Clay Freeman — 4 May 2018
ME 306 : Computer Aided Engineering — Take-Home Final

1. A steady-state heat balance for a rod can be represented as:

$$\frac{d^2T}{dx^2} - 0.15T = 0$$

Methods:

a. Obtain the analytical solution for a 10 m rod with T(0) = 240 and T(10) = 150

$$T'' - 0.15T = 0$$
$$m^2 = 0.15$$
$$m = \pm\sqrt{0.15}$$
$$T(x) = C_1 e^{x\sqrt{0.15}} + C_2 e^{-x\sqrt{0.15}}$$
$$T(0) = 240 = C_1 + C_2$$
$$T(10) = 150 = C_1 e^{10\sqrt{0.15}} + C_2 e^{-10\sqrt{0.15}}$$
$$150 = (240 - C_2) e^{10\sqrt{0.15}} + C_2 e^{-10\sqrt{0.15}}$$
$$150 - 240 e^{10\sqrt{0.15}} = C_2 e^{-10\sqrt{0.15}} - C_2 e^{10\sqrt{0.15}}$$
$$-11390.55081 = C_2 (e^{-10\sqrt{0.15}} - e^{10\sqrt{0.15}})$$
$$C_2 = 236.9830561$$
$$C_1 = 3.016943933$$

Results:

$$T(x) = 3.02 e^{x\sqrt{0.15}} + 236.98 e^{-x\sqrt{0.15}}$$

b. Use the shooting method to solve the problem

Methods:

The issue with boundary problems is that we know something about the state of the beginning and end of the integral, but we are missing information about the initial rate to set up the ODE as an initial value problem. Using initial guesses of 0 and -100, the ode45 solver produced a solution of 5772.77 degrees and -432.37 degrees respectively. While these answers are completely off, a better estimate for the second initial condition can be found through linear interpolation. Running the ode solver a third time using the interpolated initial condition of 90.61 yields an error of $3.8*10^{-5}$. Figure 1 shows a depiction of the results using initial condition guesses of -85 and -95. Error is calculated as the sum of the errors from the analytical solution for each point calculated.

Results:

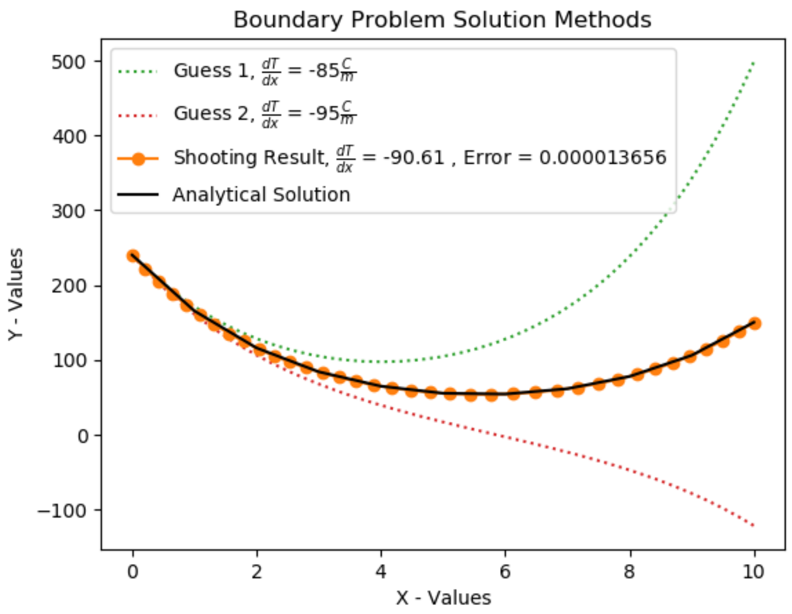

Figure 1: The Shooting Method takes the results from two educated guesses and does linear interpolation to find a better guess for the initial conditions. This turns out to be an accurate estimator.

c. Use the finite-difference approach with Δx = 1 to solve the problem.

Methods:

Using the equation for the second derivative of the centered finite divided difference the bar can be modeled as a tridiagonal system where each row represents a segment along the length.

$$f''(x) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} = \frac{(1) - 2(0 - 0.15) + (1)}{(1)^2} = 2.15$$

The Thomas algorithm is a computationally efficient way to solve tridiagonal matrices based on easy LU decomposition. If we set the primary diagonal, f = 2.15, and the top and bottom coefficients set to 1, the algorithm solves for the system at each segment. Figure 2 shows the analytical solution and the finite difference method. This method is faster, but produces a greater amount of error. Initially the raw output generated positive and negative values. The error was calculated from the absolute values from the output.

