Computing Tasks

May 13, 2018

Exercise 1

Integration Practice

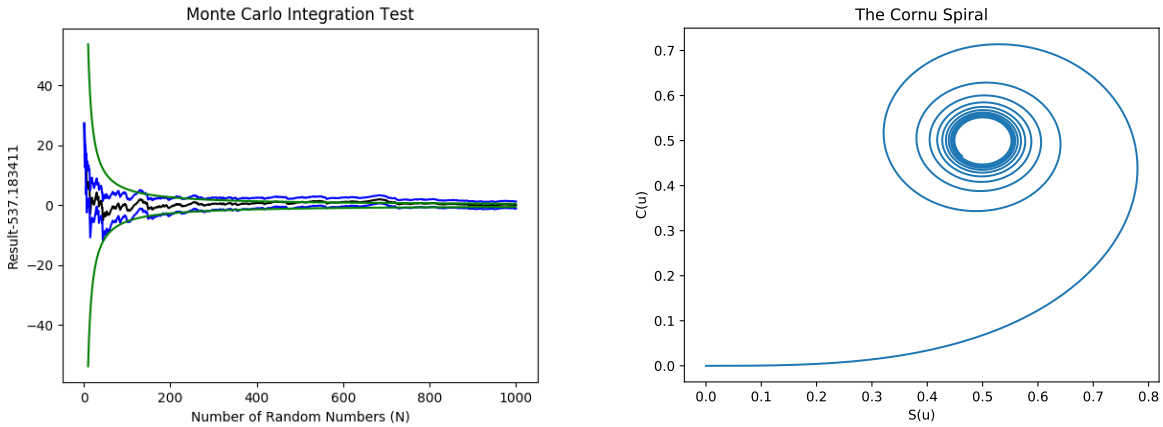

Figure 1

0.0.1 IntegrateMK2.py Code

```
import numpy as np
import operator
import matplotlib.pyplot as plt
def function(coordinate):
        return np.sin(coordinate)

def Monte_Gradual_integrate(Lower_Limit=0,Upper_Limit=np.pi/8,Dimensions=8,N=10):
        count_in=0; square_count_in=0;Results=[];Error=[]
        Volume = (Upper_Limit-Lower_Limit)**Dimensions

        for i in range(1,int(N)+1):
                coordinate = 0
                for j in range(int(Dimensions)):
                        coordinate += np.random.rand()*(Upper_Limit-Lower_Limit) + Lower_Limit
                count_in+=float(function(coordinate)); square_count_in += float((function(coordinate
                #print(count_in, square_count_in)
                Results.append(Volume*count_in/i)
                Error.append(Volume*(np.power(((-np.power(((count_in)/i),2)+(square_count_in/i))/i),
        return [x*10**6 for x in Results], [x*(10**6) for x in Error]
N=1000
Results, Error = Monte_Gradual_integrate(N=N)


plt.plot(range(1,len(Results)+1),Results,color='black')
plt.plot(range(1,len(Results)+1),list(map(operator.sub, Results, Error)),color='blue')
NError = [-x for x in Error]
plt.plot(range(1,len(Results)+1),list(map(operator.sub,Results,NError)),color='blue')
plt.xlabel('Number of Random Numbers (N)')
plt.title('Monte Carlo Integration Test')
plt.savefig('New_Plot.pdf')
plt.clf()

plt.plot(range(1,len(Results)+1),[x-537.1873411 for x in Results],color='black')
plt.plot(range(1,len(Results)+1),[x-537.1873411 for x in list(map(operator.sub, Results, Error))],co
NError = [-x for x in Error]
```

```
plt.plot(range(1,len(Results)+1),[x-537.1873411 for x in list(map(operator.sub,Results,NError))],col
plt.xlabel('Number of Random Numbers (N)')
plt.ylabel('Result -537.183411')
plt.title('Monte Carlo Integration Test')
plt.savefig('Comparative_Plot.pdf')
u = [537.183411/x for x in range(10,1000)]
l = [-537.183411/x for x in range(10,1000)]
plt.plot(range(10,1000),u,color='green')
plt.plot(range(10,1000),l,color='green')
plt.savefig('Compartative_Plot_Add_Lines.png')
```

0.0.2 CornuIntegrate.py Code

```
import scipy
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import numpy.random as rand
from scipy.integrate import quad

def cosser(x):
        return np.cos(np.pi*(x**2)/2)

def sinner(x):
        return np.sin(np.pi*(x**2)/2)

def Integrator(u):
        C = quad(cosser,0,u)[0]
        S = quad(sinner,0,u)[0]
        return C,S

x=[]; y=[]
for i in np.linspace(0,2*np.pi,num=1000):
        C,S = Integrator(i)
        y.append(S); x.append(C)

plt.plot(x,y)
plt.xlabel('S(u)')
plt.ylabel('C(u)')
plt.title('The Cornu Spiral')
plt.savefig('Cornu_Spiral.pdf')
```

