HW2_Solutions

January 29, 2024

[1]:
```python
from matplotlib import rcParams
rcParams['font.size'] = 16
rcParams['figure.figsize'] = (5,4)
```

1 Homework 2 Solutions

1.1 Problem 1: Adaptive integration

[2]:
```python
from math import sin, sqrt
import numpy as np
```

[3]:
```python
def f(x):
    return (np.sin(np.sqrt(100*x)))**2

target = 1e-6  # Target accuracy
a = 0.0
b = 1.0
N = 1     # Initial number of steps
h = b-a  # Initial step size

I1 = 0.5 * h * (f(a) + f(b))  # First approximation to the integral
print("N = %d, integral = %s" % (N, I1))

err = 1.0
while abs(err) > target:

    # Calculate a new trapezoidal estimate
    h *= 0.5
    N *= 2
    I2 = 0.5*I1
    x2 = a + (2 * np.arange(N//2) + 1) * h  # x values of the new steps␣
 ↪(between the previous step's x-values)
    I2 += (h * f(x2)).sum()

    # Compute error and save current estimate
    err = (I2-I1)/3
    I1 = I2
```

```
    print("N = %d, integral = %f, error = %g" % (N, I2, err))
```

```
N = 1, integral = 0.147979484546652
N = 2, integral = 0.325232, error = 0.0590841
N = 4, integral = 0.512283, error = 0.0623503
N = 8, integral = 0.402997, error = -0.0364285
N = 16, integral = 0.430103, error = 0.00903531
N = 32, integral = 0.448415, error = 0.00610377
N = 64, integral = 0.453913, error = 0.00183276
N = 128, integral = 0.455349, error = 0.000478524
N = 256, integral = 0.455711, error = 0.000120921
N = 512, integral = 0.455802, error = 3.03111e-05
N = 1024, integral = 0.455825, error = 7.58283e-06
N = 2048, integral = 0.455831, error = 1.89602e-06
N = 4096, integral = 0.455832, error = 4.74026e-07
```

1.2 Problem 2: Heat capacity of a solid

[4]:
```
import matplotlib.pyplot as plt
from gaussxw import gaussxw
```

[5]:
```
V = 0.001         # Volume in cubic meters
rho = 6.022e28    # Number density of aluminum
thetaD = 428.0    # Debye temperature of aluminum
kB = 1.38065e-23 # Boltzmann's constant
N = 50
x,w = gaussxw(N)
```

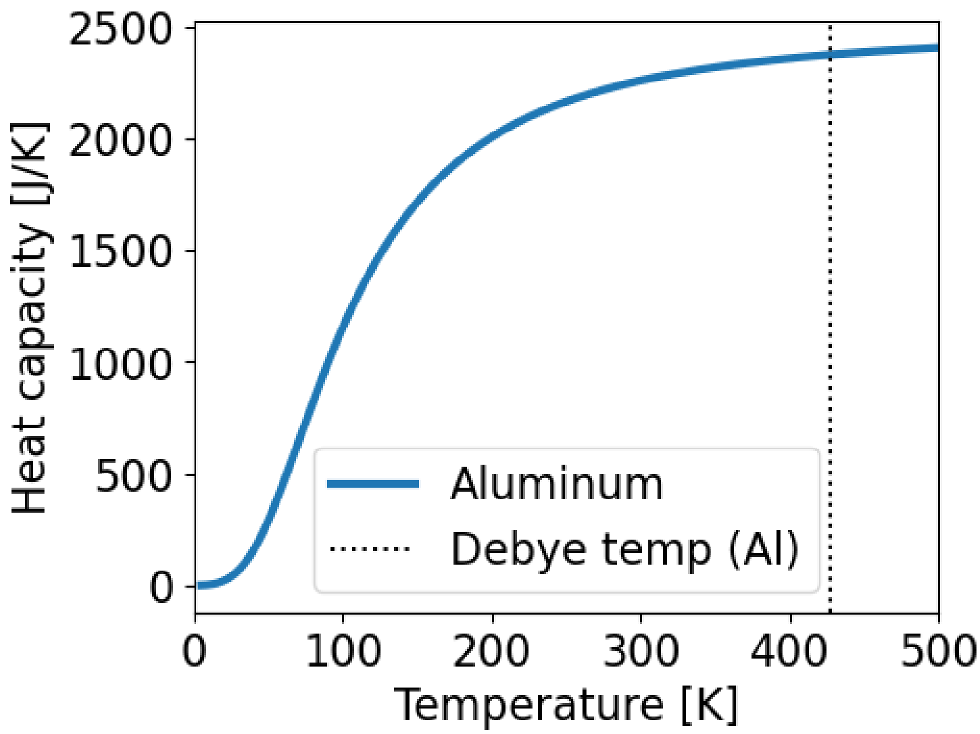
[6]:
```
# Integrand
def f(x):
    return x**4 * np.exp(x) / np.expm1(x)**2
```

