Jacob Dennis

Physics 411

14 March 2016

Prof. Gull

**Midterm 2 – Code and Results**

**Shooting with Friction**

*Code*

import numpy as np

import matplotlib.pyplot as plt

#Constants:

g = 9.81

def RDoublePrime(V, gamma):

vAbs = np.linalg.norm(V)

rX = -gamma \* vAbs \* V[0]

rZ = -g - gamma \* vAbs \* V[1]

return np.asarray([rX, rZ])

def VectorF(RV, t, gamma):

V = RV[1]

return np.asarray([V, RDoublePrime(V, gamma)])

def ForwardEuler(R0, magV0, alpha, f, a, b, N, gamma):

h = (b - a) / N

T = np.linspace(a, b, N)

RV = np.zeros((N + 1, 2, 2))

V0 = magV0 \* np.asarray([np.cos(alpha), np.sin(alpha)])

RV[0] = np.asarray([R0, V0])

for i in range(N):

RV[i + 1] = RV[i] + h \* f(RV[i], T[i], gamma)

return RV

def RungeKutta(R0, magV0, alpha, f, a, b, N, gamma):

h = (b - a) / N

T = np.linspace(a, b, N)

RV = np.zeros((N + 1, 2, 2))

V0 = magV0 \* np.asarray([np.cos(alpha), np.sin(alpha)])

RV[0] = np.asarray([R0, V0])

for i in range(N):

k1 = h \* f(RV[i], T[i], gamma)

k2 = h \* f(RV[i] + .5 \* k1, T[i] + 0.5 \* h, gamma)

k3 = h \* f(RV[i] + .5 \* k2, T[i] + 0.5 \* h, gamma)

k4 = h \* f(RV[i] + k3, T[i] + h, gamma)

RV[i + 1] = RV[i] + 1.0 / 6.0 \* (k1 + 2.0 \* k2 + 2.0 \* k3 + k4)

return RV

def GetXsandZs(RV):

X = []

Z = []

for rv in RV:

X.append(rv[0][0])

Z.append(rv[0][1])

return np.asarray([np.asarray(X), np.asarray(Z)])

def GetPositiveXsandZs(RV):

X = []

Z = []

for rv in RV:

if rv[0][1] < -0.5:

return np.asarray([np.asarray(X), np.asarray(Z)])

else:

X.append(rv[0][0])

Z.append(rv[0][1])

return np.asarray([np.asarray(X), np.asarray(Z)])

def heightDifference(alpha):

RungeKuttaRV = GetXsandZs(RungeKutta(r0, magV, alpha, VectorF, 0.0, 5.0, 1000, 0.2))

for i in range(1000):

if abs(RungeKuttaRV[0][i] - 6.0) <= 0.02:

return RungeKuttaRV[1][i] - 3.048

return (RungeKuttaRV[1][1000] - 3.048)

def RegulaFalsi(a, b, f, tol):

iterationDiff = abs(a - b)

iterationNum = 0

while iterationDiff > tol:

xNew = (f(b) \* a - f(a) \* b) / (f(b) - f(a))

if (np.sign(f(a)) == np.sign(f(xNew))):

iterationDiff = abs(a - xNew)

a = xNew

elif (np.sign(f(b)) == np.sign(f(xNew))):

iterationDiff = abs(b - xNew)

b = xNew

elif f(xNew) == 0.0:

return xNew

return xNew

allEulerResults = ForwardEuler(np.asarray([0.0, 3.0]), 5.0, np.pi / 4.0, VectorF, 0.0, 5.0, 100, 1.0)

allRKResults = RungeKutta(np.asarray([0.0, 3.0]), 4.0, 3.0 \* np.pi / 4.0, VectorF, 0.0, 5.0, 100, 1.0)

positionEuler = GetPositiveXsandZs(allEulerResults)

positionRK = GetPositiveXsandZs(allRKResults)

tRange = np.arange(0.0, 5.0, .05)

tMax = 0.0

maxHeight = 0.0

for i in range(len(positionRK[1])):

if positionRK[1][i] >= maxHeight:

maxHeight = positionRK[1][i]

tMax = tRange[i]

print 'The maximum height of', maxHeight,'meters occurs at', tMax, 'seconds after launch.'

