, 0, 0)$ and a final condition of $(0, 0, 0, 0)$. It could make sense for the path to move "over" the obstacle going through a point at around $(3, 1.75)$. Thus, we could make our initial guess for x be a linspace from 6 to 0 and our initial guess for y be a line that moves from $(6, 1.5)$ to $(3, 1.75)$ and then a line that moves from $(3, 1.75)$ to $(0, 0)$. It also makes sense for us to initialize the derivatives of x as negative values since we will be moving from right to left. The following code reflects these assumptions in the creation of the initial guess:

```
# Make initial guess
y0 = np.ones((8, t_steps))
x = np.linspace(0, 6, t_steps)
y1 = (1.75 / 3) * x[:int(t_steps/2)]
y2 = (1.5 - 1.75) / 3 * (x[int(t_steps/2):] - 3) + 1.75
y_init = np.concatenate((y1, y2))
y0[0,:] = x[::-1]
y0[1,:] = y_init[::-1]
y0[2,:] = -1*np.ones(t_steps)
```

Problem 2. Using the ODEs found in 22.8 and 22.12, the obstacle function 22.9, and the following boundary conditions and parameters solve for and plot the optimal path.

$$
\begin{aligned}
t_0 = 0, \quad t_f = 20 \\
(c_x, c_y) = (4, 1) \\
(r_x, r_y) = (5, .5) \\
\lambda = 20
\end{aligned}
$$

$$
s_0 = \begin{bmatrix} 6 \\ 1.5 \\ 0 \\ 0 \end{bmatrix}, \quad s_f = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
$$

You will need to choose a W_1 and W_2 which allow the solver to find a valid path. If these parameters are not chosen correctly, the solver may find a path which goes through the obstacle, not around it. Plot the obstacle using `plt.contour()` to be certain path doesn't pass through the obstacle. Also plot the initial guess for x and y provided to `solve_bvp`.

Hint: The default for a parameter of `solve_bvp()` called `max_nodes` is not large enough. Try at least `max_nodes = 30000`. You may also find it helpful to use the function `partial` from the module `functools` to preset the parameters for the functions you will be using. If you are struggling to find viable values for W_1 and W_2, try $W_1 \in (1, 40)$ and $W_2 \in (0, 9)$.

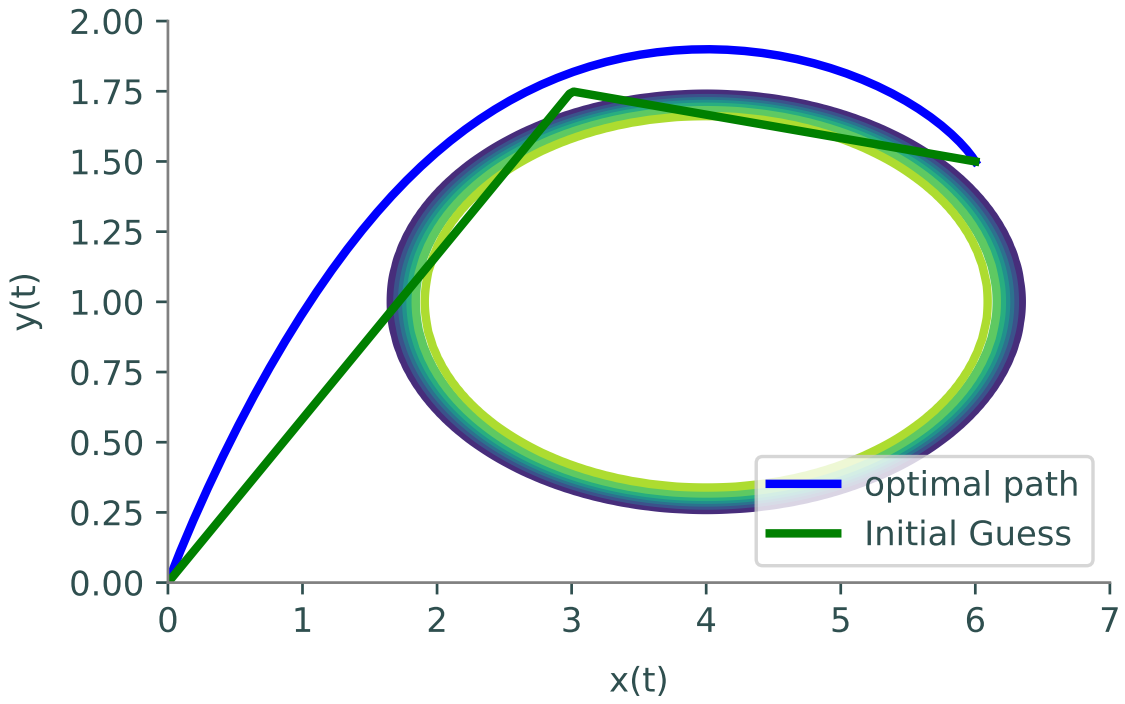


Figure 22.2: Solution to problem 2 for certain choice of parameters. Here we used $W_1 = 3$ and $W_2 = 70$, but those parameters are a choice. Other choices work too, but will result in a different optimal path around the obstacle.

Free Time Horizon Problems

In the previous sections and problems, we were working with BVPs that had a fixed start time t_0, and a fixed end time t_f. However, we may also encounter systems that have a free end time. In order to solve these problems we will need to make some alterations to the problem. First we will perform a change of basis so that we can work with a fixed end time. Consider the following system

$$\dot{x}(t) = f(x(t), t) \quad t \in [0, t_f],$$

we can do the following change of basis for the time variable

$$\begin{aligned} t &= t_f \hat{t} \\ \implies \frac{d}{dt} &= \frac{d}{d\hat{t}} \frac{d\hat{t}}{dt} \\ \implies \frac{d}{dt} &= \frac{d}{d\hat{t}} \frac{1}{t_f}. \end{aligned}$$

We can now define $z(\hat{t}) := x(t_f \hat{t})$ which gives us the following new system

$$\dot{z}\left(\hat{t}\right) = t_f f\left(z\left(\hat{t}\right), \hat{t}\right) \quad \hat{t} \in [0, 1].$$

This system can be solved in the same way we solve the fixed time horizon problems. But you may notice that we now have an extra unknown parameter, the final time. Because of this, a free time horizon problem will need one more boundary value to make the system solvable.

Lets now examine an example of a free time horizon problem. We start with a first-order ODE system. Note that the fixed final time has been replaced with a free final time, and that a needed third boundary condition has been included

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ \cos(t) - 9y_1 \end{bmatrix}, \quad y_1(0) = 5/3, \quad y_2(0) = 5, \quad y_1(t_f) = -\frac{5}{3}.$$

Now we make the coordinate change, giving the following system

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}' = t_f \begin{bmatrix} z_2 \\ \cos(t_f \hat{t}) - 9z_1 \end{bmatrix}, \quad z_1(0) = 5/3, \quad z_2(0) = 5, \quad z_1(1) = -\frac{5}{3}. \tag{22.14}$$

Now we can solve this system using `solve_bvp` in python. The new argument `p` that we have included in `ode()` and `bc()` is an `ndarray` that contains our parameter t_f.

```
def ode(t,y,p):
    ''' define the ode system '''
    return p[0]*np.array([y[1], np.cos(p[0]*t) - 9*y[0]])

def bc(ya,yb,p):
    ''' define the boundary conditions '''
    return np.array([ya[0] - (5/3), ya[1] - 5, yb[0] + 5/3])

# give the time domain
t_steps = 100
t = np.linspace(0, 1, t_steps)

# give an initial guess
y0 = np.ones((2, t_steps))
p0 = np.array([6])

# solve the system
sol = solve_bvp(ode, bc, t, y0, p0)
```

The attribute `sol.p[0]` will give the final time the solver found.

When plotting we need to make sure that we remember that $x(t_f \hat{t}) = z(\hat{t})$, so we plot in the following way

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

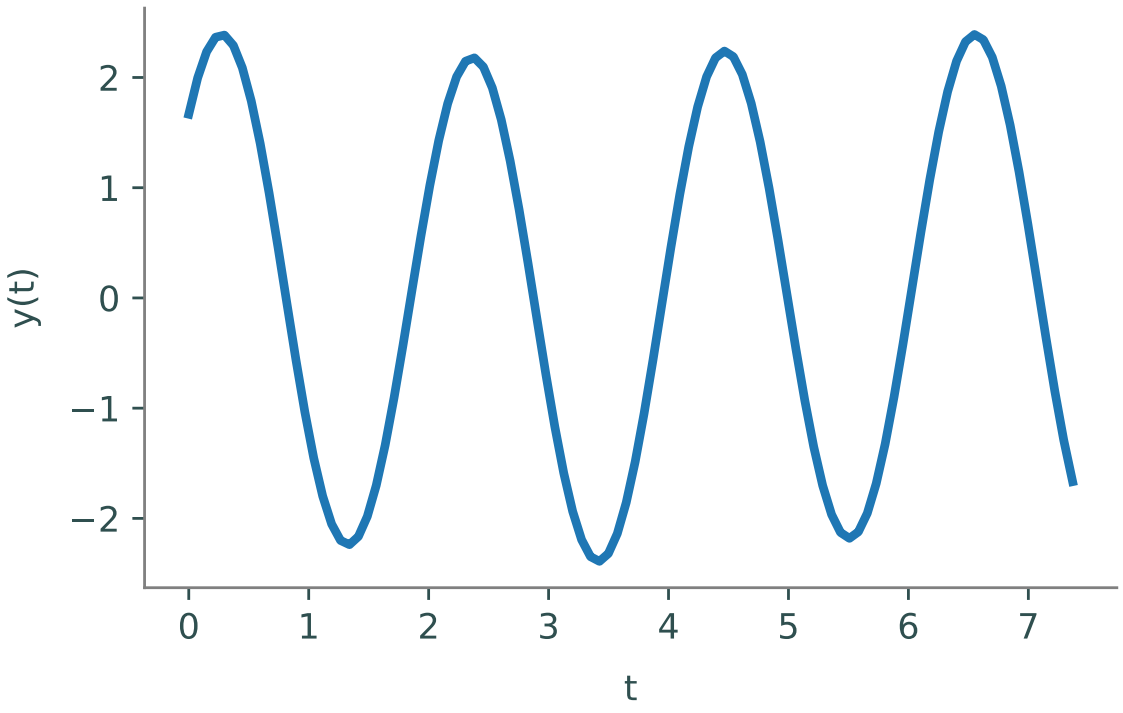


Figure 22.3: The solution to 22.14

Problem 3. Solve the following boundary value problem, using 5π as the initial guess for `p[0]`:

$$y'' + 3y = \sin(t)$$
$$y(0) = 0, \quad y(t_f) = \frac{\pi}{2}, \quad y'(t_f) = \frac{1}{2}\left(\sqrt{3}\pi\cot(\pi\sqrt{75}) - 1\right).$$

Plot your solution. What t_f did the solver find?