Apertures

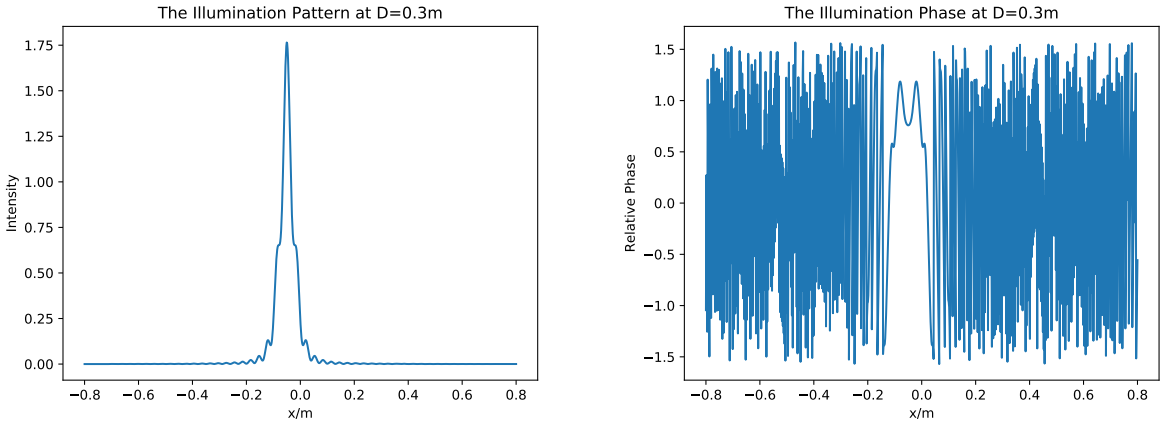

Figure 2

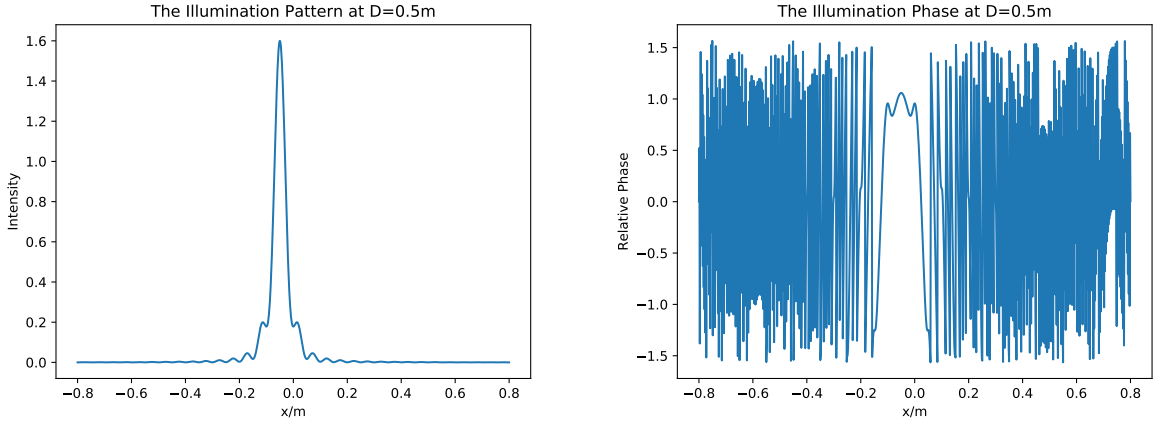

Figure 3

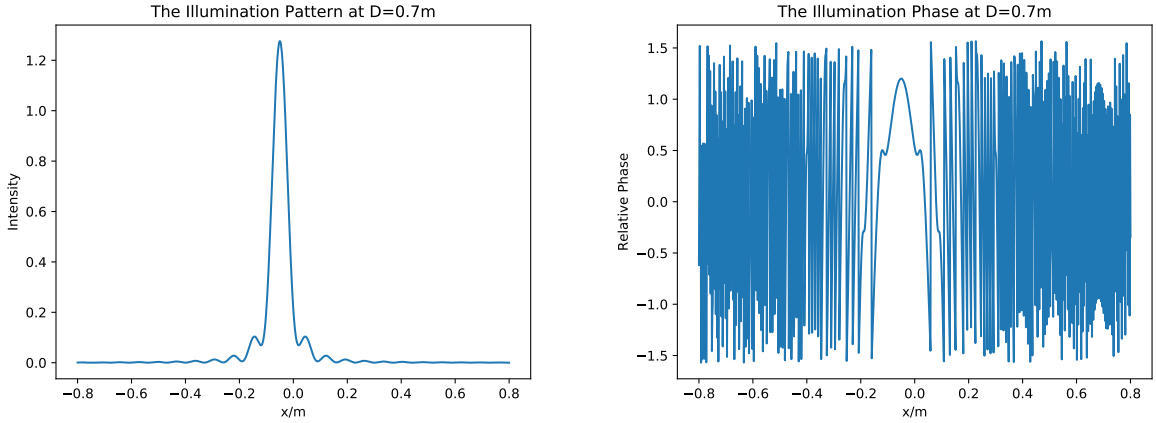

Figure 4

Aperture Code

```
import scipy
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import numpy.random as rand
from scipy.integrate import quad

def cosser(x):
        return np.cos(np.pi*(x**2)/2)

def sinner(x):
        return np.sin(np.pi*(x**2)/2)

def Integrator(u):
        C = quad(cosser,0,u)[0]
        S = quad(sinner,0,u)[0]
        return C,S

x=[]; y=[]

def Two_Ended(x0,x1,lam=0.01,D=0.3):
        Scaling = (2/(lam*D))**0.5
        u2 = x1*Scaling; u1 = x0*Scaling
        imag1,real1 = Integrator(u1)
        imag2,real2 = Integrator(u2)
        imag = -imag1+imag2
        real = -real1+real2
        mag = (imag**2+real**2)*0.5
        arg = np.arctan(imag/real)
        return mag, arg


def Plot_Mag_Arg(d=0.1,lam = 0.01,D=0.3):
        mag_list = []; arg_list =[]
        x = np.linspace(-0.8,0.8,num=1000)
        for i in x:
                mag, arg = Two_Ended(i,i+d,D=D,lam=lam)
                mag_list.append(mag); arg_list.append(arg)
        plt.plot(x,mag_list)
        plt.xlabel('x/m')
        plt.ylabel('Intensity')
        plt.title('The Illumination Pattern at D='+str(D)+'m')
        plt.savefig('Apperture_Pattern_'+str(D)+'_'+'.pdf')
        plt.clf()
        plt.plot(x,arg_list)
        plt.xlabel('x/m')
        plt.ylabel('Relative Phase')
        plt.title('The Illumination Phase at D='+str(D)+'m')
        plt.savefig('Phase_Pattern_'+str(D)+'_.pdf')
        plt.clf()

Plot_Mag_Arg()
Plot_Mag_Arg(D=0.5)
Plot_Mag_Arg(D=0.7)
```