#Our two roots lie on either side of pi / 4 - so conduct Regula Falsi searches on either side

alpha1 = RegulaFalsi(0.0, np.pi / 4.0, heightDifference, 0.01)

alpha2 = RegulaFalsi(np.pi / 4.0, np.pi / 2.0, heightDifference, 0.01)

print 'alpha1:', alpha1, 'radians'

print 'alpha2:', alpha2, 'radians'

alpha1trajectory = GetPositiveXsandZs(RungeKutta(np.asarray([0.0, 2.0]), 20.0, alpha1, VectorF, 0.0, 5.0, 100, 0.2))

alpha2trajectory = GetPositiveXsandZs(RungeKutta(np.asarray([0.0, 2.0]), 20.0, alpha2, VectorF, 0.0, 5.0, 100, 0.2))

plt.clf()

fig1, axes1 = plt.subplots(1, 1)

axes1.plot(positionEuler[0], positionEuler[1])

axes1.set\_ylim(bottom = 0.0)

plt.xlabel('X')

plt.ylabel('Z')

plt.title('Shooting with Friction - Forward Euler, gamma = 1')

plt.savefig('Midterm 2 - Friction - Euler.png')

plt.close(fig1)

plt.clf()

fig2, axes2 = plt.subplots(1, 1)

axes2.plot(positionRK[0], positionRK[1])

axes2.set\_ylim(bottom = 0.0)

plt.xlabel('X')

plt.ylabel('Z')

plt.title('Shooting with Friction - Runge Kutta, gamma = 1')

plt.savefig('Midterm 2 - Friction - Runge Kutta.png')

plt.close(fig2)

plt.clf()

fig3, axes3 = plt.subplots(1, 1)

axes3.plot(alpha1trajectory[0], alpha1trajectory[1], label = 'alpha1')

axes3.plot(alpha2trajectory[0], alpha2trajectory[1], label = 'alpha2')

plt.legend(loc = 0)

axes3.set\_ylim(bottom = 0.0)

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Shooting with Friction - Hitting a Basketball Hoop')

plt.savefig('Midterm 2 - Friction - basketball hoop.png')

plt.close(fig3)

*Results*

The maximum height of 3.25949428286 meters occurs at 0.2 seconds after launch.

alpha1: 0.395743886175 radians

alpha2: 1.02070997091 radians

**The Poisson Problem**

*Code*

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.cm as cm

def rho(x, y):

return np.sin(np.pi \* x) \* np.sin(2.0 \* np.pi \* y)

def poissonPhi(x, y):

return -4.0 \* np.pi \* rho(x, y)

def Relaxation(x0, y0, xf, yf, a, poisson, tol):

Nx = int((xf - x0) / a)

Ny = int((yf - y0) / a)

Phi = np.zeros((Nx + 1, Ny + 1))

PhiPrevious = np.zeros((Nx + 1, Ny + 1))

for i in range(Nx + 1):

Phi[0][i] = i \* a

PhiPrevious[0][i] = i \* a

Phi[-1][i] = i \* a

PhiPrevious[-1][i] = i \* a

for i in range(Ny + 1):

Phi[i][-1] = 1.0

PhiPrevious[i][-1] = 1.0

difference = 1.0

iterationNum = 0

while (difference > tol) & (iterationNum <= 10000):

for i in range(1, Ny):

for j in range(1, Nx):

Phi[i][j] = 0.25 \* (PhiPrevious[i + 1][j] + PhiPrevious[i - 1][j] + PhiPrevious[i][j + 1] + PhiPrevious[i][j - 1] - poisson(j \* a, i \* a))

difference = np.max(np.abs(Phi - PhiPrevious))

Phi, PhiPrevious = PhiPrevious, Phi

iterationNum += 1

print 'It took', iterationNum, 'iterations to get a solution.'

return Phi

x = np.linspace(0.0, 1.0, 100)

y = np.linspace(0.0, 1.0, 100)

X, Y = np.meshgrid(x, y)

PhiApprox = Relaxation(0.0, 0.0, 1.0, 1.0, 1.0/32, rho, 1.0e-5)

x2 = np.arange(0.0, 1.0 + 1.0/32, 1.0/32)

y2 = np.arange(0.0, 1.0 + 1.0/32, 1.0/32)

X2, Y2 = np.meshgrid(x2, y2)

plt.figure(1)

plt.clf()

ChargeDistribution = plt.contour(X, Y, rho(X, Y), 20)

plt.clabel(ChargeDistribution, inline = True, fontsize = 10)

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Initial Charge Distribution')

plt.savefig('Midterm 2 - Poisson - Charge Distribution.png')

plt.close(1)

plt.figure(2)

plt.clf()

contoursPhiApprox = plt.contour(X2, Y2, PhiApprox, 10)

plt.clabel(contoursPhiApprox, inline = True, fontsize = 10)

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Approximate Potentials')

plt.savefig('Midterm 2 - Poisson - Potential.png')

plt.close(2)

*Results*

It took 1238 iterations to get a solution.