Hint: Be careful with $\sin(t)$. We have made the substitution for t (it is now scaled by `p[0]`).

Free Time, Fixed Endpoint Control Problems

Now that we understand how to formulate free time horizon problems, we can modify our optimal control BVP to become a free time horizon problem. This is actually the best way to formulate many optimal control problems, as we usually don't know exactly how long it takes to traverse the optimal path. The methodology is exactly the same as we used in the last problem, we only need to find the extra boundary value which will allow us to make the end time a free variable.

To find this extra boundary value, we will use the fact that the Hamiltonian is 0 for all t along the optimal path. It is standard to use the final time as the representative so we will assert that

$$H(t_f, p(t_f), s(t_f), u(t_f)) = 0. \tag{22.15}$$

You may notice that when you solve an optimal control problem as a free end time BVP, the optimal path you get is different than what you found when it was a fixed end time BVP. This is because the solution found in the fixed end time formulation is the optimal path for a certain end time, but it may not be the optimal path when time is allowed to vary. The cost functional will control how time constraints are balanced with other costs and thus determine the optimal path.

Problem 4. Refactor your code from problem 2 to create a free end time BVP and use a new boundary value derived from 22.15. Let $W_1 = 4$ and $W_2 = 0.1$, and use `max_nodes = 60000`. Plot the solution you found along with the initial guess for x and y and print the optimal time.

Hint: You may find that the initial guess provided for Problem 2 becomes more unstable when used with this problem. Try using an initial guess for `p[0]` that is close to 3 or change the initial guess `y0` so that the path runs underneath the obstacle. One such initial guess is shown in Figure 22.4. It does not matter which path your solution takes, it need only look reasonable and have a reasonable optimal time.

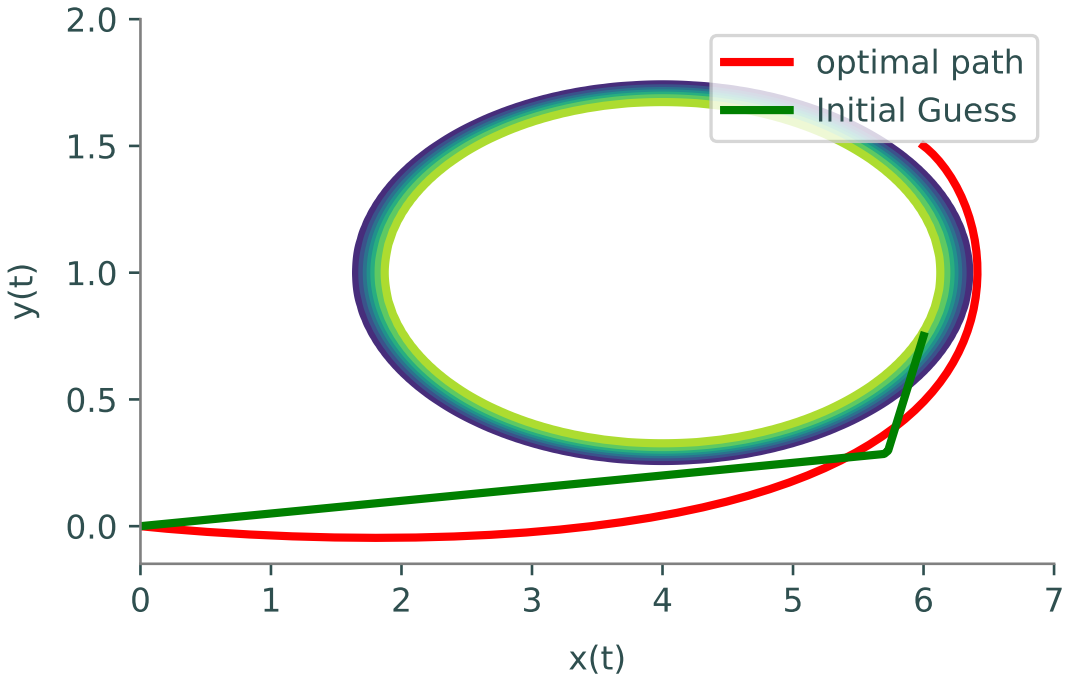


Figure 22.4: The solution to Problem 4. Here we used $W_1 = 4$ and $W_2 = 0.1$ and got an optimal time of about 4.47. Those parameters are a choice. Other choices work too, but will result in a different optimal path around the obstacle and in a different optimal time.

23 The Inverted Pendulum

Lab Objective: *We will set up the LQR optimal control problem for the inverted pendulum and compute the solution numerically.*

Think back to your childhood days when, for entertainment purposes, you'd balance objects: a book on your head, a spoon on your nose, or even a broom on your hand. Learning how to walk was likely your initial introduction to the inverted pendulum problem.

A pendulum has two rest points: a stable rest point directly underneath the pivot point of the pendulum, and an unstable rest point directly above. The generic pendulum problem is to simply describe the dynamics of the object on the pendulum (called the 'bob'). The inverted pendulum problem seeks to guide the bob toward the unstable fixed point at the top of the pendulum. Since the fixed point is unstable, the bob must be balanced relentlessly to keep it upright.

The inverted pendulum is an important classical problem in dynamics and control theory, and is often used to test different control strategies. One application of the inverted pendulum is the guidance of rockets and missiles. Aerodynamic instability occurs because the center of mass of the rocket is not the same as the center of drag. Small gusts of wind or variations in thrust require constant attention to the orientation of the rocket.

The Simple Pendulum

We begin by studying the simple pendulum setting. Suppose we have a pendulum consisting of a bob with mass m rotating about a pivot point at the end of a (massless) rod of length l. Let $\theta(t)$ represent the angular displacement of the bob from its stable equilibrium. By Hamilton's Principle, the path θ that is taken by the bob minimizes the functional

$$J[\theta] = \int_{t_0}^{t_1} L, \tag{23.1}$$

where the Lagrangian $L = T - U$ is the difference between the kinetic and potential energies of the bob.

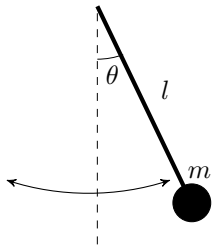


Figure 23.1: The frame of reference for the simple pendulum problem.

The kinetic energy of the bob is given by $mv^2/2$, where v is the velocity of the bob. In terms of θ, the kinetic energy becomes

$$\begin{aligned} T &= \frac{m}{2}v^2 = \frac{m}{2}(\dot{x}^2 + \dot{y}^2), \\ &= \frac{m}{2}((l\cos(\theta)\dot{\theta})^2 + (l\sin(\theta)\dot{\theta})^2), \\ &= \frac{ml^2\dot{\theta}^2}{2}. \end{aligned} \tag{23.2}$$

The potential energy of the bob is $U = mg(l - l\cos\theta)$. From these expressions we can form the Euler-Lagrange equation, which determines the path of the bob:

$$\begin{aligned} 0 &= L_\theta - \frac{d}{dx}L_{\dot{\theta}}, \\ &= -mgl\sin\theta - ml^2\ddot{\theta}, \\ &= \ddot{\theta} + \frac{g}{l}\sin\theta. \end{aligned} \tag{23.3}$$

Since in this setting the energy of the pendulum is conserved, the equilibrium position $\theta = 0$ is only Lyapunov stable. When forces such as friction and air drag are considered $\theta = 0$ becomes an asymptotically stable equilibrium.

The Inverted Pendulum

The Control System

We consider a gift suspended above a rickshaw by a (massless) rod of length l. The rickshaw and its suspended gift will have masses M and m respectively, $M > m$. Let θ represent the angle between the gift and its unstable equilibrium, with clockwise orientation. Let v_1 and v_2 represent the velocities of the rickshaw and the gift, and F the force exerted on the rickshaw. The rickshaw will be restricted to traveling along a straight line (the x-axis).

By Hamilton's Principle, the path (x, θ) of the rickshaw and the present minimizes the functional

$$J[x, \theta] = \int_{t_0}^{t_1} L, \tag{23.4}$$

where the Lagrangian $L = T - U$ is the difference between the kinetic energy of the present on the pendulum, and its potential energy.

