INFORMATICS INSTITUTE OF TECHNOLOGY

In Collaboration with

ROBERT GORDON UNIVERSITY ABERDEEN

CM2607 – Advanced Mathematics for Data Science

CW Report

by

Mohamed Rimsan Fathima Rashadha

RGU Username – 2122096

IIT Id - 20210001

December 2022

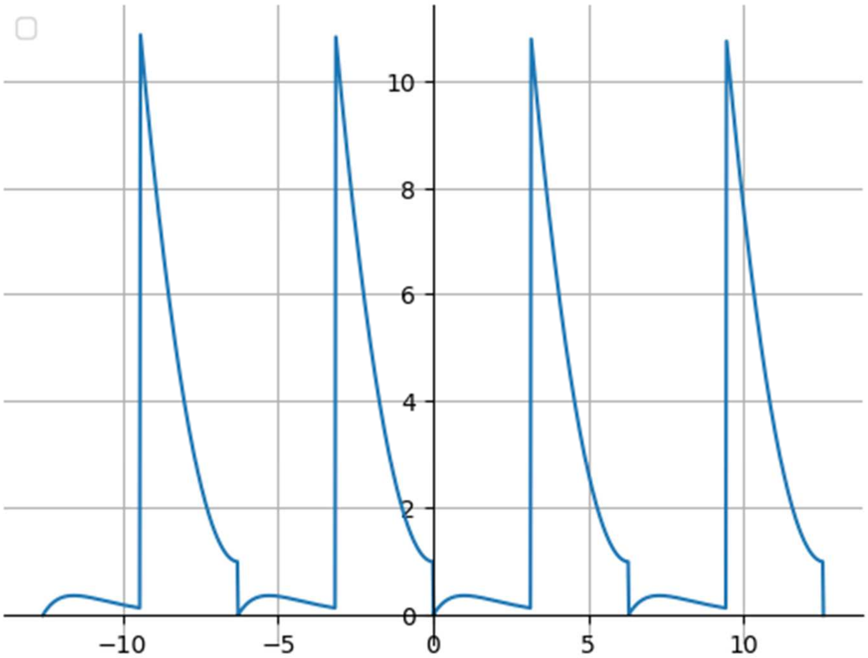
Question 1:

a)

code:

```
pip install matplotlib

def format():
   fig = plt.figure()
   ax = fig.add_subplot(1, 1, 1)
   ax.spines['left'].set_position('center')
   ax.spines['bottom'].set_position('zero')
   ax.spines['right'].set_color('none')
   ax.spines['top'].set_color('none')
   ax.xaxis.set_ticks_position('bottom')
   ax.yaxis.set_ticks_position('left')

def format_2():
   # formatting
   plt.legend(loc='upper left')
   plt.grid(True)

import numpy as np
import matplotlib.pyplot as plt

def periodicf(li, lf, f, x):
   if x >= li and x < lf:
      return f(x)
   elif x >= lf:
      x_new = x - (lf - li)
      return periodicf(li, lf, f, x_new)
   elif x < li:
      x_new = x + (lf - li)
      return periodicf(li, lf, f, x_new)

def f(x):
   if x >= -np.pi and x < 0:
      return x**2 + 1
   elif x >= 0 and x <= np.pi:
      return x*np.exp(-x)

x = np.linspace(-4*np.pi, 4*np.pi, 1000)
y = [periodicf(-np.pi, np.pi, f, xi) for xi in x]

format()
plt.plot(x, y)
format_2()
```

output:



b)

code:

```
import numpy as np
import scipy.integrate as integrate

# Define the function
def f(x):
   if x >= -np.pi and x < 0:
      return x**2 + 1
   elif x >= 0 and x <= np.pi:
      return x*np.exp(-x)

# Number of terms
N = 10

# Calculate the Fourier coefficients
a0 = (1/np.pi) * integrate.quad(lambda x: f(x), -np.pi, np.pi)[0]
an = [1/np.pi * integrate.quad(lambda x: f(x)*np.cos(n*x), -np.pi, np.pi)[0] for n in range(1,
N+1)]
bn = [1/np.pi * integrate.quad(lambda x: f(x)*np.sin(n*x), -np.pi, np.pi)[0] for n in range(1,
N+1)]

# Calculate the Fourier series approximation of f(x) (for the first 10 terms)
def series(N):
   series = f"{a0/2:.3f}"
   for n in range(1, N+1):
      series += f" + {an[n-1]:.3f} cos {n}x + {bn[n-1]:.3f} sin {n}x"
```

```
    return series

print(series(10))
```

