1 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)

$$\begin{aligned} &u_t = \triangle u + f(x,t), \quad x \in \Omega, \quad t > 0, \\ &u(x,t) = h(x), \quad x \in \partial\Omega, \\ &u(x,0) = g(x). \end{aligned} \tag{1.1}$$

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

$$\begin{aligned} &\triangle u + f(x) = 0, \quad x \in \Omega, \\ &u(x,t) = h(x), \quad x \in \partial\Omega. \end{aligned} \tag{1.2}$$

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

Poisson's equation in two dimensions

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

$$\begin{aligned} u_{xx} + u_{yy} &= f, \quad x \text{ in } R \subset \mathbb{R}^2, \\ u &= g, \quad x \text{ on } \partial R. \end{aligned} \tag{1.3}$$

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

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

Recall that

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

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

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

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

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

Then the set of equations

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

can be written in matrix form as

$$AU + p + q = f. \tag{1.4}$$

A is a block tridiagonal matrix, given by

$$\frac{1}{h^2}\begin{bmatrix} T & I & & & \\ I & T & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & T & I \\ & & & I & T \end{bmatrix} \tag{1.5}$$

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

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

The vectors p and q come from the boundary conditions of (1.3), and are given by

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

where

$$p^j = \frac{1}{h^2}\begin{bmatrix} g(x_0, y_j) \\ 0 \\ \vdots \\ 0 \\ g(x_n, y_j) \end{bmatrix}, \quad 1 \le j \le n-1,$$

and

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

The vector f comes from the source term of (1.3), and is given by

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

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

Problem 1. Complete the function `poisson_square` by implementing the finite difference method 1.4. Use `scipy.sparse.linalg.spsolve` to solve the linear system. Use your function to solve the boundary value problem:

$$\begin{aligned} \Delta u = 0, &\quad x \in [0,1] \times [0,1], \\ u(x,y) = x^3, &\quad (x,y) \in \partial([0,1] \times [0,1]). \end{aligned} \tag{1.6}$$

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

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

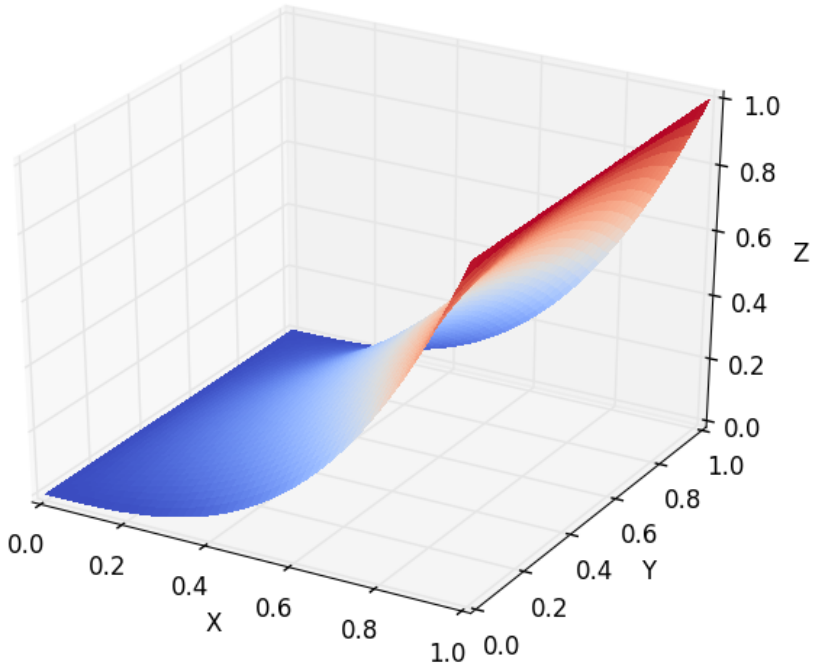


Figure 1.1: The solution of (1.6).

charge density (like a continuous cloud of electrons) being the source term:

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

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

Usually a 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

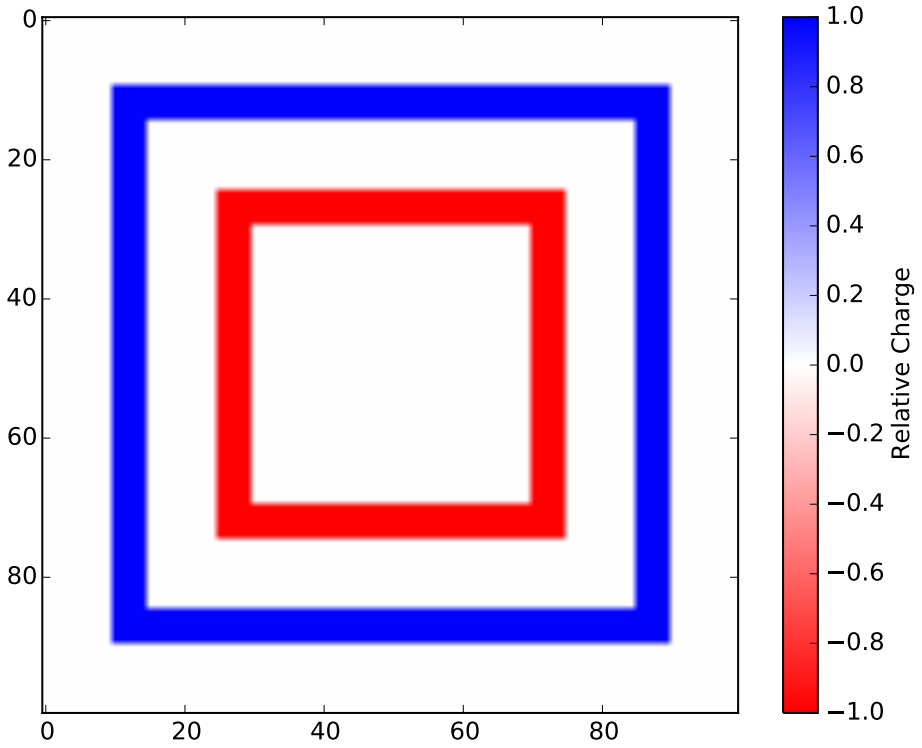


Figure 1.2: The charge density of the 3 nested pipes.

may find it useful to also apply it when you solve for the electric potential.

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

Problem 2. Using the `poisson_square` function, solve

$$\begin{aligned} \Delta V &= -\rho(x,y), \quad x \in [-2,2] \times [-2,2], \\ u(x,y) &= 0, \quad (x,y) \in \partial([-2,2] \times [-2,2]). \end{aligned} \tag{1.7}$$

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

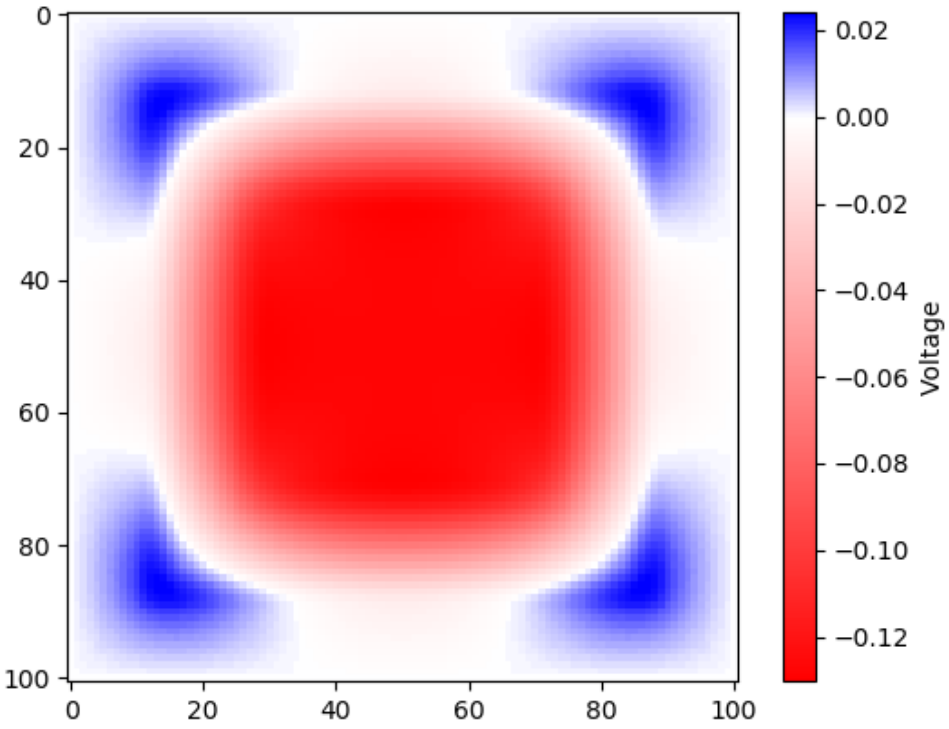


Figure 1.3: The electric potential of the 3 nested pipes.

2 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

$$MA = F \tag{2.1}$$

and

$$LU = F \tag{2.2}$$

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 (2.2) 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

$$D^2 U = F, \tag{2.3}$$

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 (2.3) 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
```

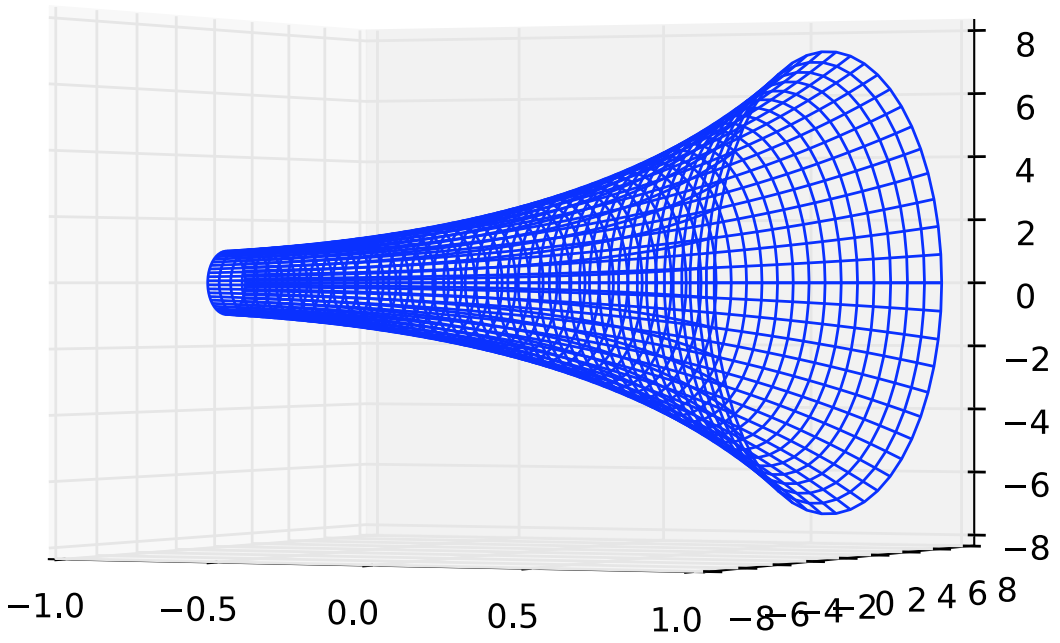


Figure 2.1: The minimal surface corresponding to Problem 5.