**The Poisson Problem II**

*Code*

import numpy as np

import matplotlib.pyplot as plt

def rho(x, y):

return np.sin(np.pi \* x) \* np.sin(2.0 \* np.pi \* y)

def del2Phi(x, y):

return -4.0 \* np.pi \* rho(x, y)

def Relaxation(InitialGuess, dx, dy, poisson, tol):

Ny, Nx = np.shape(InitialGuess)

Phi = np.empty((Ny, Nx))

PhiPrevious = InitialGuess

diff = 1.0

k = 0

while (diff > tol) & (k < 10000): #Sanity check – if it takes over 10000 iterations, the guess is probably not a good one.

for i in range(1, Ny - 1):

for j in range(1, Nx - 1):

Phi[i][j] = 0.25 \* (PhiPrevious[i][j + 1] + PhiPrevious[i][j - 1] +

PhiPrevious[i + 1][j] + PhiPrevious[i - 1][j] - poisson(j \* dx, i \* dy))

diff = np.amax(np.abs(Phi - PhiPrevious))

Phi, PhiPrevious = PhiPrevious, Phi

k += 1

print 'It took', k, 'iterations to converge.'

return Phi

def Interpolation(A, iterationNum):

newDx = 2\*\*iterationNum + 1

newDy = 2\*\*iterationNum + 1

interpolatedA = np.empty((newDx, newDy))

for i in range(newDy):

#print 'Row', i

for j in range(newDx):

#print 'Column', j

if (i % 2 == 0):

if (j % 2 == 0):

interpolatedA[i][j] = A[i/2][j/2]

elif (j % 2 == 1):

interpolatedA[i][j] = 0.5 \* (A[i/2][(j - 1)/2] + A[i/2][(j + 1)/2])

elif (i % 2 == 1):

if (j % 2 == 0):

interpolatedA[i][j] = 0.5 \* (A[(i - 1)/2][j/2] + A[(i + 1)/2][j/2])

elif (j % 2 == 1):

interpolatedA[i][j] = 0.25 \* (A[(i - 1)/2][(j - 1)/2] + A[(i - 1)/2][(j + 1)/2] +

A[(i + 1)/2][(j - 1)/2] + A[(i + 1)/2][(j + 1)/2])

return interpolatedA

zerosGuess = np.zeros((3, 3))

zerosGuess[0] = np.asarray([0.0, 0.5, 1.0])

zerosGuess[1][2] = 1.0

zerosGuess[2] = np.asarray([0.0, 0.5, 1.0])

dx, dy = (0.5, 0.5)

iterationCounter = 1

phiGuess = Relaxation(zerosGuess, 0.5, 0.5, del2Phi, 1.0e-5)

dx, dy = (0.25, 0.25)

iterationCounter += 1

while (dx > 1/1024.0):

newPhiGuess = Interpolation(phiGuess, iterationCounter)

phiGuess = Relaxation(newPhiGuess, dx, dy, del2Phi, 1.0e-5)

dx /= 2.0

dy /= 2.0

print dx

iterationCounter += 1

print phiGuess

*Results*

Code took to long to converge, for unknown reasons – started to hit the 10000-iteration sanity limit at around dx = 2\*\*(-7).

**The Heat Equation**

*Code*

Part 1: FTCS Method

import numpy as np

import matplotlib.pyplot as plt

def T(x):

return 50.0 \* np.exp(-80.0 \* (x - 0.5)\*\*2)

def FTCS(TInitial, t0, tf, h, a, xRange):

timeRange = np.arange(t0, tf + h, h)

N = len(TInitial)

Tnext = np.empty(N)

Tprev = TInitial

Tnext[N - 1] = 50.0

Tprev[N - 1] = 50.0

Tnext[0] = 0.0

Tprev[0] = 0.0

plt.plot(xRange, Tprev, label = 't = 0s')

for t in timeRange:

for i in range(1, N - 1):

Tnext[i] = Tprev[i] + h / a\*\*2 \* (Tprev[i + 1] + Tprev[i - 1] - 2.0 \* Tprev[i])

Tnext, Tprev = Tprev, Tnext

if abs(t - 0.001) < (h / 1000):

plt.plot(xRange, Tprev, label = 't = .001s')

if abs(t - 0.01) < (h / 1000):

plt.plot(xRange, Tprev, label = 't = .01s')

if abs(t - 0.1) < (h / 1000):

plt.plot(xRange, Tprev, label = 't = .1s')

if abs(t - 1.0) < (h / 1000):

plt.plot(xRange, Tprev, label = 't = 1s')

return Tprev

xRange = np.arange(0.0, 1.001, 0.01)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(1)

plt.clf()

plt.plot(xRange, TInitial)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Initial Temperature Distribution')

plt.savefig('Midterm 2 - Problem 4 - Initial Temperature Distribution.png')

plt.close(1)

