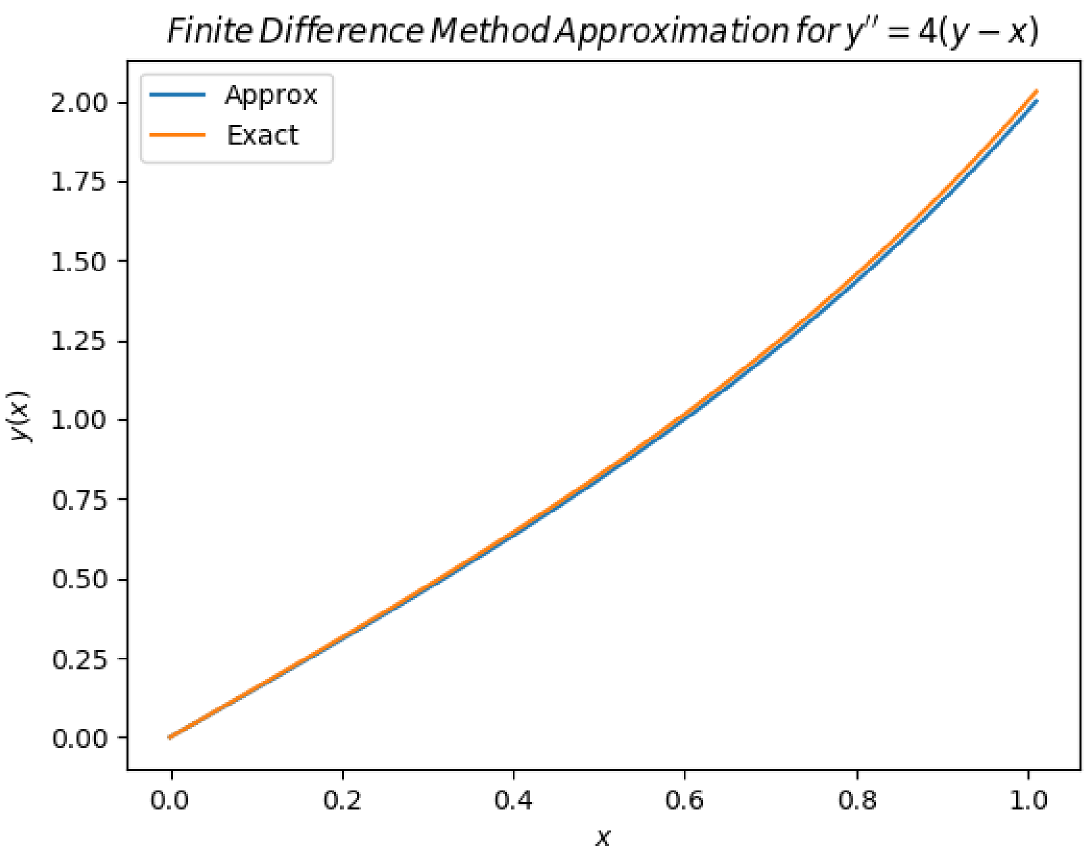
Andrew Durkiewicz
Homework 8
This homework did not require really any work by hand. I did all of it using python. Here is my work.