Exercise 2

The effect of order

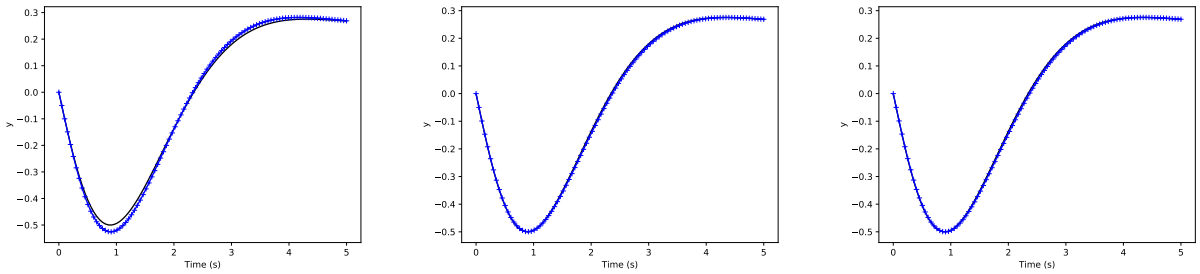

Figure 5

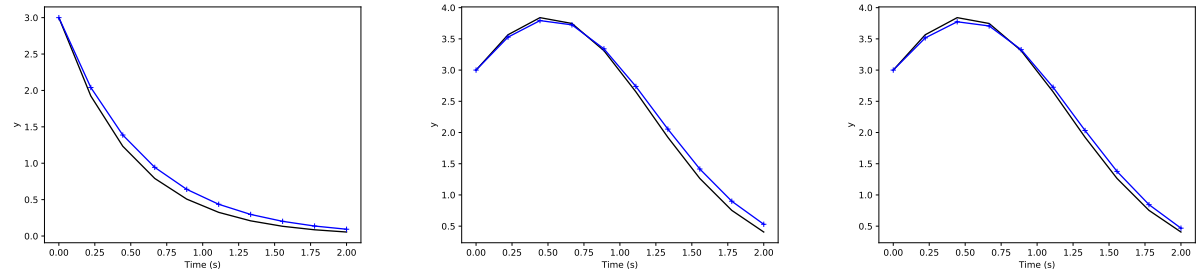

Figure 6

The effect of time-step

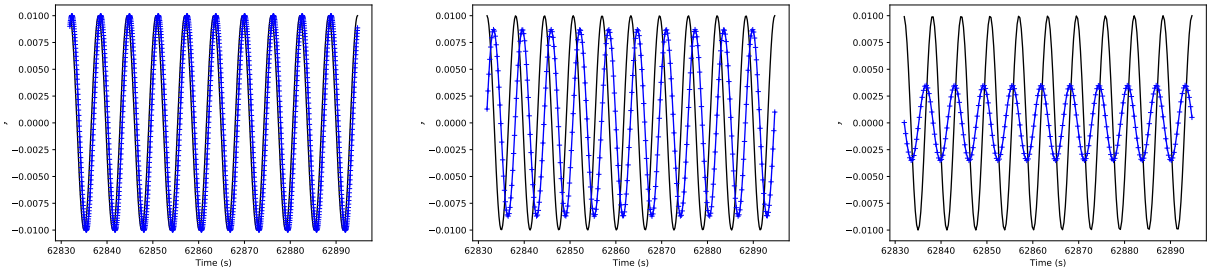

Figure 7

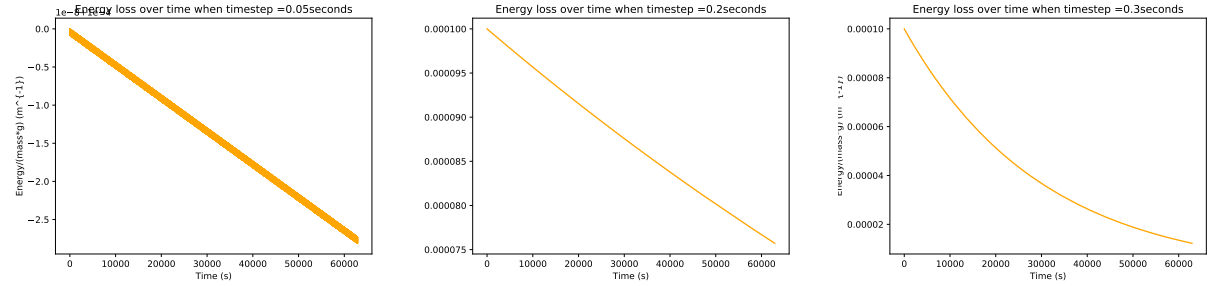

Figure 8

Light Damping

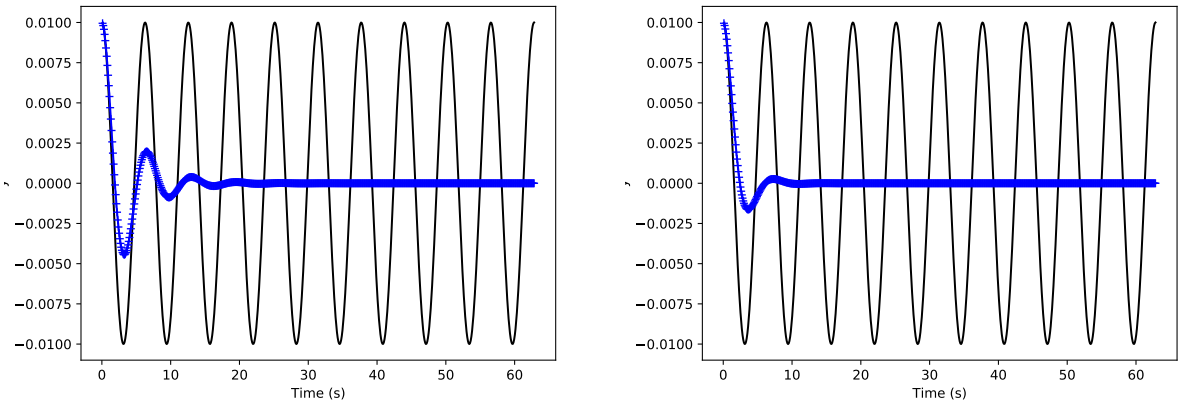

Figure 9

Overdamping

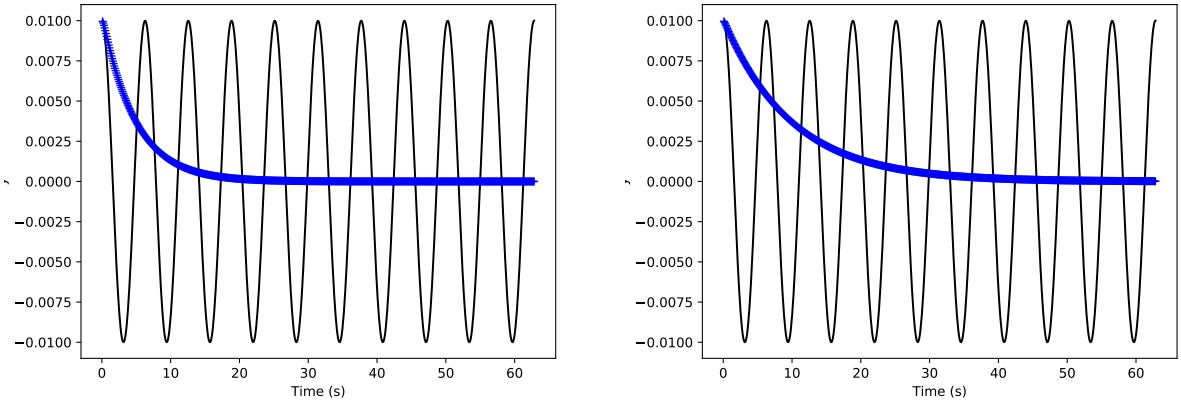

Figure 10

Forced

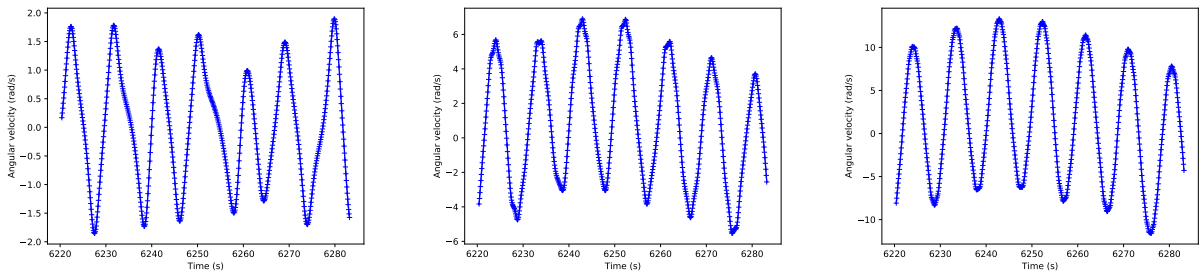

Figure 11: Apparent period of oscillation (when Forcing=0.5) (from sign changes of ang velocity): 9.42477796077 Apparent period of oscillation (when Forcing=1.2) (from sign changes of ang velocity): 11.0879740715 Apparent period of oscillation (when Forcing=1.44) (from sign changes of ang velocity): 9.42477796077 Apparent period of oscillation (when Forcing=1.465) (from sign changes of ang velocity): 9.42477796077