Results:

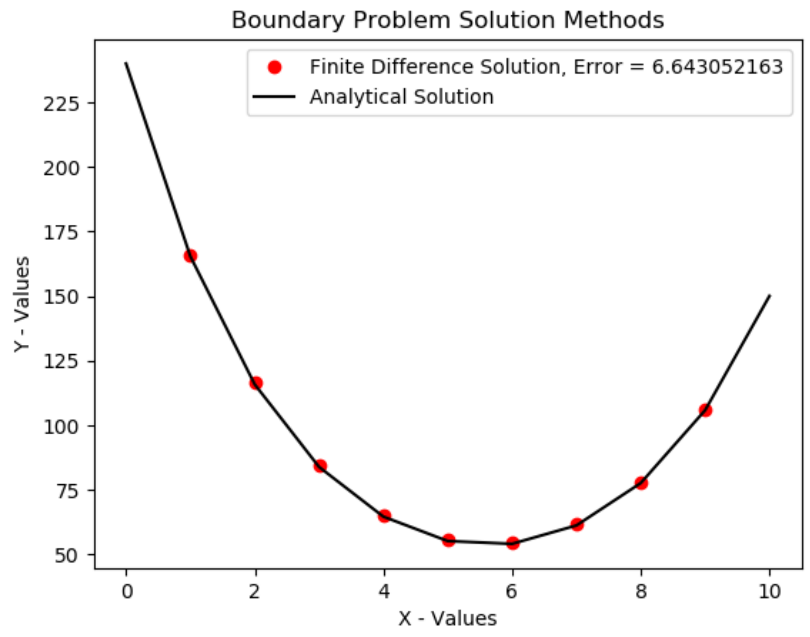

Figure 2: Finite difference temperature solution. the total error is much greater than the ODE solution for this step size.

Initially, it would seem that the ODE solver shooting method solution is always the best solution to use. Figure 3 shows how the finite difference approximation is much greater than the shooting method for this problem. Even with many more calculations, the shooting method has much less error overall in predicting the analytical values.

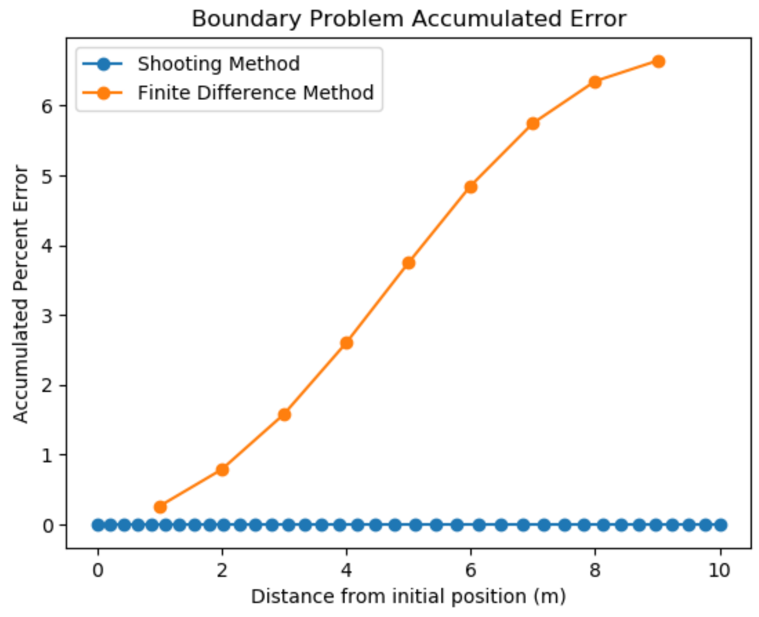

Figure 3: Error was much greater for the finite difference method, but the accumulation of error begins to flatten out at the end points.

For greater beam lengths or numbers of segments, the finite difference method can be improved, while the shooting method continues to grow in error exponentially. Figure 4 shows the rapid accumulation of error that could become significant for larger spans of beam.

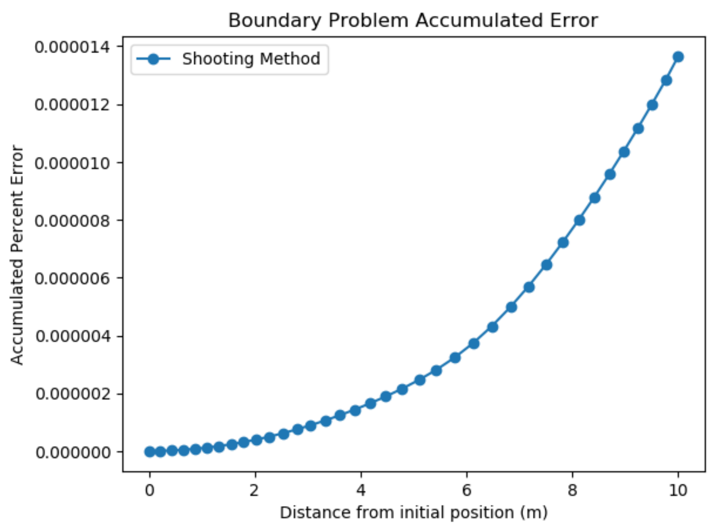

Figure 4: Though the error is very small, the ODE solver method has error that increases exponentially as the distance increases from the initial position.