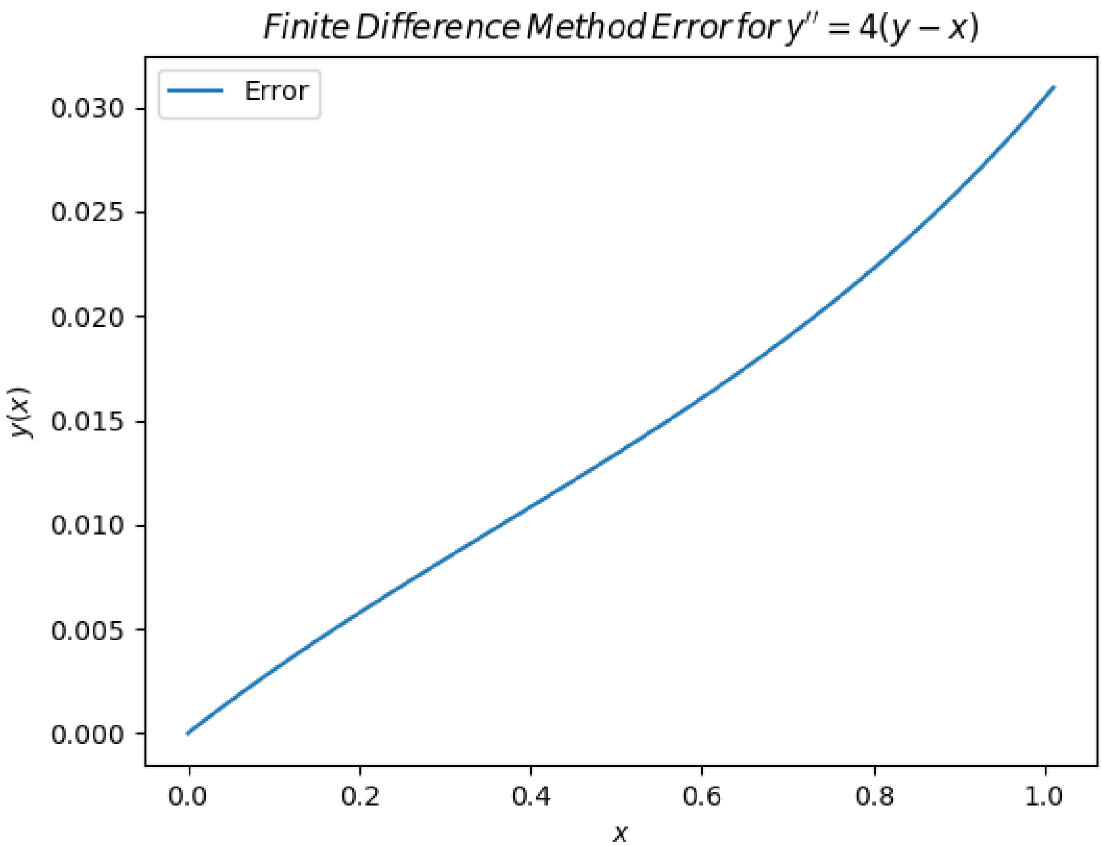
Results (Graphs, code follows immediately)

Ex 1. Use the finite difference method to solve the following (linear) problem:

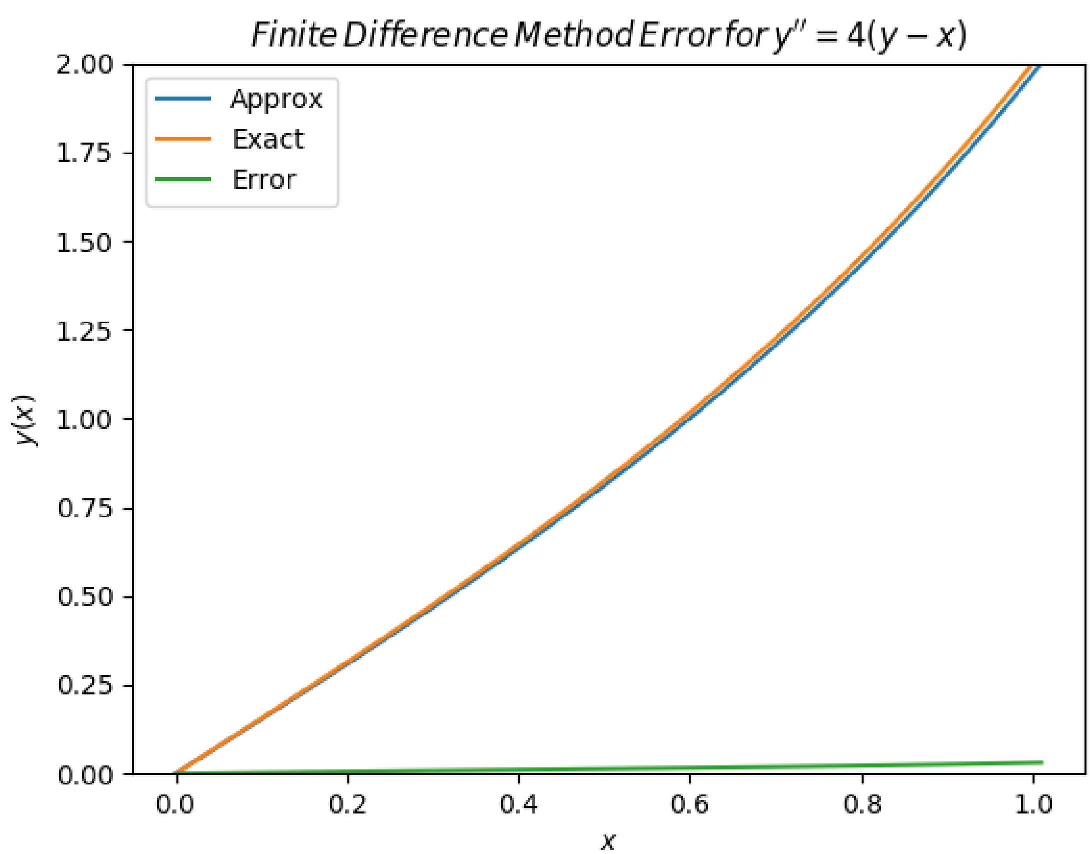
$$\begin{cases} y'' = 4(y-x) \\ y(0) = 0\ ,\ y(1) = 2. \end{cases}$$