```
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()
```

3 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

$$f(x_j) = \frac{1}{2\pi} \sum_{k=-N/2}^{N/2} \frac{e^{ikx_j}}{c_k} \hat{f}(k), \quad j = 0, \dots, N-1, \tag{3.1}$$

where

$$c_k = \begin{cases} 2 & \text{if } k = -N/2 \text{ or } k = N/2, \\ 1 & \text{otherwise.} \end{cases} \tag{3.2}$$

The inverse DFT can then be used to define a natural interpolant (sometimes called a band-limited interpolant) by evaluating (3.1) at any x rather than x_j:

$$p(x) = \frac{1}{2\pi} \sum_{k=-N/2}^{N/2} e^{ikx} \hat{f}(k). \tag{3.3}$$

The interpolant for f' is then given by

$$p'(x) = \frac{1}{2\pi} \sum_{k=-N/2+1}^{N/2-1} ike^{ikx} \hat{f}(k). \tag{3.4}$$

Consider the function $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$. Using (3.4), the derivative u' may be approximated with the following code. [1] We note that although we only approximate u' at the Fourier grid points, (3.4) 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

$$u_t + cu_x = 0 \tag{3.5}$$

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

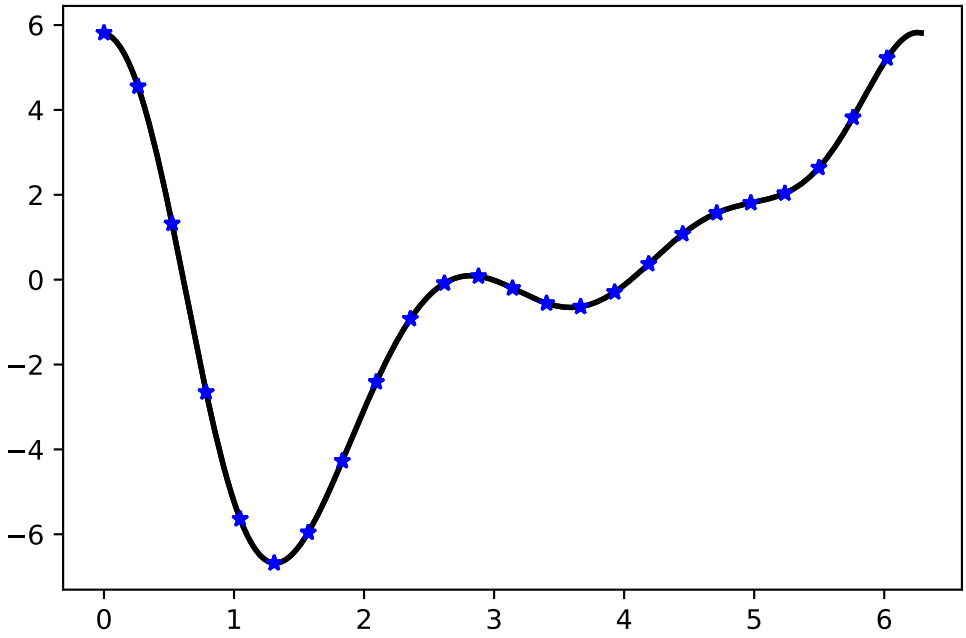


Figure 3.1: The derivative of $u(x) = \sin^2(x)\cos(x) + e^{2\sin(x+1)}$.

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

$$u_t(x_j, t) + cu_x(x_j, t) = 0, \quad t > 0, \quad j = 0, \ldots N - 1. \tag{3.6}$$

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

$$U_t + \vec{c}\mathcal{F}^{-1}\left(i\vec{k}\mathcal{F}(U)\right) = 0 \tag{3.7}$$

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

$$u_t + c(x)u_x = 0, \tag{3.8}$$

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.

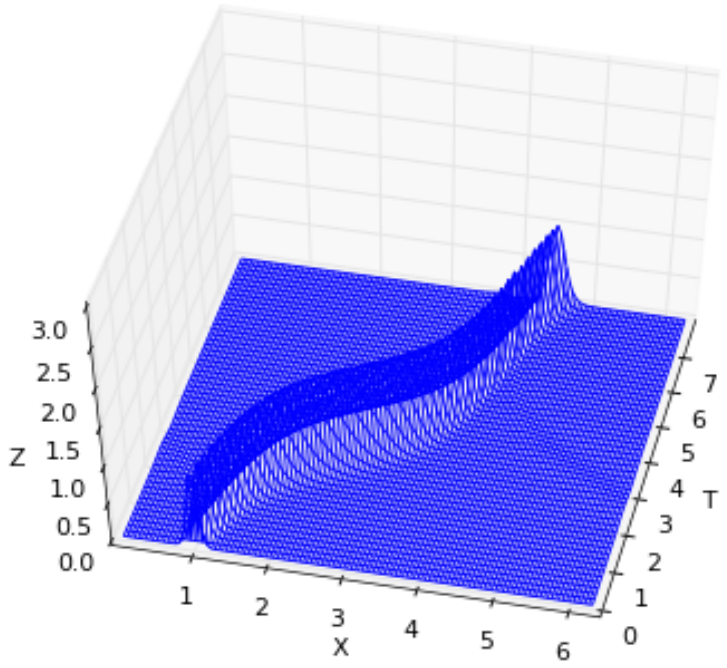


Figure 3.2: The solution of the variable speed advection equation; see Problem 2.

4 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:

$$\begin{aligned} &u_t = -u_{xx}, \quad (x,t) \in (0,L) \times (0,\infty), \\ &u(0,t) = u(L,t) = 0, \quad t \in (0,\infty), \\ &u(x,0) = f(x), \quad x \in (0,L). \end{aligned} \tag{4.1}$$

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.

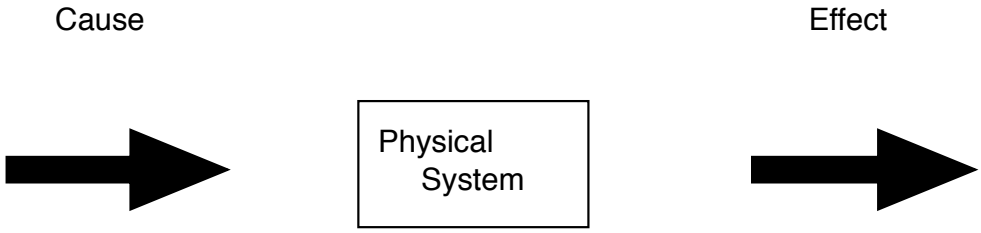


Figure 4.1: Cause and effect within a given physical system.

Inverse Problems

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

Given a physical system, together with initial data (the "cause"), the direct problem will usually predict the future state of the physical system (the "effect"); see Figure 4.1. 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]:

$$\begin{cases} -(au')' = f, & x \in (0,1), \\ a(0)u'(0) = c_0, & a(1)u'(1) = c_1. \end{cases} \tag{4.2}$$

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.

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

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

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 (4.2) 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 $a^h(x)$ that is a linear combination of tent functions. This will be of the form

$$a^h = \sum_{j=0}^{N} \alpha_j \phi_j, \quad \alpha_j = a(x_j). \tag{4.3}$$

The h in this equation indicates that each of the tent functions in the linear combination rely on $h = 1/N$, and that a different h or N will result in different tent functions, so a^h will be different. The second half of (4.3) says that a good choice of a^h is found by taking $\alpha_j = a(x_j)$. Integrating (4.2) from 0 to x, we obtain

$$\begin{aligned} \int_0^x -(au')'\, ds &= \int_0^x f(s)\, ds, \\ -\left[a(x)u'(x) - c_0\right] &= \int_0^x f(s)\, ds, \\ u'(x) &= \frac{c_0 - \int_0^x f(s)\, ds}{a(x)}. \end{aligned} \tag{4.4}$$

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 (4.3) can now be approximated by minimizing

$$\left(\frac{c_0 - \int_0^{x_j} f(s)\, ds}{\alpha_j} - u'(x_j) \right)^2.$$

Problem 1. Solve (4.2) 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 4.2.

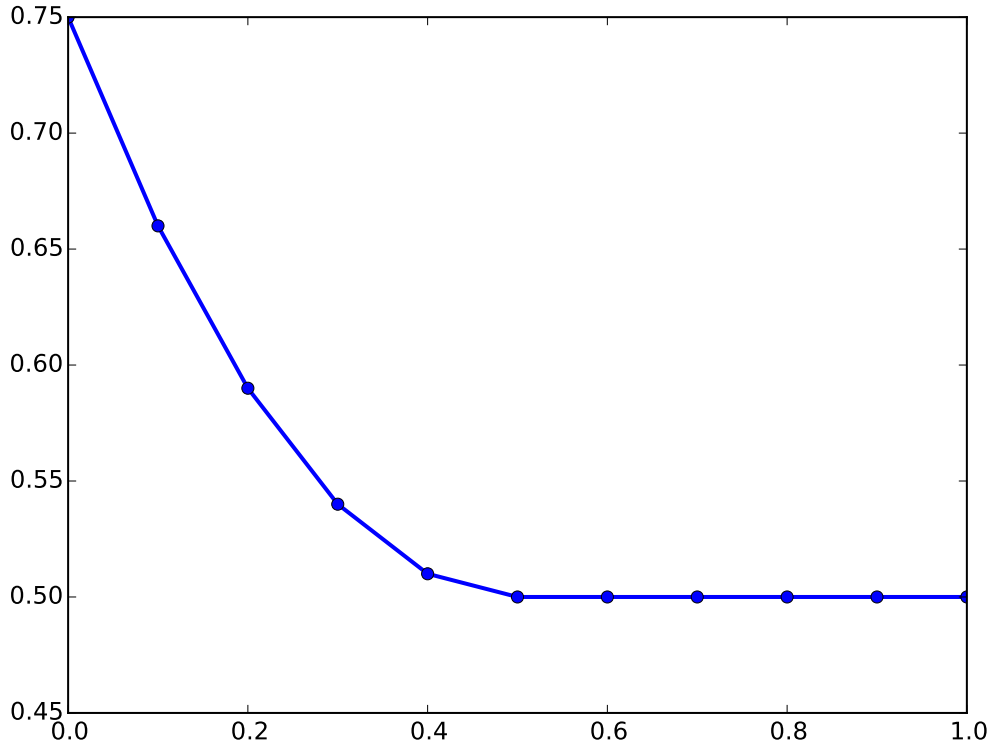


Figure 4.2: The solution $a(x)$ to Problem 1

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

$$\begin{cases} -(au')' = -1, & x \in (0,1), \\ a(0)u'(0) = 1, & a(1)u'(1) = 2. \end{cases} \tag{4.5}$$

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.

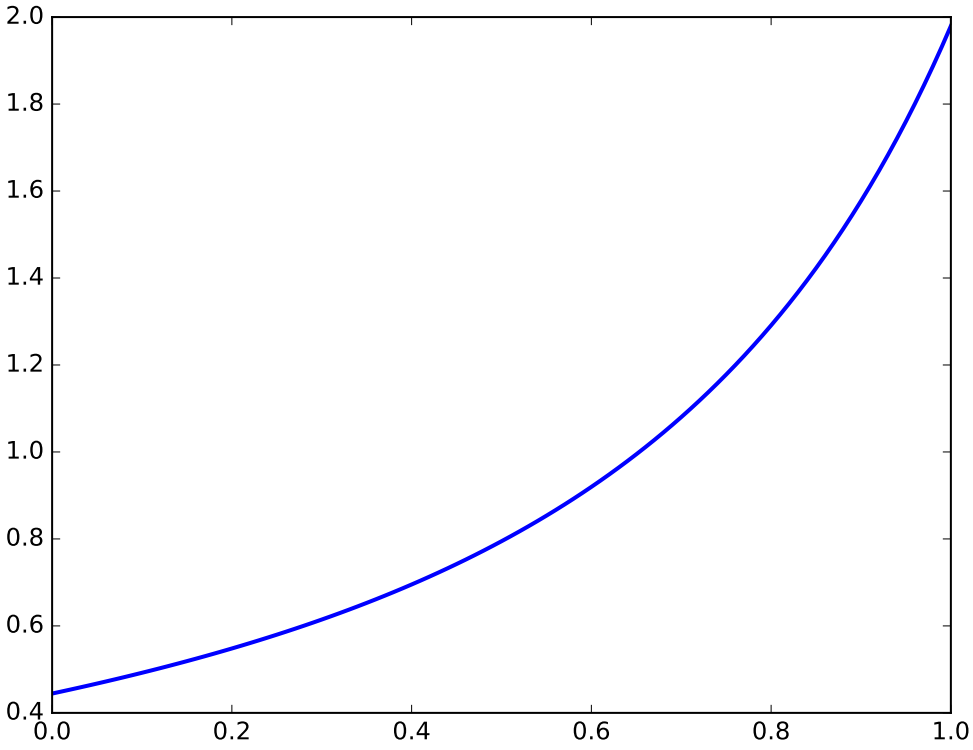


Figure 4.3: The density function $a(x)$ satisfying (4.5) for $\varepsilon = .8$.

5 The Shooting Method for Boundary Value Problems

Consider a boundary value problem of the form

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \le x \le b, \\ y(a) &= \alpha, \quad y(b) = \beta. \end{aligned} \tag{5.1}$$

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

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \le x \le b, \\ y(a) &= \alpha, \quad y'(a) = t. \end{aligned} \tag{5.2}$$