output:

```
2.276 + -1.978 cos 1x + -2.317 sin 1x + 0.455 cos 2x + 1.602 sin 2x + -
0.244 cos 3x + -1.179 sin 3x + 0.107 cos 4x + 0.784 sin 4x + -0.090 cos
5x + -0.732 sin 5x + 0.047 cos 6x + 0.519 sin 6x + -0.046 cos 7x + -0.5
28 sin 7x + 0.026 cos 8x + 0.389 sin 8x + -0.028 cos 9x + -0.412 sin 9x
+ 0.017 cos 10x + 0.310 sin 10x
```

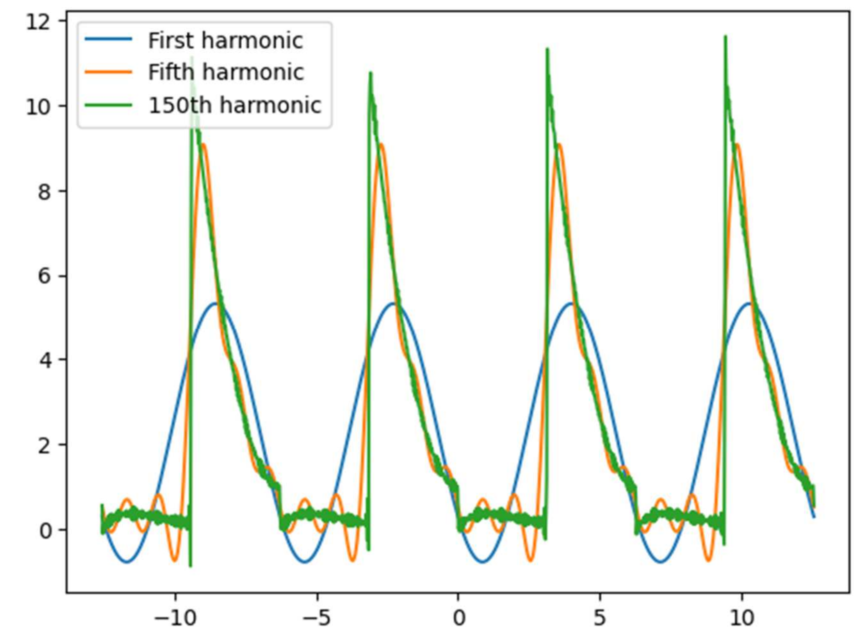
c)

code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate

# Define the function
def f(x):
    if x >= -np.pi and x < 0:
        return x**2 + 1
    elif x >= 0 and x <= np.pi:
        return x*np.exp(-x)

# Number of terms
N1 = 1
N5 = 5
N150 = 150
N200 = 200

# Calculate the Fourier coefficients
a0 = (1/np.pi) * integrate.quad(lambda x: f(x), -np.pi, np.pi)[0]
an1 = [1/np.pi * integrate.quad(lambda x: f(x)*np.cos(n*x), -np.pi, np.pi)[0] for n in range(1,
N1+1)]
bn1 = [1/np.pi * integrate.quad(lambda x: f(x)*np.sin(n*x), -np.pi, np.pi)[0] for n in range(1,
N1+1)]
an5 = [1/np.pi * integrate.quad(lambda x: f(x)*np.cos(n*x), -np.pi, np.pi)[0] for n in range(1,
N5+1)]
bn5 = [1/np.pi * integrate.quad(lambda x: f(x)*np.sin(n*x), -np.pi, np.pi)[0] for n in range(1,
N5+1)]
an150 = [1/np.pi * integrate.quad(lambda x: f(x)*np.cos(n*x), -np.pi, np.pi)[0] for n in
range(1, N150+1)]
bn150 = [1/np.pi * integrate.quad(lambda x: f(x)*np.sin(n*x), -np.pi, np.pi)[0] for n in
range(1, N150+1)]
# an200 = [2/np.pi * integrate.quad(lambda x: f(x)*np.cos(n*x), -np.pi, np.pi)[0] for n in
range(1, N200+1)]
# bn200 = [2/np.pi * integrate.quad(lambda x: f(x)*np.sin(n*x), -np.pi, np.pi)[0] for n in
range(1, N200+1)]
```