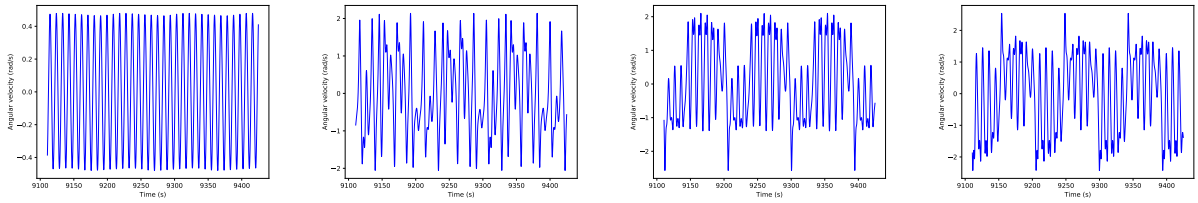

Figure 12

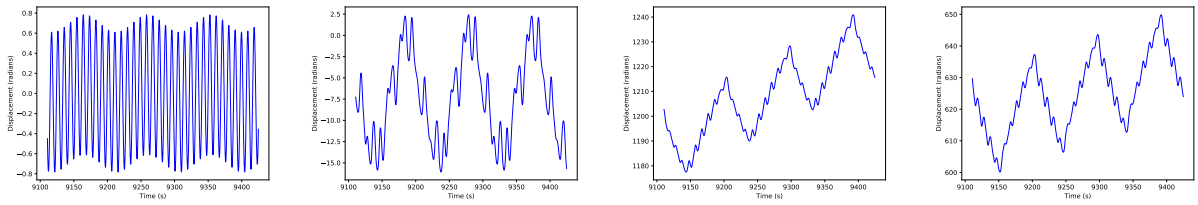

Figure 13

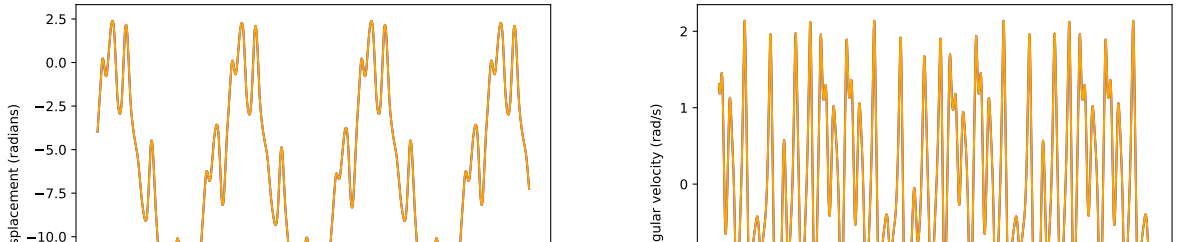

Figure 14

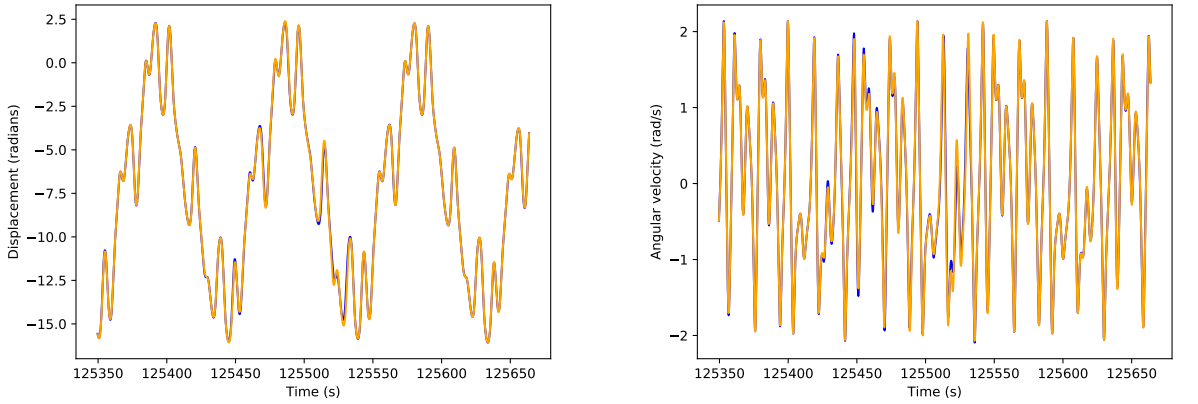

Figure 15

Task 1 Code

```
#More complicated pendulum for task1
import numpy as np
import matplotlib.pyplot as plt

h=0.05;                         # Time step
z=0.0 ## z is damping factor gamma - q was already in use
b=1 ## b = g/l ## = w0**2 in small angle limit --> w0 = 1 rad/s --> f = 1/(2pi) Hz --> T = 2pi secon
F=0 # no forcing in this task
omega_d=1 # not used in this task
initial_displacement=0.01
oscillations=170


A = np.array([[0,1],[0,-z]])
B =np.array([[0],[1]])

mint = (oscillations-10)*2*np.pi
maxt = oscillations*2*np.pi # given that b = 1
t = np.linspace(0,maxt,num=int(maxt/h))

def y_maker(time):
        return initial_displacement*np.cos(time) # the theoretical value in the small angle limit

def k_maker(q_value,time):
        return A.dot(q_value)+B*(-np.sin(q_value[0][0])*b+F*np.sin(omega_d*time))
displacement = []
acceleration = []
pend_time = []
energy = [] # Energy by weight

def Pendulum(init=initial_displacement):
        k1 = np.zeros((2,1))
        q1 = np.zeros((2,1))
        q0 = np.zeros((2,1))
        counter = 0
        clicker = 0
        for i in t:
                if i < 0.95*h:
                        q0[0][0]=init; q0[1][0]=0
                else:
                        k1 = k_maker(q0,i*h)  # Approx for y gives approx for deriv
                        q1 = (q0 +k1*h/2)
                        k2 = k_maker(q1,i*h +h/2)
                        q2 = (q0 +k2*h/2)
                        k3 = k_maker(q2,i*h+h/2)
                        q3 = (q0 + k3*h)
                        k4 = k_maker(q3,(i+1)*h)
                        q1=q0+(k1+2*k2+2*k3+k4)*(h/6)        # Intermediate value
                        if q1[0][0]*q0[0][0]<0: # should click every time passes through origin
                                clicker +=1
                        q0=q1

                if counter%20 == 0 and False:
                        energy.append(q0[0][0]**2 +q0[1][0]**2); pend_time.append(i)
                counter +=1
                if i > mint and False:
                        displacement.append(q0[0][0])
                        acceleration.append(q0[1][0])
                        pend_time.append(i)

        return (2*i)/clicker

theta = np.linspace(0.001,np.pi, num = 50)
period = []
```

```
print(Pendulum(init=np.pi/2))
if True:
        for init_d in theta:
                period.append(Pendulum(init=init_d))
        plt.plot(theta,period,color='blue',marker='+')
        plt.xlabel('Initial Displacement (radians)')
        plt.ylabel('Period (seconds)')

        plt.title('Period vs Initial Displacement for a Pendulum (T(pi/2)='"{:.2f}".format(Pendulum
        plt.savefig('Period_Displacement.pdf')
        plt.clf()

if False:
        a = Pendulum()

if False:
        yexact = [y_maker(x) for x in pend_time]
        plt.plot(pend_time,yexact,color='black')
        plt.plot(pend_time,displacement,color='blue', marker='+')
        plt.xlabel('Time (s)')
        plt.ylabel('y')
        plt.savefig('Task1Pendulum.pdf')

if False:
        plt.plot(pend_time,energy,color='orange')
        plt.xlabel('Time (s)')
        plt.ylabel('Energy/(mass*g) (m^{-1})')
        plt.title('Energy loss over time when timestep ='+str(h)+'seconds')
        plt.savefig('Energy_Loss_h '+str(h)+' .pdf')
        plt.clf()
#legend('Exact','Approximate');
```