Supporting Documents:

```
from numpy import exp
import numpy as np
import matplotlib.pyplot as plt
from NumericalMethods import ode45py, thomas
import math as math

def me306_final_1():
    # d2T/dx2 - 0.15T = 0
    side1 = 240 # Temperature of left side of rod (known)
    side2 = 150 # Temperature of right side of rod (known)
    iguess1 = -85# Initial guess for the rate of cooling along the rod
    iguess2 = -95 # Second guess for the rate fo cooling along the rod

    # Generate the analytical solution
    '''
    Ordinarily, I would put this simple 2 equation 2 unknown problem into a solver,
but since I had to do it out by hand,
    I will just generate the numbers from the operations
    '''
    epow = 10 * (0.15)**(0.5)
    lside = side2 - side1*exp(epow)
    c_2 = lside / (exp(-epow) - exp(epow))
    c_1 = side1 - c_2
    # anonymous function to generate output points for the bar
    anal = lambda in_val : (c_1 * exp(in_val*(0.15**0.5)) + c_2 * exp(-
in_val*(0.15**0.5)))
    dif_er = lambda x, y : abs((y-anal(x))/anal(x))*100
    # Set up the state function for the ODE solver
    def func(x, T):
        parray = np.zeros(2)
        parray[0] = T[1]
        parray[1] = 0.15*T[0]
        return parray
    # Generate analytical solution points
    X1 = np.arange(0, 11)
    Y1 = np.array([anal(i) for i in range(len(X1))])
    # Set up integration x segments
    x_startstop = [0,10]
    # The value we are shooting for
    needed_val = side2
    # Generate initial conditions for each of the guesses
    guess1 = np.array([side1, iguess1])
    guess2 = np.array([side1, iguess2])
    # set an initial step size. ode45py will adjust this, if we need to use constant
step size, we could use runge_kutta4 in the NumericalMethods module
    h = 1
    # Call the ode solver for each initial condition
    Xa, Ya = ode45py(func, x_startstop, guess1, st_sz=h)
    Xb, Yb = ode45py(func, x_startstop, guess2 , st_sz=h)
    # Store the output values from each integration estimate
    output1 = Ya[:,0][-1]
    output2 = Yb[:,0][-1]
    print('Output 1 : {:1.2f}, Output 2 : {:1.2f}'.format(output1, output2))

    # check to make sure that the output values will work for an interpolation on our
desired output codition
    if needed_val < output1:
        if output2 < needed_val:
            bound = True
    else:
        if output2 > needed_val:
            bound = True
        else:
            print('Initial guess end results fall on same side of known value of {} C.
Select different guesses for initial temp rates.'.format(needed_val))
            return
    if bound:
        # Find the shooting method results
        lininterp = lambda p_in, p1, p2, q1, q2: (q1 * (p2 - p_in) + q2 * (p_in - p1))
/ (p2 - p1)
        # Find shooting method interpolation as input for next ode integration
        shoot_dx0 = [240,lininterp(needed_val, output1, output2, guess1[-1], guess2[-
1])]
        # Call the ode solver one more time to generate a solution
        X2, Y2 = ode45py(func, x_startstop, shoot_dx0 , st_sz=h)
        shoot_res = Y2[:,0][-1]
        sh_error = [dif_er(X2[i], Y2[i,0]) for i in range(len(X2))]
        # plt.plot(Xa, Ya[:,0], ':', label=r'Guess 1, $\frac{dT}{dx}$ = ' +
'{}'.format(guess1[-1]) + r'$\frac{C}{m}$',c='C2')
        # plt.plot(Xb, Yb[:,0], ':', label=r'Guess 2, $\frac{dT}{dx}$ = ' +
'{}'.format(guess2[-1]) + r'$\frac{C}{m}$',c='C3')
        # plt.plot(X2, Y2[:,0], 'o-', label=r'Shooting Result, $\frac{dT}{dx}$ = ' +
'{:1.2f} , Error = {:.9f}'.format(shoot_dx0[-1], sum(sh_error)),c='C1')

    '''
    Using the centered finite difference formula for numeric differentiation, we can
represent the rod as a 9 x 9 tridiagonal system
    This can be soved using the thomas algorithm
    '''
    # Matrix length
    mat_size = 9
    # Primary diagonal value
    prim_diag = 2.15
    # Top and bottom vectors
    top_bot = 1
    # The primary diagonal
    in_f = [prim_diag] * mat_size
    # Bottom coefficients
    in_e = [top_bot] * (mat_size-1)
    # Top coefficients
    in_g = in_e
    # known vector
    in_b = np.zeros(mat_size)
    # Set the boundary conditions
    in_b[0] = side1
    in_b[-1] = side2
    # Finite difference results using thomas algorithm)
    fin_dif = thomas(in_f, in_e, in_g, in_b)
    # Generate x-points for plotting and error calculations
    fin_x = range(1, len(fin_dif)+1)
    # The algorithm produced positive and negative values
    fin_y = [abs(i) for i in fin_dif]
    # Calculate the error from the analytical solution at each output point
    er_fin = [dif_er(fin_x[i], fin_y[i]) for i in range(len(fin_x))]
    # Create a list of the accumulated error as the calculation progresses
    accumulate = lambda in_list : [in_list[i] + sum(in_list[0:i]) for i in
range(len(in_list))]
    # Store these lists for error analysis
    acc_sh_er = accumulate(sh_error)
    acc_fi_er = accumulate(er_fin)
    # plt.plot(fin_x, fin_y,'o', label='Finite Difference Solution, Error =
{:.9f}'.format(sum(er_fin)), c='r')
    # plt.plot(X1, Y1, '-', label='Analytical Solution', c='k')
    # plt.xlabel('X - Values')
    # plt.ylabel('Y - Values')
    # plt.title('Boundary Problem Solution Methods')
    # plt.legend()
    # plt.savefig('ME399_prob_1b.png',bbox_inches='tight')
    # plt.show()

    plt.plot(X2, acc_sh_er,'o-', label='Shooting Method')
    # plt.plot(fin_x, acc_fi_er,'o-', label='Finite Difference Method')
    plt.xlabel('Distance from initial position (m)')
    plt.ylabel('Accumulated Percent Error')
    plt.title('Boundary Problem Accumulated Error')
    plt.legend()
    plt.savefig('ME399_prob_1cb.png',bbox_inches='tight')
    plt.show()

me306_final_1()
def thomas(f_diag, e_diag, g_diag, b_vec):
    '''
