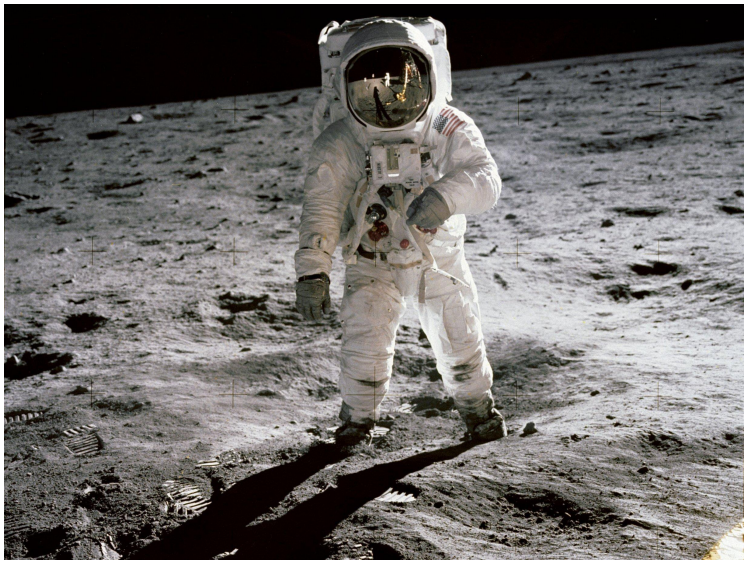
Labs for Foundations of Applied Mathematics

Volume 4
Modeling with Dynamics and Control

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

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

J. Bejarano
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

A. Frandsen
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Fuhriman
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

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

K. Jacobson
Brigham Young University

J. Leete
Brigham Young University

J. Lytle
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

D. Miller
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

M. Stauffer
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

M. Victors
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

A. Zaitzeff
Brigham Young University

This project is funded in part by the National Science Foundation, grant no. TUES Phase II DUE-1323785.

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 Introduction to Matplotlib: 3D Plotting and Animations

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

Introduction

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

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

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

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

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

Animation Background

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

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

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

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

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

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

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

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

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

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

Note

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

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

3D Plotting Introduction

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

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

3D Static Plotting

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

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

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

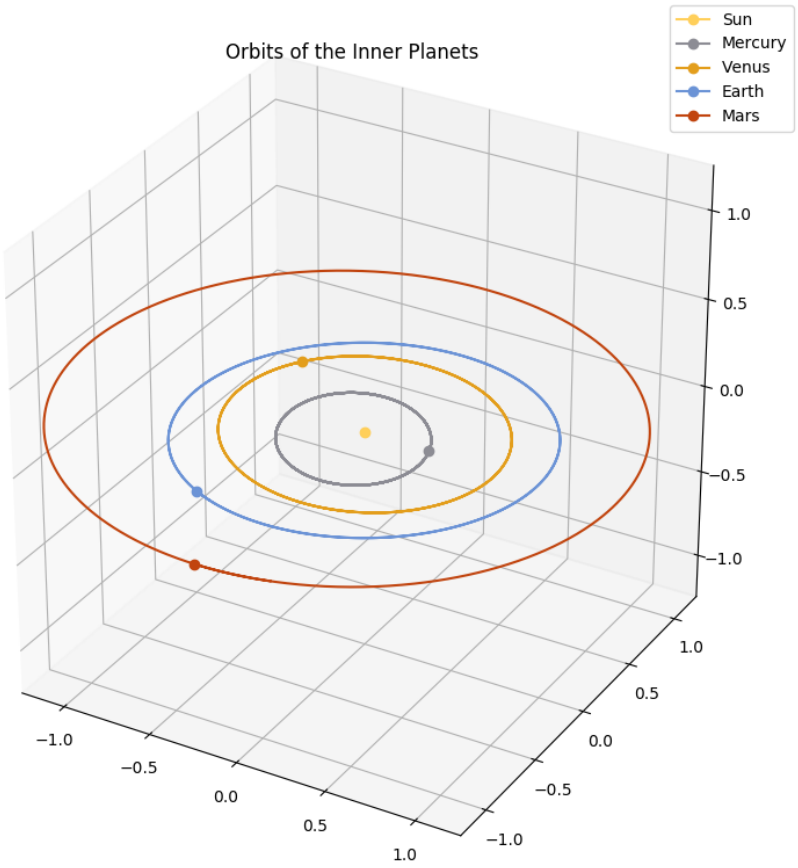


Figure 1.1: The solution to Problem 2.

3D Animations

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

Saving Animations

Animation in 3D requires more careful consideration than in the 2D case. When `matplotlib` displays a 3D plot, it does so in an interactive figure that allows the user to change the camera angle and position. Since 3D rendering is more computationally expensive than 2D rendering, interactive views of 3D animations often have poor framerates and choppy rendering. The solution is to use the `matplotlib.animation` module's `FuncAnimation.save()` method. With an installed video encoder, this allows Matplotlib to render a video file of the animation, which can then be displayed inline inside a Jupyter Notebook, or viewed using any video player supporting the chosen filetype.

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

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

```
>>> animation.writer = animation.writers['ffmpeg']
>>> plt.ioff()    # Turn off interactive mode to hide rendering animations
>>>
>>> # Code to create figure, axes, update function
>>>
>>> ani = animation.FuncAnimation(fig,update,frames)
>>> ani.save('my_animation.mp4')
```

Problem 3. Each row of the arrays in `orbits.npz` gives the position of the planets at evenly spaced time points. The arrays correspond to 1400 points in time over a 700 day period (beginning on 2018-5-30). Create a 3D animation of the planet orbits. Display lines for the trajectories of the orbits and points for the current positions of the planets at each point in time. Your `update()` function will need to return a list of Line3D objects, one for each orbit trajectory and one for each planet position marker. Using `animation.save()`, save your animated plot as "planet_ani.mp4".

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

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

Surface Plotting

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

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

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

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

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

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

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

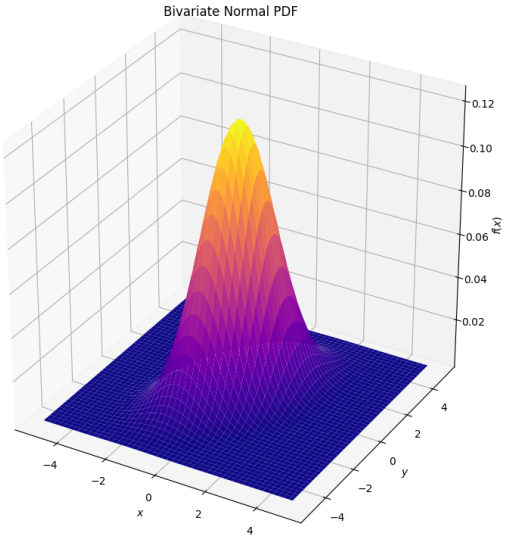


Figure 1.2: The solution to Problem 4.

Surface Animations

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

Problem 5. Use the data in `vibration.npz` to produce a surface animation of the solution to the wave equation for an elastic rectangular membrane. The file contains three NumPy arrays: `X`, `Y`, `Z`. `X` and `Y` are meshgrids of shape `(300,200)` corresponding to 300 points in the y-direction and 200 points in the x-direction, giving a 2x3 rectangle with one corner at the origin. `Z` is of shape `(150,300,200)`, giving the height of the vibrating membrane at each (x,y) point for 150 values of time. In the language of partial differential equations, this is the solution to the following initial/boundary value problem:

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

2 Numerical Methods for Initial Value Problems; Harmonic Oscillators

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

Methods for Initial Value Problems

Consider the *initial value problem* (IVP)

$$\begin{aligned} \mathbf{x}'(t) &= f(\mathbf{x}(t), t), \quad t_0 \leq t \leq t_f \\ \mathbf{x}(t_0) &= \mathbf{x}_0, \end{aligned} \tag{2.1}$$

where f is a suitably continuous function. A solution of (2.1) 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, (2.1) 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

$$\begin{aligned} x'(t) &= \sin(x(t)), \\ x(0) &= x_0. \end{aligned} \tag{2.2}$$

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 (2.2) 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 2.1 shows several of these approximate solutions, along with some of the constant, or *equilibrium*, solutions.

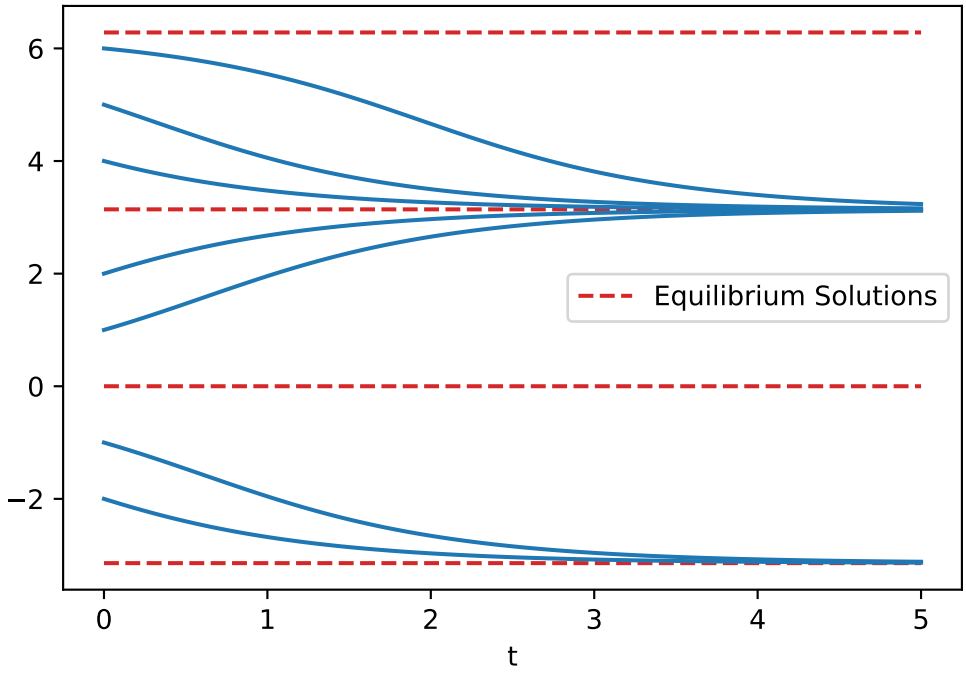


Figure 2.1: Several solutions of (2.2), using `scipy.integrate.odeint`.

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).

Problem 1. Write a function which implements Euler's method for an IVP of the form (2.1). Test your function on the IVP:

$$\begin{aligned} x'(t) &= x(t) - 2t + 4, \quad 0 \leq t \leq 2, \\ x(0) &= 0, \end{aligned} \tag{2.3}$$

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 2.2.

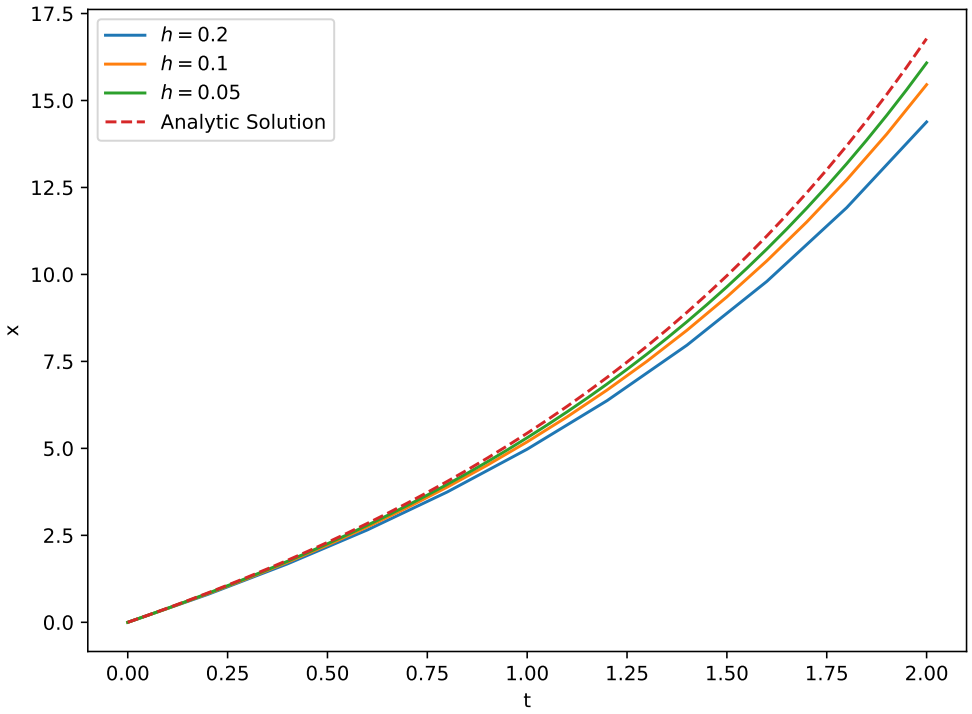


Figure 2.2: The solution of (2.3), alongside several approximations using Euler's method.

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 (2.3) from Problem 1 for step sizes of $h = 0.2$, 0.1, 0.05, 0.025, and 0.0125. Plot the true solution alongside the approximation obtained from each method when $h = 0.2$. Use `plt.loglog` to create a log-log plot of the relative error $|x(2) - x_n|/|x(2)|$ as a function of h for each approximation. Compare your results with Figure 2.3.

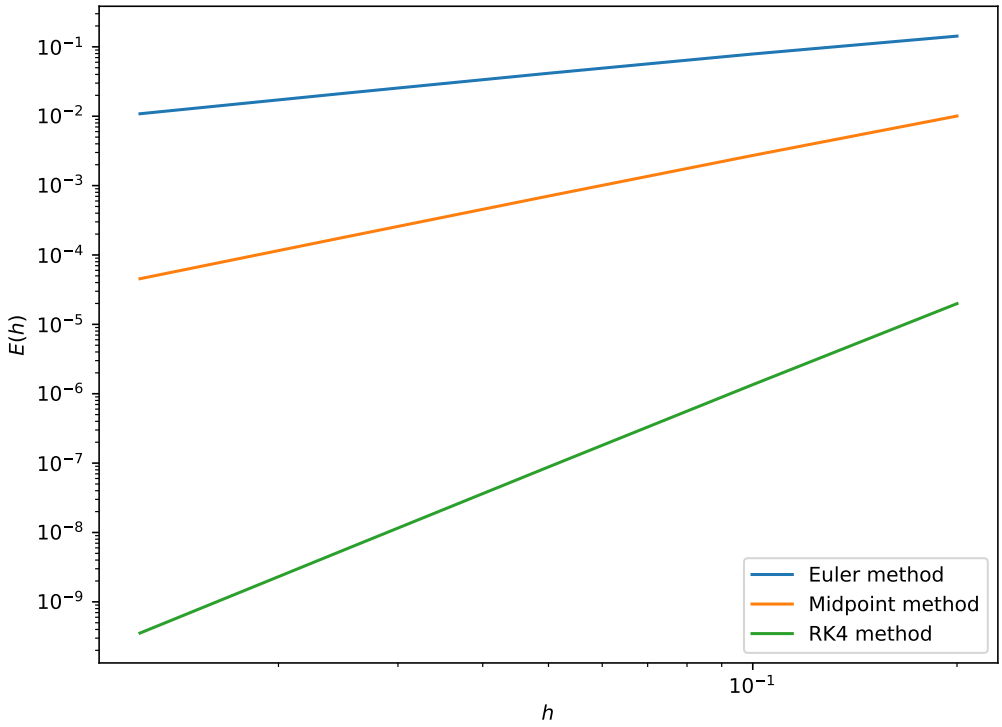


Figure 2.3: Loglog plot of the relative error in approximating $x(2)$, using step sizes $h = 0.2$, 0.1, 0.05, 0.025, and 0.0125. The slope of each line demonstrates the first, second, and fourth order convergence of the Euler, Midpoint, and RK4 methods, respectively.

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

$$\begin{aligned} x'(t) &= x(t)\cos(t), \quad t \in [0, 8], \\ x(0) &= 1. \end{aligned} \tag{2.4}$$

Figure 2.4 illustrates the comparative efficiency of the Euler, Midpoint, and RK4 methods applied to (2.4). 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.

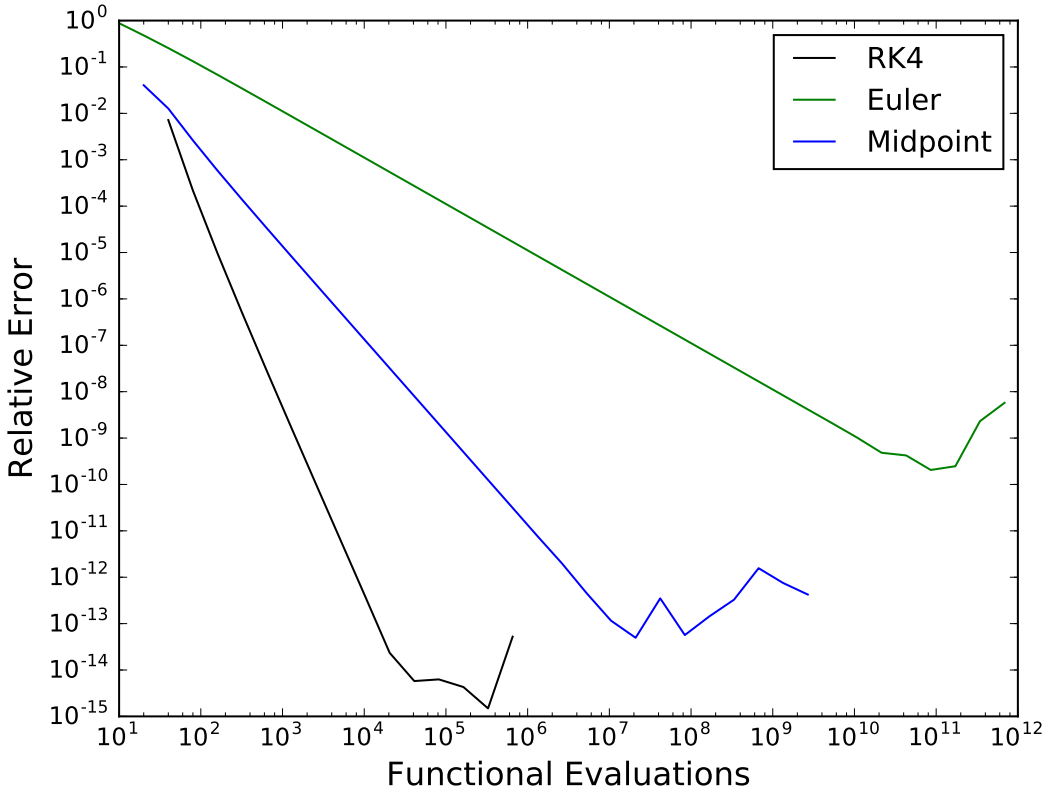


Figure 2.4: The relative error in computing the solution of (2.4) at $x = 8$ versus the number of times the right-hand side of (2.4) must be evaluated.

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 $y(t)$ is a solution to an initial value problem of the form [1]

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

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

$$\begin{aligned} &my'' + ky = 0, \quad 0 \leq t \leq 20, \\ &y(0) = 2, \quad y'(0) = -1, \end{aligned} \tag{2.5}$$

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.

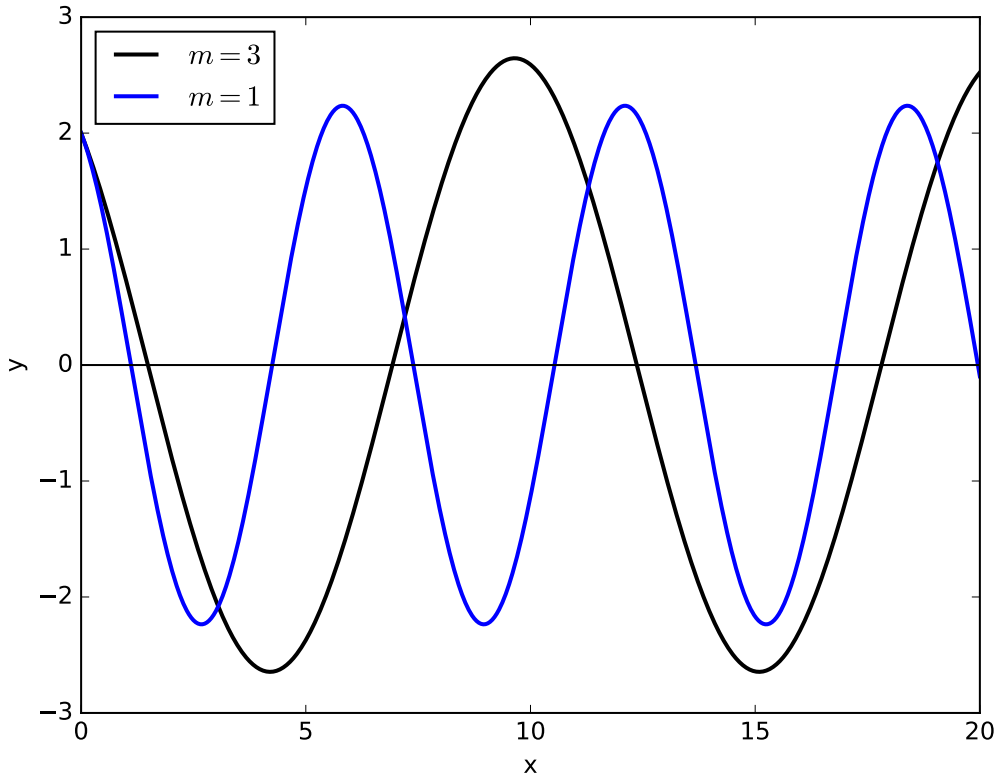


Figure 2.5: Solutions of (2.5) for several values of m.

Damped free harmonic oscillators

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

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

The roots of the characteristic equation are

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

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

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

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

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

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

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

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

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

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

Forced harmonic oscillators without damping

Consider the systems described by the differential equation

$$my''(t) + ky(t) = F(t). \tag{2.6}$$

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 2.6 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 2.6 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

$$\begin{aligned} &2y'' + \gamma y' + 2y = 2\cos(\omega t), \quad 0 \le t \le 40, \\ &y(0) = 2, \quad y'(0) = -1. \end{aligned} \tag{2.7}$$

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 Figure2.7.

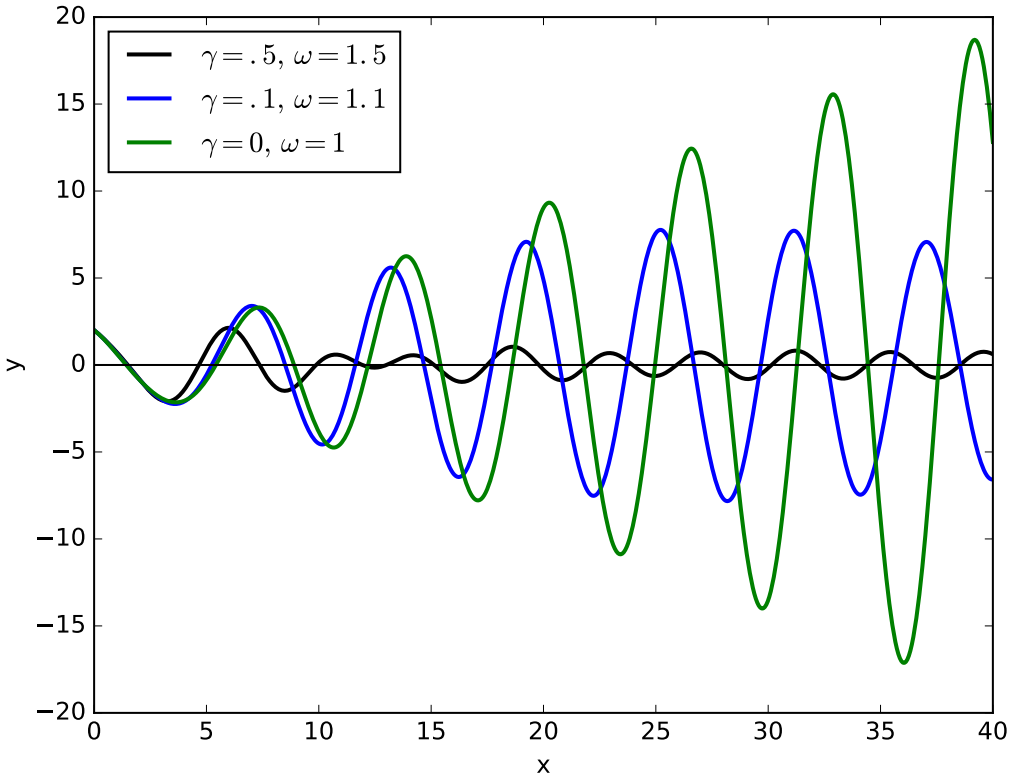


Figure 2.6: Solutions of (2.7) for several values of ω and γ.

3 Predator-Prey and Weight Change Models

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

ODE Solvers

Initial Value Problems (IVPs) are a systems of one or more ordinary differential equations (ODEs) with defined initial conditions. In some cases, these can be solved by hand, but in real life it is more practical to use numerical solvers. In the previous lab, you built your own numerical solvers. For this lab you will use the `odeint` solver from the `scipy.integrate` library.

`odeint` solves a system of ODEs given by $dy/dt = f(y,t), \quad y(t_0) = y_0$, where y can be a vector. The solver takes as parameters the callable function f, the initial condition y_0, and an array of time points t. Then `odeint` returns the array y with shape $(len(t), len(y_0))$, where each row gives the y values for one time point. The syntax for `odeint` is shown below, note this is the same syntax as the solvers built in the previous lab.

```
from scipy.integrate import odeint
sol = odeint(f, y0, t)
```

Assuming that f, $y0$, and t are previously defined (as explained above), *sol* is a vector containing the solution to the IVP and can be visualized by plotting each column of *sol* against the time domain or by plotting the columns against each other.

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 wolve populations respectively at time t, where the unit for time is years. We will make a few assumptions to simplify our model:

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

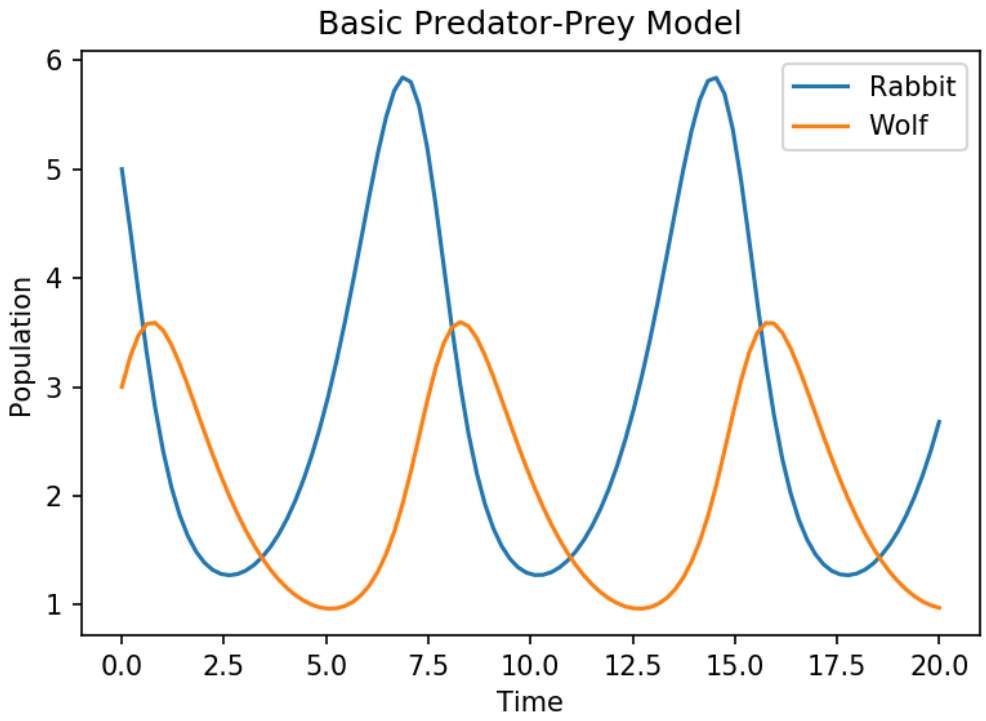


Figure 3.1: The solution to the system found in (3.1)

- In the absence of rabbits, the wolves die out. Thus when $r(t) = 0$, $dw/dt = -\delta w(t)$, where $\delta > 0$.
- The number of encounters between rabbits and wolves is proportional to the product of their populations. The wolf population grows proportional to the number of encounters by $\beta r(t)w(t)$ (where $\beta > 0$), and the rabbit population decreases proportional to the number of encounters by $-\gamma r(t)w(t)$ (where $\gamma > 0$).

This leads to the following system of ODEs:

$$\begin{aligned} \frac{dr}{dt} &= \alpha r - \beta rw = r(\alpha - \beta w) \\ \frac{dw}{dt} &= -\delta w + \gamma rw = w(-\delta + \gamma r) \end{aligned} \tag{3.1}$$

Problem 1. As mentioned above, the `odeint` solver requires a callable function representing the right hand side of the IVP. Define the function `predator_prey()` that accepts the current $r(t)$ and $w(t)$ values as a 1d array y, and the current time t, and returns the right hand side of (3.1) as a tuple. Use $\alpha = 1.0$, $\beta = 0.5$, $\delta = 0.75$, and $\gamma = 0.25$ as your growth parameters.

Problem 2. Use `odeint` to solve (3.1) with initial conditions $(r_0, w_0) = (5, 3)$ and time ranging from 0 to 20 years. Display the resulting rabbit and wolf populations over time (stored as columns in the output of `odeint`) on the same plot. Your graph should match the graph in figure 3.1.

Variations on the Predator-Prey

The Lotka-Volterra model

Reconsider (3.1). This representation of the predator-prey relationship 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.

Let us look at the dynamics of this system. The equlibria (fixed points) of a system occur when the derivatives are zero, for our system this 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

$$\begin{aligned} \frac{dU}{d\bar{t}} &= U(1 - V), \\ \frac{dV}{d\bar{t}} &= \eta V(U - 1). \end{aligned} \tag{3.2}$$

Problem 3. Similar to problem 1, define the function `Lotka_volterra()` that takes in the current predator and prey populations as a 1d array y and the current time as a float t and returns the right hand side of the system (3.2) with $\eta = 1/3$.

The following three lines of code plot the phase portrait of (3.2). For more documentation on quiver plots see https://matplotlib.org/2.0.0/api/_as_gen/matplotlib.axes.Axes.quiver.html.

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

Using `odeint`, solve (3.2) with three different initial conditions $y_0 = (1/2, 1/3)$, $y_0 = (1/2, 3/4)$, and $y_0 = (1/16, 3/4)$ and time domain $t = [0, 13]$. Plot these three solutions on the same graph as the phase portrait and the equilibria $(0, 0)$ and $(1, 1)$.

Since your solutions are being plotted with the phase portrait, plot the two populations against each other (instead of both individually against time). Your plot should match 3.2.

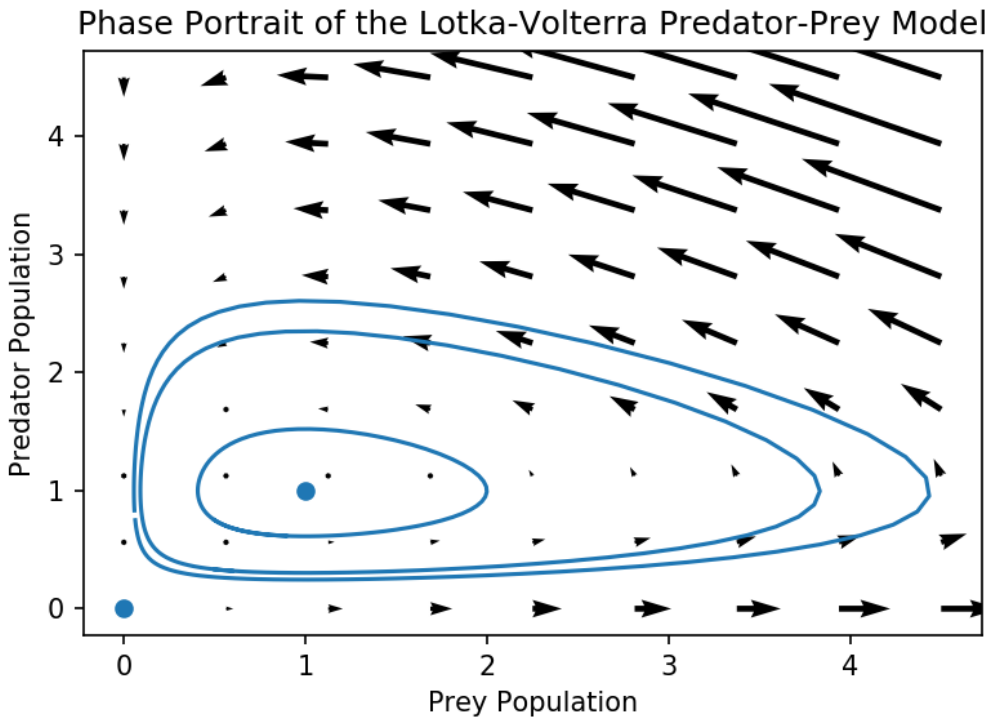


Figure 3.2: The phase portrait for the nondimensionalized Lotka-Volterra predator-prey equations with parameters $\eta = 1/3$.

The Logistic model

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

$$\begin{aligned} \frac{dU}{d\bar{t}} &= U(1 - U - V), \\ \frac{dV}{d\bar{t}} &= \eta V(U - \rho). \end{aligned} \tag{3.3}$$

Problem 4. Define a new function `Logistic_Model()` that takes in the current predator and prey populations y and the current time t and returns the right hand side of (3.3) as a tuple. Use `odeint` to compute solutions (U, V) of (3.3) for initial conditions $(1/3, 1/3)$ and $(1/2, 1/5)$. Do this for parameter values η, $\rho = 1$, 0.3 and also for values η, $\rho = 1$, 1.1.

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

A Weight Change Model

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

$$EB = EI - EE.$$

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

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

$$\begin{aligned} \frac{dF}{dt} &= \frac{(1 - p(t))EB(t)}{\rho_F}, \\ \frac{dL}{dt} &= \frac{p(t)EB(t)}{\rho_L}, \end{aligned} \tag{3.4}$$

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

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

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

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

Notice

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

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

$$p(t) = \frac{C}{C + F(t)} \quad \text{where} \quad C = 10.4\frac{\rho_L}{\rho_F}. \tag{3.5}$$

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

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

$$EE = PAL \times RMR \tag{3.6}$$

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

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

Table 3.1: This is a rough guide for physical activity level (PAL).

We will use the following equation for computing RMR,

$$RMR = K + \gamma_F F(t) + \gamma_L L(t) + \eta_F \frac{dF}{dt} + \eta_L \frac{dL}{dt} + \beta_{at} EI, \tag{3.7}$$

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

Thus, since the input EI is assumed to be known, we can use (3.6), (3.7) and (3.5) to write (3.4) in terms of F and L, thus allowing us to close the system of ODEs.

Specifically, we have

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

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

$$EB(t) = \frac{\left(\frac{1}{PAL} - \beta_{at}\right) EI - K - \gamma_F F(t) - \gamma_L L(t)}{\frac{\eta_F}{\rho_F}(1 - p(t)) + \frac{\eta_L}{\rho_L} p(t) + \frac{1}{PAL}}. \tag{3.8}$$

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

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

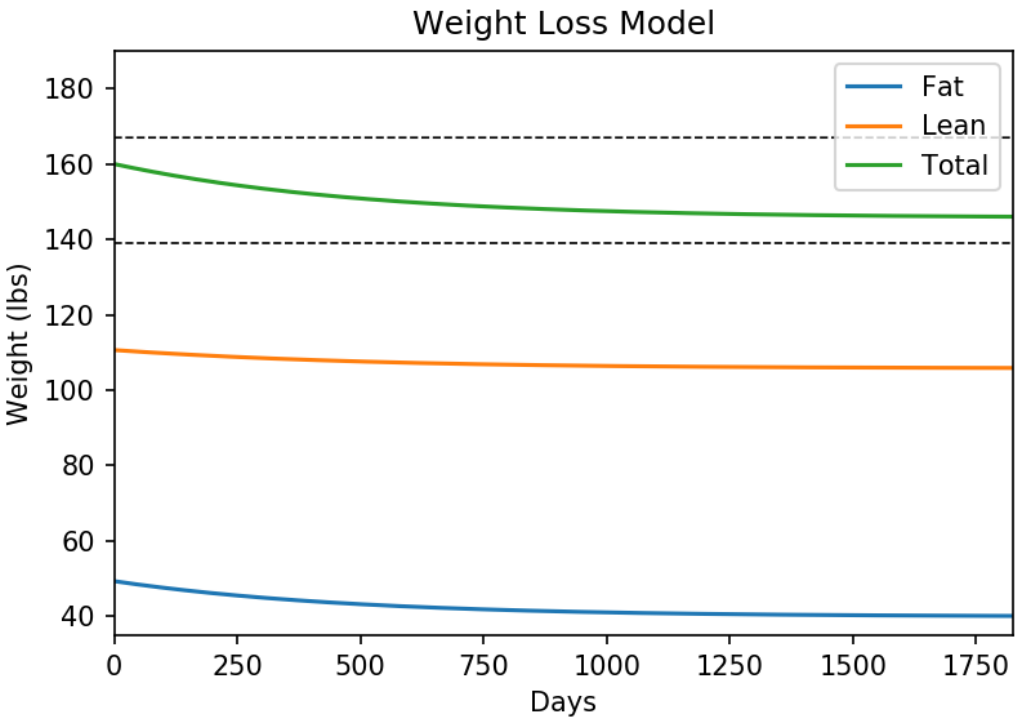


Figure 3.3: The solution of the weight change model for problem 6.

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

$$K = \left(\frac{1}{PAL} - \beta_{at}\right) EI - \gamma_F F - \gamma_L L. \tag{3.9}$$

Thus, for a subject who has maintained the same weight for a while, one can determine K by using (3.9), if they know their average caloric intake and amount of fat (assume $L = BW - F$).

Problem 5. Write the functions `forbes()` which takes as input $F(t)$ and returns Forbe's equation given in (3.5). Also write the function `energy_balance()` which takes as input $F(t)$, $L(t)$, PAL, and EI and returns the energy balance as given in (3.8). Use $\rho_F = 9400$, $\rho_L = 1800$, $\gamma_F = 3.2$, $\gamma_L = 22$, $\eta_F = 180$, $\eta_L = 230$, and $\beta_{AT} = 0.14$.

Using `forbes()` and `energy_balance()`, define the function `weight_odesystem()` which takes as input the current fat and lean weights as an array y and the current time as a float t and returns the right hand side of (3.4) as a tuple.

Problem 6. Consider the initial value problem corresponding to (3.4).

$$\begin{aligned} \frac{dF(t)}{dt} &= \frac{(1-p(t))EB(t)}{\rho_F}, \\ \frac{dL(t)}{dt} &= \frac{p(t)EB(t)}{\rho_L}, \\ F(0) &= F_0, \\ L(0) &= L_0. \end{aligned} \tag{3.10}$$

The following function returns the fat mass of an individual based on body weight (kg), age (years), height (meters), and sex. Use this function to define initial conditions F_0 and L_0 for the IVP above: $F_0 = fat\_mass(args^*)$, $L_0 = BW - F_0$.

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

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

Use (3.9) and the original intake and phyical activity levels to compute K for this system. Then use `odeint` to solve the IVP. Graph the solution curve for this single-stage weightloss intervention over a period of 5 years. Your plot should match figure 3.3.

Note the provided code requires quantities in metric units (kilograms, meters, days) while our graph is converted to units of pounds and days.

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

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

4 The Shooting Method for Boundary Value Problems

Consider a boundary value problem of the form

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \le x \le b, \\ y(a) &= \alpha, \quad y(b) = \beta. \end{aligned} \tag{4.1}$$

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

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \le x \le b, \\ y(a) &= \alpha, \quad y'(a) = t. \end{aligned} \tag{4.2}$$

> **Achtung!**
>
> Here, do not mistake t as an independent time variable. It is a parameter for the boundary condition. The prime $'$ notation represents the derivative with respect to x, which is an important distinction later in this section.

The goal is to determine an appropriate value t for the initial slope, so that the solution of the IVP is also a solution of the boundary value problem.

Let $y(x, t)$ be the solution of (4.2). We can rewrite the initial value conditions as

$$y(a, t) = \alpha, \quad y'(a, t) = t \tag{4.3}$$

We wish to find a value of t so that $y(b, t) - \beta = 0$. Applying Newton's method to the function $h(t) = y(b, t) - \beta$, we obtain the iterative method

$$\begin{aligned} t_{n+1} &= t_n - \frac{h(t_n)}{h'(t_n)}, \\ &= t_n - \frac{y(b, t_n) - \beta}{\frac{d}{dt}\, y(b, t)|_{t_n}}, \quad n = 0, 1, \ldots. \end{aligned}$$

We recall that Newton's method requires a good initial guess t_0; a plausible initial guess would be the average rate of change of the solution across the entire interval, so that $t_0 = (\beta - \alpha)/(b - a)$. If this initial guess is not sufficient, the initial guess may be refined by looking at the solution $y(x, t_0)$ of the initial value problem.

This method requires us to evaluate or approximate the function $\frac{d}{dt}\, y(b,t)|_{t_n}$. This term may be approximated with a finite difference, giving us the iterative method

$$t_{n+1} = t_n - \frac{(y(b,t_n) - \beta)(t_n - t_{n-1})}{y(b,t_n) - y(b,t_{n-1})},\ n = 1, 2, \ldots$$

This variation of the shooting algorithm is called the secant method, and requires two initial values instead of one. Notice that finding $y(b,t_n)$ requires solving the initial value problem using RK4 or some other method.

For example, consider the boundary value problem

$$\begin{aligned} y'' &= -4y - 9\sin(x),\ x \in [0, 3\pi/4], \\ y(0) &= 1, \\ y(3\pi/4) &= -\frac{1+3\sqrt{2}}{2}. \end{aligned} \tag{4.4}$$

The following code implements the secant method to solve (4.4). Notice that `odeint` is the solver used for the initial value problems.