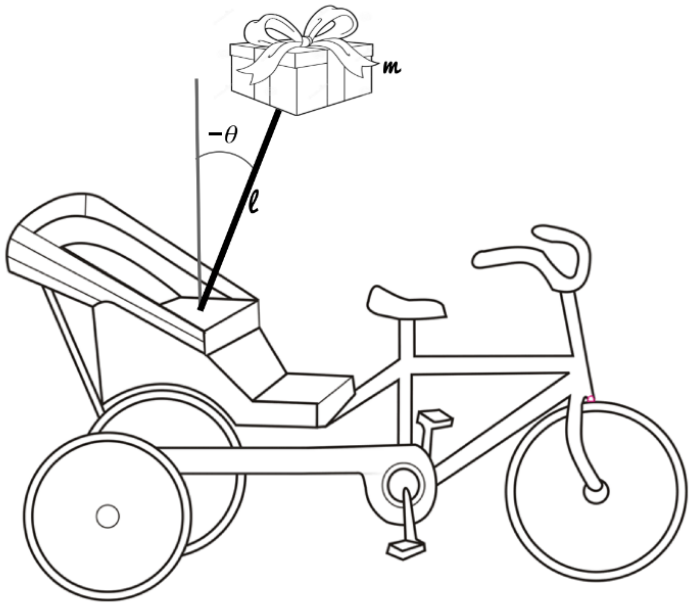


Figure 23.2: The inverted pendulum problem on a mobile rickshaw with a present suspended above.

Since the position of the rickshaw and the present are $(x(t), 0)$ and $(x - l\sin\theta, l\cos\theta)$ respectively, the total kinetic energy is

$$\begin{aligned} T &= \frac{1}{2}Mv_1^2 + \frac{1}{2}mv_2^2 \\ &= \frac{1}{2}M\dot{x}^2 + \frac{1}{2}m\left((\dot{x} - l\dot{\theta}\cos\theta)^2 + (-l\dot{\theta}\sin\theta)^2\right) \\ &= \frac{1}{2}(M+m)\dot{x}^2 + \frac{1}{2}ml^2\dot{\theta}^2 - ml\dot{x}\dot{\theta}\cos\theta. \end{aligned} \tag{23.5}$$

where v_1 is the norm of the velocity vector of the rickshaw and v_2 is that of the present.

The total potential energy is

$$U = mgl\cos\theta.$$

The path (x, θ) of the rickshaw and the present satisfy the Euler-Lagrange differential equations, but the problem involves a nonconservative force F acting in the x direction. By way of D'Alambert's Principle, our normal Euler-Lagrange equations now include the nonconservative force F on the right side of the equation:

$$\begin{aligned} \frac{\partial L}{\partial x} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}} &= F, \\ \frac{\partial L}{\partial \theta} - \frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} &= 0. \end{aligned} \tag{23.6}$$

After expanding (23.6) we see that $x(t)$ and $\theta(t)$ satisfy

$$\begin{aligned} F &= ml\ddot{\theta}\cos\theta - (M+m)\ddot{x} - ml\dot{\theta}^2\sin\theta, \\ l\ddot{\theta} &= g\sin\theta + \ddot{x}\cos\theta. \end{aligned} \tag{23.7}$$

At this point we make a further simplifying assumption. If θ starts close to 0, we may assume that the corresponding force F will keep θ small. In this case, we linearize[1] (23.7) about $(\theta, \dot{\theta}) = (0, 0)$, obtaining the equations

$$\begin{aligned} F &= ml\ddot{\theta} - (M+m)\ddot{x}, \\ l\ddot{\theta} &= g\theta + \ddot{x}. \end{aligned}$$

These equations can be further manipulated to obtain

$$\begin{aligned} \ddot{x} &= -\frac{1}{M}F + \frac{m}{M}g\theta, \\ \ddot{\theta} &= -\frac{1}{Ml}F + \frac{g}{Ml}(M+m)\theta. \end{aligned} \tag{23.8}$$

We will now write (23.8) as a first order system. Making the assignments $x_1 = x$, $x_2 = x_1'$, $\theta_1 = \theta$, $\theta_2 = \theta_1'$, letting $u = -F$ represent the control variable, we obtain

$$\begin{bmatrix} x_1 \\ x_2 \\ \theta_1 \\ \theta_2 \end{bmatrix}' = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g}{Ml}(M+m) & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \theta_1 \\ \theta_2 \end{bmatrix} + u \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ \frac{1}{Ml} \end{bmatrix},$$

which can be written more concisely as

$$z' = Az + Bu.$$

The Infinite Time Horizon LQR Problem

We consider the cost function

$$\begin{aligned} J[z] &= \int_0^\infty (q_1 x_1^2 + q_2 x_2^2 + q_3 \theta_1^2 + q_4 \theta_2^2 + r u^2)\, dt \\ &= \int_0^\infty z^\mathsf{T} Q z + u^\mathsf{T} R u \, dt \end{aligned} \tag{23.9}$$

where q_1, q_2, q_3, q_4, and r are nonnegative weights, and

$$Q = \begin{bmatrix} q_1 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 \\ 0 & 0 & q_3 & 0 \\ 0 & 0 & 0 & q_4 \end{bmatrix}, \quad R = \begin{bmatrix} r \end{bmatrix}.$$

Problem 1. Write a function that returns the matrices A, B, Q, and R given above. Let $g = 9.8$ m/s^2.

```
def linearized_init(M, m, l, q1, q2, q3, q4, r):
    '''
    Parameters:
    ----------
    M, m: floats
          masses of the rickshaw and the present
```

[1] See Additional Material section for derivation.

```
    l   : float
          length of the rod
    q1, q2, q3, q4, r : floats
         relative weights of the position and velocity of the rickshaw, ↩
            the
         angular displacement theta and the change in theta, and the ↩
            control

    Return
    -------
    A : ndarray of shape (4, 4)
    B : ndarray of shape (4, 1)
    Q : ndarray of shape (4, 4)
    R : ndarray of shape (1, 1)
    '''
    pass
```

The optimal control problem (23.9) is an example of a Linear Quadratic Regulator (LQR), and is known to have an optimal control $\tilde{u}$ described by a linear state feedback law:

$$\tilde{u} = -R^{-1}B^{\mathsf{T}}P\tilde{z}.$$

Here P is a matrix function that satisfies the Riccati differential equation (RDE)

$$\dot{P}(t) = PA + A^{\mathsf{T}}P + Q - PBR^{-1}B^{\mathsf{T}}P.$$

Since this problem has an infinite time horizon, we have $\dot{P} = 0$. Thus P is a constant matrix, and can be found by solving the algebraic Riccati equation (ARE)

$$PA + A^{\mathsf{T}}P + Q - PBR^{-1}B^{\mathsf{T}}P = 0. \tag{23.10}$$

The evolution of the optimal state vector $\tilde{z}$ can then be described by [2]

$$\dot{\tilde{z}} = (A - BR^{-1}B^{\mathsf{T}}P)\tilde{z}. \tag{23.11}$$

Problem 2. Write the following function to find the matrix P. Use `scipy.optimize.root`. Since `root` takes in a vector and not a matrix, you will have to reshape the matrix P before passing it in and after getting your result, using `P.reshape(16)` and `P.reshape((4,4))`.

```
def find_P(A, B, Q, R):
    '''
    Parameters:
    ----------
    A, Q    : ndarrays of shape (4, 4)
    B       : ndarray of shape (4, 1)
    R       : ndarray of shape (1, 1)
```

[2] See Calculus of Variations and Optimal Control Theory, Daniel Liberzon, Ch.6

```
    Returns
    -------
    P       : the matrix solution of the Riccati equation
    '''
    pass
```

Using the values

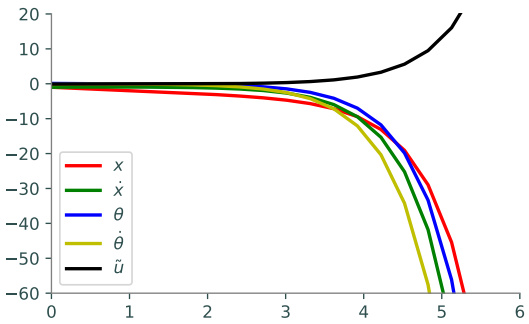
```
M, m = 23., 5.
l = 4.
q1, q2, q3, q4 = 1., 1., 1., 1.
r = 10.
```

compute the eigenvalues of $A - BR^{-1}B^{\mathsf{T}}P$. Are any of the eigenvalues positive? Consider differential equation (23.11) governing the optimal state $\tilde{z}$. Using this value of P, will we necessarily have $\tilde{z} \to 0$?

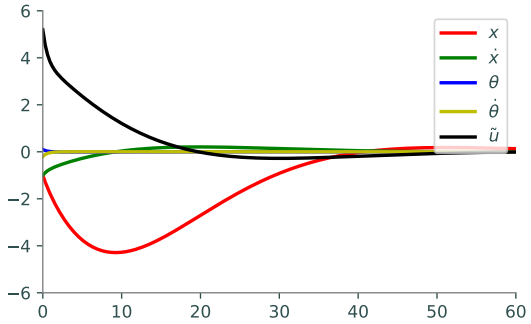
Problem 3. Write the following function that implements the LQR solution described earlier. Use `scipy.integrate.solve_ivp` to solve the IVP.

```
def rickshaw(tv, X0, A, B, Q, R, P):
    '''
    Parameters:
    ----------
    tv  : tuple containing the start and end times (t0, tf) that can be ↩
        passed into solve_ivp
    X0  : Initial conditions on state variables
    A, Q: ndarrays of shape (4, 4)
    B   : ndarray of shape (4, 1)
    R   : ndarray of shape (1, 1)
    P   : ndarray of shape (4, 4)

    Returns
    -------
    Z : ndarray of shape (n+1, 4), the state vector at each time
    U : ndarray of shape (n+1,), the control values
    '''
    pass
```

Notice that we have no information on how many solutions (23.10) possesses. In general there may be many solutions. We hope to find a unique solution P that is *stabilizing*: the eigenvalues of $A - BR^{-1}B^{\mathsf{T}}P$ have negative real part. To find this P, use the function `solve_continuous_are` from `scipy.linalg`. This function is designed to solve the continuous algebraic Riccati equation.



P is found using `scipy.optimize.root`.



P is found using `solve_continuous_are`.

Figure 23.3: The solutions of Problem 4.