    Numerical Methods : Thomas Algorithm

    A computationally lightweight method for solving tridiagonal matrices

    [ f(0) g(0)                          ][ x(0) ]   [ b(0) ]
    [ e(1) f(1) g(1)                     ][ x(1) ]   [ b(1) ]
    [      e(2) f(2) g(2)                ][ x(2) ]   [ b(2) ]
    [            ...   ...   ...         ][ ...  ] = [ ...  ]
    [                e(n-1) f(n-1) g(n-1) ][x(n-1)]   [b(n-1)]
    [                        e(n)   f(n)  ][ x(n) ]   [ b(n) ]

    f_diag = Primary diagonal
    e_diag = Bottom coefficients - begins with a zero
    g_diag = Top coefficients - ends with a zero
    b_vec = Known vector quantities

    Returns a list of solved values
    '''
    # Get the number of values to find
    n = len(b_vec)
    # Append zero to beginning of e matrix
    e_diag = [0] + e_diag
    # Append zero to end of g matrix
    g_diag.append(0)
    # Initiate a solution vector full of zeros
    sol_vec = [0 for i in range(n)]
    # Decomposition
    for k in range(1,n):
        e_diag[k] /= f_diag[k-1]
        f_diag[k] -= e_diag[k] * g_diag[k-1]
    # Forward substitution
    for k in range(1,n):
        b_vec[k] -= e_diag[k] * b_vec[k-1]
    # Back substitution
    sol_vec[-1] = b_vec[-1]/f_diag[-1]
    for k in range(n-2,-1,-1):
        sol_vec[k] = (b_vec[k] - g_diag[k] * sol_vec[k+1]) / f_diag[k]
    return sol_vec
def ode45py(func, x, y, st_sz=1.0e-4, tol=1.0e-6, iter_lim=50000):
    '''
    Numerical Methods: Differential Equations, Initial Value Problems

    4th-order / 5th-order Runge-Kutta Method
    Includes adaptive step size adjustment
    Imitates MATLAB ode45 functionality and output
    '''
    # Dormand-Prince coefficients for RK algorithm -
    a1 = 0.2; a2 = 0.3; a3 = 0.8; a4 = 8/9; a5 = 1.0; a6 = 1.0
    c0 = 35/384; c2 = 500/1113; c3 = 125/192; c4 = -2187/6784; c5=11/84
    d0 = 5179/57600; d2 = 7571/16695; d3 = 393/640; d4 = -92097/339200; d5 = 187/2100;
d6 = 1/40
    b10 = 0.2
    b20 = 0.075; b21 = 0.225
    b30 = 44/45; b31 = -56/15; b32 = 32/9
    b40 = 19372/6561; b41 = -25360/2187; b42 = 64448/6561; b43 = -212/729
    b50 = 9017/3168; b51 = -355/33; b52 = 46732/5247; b53 = 49/176; b54 = -5103/18656
    b60 = 35/384; b62 = 500/1113; b63 = 125/192; b64 = -2187/6784; b65 = 11/84
    # Store initial values
    x_f = x[-1]
    x_n = x[0]
    # y_n = y
    # Initialize variables
    X = []
    Y = []
    # Add the first set of known conditions
    X.append(x_n)
    Y.append(y)
    # I need an iteration counter that has a scope outside the computation loop
    i_count = 0
    # Set up to break the for loop at the end
    stopper = 0 # Integration stopper, 0 = off, 1 = on
    # Initialize a k0 to start with the step size
    k0 = st_sz * func(x_n, y)
    # Generate the RK coefficients
    for i in range(iter_lim):
        # Store the iteration number in the other variable for feedback string
        i_count = i
        # Compute the 4th order / 5th order algorithm
        k1 = st_sz * func(x_n + a1*st_sz, y + b10*k0)
        k2 = st_sz * func(x_n + a2*st_sz, y + b20*k0 + b21*k1)
        k3 = st_sz * func(x_n + a3*st_sz, y + b30*k0 + b31*k1 + b32*k2)
        k4 = st_sz * func(x_n + a4*st_sz, y + b40*k0 + b41*k1 + b42*k2 + b43*k3)
        k5 = st_sz * func(x_n + a5*st_sz, y + b50*k0 + b51*k1 + b52*k2 + b53*k3 +
b54*k4)
        k6 = st_sz * func(x_n + a6*st_sz, y + b60*k0 + b62*k2 + b63*k3 + b64*k4 +
b65*k5)
        # Getting to the slope is the whole point of this mess
        dy = c0*k0 + c2*k2 + c3*k3 + c4*k4 + c5*k5
        # Determine the estimated change in slope by comparing the output coefficients
for each RK coefficient
        E = (c0 - d0)*k0 + (c2 - d2)*k2 + (c3 - d3)*k3 + (c4 - d4)*k4 + (c5 - d5)*k5 -
d6*k6
        # Find the estimated error using a sum of squares method
        e = math.sqrt(np.sum(E**2)/len(y))
        # Compute a new step size to go into the weighting algorithm
        hNext = 0.9*st_sz*(tol/e)**0.2

        # If approximated error is within tolerance, accept this integration step and
move on
        if e <= tol:
            # Store the new result
            i = i-1
            y = y + dy
            # Increment the x-value by the new step size
            x_n = x_n + st_sz
            # Add the new values into the output vector
            X.append(x_n)
            Y.append(y)
            # Check to break the loop when we have reached the desired x-value
            if stopper == 1: break # Reached end of x-range
            # Set limits on how much the next step size can increase to avoid missing
data points
            if abs(hNext) > 10*abs(st_sz): hNext = 10*st_sz
            # Determine if the algorithm has reached the end of the dataset
            if (st_sz > 0.0) == ((x_n + hNext) >= x_f):
                hNext = x_f - x_n
                # Sets the break condition for the next loop iteration
                stopper = 1


            # Setting k0 to k6 * (next step size) / (current step size) forces the
algorithm to use the 4th order formula for the next step
            k0 = k6*hNext/st_sz
        else:
            # The error estimate is outside the required threshold to move on, we need
to redo the calculation with a smaller step size
            if abs(hNext) < abs(st_sz)*0.1 : hNext = st_sz*0.1
            # Set up k0 to go through the 5th order RK method on the next iteration
because the error was no good.
            k0 = k0*hNext/st_sz
        # Set the next iteration step size
        st_sz = hNext
    pcnt = (i_count/iter_lim)*100
    psolv = (x_n/x_f)*100
    print('ode45py _ Computation limit used : {:1.2f}%\n\tX-Domain Integrated:
{:1.2f}%'.format(pcnt, psolv))
    # Returns the arrays for x and y values
    return np.array(X), np.array(Y)
```