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

# y'' +4y = -9sin(x), y(0) = 1., y(3*pi/4.) = -(1.+3*sqrt(2))/2., y'(0) = -2
# Exact Solution: y(x) = cos(2x) + (1/2)sin(2x) - 3sin(x)

def find_t(f,a,b,alpha,beta,t0,t1,maxI):
    sol1 = 0
    i = 0
    while abs(sol1-beta) > 10**-8 and i < maxI:
        sol0 = odeint(f, np.array([alpha,t0]), [a,b], atol=1e-10)[1,0]
        sol1 = odeint(f, np.array([alpha,t1]), [a,b], atol=1e-10)[1,0]
        t2 =  t1 - (sol1 - beta)*(t1-t0)/(sol1-sol0)
        t0 = t1
        t1 = t2
        i = i+1
    if i == maxI:
        print "t not found"
    return t2


def solveSecant(f,X,a,b,alpha,beta,t0,t1,maxI):
    t = find_t(f,a,b,alpha,beta,t0,t1,maxI)
    sol = odeint(f,np.array([alpha,t]), X,atol=1e-10)[:,0]
    return sol


def ode(y,x):
    return np.array([y[1], -4*y[0]-9*np.sin(x)])
```

```
X = np.linspace(0,3*np.pi/4,100)
Y = solveSecant(ode,X,0,3*np.pi/4,1,-(1.+3*np.sqrt(2)) /2,
                                        (1+(1.+3*np.sqrt(2)) /2)/(-3*np.pi/4)↩
                                            ,-1,40)

plt.plot(X,Y,'-k',linewidth=2)
plt.show()
```

secant_method.py

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. You may have to determine which solution is physically interesting. The following bvp has at least two solutions. Using the secant method, find both numerical solutions and their initial slopes. (Their plots are given in Figure 4.1.) What initial values t_0, t_1 did you use to find them?

$$\begin{aligned} y'' &= -4y - 9\sin(x), \ x \in [0, \pi], \\ y(0) &= 1, \\ y(\pi) &= 1. \end{aligned}$$

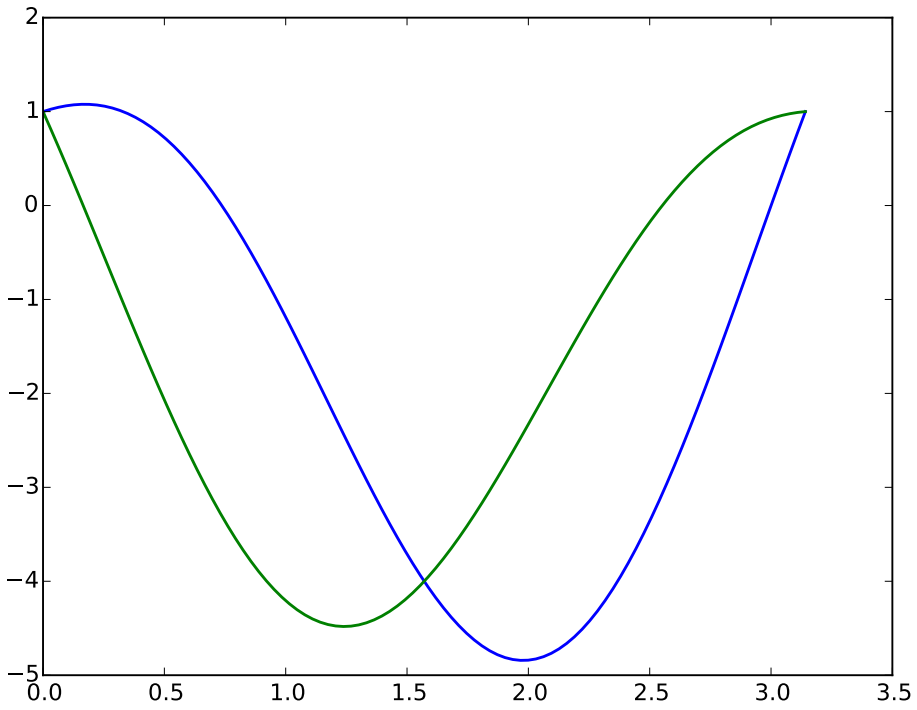


Figure 4.1: Two solutions of $y'' = -4y - 9\sin(x)$, both satisfying the boundary conditions $y(0) = y(\pi) = 1$.

Let us consider how to solve for $\frac{d}{dt}y(b,t)$. We will assume that the function $y(x,t)$ can be differentiated with respect to x and t in any order, and let $z(x,t) = \frac{d}{dt}y(x,t)$. Using the chain rule, we obtain

$$\begin{aligned} z'' = \frac{d}{dt}y''(x,t) = \frac{\partial f}{\partial y}(x, y(x,t), y'(x,t)) \cdot \frac{dy}{dt}(x,t), \\ + \frac{\partial f}{\partial y'}(x, y(x,t), y'(x,t)) \cdot \frac{dy'}{dt}(x,t), \end{aligned}$$

Using the initial conditions associated with $y(x,t)$ and noting that $z(x,t) = \frac{d}{dt}y(x,t)$ and $z'(x,t) = \frac{d}{dt}y'(x,t)$, we obtain the following initial value problem for $z(x,t)$:

$$\begin{aligned} z'' = \frac{\partial f}{\partial y}(x,y,y')z + \frac{\partial f}{\partial y'}(x,y,y')z', \ a \le x \le b, \\ z(a,t) = 0, \ z'(a,t) = 1. \end{aligned}$$

To use Newton's method, the (coupled) IVPs for y and z must be solved simultaneously. The iterative method then becomes

$$t_{n+1} = t_n - \frac{y(b,t_n) - \beta}{z(b,t_n)}, \ n = 0, 1, \ldots$$

Problem 2. Use Newton's method to solve the BVP

$$y'' = 3 + \frac{2y}{x^2},\ x \in [1, e],$$
$$y(1) = 6,$$
$$y(e) = e^2 + 6/e.$$

Plot your solution. (Compare with Figure 4.2.) What is an appropriate initial guess?

Hint: Update the ode() function from the previous problem to solve for y, y', z, z' simultaneously. This can be done by first rewriting the equations for y" and z" as a system of first order differential equations.

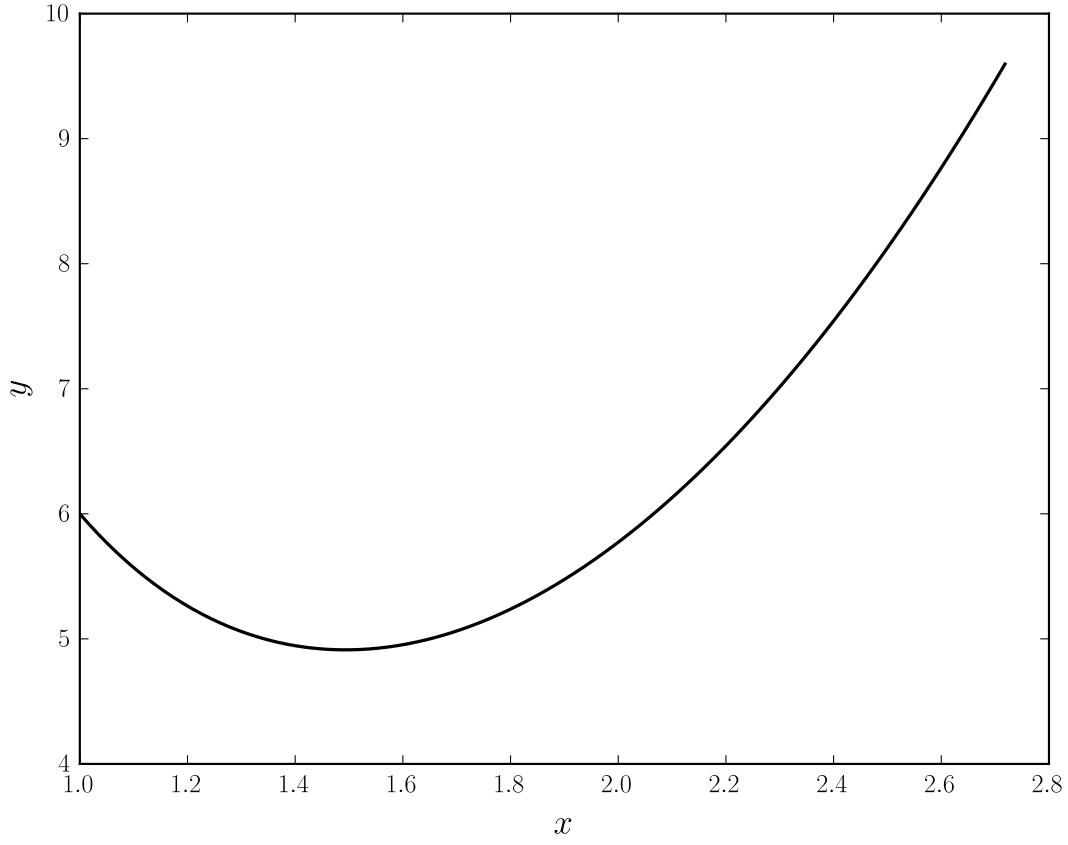


Figure 4.2: The solution of $y'' = 3 + 2y/x^2$, satisfying the boundary conditions $y(1) = 6$, $y(e) = e^2 + 6/e$.

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 the coordinates of the projectile be given by $\vec{r}(t) = \langle x(t), y(t)\rangle$. If $\theta(t)$ represents the angle of the velocity vector from the positive x-axis and $v(t)$ represents the speed of the projectile ($|\vec{v}(t)|$), then we have

$$\begin{aligned}\dot{x} &= v\cos\theta,\\ \dot{y} &= v\sin\theta.\end{aligned}$$

Note that each of x, y, θ, and v are functions of t, so the dot denotes $\dfrac{d}{dt}$. The tangent vector to the path traced by the projectile is the unit vector in the direction of the projectile's velocity, so $\vec{T}(t) = \langle\cos\theta, \sin\theta\rangle$. The unit normal vector $\vec{N}(t)$ is given by $\vec{N}(t) = \langle-\sin\theta, \cos\theta\rangle$. Thus the relationship between basis vectors $\vec{i}, \vec{j}$, and $\vec{T}(t), \vec{N}(t)$ is given by

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}\begin{bmatrix} \vec{i} \\ \vec{j} \end{bmatrix} = \begin{bmatrix} \vec{T}(t) \\ \vec{N}(t) \end{bmatrix}$$

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

$$m\dot{\vec{v}} = F_g + F_d.$$

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

$$\begin{aligned} m\dot{\vec{v}} &= -mg\vec{j} - kv^2\vec{T},\\ &= -mg(\sin\theta\vec{T} + \cos\theta\vec{N}) - kv^2\vec{T},\\ &= (-mg\sin\theta - kv^2)\vec{T} - mg\cos\theta\vec{N}. \end{aligned} \tag{4.5}$$

From the identity $\vec{v} = \langle\dot{x}, \dot{y}\rangle = \langle v\cos\theta, v\sin\theta\rangle$ we have

$$\begin{aligned} m\dot{\vec{v}} &= m\langle\dot{v}\cos\theta - v\sin\theta\cdot\dot{\theta}, \dot{v}\sin\theta + v\cos\theta\cdot\dot{\theta}\rangle\\ &= m(\dot{v}\cos\theta - v\sin\theta\cdot\dot{\theta})(\cos\theta\vec{T} - \sin\theta\vec{N})\\ &\quad + m(\dot{v}\sin\theta + v\cos\theta\cdot\dot{\theta})(\sin\theta\vec{T} + \cos\theta\vec{N}),\\ &= m(\vec{T}\cdot\dot{v} + \vec{N}\cdot v\cdot\dot{\theta}). \end{aligned} \tag{4.6}$$

From equations (4.5) and (4.6) we have

$$\begin{aligned} m\dot{v} &= -mg\sin\theta - kv^2,\\ mv\dot{\theta} &= -mg\cos\theta. \end{aligned}$$

Thus we have the coupled system of differential equations

$$\begin{aligned} \dot{x} &= v\cos\theta,\\ \dot{y} &= v\sin\theta,\\ \dot{v} &= -g\sin\theta - kv^2/m,\\ \dot{\theta} &= -g\cos\theta/v. \end{aligned}$$

The independent variable t used above is unessential to our problem. If we assume that t is an smooth invertible function of x ($t = t(x)$), then we obtain

$$\begin{aligned}\frac{dy}{dx} &= \frac{dy}{dt}\frac{dt}{dx},\\ &= \frac{dy}{dt}\frac{1}{v\cos\theta},\\ &= \frac{v\sin\theta}{v\cos\theta} = \tan\theta.\end{aligned}$$

We find $\frac{dv}{dx}$ and $\frac{d\theta}{dx}$ in a similar manner. Thus our system of differential equations becomes

$$\begin{aligned}\frac{dy}{dx} &= \tan\theta,\\ \frac{dv}{dx} &= -\frac{g\sin\theta + \mu v^2}{v\cos\theta},\\ \frac{d\theta}{dx} &= -\frac{g}{v^2},\end{aligned} \tag{4.7}$$

where $\mu = k/m$. In the next problem we will assume that the projectile has a mass of about 60 kg, so that $\mu \approx .0003$.

Problem 3. Suppose a projectile is fired from a cannon with velocity 45 m/s^2. At what angle $\theta(0)$ should it be fired to land at a distance of 195 m?

There should be two initial angles $\theta(0)$ that produce a solution for this bvp. Use the secant method to numerically compute and then plot both trajectories.

$$\begin{aligned}\frac{dy}{dx} &= \tan\theta,\\ \frac{dv}{dx} &= -\frac{g\sin\theta + \mu v^2}{v\cos\theta},\\ \frac{d\theta}{dx} &= -\frac{g}{v^2},\\ y(0) &= y(195) = 0,\\ v(0) &= 45 \text{ m/s}^2\end{aligned} \tag{4.8}$$

($g = 9.8067$ m/s^2.) Find both solutions for this boundary value problem when $\mu = .0003$. Compare with the solutions when $\mu = 0$. Their graphs are given in Figure 4.4.

Hint: This is a system of three first order differential equations, and so our secant method requires a slight modification. Keeping in mind that the unknown initial condition is $\theta(0)$, not $y'(0)$, define an appropriate function $h(t)$.

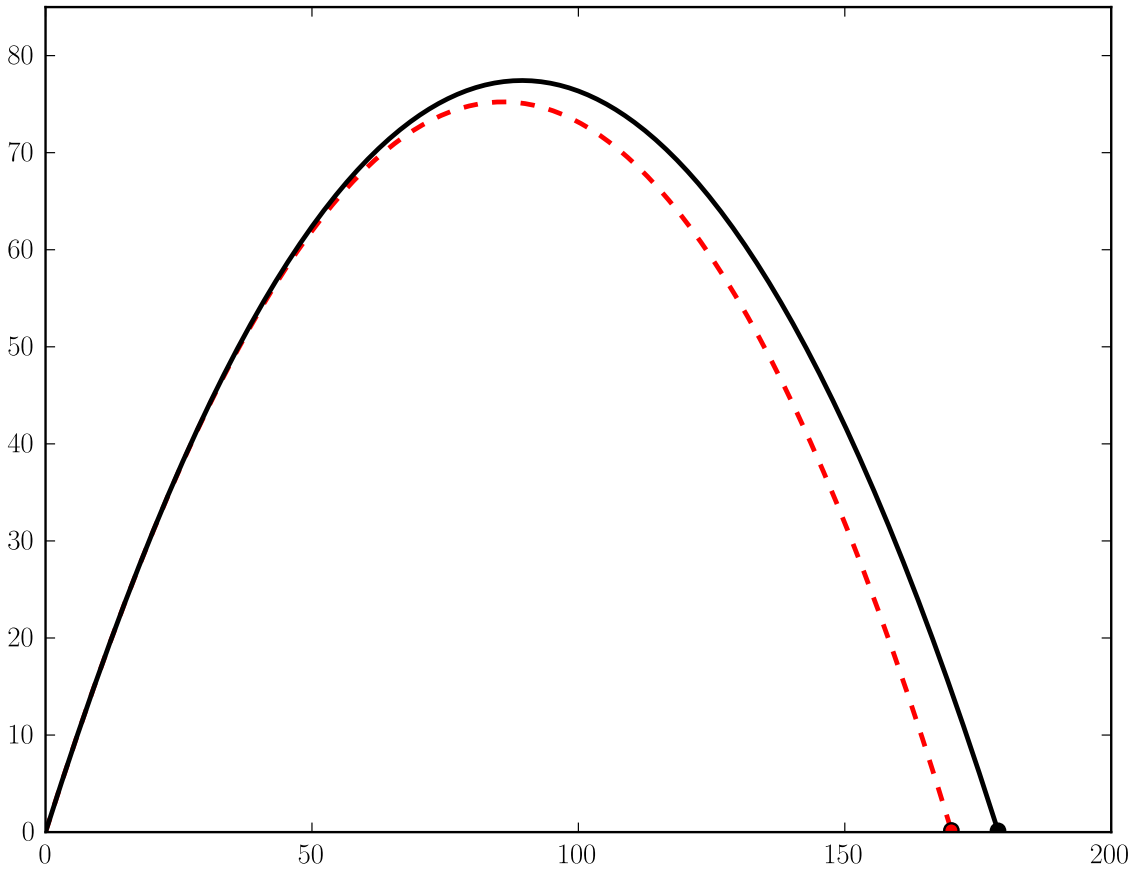


Figure 4.3: Two solutions of the system of equations (4.7), both with initial conditions $y(0) = 0$ m, $v(0) = 45$ m/s, and $\theta(0) = \pi/3$. The black curve is the trajectory of a projectile immune to air resistance ($\mu = 0$). The red curve describes the trajectory of a more realistic projectile ($\mu = .0003$).

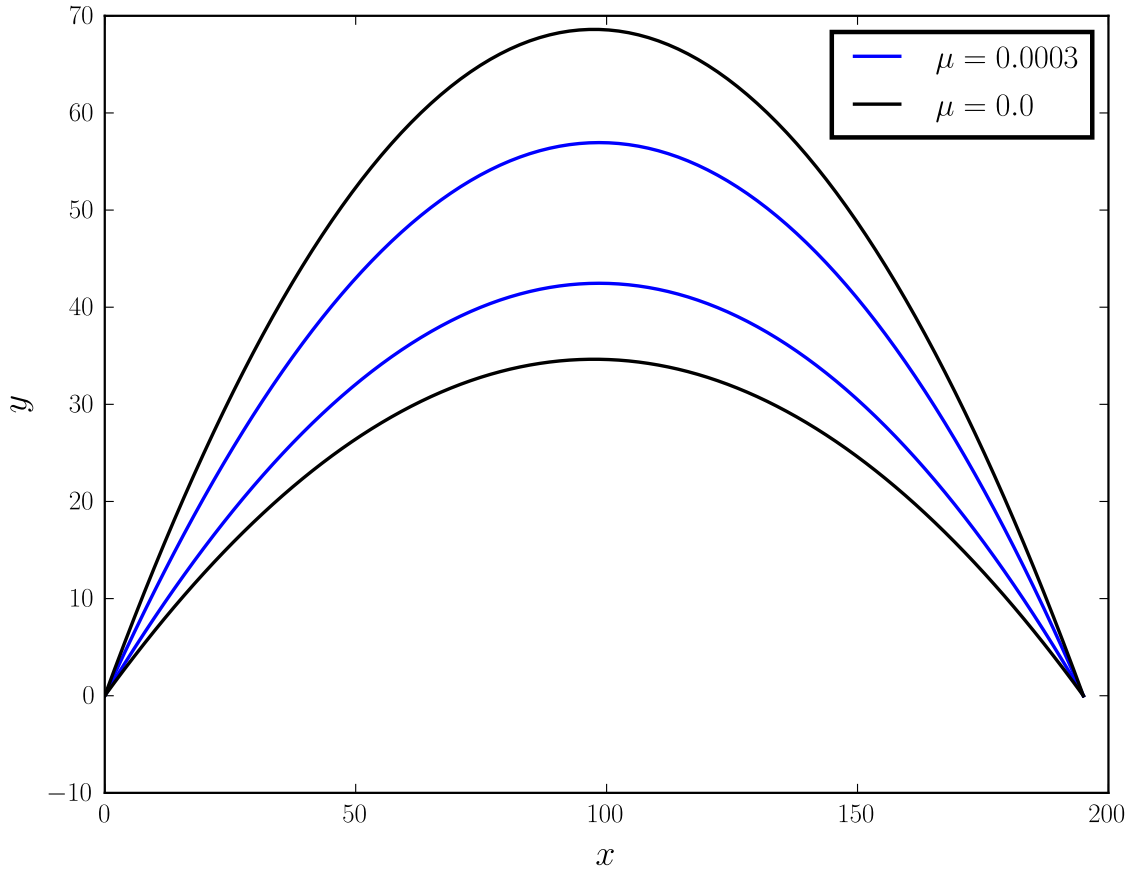


Figure 4.4: Two solutions of the boundary value problem (4.8) when the air resistance is described by the parameter $\mu = .0003$. Also both solutions when air resistance is not described in the model ($\mu = 0$).

5 Modelling the spread of an epidemic: SIR models

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 day 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

$$\begin{aligned}
\frac{dS}{dt} &= -\beta IS, \\
\frac{dI}{dt} &= \beta IS - \gamma I, \\
\frac{dR}{dt} &= \gamma I.
\end{aligned}$$

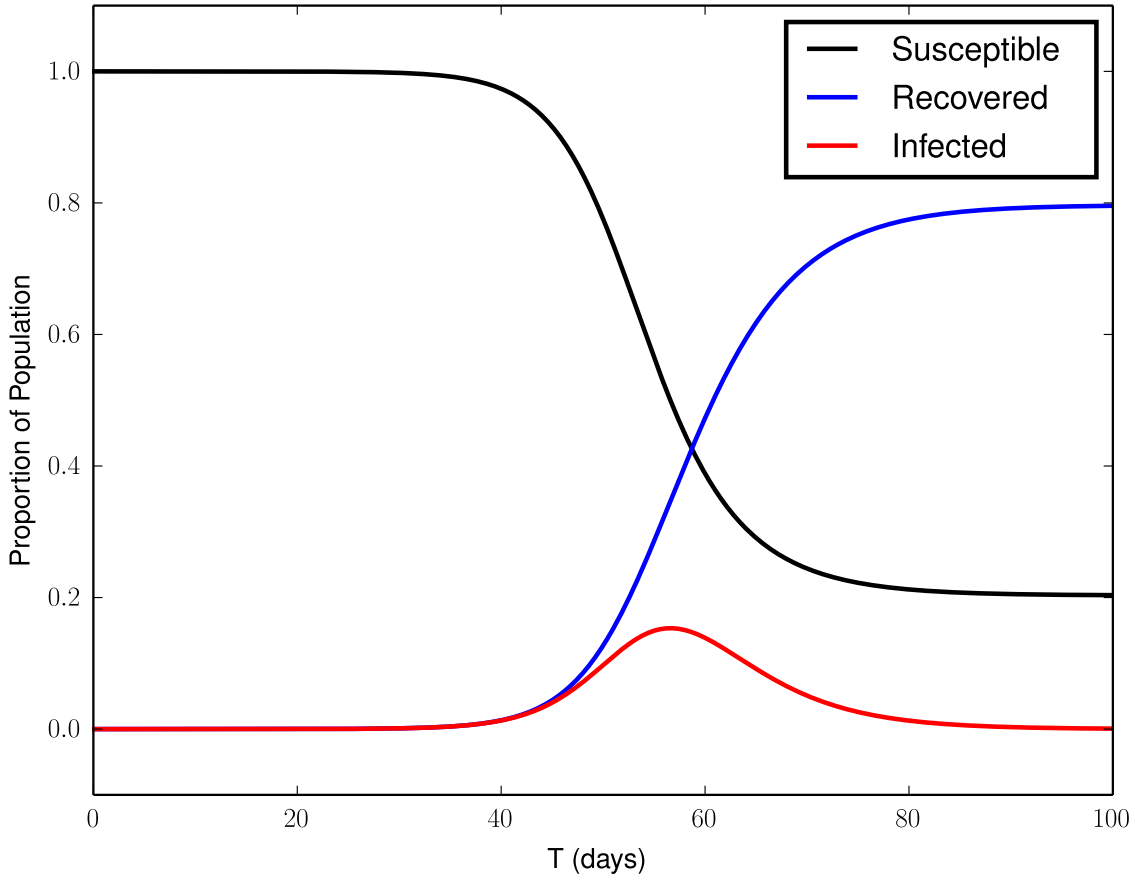


Figure 5.1: Solution to Problem (1)

Problem 1. Solve the IVP

$$\begin{aligned}
\frac{dS}{dt} &= -\frac{1}{2}IS, \\
\frac{dI}{dt} &= \frac{1}{2}IS - \frac{1}{4}I, \\
\frac{dR}{dt} &= \frac{1}{4}I, \\
S(0) &= 1 - 6.25 \cdot 10^{-7}, \\
I(0) &= 6.25 \cdot 10^{-7}, \\
R(0) &= 0,
\end{aligned}$$

on the interval $[0, 100]$, and plot your results. See Figure 5.1.

Problem 2. Suppose that, in a city of approximately three million, five have recently entered the city carrying a certain disease. (Suppose they have just entered the infectious state.)

Each of those individuals has a 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 for the next fifty days and plot your results.

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.) Print your result.

Problem 3. Suppose that, in a city of approximately three million, five have recently entered the city carrying a certain disease. (Suppose they have just entered the infectious state.)

Each of those individuals will make three contacts every ten days that could spread the disease, and an average of four days is spent in the infectious state. Find the solution of the corresponding SIR equations and plot your results. See Figure 5.2.

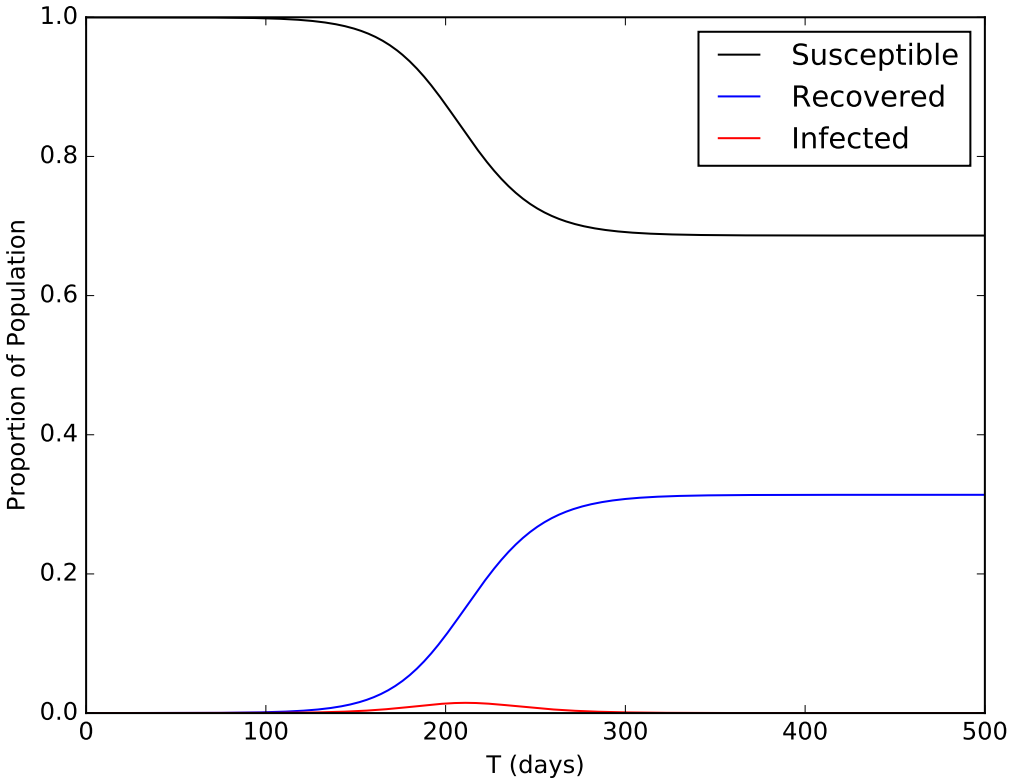


Figure 5.2: Solution to Problem (3).

Variations on 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*.

The basic equations are given by

$$\begin{aligned}\frac{dS}{dt} &= -\beta IS + \gamma I,\\ \frac{dI}{dt} &= \beta IS - \gamma I\end{aligned}$$

If we add to our basic SIR model to account for the death rate and an equal birth rate, the equations become

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

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

SIRS models take the previous model and allow the transfer of individuals from the recovered/removed class to rejoin the susceptible class.

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

Problem 4. In the world there are 7 billion people. Influenza, or the flu, is one of those viruses that everyone can be susceptible to, even after recovering from their last sickness. The flu virus is able to change in order to evade our immune system and we become susceptible once more (although technically it is now a different strain). Suppose the virus originates with 1000 people in Texas after Hurricane Harvey flooded Houston and stagnant water allowed the virus to proliferate. According to WebMD (trustworthy source, right?), once you get the virus you are contagious up to a week, and kids up to 2 weeks. For this lab, suppose you are contagious for 10 days before recovering. Also suppose that on average someone makes one contact every two days that could spread the flu. Since we can catch a new strain of the flu, suppose that a recovered individual becomes susceptible again with probability $f = 1/50$. The flu is also known to be deadly, killing hundreds of thousands every year on top of the normal death rate. To assure a steady population, let the birth rate balance out the death rate, and in particular let $\mu = .0001$.

Using the SIRS model above, plot the proportion of population that is Susceptible, Infected, and Recovered over a year span (365 days).

Boundary Value Problem

The next exercise uses a variation of the SIR model called an SEIR model to describe the spread of measles (see [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

$$\begin{bmatrix} S \\ E \\ I \end{bmatrix}' = \begin{bmatrix} \mu - \beta(t)SI \\ \beta(t)SI - E/\lambda \\ E/\lambda - I/\eta \end{bmatrix}, \tag{5.1}$$

$$S(0) = S(1), \tag{5.2}$$

$$E(0) = E(1), \tag{5.3}$$

$$I(0) = I(1) \tag{5.4}$$

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

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

$$\varepsilon y'' + yy' - y = 0, \quad y(-1) = 1, \quad y(1) = -1/3, \varepsilon = .1 \tag{5.5}$$

Look at figure 5.3 for the solution.

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

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

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

# The BVP solver expects you to pass it the boundary
# conditions as a callable function that computes the difference
# between a guess at the boundary conditions
# and the desired boundary conditions.
# When we use the BVP solver, we will tell it how many constraints
# there should be on each side of the domain so that it knows
# how many entries to expect in the tuples BCa and BCb.
```

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

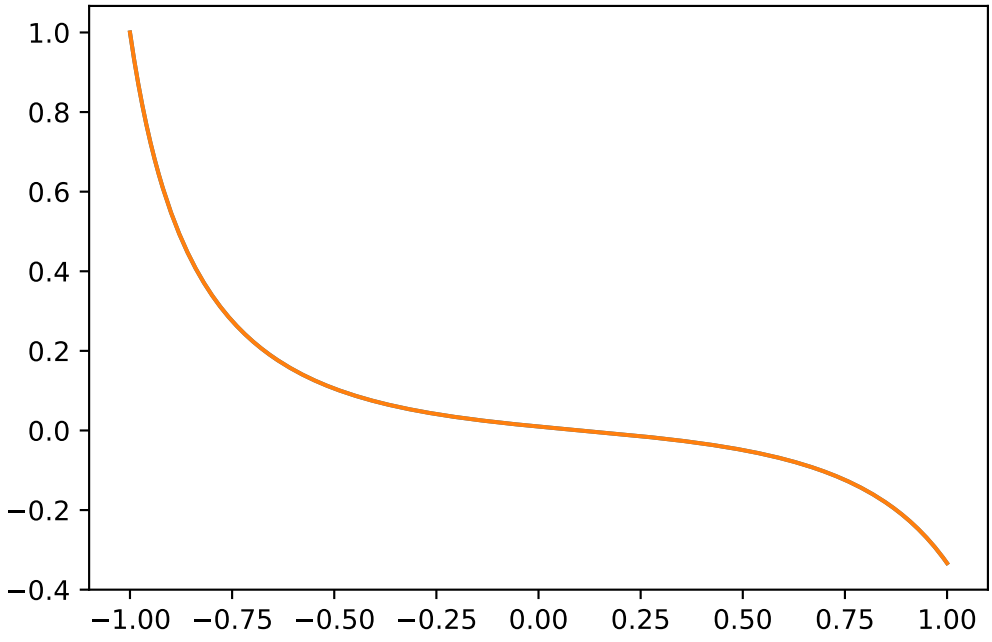


Figure 5.3: Solution to (5.5)

```
# In this case, we have one boundary condition on either side.
# These constraints are expected to evaluate to 0 when the
# boundary condition is satisfied.

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

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

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

```
y_plot = solution.sol(X)[0]

plt.plot(X, y_plot)
plt.show()
```

Problem 5. Consider (5.1) Let the periodic function for our measles case be $\beta(t) = \beta_0(1 + \beta_1 \cos 2\pi t)$. Use parameters $\beta_1 = 1$, $\beta_0 = 1575$, $\eta = 0.01$, $\lambda = .0279$, and $\mu = .02$. Note: in this case, time is measured in years, so run the solution over the interval $[0, 1]$ to show a one-year cycle. The boundary conditions are really just saying that the year will begin and end in the same state.

`solve_bvp` requires *separated boundary conditions*. In other words, each equation in the set of boundary conditions can only include values at one end of the interval. To deal with this, let $C = [C_1, C_2, C_3]$, and add the equation

$$C' = 0$$

to the system of ODEs given above (for a total of 6 equations). Then the boundary conditions can be separated using the following trick:

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

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

When you code your boundary conditions, note that `solve_bvp` changes the initial conditions to force all the entries in the return of `bcs()` to be zero. You can use the initial conditions from Fig. 5.4 as your initial guess (which will be an array of 6 elements). Remember that the initial infected proportion is small, not 0.

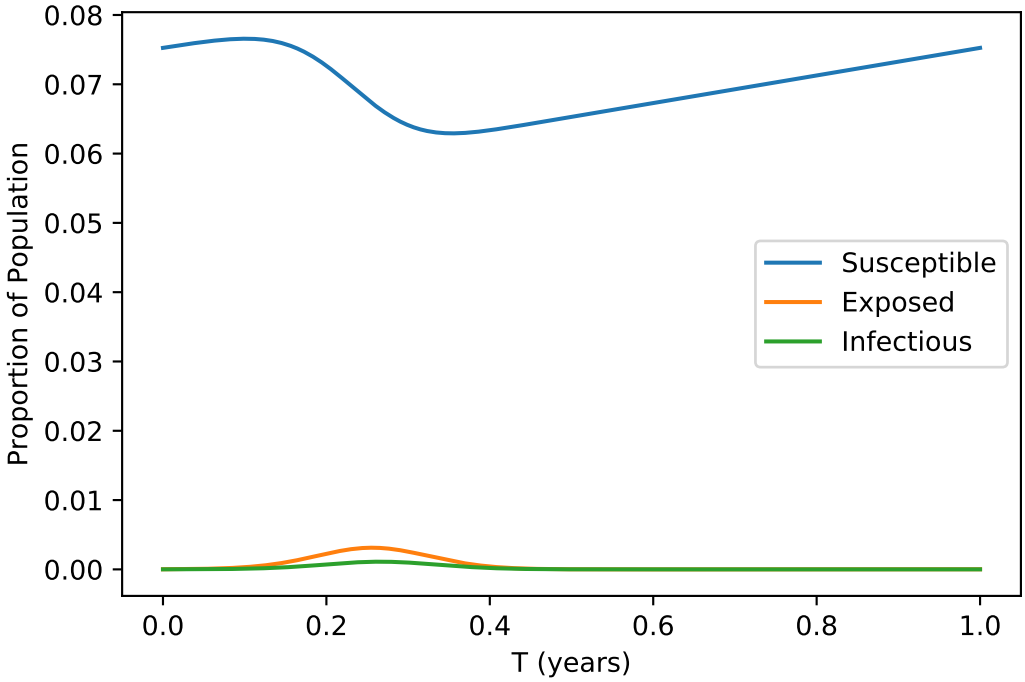


Figure 5.4: Solution to Problem (5)

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 characteristic of chaos is that small changes in initial conditions result in large changes over time in the solution curves.

The Lorenz System

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

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

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

Plotting and Interacting

We will be making interactive 3D plots using `matplotlib`. Please refer back to the *Introduction to Matplotlib* lab for tips and tricks to graphing 3D. Below is some sample code to help you out.

```
from matplotlib import rcParams, pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
rcParams['figure.figsize'] = (16,10)     #Affects output size of graphs.
'''
```

```
Code up your X, Y, Z values
'''
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot( X, Y, Z )    #Make sure X, Y, Z are same length.
                      #Connect points (X[i], Y[i], Z[i]) for i in len(X)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

ax.set_xlim3d([min(X), max(X)])    #Bounds the axes nicely
ax.set_ylim3d([min(Y), max(Y)])
ax.set_zlim3d([min(Z), max(Z)])

plt.show()
```

Because we are plotting in 3D, we are also interested in being able to interact with the plot and rotate it to analyze what our solutions are actually doing. To facilitate this, `matplotlib` has different backends which allow for various levels of control over what our graphs do. These backends are accessed using `matplotlib`'s `switch_backend()` function with a call to whichever backend we want. Your computer may default to a backend which you can already interact with (try clicking your plot and rotating it to see). If not, consider using 'qt5agg', 'qt4agg', nbagg', or 'tkagg' in the function call.

```
plt.switch_backend('qt5agg') # This backend opens the graph in a new window
```

Achtung!

Do not run `%matplotlib inline` with this code. You will crash your kernel.

Problem 1. Solve and graph the Lorenz equation by completing the code. Initialize the initial conditions with random values between -15 and 15. For this exercise, let $\sigma = 10$, $\rho = 28$, $\beta = \frac{8}{3}$. Compare to Figure 6.1

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

def lorenz_ode(inputs, T):
    '''
    Code up the sytem of equations given
    '''
    return Xprime, Yprime, Zprime

def solve_lorenx(init_cond, time=10):
    T = np.linspace(0, time, time*100)  #initialize time interval for ode
```

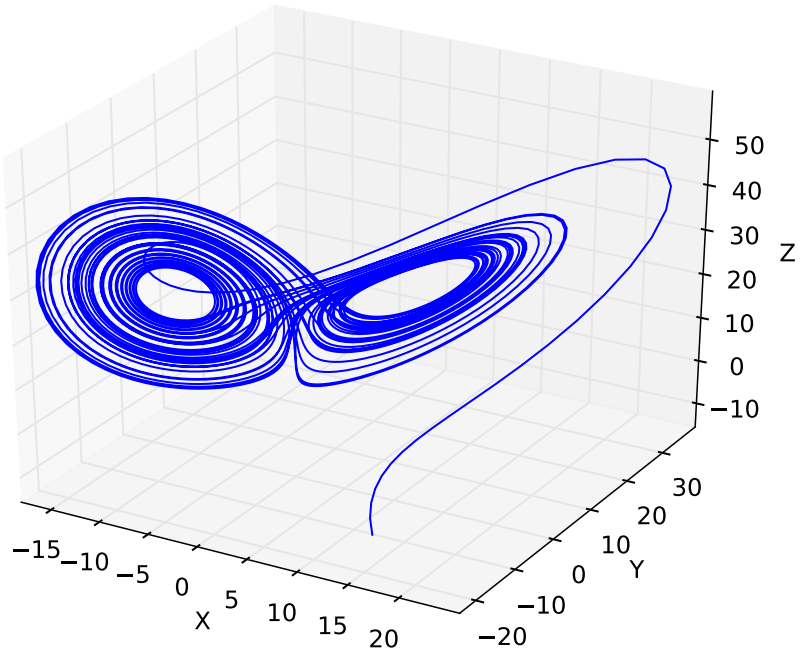


Figure 6.1: Approximate solution to the Lorenz equation with random initial conditions

```
    '''
    Use odeint in conjuction with lorenz_ode and the time interval T
    To get the X, Y, and Z values for this system.
    You will need to transpose the output of odeint to graph it correctly.
    '''
    return X, Y, Z

sigma = 'value'
rho = 'value'
beta = 'value'
init_cond = [x0, y0, z0]

X, Y, Z = solve_lorenz(init_cond, 50)
'''
Code to graph
'''
```

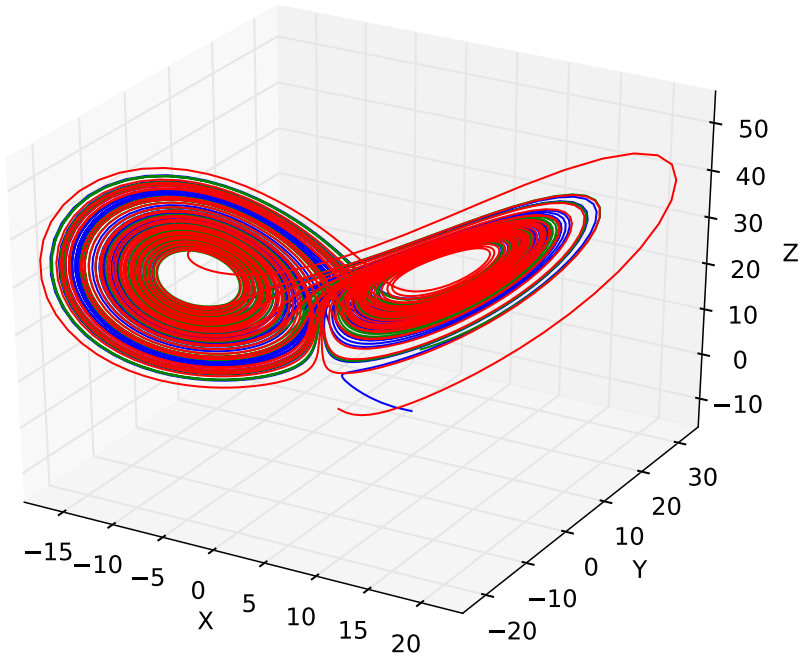


Figure 6.2: Multiple solutions to the Lorenz equation with random initial conditions

Basin of Attraction

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

Problem 2. Change your code to plot n different solution using different random initial conditions. Produce a plot with $n = 3$ different solutions. Compare to Figure 6.2

Chaos

Chaos in dynamical systems connotes a high sensitivity to initial conditions. Small differences in initial conditions can yield widely diverging outcomes for these systems. Despite having equations that completely determine the future behavior of systems, there is no way to determine where a system will be time t down the road without running through the full interval. In other words, these systems are unpredictable.