Problem 4. Test the function made in Problem (3) with the following inputs:

```
M, m = 23., 5.
l = 4.
q1, q2, q3, q4 = 1., 1., 1., 1.
r = 10.
tf = None
X0 = np.array([-1, -1, .1, -.2])
```

Find the matrix P using the `scipy.optimize.root` method with `tf=6` as well as the `solve_continuous_are` method with `tf=60`. Plot the solutions $\tilde{z}$ and $\tilde{u}$. Your results should show behavior similar to that in Figure 23.3. Be sure to include a legend.

The LQR solution we have found only works for the linearized version of the ODE system that we found. What if we were to apply the control found in the LQR formulation to the original, nonlinear ODE found in (23.7)? To do this, we need to first be able to interpolate the control variable u found in Problem 4 using SciPy's `CubicSpline` function (see the documentation for more details). We also need to solve (23.7) for $\ddot{x}$ and $\ddot{\theta}$. Note that $-F$ in (23.7) is the control variable u in (23.12).

$$\begin{aligned} \ddot{x} &= \frac{u + m\sin\theta(-l\dot{\theta}^2 + g\cos\theta)}{M + m(1-\cos^2\theta)}, \\ \ddot{\theta} &= \frac{g(m+M)\sin\theta + \cos\theta(u - lm\dot{\theta}^2\sin\theta)}{l(M + m(1-\cos^2\theta))}. \end{aligned} \tag{23.12}$$

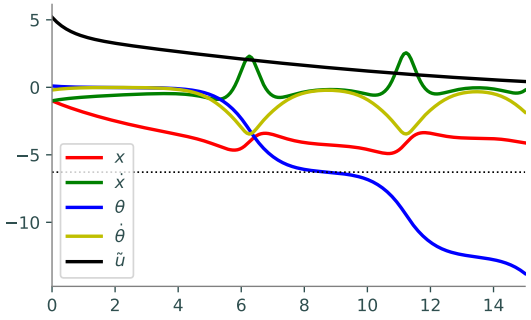
Problem 5. Using the same variables and initial conditions as in Problem 4, and the cubic spline interpolation of the control found in Problem 4, solve for the state variables $\tilde{z}$. Plot $\tilde{z}$, as well as the solution $\tilde{u}$ found in Problem 4 with `tf=15`. Compare your results with the first image in Figure 23.4. Notice that the initial θ is large enough that the inverted pendulum is not balanced. Instead, it falls over at about 5 seconds.

The problem is that the the linearized system is only a valid approximation of the true nonlinear system on small time intervals. So let's solve for the optimal control of the linearized system one small interval at a time.

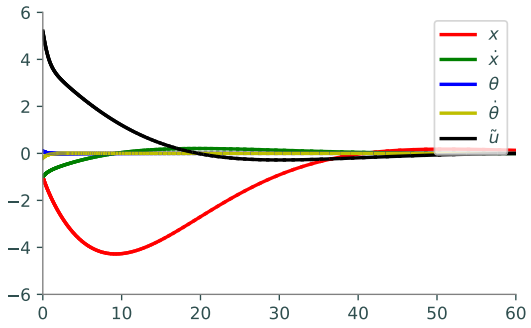
Starting with the initial condition given in Problem 4, use your `rickshaw` solver (from Problem 3) to solve the linearized system over a small interval $t \in [t_0, t_1]$ and obtain the control $\tilde{u}$. Interpolate this $\tilde{u}$ (again using `CubicSpline`), then use this interpolation to evolve the true nonlinear system on the same interval. Now we repeat the process by taking the final state $\tilde{z}$ on this interval as our new initial condition for the next interval $t \in [t_1, t_2]$. Plugging this initial condition into `rickshaw`, we obtain a control which we then interpolate and use to evolve the nonlinear system. Continue this until you have solved over the entire time interval $t \in [0, 60]$.

Plot the pieced-together $\tilde{z}$ and $\tilde{u}$. Be sure to include a legend. Compare your results with the second image in Figure 23.4.

Hint: use `np.geomspace(1,61,120)-1` to get a list of $\{t_0, t_1, ...t_f\}$ that will work well. Solving the equation more often at the beginning will keep the control variable approximately continuous. You only need to solve for the control $\tilde{u}$ at 3 points in each interval $[t_i, t_{i+1}]$ for the method to work well. Also, use `solve_continuous_are` rather than `scipy.optimize.root` when solving the linearized system.



Resulting state equations if the linearized equation is only solved once before the θ is sufficiently small. Note that the pendulum falls and starts spinning at around 5 seconds. The pendulum makes a full revolution at about 9 seconds (2π rotations is marked by the dotted line).



The resulting state equations if the linearized equation is solved again every two seconds. Note that when θ is larger in the first 2 seconds, the control is not sufficiently large. This is issue is resolved by solving the equation again at 2 seconds. Compare this solution to the second image in Figure 23.3

Figure 23.4: The solutions of Problem 5.

Additional Material

Linearization

Recall that a first-order Taylor approximation of a function $\mathbf{f}(\mathbf{x})$, centered at $\mathbf{x}_0$, is

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0).$$

We linearize the right-hand sides of equations (23.7) around $(\theta, \dot{\theta}) = (0, 0)$ as follows:

$$\begin{aligned}
H_1(\theta, \dot{\theta}) &\coloneqq ml\ddot{\theta}\cos\theta - (M+m)\ddot{x} - ml\dot{\theta}^2\sin\theta \\
&\approx H_1(0,0) + DH_1(0,0)\begin{bmatrix}\theta - 0 \\ \dot{\theta} - 0\end{bmatrix} \\
&= ml\ddot{\theta} - (M+m)\ddot{x} \\
&\quad + \left[-ml\ddot{\theta}\sin\theta - ml\dot{\theta}^2\cos\theta, -2ml\dot{\theta}\sin\theta\right]\Big|_{(\theta,\dot{\theta})=(0,0)}\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} \\
&= ml\ddot{\theta} - (M+m)\ddot{x} + [0,0]\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} \\
&= ml\ddot{\theta} - (M+m)\ddot{x}.
\end{aligned}$$

Similarly,

$$\begin{aligned}
H_2(\theta, \dot{\theta}) &\coloneqq g\sin\theta + \ddot{x}\cos\theta \\
&\approx H_2(0,0) + DH_2(0,0)\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} \\
&= \ddot{x} + [g\cos\theta - \ddot{x}\sin\theta,\ 0]\Big|_{(\theta,\dot{\theta})=(0,0)}\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} \\
&= \ddot{x} + [g, 0]\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} \\
&= \ddot{x} + g\theta.
\end{aligned}$$

24 Linear Quadratic Gaussian Control

Lab Objective: *We explore the Linear Quadratic Gaussian (LQG) controller, a combination of the Kalman filter and the Linear Quadratic Regulator (LQR).*

The Linear Quadratic Regulator (LQR) finds the optimal control given certain restrictions on the state evolution and cost functional. In a continuous-time system, the optimal control satisfies

$$\begin{aligned}\min_{\mathbf{u}} J[\mathbf{u}] = \min_{\mathbf{u}} \int_0^{t_f} \left[\mathbf{x}^\mathsf{T} Q \mathbf{x} + \mathbf{u}^\mathsf{T} R \mathbf{u}\right] dt + \mathbf{x}(t_f)^\mathsf{T} M \mathbf{x}(t_f)\\ \mathbf{x}'(t) = A(t)\mathbf{x}(t) + B(t)\mathbf{u}(t), \qquad t \in (0, t_f)\\ \mathbf{x}(0) = \mathbf{x}_0\end{aligned}$$

where $\mathbf{x}$ is the state vector, $\mathbf{u}$ is the control vector, and the cost matrices Q and M are positive semi-definite and R is positive definite.

We'd like to handle noisy state evolution and noisy, incomplete observations of the state. LQR alone isn't able to handle this, but fortunately the Kalman filter comes to our rescue. In fact, the combination of LQR with a Kalman filter is known as a Linear Quadratic Gaussian (LQG) controller. The Kalman filter computes optimal state estimates given state observations, and the LQR component computes optimal controls given these state estimates.

LQG is able to handle Gaussian noise processes in both the state evolution and measurements, accounting for measurement errors, noise in the state evolution, and model error. In this lab we'll build a discrete LQG controller. We'll start by building the LQR component.

We begin by discretizing the continuous-time LQR system above as follows:

$$\min_{\mathbf{u}} J[\mathbf{u}] = \min_{\mathbf{u}} \sum_{k=0}^{N-1} \left[\mathbf{x}_k^\mathsf{T} Q \mathbf{x}_k + \mathbf{u}_k^\mathsf{T} R \mathbf{u}_k\right] + \mathbf{x}_N^\mathsf{T} M \mathbf{x}_N \tag{24.1}$$

$$\mathbf{x}_k = A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1}, \qquad k = 1, \dots, N \tag{24.2}$$

with $\mathbf{x}_0$ fixed. Note that $\mathbf{u}$ is the concatenation of the $\mathbf{u}_k$, i.e., $\mathbf{u} = (\mathbf{u}_0, \dots, \mathbf{u}_{N-1})$.

Using Pontryagin's Maximum Principle, one can show that the optimal control is given by

$$K_{k-1} = -\left(R + B^\mathsf{T} P_k B\right)^{-1} B^\mathsf{T} P_k A, \qquad k = N, \dots, 1 \tag{24.3}$$

$$\begin{aligned} P_{k-1} &= Q + A^\mathsf{T} P_k A - A^\mathsf{T} P_k B (R + B^\mathsf{T} P_k B)^{-1} B^\mathsf{T} P_k A \\ &= Q + A^\mathsf{T} P_k A - A^\mathsf{T} P_k B(-K_{k-1}), \qquad k = N, \dots, 1 \end{aligned} \tag{24.4}$$

$$P_N = M \tag{24.5}$$

$$\mathbf{u}_k = K_k \mathbf{x}_k, \qquad k = 0, \dots, N-1. \tag{24.6}$$

Note that equation (24.4) is the *discrete-time algebraic Riccati equation*. See the Volume 4 textbook for a derivation of these equations.