> **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 (5.2). We can rewrite the initial value conditions as

$$y(a, t) = \alpha, \quad y'(a, t) = t \tag{5.3}$$

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

$$\begin{aligned} y'' &= -4y - 9\sin(x),\ x \in [0, 3\pi/4], \\ y(0) &= 1, \\ y(3\pi/4) &= -\frac{1+3\sqrt{2}}{2}. \end{aligned} \tag{5.4}$$

The following code implements the secant method to solve (5.4). 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 5.1.) What initial values t_0, t_1 did you use to find them?

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

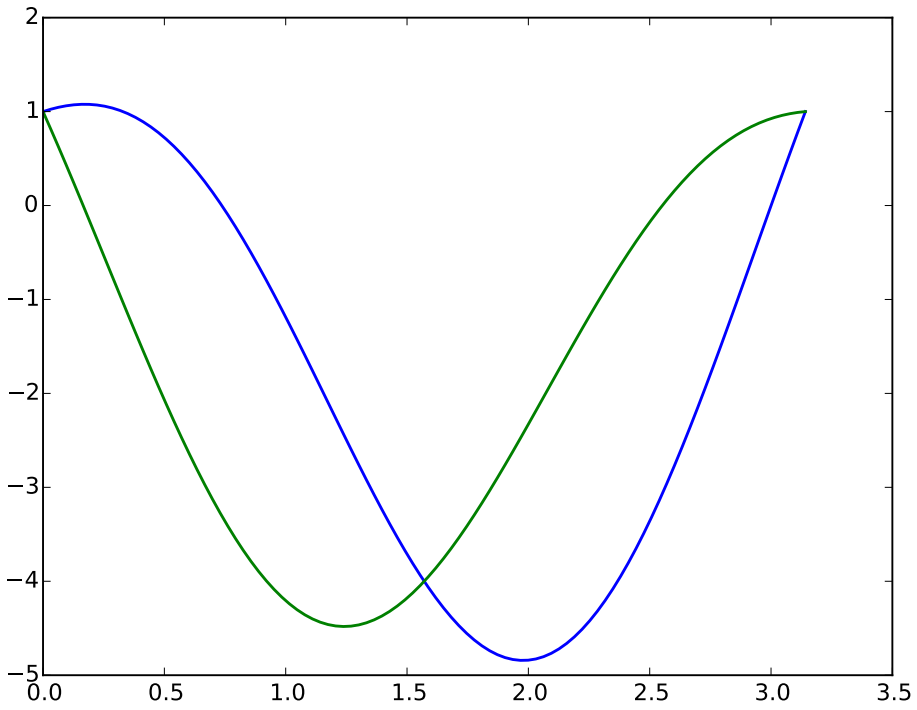


Figure 5.1: Two solutions of $y'' = -4y - 9\sin(x)$, both satisfying the boundary conditions $y(0) = y(\pi) = 1$.

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 5.2.) 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.

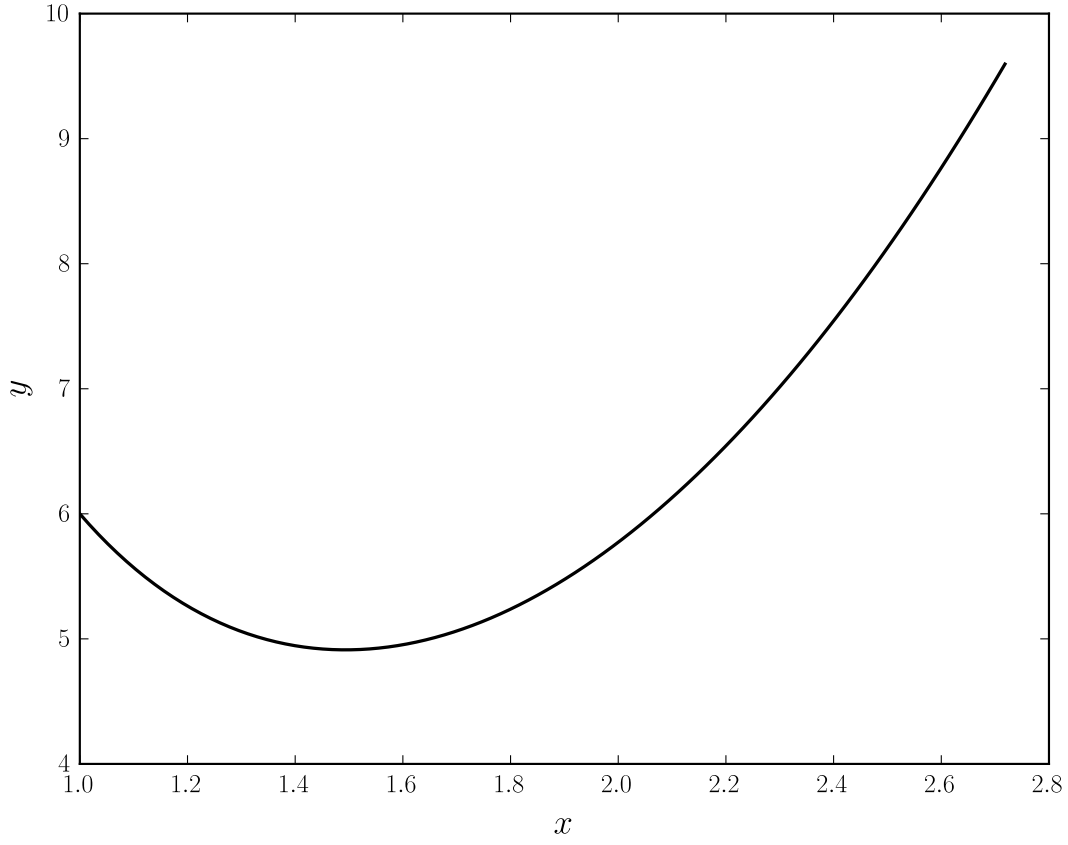


Figure 5.2: The solution of $y'' = 3 + 2y/x^2$, satisfying the boundary conditions $y(1) = 6$, $y(e) = e^2 + 6/e$.

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

$$\begin{aligned} m\dot{\vec{v}} &= -mg\vec{j} - kv^2\vec{T},\\ &= -mg(\sin\theta\vec{T} + \cos\theta\vec{N}) - kv^2\vec{T},\\ &= (-mg\sin\theta - kv^2)\vec{T} - mg\cos\theta\vec{N}. \end{aligned} \tag{5.5}$$

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

$$\begin{aligned} m\dot{\vec{v}} &= m\langle\dot{v}\cos\theta - v\sin\theta\cdot\dot{\theta}, \dot{v}\sin\theta + v\cos\theta\cdot\dot{\theta}\rangle\\ &= m(\dot{v}\cos\theta - v\sin\theta\cdot\dot{\theta})(\cos\theta\vec{T} - \sin\theta\vec{N})\\ &\quad + m(\dot{v}\sin\theta + v\cos\theta\cdot\dot{\theta})(\sin\theta\vec{T} + \cos\theta\vec{N}),\\ &= m(\vec{T}\cdot\dot{v} + \vec{N}\cdot v\cdot\dot{\theta}). \end{aligned} \tag{5.6}$$

From equations (5.5) and (5.6) 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

$$\begin{aligned}\frac{dy}{dx} &= \tan\theta,\\ \frac{dv}{dx} &= -\frac{g\sin\theta + \mu v^2}{v\cos\theta},\\ \frac{d\theta}{dx} &= -\frac{g}{v^2},\end{aligned} \tag{5.7}$$

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.

$$\begin{aligned}\frac{dy}{dx} &= \tan\theta,\\ \frac{dv}{dx} &= -\frac{g\sin\theta + \mu v^2}{v\cos\theta},\\ \frac{d\theta}{dx} &= -\frac{g}{v^2},\\ y(0) &= y(195) = 0,\\ v(0) &= 45 \text{ m/s}^2\end{aligned} \tag{5.8}$$

($g = 9.8067$ 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 5.4.

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 $h(t)$.

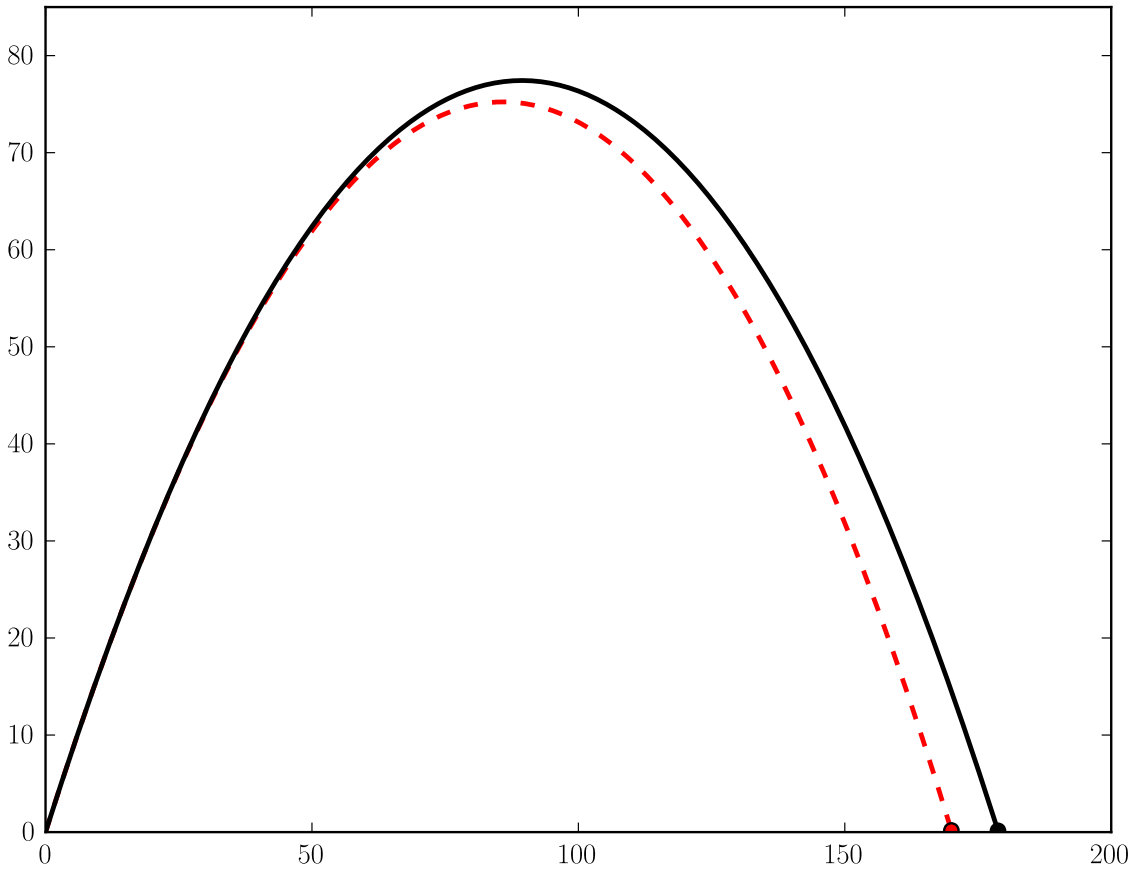


Figure 5.3: Two solutions of the system of equations (5.7), both with initial conditions $y(0) = 0$ m, $v(0) = 45$ m/s, and $\theta(0) = \pi/3$. The black curve is the trajectory of a projectile immune to air resistance ($\mu = 0$). The red curve describes the trajectory of a more realistic projectile ($\mu = .0003$).

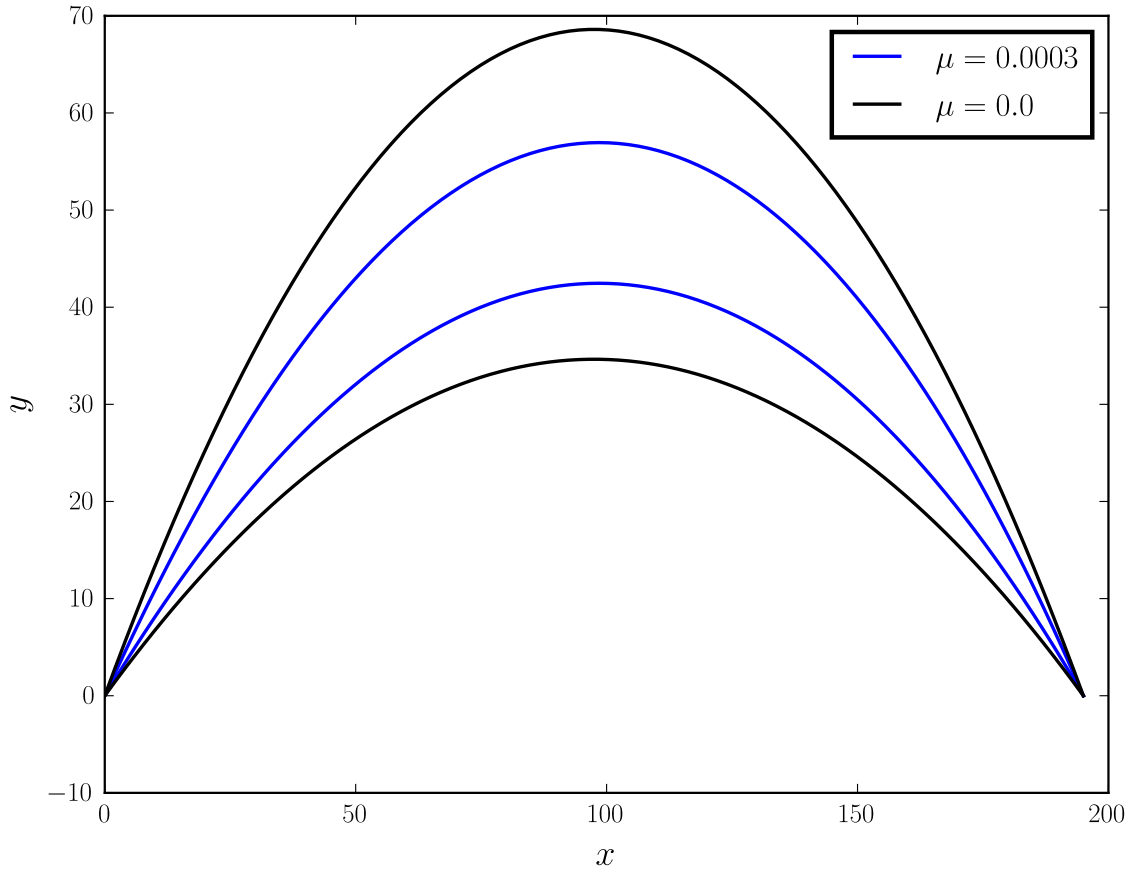


Figure 5.4: Two solutions of the boundary value problem (5.8) when the air resistance is described by the parameter $\mu = .0003$. Also both solutions when air resistance is not described in the model ($\mu = 0$).

6 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

$$L_u - \sum_{i=1}^{n} \frac{\partial L_{u_{x_i}}}{\partial x_i} = L_u - \nabla \cdot L_{\nabla u} = L_u - \operatorname{div}(L_{\nabla u}) = 0. \tag{6.1}$$

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 (6.1), we consider a related parabolic PDE,

$$\begin{aligned} u_t &= -(L_y - \operatorname{div} L_{\nabla u}), \quad t > 0, \\ u(x, 0) &= u_0(x), \quad t = 0. \end{aligned} \tag{6.2}$$

A steady state solution of (6.2) does not depend on time, and thus solves the Euler-Lagrange equation. It is often easier to evolve an initial guess using (6.2), and stop whenever its steady state is well-approximated, than to solve (6.1) directly.

Consider the energy functional

$$J[u] = \int_\Omega \|\nabla u\|^2\, dx.$$

The minimizing function u^* satisfies the Euler-Lagrange equation

$$-\operatorname{div} \nabla u = -\triangle u = 0.$$

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 (6.2) 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 (6.2) 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

$$\begin{aligned}0 &= 1 - y\frac{y''}{1+(y')^2}, \\ &= 1 + (y')^2 - yy'',\end{aligned} \tag{6.3}$$

with the gradient descent flow given by

$$\begin{aligned}&u_t = -1 - (y')^2 + yy'', \quad t > 0,\ x \in (a,b), \\ &u(x,0) = g(x), \quad t = 0, \\ &u(a,t) = y_a, \quad u(b,t) = y_b.\end{aligned} \tag{6.4}$$

Numerical Implementation

We will construct a numerical solution of (6.4) 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)
```