Find the accuracy of the method.
Hint: the exact solution is $y(x) = \frac{e^2}{e^4-1}(e^{2x} - e^{-2x}) + x$.





Note that this graph is not on the same scale as the first graph. In reality, when put on the same scale, the error is rather a small impact:



Code:

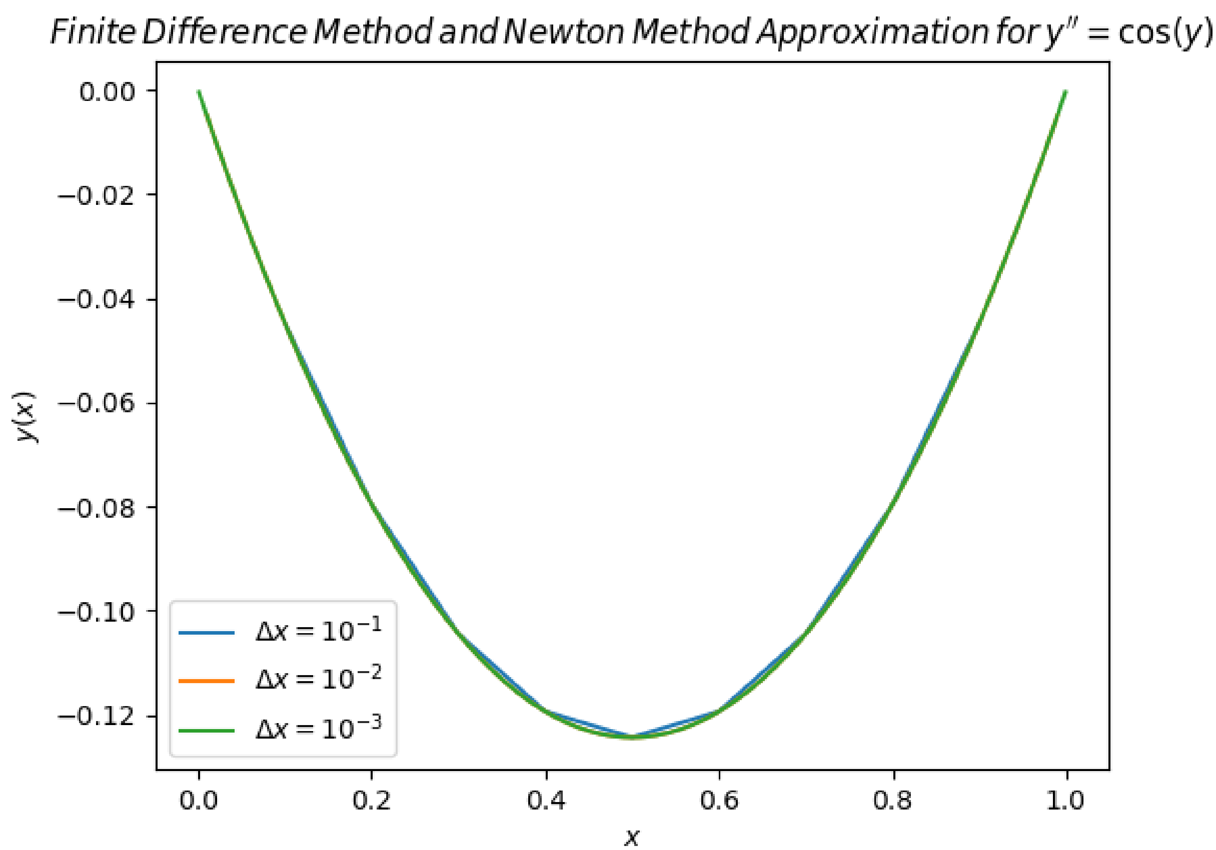
```
#!/usr/bin/env python3
#y'' = 4(y-x)
#y(0) = 0
#y(1) = 1
#exact solution y(x) = (e^2/(e^4 - 1))*(e^(2x) - e^(-2x)) + x

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

def p(x):
    return 0+0*x
def q(x):
    return 4 + 0*x
def r(x):
    return -4*x
def bvp(a,b,alpha,beta,N):
      dx = (b-a)/(N)
      x  = a+np.arange(a,b,(dx))
      A = -np.diag(1+dx/2*p(x[1:N]),-1)  + np.diag(2+dx**2*q(x)) - np.diag(1-
dx/2*p(x[0:(N-1)]),1)
      vecB = -dx**2*r(x)
      vecB[0] = vecB[0] + (1+dx/2*p(x[0]))*alpha
      vecB[N-1] = vecB[N-1] + (1-dx/2*p(x[N-1]))*beta
      Y = np.linalg.lstsq(A,vecB)
      y_sol = np.concatenate(([alpha], Y[0].reshape(-1), [beta]))
      x2  = a+dx*np.arange(0,N+2)
      return y_sol,x2
def real(x):
      return (np.exp(2)/(np.exp(4) - 1))*(np.exp(2*x) - np.exp(-2*x)) + x
def find_accuracy(y_sol,y_real):
      return np.abs(y_real - y_sol)