Problem 1. Build an `LQR` class. Define `__init__` to accept and save as attributes the cost matrices Q, M, and R, the transition matrices A and B, and the number of time steps N. Define a `fit` method that computes and saves the gain matrices K_k. Finally, define a `compute_control` method that accepts an index k and a state $\mathbf{x}_k$ and returns the optimal control $\mathbf{u}_k$.

A Specific Example

Suppose we have a shuttle in outer space at position $(s_{x0}, s_{y0}, s_{z0}) = (-10, 20, 30)$ km with initial velocity $(v_{x0}, v_{y0}, v_{z0}) = (0, 0, 0)$ km/s. We want to get the shuttle to the origin at some final time step T and also end with zero velocity. Letting the state $\mathbf{x} = (s_x, s_y, s_z, v_x, v_y, v_z)$ be the concatenation of the position and velocity and the control $\mathbf{u} = (u_x, u_y, u_z)$ be the thrust, the continuous evolution of this system is given by

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} s_x' \\ s_y' \\ s_z' \\ v_x' \\ v_y' \\ v_z' \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ u_x \\ u_y \\ u_z \end{bmatrix} \tag{24.7}$$

We then discretize with forward Euler,

$$\begin{aligned} \mathbf{x}' &= \frac{\mathbf{x}_{k+1} - \mathbf{x}_k}{\Delta t} \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \Delta t\, \mathbf{x}', \end{aligned}$$

yielding the discrete evolution of the system

$$\begin{aligned} s_{x,k+1} &= s_{x,k} + \Delta t\, v_{x,k} \\ s_{y,k+1} &= s_{y,k} + \Delta t\, v_{y,k} \\ s_{z,k+1} &= s_{z,k} + \Delta t\, v_{z,k} \\ v_{x,k+1} &= v_{x,k} + \Delta t\, u_{x,k} \\ v_{y,k+1} &= v_{y,k} + \Delta t\, u_{y,k} \\ v_{z,k+1} &= v_{z,k} + \Delta t\, u_{z,k}. \end{aligned}$$

Problem 2. Assuming the position and velocity vectors have been concatenated into one state vector $\mathbf{x}_k$, create the evolution matrices A and B from the evolution equations given above and Equation 24.2. Use $\Delta t = 1/10$.

Problem 3. Use your `LQR` class from Problem 1 to get the shuttle to the origin. Use the initial condition specified above, your transition matrices A and B from Problem 2, $Q = 0$ (the zero matrix), choose R and M diagonal, and let $N = 100$. Make sure to choose a large enough M relative to R so that your shuttle reaches the origin with velocity close to zero.

We've provided a `Simulator` class to handle the state evolution, that is, computing and storing the state using the control you provide. The code below will help you use it. You may also view the docstrings of the class and methods either with your code editor or by using Python's `help()` function (e.g., `help(Simulator)` and `help(Simulator.<some_method>)`).

```
import numpy as np
from utils import Simulator

x0 = ...

dt = 1/10
n = 6 # The dimension of the state vector

def f(x, u):
    """Return dx/dt using equation (24.7)."""
    return np.concatenate([x[3:], u])
sim = Simulator(f, dt, n)

# Set the initial state of the simulation.
sim.set_initial_state(x0)

# 1. Compute the control `u0` with your LQR class.
# ...

# 2. Evolve the system.
sim.evolve(u0)

# 3. Get the next true state.
# (`Simulator` assumes the observation matrix H is the identity.)
x1 = sim.observe()

# Repeat 1-3
```

To verify your solution, plot the position, velocity, and sequence of controls. You should see each position and velocity coordinate approach zero by the final time step. Compare with Figure 24.1.

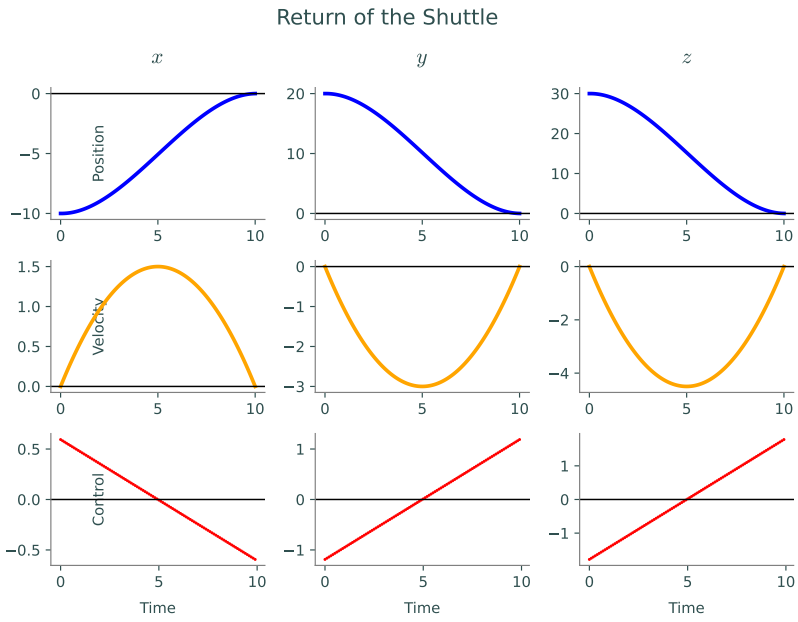


Figure 24.1: Solution to Problem 3.

Incorporating Noise and Observations

We've now handled a deterministic discrete LQR problem, in which we assume that we can see the whole state and that the state evolution has no noise. This of course isn't realistic. In a real scenario, we likely can't observe the whole state—in the situation described above, it's possible we only measure the position but we don't have access to the velocity. Moreover, there is probably noise in the observations (no measurement is perfect) as well as noise in the evolution itself (such as air currents once the shuttle enters an atmosphere, or our model of the physics isn't quite right). LQR alone simply isn't equipped to handle this scenario, so we'll build a Kalman filter class to complete our LQG controller.

We now model the state evolution and state observations with the following equations:

$$\mathbf{x}_0 = \boldsymbol{\mu}_0 + \mathbf{w}_0 \tag{24.8}$$

$$\mathbf{x}_k = A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{w}_k \tag{24.9}$$

$$\mathbf{z}_k = H\mathbf{x}_k + \mathbf{d}_k. \tag{24.10}$$

We now have Gaussian noise vectors $\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, W_k)$ and $\mathbf{d}_k \sim \mathcal{N}(\mathbf{0}, D_k)$. We let $\boldsymbol{\mu}_0$ be the expected value of the initial condition $\mathbf{x}_0$, i.e., $\mathbf{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, W_0)$. We also have noisy observations $\mathbf{z}_k$ obtained from the true state multiplied by some observation matrix H. It's important to understand that these equations represent theoretical assumptions—a model—used by the Kalman filter. When solving a real world problem, we don't know the real states $\mathbf{x}_k$, we have to design and choose the matrices ourselves, and we choose $\boldsymbol{\mu}_0$ using our best guess. Compare equation (24.9) with (24.2).

(Note that H is unrelated to the Hamiltonian which we frequently denote by the same symbol. In this lab we don't use the Hamiltonian, so H will have only one meaning.)

Because we now have noise, we must wrap our cost functional in an expectation:

$$\min_{\mathbf{u}} J[\mathbf{u}] = \min_{\mathbf{u}} \mathbb{E}\left[\sum_{k=0}^{N-1} \left[\mathbf{x}_k^\mathsf{T} Q \mathbf{x}_k + \mathbf{u}_k^\mathsf{T} R \mathbf{u}_k\right] + \mathbf{x}_N^\mathsf{T} M \mathbf{x}_N\right]. \tag{24.11}$$

We can fit a Kalman filter ahead of time using the following equations:

$$
\begin{aligned}
S_{0|-1} &= W_0 \\
S_{k|k-1} &= AS_{k-1|k-1}A^\mathsf{T} + W_k, && k = 1, \ldots, N && (24.12)\\
L_k &= S_{k|k-1}H^\mathsf{T}(HS_{k|k-1}H^\mathsf{T} + D_k)^{-1}, && k = 0, \ldots, N && (24.13)\\
S_{k|k} &= (I - L_kH)S_{k|k-1}, && k = 0, \ldots, N. && (24.14)
\end{aligned}
$$

where $S_{k|k}$ are the state covariance matrices and L_k are the Kalman gain matrices. (In Volume 3, we use P_k to denote the covariance matrices and K_k to denote the gain matrices, but this coincides with our LQR co-state and gain matrices.)

At run-time, we use the following equations to compute the optimal state estimates $\hat{\mathbf{x}}_{k|k}$ using our observations:

$$
\begin{aligned}
\hat{\mathbf{x}}_{0|-1} &= \boldsymbol{\mu}_0, \\
\hat{\mathbf{x}}_{k|k-1} &= A\hat{\mathbf{x}}_{k-1|k-1} + B\mathbf{u}_{k-1}, && k = 1, \ldots, N && (24.15)\\
\hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + L_k(\mathbf{z}_k - H\hat{\mathbf{x}}_{k|k-1}), && k = 0, \ldots, N. && (24.16)
\end{aligned}
$$

Equation (24.15) is known as the "predict" step and (24.16) is the "update" step. We use our LQR class to compute each optimal control $\mathbf{u}_k$ using equation (24.6). However, since we don't know the true state $\mathbf{x}_k$, we must substitute the state estimate $\hat{\mathbf{x}}_{k|k}$ instead.