Problem 1. Using 20 x steps, 250 time steps, $a = -1$, $b = 1$, $alpha = 1.$, $beta = 7.$, and a final time of .2, plot the solution that minimizes (6.4). It should match figure 6.1.

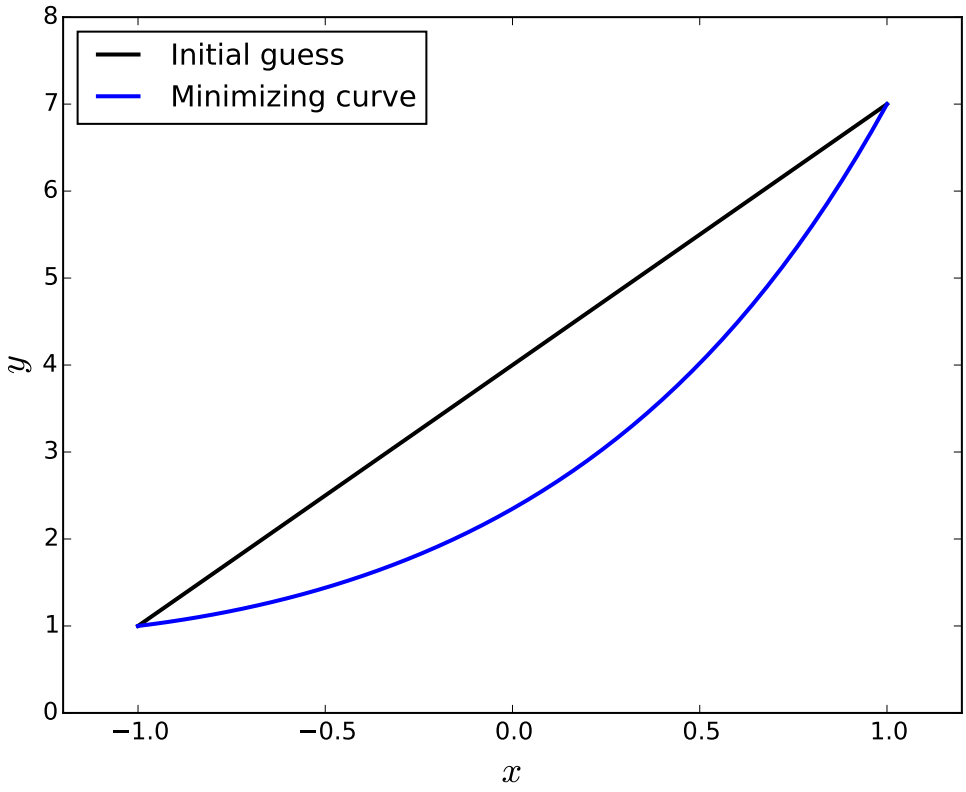


Figure 6.1: The solution of (6.3), found using the gradient descent flow (6.4).