```
# Calculate the approximation of f(x) for the first, fifth, and 150th harmonics
def approximation1(x):
  sum = a0/2
  for n in range(1, N1+1):
    sum += an1[n-1]*np.cos(n*x) + bn1[n-1]*np.sin(n*x)
  return sum

def approximation5(x):
  sum = a0/2
  for n in range(1, N5+1):
    sum += an5[n-1]*np.cos(n*x) + bn5[n-1]*np.sin(n*x)
  return sum

def approximation150(x):
  sum = a0/2
  for n in range(1, N150+1):
    sum += an150[n-1]*np.cos(n*x) + bn150[n-1]*np.sin(n*x)
  return sum

# def approximation200(x):
#   sum = a0/2
#   for n in range(1, N200+1):
#     sum += an200[n-1]*np.cos(n*x) + bn200[n-1]*np.sin(n*x)
#   return sum

# Plot the original function and the approximations
x = np.linspace(-4*np.pi, 4*np.pi, 1000)
# y = f(x)
y1 = approximation1(x)
y5 = approximation5(x)
y150 = approximation150(x)
# y200 = approximation200(x)

# plt.plot
plt.plot(x, y1, label="First harmonic")
plt.plot(x, y5, label="Fifth harmonic")
plt.plot(x, y150, label="150th harmonic")
# plt.plot(x, y200, label="200th harmonic")
plt.legend()
plt.show()
# print(y1)
# print(y5)
# print(y150)
```

output:



d)

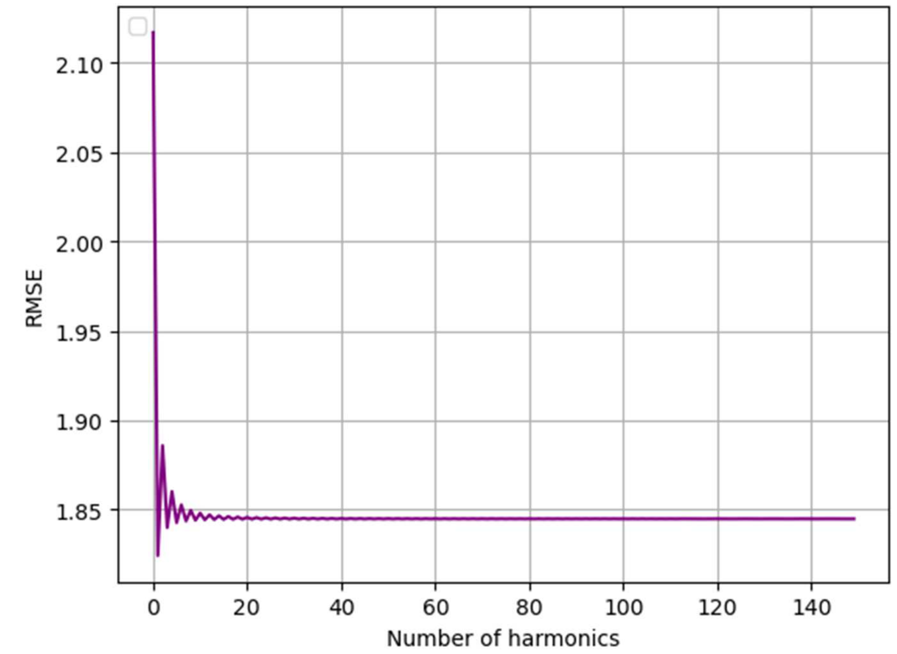
```
# Define the function
def f(x):
   return np.where((x>=-np.pi)&(x<0),(x**2)+1,
               np.where((x>=0)&(x<=np.pi),x*(np.exp(-x)),0))