Problem 4. Build a `KalmanFilter` class. Define `__init__` to accept and save as attributes the transition matrices A and B and the observation matrix H. We will assume that the noise covariance matrices are the same for all k, so also save as attributes the noise covariance matrix W and D, and the number of time steps N. Define a `fit` method that computes and saves the gain matrices L_k. Define a `predict_state` method that accepts the last estimated state $\hat{\mathbf{x}}_{k-1|k-1}$ and a control $\mathbf{u}_{k-1}$ and returns the predicted state $\hat{\mathbf{x}}_{k|k-1}$. Finally, define an `update_state` method that accepts an index k, a predicted state $\hat{\mathbf{x}}_{k|k-1}$, and an observation $\mathbf{z}_k$ and returns the updated state estimate $\hat{\mathbf{x}}_{k|k}$.

Problem 5. Use LQG—your `LQR` and `KalmanFilter` classes—to solve the same scenario as in Problem 3. However, define H to allow observation of position $\mathbf{s}$ but *not* velocity $\mathbf{v}$. Use the same initial condition $\boldsymbol{\mu}_0 = (s_{x0}, s_{y0}, s_{z0}, v_{x0}, v_{y0}, v_{z0})$, and set $W_k = 0.05I$ and $D_k = 0.5I$ for all k.

Again, use the provided `Simulator` class. This time, in addition to letting it handle the state evolution (as it did in Problem 3), it will also provide the state observations. To do so, initialize it with the observation matrix and the noise covariances, and use its `observe` method in addition to its `evolve` method, as in the code below. Part of the purpose of the `Simulator` class is to make the problem more realistic by hiding the true states. Although true states are artificially generated in this lab (and in most example code demonstrating the Kalman filter), in real problems the true states are never known. The `Simulator` class handles the true states behind-the-scenes so that the only information dealt with here are the observations and the estimated states.

```
mu0 = ...
W = ...
```

```
D = ...
H = ...

dt = 1/10
def f(x, u):
    """Return dx/dt using equation (24.7)."""
    return np.concatenate([x[3:], u])
sim = Simulator(f, dt, W=W, D=D, H=H)

# Set the initial state of the simulation.
sim.set_initial_state(mu0)

# 1. Compute the control `u0` with your LQR class and the first
# estimated state (in this case mu0).
# ...

# 2. Evolve the system.
sim.evolve(u0)

# 3. Get the observation.
z1 = sim.observe()

# 4. Estimate x1 using the observation and the last estimated state.
# ...

# Repeat 1-4
```

Again, plot the true position and velocity (accessible through `sim.true_states`) along with the sequence of controls. You should see each position and velocity coordinate approach zero, though it's likely none will reach zero due to noise. Your plots should look similar to Figure 24.2.

Nonlinear Problems

In the real world, we don't always have the luxury of working on linear problems. We often want to compute optimal controls and state estimates on nonlinear systems. Various algorithms have been devised, each with strengths and weaknesses, but this is still an active area of research. For the final part of this lab, we'll use one algorithm known as *iterative LQG* (iLQG).[1]

[1] E. Todorov and W. Li, "A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems," in Proceedings of the 2005 American Control Conference, 2005, pp. 300-306, doi: 10.1109/ACC.2005.1469949.

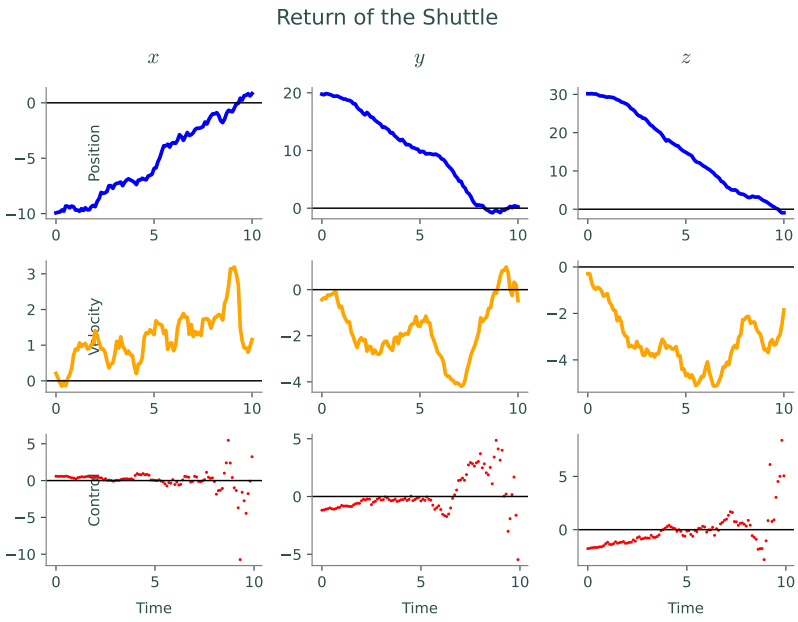


Figure 24.2: Solution to Problem 5.

Rescuing Neil Armstrong

Michael Collins, the command module pilot of Apollo 11, has been forced to leave Neil Armstrong and Buzz Aldrin in lunar orbit, and now he must return in a new shuttle to pick them up! Collins will need to carefully manage the fuel of the Apollo 11.5 to make sure he has enough to pick up Armstrong and Aldrin and make it back to Earth safely.

Collins will have to navigate the Apollo past the earth and the moon. We'll need to model the gravitational pull of both of these celestial bodies. Suppose the Apollo is at position $\mathbf{s}$ and a celestial body is at position $\mathbf{s}_P$ with mass m_P. Then the acceleration of the Apollo due to the body's gravity is given by

$$\mathbf{a}_P = \frac{\mathbf{s}_P - \mathbf{s}}{\|\mathbf{s}_P - \mathbf{s}\|} \cdot \frac{G\, m_P}{\|\mathbf{s}_P - \mathbf{s}\|^2}$$

where G is the gravitational constant and $\|\cdot\|$ is the 2-norm. The first term gives the direction of the acceleration as a unit vector and the second gives the magnitude.

As we did earlier in this lab, we'll define $\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{u})$, but this time we'll pass this evolution function directly to iLQG and let it handle the linearization and discretization. Letting $\mathbf{x} = (\mathbf{s}, \mathbf{v})$ be the concatenation of the position of the shuttle $\mathbf{s} = (s_x, s_y, s_z)$ and its velocity $\mathbf{v} = (v_x, v_y, v_z)$, we'll use

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \mathbf{v} \\ \mathbf{a}_E + \mathbf{a}_M + \mathbf{u} \end{bmatrix} \tag{24.17}$$

where $\mathbf{a}_E$ and $\mathbf{a}_M$ are the accelerations of the shuttle due to the earth's gravity and the moon's gravity, respectively.

Problem 6. Use the provided `Simulator`, `Estimator`, and iLQG `Controller` classes to plan Collins' rescue of Armstrong and Aldrin. You may find it **very helpful** to reference the code outline below and the **documentation** of the provided classes. The `Controller` class is analogous to the `LQR` class created in Problem 1 and the `Estimator` class is analogous to the `KalmanFilter` class created in Problem 4.

The `Controller` class requires a physical time horizon T and a number of time steps N. We'll use `T = 48` hours and `N = 1000`, so that the time step size Δt is `dt = T / N`. Use the cost matrices $M = 10I$, $Q = 0$, and $R = I$. For `Simulator` and `Estimator`, use the state and observation covariance matrices $W = (0.05\,\Delta t)^2 I$ and $D = (0.05\,\Delta t)^2 I$. Use the same observation matrix H you defined in Problem 5. Also, use the constants defined in the code block below to calculate accelerations. Assume that the expected initial position and velocity are as given by `mu0` in the code block below. Note that our target, the location of Armstrong and Aldrin, is placed at the origin.

Using the file `animate.py`, animate your results using the provided `animate2d` function. Make sure use `simulator.true_states` and the controls you compute, *not* `controller.xs` or `controller.us` which are the sequences of states and controls in the absence of noise. See the docstrings for documentation on how to use the provided code.

As a sanity check to debug your animation, you may wish to plot the true position and velocity and the sequence of controls, as in Problem 5, and compare with Figure 24.3. Note that this is NOT the solution to problem, only a check to help you create your animation.

Important: Since the provided classes use `jax`, when defining the state evolution (24.17), use `jax.numpy` instead of `numpy` or `scipy` functions. For example, use `jnp.linalg.norm` and `jnp.concatenate`. However, you can still define individual arrays (such as the earth's position) using `numpy`.

Hint: Remember you can use either your code editor or Python's `help()` function to view the docstrings of the provided classes and methods.

```
import numpy as np
from jax import numpy as jnp
from utils import Simulator, Estimator, Controller

from animate import animate2d

# The following quantities are in metric tons, kilometers, and hours.
# We've placed the Armstrong and Aldrin at the origin.
mass_earth = 5.9722e21
mass_moon = 7.3e19
position_earth = np.array([-96100, -480500, 0], dtype=float)
position_moon = np.array([-96100, -96100, 0], dtype=float)
mu0 = np.array([-48050, -576600, 100, 0, 0, 0], dtype=float)
G = 8.6499e-10

def f(x, u):
    # Remember to use `jnp` functions instead of `np`.
    ...
```

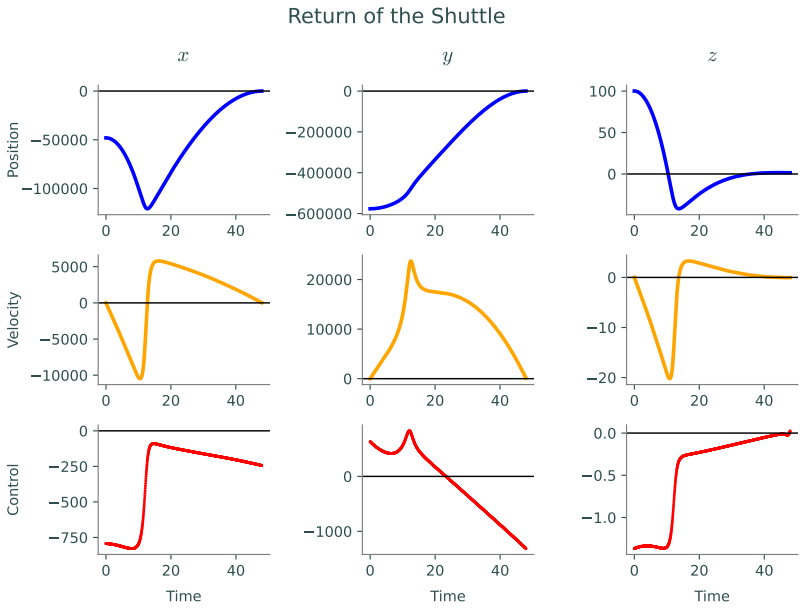
```
# Other guesses are possible for `Controller`, but not all converge
# to good solutions.
us_guess = np.full((N, 3), np.array([-100, 100, -1]), dtype=float)
controller = Controller(...)
controller.fit(mu0, ...)

simulator = Simulator(...)
sim.set_initial_state(mu0)

estimator = Estimator(...)
# Fit the Estimator using the linearized dynamics found by `Controller`.
estimator.fit(controller.As)

# Now repeat the process you used in Problem 5.
```

If you like, you may use the function `animate3d` in the `animate.py` file to animate Collins' rescue mission in 3D.