Image Processing: Denoising

A greyscale image can be represented by a scalar-valued function $u : \Omega \to \mathbb{R}$, $\Omega \subset \mathbb{R}^2$. The following code reads an image into an array of floating point numbers, adds some noise, and saves the noisy image.

```
from numpy.random import 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 range(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

$$J[u] = \int_\Omega L(x, u, \nabla u)\, dx, \tag{6.5}$$

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 (6.5) is

$$\begin{aligned} L_u - \text{div } L_{\nabla u} &= (u - f) - \lambda \triangle u, \\ &= 0. \end{aligned}$$

and the gradient descent flow is

$$\begin{aligned} u_t &= -(u - f - \lambda \triangle u), \\ u(x, 0) &= f(x). \end{aligned} \tag{6.6}$$

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

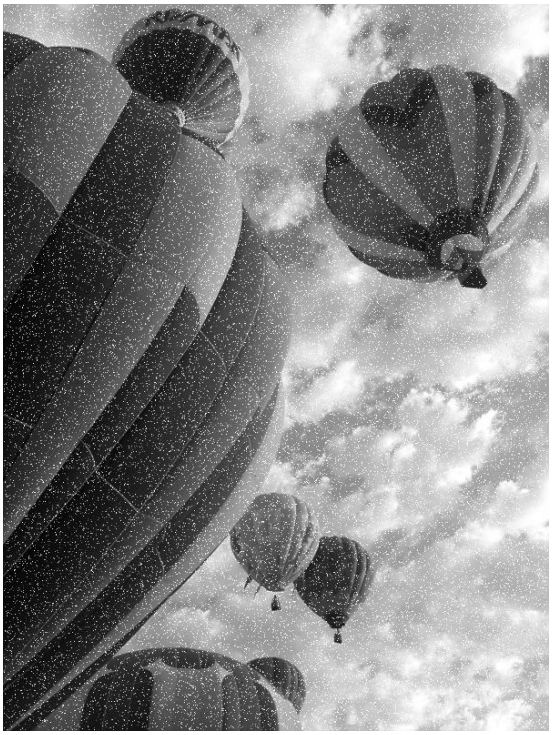


Image with white noise

Figure 6.2: 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. Plot the original image as well as the image with noise. Compare your results with Figure 6.3.

Hint: Use the function `np.roll` to compute the spatial derivatives. For example, the second derivative can be approximated at interior grid points using

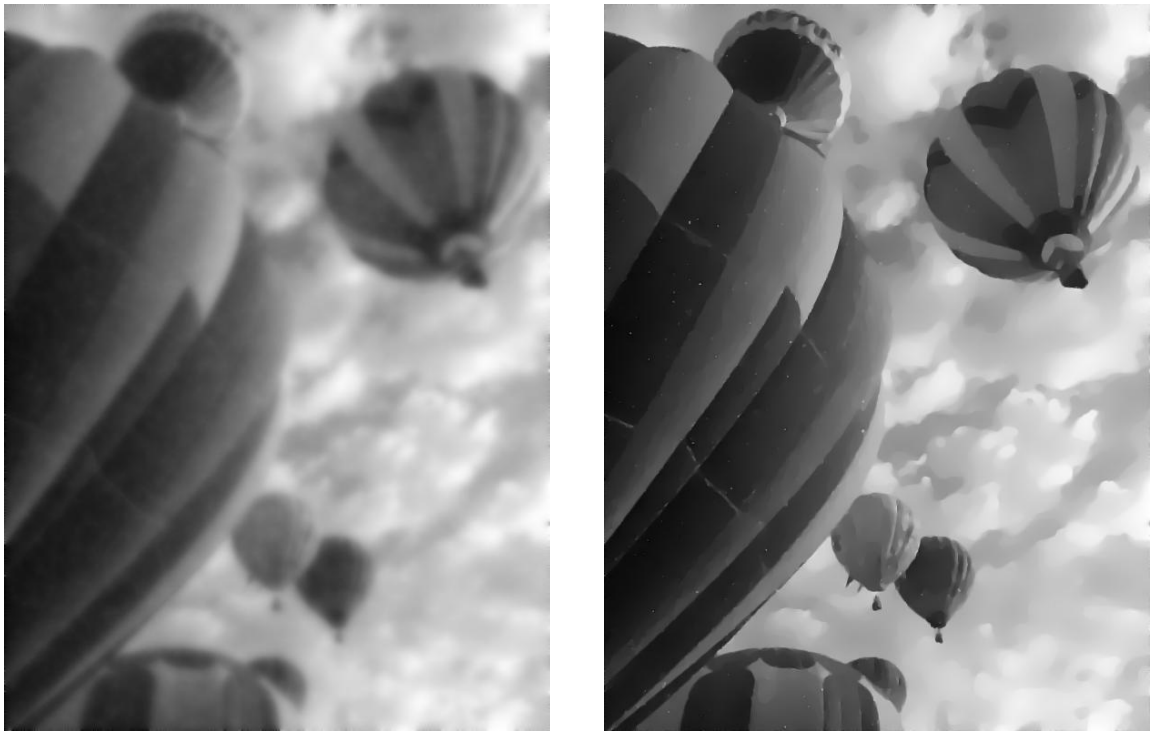
```
u_xx = np.roll(u,-1,axis=1) - 2*u + np.roll(u,1,axis=1)
```

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.

The total variation approach was originally introduced by Ruding, Osher, and Fatemi[1]. It was

[1] L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms", *Physica D.*, 1992.



Initial diffusion-based approach

Total variation based approach

Figure 6.3: The solutions of (6.6) and (6.11), found using a first order Euler step in time and centered differences in space.

formulated as follows: given a noisy image f, we look to find a denoised image u minimizing

$$\int_\Omega |\nabla u(x)|\,dx \tag{6.7}$$

subject to the constraints

$$\int_\Omega u(x)\,dx = \int_\Omega f(x)\,dx, \tag{6.8}$$

$$\int_\Omega |u(x) - f(x)|^2\,dx = \sigma|\Omega|. \tag{6.9}$$

Intuitively, (6.7) penalizes fast variations in f - this functional together with the constraint (6.8) 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 (6.9) specifying how far $u(x)$ is required to differ from the noisy image f. More precisely, (6.8) specifies that the noise in the image has zero mean, and (6.9) 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

$$F[u] = \min_{u\in BV(\Omega)} \int_\Omega |\nabla u| + \frac{\lambda}{2}(u-f)^2\,dx, \tag{6.10}$$

where $\lambda > 0$ is a fixed regularization parameter[2]. Notice how this functional differs from (6.5): $\int_\Omega |\nabla u|$ 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

$$\begin{aligned} u_t &= -\lambda(u-f) + \frac{u_{xx}u_y^2 + u_{yy}u_x^2 - 2u_x u_y u_{xy}}{(u_x^2+u_y^2)^{3/2}}, \\ u(x,0) &= f(x). \end{aligned} \tag{6.11}$$

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. Display both the diffusion-based and total variaton images of the balloon. Compare your results with Figure 6.3. 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.

7 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

$$\begin{aligned} y'(x) &= \frac{y'(t)}{x'(t)}, \\ &= \frac{\sin\theta + c}{\cos\theta}, \\ &= c\sec\theta + \tan\theta. \end{aligned} \tag{7.1}$$

The time T required to cross the river is given by

$$\begin{aligned} T &= \int_{-1}^{1} t'(x)\,dx, \\ &= \int_{-1}^{1} \frac{1}{x'(t)}\,dx \\ &= \int_{-1}^{1} \sec(\theta)\,dx. \end{aligned} \tag{7.2}$$

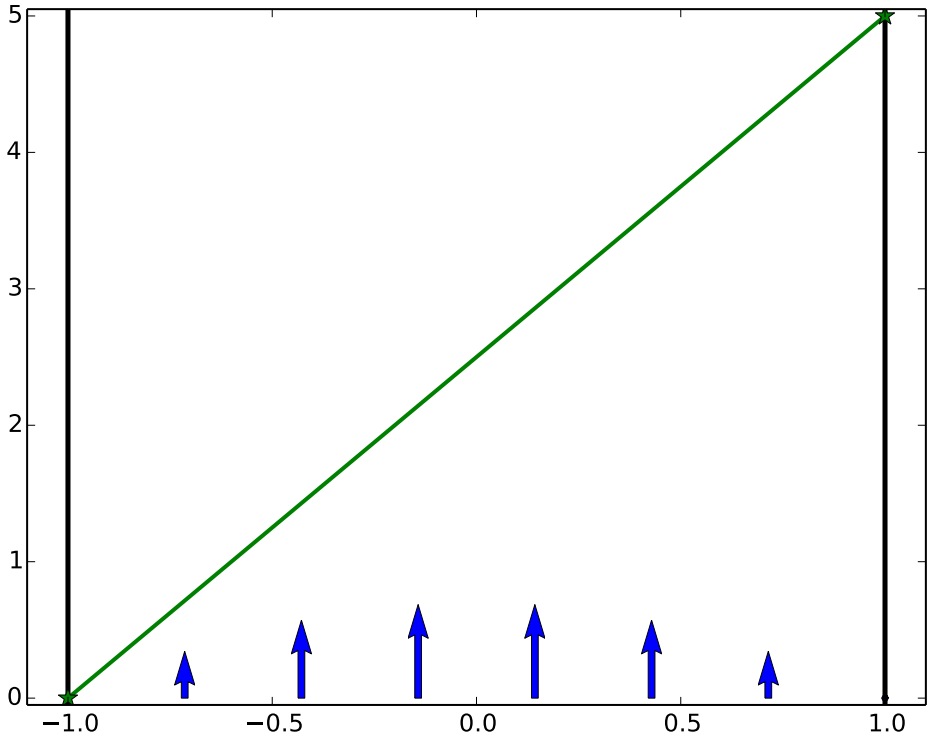


Figure 7.1: The river's current, along with a possible trajectory for the boat.

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

$$T[y] = \int_{-1}^{1} \left[\alpha(x)\sqrt{1+(\alpha y')^2(x)} - (\alpha^2 c y')(x)\right] dx, \tag{7.3}$$

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 2 python functions. The first should accept as arguments a function y, its derivative y', and an x-value, and return $L(x, y(x), y'(x))$ (where $T[y] = \int_{-1}^{1} L(x, y(x), y'(x))$). The second should use the first function to compute and return $T[y]$ for a given path $y(x)$.
(Hint: The integration for $T[y]$ can be done use an approximation method such as the midpoint method or can be done using the `quad` function from `scipy.integrate`.)

Problem 2. Let $y(x)$ be the straight-line path between $A = (-1, 0)$ and $B = (1, 5)$. Numerically calculate $T[y]$ to get an upper bound on the minimum time required to cross from A to B. Using (7.2), find a lower bound on the minimum time required to cross.
(Hint: if $G = \int f(x)dx$ and we want to minimize G, try minimizing $f(x)$.)

We look for the path $y(x)$ that minimizes the time required for the boat to cross the river, so that the function T is minimized. From the calculus of variations we know that a smooth path $y(x)$ minimizes T only if the Euler-Lagrange equation is satisfied. Recall that the Euler-Lagrange equation is

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

Since $L_y = 0$, we see that the shortest time trajectory satisfies

$$\frac{d}{dx} L_{y'} = \frac{d}{dx} \left(\alpha^3(x) y'(x) (1 + (\alpha y')^2(x))^{-1/2} - \alpha^2(x) c \right) = 0. \tag{7.4}$$

Problem 3. Numerically solve the Euler-Lagrange equation (7.4), 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 (7.1); see Figure 7.3. Note that the angle the boat should be steered is *not* described by the tangent vector to the trajectory. Consider using `scipy.optimize.root` or `scipy.interpolate.barycentric_interpolate`)

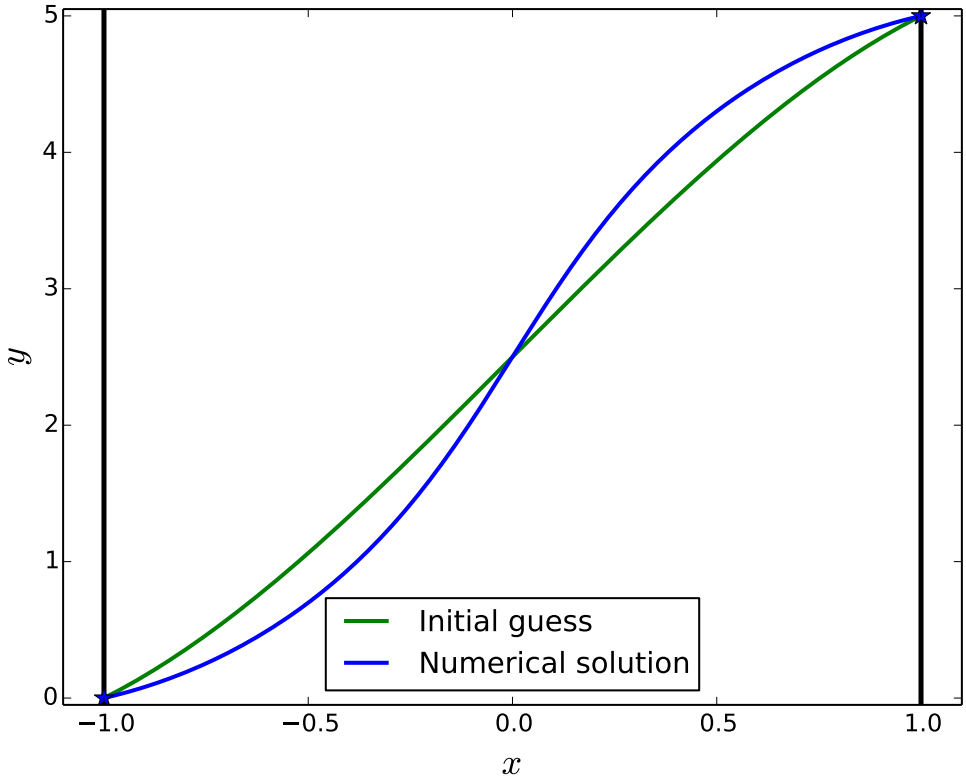


Figure 7.2: Numerical computation of the trajectory with the shortest transit time.

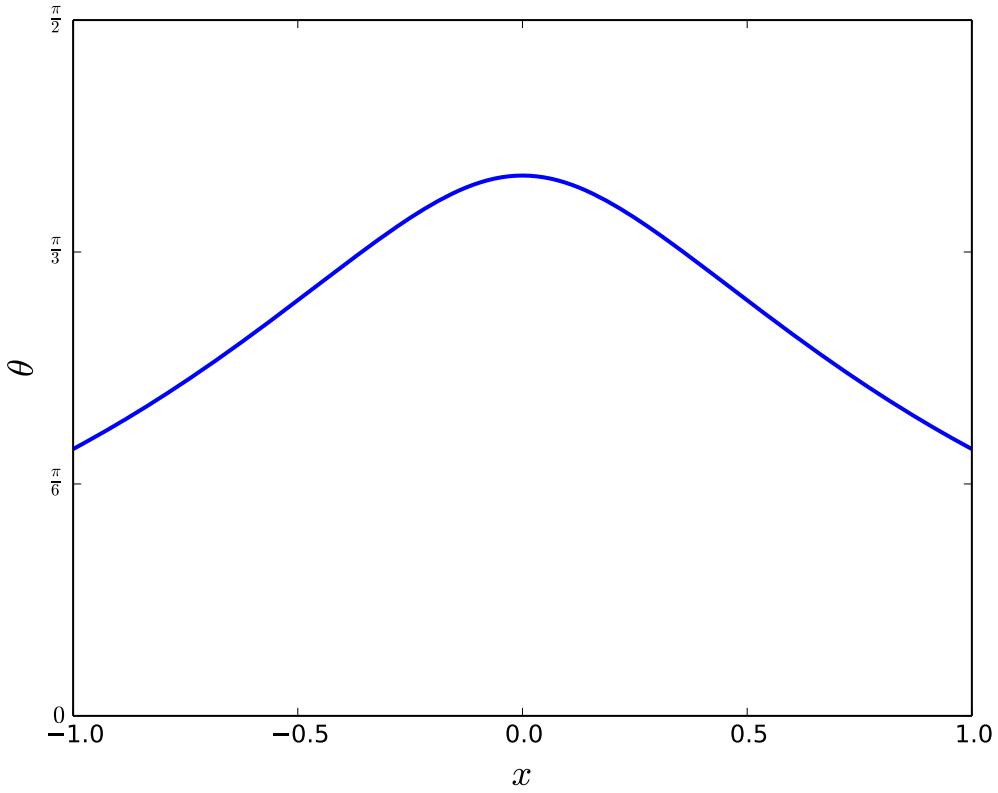


Figure 7.3: The optimal angle to steer the boat.

8 HIV Treatment Using Optimal Control

Introduction

Viruses cause many common illnesses in society today, including influenza, the common cold, and COVID-19. Viruses are obligate parasites, meaning that they must infect a host in order to replicate. After entering a host cell, viruses hijack host machinery to replicate their genome and translate their proteins. After this process, the new virus particles are assembled and lyse (break apart) the host cell to find a new host.

Mammalian immune systems are composed of two interconnected systems: the innate immune system and the adaptive immune system. While both branches of the immune system can combat viruses, the adaptive immune system is especially suited to recognize and neutralize viral infections. A major part of the adaptive immune response is helper T cells, as these cells moderate and regulate all other facets of the immune response. Helper T cells are most characterized by the presence of a receptor called CD4, which helps the cell recognize infections.

One of the most devastating viral illnesses today is acquired immunodeficiency syndrome (AIDS), caused by the human immunodeficiency virus (HIV). HIV specifically targets and replicates in helper T cells, rendering them nonfunctional and killing them. By taking out the most important regulator of the immune system, HIV makes it difficult for the body to fight infection, so sicknesses that would normally be trivial for the body to manage, such as the common cold, yeast infections, and pneumonia, become deadly.

Currently, there is no cure for HIV, and vaccines are difficult to develop. Treatments that curb the replication of HIV and help maintain healthy helper T cell population levels are available, but they are expensive and must be taken for the rest of a patient's life. Optimizing the dosage is essential to maximize the drug's effect while minimizing the cost and negative side-effects of long-term usage. In this lab, we will use optimal control to find the optimum dosage of a two-drug combination to fight HIV. In this lab we will use optimal control to find the optimal dosage of a two-drug combination[1].

Derivation of Control

We begin by defining some variables. Let T 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).

[1] *SHORT COURSES ON THE MATHEMATICS OF BIOLOGICAL COMPLEXITY*, Web. 15 Apr. 2015 http://www.math.utk.edu/ lenhart/smb2003.v2.html.

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:

$$\begin{aligned} \frac{dT(t)}{dt} &= s_1 - \frac{s_2 V(t)}{B_1 + V(t)} - \mu T(t) - kV(t)T(t) + u_1(t)T(t), \\ \frac{dV(t)}{dt} &= \frac{gV(t)}{B_2 + V(t)}(1 - u_2(t)) - cV(t)T(t). \end{aligned} \tag{8.1}$$

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:

$$\begin{aligned} \lambda_1' &= -\frac{\partial H}{\partial T} = -1 + \lambda_1[\mu + kV^* - u_1^*] + \lambda_2 cV^*, \\ \lambda_2' &= -\frac{\partial H}{\partial V} = \lambda_1 \Big[\frac{B_1 s_2}{(B_1 + V^*)^2} + kT^* \Big] - \lambda_2 \Big[\frac{B_2 g(1 - u_2^*)}{(B_2 + V^*)^2} - cT^* \Big]. \end{aligned}$$

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:

$$\frac{\partial H}{\partial u_1} = -2A_1 u_1^*(t) + \lambda_1 T^*(t) = 0$$

[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

$$\frac{\partial H}{\partial u_2} = -2A_2u_2^*(t) + \lambda_2\left[\frac{-gV^*(t)}{B_2 + V^*(t)}\right] = 0$$

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

$$\begin{aligned} T' &= s_1 - \frac{s_2 V}{B_1 + V} - \mu T - kVT + \min\{\max\{a_1, \frac{1}{2A_1}(\lambda_1 T)\}, b_1\}T, \\ V' &= \frac{g(1 - \min\{\max\{a_2, \frac{-\lambda_2}{2A_2}\frac{gV}{B_2+V}\}, b_2\})V}{B_2 + V} - cVT \end{aligned} \tag{8.2}$$

$$\begin{aligned} \lambda_1' &= -1 + \lambda_1\left[\mu + kV - \min\{\max\{a_1, \frac{1}{2A_1}(\lambda_1 T)\}, b_1\}\right] + \lambda_2 cV, \\ \lambda_2' &= \lambda_1\left[\frac{B_1 s_2}{(B_1 + V)^2} + kT\right] - \lambda_2\left[\frac{B_2 g(1 - \min\{\max\{a_2, \frac{-\lambda_2}{2A_2}\frac{V}{B_2+V}\}, b_2\})}{(B_2 + V)^2} - cT\right], \end{aligned} \tag{8.3}$$

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 (8.1) from the beginning of the lab and not the equations (8.2) 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 = 2000
```

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

$$\dot{\hat{\lambda}}_1(t) = \lambda_1(t_f - t)\left(-\mu - kV(t_f - t) + u_1(t_f - t)\right) - c\lambda_2(t_f - t)V(t_f - t) + 1,$$
$$\dot{\hat{\lambda}}_2(t) = -\lambda_1(t_f - t)\left(\frac{s_2 B_1}{(B_1 + V(t_f - t))^2} + kT(t_f - t)\right) + \lambda_2(t_f - t)\left(\frac{gB_2(1 - u_2(t_f - t))}{(B_2 + V(t_f - t))^2} - cT(t_f - t)\right).$$

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.