def harmonic(x,n):
   return np.sin(n*x)/n

def rmse(f, harmonic, x):
   return np.sqrt(np.mean(np.square(f-harmonic)))

x = np.linspace(-4*np.pi,4*np.pi,1000)

rmse_values = []
for n in range(1,151):
   rmse_values.append(rmse(f(x),harmonic(x,n),x))

print("RMSE for first harmonic: ",rmse_values[0])
print("RMSE for fifth harmonic: ",rmse_values[4])
print("RMSE for 150th harmonic: ",rmse_values[149])
```

output:

```
RMSE for first harmonic:  2.1169969261117774
RMSE for second harmonic:  1.8601013674416376
RMSE for third harmonic:  1.844820717461884
```

code:
```
plt.plot(rmse_values,color="purple")
plt.xlabel('Number of harmonics')
plt.ylabel('RMSE')
format_2()
```

output:



explanation:
In general, as the number of harmonics included in the approximation increases, the accuracy of the approximation will also increase. This is because the more harmonics that are included, the more closely the truncated Fourier series will approximate the original function.

However, as the number of harmonics increases, the computation time for the approximation will also increase. This trade-off between accuracy and computation time is reflected in the plot, where the root mean squared error decreases as the number of harmonics increases, but eventually reaches a point of diminishing returns where the improvement in accuracy is not worth the increased computation time.

Question 2:
In order to demonstrate aliasing in the discrete Fourier transform (DFT), I will use a sine wave as the function to be transformed

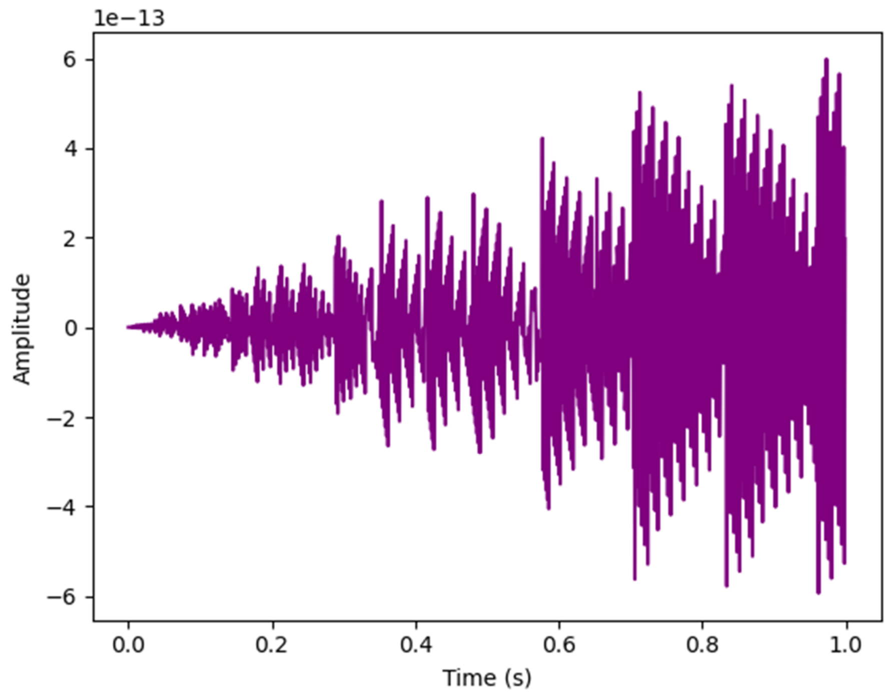
code:
```
pip install matplotlib
```

```
# function to format the graph
def format():
   fig = plt.figure()
   ax = fig.add_subplot(1, 1, 1)
   ax.spines['left'].set_position('center')
   ax.spines['bottom'].set_position('zero')
   ax.spines['right'].set_color('none')
   ax.spines['top'].set_color('none')
   ax.xaxis.set_ticks_position('bottom')
   ax.yaxis.set_ticks_position('left')# function to format the graph
def format_2():
   # formatting
   plt.legend(loc='upper left')
   plt.grid(True)
import numpy as np
import matplotlib.pyplot as plt