2. Fit the function $y = f(x) = axe^{bx}$ to the data and compute the standard deviation

x	0.5	1.0	1.5	2.0	2.5
y	0.541	0.398	0.232	0.106	0.052

Methods:

For each output (y) value let:

$$z = \ln\left(\frac{y}{x}\right)$$

$$\hat{x} = \frac{\sum y_i^2 x_i}{\sum y_i^2} = \frac{0.4147}{0.5188} = 0.7993$$

$$\hat{z} = \frac{\sum y_i^2 z_i}{\sum y_i^2} = \frac{-0.2668}{0.5188} = -0.5143$$

$$b = \frac{\sum y_i^2 z_i\,(x_i - \hat{x})}{\sum y_i^2 x_i (x_i - \hat{x})} = \frac{-0.1640}{0.0830} = -1.9759$$

$$\ln a = \hat{z} - b\hat{x} = -0.5143 + 1.9759(0.7993) = 1.0650$$

$$a = e^{1.0650} = 2.91$$

$$\therefore \boldsymbol{y = 2.91xe^{-1.9759x}}$$

Sum of squared residuals:

$$S = \sum [y_i - f(x_i)]^2 = 10.333 * 10^{-5}$$

Which produces a standard deviation, σ = 0.005869

Results:

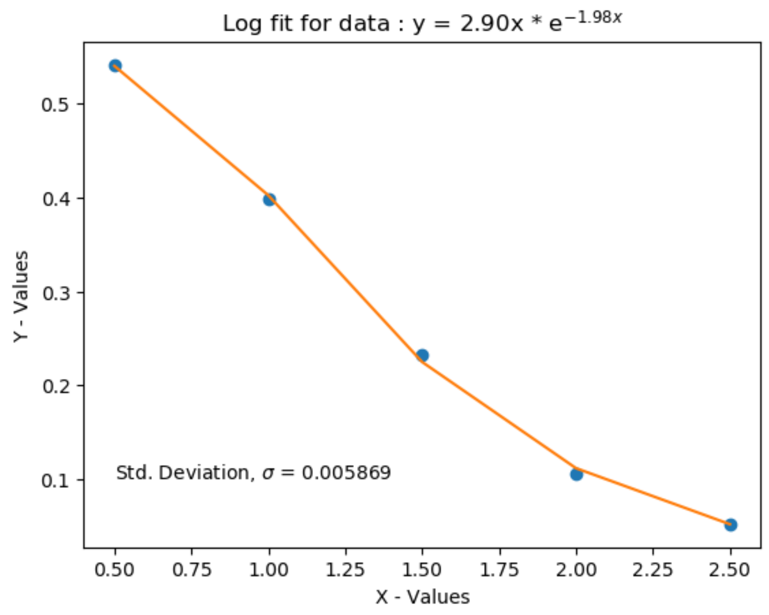

Figure 5: The small standard deviation using the xlog fit data means that the trendline is a good fit for these points.

The small standard deviation shown on figure 5 means that the generated formula is a good predictor for this data along this domain. This only applies when we have a reason to believe that the data will follow this type of an equation.