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(...)
```

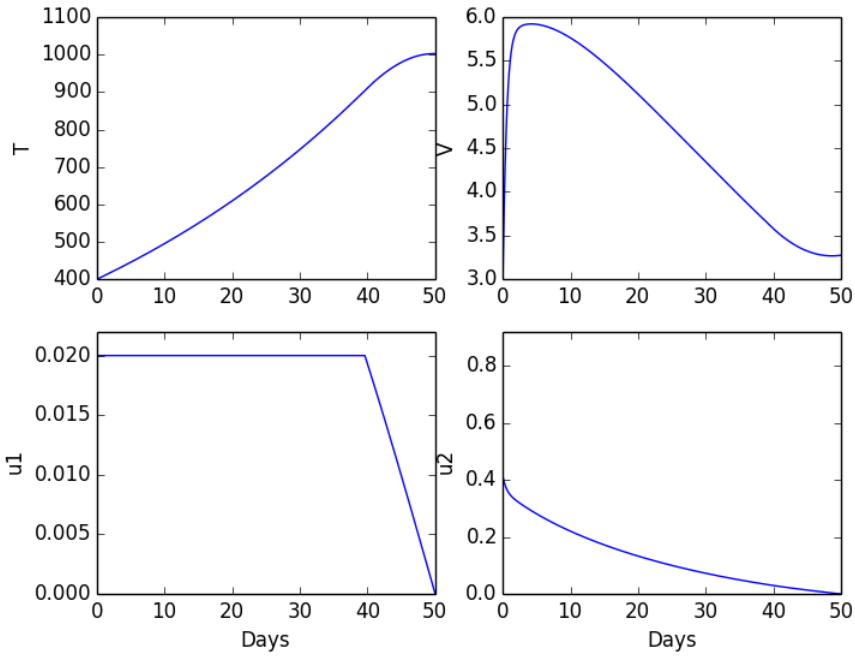


Figure 8.1: The solution to Problem 3.

```
#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 8.1.

Patients usually take several different classes of drugs at a time to prevent HIV from replicating and progressing into AIDS. Reverse transcriptase inhibitors prevent the HIV genome from inserting itself into the host genome. These prevent helper T cell death by lowering the number of HIV particles in the body. Protease inhibitors prevent the activation of HIV proteins that are needed for replication. Fusion inhibitors can be taken early in the course of HIV infection and prevent the entry of HIV into helper T cells. In reality, there are many unique drugs in each class, all with known and unknown interactions and side effects. Physicians rotate through drugs to help their patients have a positive outcome and to prevent the virus from becoming resistant to any one drug.

9 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

$$\mathcal{F}(U)_t = -\frac{i}{2}\vec{k}\mathcal{F}\left(\mathcal{F}^{-1}(\mathcal{F}(U))^2\right) + i\vec{k}^3\mathcal{F}(U) \tag{9.1}$$

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

$$Y_t = -\frac{i}{2}\vec{k}\mathcal{F}\left(\mathcal{F}^{-1}(Y)^2\right) + i\vec{k}^3 Y \tag{9.2}$$

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 (9.2) 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 = .02*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 9.1.

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 9.2.

[a] This problem is solved in *Spectral Methods in MATLAB*, by Trefethen.

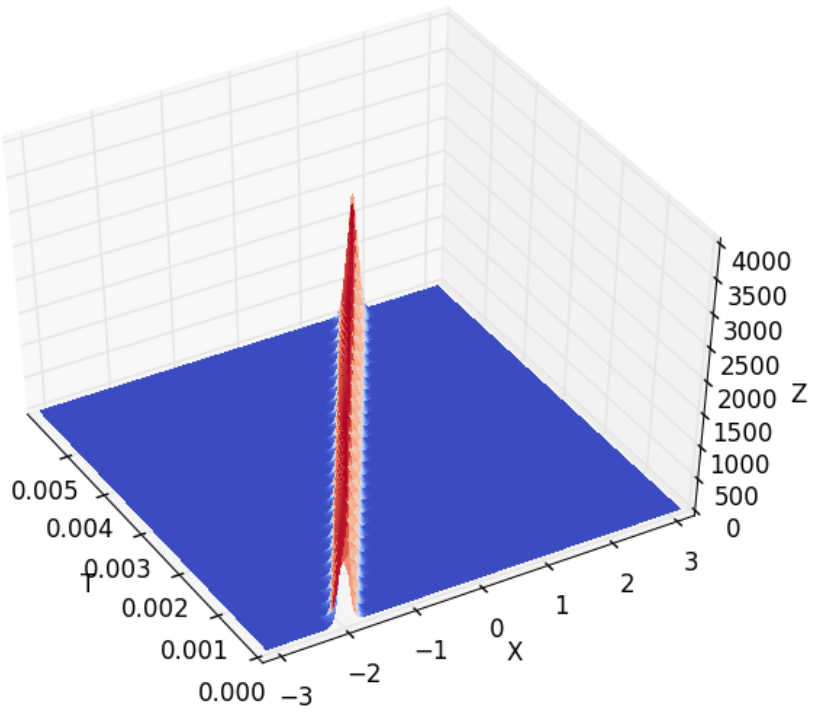


Figure 9.1: The solution to Problem 1.

Problem 3. Consider again equation (9.2). 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.

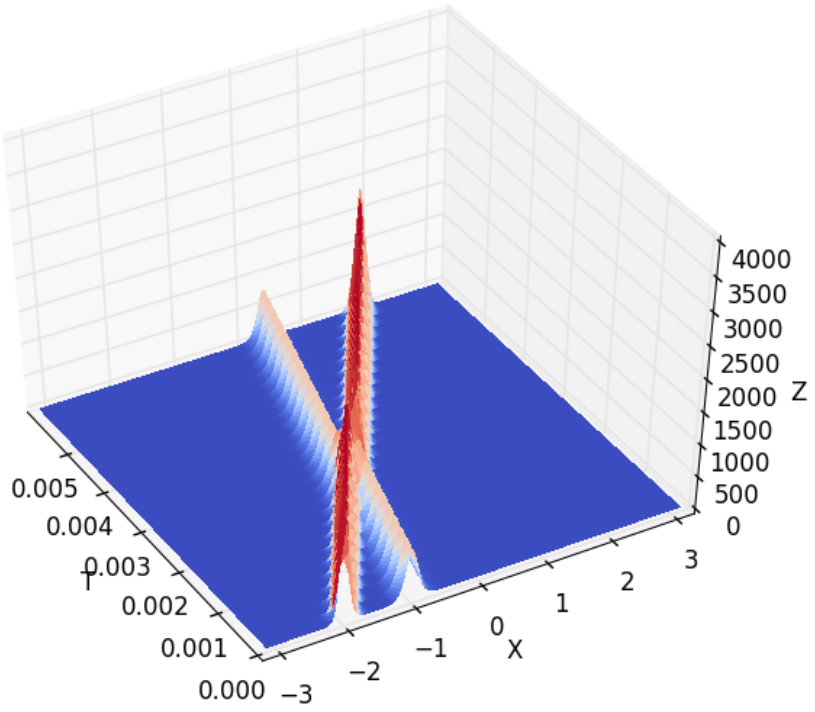


Figure 9.2: The solution to Problem 2.

10 Obstacle Avoidance

Lab Objective: *Solve boundary value problems that arise when using Pontryagin's Maximum principle.*

Pontryagin's Maximum Principle

Now that we understand how to solve boundary value problems, we can apply this to solve optimal control problems. Pontryagin's Maximum Principle is a very common way to formulate control problems as BVPs.

Fixed Time, Fixed Endpoint

We will begin with the more simple fixed time horizon problems. Fixed time horizon problems are commonly reformulated as boundary value problems, and we can apply what we have already learned about solving BVPs to make these problems easier to solve. We introduce fixed time horizon problems with a cost functional of the following form

$$J(u) = \int_{t_0}^{t_f} L(t, s(t), u(t))dt + K(t_f, s_f), \tag{10.1}$$

where t_0 and t_f are fixed. In this functional, $L(t, s(t), u(t))$ represents the cost of a certain path determined by the control u, and $K(t_f, s_f)$ is the terminal cost. We also have that

$$\dot{s} = f(t, s, u),\ s_0 = s(t_0),\ s_f = s(t_f). \tag{10.2}$$

In these equations t is time, s is the state variable, and u is the control variable. The maximum principle also uses the Hamiltonian equation

$$H(t, s, u, p) = \langle p, f(t, s, u)\rangle - L(t, s, u), \tag{10.3}$$

where p is a newly introduced variable called the costate. This Hamiltonian is then used to define an ODE system. This first equation defines a costate ODE system

$$\dot{p}^* = -H_s(t, s^*, u^*, p^*), \tag{10.4}$$

where a variable marked with an asterisk is the optimal choice of that variable, meaning that equation 10.4 is only true for the optimal state s^*, costate p^*, and control u^* functions. This next equation will allow us to solve for the control in terms of the state and costate

$$0 = H_u(t, s^*, u^*, p^*), \quad \forall t \in [t_0, t_f]. \tag{10.5}$$

The combination of these equations will allow us to create a BVP that will solve for the optimal control u^* and the associated states s^*. Our ODE comes from 10.2, 10.4, and 10.5, and the boundary values will come from our initial and final conditions on s.

Problem 1. Given the following cost functional and boundary conditions, use the ODEs found in 10.2, 10.4, and 10.5 to solve for and plot the optimal path and acceleration.

$$J(u) = \int_0^{30} x^2 + \frac{2\pi}{5} u^2 dt$$
$$s(t) = \begin{bmatrix} x(t) \\ x'(t) \end{bmatrix}, s(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad s(30) = \begin{bmatrix} 16 \\ 10 \end{bmatrix}$$

Plot your solutions for the optimal $x(t)$ and $u(t)$.