Task 2 Code

```
#Pendulum for task2 by sdat2
#Usage: 2_Task2Pendulum.py [damping coeff] [forcing coeff] [timestep] [oscillations modelled]
#Example: 2_Task2Pendulum.py 0.5 0.5 0.1 10

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


### Global Variable defaults ###
h=0.05;                         # Time step
z=0.0 ## z is damping factor gamma - q was already in use
b=1 ## b = g/l
F=0 # Forcing
omega_d=2/3 # the freq of driving
initial_displacement=0.01
oscillations=170
graph_os = 10
sample_os = 20

### Command Line Inputs ###
if len(sys.argv) >1: z = float(sys.argv[1])
if len(sys.argv) >2: F = float(sys.argv[2])
if len(sys.argv) >3: h = float(sys.argv[3])
if len(sys.argv) >4: oscillations = int(sys.argv[4])
if len(sys.argv) >5: graph_os = int(sys.argv[5])
if len(sys.argv) >6: sample_os = int(sys.argv[6])


#Matrices from solving 2nd order ODE
A = np.array([[0,1],[0,-z]])
B =np.array([[0],[1]])

#it is only ever useful to plot about 10 oscillations worth
mint = (oscillations-graph_os)*2*np.pi
mint2=(oscillations -sample_os)*2*np.pi
maxt = oscillations*2*np.pi # given that b = 1
t = np.linspace(0,maxt,num=int(maxt/h))

#sometimes it is useful to compare to perfect SHM
def y_maker(time):
        return initial_displacement*np.cos(time)

def k_maker(q_value,time): #(Otherwise known as the derivative!)
        return A.dot(q_value)+B*(-np.sin(q_value[0][0])*b+F*np.sin(omega_d*time))

displacement = []
velocity = []
pend_time = []


def Pendulum(init=initial_displacement):
        k1 = np.zeros((2,1))
        q1 = np.zeros((2,1))
        q0 = np.zeros((2,1))
        counter = 0
        clicker = 0
        for i in t:
                if i < 0.95*h:
                        q0[0][0]=init; q0[1][0]=0
                else:
                        k1 = k_maker(q0,i)  # Approx for y gives approx for deriv
                        q1 = (q0 +k1*h/2)
                        k2 = k_maker(q1,i +h/2)
```

```
                        q2 = (q0 +k2*h/2)
                        k3 = k_maker(q2,i+h/2)
                        q3 = (q0 + k3*h)
                        k4 = k_maker(q3,i*h)
                        q1=q0+(k1+2*k2+2*k3+k4)*(h/6)        # Intermediate value
                        if q1[1][0]*q0[1][0]<0 and i >mint2: # should click every time passes throug
                                clicker +=1
                        q0=q1

                counter +=1
                if i > mint:
                        displacement.append(q0[0][0])
                        velocity.append(q0[1][0])
                        pend_time.append(i)
        return (2*(i-mint2))/clicker


if True:
        apparent_period = Pendulum(init=initial_displacement)

if True:
        #yexact = [y_maker(x) for x in pend_time]
        #plt.plot(pend_time,yexact,color='black')
        plt.plot(pend_time,displacement,color='blue')
        plt.xlabel('Time (s)')
        plt.ylabel('Displacement (radians)')
        plt.savefig('T2Displacement_'+str(F)+'_Forcing.pdf')
        plt.clf()
if True:
        plt.plot(pend_time,velocity,color='blue')
        plt.xlabel('Time (s)')
        plt.ylabel('Angular velocity (rad/s)')
        plt.savefig('T2Velocity_'+str(F)+'_Forcing.pdf')

print('Apparent period of oscillation (when Forcing='+str(F)+') (from sign changes of ang velocity):
```

Task 3 Code

```
#Pendulum for task2 by sdat2
#Usage: 2_Task2Pendulum.py [damping coeff] [forcing coeff] [timestep] [oscillations modelled]
#Example: 2_Task2Pendulum.py 0.5 0.5 0.1 10

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


### Global Variable defaults ###
h=0.05;                         # Time step
z=0.0 ## z is damping factor gamma - q was already in use
b=1 ## b = g/l
F=0 # Forcing
omega_d=2/3 # the freq of driving
initial_displacement=0.01
oscillations=170
graph_os = oscillations
sample_os = 20

### Command Line Inputs ###
if len(sys.argv) >1: z = float(sys.argv[1])
if len(sys.argv) >2: F = float(sys.argv[2])
if len(sys.argv) >3: h = float(sys.argv[3])
if len(sys.argv) >4: oscillations = int(sys.argv[4])
if len(sys.argv) >5: graph_os = int(sys.argv[5])
if len(sys.argv) >6: sample_os = int(sys.argv[6])


#Matrices from solving 2nd order ODE
A = np.array([[0,1],[0,-z]])
B =np.array([[0],[1]])

#it is only ever useful to plot about 10 oscillations worth
mint = (oscillations-graph_os)*2*np.pi
mint2=(oscillations -sample_os)*2*np.pi
maxt = oscillations*2*np.pi # given that b = 1
t = np.linspace(0,maxt,num=int(maxt/h))

#sometimes it is useful to compare to perfect SHM
def y_maker(time):
        return initial_displacement*np.cos(time)

def k_maker(q_value,time,z=z,F=F): #(Otherwise known as the derivative!)
        A = np.array([[0,1],[0,-z]])
        B =np.array([[0],[1]])
        return A.dot(q_value)+B*(-np.sin(q_value[0][0])*b+F*np.sin(omega_d*time))

def Pendulum(init=initial_displacement,z=z,F=F):
        displacement = []
        velocity = []
        pend_time = []
        k1 = np.zeros((2,1))
        q1 = np.zeros((2,1))
        q0 = np.zeros((2,1))
        counter = 0
        clicker = 0
        for i in t:
                if i < 0.95*h:
                        q0[0][0]=init; q0[1][0]=0
                else:
                        k1 = k_maker(q0,i,z=z,F=F)
                        q1 = (q0 +k1*h/2)
                        k2 = k_maker(q1,i +h/2,z=z,F=F)
```

```
                        q2 = (q0 +k2*h/2)
                        k3 = k_maker(q2,i+h/2,z=z,F=F)
                        q3 = (q0 + k3*h)
                        k4 = k_maker(q3,i*h,z=z,F=F)
                        q1=q0+(k1+2*k2+2*k3+k4)*(h/6)        # Intermediate value
                        if q1[1][0]*q0[1][0]<0 and i >mint2: # should click every time passes throug
                                clicker +=1
                        q0=q1

                counter +=1
                if i > mint:
                        displacement.append(q0[0][0])
                        velocity.append(q0[1][0])
                        pend_time.append(i)
        return (2*(i-mint2))/clicker, displacement, velocity, pend_time


if True:
        apparent_period,DispA,VelA,TA = Pendulum(init=0.20)
        apparent_period,DispB,VelB,TB = Pendulum(init=0.20001)

if True:
        plt.plot(TA,DispA,color='blue')
        plt.plot(TB,DispB,color='orange')
        plt.xlabel('Time (s)')
        plt.ylabel('Displacement (radians)')
        plt.savefig('T3_Displacement.pdf')
        plt.clf()
if True:
        plt.plot(TA,VelA,color='blue')
        plt.plot(TB,VelB,color='orange')
        plt.xlabel('Time (s)')
        plt.ylabel('Angular velocity (rad/s)')
        plt.savefig('T3_Velocity.pdf')

print('Apparent period of oscillation (when Forcing='+str(F)+') (from sign changes of ang velocity):
```