TInitial = T(xRange)

plt.figure(2)

plt.clf()

FTCS(TInitial, 0.0, 1.0, (.01)\*\*2 / 2, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - FTCS.png')

plt.close(2)

Parts 2 and 3: Instability of FTCS Method

import numpy as np

import matplotlib.pyplot as plt

xRange = np.arange(0.0, 1.001, 0.01)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(3)

plt.clf()

FTCS(TInitial, 0.0, 1.0, 1.0e-4, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time - h = 1e-4')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - FTCS instability 1.png')

plt.close(3)

TTInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(4)

plt.clf()

FTCS(TInitial, 0.0, 1.0, 1.0e-3, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time - h = .001')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - FTCS instability 2.png')

plt.close(4)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(5)

plt.clf()

FTCS(TInitial, 0.0, 1.0, 1.0e-2, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time - h = .01')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - FTCS instability 3.png')

plt.close(5)

Part 4: Crank Nicolson Method

import numpy as np

import matplotlib.pyplot as plt

def Transpose1D(A):

length = len(A)

Result = np.ones((length, 1))

for i in range(length):

Result[i] \*= A[i]

return Result

def T(x):

return 50.0 \* np.exp(-80.0 \* (x - 0.5)\*\*2)

def CrankNicolson(TVector, t0, tf, h, a, xRange):

tRange = np.arange(t0, tf + h, h)

a1 = 1.0 + h / a\*\*2

a2 = -h / (2.0 \* a\*\*2)

b1 = 1.0 - h / a\*\*2

b2 = h / (2.0 \* a\*\*2)

a1Vector = np.ones(len(TVector)) \* a1

b1Vector = np.ones(len(TVector)) \* b1

A = np.diag(a1Vector)

B = np.diag(b1Vector)

A[0][1] = a2

A[-1][-2] = a2

B[0][1] = b1

B[-1][-2] = b2

for i in range(1, len(TVector) - 1):

A[i][i - 1] = a2

A[i][i + 1] = a2

B[i][i - 1] = b2

B[i][i + 1] = b2

plt.plot(xRange, TVector, label = 't = 0s')

for t in tRange:

v = Transpose1D(np.dot(B, TVector))

Tnew = np.linalg.solve(A, v)

Tnew[-1][0] = 50.0

Tnew, TVector = TVector, Tnew

if abs(t - .001) < h / 1000:

plt.plot(xRange, np.transpose(TVector)[0], label = 't = .001s')

if abs(t - .01) < h / 1000:

plt.plot(xRange, np.transpose(TVector)[0], label = 't = .01s')

if abs(t - .1) < h / 1000:

plt.plot(xRange, np.transpose(TVector)[0], label = 't = .1s')

if abs(t - 1.0) < h / 1000:

plt.plot(xRange, np.transpose(TVector)[0], label = 't = 1s')

return np.transpose(TVector)[0]

xRange = np.arange(0.0, 1.01, .01)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(6)

plt.clf()

CrankNicolson(TInitial, 0.0, 1.0, (.01)\*\*2 / 2, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time\n Crank-Nicolson Method')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - CrankNicolson.png')

plt.close(6)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(7)

plt.clf()

CrankNicolson(TInitial, 0.0, 1.0, 1.0e-4, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time - h = 1e-4\n Crank-Nicolson Method')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - CrankNicolson instability 1.png')

plt.close(7)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(8)

plt.clf()

CrankNicolson(TInitial, 0.0, 1.0, 1.0e-3, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time - h = 1e-3\n Crank-Nicolson Method')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - CrankNicolson instability 2.png')

plt.close(8)

TInitial = T(xRange)

TInitial[-1] = 50.0

plt.figure(9)

plt.clf()

CrankNicolson(TInitial, 0.0, 1.0, 1.0e-2, .01, xRange)

plt.xlabel('X')

plt.ylabel('T')

plt.title('Temperature Distribution over Time - h = 1e-2\n Crank-Nicolson Method')

plt.legend(loc = 0)

plt.savefig('Midterm 2 - Problem 4 - CrankNicolson instability 3.png')

plt.close(9)

*Results*

Interestingly, the Crank Nicolson method also seemed to break down for too large of a stepsize.