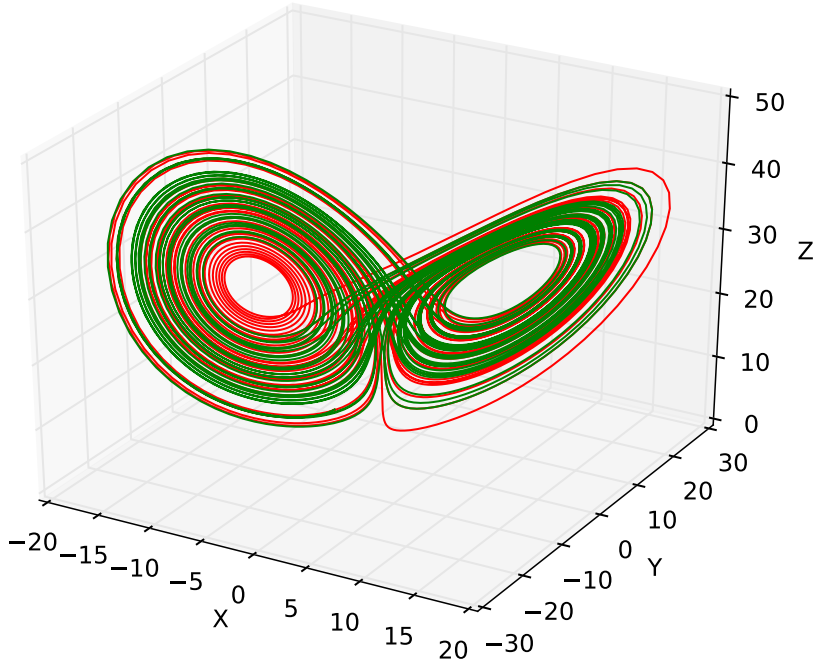


Figure 6.3: Two solutions to the Lorenz equation with perturbed initial conditions

Problem 3. Change the code above so that it initializes one set of initial conditions and creates a second set of initial conditions by adding the array `np.random.randn(3)*(1E-10)`. This will represent a small perturbation in the initial conditions. Make sure the `time` variable is large enough to notice a difference in the two solutions. Plot both solutions together. Refer to Figure 6.3

You may be wondering if your code was even correct. You could distinctly see the two curves because there was some separation of color, but who is to say that the two solutions moved away from each other only slightly as time went on. Our next task will be to *animate* this beast.

Intro to Animation

Let's get you a degree in Computer Animation. You will be provided with some code that helps with animation. Please feel free to play around and try to animate different things. Again, refer to the *Introduction to Matplotlib* lab for additional help.

We will call on `matplotlib.animate` to assist in animation. Import the following:

```
from matplotlib.animation import FuncAnimation
```

Follow these steps for setting up an animation.

- Calculate all data needed for the animation (not necessary in some cases, but it simplifies things).
- Define a figure explicitly with `plt.figure()` and set its window boundaries.
- Draw empty objects that can be altered dynamically.
- Define a function to update the drawing objects.
- make a call to `FuncAnimation()` which will start the animation.

`FuncAnimation()` accepts the figure to be animated, the function that updates the figure, the number of frames to show before repeating, and how fast to run the animation (lower number = faster).

Check out the sample code for animating two 2D objects simultaneously.

```
from matplotlib.animation import FuncAnimation

def sine_cos_animation():
    #Calculate the data to be animated
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y1, y2 = np.sin(x), np.cos(x)

    #Create a figure and set the window boundaries
    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    #Initiate empty lines of the correct dimension
    sin_drawing, = plt.plot([], [])
    cos_drawing, = plt.plot([], []) #note the comma after the variable name

    #Define a function that updates each line
    def update(index):
        sin_drawing.set_data(x[:index], y1[:index])
        cos_drawing.set_data(x[:index], y2[:index])
        return sin_drawing, cos_drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

ACHTUNG!

The above animation code works great for 2D animations, but `set_data()` is only good for 2D. When you want to animate in 3D, you will need an extra empty list in `plt.plot()` and a call to `set_3d_properties()` in the `update(index)` function.

Animate

You now know how to plot the Lorenz equation, how to plot multiple equations, and even how to animate (yay you, you're a star) simple plots. It's time to put all your know-how to good use.

Problem 4. Animate the solutions of the Lorenz equation for an initial set of conditions and the perturbed conditions on the same plot. To make the animation go faster, decrease the `interval` value in the `FuncAnimation()` call. It will take several seconds before the curves split, so be patient.

If our system is very chaotic, then even small round off errors in your computer can lead to drastically different solutions.

Problem 5. Now set one initial condition. Use `odeint` to solve the system, but use the arguments `atol=1E-14`, `rtol=1E-12`, and then again with `atol=1E-15`, `rtol=1E-13`. Animate both solutions on the same plot.

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. Usually, in dynamical systems, the sensitivity to changes in initial conditions depends exponentially 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. For the Lorenz system, experimentally it can be verified that $\lambda \approx .9$.

Suppose we want tol find the Lyapunov exponent for the Lorenz System. Start by importing `linalg` and `linregress` modules from `scipy`.

```
from scipy import linalg as la
from scipy.stats import linregress
```

To get a good estimate of the Lyapunov exponent, we want to make sure our initial solution already lies on the attractor. To do this, choose some random initial conditions, run your `solve_lorenz` function, then pick out the final coordinates. Let these coordinates be the starting point for our next system.

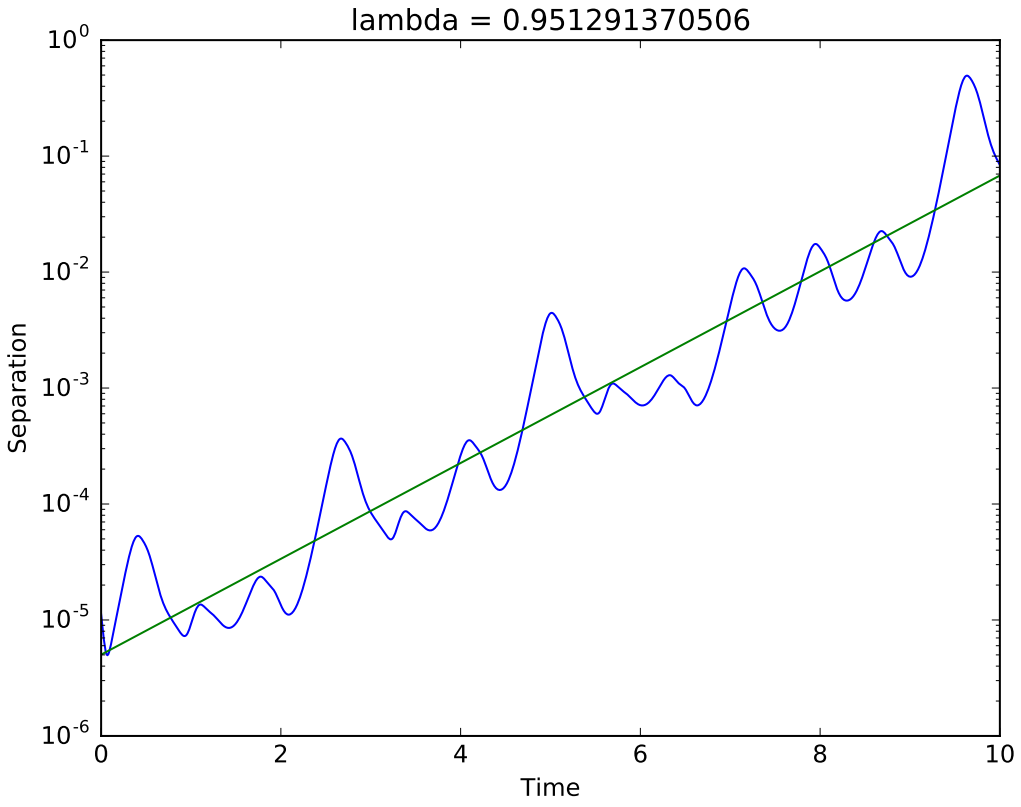


Figure 6.4: A semilog plot of the separation between two solutions to the Lorenz equations together with a fitted line that gives a rough estimate of the Lyapunov exponent of the system.

Next perturb the conditions slightly. Find the solution curve using these two sets of initial coordinates and then calculate the `norm` between the solution at each point. Plot the norm using `plt.semilogy(time, norm)`.

Finally we want to calculate the exponential line fitted to the data. Because we are using the semilog plot, we can find this line as follows:

- take the `log` of the norms.
- use `linregress()` to compute the linear regression of the log of the norms.
- take the `exp` of the linear regression to turn back into an exponential regression.
- plot the exponential regression using `plt.semilogy()` (your solution will be a line)

Problem 6. Write a Python function that finds an initial point on the strange attractor, runs the simulation to a given time t, and produces a semilog plot of the norm of the difference between the two solution curves. Also have it plot an exponential line fitted to match the curve (this will be linear on the semilog plot). Have it return a rough estimate of the Lyapunov exponent. The output should be something like Figure 6.4.

Note: In order to get a good estimate of the Lyapunov exponent, your initial guess should already lie on the strange attractor. You can get a value on the attractor by running the system for a while to find a good initial guess.

Hint: To find the fitting line, take the logarithm of the norms of the differences, compute a linear fit, then take the exponential function of the resulting line. The Lyapunov exponent will be approximately equal to the slope found by the linear regression.

7 Bifurcations and Hysteresis

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

$$\dot{x} = F(x), \quad \dot{x} := \frac{d}{dt}x(t). \tag{7.1}$$

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

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

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

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

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

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.

```
import numpy as np
```

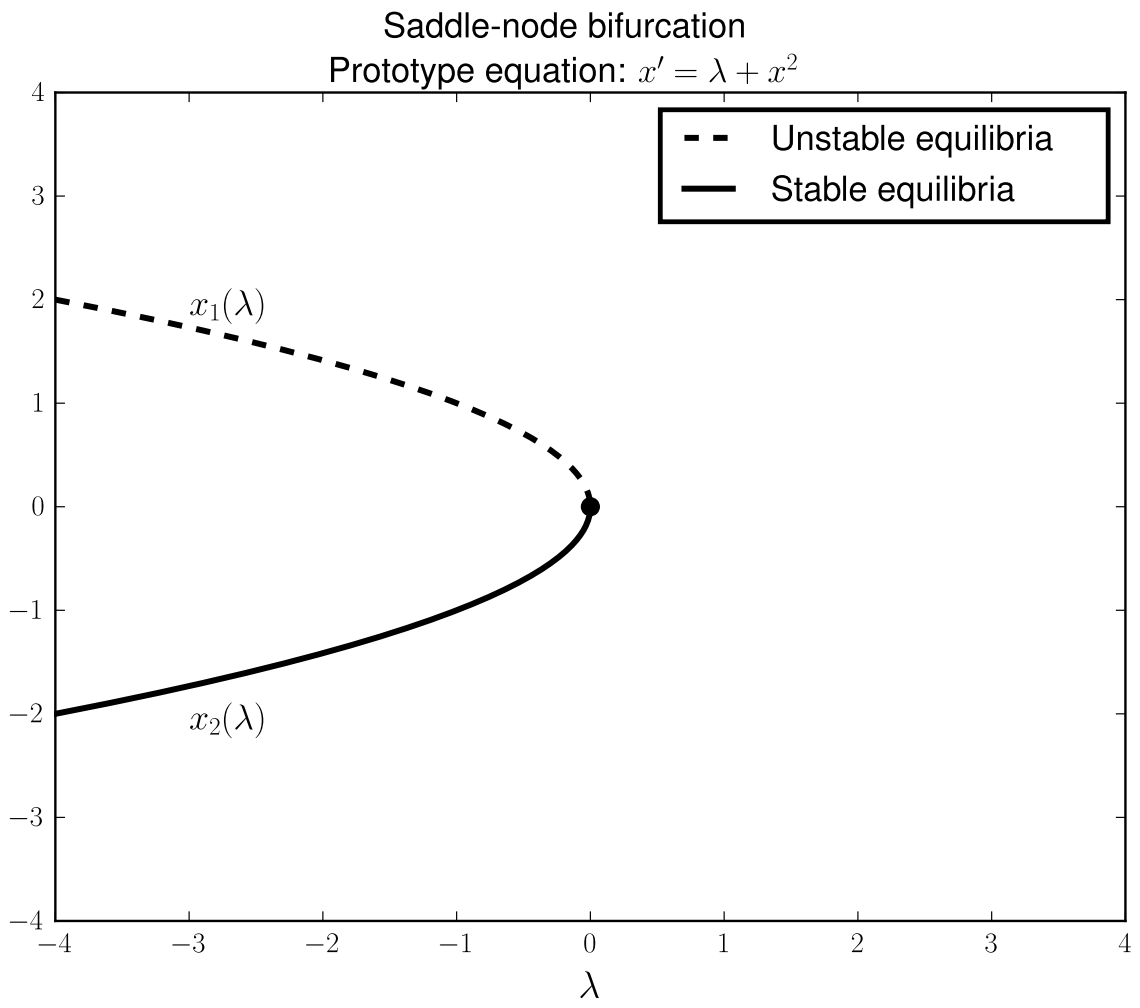


Figure 7.1: Bifurcation diagram for the equation $\dot{x} = \lambda + x^2$.

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

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

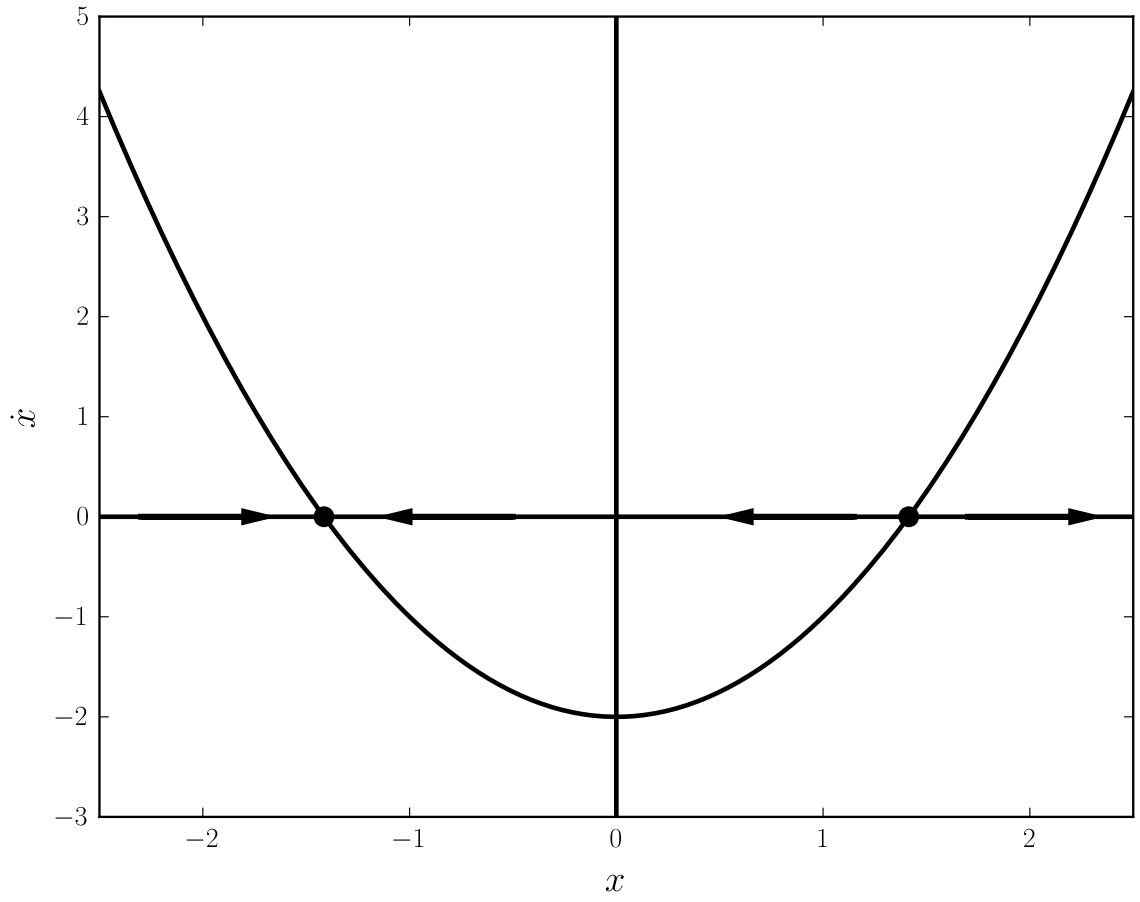


Figure 7.2: Phase Portrait for the equation $\dot{x} = -2 + x^2$.

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

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

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

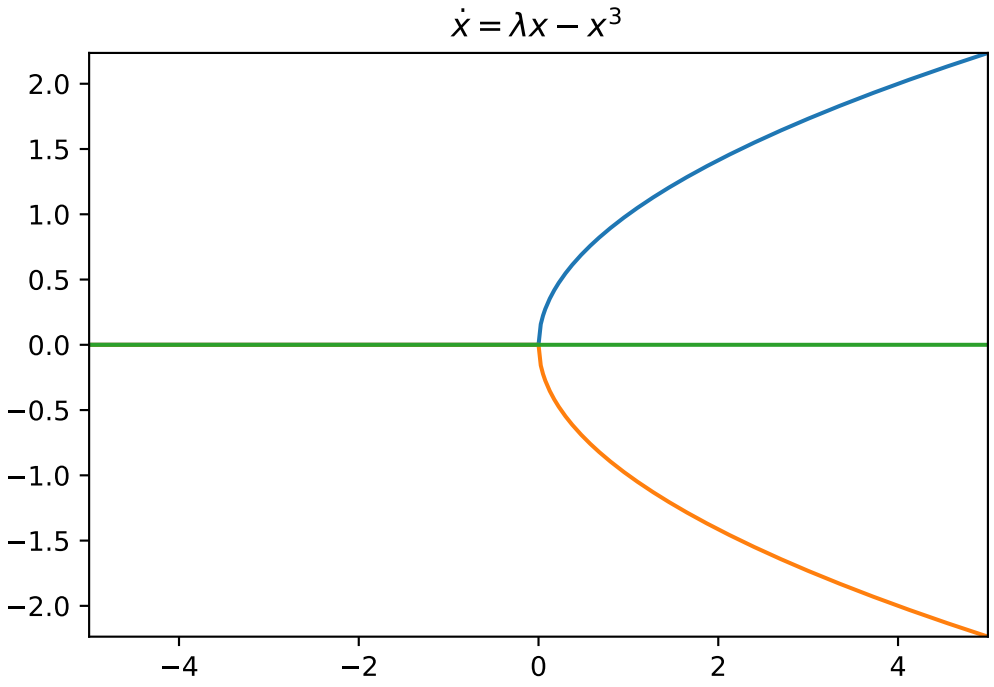


Figure 7.3: Bifurcation diagram for the equation $\dot{x} = \lambda x - x^3$.

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

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

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

Hints: Essentially this amounts to running the same code as the example, but with different parameters and function calls so that you are tracing through the right curves for this problem. To make this first problem work, you will want to have your 'linspace' run from high to low instead of from low to high. There will be three different lines in this image. See Figure 7.3.

Problem 2. Create bifurcation diagrams for the differential equation

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

where $\eta = -1, -.2, .2$ and 1. Notice that when $\eta = 0$ you can see the pitchfork bifurcation of the previous problem. There should be four different images, one for each value of η. Each image will be built of 3 pieces. See Figure 7.4.

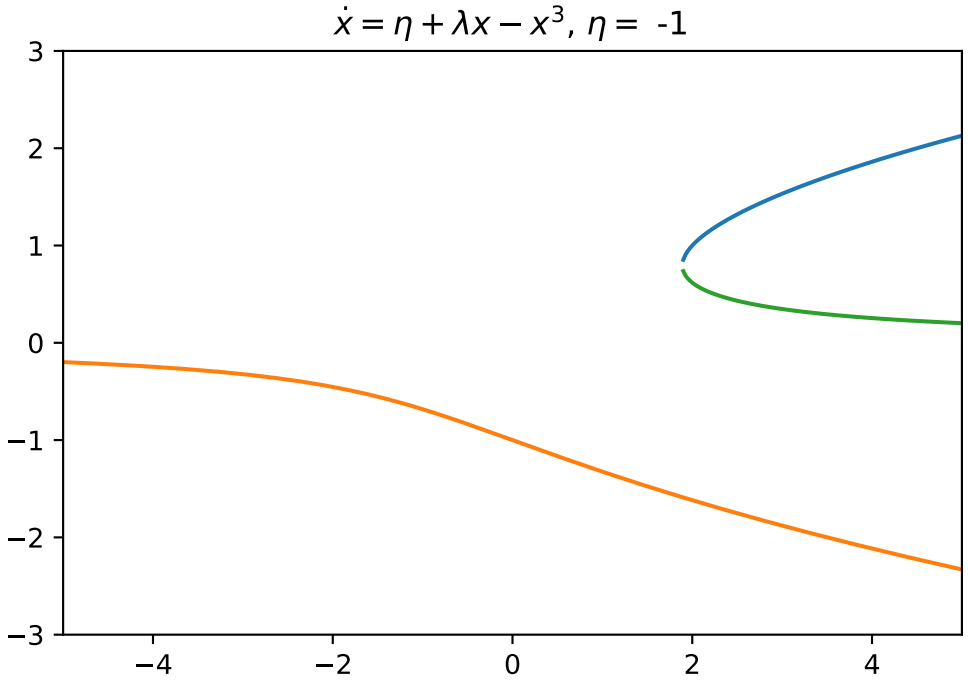


Figure 7.4: Bifurcation diagram for the equation $\dot{x} = \eta + \lambda x - x^3$ with $\eta = -1$.

Hysteresis

The following ODE exhibits an interesting bifurcation phenomenon called hysteresis:

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

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

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

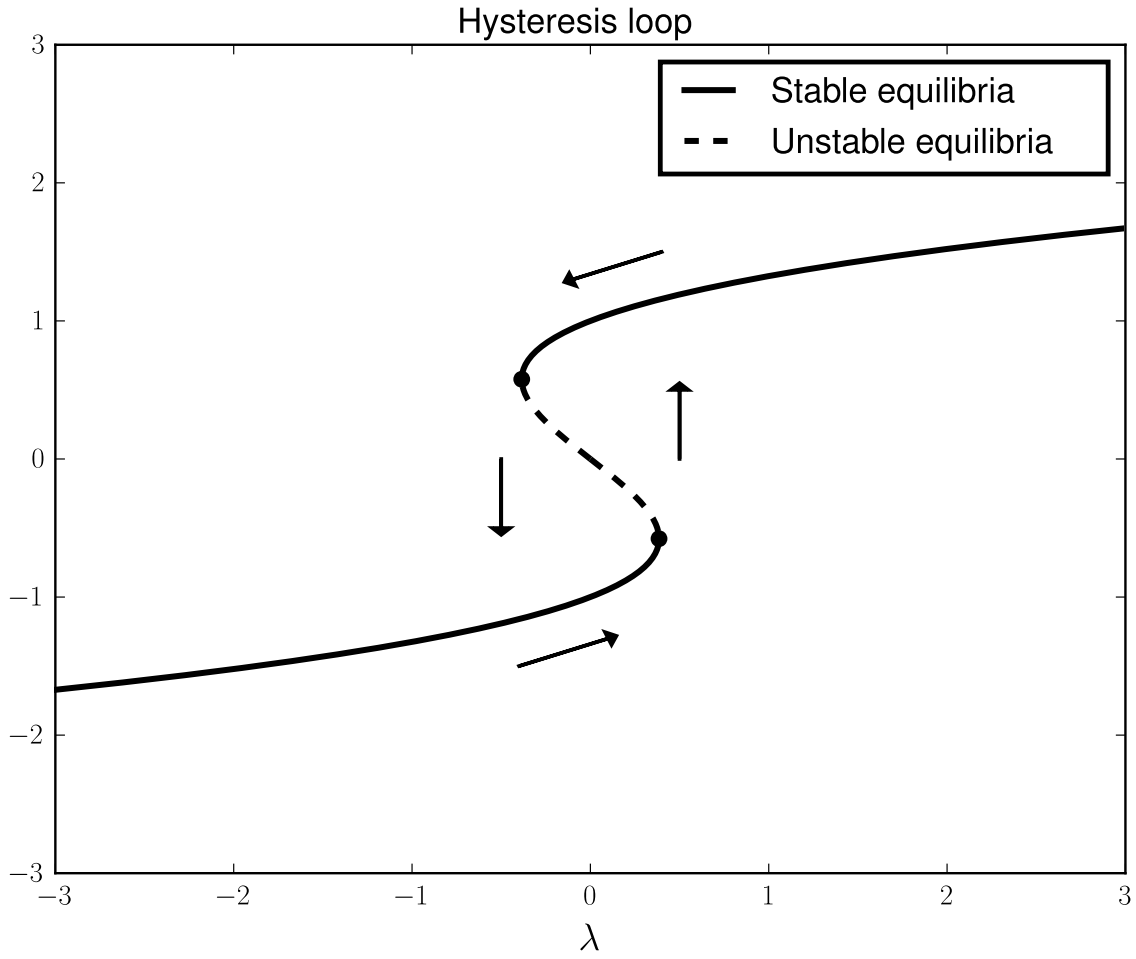


Figure 7.5: Bifurcation diagram for the ODE $x' = \lambda + x - x^3$.

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

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).

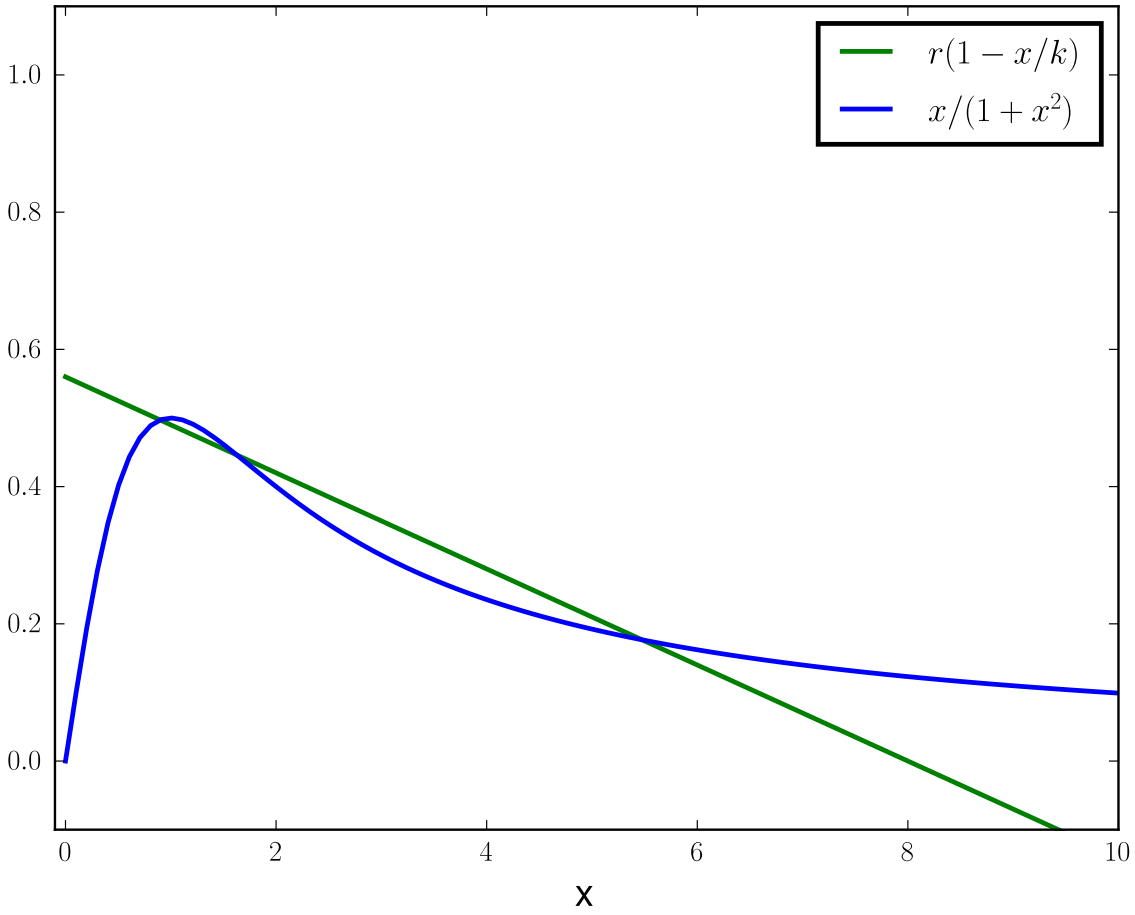


Figure 7.6: Graphical demonstration of nonzero equilibrium solutions for the budworm population (here $r = .56,\ k = 8$); equilibrium solutions occur where the curves cross. As k increases, the line $y = r(1 - x/k)$ gets more shallow and the number of solutions goes from one to three and then back to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

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

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

where $r = 0.56$.

Hint: Find a value for k that you know is in the middle of the plot (i.e. where there are three possible solutions), then use the code above to expand along each contour till you obtain the desired curve. Now find the proper initial guesses that give you the right bifurcation curve. The final plot will look like the one in Figure 7.7, but you will have to run the embedding algorithm 4-6 times to get every part of the plot.

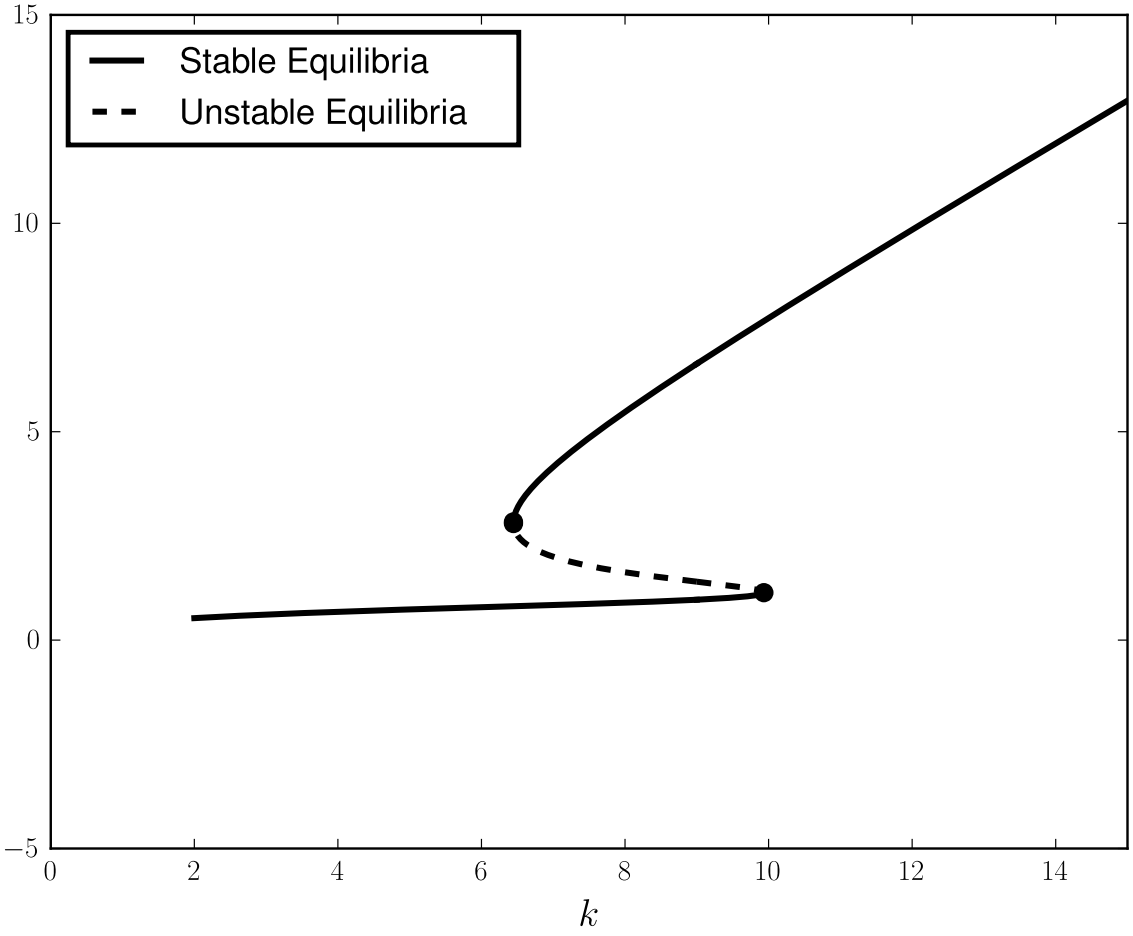


Figure 7.7: Bifurcation diagram for the budworm population model. The parameter r is fixed at 0.56. The lower stable branch is known as the refuge level of the bundworm population, while the upper stable branch is known as the outbreak level. Once the budworm population reaches an outbreak level, the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

8 The Finite Difference Method

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 $a = x_{-1}, x_0, x_1, \ldots x_{n-1} = b$ be a grid of $n+1$ evenly spaced points, with $x_i = a + (i+1)h$, $h = (b-a)/n$.

You are used to seeing the derivative $u'(x)$ which can be written as

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

Since we are interested in the derivative at certain fixed points x_i, we can consider the approximation of $u'(x)$ 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

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

From the Taylor expansion, this term has error $E(h) = \mathcal{O}(h^2)$. In terms of our grid points $\{x_i\}$, we can rewrite $u'(x)$ as $u'(x_i)$ and

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

We won't worry about the derivative at the endpoints, $u'(x_{-1})$ and $u'(x_{n-1})$. This allows us to write the set of points $\{u'(x_i)\}$ as the solution to a system of equations

$$\frac{1}{2h}\underbrace{\begin{bmatrix} -1 & 0 & 1 & & & \\ & -1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 0 & 1 \end{bmatrix}}_{(n-1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} u(x_{-1}) \\ u(x_0) \\ \vdots \\ u(x_{n-2}) \\ u(x_{n-1}) \end{bmatrix}}_{(n+1)\times 1} = \underbrace{\begin{bmatrix} u'(x_0) \\ u'(x_1) \\ \vdots \\ u'(x_{n-3}) \\ u'(x_{n-2}) \end{bmatrix}}_{(n-1)\times 1}. \tag{8.3}$$

This can be rewritten with an $(N-1)\times(N-1)$ tridiagonal matrix on the left.

$$\frac{1}{2h}\underbrace{\begin{bmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-3}) \\ u(x_{n-2}) \end{bmatrix}}_{(n-1)\times 1} + \underbrace{\begin{bmatrix} -u(x_{-1})/(2h) \\ 0 \\ \vdots \\ 0 \\ u(x_{n-1})/(2h) \end{bmatrix}}_{(n-1)\times 1} = \underbrace{\begin{bmatrix} u'(x_0) \\ u'(x_1) \\ \vdots \\ u'(x_{n-3}) \\ u'(x_{n-2}) \end{bmatrix}}_{(n-1)\times 1}. \tag{8.4}$$

Next we will consider the matrix representation for $u''(x)$. If we let

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

then

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

with error $E(h) = \mathcal{O}(h^3)$. You can achieve the same result by again consider the Taylor polynomial expansion and adding (8.1) and (8.2) and rearranging. Thus

$$u''(x_i) = \frac{u(x_i+h) - 2u(x_i) + u(x_i-h)}{h^2} = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2}, \quad i = 0, \ldots, n-2.$$

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

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \end{bmatrix}}_{(n-1)\times(n+1)} \cdot \underbrace{\begin{bmatrix} u(x_{-1}) \\ u(x_0) \\ \vdots \\ u(x_{n-2}) \\ u(x_{n-1}) \end{bmatrix}}_{(n+1)\times 1} = \underbrace{\begin{bmatrix} u''(x_0) \\ u''(x_1) \\ \vdots \\ u''(x_{n-3}) \\ u''(x_{n-2}) \end{bmatrix}}_{(n-1)\times 1}. \tag{8.5}$$

This can also be written as an $(N-1)\times(N-1)$ tridiagonal matrix on the left.

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}}_{(n-1)\times(n-1)} \cdot \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-3}) \\ u(x_{n-2}) \end{bmatrix}}_{(n-1)\times 1} + \underbrace{\begin{bmatrix} u(x_{-1})/h^2 \\ 0 \\ \vdots \\ 0 \\ u(x_{n-1})/h^2 \end{bmatrix}}_{(n-1)\times 1} = \underbrace{\begin{bmatrix} u''(x_0) \\ u''(x_1) \\ \vdots \\ u''(x_{n-3}) \\ u''(x_{n-2}) \end{bmatrix}}_{(n-1)\times 1} \tag{8.6}$$

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.

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 at the grid points?

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 simpler 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 $\{U_i\}_{i=-1}^{N-1}$, where

$$U_i = u(x_i)$$

on an evenly spaced grid of N subintervals, $a = x_{-1}, x_0, \ldots, x_{N-1} = b$ with $h = x_{i+1} - x_i$ for each i. Our finite difference method will replace the differential operator $D = \varepsilon \frac{d^2}{dx^2} - \frac{d}{dx}$, defined on an infinite-dimensional space of functions, with difference operators defined on a finite vector space (the space of grid functions $\{U_i\}_{i=-1}^{N-1}$). 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

$$\frac{\varepsilon}{h^2} (U_{i+1} - 2U_i + U_{i-1}) - \frac{1}{2h} \left(U_{i+1} - U_{i-1} \right) = f(x_i), \quad i = 0, \ldots, N-2,$$

along with boundary conditions $U_{-1} = \alpha$, $U_{N-1} = \beta$.

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

$$\frac{1}{h^2} \underbrace{\begin{bmatrix} h^2 & 0 & 0 & \ldots & 0 \\ (\varepsilon + h/2) & -2\varepsilon & (\varepsilon - h/2) & \ldots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \ldots & (\varepsilon + h/2) & -2\varepsilon & (\varepsilon - h/2) \\ 0 & \ldots & & 0 & h^2 \end{bmatrix}}_{(N+1)\times(N+1)} \cdot \underbrace{\begin{bmatrix} U_{-1} \\ U_0 \\ \vdots \\ U_{N-2} \\ U_{N-1} \end{bmatrix}}_{(N+1)\times 1} = \underbrace{\begin{bmatrix} U_{-1} \\ f(x_0) \\ \vdots \\ f(x_{N-2}) \\ U_{N-1} \end{bmatrix}}_{(N+1)\times 1}.$$

We can further modify the system to obtain an $(N-1)\times(N-1)$ tridiagonal matrix on the left:

$$\frac{1}{h^2}\underbrace{\begin{bmatrix} -2\varepsilon & (\varepsilon - h/2) & 0 & \dots & 0 \\ (\varepsilon + h/2) & -2\varepsilon & (\varepsilon - h/2) & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & (\varepsilon + h/2) & -2\varepsilon & (\varepsilon - h/2) \\ 0 & \dots & & (\varepsilon + h/2) & -2\varepsilon \end{bmatrix}}_{(N-1)\times(N-1)} \cdot \underbrace{\begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_{N-3} \\ U_{N-2} \end{bmatrix}}_{(N-1)\times 1} = \underbrace{\begin{bmatrix} f(x_0) - U_{-1}(\varepsilon + h/2)/h^2 \\ f(x_1) \\ \vdots \\ f(x_{N-3}) \\ f(x_{N-2}) - U_{n-1}(\varepsilon - h/2)/h^2 \end{bmatrix}}_{(N-1)\times 1}. \tag{8.9}$$

Problem 2. Use equation (8.9) to solve the singularly perturbed BVP (8.7) with $\varepsilon = 1/10$, $f(x) = -1$, $\alpha = 1$, and $\beta = 3$. 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.

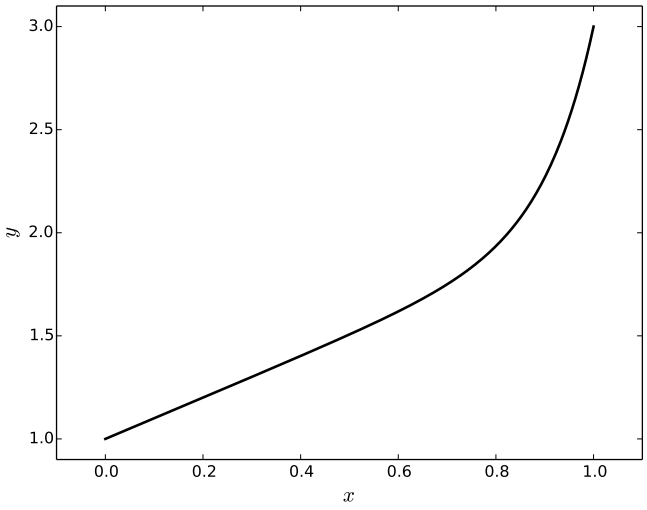


Figure 8.1: The solution to Problem 2. The solution gets steeper near $x = 1$ as ε gets small.

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?

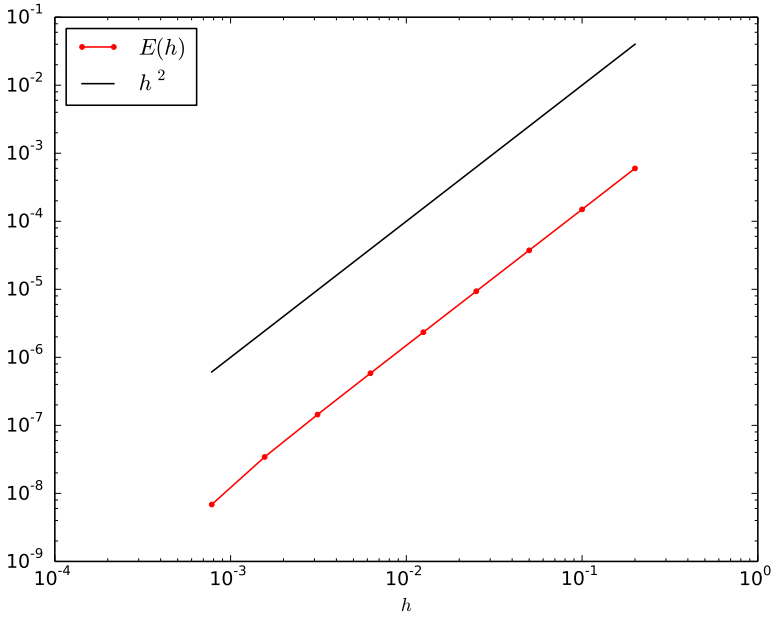


Figure 8.2: Demonstration of second order convergence for the finite difference approximation (8.8) of the BVP given in (8.7) with $\varepsilon = .5$.

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

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

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

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. We should note that demonstrating second-order convergence does NOT imply that the numerical approximation is converging to the correct solution.

Problem 3. Return to problem 2. How many subintervals are needed to obtain 4 digits of accuracy?

This is a question about the convergence of your solution. The following code generates the log-log plot in Figure 8.2, and demonstrates second-order convergence for our finite difference approximation of (8.7). Use this code to determine what h (and hence what N) is needed for the error to be less than 10^{-4}. You don't need to return the value of h, but make sure you understand by looking at the plot.

NOTE: The function `bvp` is not provided; you need to use your code from problem 2 to define it. Make sure your function is compatible with the code below. It must take 5 parameters as input and return the solution.