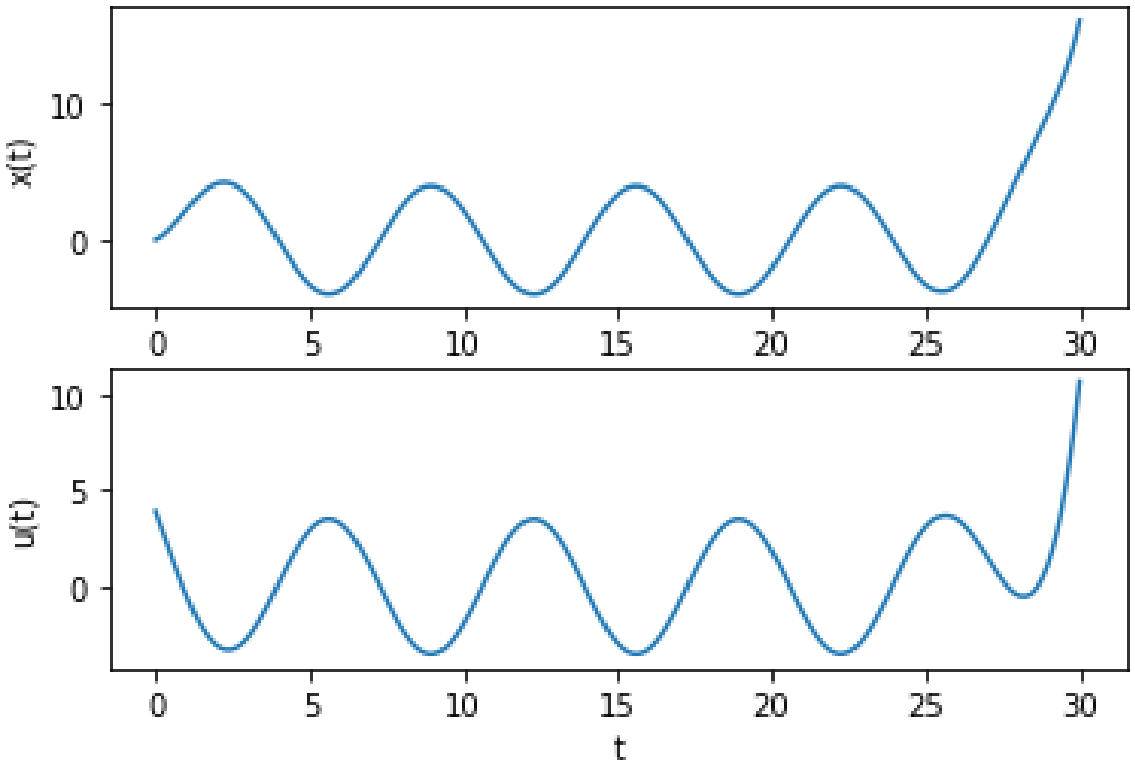


Figure 10.1: Solution to problem 1

Avoiding Collision

We now expand upon the technique learned above by adding an obstacle in our path. One area of application that relies heavily on optimal control is autonomous driving. A common problem in autonomous driving is the avoidance of obstacles. In this section we will outline a naïve solution to obstacle avoidance with a fixed time horizon.

First we can begin by defining our state variable s. We will want to understand the position and velocity at a given time so we will define the following state variable

$$s(t) = \begin{bmatrix} x(t) \\ y(t) \\ \dot{x}(t) \\ \dot{y}(t) \end{bmatrix} = \begin{bmatrix} s_1(t) \\ s_2(t) \\ s_3(t) \\ s_4(t) \end{bmatrix}, \tag{10.6}$$

which allows us to track those states in $\mathbb{R}^2$.

We can then establish the ODE defined in equation 10.2 by examining $\dot{s}(t)$

$$\dot{s}(t) = \begin{bmatrix} \dot{s_1}(t) \\ \dot{s_2}(t) \\ \dot{s_3}(t) \\ \dot{s_4}(t) \end{bmatrix} = \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \ddot{x}(t) \\ \ddot{y}(t) \end{bmatrix},$$

and if we define our control u_1 and u_2 to be acceleration in the x and y directions respectively, then we have

$$\dot{s}(t) = f(t, s, u) = \begin{bmatrix} s_3(t) \\ s_4(t) \\ u_1(t) \\ u_2(t) \end{bmatrix}. \tag{10.7}$$

Next we will define an obstacle. Since we are using integration to define cost, a reasonable way to model an obstacle in this problem would be to use a function. It would be helpful if this function is malleable, allowing us to reposition and resize it, based on the needs of the specific situation. This function also needs to have a large, preferably positive, value in a concentrated location, and it needs to vanish relatively quickly. A decent selection could be a function based on an ellipse, such as this function

$$C(x, y) = \frac{W_1}{((x - c_x)^2/r_x + (y - c_y)^2/r_y)^\lambda + 1}. \tag{10.8}$$

With the function 10.8 we can manipulate the center by changing c_x and c_y, and we can control the size by changing r_x and r_y. Changing the constant W_1 allows us to change the relative penalty of occupying the same location as the obstacle, and a reasonable value for λ will control the vanishing rate. We will also include a term in the cost functional that weights against high acceleration. This will allow us to model the real world more accurately, though the term we will be using is not a perfect representation of real world acceleration limitations. Our cost functional is the following

$$J(u) = \int_{t_0}^{t_f} 1 + C(x(t), y(t)) + W_2 \left|u(t)\right|^2 dt, \tag{10.9}$$

where $W_2 > 0$ defines the relative penalty of high acceleration. This functional will penalize passing near the obstacle and high levels of acceleration.

With the cost functional defined, we can now create the Hamiltonian and the rest of our BVP. We get the following Hamiltonian

$$H(t, p, s, u) = p_1 s_3 + p_2 s_4 + p_3 u_1 + p_4 u_2 - \left(1 + C(x, y) + W_2 \left|u(t)\right|^2\right), \tag{10.10}$$

which gives the following costate ODE by equation 10.4

$$\dot{p} = \begin{bmatrix} \dot{p}_1 \\ \dot{p}_2 \\ \dot{p}_3 \\ \dot{p}_4 \end{bmatrix} = \begin{bmatrix} C_x(x,y) \\ C_y(x,y) \\ -p_1 \\ -p_2 \end{bmatrix}. \tag{10.11}$$

Since we're given $H_u = 0$ in equation 10.5, then we also have the following relations

$$\begin{aligned} u_1(t) &= \frac{1}{2W_2} p_3(t) \\ u_2(t) &= \frac{1}{2W_2} p_4(t). \end{aligned} \tag{10.12}$$

Problem 2. Using the ODEs found in 10.7 and 10.11, the obstacle function 10.8, and the following boundary conditions and parameters solve for and plot the optimal path.

$$\begin{aligned} t_0 = 0, \quad t_f &= 20 \\ (c_x, c_y) &= (4, 1) \\ (r_x, r_y) &= (5, .5) \\ \lambda &= 20 \\ s_0 = \begin{bmatrix} 6 \\ 1.5 \\ 0 \\ 0 \end{bmatrix}, \quad s_f &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{aligned}$$

You will need to choose a W_1 and W_2 which allow the solver to find a valid path. If these parameters are not chosen correctly, the solver may find a path which goes through the obstacle, not around it. Plot the obstacle using `plt.contour()` to be certain path doesn't pass through the obstacle.

Hint: The default for a parameter of `solve_bvp()` called `max_nodes` is not large enough. Try at least `max_nodes = 30000`. You may also find it helpful to use the function `partial` from the module `functools` to preset the parameters for the functions you will be using. If you are struggling to find viable values for W_1 and W_2, try $W_1 \in (1, 40)$ and $W_2 \in (0, 9)$.

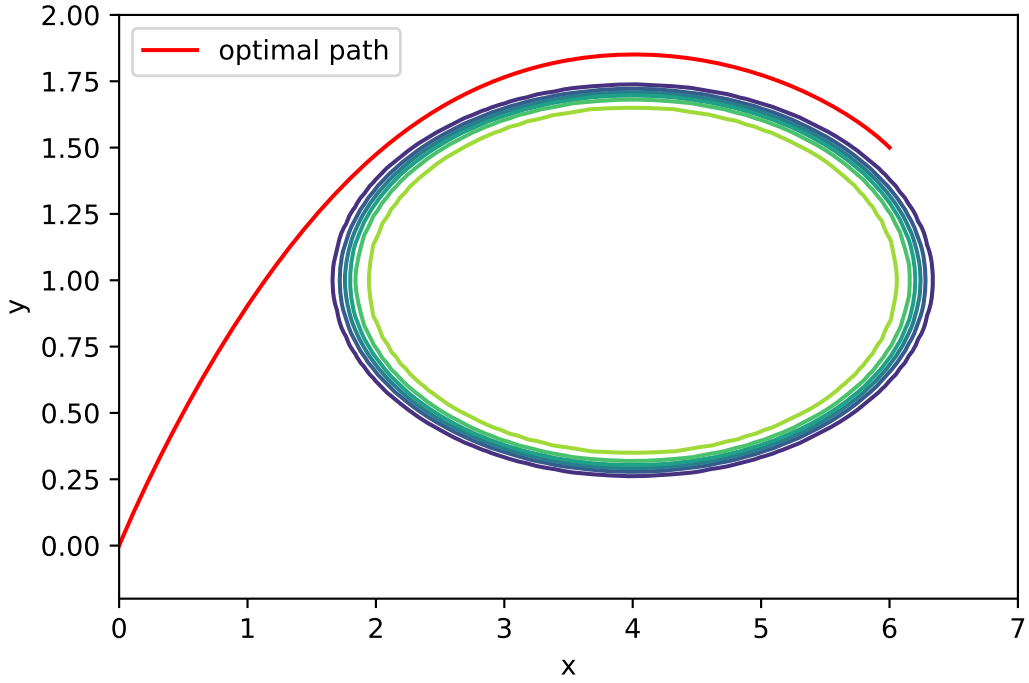


Figure 10.2: Solution to problem 2 for certain choice of parameters

Free Time Horizon Problems

In the previous sections and problems, we were working with BVPs that had a fixed start time t_0, and a fixed end time t_f. However, we may also encounter systems that have a free end time. In order to solve these problems we will need to make some alterations to the problem. First we will perform a change of basis so that we can work with a fixed end time. Consider the following system

$$\dot{x}(t) = f(x(t), t) \quad t \in [0, t_f],$$

we can do the following change of basis for the time variable

$$\begin{aligned} & t = t_f \hat{t} \\ \implies & \frac{d}{dt} = \frac{d}{d\hat{t}} \frac{d\hat{t}}{dt} \\ \implies & \frac{d}{dt} = \frac{d}{d\hat{t}} \frac{1}{t_f}. \end{aligned}$$

We can now define $z(\hat{t}) := x(t_f \hat{t})$ which gives us the following new system

$$\dot{z}\left(\hat{t}\right) = t_f f\left(z\left(\hat{t}\right), \hat{t}\right) \quad \hat{t} \in [0, 1].$$

This system can be solved in the same way we solve the fixed time horizon problems. But you may notice that we now have an extra unknown parameter, the final time. Because of this, a free time horizon problem will need one more boundary value to make the system solvable.

So lets examine the earlier example as a free time horizon problem. We start with the ODE system we derived from the second order equation, replacing the fixed final time with a free final time and including the needed third boundary condition

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ cos(t) - 9y_1 \end{bmatrix}, \quad y_1(0) = 5/3, \quad y_2(0) = 5, \quad y_1(t_f) = -\frac{5}{3}.$$

Now we make the coordinate change giving the following system

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}' = t_f \begin{bmatrix} z_2 \\ cos(\hat{t}) - 9z_1 \end{bmatrix}, \quad z_1(0) = 5/3, \quad z_2(0) = 5, \quad z_1(1) = -\frac{5}{3}. \tag{10.13}$$

Now we can solve this system using `solve_bvp` in python. The new argument `p` that we have included in `ode()` and `bc()` is an `ndarray` that contains our parameter t_f.

```
def ode(t,y,p):
    ''' define the ode system '''
    return p[0]*np.array([y[1], np.cos(t) - 9*y[0]])

def bc(ya,yb,p):
    ''' define the boundary conditions '''
    return np.array([ya[0] - (5/3), ya[1] - 5, yb[0] + 5/3])

# give the time domain
t_steps = 100
t = np.linspace(0,1,t_steps)

# give an initial guess
y0 = np.ones((2,t_steps))
p0 = np.array([6])

# solve the system
sol = solve_bvp(ode, bc, t, y0, p0)
```

The attribute `sol.p[0]` will give the final time the solver found.

When plotting we need to make sure that we remember that $x(t_f\hat{t}) = z(\hat{t})$, so we plot in the following way

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

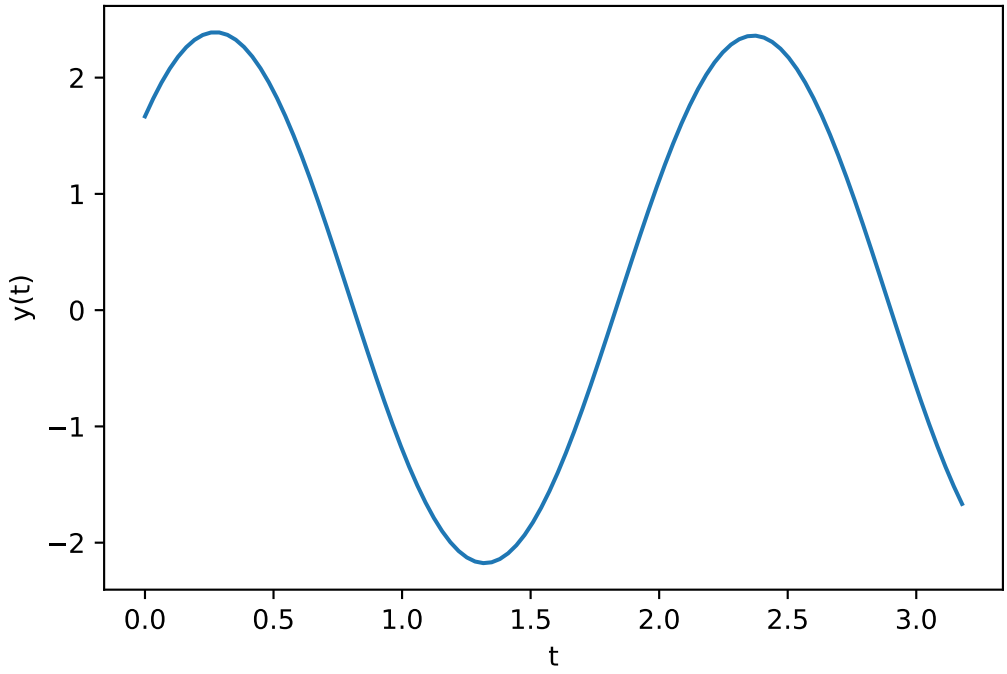


Figure 10.3: The solution to 10.13

Problem 3. Solve the following boundary value problem:

$$y'' + 3y = \sin(t)$$
$$y(0) = 0, \quad y(t_f) = \frac{\pi}{2}, \quad y'(t_f) = \frac{1}{2}\left(\sqrt{3}\pi \cot(\pi\sqrt{75}) - 1\right).$$

Plot your solution. What t_f did the solver find?