# Generate a sine wave signal with a frequency of 500 Hz
sampling_frequency = 1000
signal = np.sin(2 * np.pi * 500 * np.arange(0, 1, 1/sampling_frequency))

# Take the DFT of the signal
dft = np.fft.fft(signal)

# Get the frequencies of the DFT
frequencies = np.fft.fftfreq(signal.size, 1/sampling_frequency)

# Find the index of the Nyquist frequency
nyquist_index = np.argwhere(frequencies == frequencies[frequencies.size//2])[0, 0]

# Set the frequency of the signal to be higher than the Nyquist frequency
signal_aliased = np.sin(2 * np.pi * 3500 * np.arange(0, 1, 1/sampling_frequency))

# Take the DFT of the aliased signal
dft_aliased = np.fft.fft(signal_aliased)

# Plot the original signal and the aliased signal
print("Original signal")
plt.plot(np.arange(0, 1, 1/sampling_frequency), signal,color="purple")
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()

print("Aliased signal")
plt.plot(np.arange(0, 1, 1/sampling_frequency), signal_aliased,color="red")
plt.legend()
plt.xlabel('Time (s)')
```

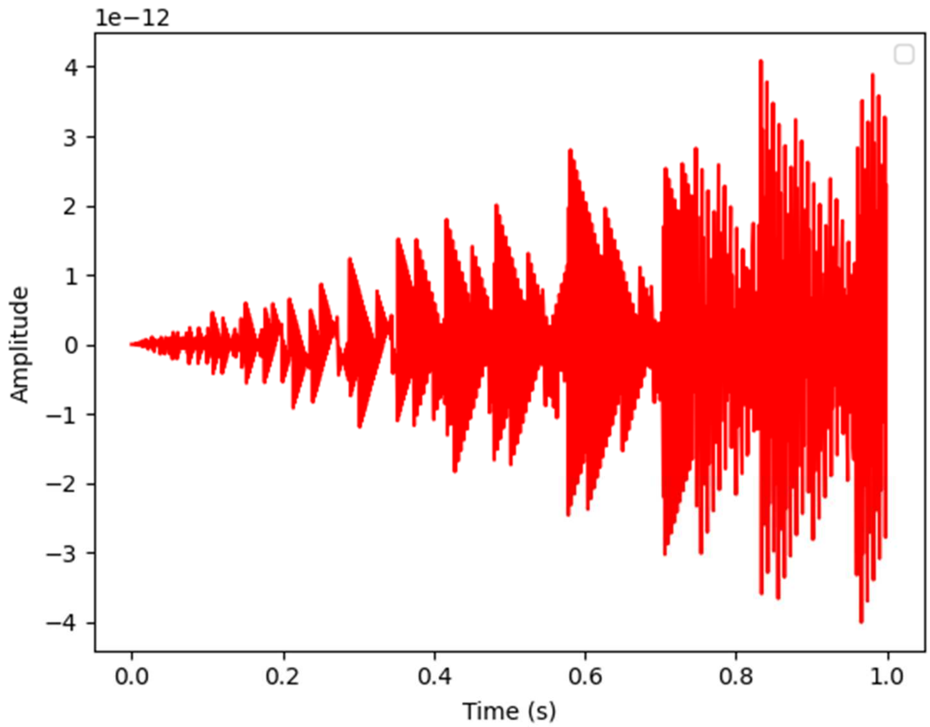
```
plt.ylabel('Amplitude')
plt.show()