```
num_approx = 10 # Number of Approximations
N = 5*np.array([2**j for j in range(num_approx)])
h, max_error = (1.-0)/N[:-1], np.ones(num_approx-1)

# Best numerical solution, used to approximate the true solution.
# bvp returns the grid, and the grid function, approximating the solution
# with N subintervals of equal length.
num_sol_best = bvp(lambda x:-1, epsilon=.1, alpha=1, beta=3, N=N[-1])
for j in range(len(N)-1):
    num_sol = bvp(lambda x:-1, epsilon=.1, alpha=1, beta=3, N=N[j])
    max_error[j] = np.max(np.abs(num_sol-num_sol_best[::2**(num_approx-j-1)]))
plt.loglog(h,max_error,'.-r',label="$E(h)$")
plt.loglog(h,h**(2.),'-k',label="$h^{\, 2}$")
plt.xlabel("$h$")
plt.legend(loc='best')
plt.show()
print("The order of the finite difference approximation is about ",
    (np.log(max_error[0])-np.log(max_error[-1]))/(np.log(h[0])-np.log(h[-1])),
    ".")
```

Problem 4. Extend your finite difference code to the case of a general second order linear BVP with boundary conditions (These boundary conditions are sometimes called Dirichlet conditions):

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

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 $\varepsilon = 0.1$. Be sure to modify the finite difference operator D_h in (8.8) correctly.

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

Problem 5. Numerically solve the boundary value problem

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

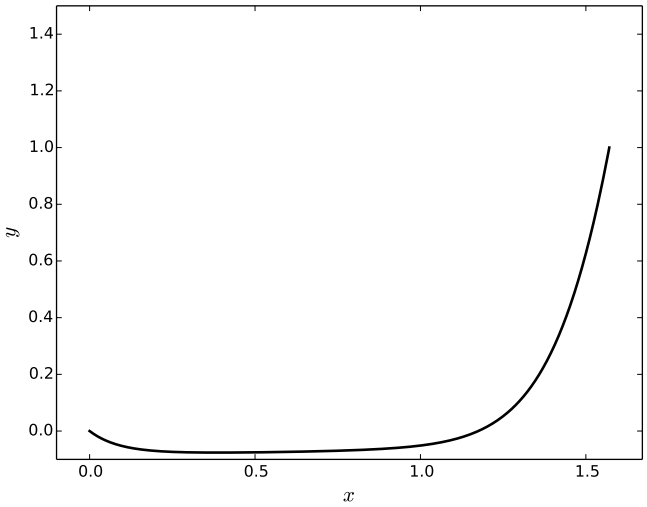


Figure 8.3: The solution to Problem 4.

for $\varepsilon = 0.1, 0.01$, and 0.001.

Problem 6. Numerically solve the boundary value problem

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

for $\varepsilon = 0.05, 0.02$.

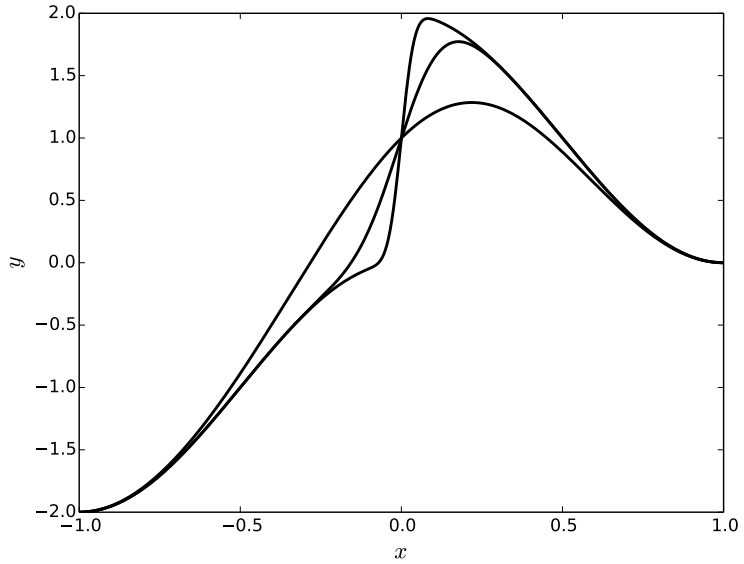


Figure 8.4: The solution to Problem 5.

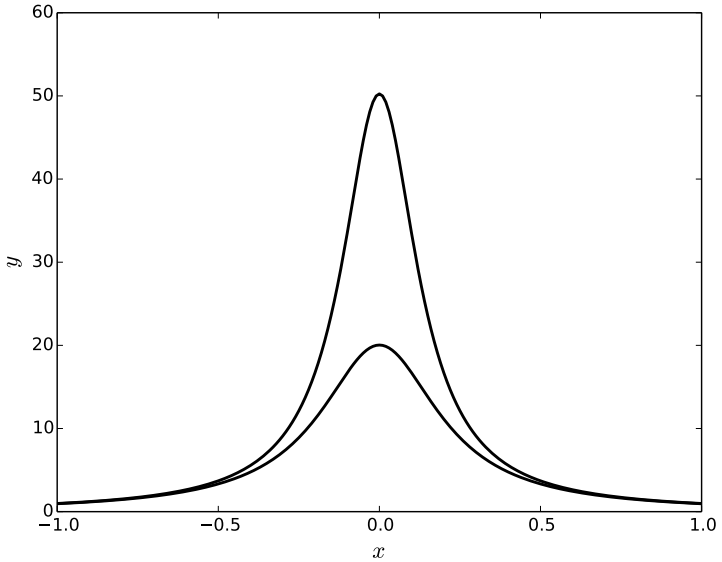


Figure 8.5: The solution to Problem 6.

9 Conservation laws and heat flow

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

Derivation of the Conservation equation in multiple dimensions

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

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

and the rate at which the quantity enters V is

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

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

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

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

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

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

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

and

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

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

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

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

Constitutive Relations

Currently our conservation law appears in the form

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

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

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

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

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

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

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

Numerically modeling heat flow

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

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

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

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

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

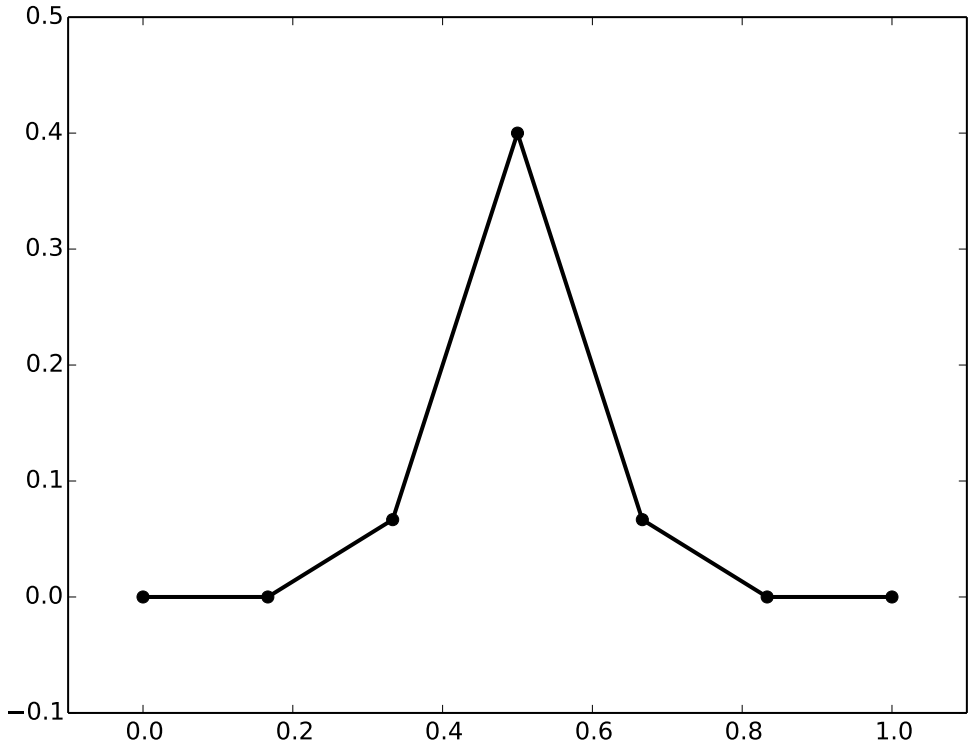


Figure 9.1: The graph of U^0, the approximation to the solution $u(x, t = 0)$ for Problem 1.

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

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

Similarly, by adding the equations

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

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

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

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

$$\begin{aligned} \frac{U_i^{j+1} - U_i^j}{k} &= \nu \frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2}, \\ U_i^{j+1} &= U_i^j + \frac{\nu k}{h^2}(U_{i+1}^j - 2U_i^j + U_{i-1}^j). \end{aligned} \tag{9.1}$$

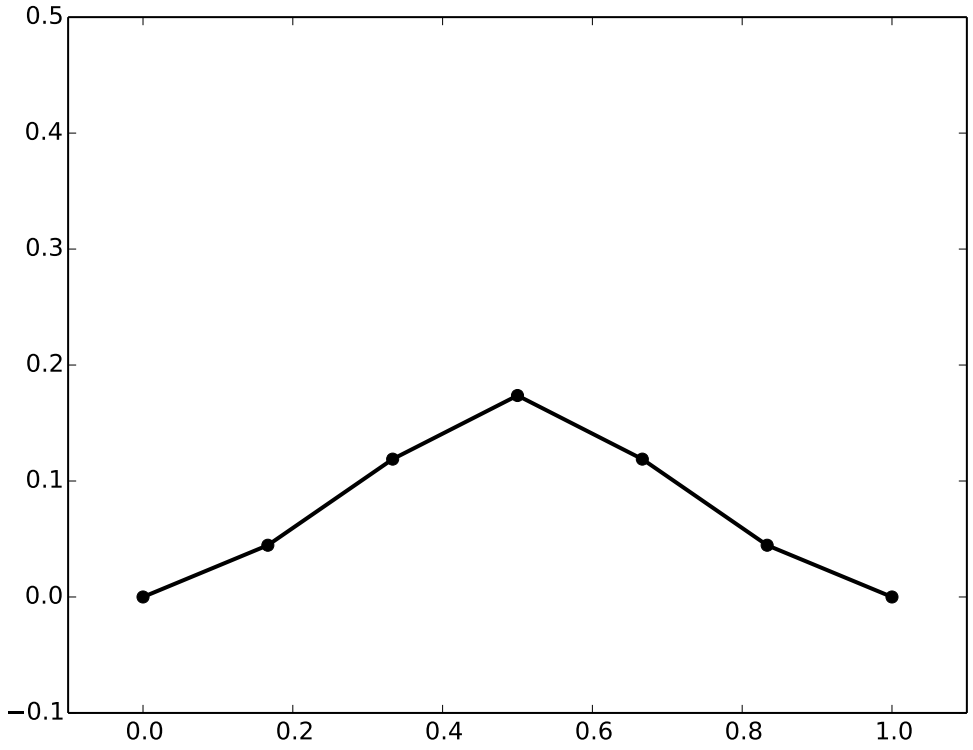


Figure 9.2: The graph of U^4, the approximation to the solution $u(x, t = .4)$ for Problem 1.

This method can be written in matrix form as

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

where A is the tridiagonal matrix given by

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

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

NOTE

Finite difference schemes, though they can be *represented* using matrix multiplication, should not be *implemented* using raw matrix multiplication. Using NumPy, it is best to vectorize the difference scheme so that you do not have to loop over the spatial indices. If you are using a language with faster loops (like C, C++, Fortran, or Cython), it could work well to loop directly through the indices in both time and space.

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

Problem 1. Consider the specific initial boundary value problem

$$\begin{aligned} &u_t = .05u_{xx}, \quad x \in [0,1], \quad t \in [0,1] \\ &u(0,t) = 0, \quad u(1,t) = 0, \\ &u(x,0) = 2\max\{.2 - |x - .5|, 0\}. \end{aligned} \tag{9.2}$$

Approximate the solution $u(x,t)$ at time $t = .4$ by taking 6 subintervals in the x dimension and 10 subintervals in time. The graphs for U^0 and U^4 are given in Figures 9.1 and 9.2.

For the next problem, we need to show how Matplotlib can be used to create a 2D animation. The following is a simple working example that animates a sine wave.

```
import numpy as np
from matplotlib import animation, pyplot as plt

def sine_animation(res=100):
    # Make the x and y data.
    x = np.linspace(-1, 1, res+1)[:-1]
    y = np.sin(np.pi * x)
    # Initialize a matplotlib figure.
    f = plt.figure()
    # Set the x and y axes by constructing an axes object.
    plt.axes(xlim=(-1,1), ylim=(-1,1))
    # Plot an empty line to use in the animation.
    # Notice that we are unpacking a tuple of length 1.
    line, = plt.plot([], [])
    # Define an animation function that will update the line to
    # reflect the desired data for the i'th frame.
    def animate(i):
        # Set the data for updated version of the line.
        line.set_data(x, np.roll(y, i))
        # Notice that this returns a tuple of length 1.
        return line,
    # Create the animation object.
    # 'frames' is the number of frames before the animation should repeat.
    # 'interval' is the amount of time to wait before updating the plot.
```

```
    # Be sure to assign the animation a name so that Python does not
    # immediately garbage collect (delete) the object.
    a = animation.FuncAnimation(f, animate, frames=y.size, interval=20)
    # Show the animation.
    plt.show()

# Run the animation function we just defined.
sine_animation()
```

Problem 2. Solve the specific initial boundary value problem

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12, 12], \quad t \in [0, 1], \\ &u(-12, t) = 0, \quad u(12, t) = 0, \\ &u(x, 0) = \max\{1 - x^2, 0\} \end{aligned} \tag{9.3}$$

using the first order explicit method 9.1. Use 140 subintervals in the x dimension and 70 subintervals in time. The initial and final states are shown in Figure 9.3. Animate your results.

Explicit methods usually have a stability condition, called a CFL condition (for Courant-Friedrichs-Lewy). For method 9.1 the CFL condition that must be satisfied is that

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

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

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

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

The first equation is a Finite Difference approximation, and the second is a midpoint approximation. These approximations give the method

$$\begin{aligned} \frac{U_i^{j+1} - U_i^j}{k} &= \frac{1}{2}\left(\frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2} + \frac{U_{i+1}^{j+1} - 2U_i^{j+1} + U_{i-1}^{j+1}}{h^2}\right), \\ U_i^{j+1} &= U_i^j + \frac{k}{2h^2}\left(U_{i+1}^j - 2U_i^j + U_{i-1}^j + U_{i+1}^{j+1} - 2U_i^{j+1} + U_{i-1}^{j+1}\right). \end{aligned} \tag{9.4}$$

This method can be written in matrix form as

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

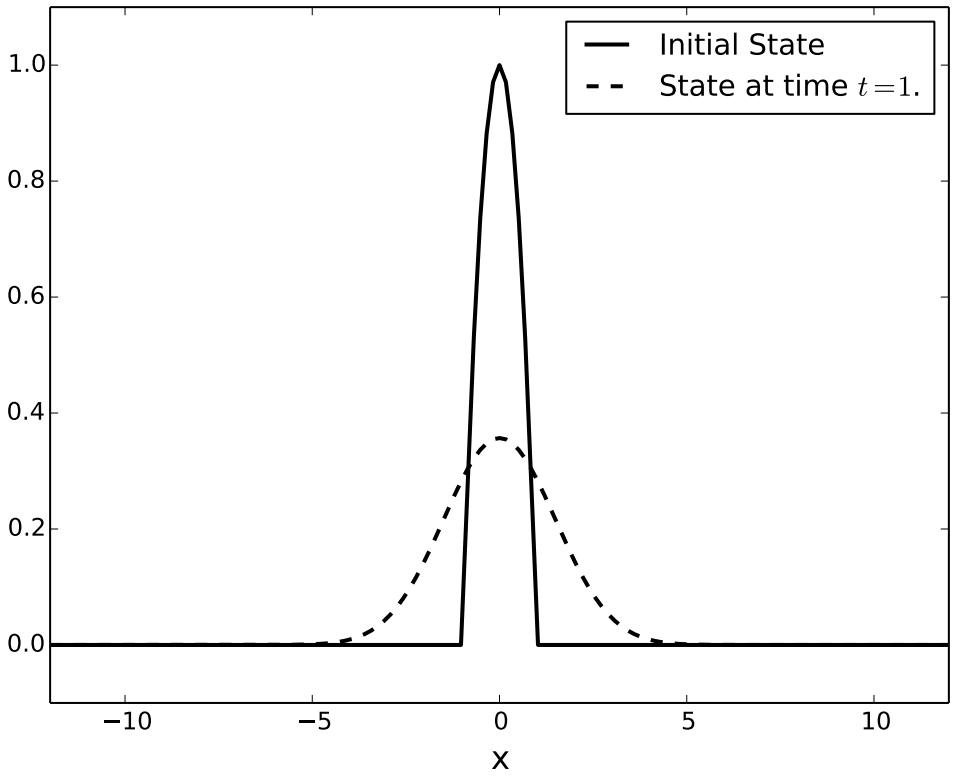


Figure 9.3: The initial and final states for equation Problem 2.

where A and B are tridiagonal matrices given by

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

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

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

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

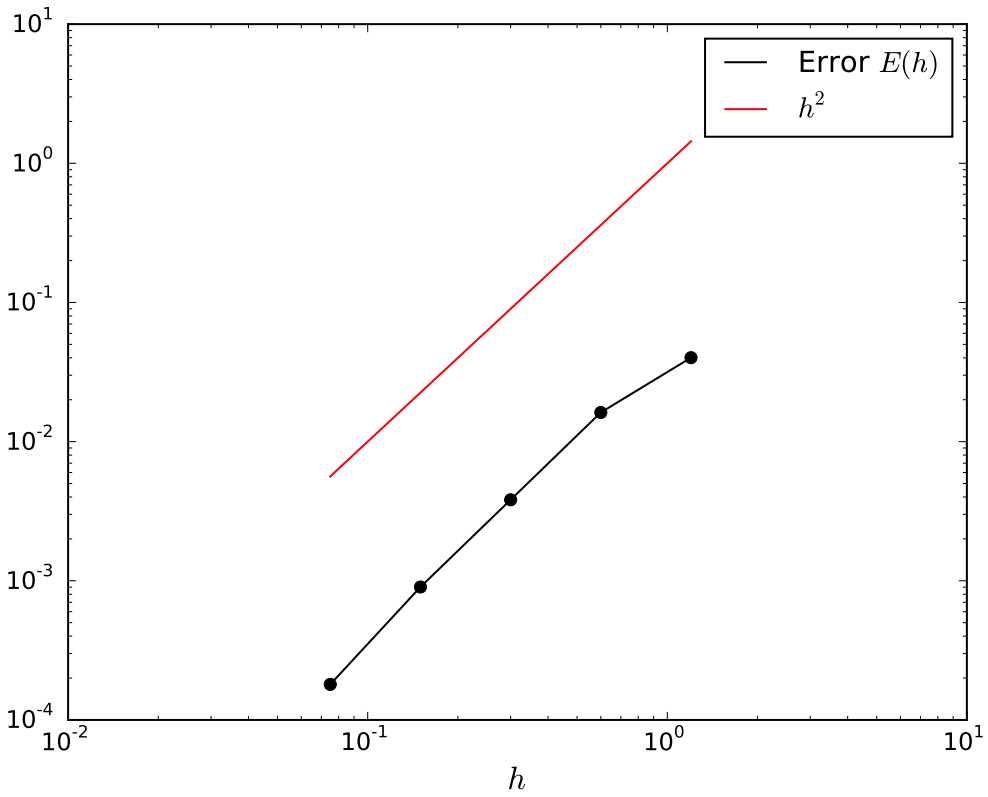


Figure 9.4: $E(h)$ represents the (approximate) maximum error in the numerical solution U to Problem 3 at time $t = 1$, using a stepsize of h.

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

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

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

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12, 12], \quad t \in [0, 1], \\ &u(-12, t) = 0, \quad u(12, t) = 0, \\ &u(x, 0) = \max\{1 - x^2, 0\}. \end{aligned} \tag{9.5}$$

Demonstrate that the numerical approximation at $t = 1$ converges to $u(x, t = 1)$. Do this by computing U at $t = 1$ using $20, 40, 80, 160, 320$, and 640 steps. Use the same number of steps in both time and space. Reproduce the loglog plot shown in Figure 9.4. The slope of the line there shows the proper rate of convergence.

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

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

10 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

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

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

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

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

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

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

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

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

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

Problem 1. Complete the following fuction, implementing anisotropic diffusion for black and white images using the following boundary conditions:

For the top edge let

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

Do the other edges similarly.

For the top left corner let

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

Do the other corners similarly.

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

In your function, use

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

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

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

original image

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

20 iterations

100 iterations

Color Schemes

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

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

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

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

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

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

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

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

original image

after 50 iterations

Figure 10.1: Smearing of similar colors when using an anisotropic diffusion filter.

Noisy Images

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

```
from numpy.random import randint

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

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

Hint: Don't forget to rescale.

Minimum Bias (Optional)

This sort of anisotropic diffusion can be very effective, but, depending on the image, it may also smear out edges that do not have large differences between them. An example of this limitation can be seen in Figure 10.1

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

***Problem 4.** (Optional)

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

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

Hint: Don't forget to rescale.

original image

randomly changed 100000 color values

300 iterations of a min-biased scheme

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

11 Wave Phenomena

Advection Equation

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

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

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

Wave Equation

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

$$u_{tt} = s^2 \triangle u. \tag{11.1}$$

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

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

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

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

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

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

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

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

Numerical solution of the wave equation

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

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

The resulting method is given by

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

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

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

where

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

and

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

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

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

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

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

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

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

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

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

or if f is not readily differentiable,

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

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

Problem 1. Consider the initial boundary value problem

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

Numerically approximate the solution $u(x, t)$ for $t \in [0, .5]$. Use $J = 50$ subintervals in the x dimension and $M = 50$ subintervals in the t dimension. Animate the results. Compare your results with the analytic solution $u(x, t) = \sin(2\pi x)\cos(2\pi t)$. This function is known as a standing wave. See Figure 11.1.

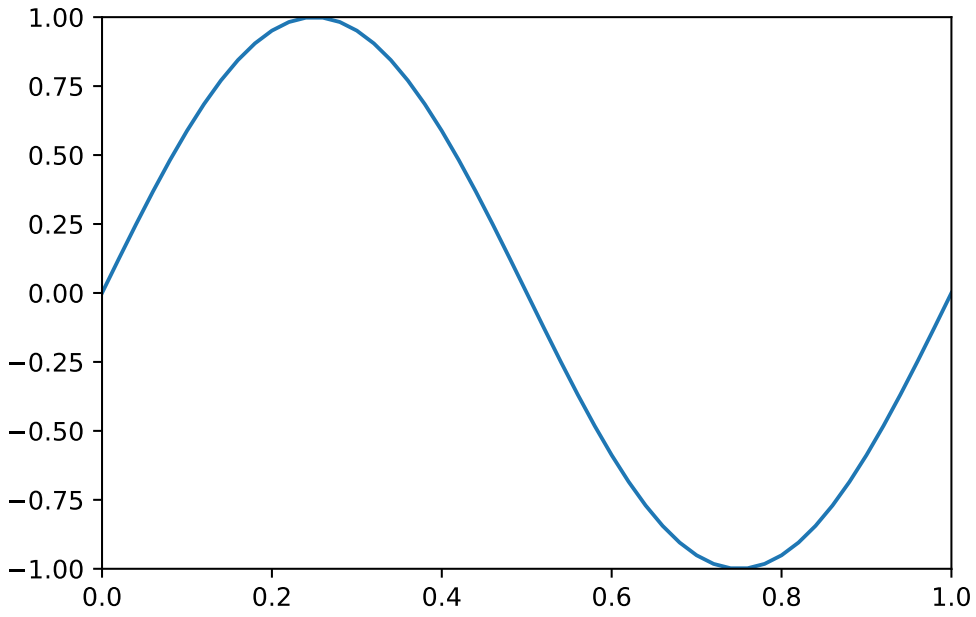


Figure 11.1: $u(x, t = 0)$.

Problem 2. Consider the initial boundary value problem

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

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

Numerically approximate the solution $u(x, t)$ for $t \in [0, 1]$. Set $m = 20$. Use 200 subintervals in space and 220 in time, and animate your results. Then use 200 subintervals in space and 180 in time, and animate your results. Note that the stability condition is not satisfied for the second mesh. See 11.2.

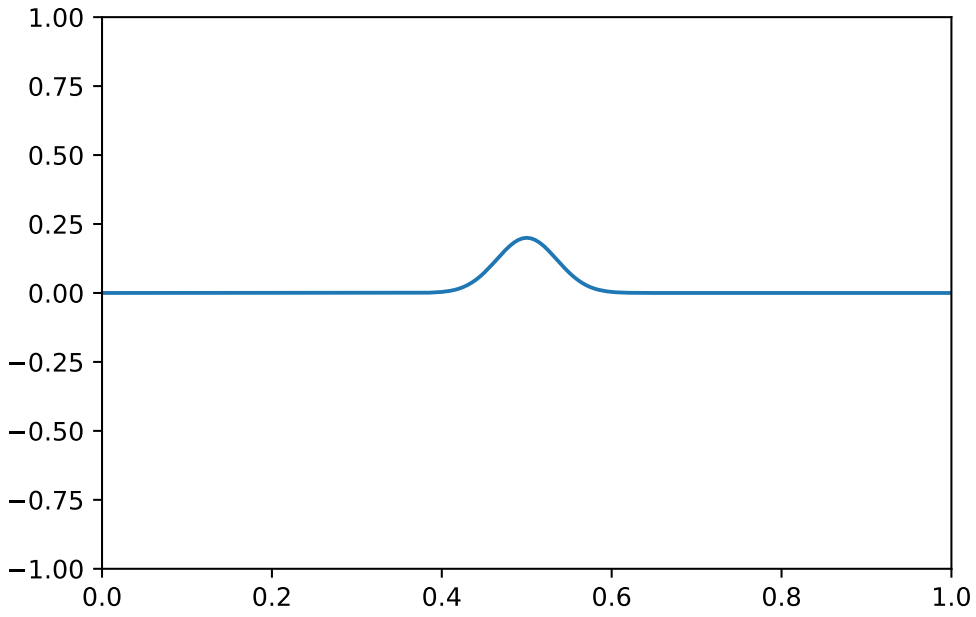


Figure 11.2: $u(x, t = 0)$.

Problem 3. Consider the initial boundary value problem

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

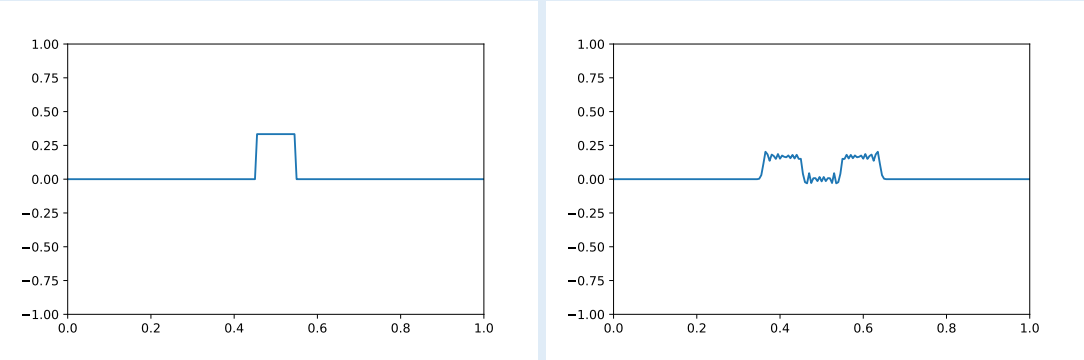
The initial condition separates into two smaller, slower-moving pulses, one travelling to the right and the other to the left. This solution models, for example, a plucked guitar string

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

Problem 4. Consider the initial boundary value problem

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

Numerically approximate the solution $u(x,t)$ for $t \in [0,2]$. Use 200 subintervals in space and 440 in time, and animate your results. Even though the method is second order and stable for this discretization, since the initial condition is discontinuous there are large dispersive errors. See Figure 11.3.



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

Figure 11.3: The graphs for Problem 4 at various times t.

Travelling Wave Solutions of an Evolution Equation

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

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

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

$$u_t = G(u, u_x, u_{xx}, \ldots) \tag{11.2}$$

An interesting question to ask is whether (11.2) has travelling wave solutions: is there a signal or wave profile $f(x)$, so that $u(x,t) = f(x - st)$ is a solution of (11.2) that carries the signal at a constant speed s? These travelling waves are often significant physically. For example, in a PDE modeling insect population dynamics a travelling wave could represent a swarm of locusts; in a PDE describing a combustion process a travelling wave could represent an explosion or detonation.

Burgers' equation

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

$$u_t + \left(\frac{u^2}{2}\right)_x = \nu u_{xx}, \tag{11.3}$$

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

Let us look for a travelling wave solution $u(x,t) = \hat{u}(x-st)$ for Burgers equation. We transform (11.3) into the moving frame $(x,t) \to (\bar{x},\bar{t}) = (x-st,t)$. In this frame (11.3) becomes

$$u_{\bar{t}} - su_{\bar{x}} + \left(\frac{u^2}{2}\right)_{\bar{x}} = \nu u_{\bar{x}\bar{x}} \tag{11.4}$$

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

$$-su_{\bar{x}} + \left(\frac{u^2}{2}\right)_{\bar{x}} = \nu u_{\bar{x}\bar{x}}. \tag{11.5}$$

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

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

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

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

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

Usually at this point, the travelling wave must be numerically solved using the profile ODE ((11.5) for Burgers equation). However, the profile ODE for Burgers is simple enough that it is possible to obtain an analytic solution. The travelling wave is given by

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

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

Stability of travelling waves

Suppose that an evolutionary PDE

$$u_t = G(u, u_x, u_{xx}, \ldots). \tag{11.6}$$

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

We begin by translating (11.6) into the moving frame $(x,t) \to (\bar{x},\bar{t}) = (x-st,t)$. In this frame the PDE becomes

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

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

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

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

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

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

Visualizing stability of the travelling wave solution of Burgers' equation

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

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

using the second order centered approximations

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

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

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

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

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

Problem 5. Numerically solve the initial value problem

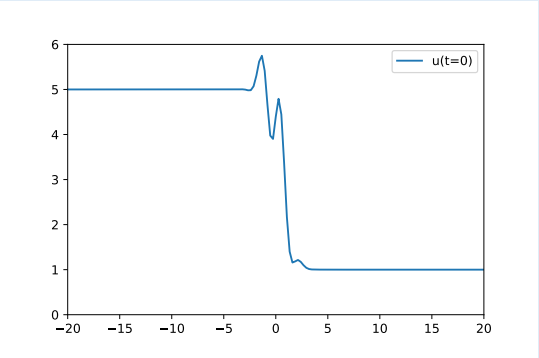
$$\begin{aligned} u_t - su_x + uu_x &= u_{xx}, \quad x \in (-\infty, \infty), \\ u(x,0) &= v(x), \end{aligned}$$

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

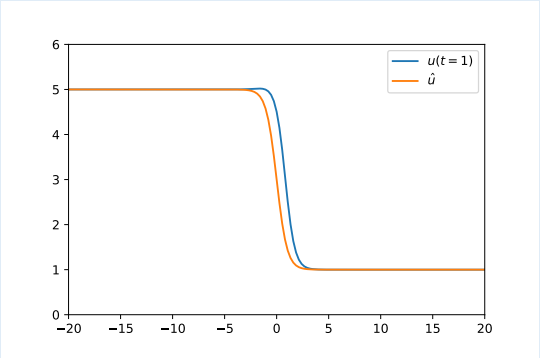
$$v(x) = 3.5(\sin(3x) + 1)\frac{1}{\sqrt{2\pi}} \exp(-x^2/2)$$

And let the initial condition be $u(x, 0) = \hat{u}(x) + v(x)$ Approximate the x domain,$(-\infty, \infty)$, numerically by the finite interval $[-20, 20]$, and fix $u(-20) = u_-$, $u(20) = u_+$. Let $u_- = 5$, $u_+ = 1$. Use 150 intervals in space and 350 steps in time. Animate your results. You should see the solution converge to a translate of the travelling wave $\hat{u}$. See Figure 11.4.

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



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



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

Figure 11.4: The graphs for Problem 5

12 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)

$$\begin{aligned} &u_t = \triangle u + f(x,t), \quad x \in \Omega, \quad t > 0, \\ &u(x,t) = h(x), \quad x \in \partial\Omega, \\ &u(x,0) = g(x). \end{aligned} \tag{12.1}$$

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

$$\begin{aligned} &\triangle u + f(x) = 0, \quad x \in \Omega, \\ &u(x,t) = h(x), \quad x \in \partial\Omega. \end{aligned} \tag{12.2}$$

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 square domain $R = [a,b] \times [c,d]$:

$$\begin{aligned} u_{xx} + u_{yy} &= f, \quad x \text{ in } R \subset \mathbb{R}^2, \\ u &= g, \quad x \text{ on } \partial R. \end{aligned} \tag{12.3}$$

Let $a = x_{-1}, x_0, \ldots, x_{N-1} = b$ be a partition of $[a,b]$, and let $c = y_{-1}, y_0, \ldots, y_{N-1} = d$ be a partition of $[c,d]$. Suppose that there are $N+1$ evenly spaced points, so that N is the number of subintervals in each dimension, and x_i, y_j are given by

$$\begin{aligned} x_i &= a + (i+1)h, \\ y_j &= c + (j+1)h, \end{aligned}$$

for $i, j = 0, \ldots, N-2$, where $h = x_i - x_{i-1} = y_i - y_{i-1}$. We look for an approximation $U_{i,j}$ on the grid $\{(x_i, y_j)\}_{i,j=-1}^{N-1}$.

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

$$\begin{aligned}\triangle_h U_{ij} &= \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2} + \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{h^2}, \\ &= \frac{1}{h^2}(U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{i,j}).\end{aligned}$$

Then the set of equations

$$\triangle_h U_{ij} = f_{ij}, \quad i, j = 0, \ldots, N-2,$$

can be written in matrix form as

$$AU + p + q = f.$$

A is a block tridiagonal matrix, given by

$$\frac{1}{h^2}\begin{bmatrix} T & I & & & \\ I & T & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & T & I \\ & & & I & T \end{bmatrix} \tag{12.4}$$

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

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

The vector U is given by

$$U = \begin{bmatrix} U^0 \\ U^1 \\ \\ U^{N-2} \end{bmatrix} \text{ where } U^j = \begin{bmatrix} U_{0,j} \\ U_{1,j} \\ \\ U_{N-2,j} \end{bmatrix} \text{ for each } j,\ 0 \le j \le N-2.$$

The vectors p and q come from the boundary conditions of (12.3), and are given by

$$p = \begin{bmatrix} p^0 \\ \cdots \\ \\ p^{N-2} \end{bmatrix}, \quad q = \begin{bmatrix} q^0 \\ \cdots \\ \\ q^{N-2} \end{bmatrix},$$

where

$$p^j = \frac{1}{h^2}\begin{bmatrix} g_{-1,\,j} \\ 0 \\ \vdots \\ 0 \\ g_{N-1,\,j} \end{bmatrix}, \quad 0 \le j \le N-2,$$

and

$$q^0 = \frac{1}{h^2}\begin{bmatrix} g_{0,-1} \\ g_{1,-1} \\ \vdots \\ g_{N-3,-1} \\ g_{N-2,-1} \end{bmatrix}, \quad q^{N-2} = \frac{1}{h^2}\begin{bmatrix} g_{0,N-1} \\ g_{1,N-1} \\ \vdots \\ g_{N-3,N-1} \\ g_{N-2,N-1} \end{bmatrix}, \quad q^j = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \quad 1 \le j \le N-3.$$

The following code implements the greater portion of the finite difference method; it leaves the construction of the matrix A in (12.4) to problem 12.5.

```
from __future__ import division
from scipy.sparse import spdiags
from scipy.sparse.linalg import spsolve

def poisson_square(a1,b1,c1,d1,n,bcs, source):
    # n = number of subintervals
    # We discretize in the x dimension by
    # a1 = x_0 < x_1< ... < x_n=b1, and
    # We discretize in the y dimension by
    # c1 = y_0 < y_1< ... < y_n=d1.
    # This means that we have interior points
    # {x_1, ..., x_{n-1}}\times {y_1, ..., y_{n-1}}
    # or {x_1, ..., x_m}\times {y_1, ..., y_m} where m = n-1.
    # In Python, this is indexed as
    # {x_0, ..., x_{m-1}}\times {y_0, ..., y_{m-1}}
    # We will have m**2 pairs of interior points, and
    # m**2 corresponding equations.
    # We will organize these equations by their
    # y coordinates: all equations centered
    # at (x_i, y_0) will be listed first,
    # then (x_i, y_1), and so on till (x_i, y_{m-1})
    delta_x, delta_y, h, m = (b1-a1)/n, (d1-c1)/n, (b1-a1)/n, n-1

    ####    Construct the matrix A    ####


    ####    Here we construct the vector b    ####
    b, Array = np.zeros(m**2), np.linspace(0.,1.,m+2)[1:-1]
    # In the next line, source represents
    # the inhomogenous part of Poisson's equation
    for j in xrange(m):
        b[j*m:(j+1)*m] = source(a1+(b1-a1)*Array, c1+(j+1)*h*np.ones(m) )
```

```
    # In the next four lines, bcs represents the
    # Dirichlet conditions on the boundary
    # y = c1+h, d1-h
    b[0:m] -= h**(-2.)*bcs(a1+(b1-a1)*Array,c1*np.ones(m))
    b[(m-1)*m:m**2] -= h**(-2.)*bcs(a1+(b1-a1)*Array,d1*np.ones(m))
    # x = a1+h, b1-h
    b[0::m] -= h**(-2.)*bcs(a1*np.ones(m),c1+(d1-c1)*Array)
    b[(m-1)::m] -= h**(-2.)*bcs(b1*np.ones(m),c1+(d1-c1)*Array)

    ####    Here we solve the system A*soln = b    ####
    soln = spsolve(A,b)

    # We return the solution, and the boundary values,
    # in the array z.
    z = np.zeros((m+2,m+2) )
    for j in xrange(m):
        z[1:-1,j+1] = soln[j*m:(j+1)*m]

    x, y = np.linspace(a1,b1,m+2), np.linspace(c1,d1,m+2)
    z[:,0], z[:,m+1]  = bcs(x,c1*np.ones(len(x)) ), bcs(x,d1*np.ones(len(x)) )
    z[0,:], z[m+1,:] = bcs(a1*np.ones(len(x)),y), bcs(b1*np.ones(len(x)),y)
    return z
```

Problem 1. Construct the matrix A in (12.4). Make sure your matrix is sparse. Then use the code above to solve the boundary value problem

$$\begin{aligned} \Delta u &= 0, \quad x \in [0,1] \times [0,1], \\ u(x,y) &= x^3, \quad (x,y) \in \partial([0,1] \times [0,1]). \end{aligned} \tag{12.5}$$

Plot the 3D solution with $n = 100$.

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 become the source term) and Coulomb's Law, which gives the force between two charge particles (where charge is the source term).

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:

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

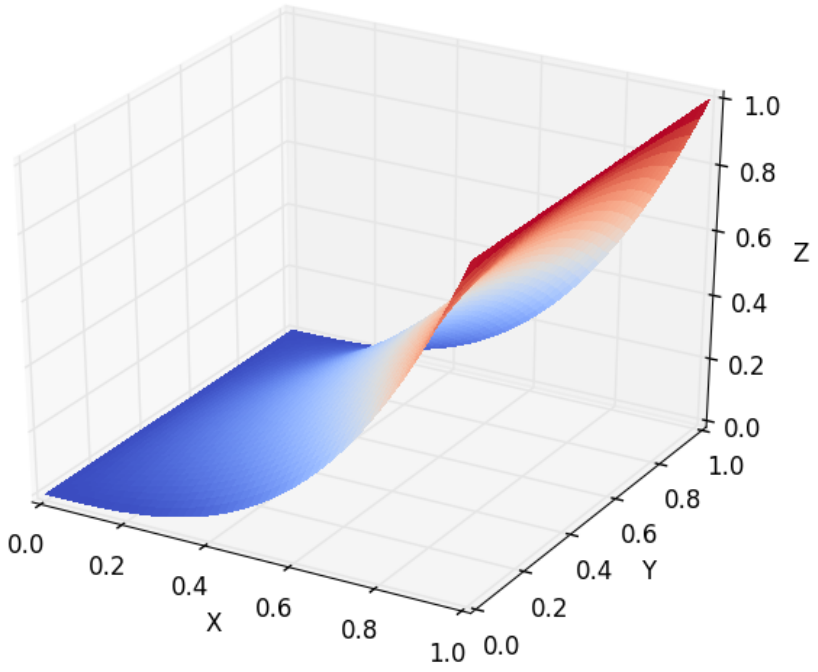


Figure 12.1: The solution of (12.5).