def plot1(x2,y_sol,y_real):
      plt.plot(x2,y_sol, label = 'Approx')
      plt.plot(x2,y_real, label = 'Exact')
      plt.legend()
      plt.title(r'$Finite \/Difference \/Method \/Approximation \/for
\/y^{\prime\prime} = 4(y-x)$')
      plt.xlabel(r'$x$')
      plt.ylabel(r'$y(x)$')
      plt.savefig("Problem_1_Approx_vs_Real.png")

def plot2(x2,error):
      c = plt.plot(x2,error, label = 'Error')
      plt.legend()
      plt.title(r'$Finite\/ Difference\/ Method\/ Error\/ for \/y^{\prime\prime} =
4(y-x)$')
      plt.xlabel(r'$x$')
      plt.ylabel(r'$y(x)$')
      plt.ylim(0,2)
      plt.savefig("Problem_1_Error_Autofit.png")
y_sol, x2 = bvp(0,1,0,2,100)
y_real = real(x2)
error = find_accuracy(y_sol,y_real)
#plot1(x2,y_sol,y_real)
plot2(x2,error)
```

Problem2

Ex 2. Use the finite difference method combined with the Newton method to solve:

$$\begin{cases} y'' = \cos y \\ y(0) = y(1) = 0. \end{cases}$$

[Extra] Find the accuracy of the scheme.
Hint: find the solutions for different $\Delta x = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$ *(5 iterations of the Newton's method is usually enough). Compare the solutions with the one having the higher accuracy.*