Hint: If you are struggling to find viable values for W_1 and W_2, try $W_1 \in (4, 12)$ and $W_2 = 0.1$.

Free Time, Fixed Endpoint Control Problems

Now that we understand how to formulate free time horizon problems, we can modify our optimal control BVP to become a free time horizon problem. This is actually the best way to formulate many optimal control problems, as we usually don't know exactly how long it takes to traverse the optimal path. The methodology is exactly the same as we used in the last problem, we only need to find the extra boundary value which will allow us to make the end time a free variable.

To find this extra boundary value we will use the fact that the Hamiltonian is 0 for all t along the optimal path. It is standard to use the final time as the representative so we will assert that

$$H(t_f, p(t_f), s(t_f), u(t_f)) = 0. \tag{10.14}$$

You may notice that when you solve an optimal control problem as a free end time BVP, the optimal path you get is different than what you found when it was a fixed end time BVP. This is because the free end time solution actually arrives faster. The solution found in the fixed end time formulation is the optimal path for a certain fixed end time, but it may not be the overall fastest path that avoids the obstacle.

Problem 4. Refactor your code from problem 2 to create a free end time BVP and use a new boundary value derived from 10.14. Plot the solution you found. What is the optimal time?

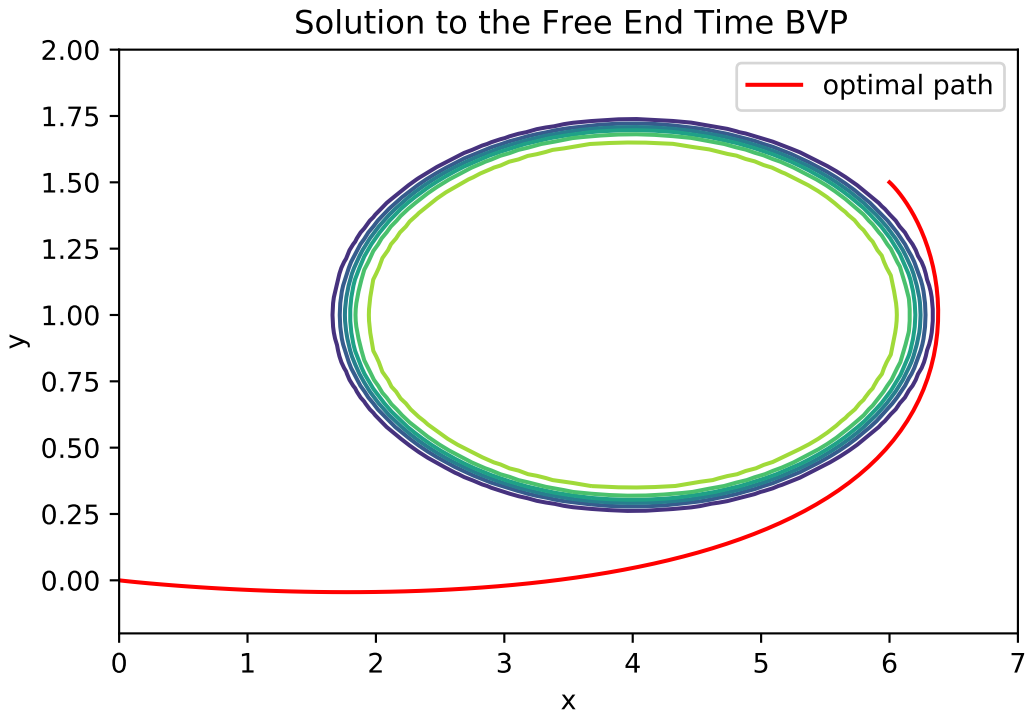


Figure 10.4: The solution to 4

11 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

$$J[\theta] = \int_{t_0}^{t_1} L, \tag{11.1}$$

where the Lagrangian $L = T - U$ is the difference between the kinetic and potential energies of the bob.

The kinetic energy of the bob is given by $mv^2/2$, where v is the velocity of the bob. In terms

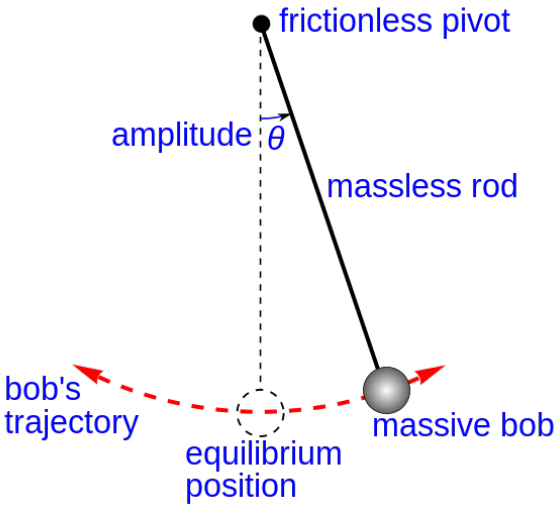


Figure 11.1: The frame of reference for the simple pendulum problem.

of θ, the kinetic energy becomes

$$\begin{aligned} T &= \frac{m}{2}v^2 = \frac{m}{2}(\dot{x}^2 + \dot{y}^2), \\ &= \frac{m}{2}((l\cos(\theta)\dot{\theta})^2 + (l\sin(\theta)\dot{\theta})^2), \\ &= \frac{ml^2\dot{\theta}^2}{2}. \end{aligned} \tag{11.2}$$

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:

$$\begin{aligned} 0 &= L_\theta - \frac{d}{dx}L_{\dot{\theta}}, \\ &= -mgl\sin\theta - ml^2\ddot{\theta}, \\ &= \ddot{\theta} + \frac{g}{l}\sin\theta. \end{aligned} \tag{11.3}$$

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).

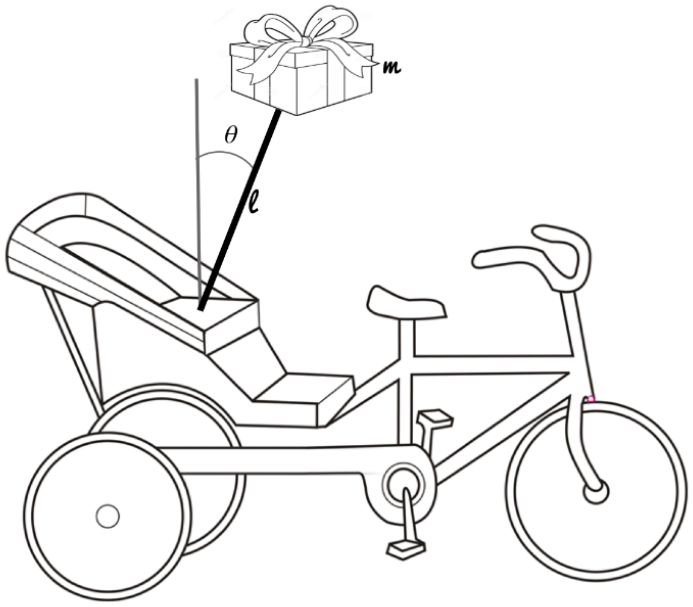


Figure 11.2: The inverted pendulum problem on a mobile rickshaw with a present suspended above.

By Hamilton's Principle, the path (x, θ) of the rickshaw and the present minimizes the functional

$$J[x, \theta] = \int_{t_0}^{t_1} L, \tag{11.4}$$

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

$$\begin{aligned} T &= \frac{1}{2}Mv_1^2 + \frac{1}{2}mv_2^2, \\ &= \frac{1}{2}M\dot{x}^2 + \frac{1}{2}m((\dot{x} - l\dot{\theta}\cos\theta)^2 + (-l\dot{\theta}\sin\theta)^2), \\ &= \frac{1}{2}(M+m)\dot{x}^2 + \frac{1}{2}ml^2\dot{\theta}^2 - ml\dot{x}\dot{\theta}\cos\theta. \end{aligned} \tag{11.5}$$

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:

$$\begin{aligned} \frac{\partial L}{\partial x} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}} &= F, \\ \frac{\partial L}{\partial \theta} - \frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} &= 0. \end{aligned} \tag{11.6}$$

After expanding (11.6) we see that $x(t)$ and $\theta(t)$ satisfy

$$\begin{aligned} F &= (M+m)\ddot{x} - ml\ddot{\theta}\cos\theta + ml\dot{\theta}^2\sin\theta, \\ l\ddot{\theta} &= g\sin\theta + \ddot{x}\cos\theta. \end{aligned} \tag{11.7}$$

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 (11.7) 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

$$\begin{aligned} \ddot{x} &= \frac{1}{M}F - \frac{m}{M}g\theta, \\ \ddot{\theta} &= \frac{1}{Ml}F + \frac{g}{Ml}(M+m)\theta. \end{aligned} \tag{11.8}$$

We will now write (11.8) 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

$$\begin{aligned} J[z] &= \int_0^\infty (q_1 x_1^2 + q_2 x_2^2 + q_3 \theta_1^2 + q_4 \theta_2^2 + r u^2)\, dt \\ &= \int_0^\infty z^T Q z + u^T R u \, dt \end{aligned} \tag{11.9}$$

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 (11.9) 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)

$$PA + A^T P + Q - PBR^{-1}B^T P = 0. \tag{11.10}$$

The evolution of the optimal state vector $\tilde{z}$ can then be described by [1]

$$\dot{\tilde{z}} = (A - BR^{-1}B^T P)\tilde{z}. \tag{11.11}$$

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.
```

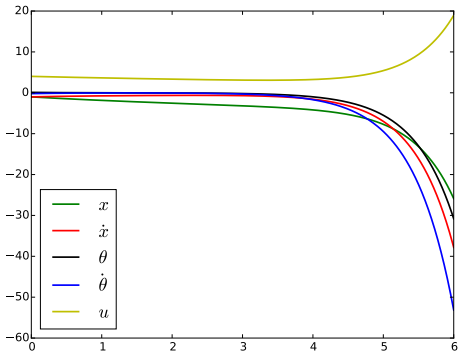
compute the eigenvalues of $A - BR^{-1}B^TP$. Are any of the eigenvalues positive? Consider differential equation (11.11) governing the optimal state $\tilde{z}$. Using this value of P, will we necessarily have $\tilde{z} \to 0$?

Problem 3. Write the following function that implements the LQR solution described earlier. For the IVP solver, you can use your own or you may use the function `odeint` from `scipy.integrate`.

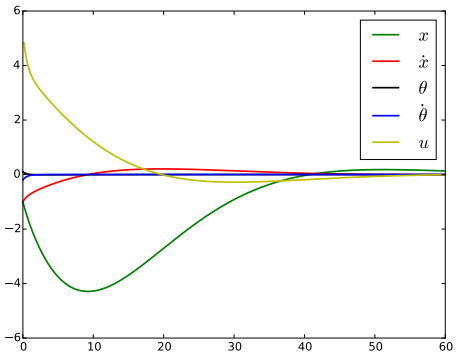
```
def rickshaw(tv, X0, A, B, Q, R, P):
    '''
    Parameters:
    ----------
    tv  : ndarray of time values, with shape (n+1,)
    X0  : Initial conditions on state variables
    A, Q: ndarrays of shape (4,4)
    B   : ndarray of shape (4,1)
    R   : ndarray of shape (1,1)
    P   : ndarray of 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
```

Notice that we have no information on how many solutions (11.10) possesses. In general there may be many solutions. We hope to find a unique solution P that is *stabilizing*: the eigenvalues of



P is found using `scipy.optimize.root`.



P is found using `solve_continuous_are`.

Figure 11.3: The solutions of Problem 4.

$A - BR^{-1}B^TP$ 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 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 11.3.