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 non zero 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) ↩
            < 1.0),
        np.logical_and( np.logical_or(abs(Y-0.9) < .1,abs(Y+0.9) < .1) ,abs(X) ↩
            < 1.0))] = -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':   [(0.0,  1.0, 1.0),
                       (zero,  1.0, 1.0),
                       (1.0,  0.0, 0.0)],
             'green': [(0.0,  0.0, 0.0),
                       (zero,  1.0, 1.0),
                       (1.0,  0.0, 0.0)],
             'blue':  [(0.0,  0.0, 0.0),
                       (zero,  1.0, 1.0),
                       (1.0,  1.0, 1.0)]}
    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))))
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.

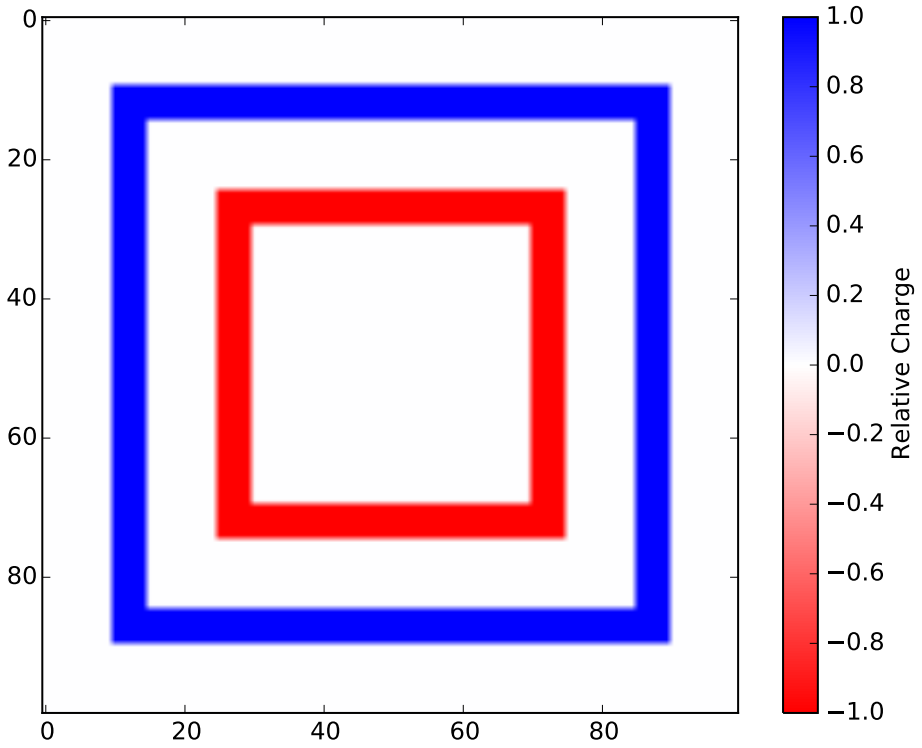


Figure 12.2: The charge density of the 3 nested pipes.

Problem 2. Solve

$$\begin{aligned} \Delta V &= -\rho(x,y), \quad x \in [-2,2] \times [-2,2], \\ u(x,y) &= 0, \quad (x,y) \in \partial([-2,2] \times [-2,2]). \end{aligned} \tag{12.6}$$

for the electric potential V. Use the source function $(-\rho)$ defined above. Plot the 2D solution using $n = 100$.

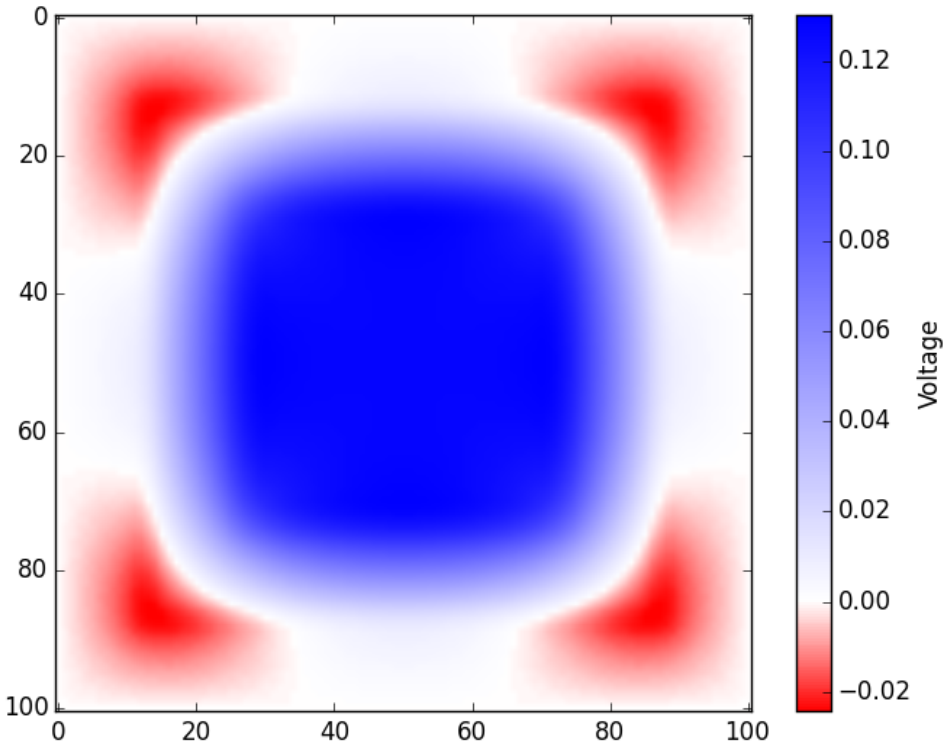


Figure 12.3: The electric potential of the 3 nested pipes.

13 Finite Volume Methods

When solving a PDE numerically, how do we deal with discontinuous initial data? The Finite Volume method has particular strength in this area. It is commonly used for hyperbolic PDEs whose solutions can spontaneously develop discontinuities as they evolve in time. These solutions are often called shock waves.

Conservation Laws

Consider the conservation law

$$u_t + f(u)_x = 0, \tag{13.1}$$

where u is a (spatially) one-dimensional conserved quantity, and $f(u)$ is the flux of u. The continuous integral formulation of (13.1) states that

$$\frac{d}{dt}\int_a^b u(x,t)dx + \int_a^b f(u)_x\,dx = 0.$$

$\frac{d}{dt}\int_a^b u(x,t)dx$ may be thought of as the time evolution of the total 'mass' of u across the domain $[a,b]$, and is dependent only on the flux through the boundaries, since

$$\frac{d}{dt}\int_a^b u(x,t)dx = f(u(a)) - f(u(b)).$$

This fact is an important idea utilized by finite volume methods, which generally consider the evolution of u not at a given point, but instead in volume-averaged regions. For example, let $\{x_i\}$ be a grid of equally spaced points with spacing Δx, and let C_i be the i-th 'volume' (subinterval) defined by $(x_{i-1/2}, x_{i+1/2})$. We are interested in the evolution of the volume average of u over this interval,

$$U_i^n = \frac{1}{\Delta x}\int_{C_i} u(x,t^n)dx,$$

where $\{t^n\}$ is the time discretization.

The evolution of these volume-averaged quantities will depend only on the flux through the cell edges, so that

$$\frac{d}{dt}\int_{C_i} u(x,t)dx = f(u(x_{i-1/2},t)) - f(u(x_{i+1/2},t)). \tag{13.2}$$

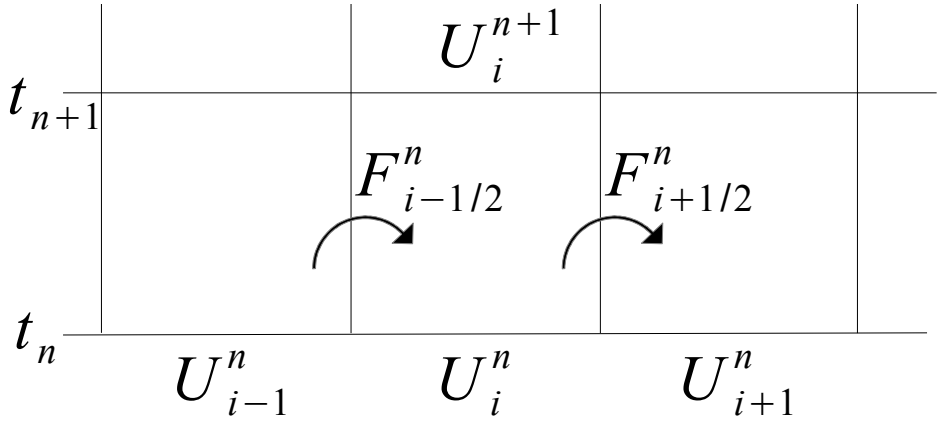


Figure 13.1: A schematic of the fluxes for the finite volume method as indicated by (13.3).

We can then construct a time-stepping method where $\sum_i U_i^n \Delta x$ (the total 'mass' of the system) is conserved from one time step n to the next.

Let $F_{i-1/2}^n = \frac{1}{\triangle t}\int_{t^n}^{t^{n+1}} f(u(x_{i-1/2},t))\,dt$. Then

$$\begin{aligned}\int_{t^n}^{t^{n+1}} \left[\frac{d}{dt}\int_{C_i} u(x,t)\,dx\right] &= \int_{C_i} u(x,t^{n+1}) - u(x,t^n)\,dx,\\ &= \triangle t\left(F_{i-1/2}^n - F_{i+1/2}^n\right).\end{aligned}$$

Thus, by integrating (13.2) in time, we may approximate the evolution of the cell ('volume') averages with the method

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1/2}^n - F_{i-1/2}^n\right). \tag{13.3}$$

where $U_i^n = \frac{1}{\triangle x}\int_{C_i} u(x,t^n)\,dt$. This formulation guarantees the conservation properties that are so desirable for conservation laws, if the time-averaged fluxes $F_{i-1/2}^n$ can be discretized in a natural way.

The key contribution of finite volume methods is the computation of $F_{i-1/2}^n$. For a truly nonlinear $f(u)$ this can be rather complicated and messy, and typically will involve solving what is usually referred to as the Riemann problem for the conservation law. The interested student can look at [LeV02] for a very thorough introduction and discussion on the subject. We will consider the linear problem in one dimension. The analog to higher dimensions is obtained by considering the eigenvector decomposition of any linear system. Nonlinear equations complicate things further.

The linear advection equation and upwinding

The simplest conservation law describes the advection or transport of a quantity. The PDE is given by

$$u_t + a u_x = 0, \tag{13.4}$$

and describes the motion of a concentration of some constituent u by a constant velocity one-dimensional 'wind' $a > 0$. In higher dimensions this is an important problem in many fields, for example the transport of chemicals in the atmosphere and oceans, proper mixing of various properties in metallurgy, and the passing of information along a network.

Note that whenever $u(x,t)$ is a solution of the advection equation, then $u(x-at,t_0)$ (for any fixed t_0) is also a solution. Thus, if $u(x,0)=u_0(x)$ then the solution for all time can be represented by $u(x,t)=u_0(x-at)$. This is an important property of (13.4), and gives a new meaning to the term advection: this equation merely takes the initial conditions and passively transports them with velocity a.

For this equation the computation of the flux appears straightforward: $F^n_{i-1/2}=a\overline{U}^n_{i-1/2}$ where the $\overline{U}^n_{i-1/2}$ refers to the time average of $U_{i-1/2}$ over the interval t_n to t_{n+1}. Let us determine how to approximate this time average. Note from Figure 13.2 that when $a>0$ the flux that determines U^{n+1}_i will be dependent on the value of U^n_{i-1}. Thus, one possibility is to approximate the flux by $F^n_{i-1/2}=aU^n_{i-1}$. Using this approximation of the flux together with the flux differencing formula (13.3) yields the first order upwind method, given by

$$U^{n+1}_i = U^n_i - \frac{a\Delta t}{\Delta x}\left(U^n_i - U^n_{i-1}\right).$$

Another way to derive the upwind method is to instead suppose that what we want to do is reconstruct $u(x)$ at each time step n inside each cell $(x_{i-1/2},x_{i+1/2})$from the mean values in that cell and its surrounding neighbors. This reconstructed $\tilde{u}(x)^n$ is then defined piecewise for each cell i. The solution at the next time step can be found as $\tilde{u}^{n+1}(x)=\tilde{u}^n(x-a\triangle t)$ which allows us to determine the fluxes $F^n_{i-1/2}$ once we have settled on a method for determining $\tilde{u}^n(x)$ in each cell. The simplest approach is

$$\tilde{u}^n(x) = U^n_i \text{ for } x\in(x_{i-1/2},x_{i+1/2})$$

This leads to fluxes given by

$$\begin{aligned} F^n_{i-1/2} &= \frac{a}{\triangle t}\int_0^{t_{n+1}-t_n} \tilde{u}^n(x_{i-1/2},t)\,dt, && (13.5)\\ &= \frac{a}{\triangle t}\int_0^{\triangle t} \tilde{u}^n(x_{i-1/2}-at)\,dt, && (13.6)\\ &= aU^n_{i-1}. \end{aligned}$$

The following code solves the problem

$$\begin{aligned} u_t + au_x &= 0, \quad 0<x<1,\\ u(x,t) &= f(x), \\ u(0,t) &= u(1,t), \end{aligned} \tag{13.7}$$

where f represents a signal with two parts: one is smooth and the other is discontinuous. Notice that this PDE has periodic boundary conditions. Essentially we are evolving the signal around the unit circle. This allows us to evolve the signal much further to test our numerical methods, since we only have to discretize the interval $[0,1]$ instead of a much larger domain. To see how to implement the boundary conditions, consider a grid $0=x_0<x_1<\ldots<x_{N-1}<x_N=1$ of evenly spaced points. Since $u(x)$ is periodic then $u(x_N)=u(x_0)$, so it is sufficient to track $x_0,\ldots,x_{N-1}$.

```
import numpy as np
from matplotlib import pyplot as plt
from math import floor

def upwind(u0, a, xmin, xmax, t_final, nt):
    """ Solve the advection equation with periodic
    boundary conditions on the interval [xmin, xmax]
```

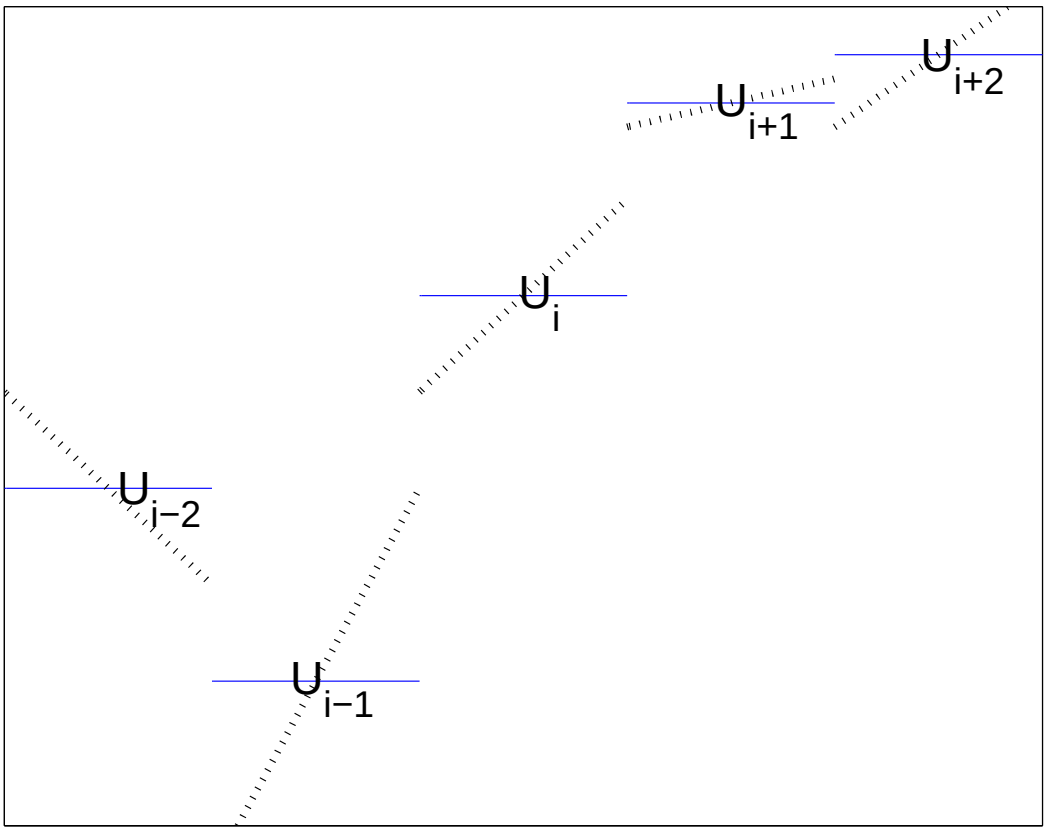


Figure 13.2: The piecewise linear reconstruction for the upwind and Lax-Wendroff methods. The solid lines represent the simplest reconstruction of the cell averages leading to the upwind method, and the dashed lines are those whose slope is obtained via the Lax-Wendroff method. Note that the LW method introduces a spurious maximum at $i + 3/2$ (the cell edge between U_{i+1} and U_{i+2}) and the minimum at $i - 3/2$ will be unphysical exaggerated. The upwind method avoids this difficulties, but clearly loses a significant amount of the available information. This provides the motivation for the slope limiters.

```
    using the upwind finite volume scheme.
    Use u0 as the initial conditions.
    a is the constant from the PDE.
    Use the size of u0 as the number of nodes in
    the spatial dimension.
    Let nt be the number of spaces in the time dimension
    (this is the same as the number of steps if you do
    not include the initial state).
    Plot and show the computed solution along
    with the exact solution. """
```

```
    dt = float(t_final) / nt
    # Since we are doing periodic boundary conditions,
    # we need to divide by u0.size instead of (u0.size - 1).
    dx = float(xmax - xmin) / u0.size
    lambda_ = a * dt / dx
    u = u0.copy()
    for j in xrange(nt):
        # The Upwind method. The np.roll function helps us
        # account for the periodic boundary conditions.
        u -= lambda_ * (u - np.roll(u, 1))
    # Get the x values for the plots.
    x = np.linspace(xmin, xmax, u0.size+1)[:-1]
    # Plot the computed solution.
    plt.plot(x, u, label='Upwind Method')
    # Find the exact solution and plot it.
    distance = a * t_final
    roll = int((distance - floor(distance)) * u0.size)
    plt.plot(x, np.roll(u0, roll), label='Exact solution')
    # Show the plot with the legend.
    plt.legend(loc='best')
    plt.show()

# Define the initial conditions.
# Leave off the last point since we're using periodic
# boundary conditions.
nx = 30
nt = nx * 3 // 2
x = np.linspace(0., 1., nx+1)[:-1]
u0 = np.exp(-(x - .3)**2 / .005)
arr = (.6 < x)  &  (x < .7 )
u0[arr] += 1.

# Run the simulation.
upwind(u0, 1.2, 0, 1, 1.2, nt)
```

Try running the previous code block with `nx` set to 30, 60, 120, and 240. You will notice that the numerical solution diffuses with time. It diffuses especially fast at the points of discontinuity.

Piecewise linear reconstruction and slope limiters

The upwind method is formally only first order, and actually does relatively poorly in terms of actually transporting the initial data with velocity a. You can notice from the example code that the upwind method has errors that are 'diffusive' meaning that the initial data is diffused as time evolves, losing the peaks and fine details. This is because the error for the upwind method is on the order of the second derivative of u which is of a diffusive nature. To get an improved method, consider a better reconstruction inside each cell, i.e.

$$\tilde{u}^n(x) = U_i^n + m_i^n(x - x_i) \text{ for } x \in (x_{i-1/2}, x_{i+1/2}) \tag{13.8}$$

where the slope of this linear reconstruction m_i^n is determined as a function of the neighboring cell averages at time n and U_i^n itself. Then the flux is given by

$$\begin{aligned} F_{i-1/2}^n &= \frac{a}{\triangle t}\int_0^{t_{n+1}-t_n} \tilde{u}^n(x_{i-1/2}-at)\,dt, \\ &= \frac{a}{\triangle t}\int_0^{\triangle t} U_{i-1}^n + m_i^n(x_{i-1/2}-at-x_i), \\ &= a\left(U_{i-1}^n + \frac{m_{i-1}^n}{2}(\triangle x - a\triangle t)\right). \end{aligned} \tag{13.9}$$

One of the most natural approaches is to just estimate the slope depending on the cell i and a neighboring cell $i+1$ or $i-1$. This leads to two popular methods, the Lax-Wendroff method and the Beam-Warming method (that really is the name). The Lax-Wendroff method has a slope chosen as

$$m_i^n = \frac{U_{i+1}^n - U_i^n}{\Delta x}. \tag{13.10}$$

which it turns out is formally second-order accurate. It turns out though that the errors for this method are dispersive, meaning that near very steep gradients, the method will generate very rapid oscillations (due to the third derivative of u not being approximated accurately). Another way to consider how these errors arise is to notice from Figure 13.2 that if the piecewise linear reconstruction is advocated by some positive wind a then there will be places where the discontinuous nature of the reconstruction will introduce spurious maxima or minima into the solution. These become the spurious waves seen in simulations using the Lax-Wendroff method.

A solution to this dilemma between balancing the diffusive and dispersive errors comes from constructing slopes m_i^n that ensure no such non-monotonic transport takes place. The basic idea is to constrain the slope so that the reconstructed piecewise linear function $\tilde{u}^n(x)$ will not generate unphysical extremal values when it is advocated by some finite wind a. The Minmod limiter chooses the slope as

$$m_i^n = minmod\left(\frac{U_i^n - U_{i-1}^n}{\Delta x}, \frac{U_{i+1}^n - U_i^n}{\Delta x}\right) \tag{13.11}$$

where

$$minmod(a,b) = \begin{cases} a \text{ if } |a| < |b| \text{ and } ab > 0 \\ b \text{ if } |b| < |a| \text{ and } ab > 0 \\ 0 \text{ if } ab < 0. \end{cases} \tag{13.12}$$

Problem 1. Implement the Lax Wendroff method and use it to solve (13.7). For $N = 30, 60, 120, 240$, plot the analytic solution, the upwind solution, and the Lax-Wendroff solution. (You should have 4 separate plots, each with 3 graphs.) You should be able to tell that the Lax Wendroff method approximates the smooth portion of the signal much better, as it does not struggle with diffusion. Unfortunately, it has some difficulty with the discontinuous portion, where unphysical oscillations are seen. Recall that we saw something similar in the waves lab when there were discontinuous initial conditions.

Hint: Use equations 13.9 and 13.3.

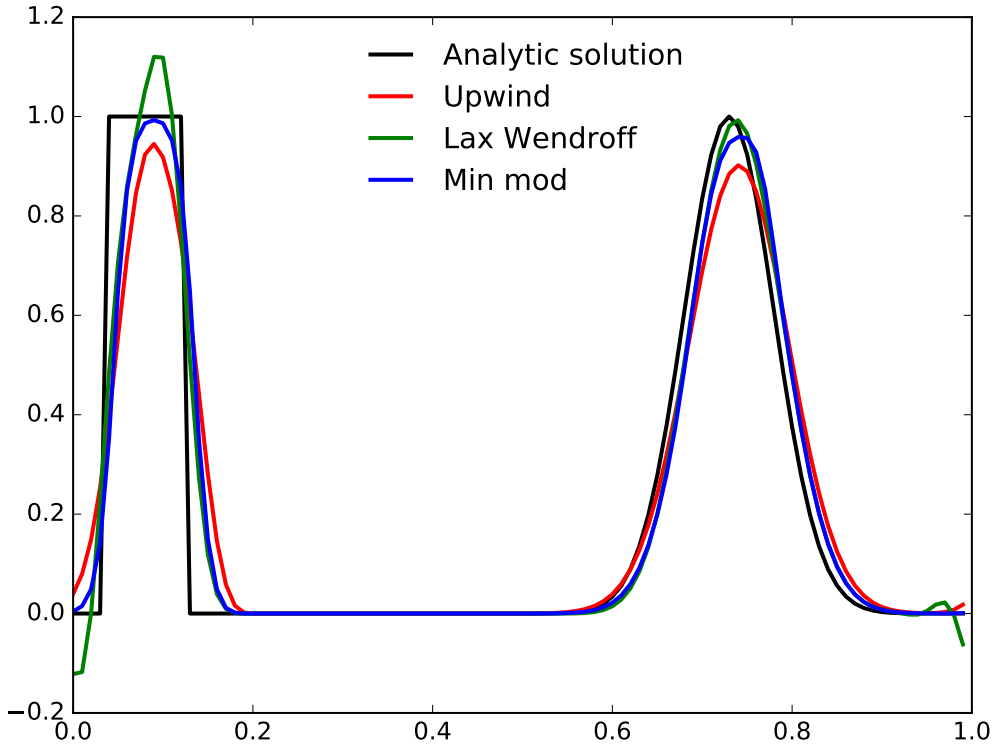


Figure 13.3: Solutions of (13.7) at time $t = 1.2$ using various methods. Here the advection coefficient is $a = 1.2$, and there are $N = 100$ subintervals in space, 150 subintervals in time.

Problem 2. Implement the Minmod method and use it to solve (13.7). For $N = 30, 60, 120, 240$, plot the anaytic solution, the upwind solution, the Lax-Wendroff solution, and the Minmod solution. (You should have 4 separate plots, each with 4 graphs.) Be sure to vectorize the minmod operation.

Hint: Use equations 13.9 and 13.3.

Beyond piecewise linear reconstructions

As you can imagine, using a linear approximation is not the only option. There are a host of high order finite volume methods that consider polynomial reconstructions of $\tilde{u}^n$ inside each cell. The key is then to use some nonlinear limiting technique that will ensure that when $\tilde{u}^n(x)$ is advocated that no new extrema are introduced. Choosing the correct limiter for the given application then becomes an art unto itself.

14 The Finite Element method

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

Advection-Diffusion of Heat in a Fluid

We begin with the heat equation

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

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

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

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

These conditions yield

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

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

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

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

$$\begin{aligned} &\varepsilon y'' - y' = -1, \quad 0 < x < 1, \\ &y(0) = 2, \quad y(1) = 4. \end{aligned} \tag{14.1}$$

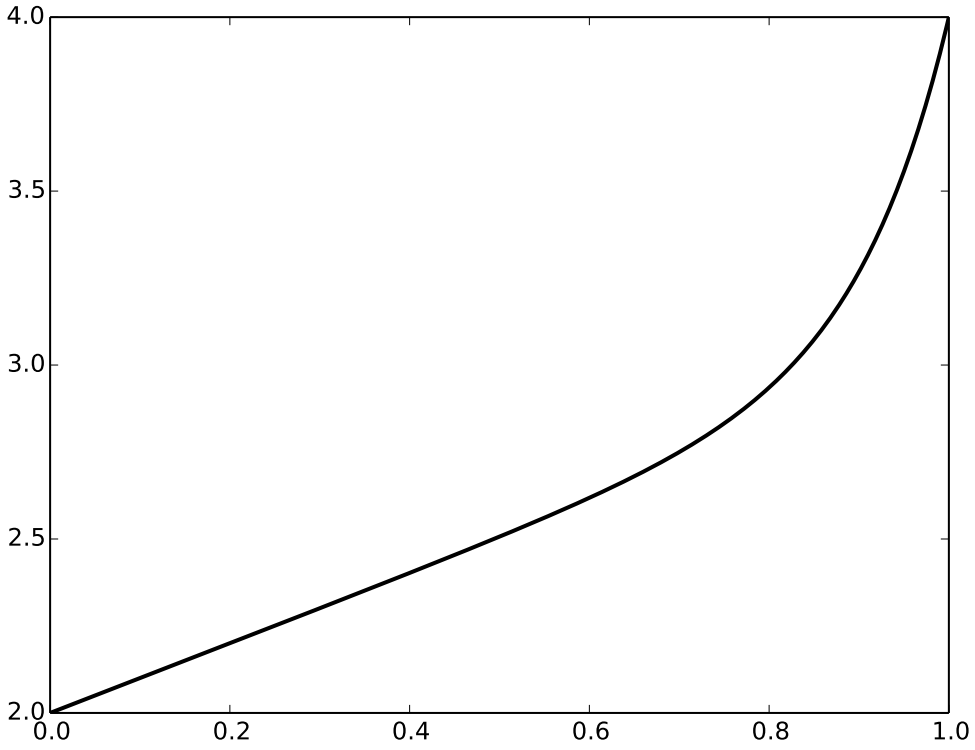


Figure 14.1: The solution of (14.1) for $\varepsilon = .1$.

The Weak Formulation

Consider the equation

$$\begin{aligned} &\varepsilon y'' - y' = f, \quad 0 < x < 1, \\ &y(0) = \alpha, \quad y(1) = \beta. \end{aligned} \tag{14.2}$$

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

Let w be a smooth function on $[0, 1]$ satisfying $w(0) = w(1) = 0$. Multiplying (14.2) by w and integrating over $[0, 1]$ yields

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

Define a bilinear function a and a linear function l by

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

Rather than trying to solve (14.2), we instead consider the problem of finding a function y such that

$$a(y,w) = l(w), \quad \forall\, w \in V_0, \tag{14.3}$$

where V is some appropriate vector space that is expected to allow us to approximate the solution y, and $V_0 = \{w \in V | w(0) = w(1) = 0\}$. (For example, we could consider the space of functions that are piecewise linear with vertices at a fixed set of points. This example is discussed further below.) This equation is called the weak formulation of (14.2).

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

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

The Numerical Method

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

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

where $h_i = x_i - x_{i-1}$; see Figures 14.2 and 14.3.

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

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

Equivalently, we require that

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

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

Since a is bilinear, we obtain

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

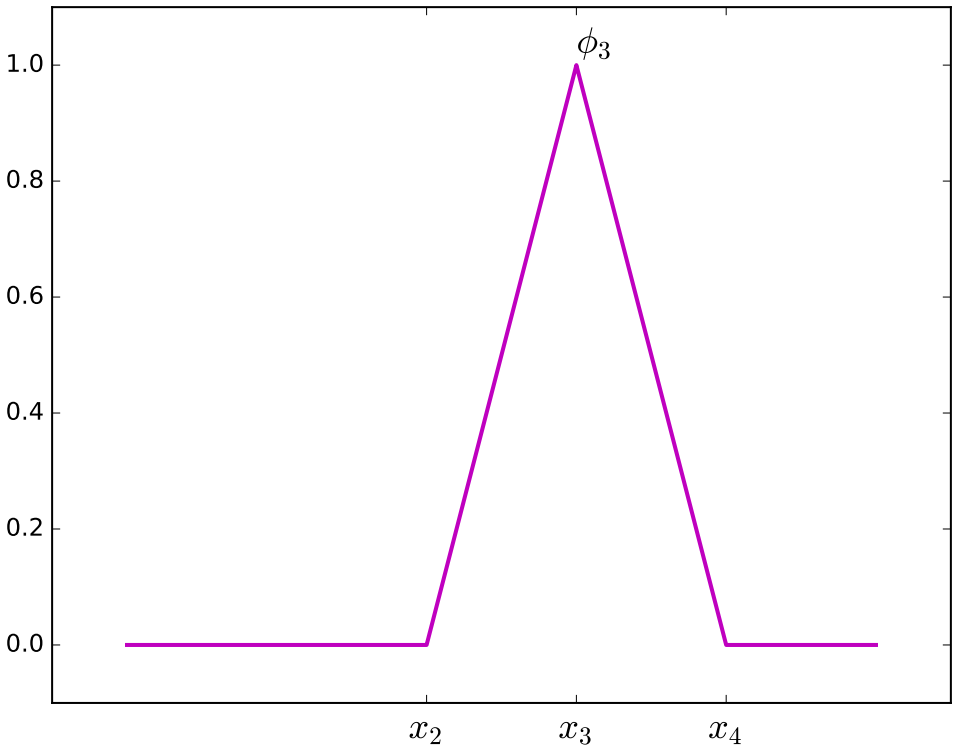


Figure 14.2: The basis function ϕ_3.

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

$$AK = \Phi, \tag{14.4}$$

where

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

and

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

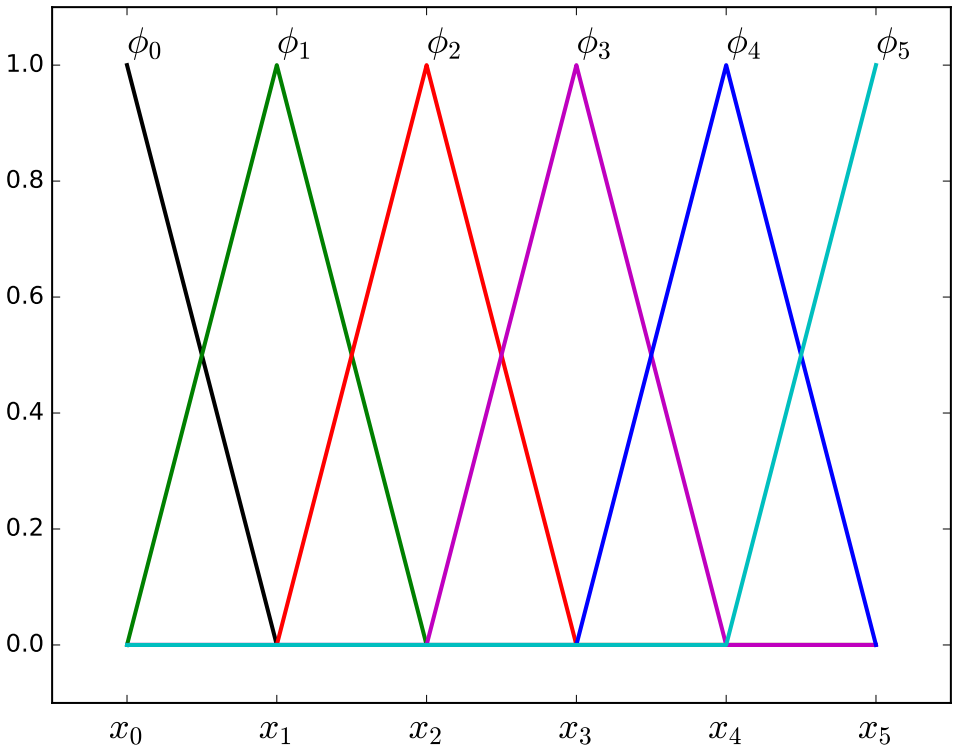


Figure 14.3: Basis functions for V_5.

Note that $a(\phi_i, \phi_j) = 0$ for most values of i, j (that is, when the hat functions do not have overlapping domains). Thus the finite element method results in a sparse linear system. To compute the coefficients of (14.4) we begin by evaluating some integrals. Since

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

we obtain

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

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

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

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

Equation (14.4) may now be solved using any standard linear solver. To handle the large number of elements required for Problem 3, you will want to use the tridiagonal algorithm provided in several of the earlier labs or the banded matrix solver included in `scipy.linalg`.

Problem 1. Use the finite element method to solve

$$\begin{aligned} \varepsilon y'' - y' &= -1, \\ y(0) = \alpha, \quad y(1) &= \beta, \end{aligned} \tag{14.5}$$

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

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

Problem 2. One of the strengths of the finite element method is the ability to generate grids that better suit the problem. The solution of (14.5) changes most rapidly near $x = 1$. Compare the numerical solution when the grid points are unevenly spaced versus when the grid points are clustered in the area of greatest change; see Figure 14.4. Specifically, use the grid points defined by

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

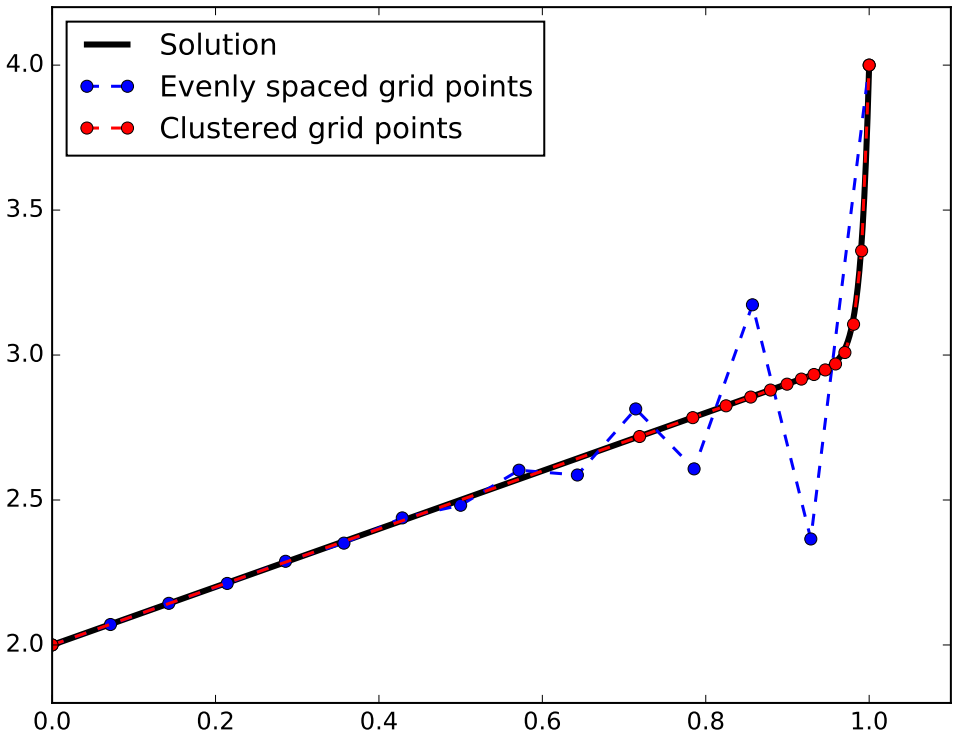


Figure 14.4: We plot two finite element approximations using 15 grid points.

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

Solve (14.5) with the finite element method using $N = 2^i$ finite elements, $i = 4, 5, \ldots, 21$. Use a log-log plot to graph the error; see Figure 14.5.

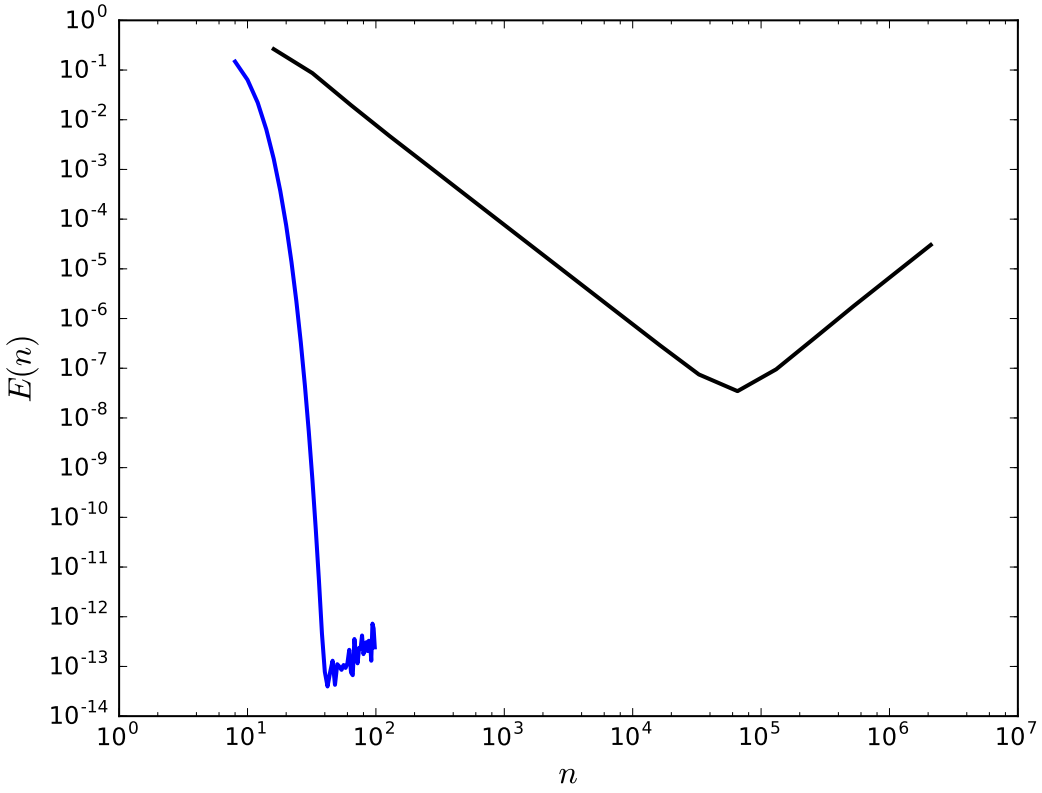


Figure 14.5: Error for the second order finite element method, as the number of subintervals n grows. Round-off error eventually overwhelms the approximation.

15 The Finite Element Method in Two Dimensions

In Lab 14 we discussed some of the basic details of how the Finite Element Method works. We demonstrated how a grid can be refined around areas of interest to give a more accurate approximation to a desired solution. One of the other great strengths of the finite element method is that, when it is used in two or more dimensions, it is easily applied to unusually shaped domains. There are a wide variety of elements that are commonly used, but the simplest elements are triangles. Triangles are often used because they allow us to define continuous piecewise linear basis functions on their interiors without much trouble.

Working With Triangle Meshes

There are a variety of ways to store triangle meshes. We will present a simple one here. To store a mesh we need to store several pieces of information. We need to store the points used in the mesh. We also need to store which points are connected to make the triangles that are part of the mesh we are studying. Later, when working with boundary conditions, we will also need to know which nodes have fixed values (or some other sort of boundary condition, as the case may be).