Code below

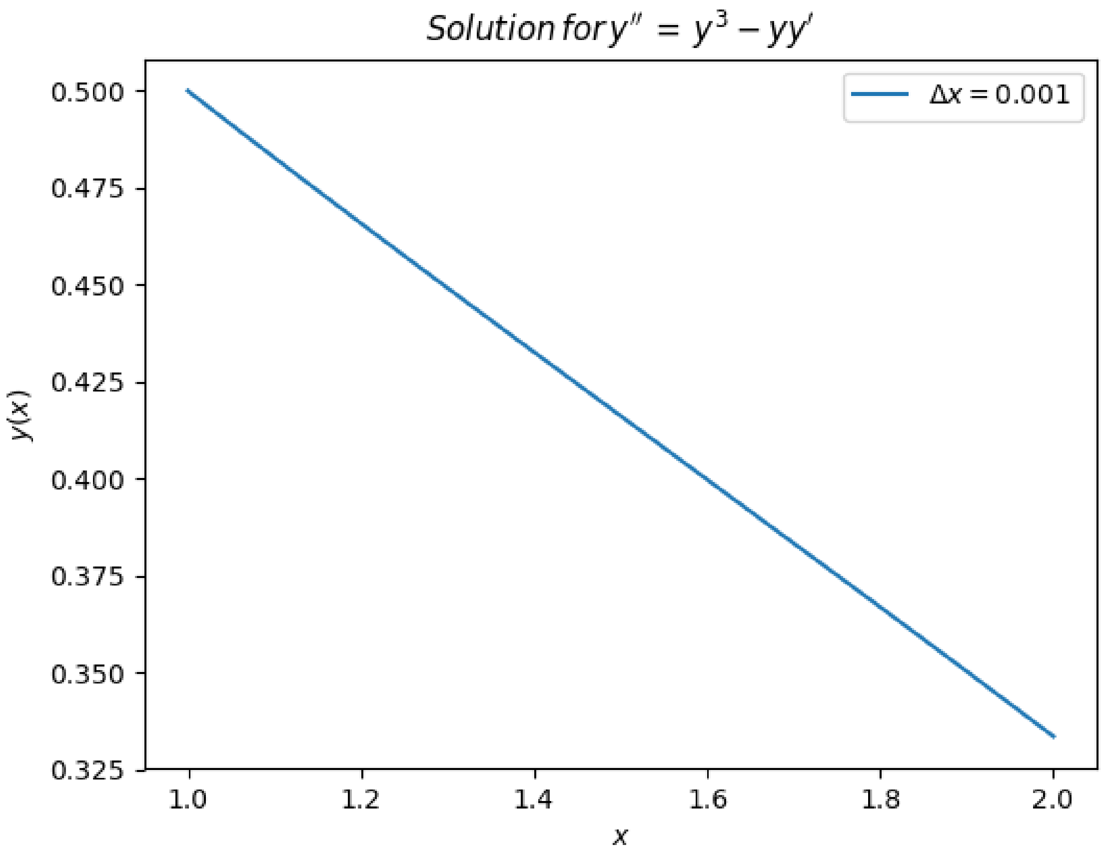
```
#!/usr/bin/env/ python3
import numpy as np
import matplotlib.pyplot as plt
def F(y):
    return np.cos(y)
def DF(y):
    return np.diag(-np.sin(y))
def calculate_solution(a,b,alpha,beta,N):
      N = N-1
      dx = (b-a)/(N+1)
      x  = a+dx*np.arange(1,N+1)
      iterations = 5
      yaxis = np.diag(np.ones(N))- np.diag(np.ones(N-1),1) - np.diag(np.ones(N-1),-
1)
      Y = 1+x
      L = 2*np.diag(np.ones(N)) - np.diag(np.ones(N-1),1) - np.diag(np.ones(N-1),-
1)
      vecBC = np.zeros(N)
      vecBC[0] = alpha
      vecBC[-1] = beta
      yaxis[0,:] = Y
      for k in range(iterations):
          DJ = L + dx**2*DF(Y)
          tp = np.linalg.lstsq(DJ,L.dot(Y) + dx**2*F(Y)- vecBC)
          Y  = Y - tp[0]
          yaxis[k+1,:] = Y
      return x,Y

a = 0
b = 1
alpha = 0
beta = 0
exponent = np.arange(1,4)
N = []
for i in exponent:
      N.append(10**(i))
ysolutions = [[]]*10
xsolutions = [[]]*10
tick = 0
while tick < len(N):
      x,y = calculate_solution(a,b,alpha,beta,N[tick])
      yvalues = np.array(y,dtype = object)
      xvalues = np.array(x,dtype = object)

      ysolutions[tick] = yvalues
      xsolutions[tick] = xvalues
      tick+=1
plotnumber = np.arange(0,len(N))

plt.plot(xsolutions[0],ysolutions[0],label = '$\Delta{x} = 10^{-%d}$' %exponent[0])
plt.plot(xsolutions[1],ysolutions[1],label = '$\Delta{x} = 10^{-%d}$' %exponent[1])
plt.plot(xsolutions[2],ysolutions[2],label = '$\Delta{x} = 10^{-%d}$' %exponent[2])

plt.title(r'$Finite \/Difference \/Method\/ and \/Newton
\/Method\/Approximation \/for \/y^{\prime\prime} = \cos(y)$')
plt.xlabel(r'$x$')
plt.ylabel(r'$y(x)$')
plt.legend()
plt.savefig("Problem_2_Solution_vs_dx.png")
```