Achtung!

Make sure to push your animation with your solutions! Remember that you can embed your animation in your notebook using the code:

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

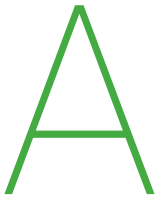


!h

Figure 24.3: Sanity check for Problem 6. This is NOT the solution.

Part II

Appendices



NumPy Visual Guide

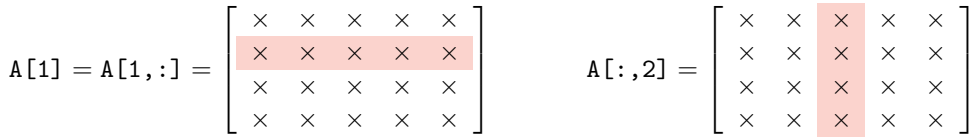
Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

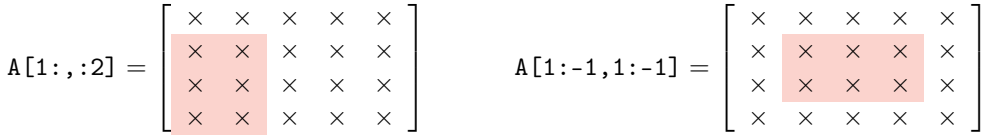
Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

Slicing

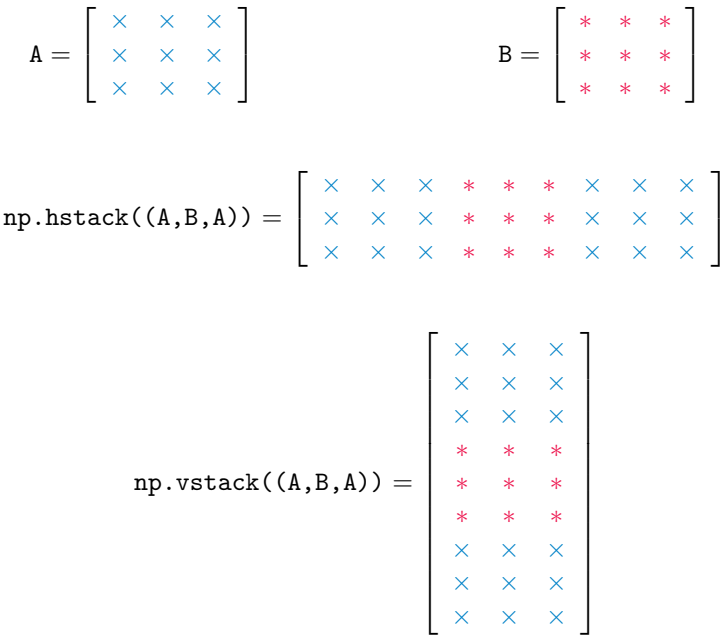
A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



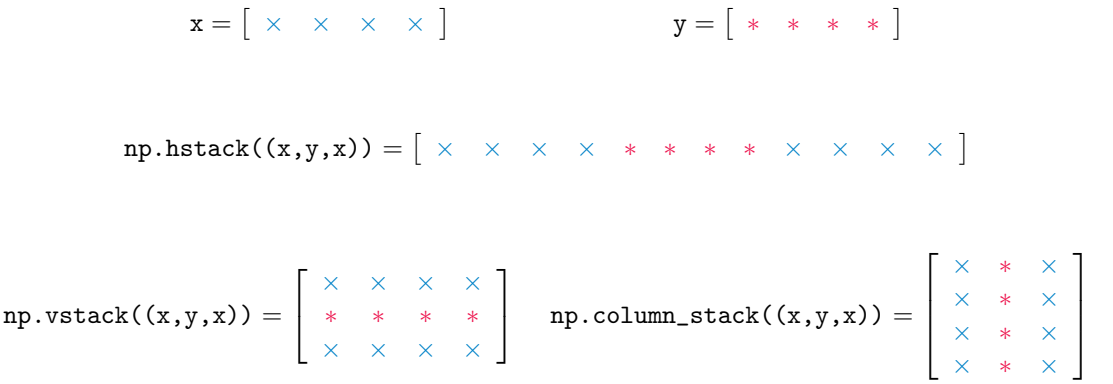


Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.



Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.



The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((1,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

B Matplotlib Syntax and Customization Guide

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at* `https://matplotlib.org/`.

Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.

2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
`add_subplot()`	Add a single subplot to the current figure
`axes()`	Add an axes to the current figure
`clf()`	Clear the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

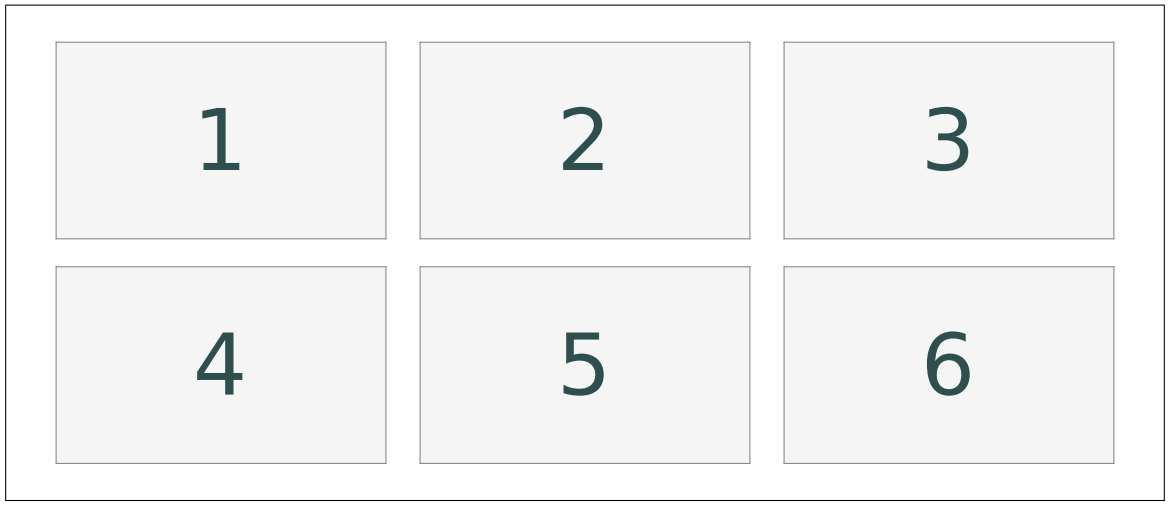


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current axes.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

Achtung!

Be careful not to mix up the following similarly-named functions:

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` or `ax.axis()` sets properties of the x- and y-axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Plot Customization

Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at `https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`.

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    # ...
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

Plot layout

Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the x and y ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin, xmax, ymin, ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the x- and y-scales equal (i.e. makes circles circular).