It is common to store this information in arrays, one containing the x and y coordinates of each node in each of its rows and another containing the indices of the nodes that form the vertices of each triangle.

The following is a short example that divides the unit square up into triangles and returns arrays of the desired form.

```
import numpy as np

def triangles(n):
    ''' Generate the indices of the triangles for a triangular mesh
    on a square grid of points.
    'n' is expected to be the number of nodes on each edge. '''
    # Make the indices for a single row.
    row = np.empty((2 * (n - 1), 3), dtype=np.int32)
    row[::2,0] = row[1::2,0] = row[::2,1] = np.arange(n-1)
    row[1::2,0] += 1
    row[::2,1] += n
    row[1::2,1] = row[::2,1]
    row[::2,2] = row[1::2,2] = row[1::2,0]
```

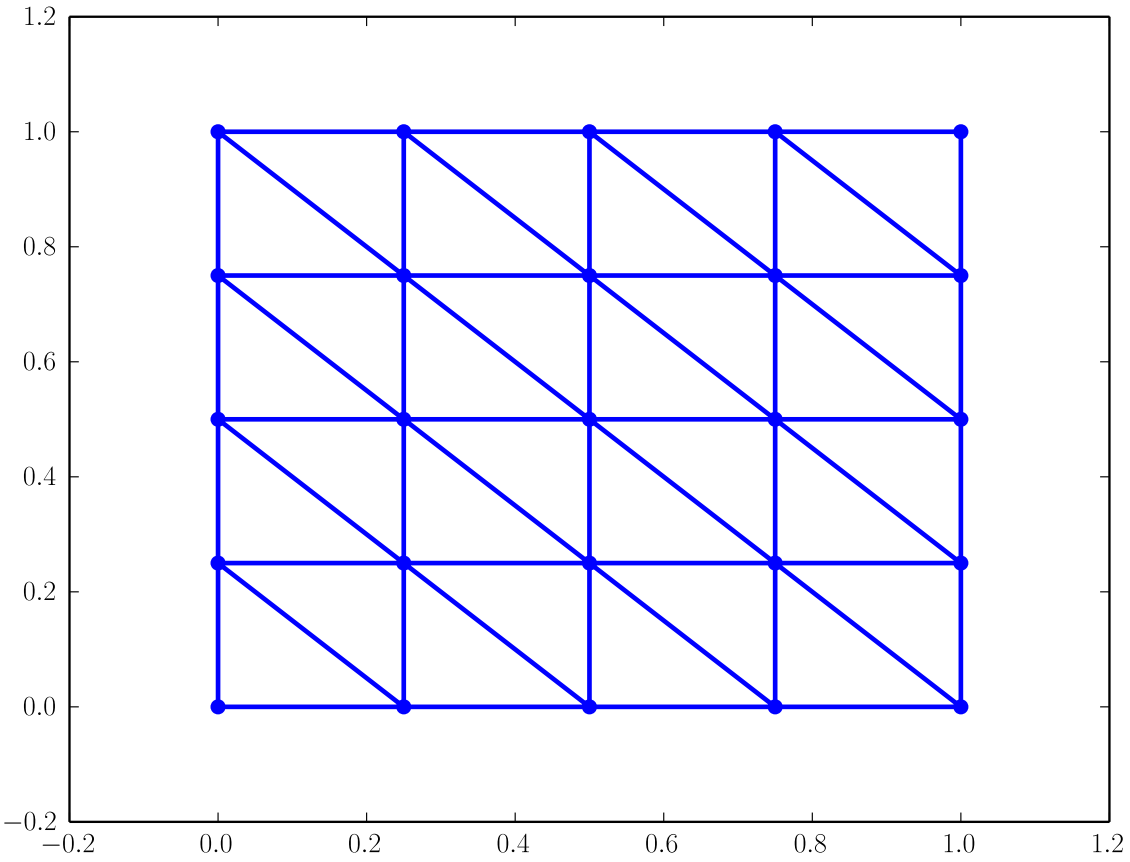


Figure 15.1: A triangulation of a square grid.

```
    row[1::2,2] += n
    # Now use broadcasting to make the indices for the square.
    return (row + np.arange(0, n * (n-1), n)[:,None,None]).reshape((-1,3))
```

Matplotlib's `triplot` function can be used to plot triangulations. To plot the mesh generated by the above function, we can do the following:

```
from matplotlib import pyplot as plt
n=5
x = np.linspace(0, 1, n)
x, y = map(np.ravel, np.meshgrid(x, x))
t = triangles(n)
plt.triplot(x, y, t, color='b')
plt.scatter(x, y, color='b')
plt.show()
```

This mesh is shown in figure 15.1.

Matplotlib and Mayavi both include the functionality to plot 3d surfaces based on a triangulation and the values of a function at the nodes of the triangulation. For example, we can plot a piecewise linear function that is one at a single vertex and zero at every other vertex using Mayavi using the `triangular_mesh` function included in `mlab`.

```
from mayavi import mlab as ml
n=5
x = np.linspace(0, 1, n)
x, y = np.meshgrid(x, x)
t = triangle_mesh(n)
vals = np.zeros(x.size)
vals[n**2 // 2] = 1
ml.triangular_mesh(x.ravel(), y.ravel(), vals, t)
ml.show()
```

The output from this code is shown in Figure 15.2. These functions are often called "hat functions," and are often used as the basis functions for 2d finite element analysis on domains that can be represented as meshes of triangles. These are the two dimensional analogues of the piecewise linear functions shown in Figure 14.2. You may recall that, in the one-dimensional case, we could represent any function that was continuous and linear over each element as a sum of these basis functions. The same is true in this case. We can represent any continuous function that is linear on each triangle in the triangulation as a sum of these basis functions. Strictly speaking, if f is a continuous function that is linear on each of the triangles in the triangulation, p_i are the vertices of the trianglulation, and ϕ_i are the basis functions corresponding to each vertex, we may say

$$f(x) = \sum_i f(p_i)\phi_i(x, y)$$

We can plot funcitons like this on triangulations using matplotlib as well. The following code will plot the same basis function in matplotlib

```
from mpl_toolkits.mplot3d import Axes3D
n=6
x = np.linspace(0, 1, n)
x, y = map(np.ravel, np.meshgrid(x, x))
t = triangles(n)
vals = np.random.rand(x.size)
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.plot_trisurf(x, y, vals, triangles=t)
plt.show()
```

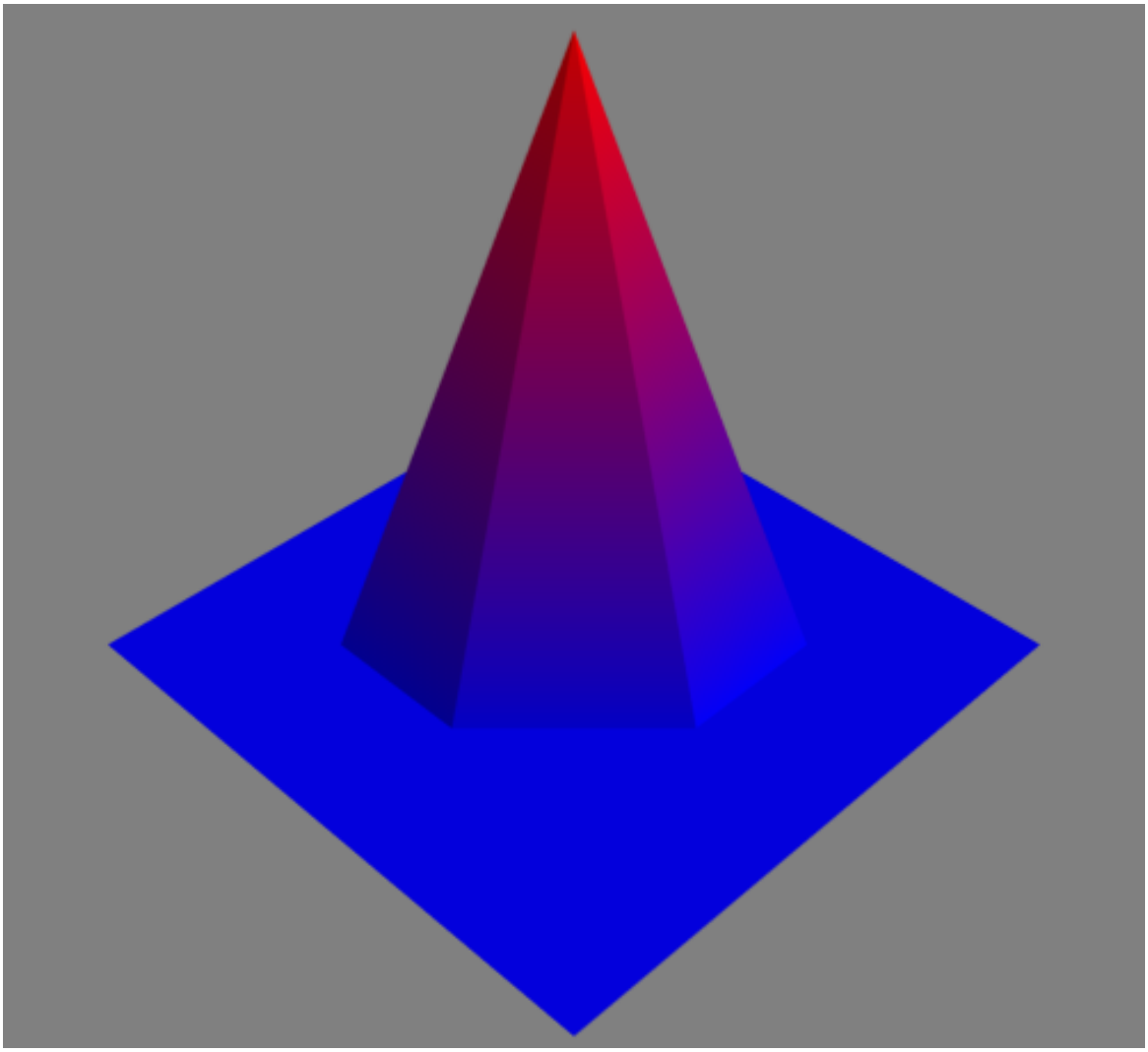


Figure 15.2: A hat function in two dimensions. Compare with the one dimensional hat function shown in Figure 14.2.

Using Triangulations for Finite Element Analysis

As was mentioned before, hat functions on triangulations are often used in Finite Element analysis. When using the Finite element method, regardless of the number of dimensions, it is necessary to transform a problem to its weak formulation. This allows the approximation of the action of a PDE on a domain via the computation of integrals. To begin, we will consider the differential operator

$$-\nabla \cdot (a(x) \nabla u(x)) + b(x) \cdot \nabla u(x) + c(x) u(x) = d(x)$$

on a triangulated domain Ω. For simplicity, we will first consider the case where u is assumed to have Neumann boundary conditions. The weak formulation of this problem on a function space V is to find a function u such that $a(u,v) = l(v)$ for all $v \in V$ with a and l defined as the following integral operators.

$$a(u,v) = \int_\Omega \left(a(x)\nabla u(x) \cdot \nabla v(x) + (b(x) \cdot \nabla u(x))\, v(x) + c(x)\, u(x)\, v(x)\right) dx$$
$$l(v) = \int_\Omega d(x)\, v(x)\, dx$$

We will find an approximate solution to the PDE that lies in the space V of functions that are continuous and linear on each of the triangles in the triangulation of Ω. Let ϕ_i be the hat function centered at the i'th vertex of the triangulation of Ω. The ϕ_i are a basis for V, so we may say that we seek coefficients u_i such that for any set of coefficients v_i,

$$a\left(\sum_i u_i\phi_i, \sum_j v_j\phi_j\right) = l\left(\sum_j v_j\phi_j\right)$$

Since a is linear in its second term and l is linear, this is equivalent to finding coefficients u_i such that

$$a\left(\sum_i u_i\phi_i, \phi_j\right) = l(\phi_j)$$

Since a is also linear in its first term, this is equivalent to finding u_i such that

$$\sum_i u_i a(\phi_i, \phi_j) = l(\phi_j)$$

This problem can be represented as a linear system $Au = \Phi$ where $\Phi_j = l(\phi_j)$ and $A_{j,i} = a(\phi_i, \phi_j)$. Generally speaking, this problem can be solved by construct the matrix A, and the vector Φ, and then solving the resulting system.

> **Note**
>
> The matrix A in the linear system constructed here is often referred to as the "stiffness matrix." The vector Φ is commonly known as the "load vector."

> **Note**
>
> Notice that $a(\phi_i, \phi_j)$ depends entirely on the area where ϕ_i and ϕ_j are both nonzero. If the supports of the functions ϕ_i and ϕ_j do not overlap, $a(\phi_i, \phi_j) = 0$. Since only neighboring hat functions yield nonzero terms in the sum, the matrix A is usually sparse.

Constructing the Stiffness Matrix and Load Vector

It is not always easiest to construct the stiffness matrix and load vector by considering the basis functions one at a time.

16 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.

We will let the basis functions ϕ_i be the Chebyshev polynomials,

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

and the collocation points will be the Gauss-Lobatto points, $x_i = \cos(\pi i/N)$, $i = 0, \ldots, N$. The appropriate solution u_N may be represented with two equivalent forms. First, u_N can be described with the first $N+1$ coefficients $\{a_i\}_{i=0}^N$ of its expansion in the Chebyshev polynomials. Since u_N is a polynomial of order N, it may 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

$$MA = F \tag{16.1}$$

and

$$LU = F \tag{16.2}$$

where

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

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 the cardinal functions:

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

When $L = d/dx$, the matrix corresponding to equation (16.2) 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, x
```

Using the Differentiation Matrix

Problem 1. Use the differentiation matrix to numerically approximate the derivative of $u(x) = e^x \cos(6x)$ on a grid of N Chebychev 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

$$D^2 U = F, \tag{16.3}$$

where $F = [f(x_0), \ldots, f(x_N)]^T$. 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 (16.3) 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

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

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.

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 will update U until the output is all 0's.

guess = None    #Make your guess, same size as the cheb(N) output
solution = root(F, guess).x
```

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

$$Y \cdot (D^2Y) - (DY) \cdot (DY) = I,$$

where I 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.

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

```
from mpl_toolkits.mplot3d import Axes3D
```

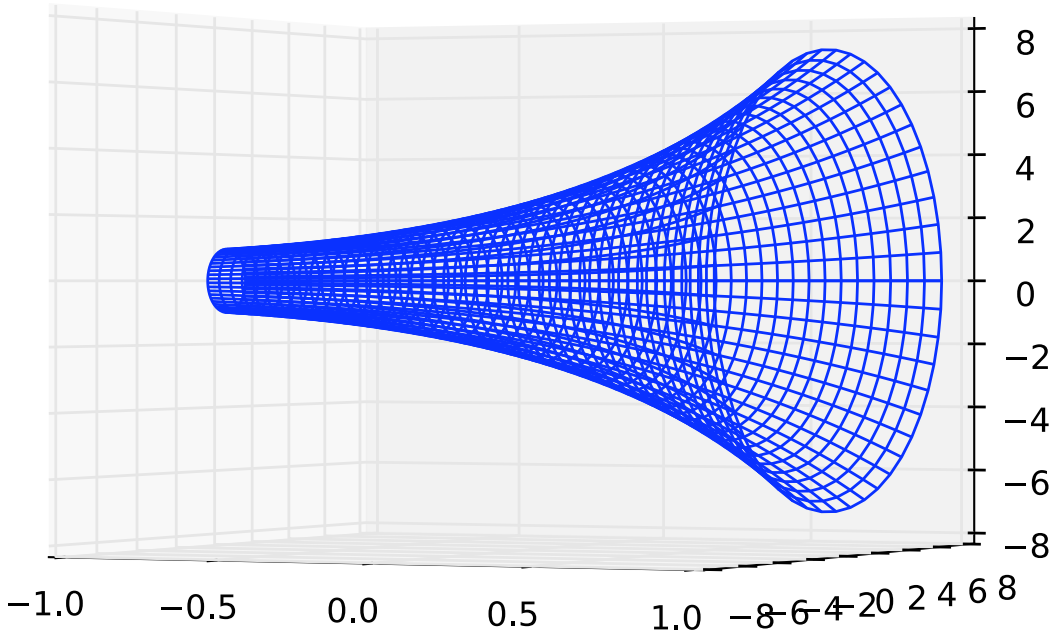


Figure 16.1: The minimal surface corresponding to Problem 5.

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

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.gca(projection="3d")
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.show()
```

17 A Pseudospectral method for periodic functions

Lab Objective: *We look at a pseudospectral method with a Fourier basis, and numerically solve the advection equation 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

$$f(x_j) = \frac{1}{2\pi} \sum_{k=-N/2}^{N/2} \frac{e^{ikx_j}}{c_k} \hat{f}(k), \quad j = 0, \dots, N-1, \tag{17.1}$$

where

$$c_k = \begin{cases} 2 & \text{if } k = -N/2 \text{ or } k = N/2, \\ 1 & \text{otherwise.} \end{cases} \tag{17.2}$$

The inverse DFT can then be used to define a natural interpolant (sometimes called a band-limited interpolant) by evaluating (17.1) at any x rather than x_j:

$$p(x) = \frac{1}{2\pi} \sum_{k=-N/2}^{N/2} e^{ikx} \hat{f}(k). \tag{17.3}$$

The interpolant for f' is then given by

$$p'(x) = \frac{1}{2\pi} \sum_{k=-N/2+1}^{N/2-1} ike^{ikx} \hat{f}(k). \tag{17.4}$$

Consider the function $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$. Using (17.4), the derivative u' may be approximated with the following code. [1] We note that although we only approximate u' at the Fourier grid points, (17.4) 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()
```

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

$$u_t + cu_x = 0 \tag{17.5}$$

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

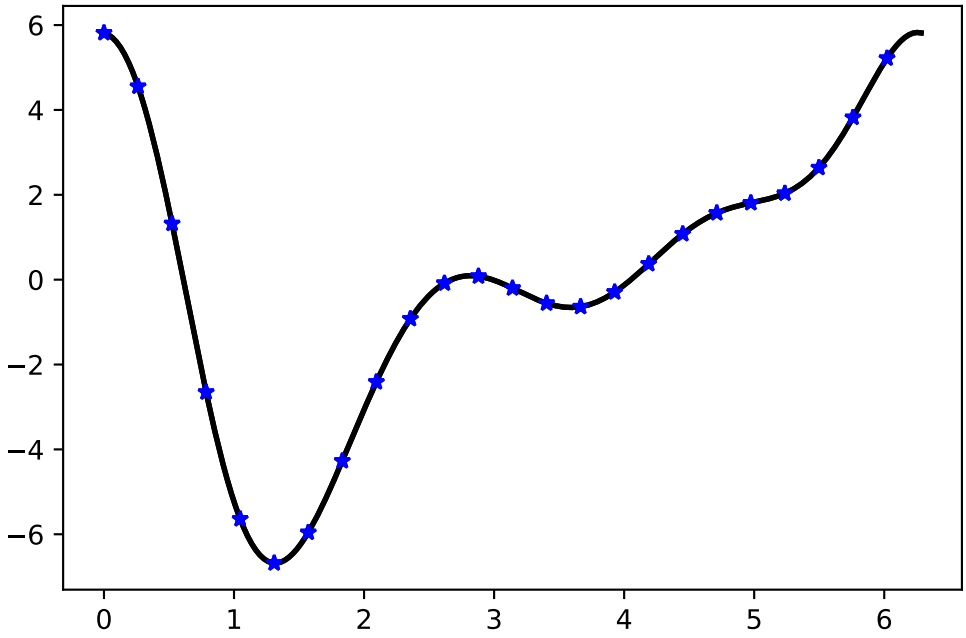


Figure 17.1: The derivative of $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$.

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 circle; this essentially amounts to solving the advection equation on $[0, 2\pi]$ and assuming 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 $[0, \pi]$: given an even N, let $h = 2\pi/N$, so that $\{x_0, \ldots, x_{N-1}\} = \{0, h, 2h, \ldots, 2\pi - h\}$. By using these grid points we obtain the collection of equations

$$u_t(x_j, t) + cu_x(x_j, t) = 0, \quad t > 0, \quad j = 0, \ldots N - 1. \tag{17.6}$$

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

$$U_t + \vec{c}\mathcal{F}^{-1}\left(i\vec{k}\mathcal{F}(U)\right) = 0 \tag{17.7}$$

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.

Problem 2. Using a fourth order Runge-Kutta method (RK4), solve the initial value problem

$$u_t + c(x)u_x = 0, \tag{17.8}$$

where $c(x) = .2 + \sin^2(x-1)$, and $u(x, t=0) = e^{-100(x-1)^2}$. Plot your numerical solution from $t = 0$ to $t = 8$ over 150 time steps and 100 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.

```
t_steps = 150    # Time steps
x_steps = 100     # x steps

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

sol = # RK4 method. Should return a t_steps by x_steps array

X,Y = np.meshgrid(x_domain, t_domain)
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_wireframe(X,Y,sol)
ax.set_zlim(0,3)
plt.show()
```

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

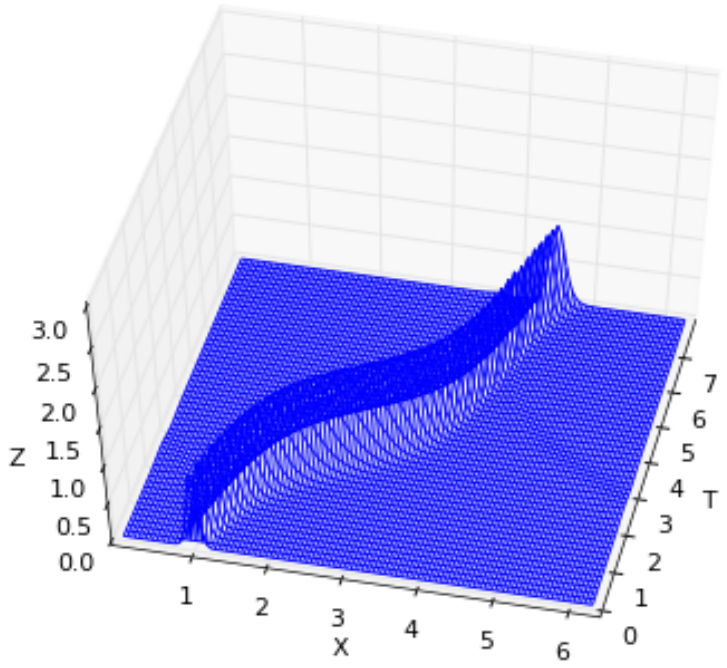


Figure 17.2: The solution of the variable speed advection equation; see Problem 2.

18 Solitons

Lab Objective: *We study traveling wave solutions of the Korteweg-de Vries (KdV) equation, using a pseudospectral discretization in space and a Runge-Kutta integration scheme in time.*

Here we consider soliton solutions of the Korteweg-de Vries (KdV) equation. This equation is given by

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0.$$

The KdV equation is a canonical equation 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 of the soliton depends on the speed of the wave. Solitons are traveling waves in the shape of a pulse, they are nonlinearly stable (bumped waves return to their previous shape), and they maintain their energy as they travel. They also enjoy an additional stability property: they play well with others. Two interacting solitons will maintain their shape after crossing paths.

Numerical solution

Consider the KdV equation on $[-\pi, \pi]$, together with an appropriate initial condition:

$$u_t = -\left(\frac{u^2}{2}\right)_x - u_{xxx},$$
$$u(x,0) = u_0(x).$$

We will use initial data that is zero at the endpoints. This will allow us to use a pseudospectral method for periodic initial data to find a numerical approximation for the solution $u(x,t)$.

If we use N subintervals in space, we then obtain the spatial step $h = 2\pi/N$ and the grid points $\{x_j\}_{j=1} = \{-\pi, -\pi + h, \ldots, \pi - h\}$. Let $\mathcal{F}(u)(t) = \hat{u}(t)$ denote the Fourier transform of $u(x,t)$ (in space), so that

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

Similarly we let $\mathcal{F}^{-1}$ represent the discrete inverse Fourier transform. Recall that k represents the wave numbers in Fourier space; our code defines it by

```
# Array of wave numbers.  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])  ,
                     np.arange(-N/2+1,0,1)  )).reshape(N,)
```

We now apply the Fourier transform to the KdV equation. In Fourier space, we obtain

$$\mathcal{F}(u)_t = -\frac{ik}{2}\mathcal{F}(u^2) - (ik)^3\mathcal{F}(u).$$

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

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

Similarly we let $\mathcal{F}^{-1}$ represent the discrete inverse Fourier transform. Using the pseudospectral approximation in space leads to the system of ODEs

$$\mathcal{F}(U)_t = -\frac{i}{2}\vec{k}\mathcal{F}\left(\mathcal{F}^{-1}(\mathcal{F}(U))^2\right) + i\vec{k}^3\mathcal{F}(U) \tag{18.1}$$

where $\vec{k}$ is a vector, and multiplication is done element-wise. In terms of $Y = \mathcal{F}(U)$, this simplifies to

$$Y_t = -\frac{i}{2}\vec{k}\mathcal{F}\left(\mathcal{F}^{-1}(Y)^2\right) + i\vec{k}^3 Y \tag{18.2}$$

and is implemented below.

```
# Defines the left hand side of the ODE y' = G(t,y)
# defined above.
ik3 = 1j*k**3.
def G_unscaled(t,y):
    out = -.5*1j*k*fft(ifft(y,axis=0)**2.,axis=0)  + ik3*y
    return out
```

Equation (18.2) is solved below, using a soliton as initial data for the KdV equation. Note that the Fourier transform must be applied to the soliton before solving, and that the final numerical solution must be transformed back from Fourier space before plotting.

```
N = 256
x = (2.*np.pi/N)*np.arange(-N/2,N/2).reshape(N,1)   # Space discretization
s, shift = 25.**2., 2.                              # Initial data is a soliton
y0 = (3.*s*np.cosh(.5*(sqrt(s)*(x+shift)))**(-2.)).reshape(N,)

# Solves the ODE.
max_t = .0075
dt = .2*N**(-2.)
```

```
max_tsteps = int(round(max_t/dt))
y0 = fft(y0,axis=0)
T,Y = RK4(G_unscaled, y0, t0=0, t1=max_t, n=max_tsteps)

# Using the variable stride, we step through the data,
# applying the inverse fourier transform to obtain u.
# These values will be plotted.
stride = int(np.floor((max_t/25.)/dt))
uvalues, tvalues = np.real(ifft(y0,axis=0)).reshape(N,1), np.array(0.).reshape↩
    (1,1)
for n in range(1,max_tsteps+1):
    if np.mod(n,stride) == 0:
        t = n*dt
        u = np.real( ifft(Y[n], axis=0) ).reshape(N,1)
        uvalues = np.concatenate((uvalues,np.nan_to_num(u)),axis=1)
        tvalues = np.concatenate((tvalues,np.array(t).reshape(1,1)),axis=1)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.view_init(elev=45., azim=150)
tv, xv = np.meshgrid(tvalues,x,indexing='ij')
surf = ax.plot_surface(tv,xv, uvalues.T, rstride=1, cstride=1, cmap=cm.coolwarm↩
    ,
                       linewidth=0, antialiased=False)
tvalues = tvalues[0]; ax.set_xlim(tvalues[0], tvalues[-1])
ax.set_ylim(-pi, pi); ax.invert_yaxis()
ax.set_zlim(0., 4000.)
ax.set_xlabel('T'); ax.set_ylabel('X'); ax.set_zlabel('Z')
plt.show()
```

The method we have used requires the use of an algorithm for (ODE) initial value problems, such as the RK4 algorithm. The RK4 method is implemented below.

```
def initialize_all(y0, t0, t1, n):
    """ An initialization routine for the different ODE solving
    methods in the lab. This initializes Y, T, and h. """

    if isinstance(y0, np.ndarray):
        Y = np.empty((n, y0.size),dtype=complex).squeeze()
    else:
        Y = np.empty(n,dtype=complex)
    Y[0] = y0
    T = np.linspace(t0, t1, n)
    h = float(t1 - t0) / (n - 1)
    return Y, T, h

def RK4(f, y0, t0, t1, n):
    """ Use the RK4 method to compute an approximate solution
    to the ODE y' = f(t, y) at n equispaced parameter values from t0 to t
```

```
    with initial conditions y(t0) = y0.

    'y0' is assumed to be either a constant or a one-dimensional numpy array.
    't0' and 't1' are assumed to be constants.
    'f' is assumed to accept two arguments.
    The first is a constant giving the current value of t.
    The second is a one-dimensional numpy array of the same size as y.

    This function returns an array Y of shape (n,) if
    y is a constant or an array of size 1.
    It returns an array of shape (n, y.size) otherwise.
    In either case, Y[i] is the approximate value of y at
    the i'th value of np.linspace(t0, t, n).
    """
    Y, T, h = initialize_all(y0, t0, t1, n)
    for i in xrange(1, n):
        K1 = f(T[i-1], Y[i-1])
        tplus = (T[i] + T[i-1]) * .5
        K2 = f(tplus, Y[i-1] + .5 * h * K1)
        K3 = f(tplus, Y[i-1] + .5 * h * K2)
        K4 = f(T[i], Y[i-1] + h * K3)
        Y[i] = Y[i-1] + (h / 6.) * (K1 + 2 * K2 + 2 * K3 + K4)
    return T, Y
```

Problem 1. Run the code above to 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 $[0, .0075]$. The solution is shown in Figure 18.1.

Problem 2. 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 $s_2 = 16^2$, $a_1 = 1$.[a] Solve on the time domain $[0, .0075]$. The solution is shown in Figure 18.2.

[a] This problem is solved in *Spectral Methods in MATLAB*, by Trefethen.

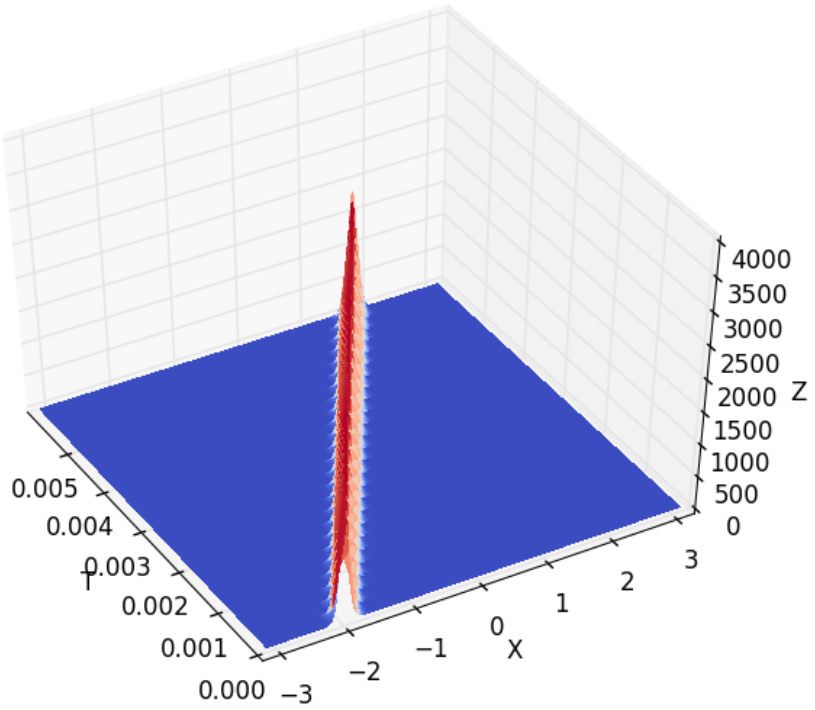


Figure 18.1: The solution to Problem 1.

Problem 3. Consider again equation (18.2). The linear term in this equation is $i\vec{k}^3 Y$. This term contributes much of the exponential growth in the ODE, and responsible for how short the time step must be to ensure numerical stability. Make the substitution $Z = e^{-ik^3 t} Y$ and find a similar ODE for Z. This essentially allows the exponential growth to be scaled out (it's solved for analytically). Use the resulting equation to solve the previous problem.

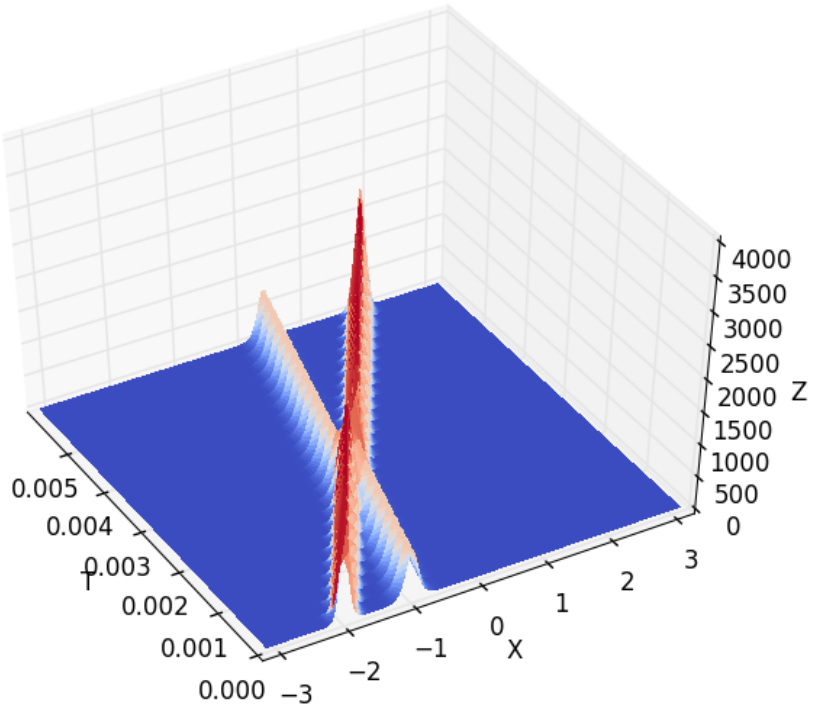


Figure 18.2: The solution to Problem 2.

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

$$\begin{aligned} s(t) &= \langle x(t), y(t)\rangle, \quad t \in [0, T], \\ s'(t) &= \langle x'(t), y'(t)\rangle, \\ &= \langle \cos\theta(x(t)), \sin\theta(x(t))\rangle + \langle 0, c(x(t))\rangle. \end{aligned}$$

Here $\langle \cos\theta, \sin\theta\rangle$ represents the motion of the boat due to the rower, and $\langle 0, c\rangle$ 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

$$\begin{aligned} y'(x) &= \frac{y'(t)}{x'(t)}, \\ &= \frac{\sin\theta + c}{\cos\theta}, \\ &= c\sec\theta + \tan\theta. \end{aligned} \tag{19.1}$$

The time T required to cross the river is given by

$$\begin{aligned} T &= \int_{-1}^{1} t'(x)\,dx, \\ &= \int_{-1}^{1} \frac{1}{x'(t)}\,dx \\ &= \int_{-1}^{1} \sec\theta(x)\,dx. \end{aligned} \tag{19.2}$$

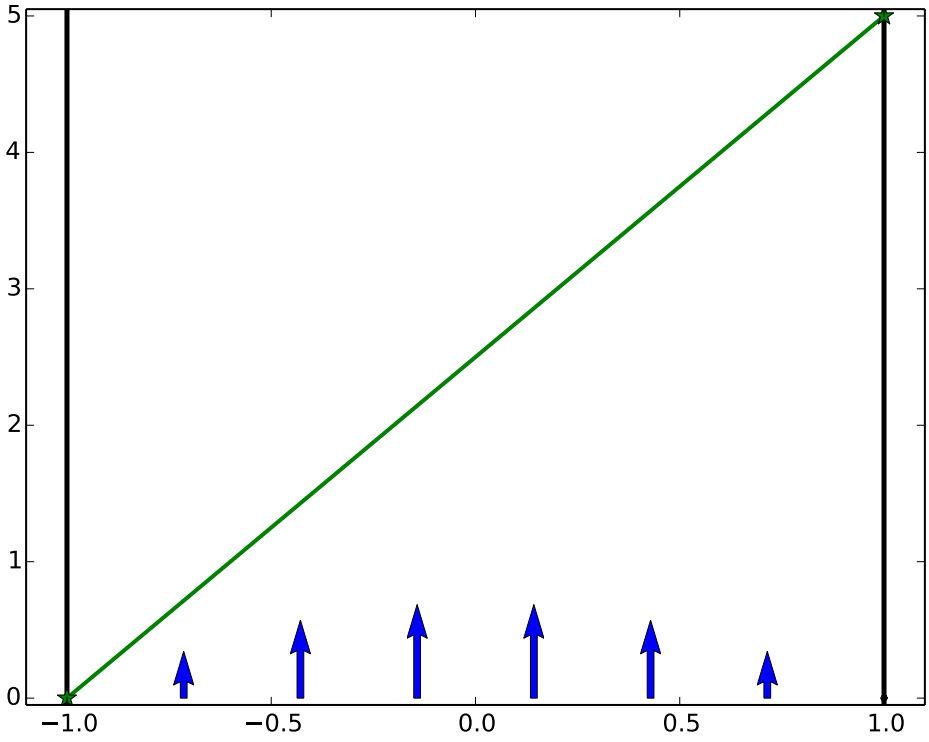


Figure 19.1: The river's current, along with a possible trajectory for the boat.

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

$$\begin{aligned} T[y] &= \int_{-1}^{1} \sec\theta \, dx, \\ &= \int_{-1}^{1} \frac{1}{1-c^2}(c\tan\theta + \sec\theta - c^2\sec\theta - c\tan\theta)\, dx, \\ &= \int_{-1}^{1} \frac{1}{1-c^2}(c\tan\theta + \sec\theta - cy')\, dx. \end{aligned}$$

Since

$$\begin{aligned} c\tan\theta + \sec\theta &= \sqrt{1-c^2+(c\sec\theta+\tan\theta)^2}, \\ &= \sqrt{1-c^2+(y')^2}, \end{aligned}$$

we obtain at last

$$T[y] = \int_{-1}^{1} \left[\alpha(x)\sqrt{1+(\alpha y')^2(x)} - (\alpha^2 c y')(x)\right] dx, \tag{19.3}$$

where $\alpha = (1-c^2)^{-1/2}$.

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 a Python function that accepts as arguments a function y, its derivative y', and an x-value, and returns $L(x, y(x), y'(x))$ (where $T[y] = \int_{-1}^{1} L(x, y(x), y'(x))$). Use that function to define a second function that numerically computes $T[y]$ for a given path $y(x)$.

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.

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

$$\frac{d}{dx} L_{y'} = \frac{d}{dx} \left(\alpha^3(x) y'(x) (1 + (\alpha y')^2(x))^{-1/2} - \alpha^2(x) c \right) = 0. \tag{19.4}$$

Problem 3. Numerically solve the Euler-Lagrange equation (19.4), using $c(x) = -\frac{7}{10}(x^2 - 1)$ and $\alpha = (1 - c^2)^{-1/2}$, and $y(-1) = 0$, $y(1) = 5$.

Hint: Since this boundary value problem is defined over the doimain $[-1, 1]$, 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.

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.)

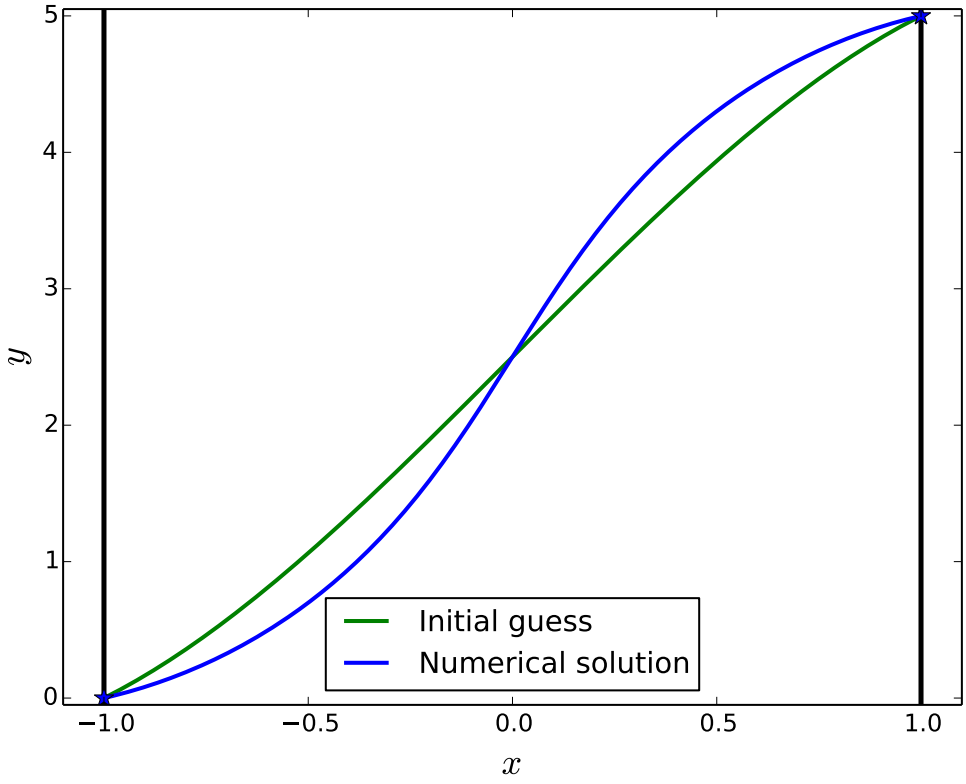


Figure 19.2: Numerical computation of the trajectory with the shortest transit time.