[7]:
```
# Function to compute cv using Gaussian quadrature
def cv(T):
    a = 0.0
    b = thetaD / T
    xp = 0.5 * (b+a) + 0.5 * (b-a)*x  # scaled sampling points
    wp = 0.5 * (b-a) * w              # scaled weights
    series = wp * f(xp)               # weighted values
    return 9 * V * rho * kB * ((T/thetaD)**3) * series.sum()
```

[8]:
```
# Main program to plot graph
allT = np.arange(5,501)
all_cv = np.zeros(allT.size)
for i,T in enumerate(allT):
    all_cv[i] = cv(T)
```

[9]:

```python
plt.plot(allT, all_cv, lw=3, label='Aluminum')
plt.axvline(thetaD, c='k', ls=':', label='Debye temp (Al)')
plt.xlabel('Temperature [K]')
plt.ylabel('Heat capacity [J/K]')
plt.legend(loc='best')
plt.xlim(0,500)
plt.savefig('HW2-2.pdf')
plt.show()
```



1.3 Problem 3: Wave diffraction

[10]:

```python
wavelength = 1.0  # meters
z = 3.0           # height (meters)
N = 50
xx, ww = gaussxw(N)
```

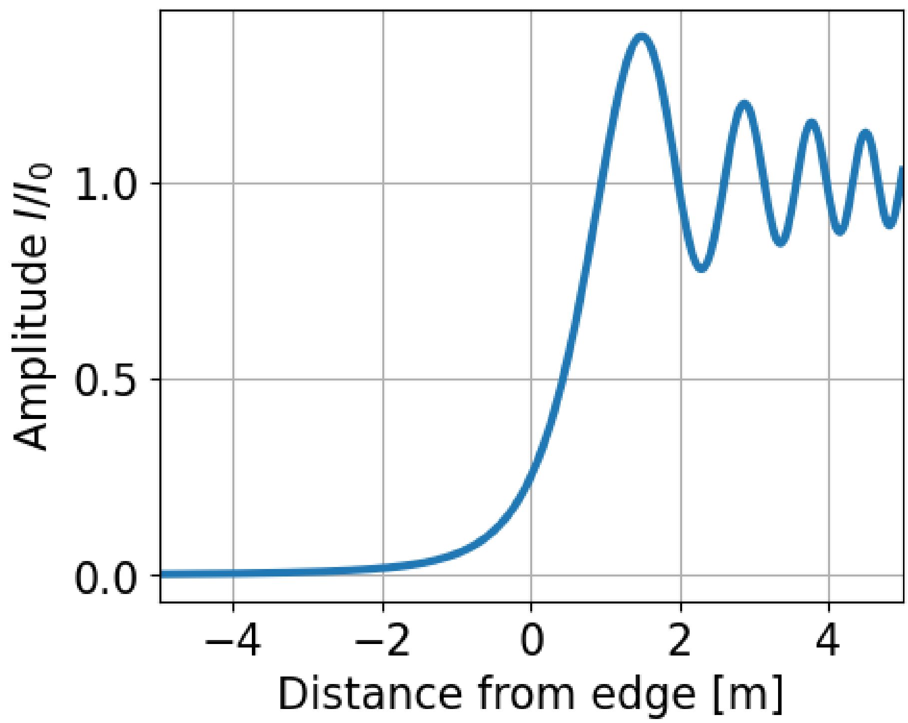
[11]:

```python
# Function to calculate the intensity at position x
def I(x):
    u = x * np.sqrt(2 / (wavelength*z))
    xp = 0.5 * u * (xx+1)  # Scaled sample points
    wp = 0.5 * u * ww      # Scaled weights

    # Elements of the Gaussian quadrature series
```

```
    C = wp * np.cos(0.5 * np.pi * xp**2)
    S = wp * np.sin(0.5 * np.pi * xp**2)

    return 0.125*((2*C.sum() + 1)**2 + (2*S.sum() + 1)**2)
```