Supporting Documents

```
from NumericalMethods import xexp_reg
import numpy as np
import matplotlib.pyplot as plt
import math as math

def final_2():
    # data set is given
    x_data = [0.5*(i+1) for i in range(5)]
    y_data = [0.541, 0.398, 0.232, 0.106, 0.052]
    # Run exp_reg and produce the requested data
    a_con, b_con , sd = xexp_reg(x_data, y_data)
    # Generate a list of points from the regression function
    y_reg = [i*a_con*np.exp(b_con*i) for i in x_data]
    # Create the plot
    plt.plot(x_data, y_data,'o')
    plt.plot(x_data, y_reg)
    plt.text(.5, .1, r'Std. Deviation, $\sigma$ = {:1.6f}'.format(sd))
    plt.xlabel('X - Values')
    plt.ylabel('Y - Values')
    plt.title('Log fit for data : y = {:1.2f}x * '.format(a_con) + r'e$^{' +
'{:1.2f}x'.format(b_con) + r'}$')
    plt.savefig('ME306_final_prob2.png', bbox_inches='tight')
    plt.show()

final_2()
```

```
def xexp_reg(x_data, y_data):
    '''
    Numerical Methods : Exponential curve fit
    Input x and y data points,

    Returns exponential curve fit constants A and b
    y = Ax * exp(b*x)
    '''
    # Get the size of the dataset
    size = len(x_data)
    # To get the desired regression equation result, we take the natural log of
y_i/x_i
    ln_y = [math.log(y_data[i]/x_data[i]) for i in range(size)]

    # X_hat and Z_hat comes from using weighted averages for the data to generate a
better fit
    x_hat_num = sum([(y_data[i]**2)*x_data[i] for i in range(size)])
    z_hat_num = sum([(y_data[i]**2)*ln_y[i] for i in range(size)])
    # Sum of y_data square
    sy_sq = sum([y_data[i]**2 for i in range(size)])
    # New coefficients for linear fit
    x_hat = x_hat_num / sy_sq
    z_hat = z_hat_num / sy_sq
    # Generate the b-parameter
    coef_b_num = sum([(y_data[i]**2) * ln_y[i] * (x_data[i] - x_hat) for i in
range(size)])
    coef_b_den = sum([(y_data[i]**2) * x_data[i] * (x_data[i] - x_hat) for i in
range(size)])
    coef_b = coef_b_num / coef_b_den
    # Find the A coefficient
    coef_A = np.exp(z_hat - coef_b * x_hat)
    # Set up the function to calculate residuals and give feedback on how the curve
fits the data
    func = lambda x : x * coef_A * np.exp(coef_b * x)
    # Sum of the residuals from the calculation
    s_resi = sum([(y_data[i] - func(x_data[i]))**2 for i in range(size)])
    # Standard deviaton from the data. Small numbers are good here
    std_dev = (s_resi / (size - 2))**(0.5)
    # print('Exponential Curve Fit Standard Deviation : {}'.format(std_dev))
    return coef_A, coef_b, std_dev
```

3. The equations of motion for a double pendulum are given by:

$$(m_1+m_2)L_1\frac{d^2\theta_1}{dt^2}+\ m_2L_2\frac{d^2\theta_2}{dt^2}\cos(\theta_1-\theta_2)+m_2L_2\left(\frac{d\theta_2}{dt^2}\right)^2\sin(\theta_1-\theta_2)+g(m_1+m_2)\sin\theta_1=0$$

$$m_2L_2\frac{d^2\theta_1}{dt^2}+m_2L_1\frac{d^2\theta_1}{dt^2}\cos(\theta_1-\theta_2)-\ m_2L_1\left(\frac{d\theta_1}{dt^2}\right)^2\sin(\theta_1-\theta_2)+m_2g\sin\theta_2=0$$

Solve the equations of motion as a system of first order ODEs, plot angular displacement vs time, and plot the trajectories of the two masses in cartesian coordinates.

Methods:

First the equations need simplification. Making the following substitutions, the system is broken down into a system of second order equations. Making substitutions for theta 1 and theta 2, we get the solution for both second order equations in terms of the constants, initial angles, and initial velocities. Coding these equations into the double_pend function results in the position and velocity at each time increment between 0 and 100 seconds.

Let:

$$a=\ (m_1+m_2)L_1$$

$$b=\ m_2L_2\cos(\theta_1-\theta_2)$$

$$c=\ m_2L_1\cos(\theta_1-\theta_2)$$

$$d=\ m_2L_2$$

$$e=\ -m_2L_2\left(\frac{d\theta_2}{dt^2}\right)^2\sin(\theta_1-\theta_2)-g(m_1+m_2)\sin\theta_1$$

$$f=\ m_2L_1\left(\frac{d\theta_1}{dt^2}\right)^2\sin(\theta_1-\theta_2)-m_2g\sin\theta_2$$

$$\boldsymbol{\ddot{\theta}_1=x} \qquad \boldsymbol{\ddot{\theta}_2=y}$$

Then:

$$ax+by=e$$

$$y=\frac{e-ax}{b}$$

$$cx+dy=f$$

$$cx=f-dy=f-d\left(\frac{e-ax}{b}\right)=\frac{fb-de+dax}{b}$$

$$fb-de=bcx-dax=x(bc-da)$$

$$\boldsymbol{\ddot{\theta}_1=x=\frac{fb-de}{bc-da}} \qquad (\mathbf{1})$$

$$by = e - ax = e - a\left(\frac{fb - de}{bc - da}\right) = \frac{ebc - \cancel{eda} - afb + \cancel{eda}}{bc - da} = \frac{b(ec - af)}{bc - da}$$

$$\boldsymbol{\ddot{\theta}_2 = y = \frac{ec - af}{bc - da}} \qquad \mathbf{(2)}$$