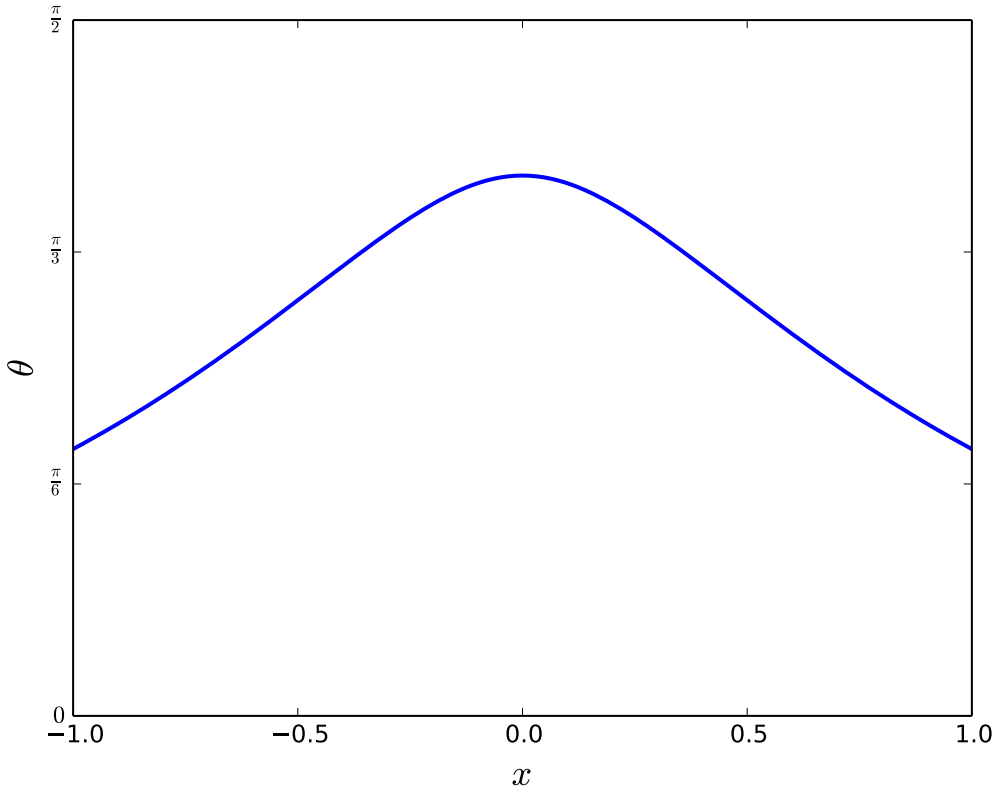


Figure 19.3: The optimal angle to steer the boat.

20 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:

$$\begin{aligned} &u_t = -u_{xx}, \quad (x,t) \in (0,L) \times (0,\infty), \\ &u(0,t) = u(L,t) = 0, \quad t \in (0,\infty), \\ &u(x,0) = f(x), \quad x \in (0,L). \end{aligned} \tag{20.1}$$

For the initial data $f(x)$ 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.

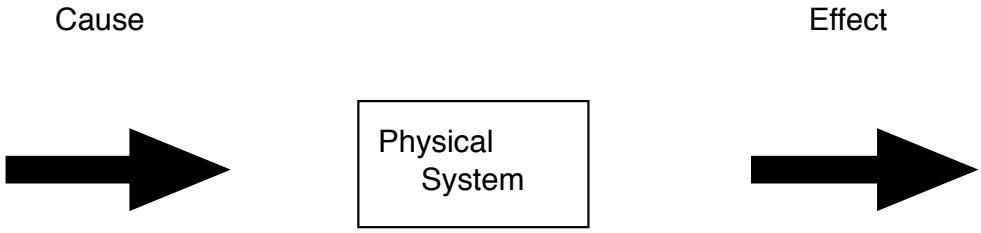


Figure 20.1: Cause and effect within a given physical system.

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. The direct problem is usually studied first historically.

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 20.1. 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]:

$$\begin{cases} -(au')' = f, & x \in (0,1), \\ a(0)u'(0) = c_0, & a(1)u'(1) = c_1. \end{cases} \tag{20.2}$$

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 density 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 density $a(x)$ 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 density of the rod? This is an example of a *parameter estimation problem.*

Let us consider a numerical method for solving (20.2) for the density $a(x)$. 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 $a^h(x)$ of the form

$$a^h = \sum_{j=0}^{N} \alpha_j \phi_j, \quad \alpha_j = a(x_j). \tag{20.3}$$

Integrating (20.2) from 0 to x, we obtain

$$\begin{aligned} \int_0^x -(au')'\,ds &= \int_0^x f(s)\,ds, \\ -\left[a(x)u'(x) - c_0\right] &= \int_0^x f(s)\,ds, \\ u'(x) &= \frac{c_0 - \int_0^x f(s)\,ds}{a(x)}. \end{aligned} \tag{20.4}$$

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 (20.3) can now be approximated by minimizing

$$\sum_{j=0}^{N} \left(\frac{c_0 - \int_0^{x_j} f(s)\,ds}{\alpha_j} - u'(x_j) \right)^2.$$

Problem 1. Solve (20.2) for $a(x)$ using the following conditions:
$c_0 = 3/8$, $c_1 = 5/4$, $u(x) = x^2 + x/2 + 5/16$, $x_j = .1j$ for $j = 0, 1, \ldots, 10$, and

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

Produce the plot shown in Figure 20.2.

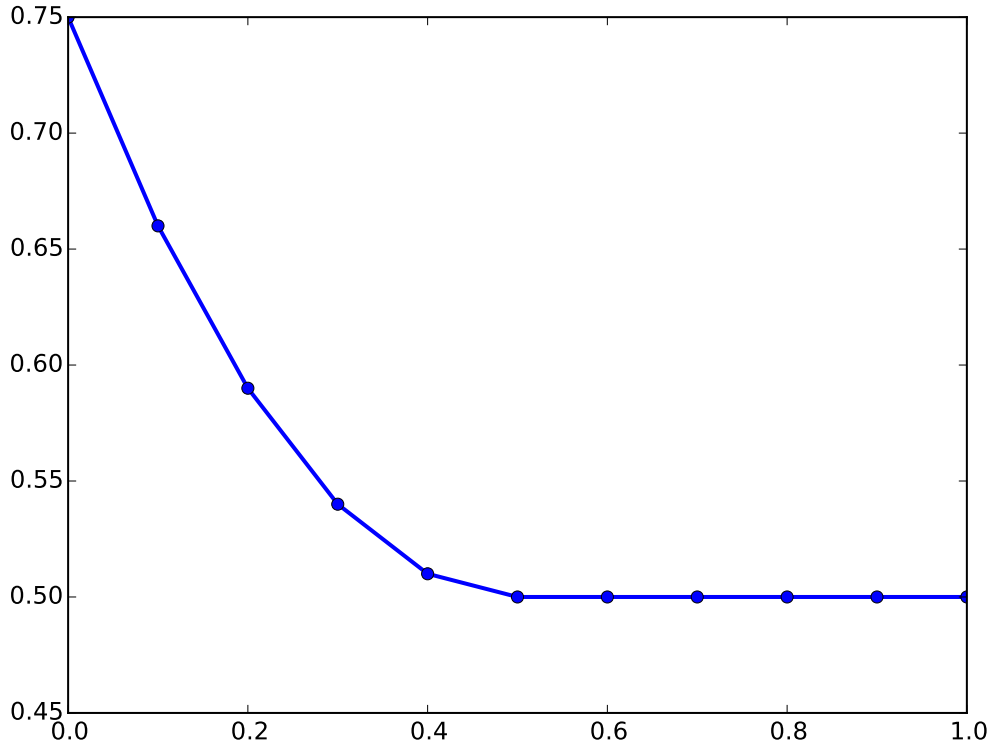


Figure 20.2: The solution $a(x)$ to Problem 1

Hint: use the `minimize` function in `scipy.optimize` and some initial guess to find the a_j.

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

$$\begin{cases} -(au')' = -1, & x \in (0,1), \\ a(0)u'(0) = 1, & a(1)u'(1) = 2. \end{cases} \tag{20.5}$$

where $u(x) = x + 1 + \varepsilon \sin(\varepsilon^{-2}x)$. Using several values of $\varepsilon > 0.66049142$, plot the corresponding density $a(x)$ for x in `np.linspace(0,1,11)` to demonstrate that the problem is ill-posed.

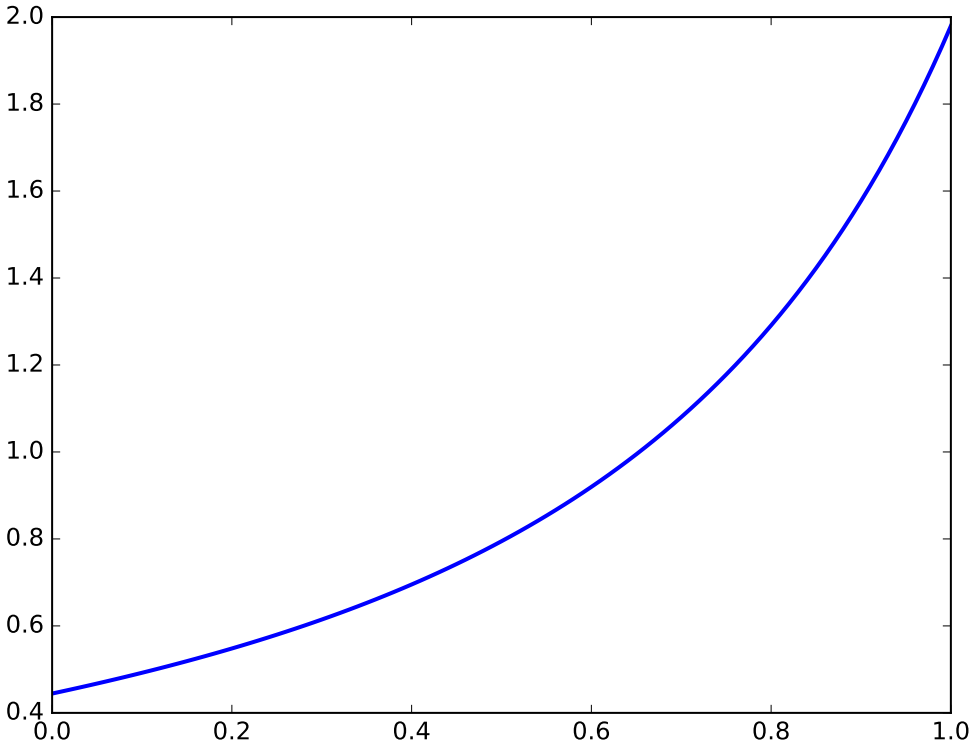


Figure 20.3: The density function $a(x)$ satisfying (20.5) for $\varepsilon = .8$.

21 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

$$L_u - \sum_{i=1}^{n} \frac{\partial L_{u_{x_i}}}{\partial x_i} = L_u - \nabla \cdot L_{\nabla u} = L_u - \operatorname{div}(L_{\nabla u}) = 0. \tag{21.1}$$

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 (21.1), we consider a related parabolic PDE,

$$\begin{aligned} u_t &= -(L_y - \operatorname{div} L_{\nabla u}), \quad t > 0, \\ u(x, 0) &= u_0(x), \quad t = 0. \end{aligned} \tag{21.2}$$

A steady state solution of (21.2) does not depend on time, and thus solves the Euler-Lagrange equation. It is often easier to evolve an initial guess using (21.2), and stop whenever its steady state is well-approximated, than to solve (21.1) directly.

Example 21.1. Consider the energy functional

$$J[u] = \int_\Omega \|\nabla u\|^2\, dx.$$

The minimizing function u^* satisfies the Euler-Lagrange equation

$$-\operatorname{div} \nabla u = -\triangle u = 0.$$

The gradient descent flow is the well-known heat equation

$$u_t = \triangle u.$$

■

The Euler-Lagrange equation could equivalently be described as $\triangle u = 0$, leading to the PDE $u_t = -\triangle u$. 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 (21.2) more rigorous. We recall that

$$\begin{aligned}\delta J(u;h) &= \frac{d}{dt} J(u+\varepsilon h)\Big|_{\varepsilon=0}, \\ &= \int_\Omega (L_y(u) - \operatorname{div} L_{\nabla u}(u))h\,dx, \\ &= \langle L_y(u) - \operatorname{div} L_{\nabla u}(u), h\rangle_{L^2(\Omega)},\end{aligned}$$

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 $h = \alpha(L_y(u) - \operatorname{div} L_{\nabla u}(u))$ for some $\alpha \in \mathbb{R}$. This implies that the "direction" $h = L_y(u) - \operatorname{div} L_{\nabla u}(u)$ is the direction of steepest ascent and maximizes $\delta J(u;h)$. Similarly,

$$h = -(L_y(u) - \operatorname{div} L_{\nabla u}(u))$$

points in the direction of steepest descent, and the flow described by (21.2) 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 $y(x)$ about the x-axis is

$$A[y] = \int_a^b 2\pi y\sqrt{1+(y')^2}\,dx.$$

To minimize the functional A over the collection of smooth curves with fixed end points $y(a) = y_a$, $y(b) = y_b$, we use the Euler-Lagrange equation

$$\begin{aligned}0 &= 1 - y\frac{y''}{1+(y')^2}, \\ &= 1 + (y')^2 - yy'',\end{aligned} \tag{21.3}$$

with the gradient descent flow given by

$$\begin{aligned}&u_t = -1 - (y')^2 + yy'', \quad t > 0,\ x \in (a,b), \\ &u(x,0) = g(x), \quad t = 0, \\ &u(a,t) = y_a, \quad u(b,t) = y_b.\end{aligned} \tag{21.4}$$

Numerical Implementation

We will construct a numerical solution of (21.4) using the conditions $y(-1) = 1$, $y(1) = 7$. 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 $\frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}$. 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])


# Time step until successive iterations are close
iteration = 0
while iteration < time_steps:
    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
```

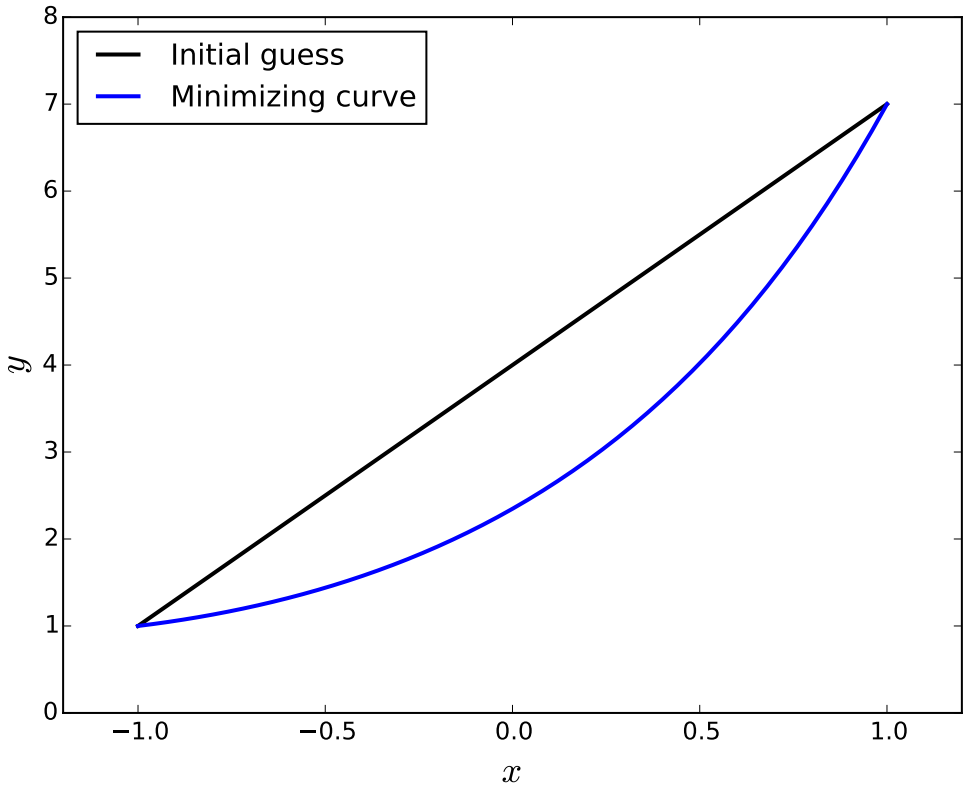


Figure 21.1: The solution of (21.3), found using the gradient descent flow (21.4).

Problem 1. Using 20 x steps, 250 time steps, and a final time of .2, plot the solution that minimizes (21.4). It should match figure 21.1.

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 random_integers, uniform, randn
import matplotlib.pyplot as plt
from matplotlib import cm
from imageio import imread, imwrite

imagename = 'baloons_resized_bw.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, as_gray=True) * (1. / 255)
IM_x, IM_y = IM.shape

for lost in xrange(changed_pixels):
    x_,y_ = random_integers(1,IM_x-2), random_integers(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

$$J[u] = \int_\Omega L(x, u, \nabla u)\, dx, \tag{21.5}$$

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 (21.5) is

$$\begin{aligned} L_u - \text{div } L_{\nabla u} &= (u - f) - \lambda \triangle u, \\ &= 0. \end{aligned}$$

and the gradient descent flow is

$$\begin{aligned} u_t &= -(u - f - \lambda \triangle u), \\ u(x, 0) &= f(x). \end{aligned} \tag{21.6}$$

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 $\triangle u$ with centered differences:

$$\begin{aligned} u_t &\approx \frac{u_{ij}^{n+1} - u_{ij}^n}{\triangle t}, \\ u_{xx} &\approx \frac{u_{i+1,j}^n - 2u_{ij}^n + u_{i-1,j}^n}{\triangle x^2}, \\ u_{yy} &\approx \frac{u_{i,j+1}^n - 2u_{ij}^n + u_{i,j-1}^n}{\triangle y^2}. \end{aligned}$$

Original image

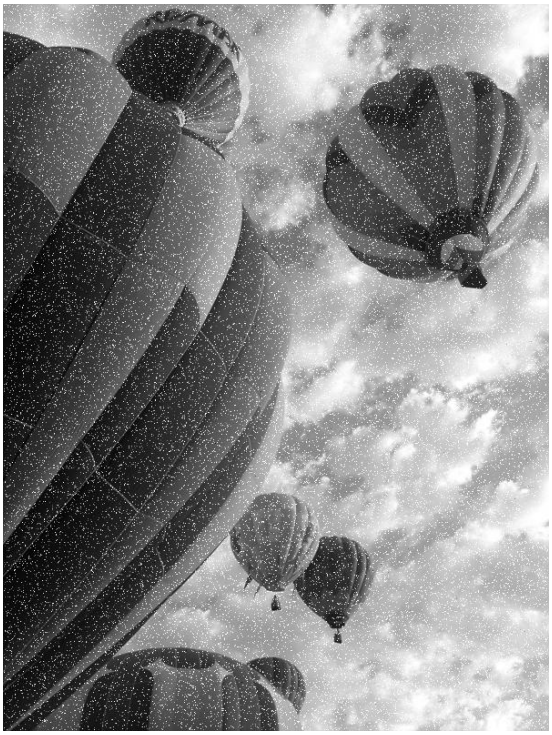


Image with white noise

Figure 21.2: Noise.

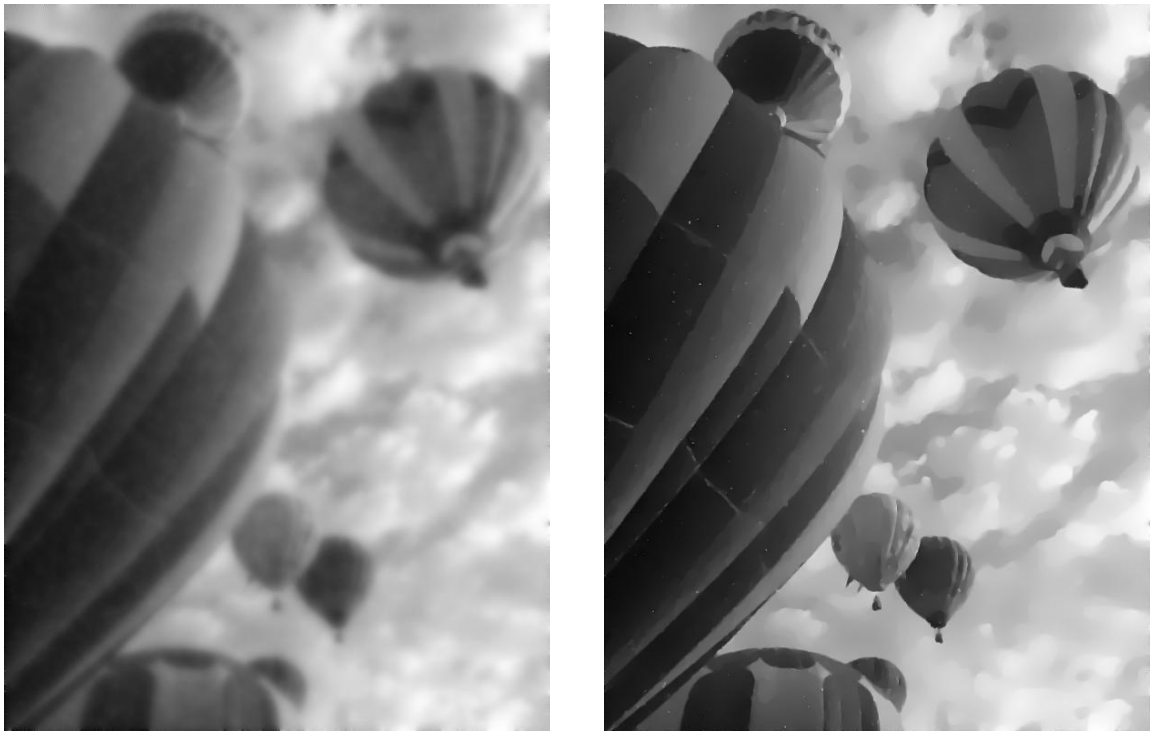
Problem 2. Using $\triangle t = 1e{-}3$, $\lambda = 40$, $\triangle x = 1$, and $\triangle y = 1$, implement the numerical scheme mentioned above to obtain a solution u. (So $\Omega = [0, n_x] \times [0, n_y]$, where n_x and n_y represent the number of pixels in the x and y dimensions, respectively.) Take 250 steps in time. Compare your results with Figure 21.3.

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 21.3: The solutions of (21.6) and (21.11), 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

$$\int_\Omega |\nabla u(x)|\,dx \tag{21.7}$$

subject to the constraints

$$\int_\Omega u(x)\,dx = \int_\Omega f(x)\,dx, \tag{21.8}$$

$$\int_\Omega |u(x) - f(x)|^2\,dx = \sigma|\Omega|. \tag{21.9}$$

Intuitively, (21.7) penalizes fast variations in f - this functional together with the constraint (21.8) 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 (21.9) specifying how far $u(x)$ is required to differ from the noisy image f. More precisely, (21.8) specifies that the noise in the image has zero mean, and (21.9) 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

$$F[u] = \min_{u\in BV(\Omega)} \int_\Omega |\nabla u| + \frac{\lambda}{2}(u-f)^2\,dx, \tag{21.10}$$

[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 (21.5): $\int_\Omega |\nabla u|$ 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

$$\begin{aligned} u_t &= -\lambda(u-f) + \frac{u_{xx}u_y^2 + u_{yy}u_x^2 - 2u_x u_y u_{xy}}{(u_x^2+u_y^2)^{3/2}}, \\ u(x,0) &= f(x). \end{aligned} \tag{21.11}$$

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 $\triangle t = 1e-3, \lambda = 1, \triangle x = 1$, and $\triangle y = 1$, implement the numerical scheme mentioned above to obtain a solution u. Take 200 steps in time. Compare your results with Figure 21.3. 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.

22 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

$$J[\theta] = \int_{t_0}^{t_1} L, \tag{22.1}$$

where the Lagrangian $L = T - U$ is the difference between the kinetic and potential energies of the bob.

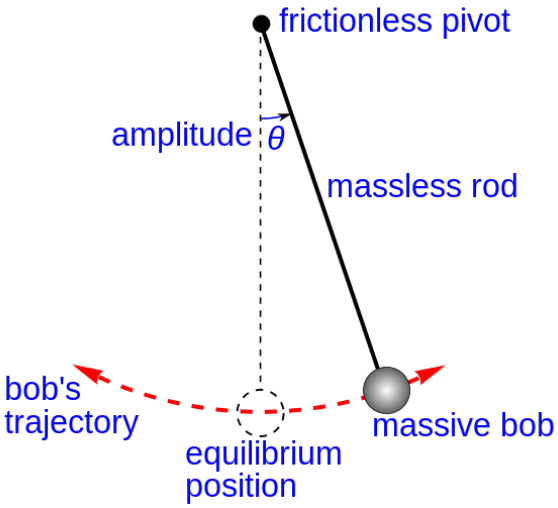


Figure 22.1: The frame of reference for the simple pendulum problem.

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

$$\begin{aligned} T &= \frac{m}{2}v^2 = \frac{m}{2}(\dot{x}^2 + \dot{y}^2), \\ &= \frac{m}{2}((l\cos(\theta)\dot{\theta})^2 + (l\sin(\theta)\dot{\theta})^2), \\ &= \frac{ml^2\dot{\theta}^2}{2}. \end{aligned} \tag{22.2}$$

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:

$$\begin{aligned} 0 &= L_\theta - \frac{d}{dx}L_{\dot{\theta}}, \\ &= -mgl\sin\theta - ml^2\ddot{\theta}, \\ &= \ddot{\theta} + \frac{g}{l}\sin\theta. \end{aligned} \tag{22.3}$$

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).

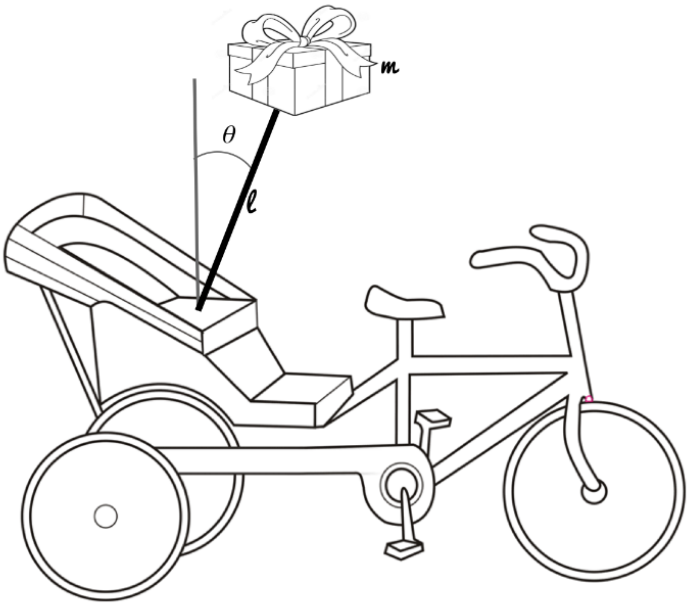


Figure 22.2: The inverted pendulum problem on a mobile rickshaw with a present suspended above.

By Hamilton's Principle, the path (x, θ) of the rickshaw and the present minimizes the functional

$$J[x, \theta] = \int_{t_0}^{t_1} L, \tag{22.4}$$

where the Lagrangian $L = T - U$ is the difference between the kinetic energy of the present on the pendulum, and its potential energy.

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

$$\begin{aligned} T &= \frac{1}{2}Mv_1^2 + \frac{1}{2}mv_2^2, \\ &= \frac{1}{2}M\dot{x}^2 + \frac{1}{2}m((\dot{x} - l\dot{\theta}\cos\theta)^2 + (-l\dot{\theta}\sin\theta)^2), \\ &= \frac{1}{2}(M+m)\dot{x}^2 + \frac{1}{2}ml^2\dot{\theta}^2 - ml\dot{x}\dot{\theta}\cos\theta. \end{aligned} \tag{22.5}$$

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:

$$\begin{aligned} \frac{\partial L}{\partial x} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}} &= F, \\ \frac{\partial L}{\partial \theta} - \frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} &= 0. \end{aligned} \tag{22.6}$$

After expanding (22.6) we see that $x(t)$ and $\theta(t)$ satisfy

$$\begin{aligned} F &= (M+m)\ddot{x} - ml\ddot{\theta}\cos\theta + ml\dot{\theta}^2\sin\theta, \\ l\ddot{\theta} &= g\sin\theta + \ddot{x}\cos\theta. \end{aligned} \tag{22.7}$$

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 (22.7) about $(\theta, \dot{\theta}) = (0, 0)$, obtaining the equations

$$\begin{aligned} F &= (M+m)\ddot{x} - ml\ddot{\theta}, \\ l\ddot{\theta} &= g\theta + \ddot{x}. \end{aligned}$$

These equations can be further manipulated to obtain

$$\begin{aligned} \ddot{x} &= \frac{1}{M}F - \frac{m}{M}g\theta, \\ \ddot{\theta} &= \frac{1}{Ml}F + \frac{g}{Ml}(M+m)\theta. \end{aligned} \tag{22.8}$$

We will now write (22.8) 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

$$\begin{aligned} J[z] &= \int_0^\infty (q_1 x_1^2 + q_2 x_2^2 + q_3 \theta_1^2 + q_4 \theta_2^2 + r u^2)\, dt \\ &= \int_0^\infty z^T Q z + u^T R u\, dt \end{aligned} \tag{22.9}$$

where q_1, q_2, q_3, q_4, and r are nonnegative weights, and

$$Q = \begin{bmatrix} q_1 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 \\ 0 & 0 & q_3 & 0 \\ 0 & 0 & 0 & q_4 \end{bmatrix}, R = \begin{bmatrix} r \end{bmatrix}.$$

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
    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 (22.9) 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:

$$\tilde{u} = -R^{-1}B^T P\tilde{z}.$$

Here P is a matrix function that satisfies the Riccati differential equation (RDE)

$$\dot{P}(t) = PA + A^T P + Q - PBR^{-1}B^T P.$$

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)

$$PA + A^T P + Q - PBR^{-1}B^T P = 0. \tag{22.10}$$

The evolution of the optimal state vector $\tilde{z}$ can then be described by [1]

$$\dot{\tilde{z}} = (A - BR^{-1}B^T P)\tilde{z}. \tag{22.11}$$

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 `np.reshape(16)` and `np.reshape((4,4))`.

```
def find_P(A, B, Q, R):
    '''
    Parameters:
    ----------
    A, Q    : ndarrays of shape (4,4)
```

[1] See Calculus of Variations and Optimal Control Theory, Daniel Liberzon, Ch.6

```
    B       : ndarray of shape (4,1)
    R       : ndarray of shape (1,1)

    Returns
    -------
    P       : the matrix solution of the Riccati equation
    '''
    pass
```

Using the values

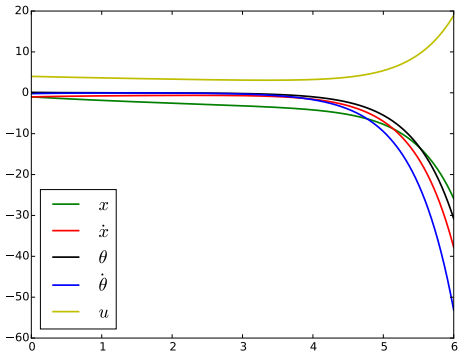
```
M, m = 23., 5.
l = 4.
q1, q2, q3, q4 = 1., 1., 1., 1.
r = 10.
```

compute the eigenvalues of $A - BR^{-1}B^TP$. Are any of the eigenvalues positive? Consider differential equation (22.11) governing the optimal state $\tilde{z}$. Using this value of P, will we necessarily have $\tilde{z} \to 0$?

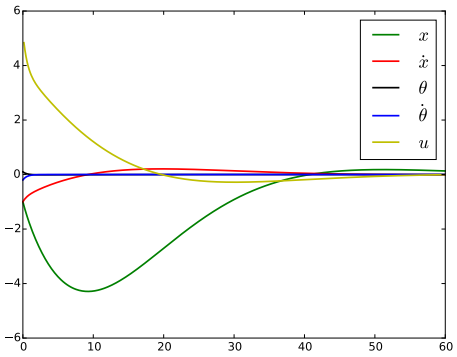
Notice that we have no information on how many solutions (22.10) possesses. In general there may be many solutions. We hope to find a unique solution P that is *stabilizing*: the eigenvalues of $A - BR^{-1}B^TP$ have negative real part. To find this P, use the function `solve_continuous_are` from `scipy.linalg`. This function is designed to solve the continuous algebraic Riccati equation.

Problem 3. Write the following function that implements the LQR solution described earlier. For the IVP solver, you can use your own or you may use the function `odeint` from `scipy.integrate`.

```
def rickshaw(tv, X0, A, B, Q, R, P):
    '''
    Parameters:
    ----------
    tv  : ndarray of time values, with shape (n+1,)
    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
```



P is found using `scipy.optimize.root`.



P is found using `solve_continuous_are`.

Figure 22.3: 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 $\tilde{u}$. Compare your results as shown in Figure 22.3.

23 Optimal Reentry of a Spacecraft

Lab Objective: *We consider the problem of minimizing the heating experienced by a spacecraft during reentry. The boundary value problem (BVP) associated with the reentry of a spacecraft is inherently challenging: the craft must descend quickly enough to enter the atmosphere, but pull out soon enough to prevent overheating or crashing. Problems involving variational calculus and optimal control often include the numerical solution of a challenging BVP.*

A fundamental topic considered in aerospace engineering is the process of landing a spacecraft. Landing a spacecraft requires a massive reduction in the kinetic energy of the craft. That reduction can be accomplished either through the use of massive quantities of fuel (very expensive), or by transforming kinetic energy into heat. That heat must then be absorbed by the atmosphere and the spacecraft. The question then is how to choose the optimal path for reentry into the atmosphere, where the total heating experienced by the craft is minimized.

We begin with a control system[1] that describes the path of a spacecraft through the atmosphere (we assume the spacecraft is similar to the Apollo craft). The dependent variables are the velocity v of the spacecraft, the angle γ of the flight path, and the normalized altitude $\xi = h/R$ above the Earth's surface, where R is the radius of the Earth and h is the altitude of the spacecraft above the Earth. The control variable u represents the angle of attack of the spacecraft. The flight path is given by

$$\begin{aligned}
\dot{v} &= -s\rho v^2 C_D(u) - \frac{g\sin(\gamma)}{(1+\xi)^2},\\
\dot{\gamma} &= s\rho v C_L(u) + \frac{v\cos(\gamma)}{R(1+\xi)} - \frac{g\cos\gamma}{v(1+\xi)^2},\\
\dot{\xi} &= \frac{v\sin\gamma}{R}.
\end{aligned} \tag{23.1}$$

Coefficients C_D and C_L represent drag and lift coefficients, and depend on the angle of attack:

$$\begin{aligned}
C_D(u) &= 1.174 - .9\cos u,\\
C_L(u) &= 0.6\sin u.
\end{aligned}$$

The atmospheric density ρ is a function of height,

$$\rho(\xi) = \rho_0 e^{-R\beta\xi},$$

[1]This control problem and its numerical solution are thoroughly described in 'Introduction to Numerical Analysis' by J. Stoer, R. Bulirsch (pg 524). We will mirror their presentation throughout this lab.

Figure 23.1: Apollo 8 during launch

where ρ_0 is the atmospheric density at the surface of the earth. Other parameters include the force of gravity g, and $s = \frac{1}{2}S/m$, where S is the frontal area of the craft and m is its mass. The numerical values we will use are coded below, along with the drag and lift functions.

```
from __future__ import division
from math import pi, sqrt, sin, cos, exp
from numpy import linspace, array, tanh, cosh, ones, arctan
import numpy as np
from scipy.special import erf
from scipy.optimize import root

from bvp6c import bvp6c, bvpinit, deval
from structure_variable import struct

R = 209
beta = 4.26
rho0 = 2.704e-3
g = 3.2172e-4
s = 26600

def C_d(u):
    return 1.174 - 0.9*cos(u)

def C_l(u):
    return 0.6*sin(u)
```

Realistic boundary conditions for the trajectory of the spacecraft are

$$\begin{aligned} v(0) &= 0.36 \quad (36000 \text{ ft/sec}) & v(T) &= 0.27 \\ \gamma(0) &= -8.1^\circ \frac{\pi}{180^\circ} & \gamma(T) &= 0 \\ \xi(0) &= \frac{4}{R} \quad (h = 400000 \text{ ft}) & \xi(T) &= \frac{2.5}{R} \end{aligned} \tag{23.2}$$

where T represents the time at the end of the (first) reentry maneuver. These boundary conditions are similar to those encountered at the end of each Apollo mission to the moon.

The total heating is

$$J[u] = \int_0^T 10v^3\sqrt{\rho}.$$

The Hamiltonian corresponding to this control system[2] is

$$\begin{aligned} H = 10v^3\sqrt{\rho} + \lambda_1 \left(-s\rho v^2 C_D(u) - \frac{g\sin(\gamma)}{(1+\xi)^2} \right) + \\ \lambda_2 \left(s\rho v C_L(u) + \frac{v\cos(\gamma)}{R(1+\xi)} - \frac{g\cos\gamma}{v(1+\xi)^2} \right) + \\ \lambda_3 \left(\frac{v\sin\gamma}{R} \right), \end{aligned} \tag{23.3}$$

where $\lambda = [\lambda_1, \lambda_2, \lambda_3]^T$ is the adjoint variable. The state and adjoint equations are thus given by

$$\begin{aligned} \dot{y} &= H_\lambda, \quad \dot{} = \frac{d}{dt}, \\ \dot{\lambda} &= -H_y, \end{aligned} \tag{23.4}$$

where $y = [y_1, y_2, y_3]^T = [v, \gamma, \xi]^T$. To our boundary conditions we add the terminal condition that $H = 0$ at $t = T$. Finally, from the condition $\frac{\partial H}{\partial u} = 0$ we find that the optimal control satisfies

$$\tan u = \frac{6\lambda_2}{9v\lambda_1}. \tag{23.5}$$

Most BVP solvers require an equal number of differential equations and boundary conditions. Currently we have a free boundary value problem; there are 6 ODEs and 7 boundary conditions, and the length of the reentry maneuver, T, is still unknown. By making the transformation $x = t/T$, and treating T as a dependent variable, the BVP is now defined on the interval $(0, 1)$ and is augmented with an additional ODE:

$$\begin{aligned} y' &= TH_\lambda, \quad ' = \frac{d}{dx}, \\ \lambda' &= -TH_y, \\ T' &= 0. \end{aligned} \tag{23.6}$$

This BVP has 7 ODEs, and with the 7 boundary conditions introduced earlier it has the required form.

[2]Here we are using the Pontryagin Minimum Principle, rather than the Maximum Principle. Due to this slight variation, the Hamiltonian is defined as $\lambda \cdot f + L$, where λ is the costate vector, the state equation is $\dot{x} = f$, and the functional $J = \int_0^T L$.

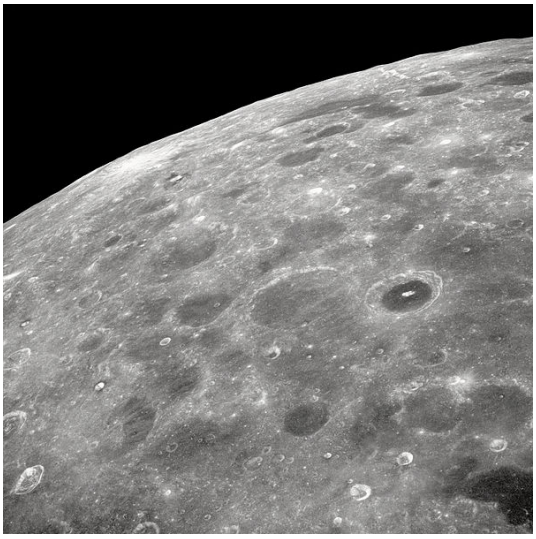


Figure 23.2: The Apollo 8 mission was the first to orbit the moon and return to earth. After a flight of three days from earth, they orbited the moon ten times in 20 hours before making the return trip. This photograph shows a portion of the far side of the moon, as seen by the Apollo 8.

Problem 1. Complete the function `ode` below that implements the right hand side of (23.6). Notice that the adjoint variables and the final time are coordinates of y : $y_4 = \lambda_1$, $y_5 = \lambda_2$, $y_6 = \lambda_3$, and $y_7 = T$. Finally, note that we use Python zero based indexing below.

```
def ode(x,y):
    # Parameters:
    # x: independent variable (unused in our ODEs)
    # y: vector-valued dependent variable; it is an ndarray
    #    with shape (7,)

    # Returns:
    # ndarray of length (7,) that evalutes the RHS of the ODES
    u =  arctan((6*y[4])/(9*y[0]*y[3] ))
    rho = rho0*exp(-beta*R*y[2])
    out = y[6]*array([
                 # G_0
                 -s*rho*y[0]**2*C_d(u) - g*sin(y[1])/(1+y[2])**2,
                 # G_1
                ( s*rho*y[0]*C_l(u) + y[0]*cos(y[1])/(R*(1 + y[2])) -
                  g*cos(y[1])/(y[0]*(1+y[2])**2) ),
                 # G_2
                y[0]*sin(y[1])/R,
                 # G_3
                -( 30*y[0]**2.*sqrt(rho)+ y[3]*(-2*s*rho*y[0]*C_d(u)) +
                   y[4]*( s*rho*C_l(u) +cos(y[1])/(R*(1 + y[2])) +
                          g*cos(y[1])/( y[0]**2*(1+y[2])**2 )
```

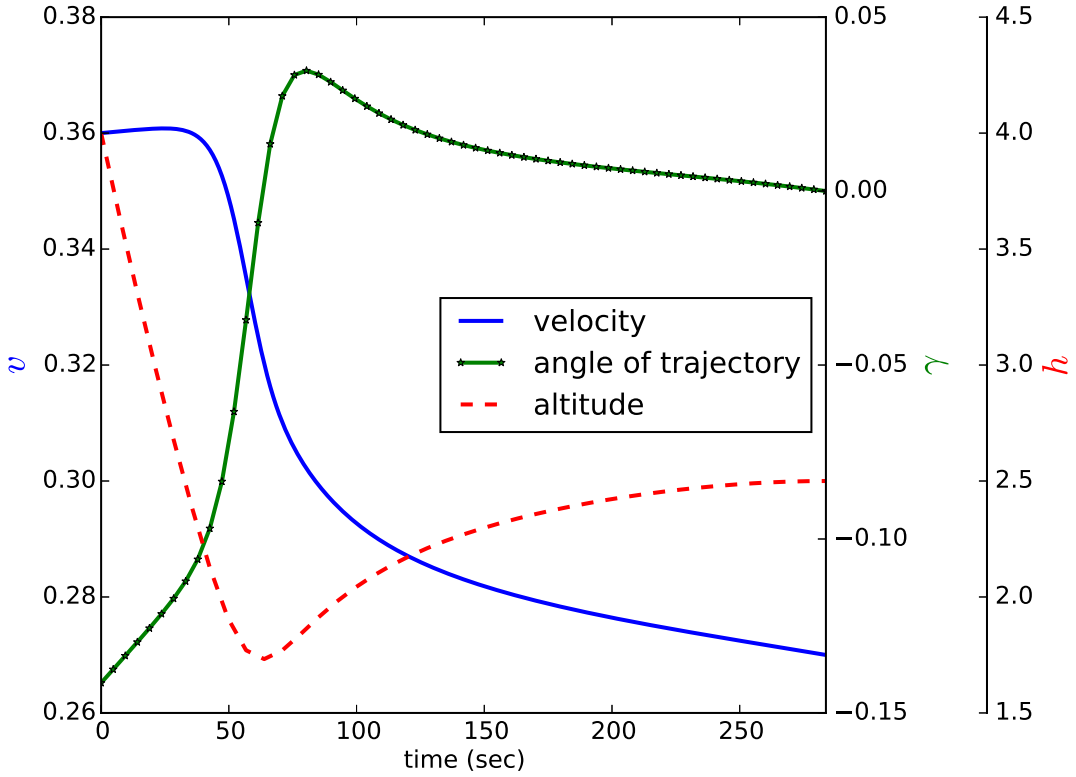


Figure 23.3: The optimal path for the reentry maneuver of a spacecraft. This path minimizes the heating of the spacecraft, and satisfies (23.6),(23.2), and the terminal condition $H(T) = 0$.