Function	Description
`axis()`	set the x- and y-limits of the plot
`grid()`	add gridlines
`xlim()`	set the limits of the x-axis
`ylim()`	set the limits of the y-axis
`xticks()`	set the location of the tick marks on the x-axis
`yticks()`	set the location of the tick marks on the y-axis
`xscale()`	set the scale type to use on the x-axis
`yscale()`	set the scale type to use on the y-axis
`ax.spines[side].set_position()`	set the location of the given spine
`ax.spines[side].set_color()`	set the color of the given spine
`ax.spines[side].set_visible()`	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an x- or y-coordinate of zero, respectively. The others are a passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an x- or y-coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of $\sin(x)$. The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```
>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()
```

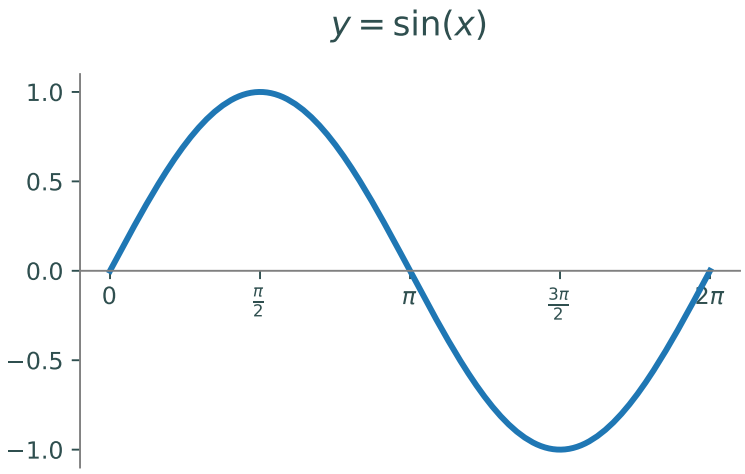


Figure B.2: Plot of $y = \sin(x)$ with axes modified for clarity

Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`.

The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements.

The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

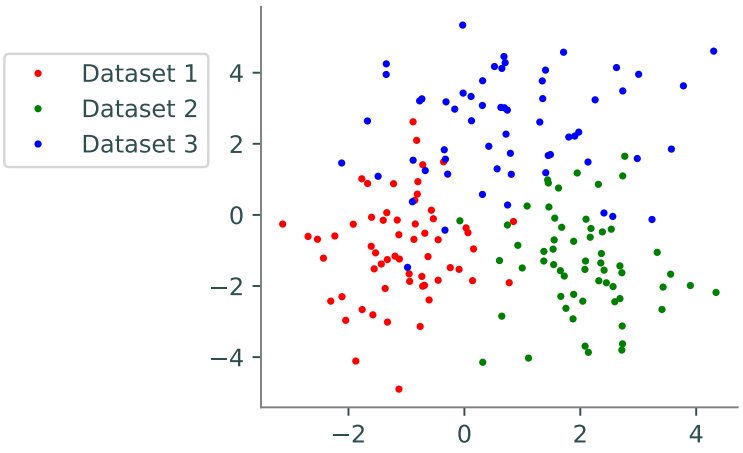


Figure B.3: Example of repositioning axes.

Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specific colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at `https://matplotlib.org/stable/gallery/color/named_colors.html`. If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta

Code	Color
`'y'`	yellow
`'k'`	black
`'w'`	white
`'C0'` - `'C9'`	Default colors

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of `1.0` corresponds to fully opaque, and `0.0` corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at `https://matplotlib.org/stable/gallery/color/colormap_reference.html`. Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
`annotate()`	adds a commentary at a given point on the plot	annotate('text',(x,y))
`arrow()`	draws an arrow from a given point on the plot	arrow(x,y,dx,dy)
`colorbar()`	Create a colorbar	colorbar()
`legend()`	Place a legend in the plot	legend(loc='best')
`text()`	Add text at a given position on the plot	text(x,y,'text')
`title()`	Add a title to the plot	title('text')
`suptitle()`	Add a title to the figure	suptitle('text')
`xlabel()`	Add a label to the x-axis	xlabel('text')
`ylabel()`	Add a label to the y-axis	ylabel('text')

Table B.4: Text and annotation functions in Matplotlib

Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with LaTeX, a system for creating technical documents.[1] To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be $\frac{1}{2}\sin(x^2)$:

```
>>> plt.title(r"$\frac{1}{2}\sin(x^2)$")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of `(x,y)` can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point (0,0) corresponds to the bottom-left of the current subplot, and (1,1) corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')
```

[1] See `http://www.latex-project.org/` for more information.

```
# Create the legend
>>> plt.legend()
```

Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Maplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:
>>> plt.plot(x, y, 'r.')

#To change the size:
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)
>>> plt.scatter(x, y, marker='+', s=12)
```

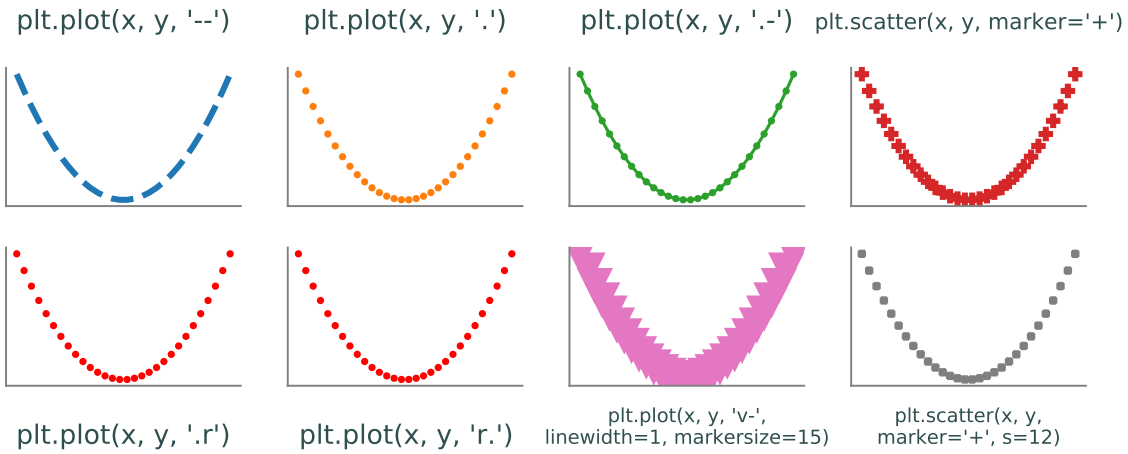


Figure B.4: Examples of setting line and marker styles.

Plot Types

Matplotlib has functions for creaing many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
`bar`	makes a bar graph	bar(x,height)
`barh`	makes a horizontal bar graph	barh(y,width)
`boxplots`	makes one or more boxplots	boxplots(data)
`contour`	makes a contour plot	contour(X,Y,Z)
`contourf`	makes a filled contour plot	contourf(X,Y,Z)
`imshow`	shows an image	imshow(image)
`fill`	plots lines with shading under the curve	fill(x,y)
`fill_between`	plots lines with shading between two given y values	fill_between(x,y1, y2=0)
`hexbin`	creates a hexbin plot	hexbin(x,y)
`hist`	plots a histogram from data	hist(data)
`pcolormesh`	makes a heatmap	pcolormesh(X,Y,Z)
`pie`	makes a pie chart	pie(x)
`plot`	plots lines and data on standard axes	plot(x,y)
`plot_surface`	plot a surface in 3-D space	plot_surface(X,Y,Z)
`polar`	plots lines and data on polar axes	polar(theta,r)
`loglog`	plots lines and data on logarithmic x and y axes	loglog(x,y)
`scatter`	plots data in a scatterplot	scatter(x,y)
`semilogx`	plots lines and data with a log scaled x axis	semilogx(x,y)
`semilogy`	plots lines and data with a log scaled y axis	semilogy(x,y)
`specgram`	makes a spectogram from data	specgram(x)
`spy`	plots the sparsity pattern of a 2D array	spy(Z)
`triplot`	plots triangulation between given points	triplot(x,y)

Table B.6: Some basic plotting functions in Matplotlib.

Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of $z = (x^2 + y)\sin(y)$. The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
```

```
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

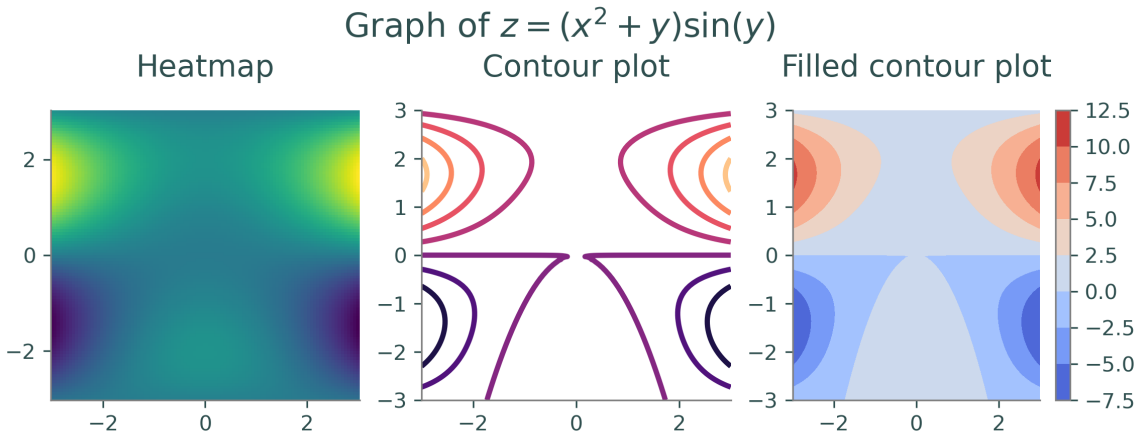


Figure B.5: Example of heatmaps and contour plots.

Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D $n \times m$ array for a grayscale image, or a 3-D $n \times m \times 3$ array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range $[0, 1]$. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")

#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

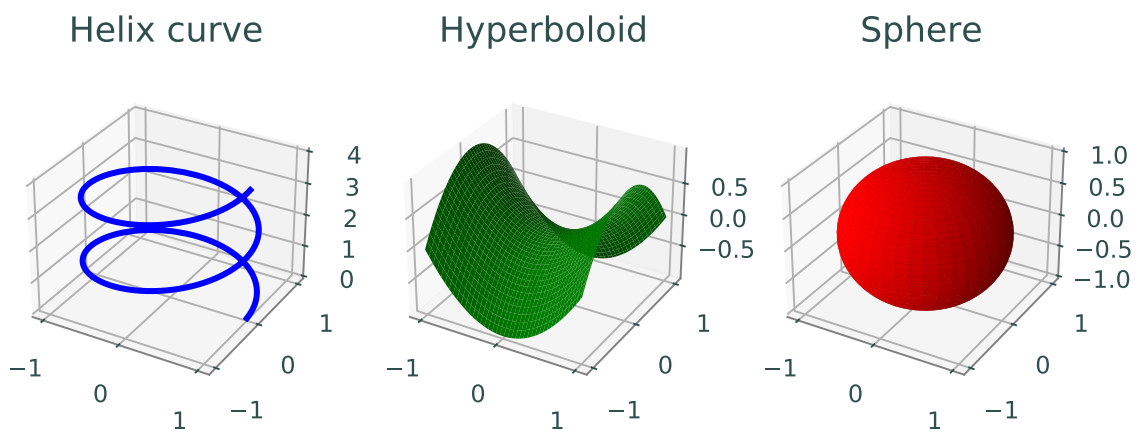


Figure B.6: Examples of 3-D plotting.

Additional Resources

rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the `"figure.dpi"` parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at `https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams`.

Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at `https://matplotlib.org/stable/tutorials/index.html`. It also has a large gallery of examples, found at `https://matplotlib.org/stable/gallery/index.html`. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.