Results

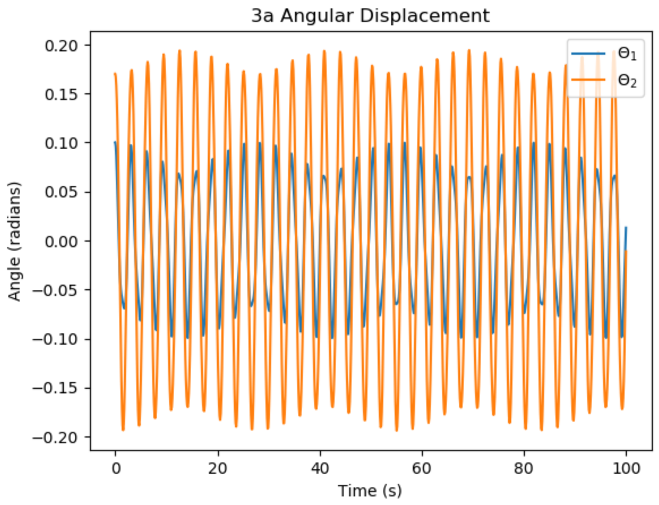

Figure 6: Angular displacement vs time for the first set of initial conditions.

The first set of initial conditions has each rod at a small angle away from the down (equilibrium) position. This system does not have very much energy, so it oscillates back and forth in a predictable sinusoidal pattern. Figure 6 shows angular displacement across the time interval. Figure 7 shows how the masses just draw smooth arcs across the Cartesian coordinate system. The dependent second order system was set up as outlined in the calculations section and the equations sent through the adaptive step RK 4/5 algorithm outlined from problem 1. The resulting Y vector had angular displacement and velocity information.

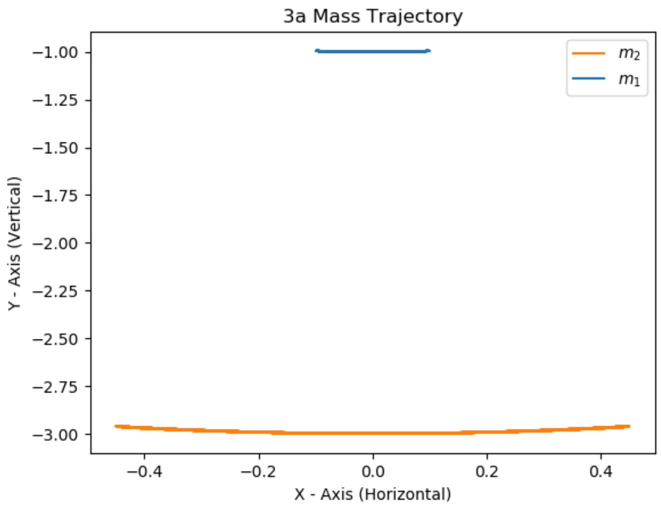

Figure 7: Position of swinging pendulum from initial conditions in part a. The masses swing back and forth across the same arc.

Figure 8 shows the angular displacement with the initial conditions provided in part b. The bars start out vertical and balanced where the graph is flat for the first 10 seconds, but then the two masses fall and undergo chaotic behavior. These specific results are extremely sensitive to initial conditions. Changing the integrator, or modifying any weighting coefficient in the RK algorithm would change the shape of the graph. Figure 9 shows how the short arm attached to m1 restricts the movement of that mass to the circle of length L1 at the center. The mass m2 is free to swing around m1 as its pivot point so it follows a chaotic path.

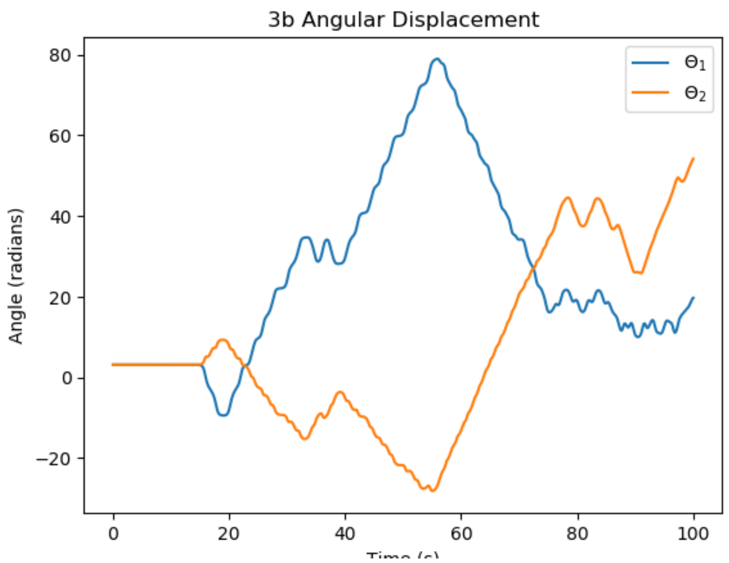

Figure 8: The angular displacement of each rod increases as it completes a revolution. This integral shows the shorter bar spinning almost 80 radians from the equilibrium position.