[12]:
```
# Make a plot of the function from x = [-5,5]
xpoints = np.linspace(-5, 5, 500)
Ipoints = np.zeros(xpoints.size)
for i,x in enumerate(xpoints):
    Ipoints[i] = I(x)
```

[13]:
```
plt.plot(xpoints, Ipoints, lw=3)
plt.xlim(-5, 5)
plt.grid()
plt.xlabel('Distance from edge [m]')
plt.ylabel(r'Amplitude $I/I_0$')
plt.savefig('HW2-3.pdf')
plt.show()
```



1.4 Problem 4: Lotka-Volterra equations

[14]:
```
# Constants and parameters
alpha = 1.0
beta = gamma = 0.5
delta = 2.0
a = 0.0
b = 20.0
N = 1000
h = (b-a)/N
r = np.array([2,2], float)
```

[15]:
```
# Governing equations
def f(r,t):
    x = r[0]
    y = r[1]
    fx = alpha*x - beta*x*y
    fy = gamma*x*y - delta*y
    return np.array([fx,fy], float)
```

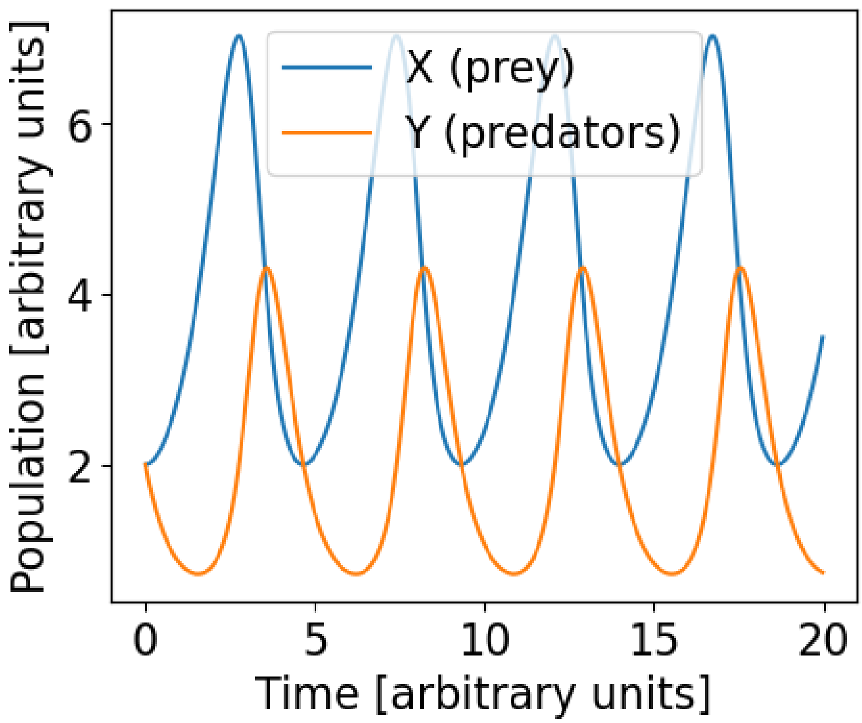
[16]:
```
tpoints = np.arange(a,b,h)
xpoints = []
ypoints = []
```

1.5 Main loop

[17]:
```
for t in tpoints:
    xpoints.append(r[0])
    ypoints.append(r[1])
    k1 = h*f(r,t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)
    r += (k1 + 2*k2 + 2*k3 + k4) / 6.0
```

1.6 Plot the results

[18]:
```
plt.plot(tpoints, xpoints, label='X (prey)')
plt.plot(tpoints, ypoints, label='Y (predators)')
plt.xlabel('Time [arbitrary units]')
plt.ylabel('Population [arbitrary units]')
plt.legend(loc='best')
plt.savefig('HW2-4.pdf')
plt.show()
```



As we can see, the solution of the equations oscillates, both of the populations going up and down with the same frequency, but with a delay between them. When there are few foxes, the the rabbits proliferate because they are not being eaten, but once there are many rabbits then the foxes get lots of food and start to proliferate too. But lots of foxes means the rabbit population goes down again, because they are getting eaten, and then the fox population goes down too, because the food supply has disappeared, and so the cycle starts again.