```
                        ) +
                    y[5]*(sin(y[1])/R)      ),
                  # G_4
                -( y[3]*( -g*cos(y[1])/(1+y[2])**2  ) +
                    y[4]*( -y[0]*sin(y[1])/(R*(1+y[2])) +
                           g*sin(y[1])/(y[0]*(1+y[2])**2 )
                            ) +
                    y[5]*(y[0]*cos(y[1])/R )    ),
                  # G_5 -- This line needs to be completed.
                  ,
                  # G_6
                    0
            ])
    return out
```

Constructing an Initial Guess

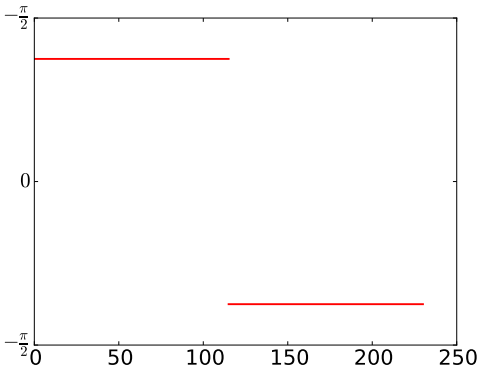
We will use the BVP solver `scikits.bvp_solver`. Like any solver capable of handling nonlinear problems, `bvp_solver` requires an initial guess to jump-start its Newton-like iteration process. Our nonlinear BVP is very sensitive, and requires an initial guess that is quite close to the solution. This sensitivity is physically meaningful. The spacecraft is traveling at a speed far greater than a typical aircraft. If the control is not aggressive, the spacecraft will fall/'bounce' back into space as it encounters the atmosphere at a high velocity. However, if the control lasts too long, the craft will overheat or crash.

Since this is a sensitive problem, we will use a heuristic method to construct good initial guesses for $v, \gamma, \xi, \lambda_1, \lambda_2, \lambda_2$, and u. From aerospace engineers we know that the control u should empirically look like Figure 23.4; we can create a smooth approximation of the form $u = p_1 \operatorname{erf}(p_2(p_3 - t/T))$, where p_1, p_2, and p_3 are unknown constants. To help us determine these constants, and to find good initial guesses for v, γ, and ξ, we define an auxiliary BVP

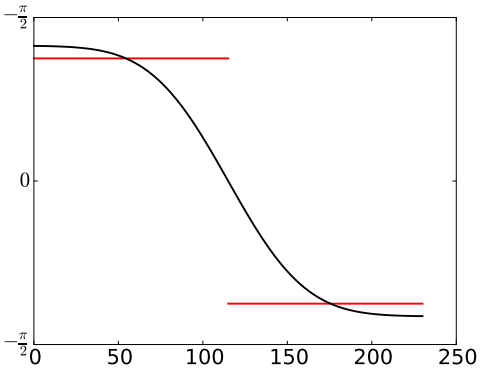
$$\begin{aligned}
\dot{y_0} &= -s\rho y_0^2 C_D(u) - \frac{g\sin(y_1)}{(1+y_2)^2}, \\
\dot{y_1} &= s\rho y_0 C_L(u) + \frac{y_0\cos(y_1)}{R(1+y_2)} - \frac{g\cos y_1}{y_0(1+y_2)^2}, \\
\dot{y_2} &= \frac{y_0 \sin y_1}{R}, \\
\dot{p_1} &= 0, \\
\dot{p_2} &= 0, \\
\dot{p_3} &= 0.
\end{aligned} \tag{23.7}$$

This auxiliary BVP is defined on the interval $[0, T]$, where T is unknown. We guess at T: the maneuver will occur quickly, so how about 230 seconds? After this boundary value problem has been solved, we will have good initial guesses for the correct v, γ, ξ, and u. We will still need to construct initial guesses for λ_1, λ_2, and λ_3. Below we code functions for (23.7) and for the boundary conditions.

```
T0 = 230

def ode_auxiliary(t,y):
    u = y[3]*erf( y[4]*(y[5]-(1.*t)/T0) )
    rho = rho0*exp(-beta*R*y[2])
    out = array([-s*rho*y[0]**2*C_d(u) - g*sin(y[1])/(1+y[2])**2,
                 ( s*rho*y[0]*C_l(u) + y[0]*cos(y[1])/(R*(1 + y[2])) -
                 g*cos(y[1])/(y[0]*(1+y[2])**2) ),
                 y[0]*sin(y[1])/R,
                 0,
                 0,
                 0     ])
    return out

def bcs_auxiliary(ya,yb):
    out1 = array([ ya[0]-.36,
                 ya[1]+8.1*pi/180,
                 ya[2]-4/R
                 ])
```



Heuristic for the control u, provided by engineers.



A smooth initial approximation of the control.

Figure 23.4: We construct a smooth estimate for the control u, by supposing the control has the form $u = p_1 \operatorname{erf}(p_2(p_3 - t/T))$ and estimating parameters p_1, p_2, p_3.

```
    out2 = array([ yb[0]-.27,
                   yb[1],
                   yb[2]-2.5/R
                   ])
    return out1, out2
```

The two main functions used are `ProblemDefinition` and `solve`. The function `solve` requires an initial guess, which you will create in Problem 2.

```
problem_auxiliary = bvp_solver.ProblemDefinition(num_ODE = 6,
                                          num_parameters = 0,
                                          num_left_boundary_conditions = 3,
                                          boundary_points = (0, T0),
                                          function = ode_auxiliary,
                                          boundary_conditions = bcs_auxiliary)

solution_auxiliary = bvp_solver.solve(problem_auxiliary,
                                  solution_guess = guess_auxiliary)

N = 240
t_guess = linspace(0,T0,N+1)
guess = solution_auxiliary(t_guess)
```

Problem 2. Complete the function `guess_auxiliary` given below. Then run the code above to check that your initial guess is adequate. This function provides an initial guess to `bvp_solver` for the auxiliary BVP described by (23.7) and (23.2). Use the heuristic data provided in Figure 23.4 to find good estimates of p_1, p_2, and p_3. Use Figure 23.3 to estimate the trajectories of y_1, y_2, and y_3. Hint: Try using the tanh function.

```
def guess_auxiliary(t):
    out = array([ .5*(.36+.27)-.5*(.36-.27)*tanh(.025*(t-.45*T_init)),
            # Finish this line,
            # And this one,
            p1*ones(t.shape),
            p2*ones(t.shape),
            p3*ones(t.shape)   ])
    return out
```

At this point we have constructed good initial guesses for the dependent variables y_1, y_2, y_3, and y_7 (representing the total time of the manuever) in the original BVP (23.6). We now need to construct initial guesses for the adjoint variables y_4, y_5, and y_6.

By reexamining the condition $H_u = 0$, we find that the optimal control u satisfies

$$\sin u = \frac{-0.6y_5}{\alpha} \qquad \cos u = \frac{-0.9y_1y_4}{\alpha}$$

where $\alpha = \sqrt{(0.6y_5)^2 + (0.9y_1y_4)^2}$. From this we know that $y_4 < 0$, since $\cos u > 0$. A simple guess would be $y_4 = -1$. (Recall that the adjoint variables are unique up to some scaling.) We can then approximate y_5 from the relationship

$$\tan u = \frac{6y_5}{9y_1y_4}.$$

To approximate y_6, we use the identity $H = 0$.

Problem 3. Adapt your previous code to solve the original, dimension seven BVP. Use the solution of the auxiliary BVP to construct a good initial guess. Plot the control u. How long does the reentry maneuver take?

24 HIV Treatment Using Optimal Control

Introduction

Viruses are the cause of many common illnesses in society today, such as ebola, influenza, the common cold, and Human Immunodeficiency Virus (HIV). Viruses are not considered to be living organisms as they cannot reproduce on their own. Instead they inject their genes in the form of DNA or RNA into a host's genome. They then use the cell's ribosomes and proteins to make the protein coat and replicate their genes. At the end they lyse the cell (tear it apart) and release many virus particles to infect other cells.

The body has an adaptive immune system which learns to recognize viruses and bacteria and their hosts, and how to destroy them. A major component of this system are T cells. These cells perform many necessary functions such as recognizing invaders, destroying infected cells, and remembering previous infections long after recovery. Of particular interest is the helper T cell, also known as the CD4+T cell, due to a protein found on its surface which regulates the immune responses. HIV is unique in that it specifically targets this particular type of T cell. This means that the system responsible for fighting infections is specifically targeted.

This loss of CD4+T cells is what causes Acquired Immune Deficiency Syndrome (AIDS). Note that AIDS itself is not an infection, which is a common misconception among the population. Due to the lack of T cells to recognize viruses and bacteria, the body becomes susceptible to other forms of infection. Whereas most people are easily able to shake off a common cold, someone suffering from the advanced stages of AIDS will be at serious risk of dying. Since AIDS comes from a loss of T cells, it may be several years before the host notices the effects of the infection. This enables the HIV virus to spread more easily since the host might not realize they are infected and continue in whatever behavior made them susceptible to the infection initially.

Currently there is no cure or vaccine for HIV. However, there are treatments that reduce the virus and bolster the immune system by increasing the CD4+T cell count. Since these treatments can be expensive and often have negative side effects, it is important to optimize the amount of drugs used. Sometimes combinations of these drugs are used to provide a better effect. In this lab we will use optimal control to find the optimal dosage of a two-drug combination[1].

[1] *SHORT COURSES ON THE MATHEMATICS OF BIOLOGICAL COMPLEXITY*, Web. 15 Apr. 2015 http://www.math.utk.edu/ lenhart/smb2003.v2.html.

Derivation of Control

We begin by defining some variables. Let T represents the concentration of $CD4^+T$ cells and V the concentration of HIV particles. s_1 and s_2 represent the production of T cells by various processes. B_1 and B_2 are half saturation constants (sort of like crowd control in the blood stream and plasma). Let μ be the death rate of uninfected T cells, k the rate of infection of T cells, and c the death rate of the virus. Let g be the input rate of some external viral source. The control variables u_1 and u_2 represent 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:

$$\begin{aligned}\frac{dT(t)}{dt} &= s_1 - \frac{s_2 V(t)}{B_1 + V(t)} - \mu T(t) - kV(t)T(t) + u_1(t)T(t),\\ \frac{dV(t)}{dt} &= \frac{gV(t)}{B_2 + V(t)}(1 - u_2(t)) - cV(t)T(t).\end{aligned} \tag{24.1}$$

The term $s_1 - \frac{s_2 V(t)}{B_1+V(t)}$ is the source/proliferation of unaffected T cells, $\mu T(t)$ the natural loss of T cells, $kV(t)T(t)$ the loss of T cells by infection. $\frac{gV(t)}{B_2+V(t)}$ represents the viral contribution to plasma, and $cV(t)T(t)$ the viral loss. To these equations we add initial conditions $T(0) = T_0$ and $V(0) = V_0$. [3]

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 i) the benefit of T cells, and ii) 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

$$J(u_1^*, u_2^*) = \max\{J(u_1, u_2)|(u_1, u_2) \in U\} = \min\{-J(u_1, u_2)|(u_1, u_2) \in U\},$$

where $U = \{(u_1, u_2)|u_i \text{ is measurable }, a_i \le u_i \le b_i, t \in [0, t_f] \text{ for } i = 1, 2\}$.

Optimality System

The Hamiltonian is defined as:

$$\begin{aligned}H &= \vec{\lambda} \cdot \vec{f} - L\\ H &= \lambda_1\Big[s_1 - \frac{s_2 V}{B_1 + V} - \mu T - kVT + u_1 T\Big] + \lambda_2\Big[\frac{g(1 - u_2)V}{B_2 + V} - cVT\Big]\\ &\quad + [T - (A_1 u_1^2 + A_2 u_2^2)].\end{aligned}$$

Note that the costate is represented with λ instead of p. The costate evolution equations are:

$$\begin{aligned}\lambda_1^{'} &= -\frac{\partial H}{\partial T} = -1 + \lambda_1[\mu + kV^* - u_1^*] + \lambda_2 cV^*,\\ \lambda_2^{'} &= -\frac{\partial H}{\partial V} = \lambda_1\Big[\frac{B_1 s_2}{(B_1 + V^*)^2} + kT^*\Big] - \lambda_2\Big[\frac{B_2 g(1 - u_2^*)}{(B_2 + V^*)^2} - cT^*\Big].\end{aligned}$$

[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

The transversality (or endpoint) conditions are $\lambda_1(t_f) = \lambda_2(t_f) = 0$, with $T(0) = T_0$ and $V(0) = V_0$. The optimality equations are:

$$\frac{\partial H}{\partial u_1} = -2A_1 u_1^*(t) + \lambda_1 T^*(t) = 0$$

$$\frac{\partial H}{\partial u_2} = -2A_2 u_2^*(t) + \lambda_2 \left[\frac{-gV^*(t)}{B_2 + V^*(t)}\right] = 0$$

From these conditions we obtain

$$u_1^*(t) = \frac{1}{2A_1}\left[\lambda_1 T^*(t)\right],$$
$$u_2^*(t) = \frac{-1}{2A_2}\left[\lambda_2 \frac{gV^*(t)}{B_2 + V^*(t)}\right].$$

From the bounds on the controls we have

$$u_1^*(t) = \min\left\{\max\{a_1, \frac{1}{2A_1}(\lambda_1 T^*(t))\}, b_1\right\},$$
$$u_2^*(t) = \min\left\{\max\{a_2, \frac{-\lambda_2}{2A_2}\frac{gV^*(t)}{B_2 + V^*(t)}\}, b_2\right\}.$$

This gives us the optimal system

$$\begin{aligned} T' &= s_1 - \frac{s_2 V}{B_1 + V} - \mu T - kVT + \min\{\max\{a_1, \frac{1}{2A_1}(\lambda_1 T)\}, b_1\}T, \\ V' &= \frac{g(1 - \min\{\max\{a_2, \frac{-\lambda_2}{2A_2}\frac{gV}{B_2+V}\}, b_2\})V}{B_2 + V} - cVT \end{aligned} \tag{24.2}$$

$$\begin{aligned} \lambda_1' &= -1 + \lambda_1\left[\mu + kV - \min\{\max\{a_1, \frac{1}{2A_1}(\lambda_1 T)\}, b_1\}\right] + \lambda_2 cV, \\ \lambda_2' &= \lambda_1\left[\frac{B_1 s_2}{(B_1 + V)^2} + kT\right] - \lambda_2\left[\frac{B_2 g(1 - \min\{\max\{a_2, \frac{-\lambda_2}{2A_2}\frac{V}{B_2+V}\}, b_2\})}{(B_2 + V)^2} - cT\right], \end{aligned} \tag{24.3}$$

with end conditions $\lambda_1(t_f) = \lambda_2(t_f) = 0$, and $T(0) = T_0, V(0) = V_0$.

Creating a Numerical Solver

We iteratively solve for our 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 the RK4 solver from a previous lab with one minor adjustment. Our state equations depend on u, and our costate equations depend on our state equations. Therefore, we will pass another parameter into the function that RK4 takes in that will index the arrays our equations depend on.

```
# Dependencies for this lab's code:
import numpy as np
from matplotlib import pyplot as plt

#Code from RK4 Lab with minor edits
def initialize_all(y0, t0, tf, n):
```

```
    """ An initialization routine for the different ODE solving
    methods in the lab. This initializes Y, T, and h. """
    if isinstance(y0, np.ndarray):
        Y = np.empty((n, y0.size)).squeeze()
    else:
        Y = np.empty(n)
    Y[0] = y0
    T = np.linspace(t0, tf, n)
    h = float(tf - t0) / (n - 1)
    return Y, T, h

def RK4(f, y0, t0, tf, n):
    """ Use the RK4 method to compute an approximate solution
    to the ODE y' = f(t, y) at n equispaced parameter values from t0 to t
    with initial conditions y(t0) = y0.

    y0 is assumed to be either a constant or a one-dimensional numpy array.
    tf and t0 are assumed to be constants.
    f is assumed to accept three arguments.
    The first is a constant giving the value of t.
    The second is a one-dimensional numpy array of the same size as y.
    The third is an index to the other arrays.

    This function returns an array Y of shape (n,) if
    y is a constant or an array of size 1.
    It returns an array of shape (n, y.size) otherwise.
    In either case, Y[i] is the approximate value of y at
    the i'th value of np.linspace(t0, tf, n).
    """
    Y,T,h = initialize_all(y0,t0,tf,n)
    for i in range(n-1):
        K1 = f(T[i],Y[i],i)
        K2 = f(T[i]+h/2.,Y[i]+h/2.*K1,i)
        K3 = f(T[i]+h/2.,Y[i]+h/2.*K2,i)
        K4 = f(T[i+1],Y[i]+h*K3,i)
        Y[i+1] = Y[i] + h/6.*(K1+2*K2 +2*K3+K4)
    return Y
```

Problem 1. Create a function that defines the state equations and returns both equations in a single array. The function should be able to be passed into the RK4 solver. This function can depend on the global variables defined below.

Achtung!

When solving the state equations, because of the nature of T' and V', solve the original equations (24.1) from the beginning of the lab and not the equations (24.2) with $u_i^*(t)$ replaced by the minmax function.

```
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
T0, V0 = 400, 3
t_f = 50
n = 1000
```

These constants come from both references cited at the end of this lab.

```
# initialize global variables, state, costate, and u.
state = np.zeros((n,2))
state0 = np.array([T0, V0])

costate = np.zeros((n,2))
costate0 = np.zeros(2)

u=np.zeros((n,2))
u[:,0] += .02
u[:,1] += .9

# define state equations
def state_equations(t,y,i):
    '''
    Parameters
    ----------
    t : float
        the time
    y : ndarray (2,)
        the T cell concentration and the Virus concentration at time t
    i : int
        index for the global variable u.
    Returns
    -------
    y_dot : ndarray (2,)
```

```
        the derivative of the T cell concentration and the virus ↩
            concentration at time t
    '''
    pass
```

The state equations work great in the RK4 solver; however, the costate equations have end conditions rather than initial conditions. Thus we want our RK4 solver to iterate backwards from the end to the beginning. An easy way to accomplish this is to define a function $\hat{\lambda}_i(t) = \lambda_i(t_f - t)$. Then $\hat{\lambda}_i$ has the initial conditions $\hat{\lambda}_i(0) = \lambda_i(t_f)$. We get the new equations

$$\begin{aligned}
\dot{\hat{\lambda}}_1(t) &= \lambda_1(t_f - t)\left(-\mu - kV(t_f - t) + u_1(t_f - t)\right) - c\lambda_2(t_f - t)V(t_f - t) + 1, \\
\dot{\hat{\lambda}}_2(t) &= -\lambda_1(t_f - t)\left(\frac{s_2 B_1}{(B_1 + V(t_f - t))^2} + kT(t_f - t)\right) \\
&\quad + \lambda_2(t_f - t)\left(\frac{gB_2(1 - u_2(t_f - t))}{(B_2 + V(t_f - t))^2} - cT(t_f - t)\right).
\end{aligned}$$

These we can solve with our RK4 solver and recover the original costate equations by simply indexing the array backwards.

Problem 2. Create a function that defines the costate equations and returns both equations in a single array. The function should be able to be passed into the RK4 solver. Use the global variables as defined in Problem 1.

```
def lambda_hat(t,y,i):
    '''
    Parameters
    ----------
    t : float
        the time
    y : ndarray (2,)
        the lambda_hat values at time t
    i : int
        index for global variables, u and state.
    Returns
    -------
    y_dot : ndarray (2,)
        the derivative of the lambda_hats at time t.
    '''
    pass
```

Finally, we can put these together to create our solver.

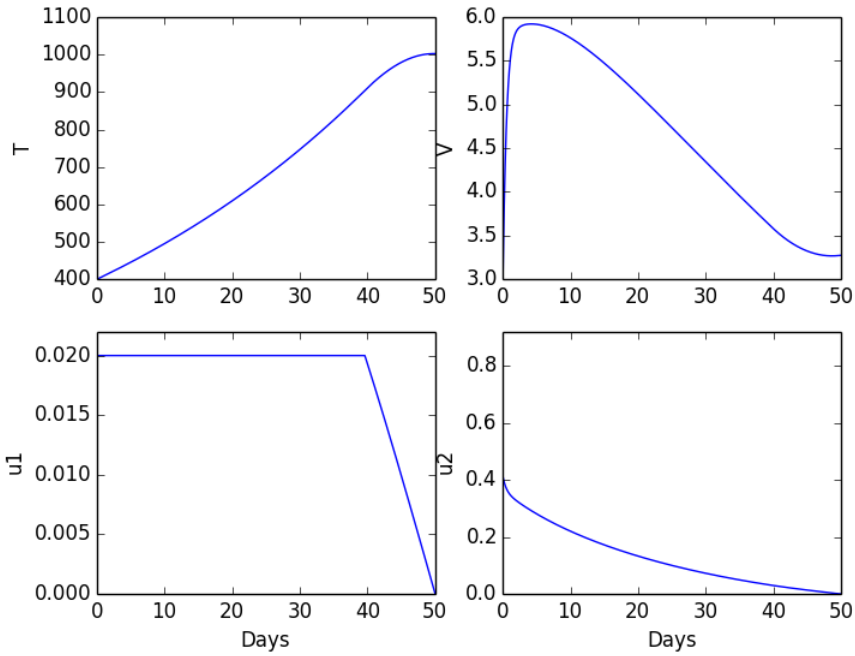


Figure 24.1: The solution to Problem 3.

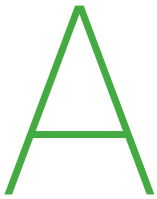
Problem 3. Create and run a numerical solver for the HIV two drug model using the code below.

```
epsilon = 0.001
test = epsilon + 1

while(test > epsilon):
    oldu = u.copy();

    #solve the state equations with forward iteration
    #state = RK4(...)

    #solve the costate equations with backwards iteration
    #costate = RK4(...)[::-1]

    #solve for u1 and u2

    #update control
    u[:,0] = 0.5*(u1 + oldu[:,0])
    u[:,1] = 0.5*(u2 + oldu[:,1])

    #test for convergence
```

```
    test = abs(oldu - u).sum()
```

Your solutions should match Figure 24.1.

In modern medicine, patients generally take combinations of five or more medications with different functions. These include Nucleotide Reverse Transcriptase Inhibitors, which prevent HIV inserting its genes into host DNA, Non-Nucleoside Reverse Transcriptase Inhibitors, which do the same job as NRTIs in a different fashion, Protease Inhibitors, which cut up replicated HIV strands, Fusion Inhibitors, which block the virus from entering the cells to begin with, and Integrase Inhibitors, which prevents the virus' replicated DNA from being inserted into a cell's DNA. These drugs often can interact with each other and have different side effects on the body. Also, doctors rotate medications as the body and virus develop immunity.



Getting Started

The labs in this curriculum aim to introduce computational and mathematical concepts, walk through implementations of those concepts in Python, and use industrial-grade code to solve interesting, relevant problems. Lab assignments are usually about 5–10 pages long and include code examples (yellow boxes), important notes (green boxes), warnings about common errors (red boxes), and about 3–7 exercises (blue boxes). Get started by downloading the lab manual(s) for your course from `http://foundations-of-applied-mathematics.github.io/`.

Submitting Assignments

Labs

Every lab has a corresponding specifications file with some code to get you started and to make your submission compatible with automated test drivers. Like the lab manuals, these materials are hosted at `http://foundations-of-applied-mathematics.github.io/`.

Download the `.zip` file for your course, unzip the folder, and move it somewhere where it won't get lost. This folder has some setup scripts and a collection of folders, one per lab, each of which contains the specifications file(s) for that lab. See `Student-Materials/wiki/Lab-Index` for the complete list of labs, their specifications and data files, and the manual that each lab belongs to.

> ACHTUNG!
>
> Do **not** move or rename the lab folders or the enclosed specifications files; if you do, the test drivers will not be able to find your assignment. Make sure your folder and file names match `Student-Materials/wiki/Lab-Index`.

To submit a lab, modify the provided specifications file and use the file-sharing program specified by your instructor (discussed in the next section). The instructor will drop feedback files in the lab folder after grading the assignment. For example, the Introduction to Python lab has the specifications file `PythonIntro/python_intro.py`. To complete that assignment, modify `PythonIntro/python_intro.py` and submit it via your instructor's file-sharing system. After grading, the instructor will create a file called `PythonIntro/PythonIntro_feedback.txt` with your score and some feedback.

Homework

Non-lab coding homework should be placed in the `_Homework/` folder and submitted like a lab assignment. Be careful to name your assignment correctly so the instructor (and test driver) can find it. The instructor may drop specifications files and/or feedback files in this folder as well.

Setup

> **ACHTUNG!**
>
> We strongly recommend using a Unix-based operating system (Mac or Linux) for the labs. Unix has a true bash terminal, works well with git and python, and is the preferred platform for computational and data scientists. It is possible to do this curriculum with Windows, but expect some road bumps along the way.

There are two ways to submit code to the instructor: with git (`http://git-scm.com/`), or with a file-syncing service like Google Drive. Your instructor will indicate which system to use.

Setup With Git

Git is a program that manages updates between an online code repository and the copies of the repository, called *clones*, stored locally on computers. If git is not already installed on your computer, download it at `http://git-scm.com/downloads`. If you have never used git, you might want to read a few of the following resources.

- Official git tutorial: `https://git-scm.com/docs/gittutorial`
- Bitbucket git tutorials: `https://www.atlassian.com/git/tutorials`
- GitHub git cheat sheet: `services.github.com/.../github-git-cheat-sheet.pdf`
- GitLab git tutorial: `https://docs.gitlab.com/ce/gitlab-basics/start-using-git.html`
- Codecademy git lesson: `https://www.codecademy.com/learn/learn-git`
- Training video series by GitHub: `https://www.youtube.com/playlist?list=PLg7.../`

There are many websites for hosting online git repositories. Your instructor will indicate which web service to use, but we only include instructions here for setup with Bitbucket.

1. *Sign up.* Create a Bitbucket account at `https://bitbucket.org`. If you use an academic email address (ending in `.edu`, etc.), you will get free unlimited public and private repositories.
2. *Make a new repository.* On the Bitbucket page, click the + button from the menu on the left and, under **CREATE**, select **Repository**. Provide a name for the repository, mark the repository as **private**, and make sure the repository type is **Git**. For **Include a README?**, select **No** (if you accidentally include a `README`, delete the repository and start over). Under **Advanced settings**, enter a short description for your repository, select **No forks** under forking, and select **Python** as the language. Finally, click the blue **Create repository** button. Take note of the URL of the webpage that is created; it should be something like `https://bitbucket.org/<name>/<repo>`.

3. *Give the instructor access to your repository.* On your newly created Bitbucket repository page (`https://bitbucket.org/<name>/<repo>` or similar), go to **Settings** in the menu to the left and select **User and group access**, the second option from the top. Enter your instructor's Bitbucket username under **Users** and click **Add**. Select the blue **Write** button so your instructor can read from and write feedback to your repository.

4. *Connect your folder to the new repository.* In a shell application (Terminal on Linux or Mac, or Git Bash (`https://gitforwindows.org/`) on Windows), enter the following commands.

```
# Navigate to your folder.
$ cd /path/to/folder  # cd means 'change directory'.

# Make sure you are in the right place.
$ pwd                 # pwd means 'print working directory'.
/path/to/folder
$ ls *.md             # ls means 'list files'.
README.md             # This means README.md is in the working directory.

# Connect this folder to the online repository.
$ git init
$ git remote add origin https://<name>@bitbucket.org/<name>/<repo>.git

# Record your credentials.
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

For example, if your Bitbucket username is `greek314`, the repository is called `acmev1`, and the folder is called `Student-Materials/` and is on the desktop, enter the following commands.

```
# Navigate to the folder.
$ cd ~/Desktop/Student-Materials

# Make sure this is the right place.
$ pwd
/Users/Archimedes/Desktop/Student-Materials
$ ls *.md
README.md

# Connect this folder to the online repository.
$ git init
$ git remote add origin https://greek314@bitbucket.org/greek314/acmev1.git

# Record credentials.
$ git config --local user.name "archimedes"
```

```
$ git config --local user.email "greek314@example.com"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

At this point you should be able to see the files on your repository page from a web browser. If you enter the repository URL incorrectly in the `git remote add origin` step, you can reset it with the following line.

```
$ git remote set-url origin https://<name>@bitbucket.org/<name>/<repo>.git
```

5. *Download data files.* Many labs have accompanying data files. To download these files, navigate to your clone and run the `download_data.sh` bash script, which downloads the files and places them in the correct lab folder for you. You can also find individual data files through `Student-Materials/wiki/Lab-Index`.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash download_data.sh
```

6. *Install Python package dependencies.* The labs require several third-party Python packages that don't come bundled with Anaconda. Run the following command to install the necessary packages.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash install_dependencies.sh
```

7. (Optional) *Clone your repository.* If you want your repository on another computer after completing steps 1–4, use the following commands.

```
# Navigate to where you want to put the folder.
$ cd ~/Desktop/or/something/

# Clone the folder from the online repository.
$ git clone https://<name>@bitbucket.org/<name>/<repo>.git <foldername>

# Record your credentials in the new folder.
$ cd <foldername>
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Download data files to the new folder.
$ bash download_data.sh
```

Setup Without Git

Even if you aren't using git to submit files, you must install it (`http://git-scm.com/downloads`) in order to get the data files for each lab. Share your folder with your instructor according to their directions, and follow steps 5 and 6 of the previous section to download the data files and install package dependencies.

Using Git

Git manages the history of a file system through *commits*, or checkpoints. Use `git status` to see the files that have been changed since the last commit. These changes are then moved to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. Save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

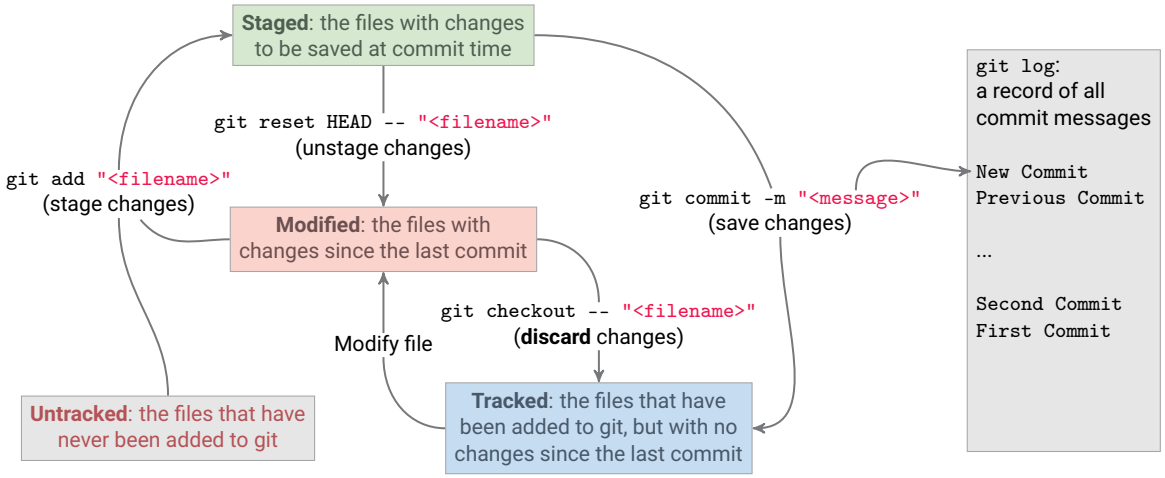


Figure A.1: Git commands to stage, unstage, save, or discard changes. Commit messages are recorded in the log.

All of these commands are done within a clone of the repository, stored somewhere on a computer. This repository must be manually synchronized with the online repository via two other git commands: `git pull origin master`, to pull updates from the web to the computer; and `git push origin master`, to push updates from the computer to the web.

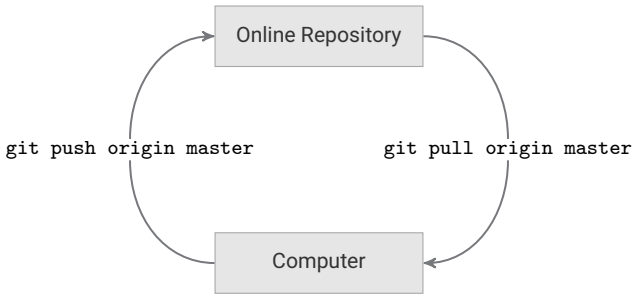


Figure A.2: Exchanging git commits between the repository and a local clone.

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin master`	Pull changes from the online repository.
`git push origin master`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout -- <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD -- <filename>`	Remove a file from the staging area.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Table A.1: Common git commands.

NOTE

When pulling updates with `git pull origin master`, your terminal may sometimes display the following message.

```
Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This means that someone else (the instructor) has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have. This screen, displayed in *vim* (`https://en.wikipedia.org/wiki/Vim_(text_editor)`), is asking you to enter a message (or use the default message) to create a *merge commit* that will reconcile both changes. To close this screen and create the merge commit, type `:wq` and press `enter`.

Example Work Sessions

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master                          # Pull updates.
### Make changes to a file.
$ git add -u                                      # Track changes.
$ git commit -m "Made some changes."              # Commit changes.
$ git push origin master                          # Push updates.
```

```
# Pull any updates from the online repository (such as TA feedback).
$ cd ~/Desktop/Student-Materials/
$ git pull origin master
From https://bitbucket.org/username/repo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

### Work on the labs. For example, modify PythonIntro/python_intro.py.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    PythonIntro/python_intro.py

# Track the changes with git.
$ git add PythonIntro/python_intro.py
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   PythonIntro/python_intro.py

# Commit the changes to the repository with an informative message.
$ git commit -m "Made some changes"
[master fed9b34] Made some changes
 1 file changed, 10 insertion(+) 1 deletion(-)

# Push the changes to the online repository.
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://username@bitbucket.org/username/repo.git
   5742a1b..fed9b34  master -> master

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

B Installing and Managing Python

Lab Objective: *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to `https://www.anaconda.com/download/`.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see `https://docs.anaconda.com/anaconda/install/`. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

> ACHTUNG!
>
> This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see `https://xkcd.com/349/`).

Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See `https://docs.anaconda.com/anaconda/packages/pkg-docs` for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
`conda install <package-name>`	Install the specified package.
`conda update <package-name>`	Update the specified package.
`conda update conda`	Update `conda` itself.
`conda update anaconda`	Update **all** packages included in Anaconda.
`conda --help`	Display the documentation for `conda`.

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda                # Make sure that conda is up to date.
$ conda install matplotlib          # Attempt to install matplotlib.
$ conda update matplotlib           # Attempt to update matplotlib.
```

See `https://conda.io/docs/user-guide/tasks/manage-pkgs.html` for more examples.

Note

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

Achtung!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
`pip install package-name`	Install the specified package.
`pip install --upgrade package-name`	Update the specified package.
`pip freeze`	Display the version number on all installed packages.
`pip --help`	Display the documentation for `pip`.

See `https://pip.pypa.io/en/stable/user_guide/` for more complete documentation.

Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: `https://atom.io/`
- Sublime Text: `https://www.sublimetext.com/`
- Notepad++ (Windows): `https://notepad-plus-plus.org/`
- Geany: `https://www.geany.org/`
- Vim: `https://www.vim.org/`
- Emacs: `https://www.gnu.org/software/emacs/`

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                        # List the files in the current directory.
hello_world.py
$ cat hello_world.py        # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py     # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

> **Note**
>
> While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:
>
> - Windows subsystem for linux: `docs.microsoft.com/en-us/windows/wsl/`.
> - Git bash: `https://gitforwindows.org/`.

Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See `https://github.com/jupyter/jupyter/wiki/` for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Juptyer Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and LaTeX, and can embedded images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: `http://jupyterlab.readthedocs.io/en/stable/`
- PyCharm: `https://www.jetbrains.com/pycharm/`
- Spyder: `http://code.google.com/p/spyderlib/`
- Eclipse with PyDev: `http://www.eclipse.org/`, `https://www.pydev.org/`

See `https://realpython.com/python-ides-code-editors-guide/` for a good overview of these (and other) workflow tools.

C NumPy Visual Guide

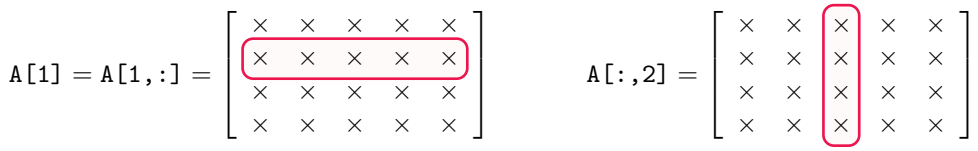
Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

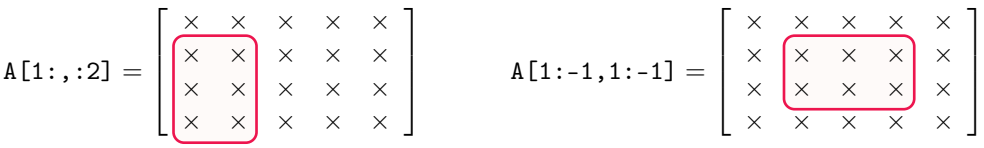
Data Access

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

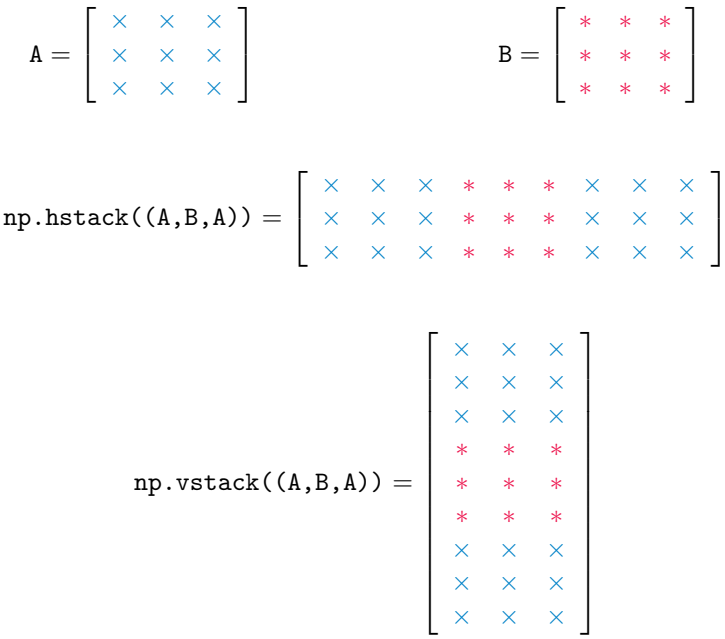
A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



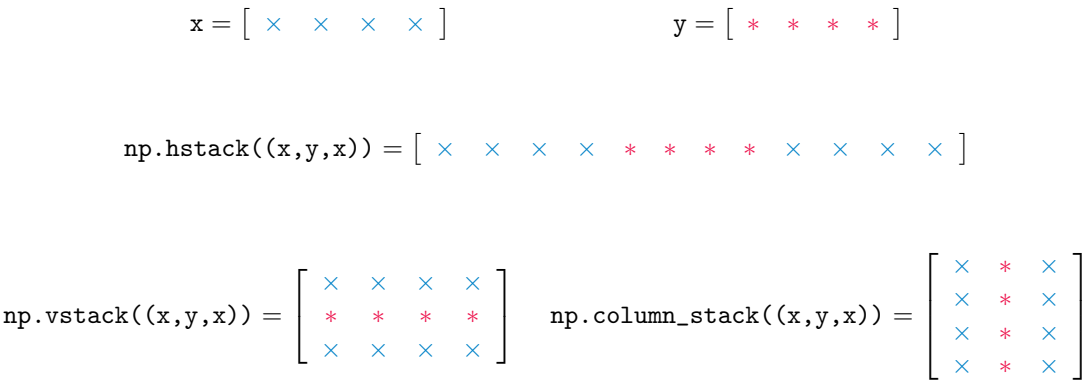


Stacking

`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`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} \boxed{1} & \boxed{2} & \boxed{3} & \boxed{4} \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

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.

[Kim09] Seongjai Kim. Edge-preserving noise removal, part i: Second order anisotropic diffusion. Technical report, University of Kentucky Department of Mathematics, 2009.

[LeV02] Randall J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.

[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.

[PM88] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. Technical Report UCB/CSD-88-483, EECS Department, University of California, Berkeley, Dec 1988.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.