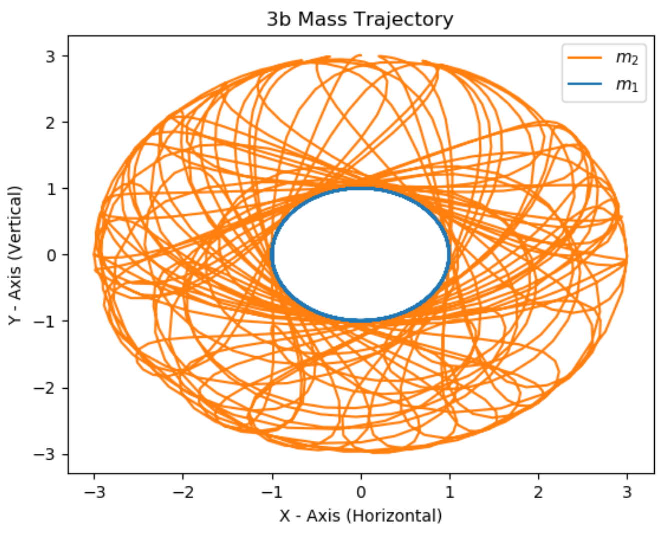

Figure 9: Starting the masses in the vertical position results in chaotic behavior.

Supporting Documents

```
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
from NumericalMethods import ode45py

def me306_final_3():
    G = 9.8  # gravity m/s^2
    L1 = 1  # length pendulum 1 m
    L2 = 2  # length pendulum 2 m
    M1 = 2  # mass pendulum 1 kg
    M2 = 1  # mass pendulum 2 kg
    # t1_0 = 0.1 # Initial value for theta 1 displacement
    # w1_0 = 0 # Initial value for angular velocity
    # t2_0 = 0.17 # Initial value for theta 2 displacement
    # w2_0 = 0 # Initial value for angular velocity
    # fig_name = '3a'
    t1_0 = np.pi # Initial value for theta 1 displacement
    w1_0 = 0 # Initial value for angular velocity
    t2_0 = np.pi # Initial value for theta 2 displacement
    w2_0 = 0 # Initial value for angular velocity
    fig_name = '3b'
    # Create an initial value array from the inputs above
    i_theta = np.array([t1_0, w1_0, t2_0, w2_0])
    # Function to hold the state variables. In MATLAB this is done in a separate m-
file.
    def double_pend(t, theta):
        # Initialize the return variable
        dydx = np.zeros(4)
        # Place the velocities in to the state vector
        dydx[0] = theta[1]
        dydx[2] = theta[3]
        # Delta theta is equal to theta 2 minus theta 1
        d_0 = theta[2] - theta[0]
        # Letter substitutions for handling the coefficients to the derivatives
        sA = (M1+M2)*L1
        sB = M2*L2*cos(d_0)
        sC = M2*L1*cos(d_0)
        sD = M2*L2
        sE = -M2*L2*(theta[3]**2)*sin(d_0) - G*(M1+M2)*sin(theta[0])
        sF = M2*L1*(theta[1]**2)*sin(d_0) - G*M2*sin(theta[2])
        # Set your second derivatives into the state vector
        dydx[1] = (sF*sB - sD*sE) / (sB*sC - sD*sA)
        dydx[3] = (sE*sC - sA*sF) / (sB*sC - sD*sA)
        return dydx
    # Create a time array to hold the start and end times. The ODE routine will handle
the time steps as it needs
```

```
    d_time = np.array([0,100])
    # Call my home brew ODE45 routine much like you would in MATLAB
    X, Y = ode45py(double_pend, d_time, i_theta)
    # Collect the position of each of the two masses over the time interval from the
angular displacement information
    # X-Displacement is the length of the bar * sin of the displacement angle
    x1 = L1*sin(Y[:,0])
    # Use negative here because the anchor point is above the position of the masses
at the equilibrium position
    y1 = -L1*cos(Y[:,0])
    # Mass 2 is stuck at the end of arm L1, so we need to add those coordinates onto
the end of the position output
    x2 = L2*sin(Y[:,2]) + x1
    y2 = -L2*cos(Y[:,2]) + y1
    # Plot the angular displacements
    plt.plot(X,Y[:,0],label=r'$\Theta_1$')
    plt.plot(X,Y[:,2],label=r'$\Theta_2$')
    plt.xlabel('Time (s)')
    plt.ylabel('Angle (radians)')
    plt.title(fig_name + ' Angular Displacement')
    plt.legend()
    plt.savefig('ME306_prob_{}_disp.png'.format(fig_name),bbox_inches='tight')
    plt.show()

    # Plot the trajectory for each mass
    plt.plot(x2,y2,label=r'$m_2$',c='C1')
    plt.plot(x1,y1,label=r'$m_1$',c='C0')
    plt.xlabel('X - Axis (Horizontal)')
    plt.ylabel('Y - Axis (Vertical)')
    plt.title(fig_name + ' Mass Trajectory')
    plt.legend()
    plt.savefig('ME306_prob_{}_traj.png'.format(fig_name),bbox_inches='tight')
    plt.show()

me306_final_3()
```

ODE45PY PROVIDED AS PART OF THE SOLUTION TO PROBLEM 1