Exercise 3

Task 1

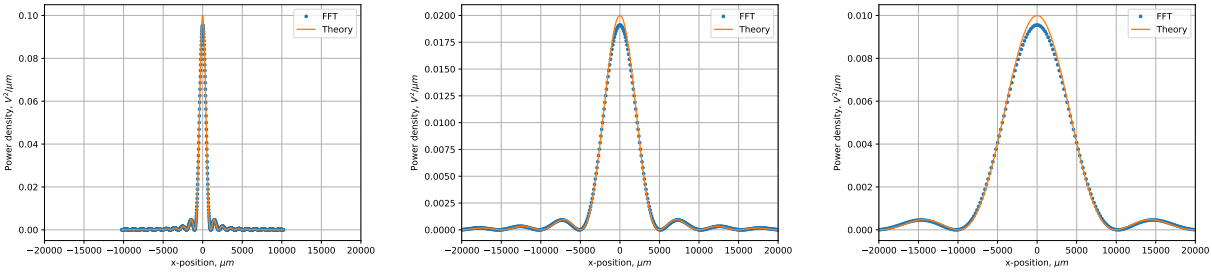

Figure 16

Task 2

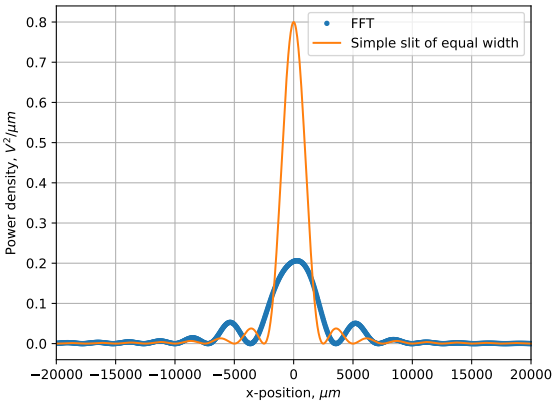

Figure 17

Task 3

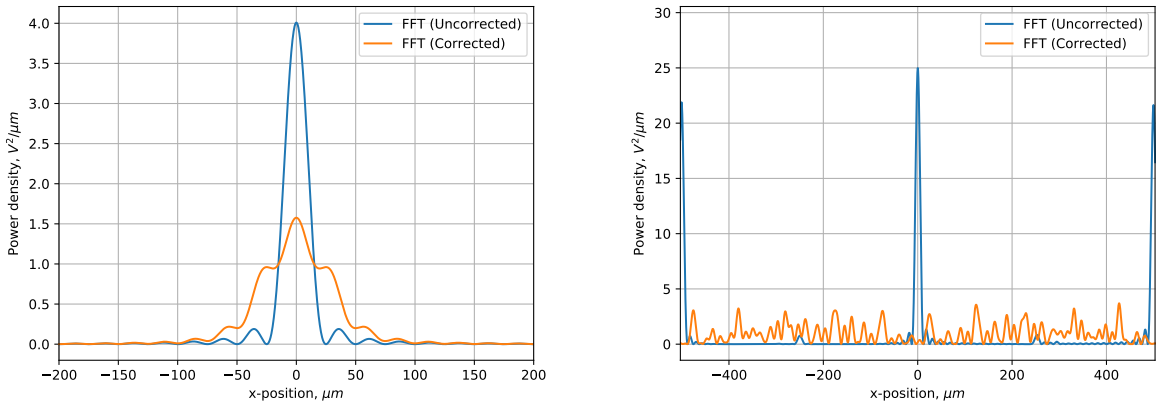

Figure 18

Task 1 Code

```
#program for task 1 by sdat2 drawing on
# http://kmdouglass.github.io/posts/approximating-diffraction-patterns-of-rectangular-apertures-with

import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
from scipy.fftpack import fftshift, ifftshift
import sys


def sinc(x):
    if (x != 0):
        # Prevent divide-by-zero
        return np.sin(np.pi * x) / (np. pi * x)
    else:
        return 1
sinc = np.vectorize(sinc)

#globals
amplitude    = 1     # Volt / sqrt(micron)
slitWidth    = 100     # microns
wavelength   = 0.5 # microns
propDistance = 10**6 # microns (= 1 mm)
bins = 2048
AppertureWidth = 5000

#redefine by Command Line
if len(sys.argv)>1: bins = int(sys.argv[1])
if len(sys.argv)>2: wavelength = float(sys.argv[2])
if len(sys.argv)>3: slitwidth = float(sys.argv[3])
if len(sys.argv)>4: AppertureWidth = int(sys.argv[4])
if len(sys.argv)>5: propogation_distance = float(sys.argv[5])


x     = np.linspace(-AppertureWidth/2, AppertureWidth/2, num = bins)
field = np.zeros(x.size, dtype='complex128') # Field complex
field[np.logical_and(x > -slitWidth / 2, x <= slitWidth / 2)] = amplitude + 0j
#creates an apperture of the correct width and amp of 1
dx = x[1] - x[0] # Spatial sampling period, microns
fS = 1 / dx      # Spatial sampling frequency, units are inverse microns
f  = (fS / x.size) * np.arange(0, x.size, step = 1) # a frequency type vector
############ACTUAL FFT#######################
diffractedField = dx * np.fft.fft(np.fft.fftshift(field)) # take the FFT of the apperture function
# fftshift uses the symmetry of the appeture to get rid of negative part

xPrime    = np.hstack((f[-int((f.size/2)):] - fS, f[0:int(f.size/2)])) * wavelength * propDistance
IntensTheory = amplitude / (wavelength * propDistance) * \
    (slitWidth * sinc(xPrime * slitWidth / wavelength / propDistance))**2
IntensFFT    = np.fft.fftshift(diffractedField * np.conj(diffractedField)) / wavelength / propDistan

plt.plot(xPrime, np.abs(IntensFFT), '.', label = 'FFT')
plt.plot(xPrime, IntensTheory, label = 'Theory')
plt.xlim((-20000, 20000))
plt.xlabel(r'x-position, $\mu m$')
plt.ylabel(r'Power density, $V^2 / \mu m$')
plt.grid(True)
plt.legend()
plt.savefig('Task1_lam='+str(wavelength)+'_d='+str(slitWidth)+'_D='+str(AppertureWidth)+'_microns.pd
```