Problem 3

Ex 3. Use the non-linear finite difference method to solve:

$$\begin{cases} y'' = y^3 - yy' \\ y(1) = \frac{1}{2}\ ,\ y(2) = \frac{1}{3}. \end{cases}$$

Compare the solution with the one given by the shooting method (accuracy? computation time?).




```
#!/usr/bin/python3
import numpy as np
from matplotlib import pyplot as plt

def derivative_matrix(N):
    M = np.zeros((N, N))
    for i in range(N):
        line = np.array([0] * (i-1) +
                        [-1] * (i >= 1) +
                        [2] +
                        [-1] * (i < N-1) +
                        [0] * (N - i - 2))
        M[i] = line
    return M
```

```
def derivative(Y, dx):
    dY = np.zeros(Y.shape)
    dY[0] = (Y[1] - Y[0]) / dx
    for i in range(1, len(Y)-1):
        dY[i] = (Y[i-1] - Y[i+1]) / (2*dx)
    dY[-1] = (Y[-1] - Y[-2]) / dx
    return dY

def F(Y, dx):
    return Y**3 - Y * derivative(Y, dx)

def DF(Y, dx):
    N = len(Y)
    DF = np.zeros((N, N))
    for i, y_i in enumerate(Y):
        if i > 0:
            DF[i, i-1] = y_i / (2*dx)

        if i == 0:
            DF[i, i] = 3 * y_i**2 + (2*y_i - Y[i+1]) / dx
        elif i == N-1:
            DF[i, i] = 3 * y_i**2 + (Y[i-1] - 2*y_i) / dx
        else:
            DF[i, i] = 3 * y_i**2 - (Y[i+1] - Y[i-1]) / (2*dx)

        if i < N-1:
            DF[i, i+1] = -1 * y_i / (2*dx)
    return DF
plt.figure()
for k, N in enumerate((1001,)):
    x0 = 1
    y0 = 1/2
    xb = 2
    yb = 1/3
    dx = (xb - x0) / (N-1)
    x_range = np.linspace(x0, xb, N)
    Y0 = np.zeros(N)
    Y0[0] = y0
    Y0[-1] = yb
    L = derivative_matrix(N)
    Y = np.ones(N)
    for i in range(5):
        J = L.dot(Y) + dx**2 * F(Y, dx) - Y0
        DJ = L + DF(Y, dx) * dx**2
        dY = np.linalg.solve(DJ, J)
        Y -= dY
plt.plot(x_range, Y, label='$\Delta x=%.3f$' % dx)
plt.title(r'$Solution\/ for\/ y^{\prime\prime} \/= \/y ^ {3}-yy^{\prime}$')
plt.xlabel(r'$x$')
plt.ylabel(r'$y(x)$')
plt.legend()
plt.savefig('Ex3-solution.png')
```