Task 2 Code

```
#program for task 2 by sdat2 drawing on
# http://kmdouglass.github.io/posts/approximating-diffraction-patterns-of-rectangular-apertures-with

import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
from scipy.fftpack import fftshift, ifftshift
import sys


def sinc(x):
    if (x != 0):
        # Prevent divide-by-zero
        return np.sin(np.pi * x) / (np. pi * x)
    else:
        return 1
sinc = np.vectorize(sinc)

#globals
amplitude    = 1     # Volt / sqrt(micron)
slitWidth    = 2000     # microns
wavelength   = 0.5 # microns
propDistance = 10*10**6 # microns (= 10 m)
bins = 2048
AppertureWidth = 100000
m=8
s=100

#redefine by Command Line
if len(sys.argv)>1: bins = int(sys.argv[1])
if len(sys.argv)>2: wavelength = float(sys.argv[2])
if len(sys.argv)>3: slitwidth = float(sys.argv[3])
if len(sys.argv)>4: AppertureWidth = int(sys.argv[4])
if len(sys.argv)>5: propogation_distance = float(sys.argv[5])


x     = np.linspace(-AppertureWidth/2, AppertureWidth/2, num = bins)
field = np.zeros(x.size, dtype='complex128') # Field complex

def phase(x):
    return m/2*np.sin(2*np.pi*x/s)
for i in range(len(x)):
    if x[i] > -slitWidth / 2 and x[i] <= slitWidth / 2:
        field[i] = amplitude*(np.cos(phase(x[i])) + np.sin(phase(x[i]))*1j)
#creates an apperture of the correct width and amp of 1
dx = x[1] - x[0] # Spatial sampling period, microns
fS = 1 / dx      # Spatial sampling frequency, units are inverse microns
f  = (fS / x.size) * np.arange(0, x.size, step = 1) # a frequency type vector
############ACTUAL FFT#######################
diffractedField = dx * np.fft.fft(np.fft.fftshift(field)) # take the FFT of the apperture function
# fftshift uses the symmetry of the appeture to get rid of negative part

xPrime    = np.hstack((f[-int((f.size/2)):] - fS, f[0:int(f.size/2)])) * wavelength * propDistance
IntensTheory = amplitude / (wavelength * propDistance) * \
    (slitWidth * sinc(xPrime * slitWidth / wavelength / propDistance))**2
IntensFFT    = np.fft.fftshift(diffractedField * np.conj(diffractedField)) / wavelength / propDistan

plt.plot(xPrime, np.abs(IntensFFT), '.', label = 'FFT')
plt.plot(xPrime, IntensTheory, label = 'Simple slit of equal width')
plt.xlim((-20000, 20000))
plt.xlabel(r'x-position, $\mu m$')
plt.ylabel(r'Power density, $V^2 / \mu m$')
plt.grid(True)
plt.legend()
```

```
plt.savefig('Task2_lam='+str(wavelength)+'_d='+str(slitWidth)+'_D='+str(ApertureWidth)+'_microns.pd
```

Task 3 Code

```
#program for task 2 by sdat2 drawing on
# http://kmdouglass.github.io/posts/approximating-diffraction-patterns-of-rectangular-apertures-with

import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
from scipy.fftpack import fftshift, ifftshift
import sys


def sinc(x):
    if (x != 0):
        # Prevent divide-by-zero
        return np.sin(np.pi * x) / (np. pi * x)
    else:
        return 1
sinc = np.vectorize(sinc)

#globals
amplitude     = 1      # Volt / sqrt(micron)
slitWidth     = 2000      # microns
wavelength    = 0.5 # microns
propDistance = 10*10**6 # microns (= 10 m)
bins = 2048
AppertureWidth = 100000
m=8
s=100
sinusoidal=False
plt_theory=False
x_lim = 20000

#redefine by Command Line
if len(sys.argv)>1: bins = int(sys.argv[1])
if len(sys.argv)>2: wavelength = float(sys.argv[2])
if len(sys.argv)>3: slitWidth = float(sys.argv[3])
if len(sys.argv)>4: AppertureWidth = int(sys.argv[4])
if len(sys.argv)>5: propDistance = float(sys.argv[5])
if len(sys.argv)>6: x_lim  = float(sys.argv[6])


x     = np.linspace(-AppertureWidth/2, AppertureWidth/2, num = bins)
field = np.zeros(x.size, dtype='complex128') # Field complex
newfield = np.zeros(x.size,dtype='complex128')

def phase(x):
    return m/2*np.sin(2*np.pi*x/s)
for i in range(len(x)):
    if x[i] > -slitWidth / 2 and x[i] <= slitWidth / 2:
        if sinusoidal:
            field[i] = amplitude*(np.cos(phase(x[i])) + np.sin(phase(x[i]))*1j)
        else:
            field[i]=amplitude + 0j
        newfield[i]=field[i]*np.exp(1j*np.pi*(x[i]**2)/(wavelength*propDistance))


#creates an apperture of the correct width and amp of 1
dx = x[1] - x[0] # Spatial sampling period, microns
fS = 1 / dx      # Spatial sampling frequency, units are inverse microns
f  = (fS / x.size) * np.arange(0, x.size, step = 1) # a frequency type vector
##########ACTUAL FFT#####################
diffractedField = dx * np.fft.fft(field) # take the FFT of the apperture function
newdiffractedField = dx * np.fft.fft(newfield)
# fftshift uses the symmetry of the appeture to get rid of negative part
```

```
xPrime    = np.hstack((f[-int((f.size/2)):] - fS, f[0:int(f.size/2)])) * wavelength * propDistance
IntensTheory = amplitude / (wavelength * propDistance) * \
    (slitWidth * sinc(xPrime * slitWidth / wavelength / propDistance))**2
IntensFFT     = np.fft.fftshift(diffractedField * np.conj(diffractedField)) / wavelength / propDistan
newIntensFFT =np.fft.fftshift(newdiffractedField * np.conj(newdiffractedField)) / wavelength / propD

plt.plot(xPrime, np.abs(IntensFFT), label = 'FFT (Uncorrected)')
plt.plot(xPrime, np.abs(newIntensFFT), label = 'FFT (Corrected)')
if plt_theory: plt.plot(xPrime, IntensTheory, label = 'Simple slit of equal width')
plt.xlim((-x_lim , x_lim))
plt.xlabel(r'x-position , $\mu m$')
plt.ylabel(r'Power density , $V^2 / \mu m$')
plt.grid(True)
plt.legend()
plt.savefig('Task3_lam='+str(wavelength)+'_d='+str(slitWidth)+'_D='+str(propDistance)+'_microns.pdf'
```