# Plot the magnitude of the DFT coefficients for the original and aliased signals
print("Original signal")
plt.plot(frequencies, np.abs(dft),color="purple")
plt.legend()
plt.xlabel('Frequency (Hz)')
plt.ylabel('DFT Coefficient')
plt.show()

print("Aliased signal")
plt.plot(frequencies, np.abs(dft_aliased),color="red")
plt.xlabel('Frequency (Hz)')
plt.ylabel('DFT Coefficient')
plt.show()
```

outcome:

Original signal

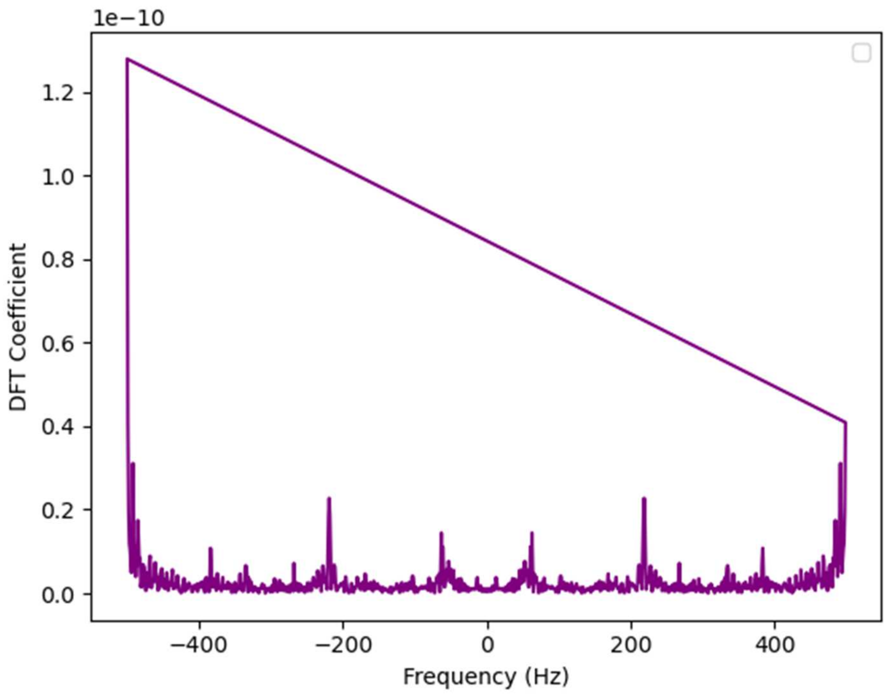


Aliased signal

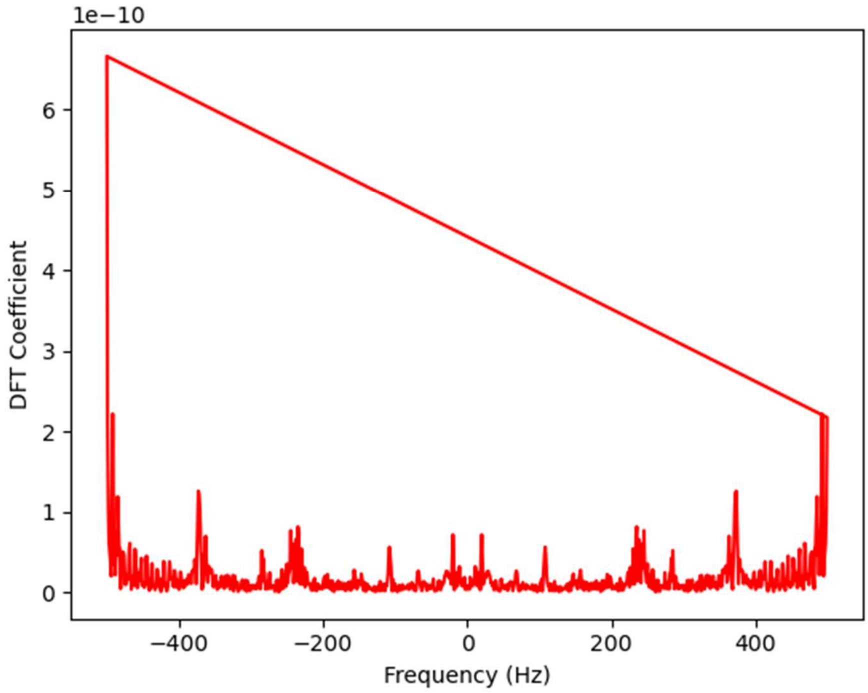


tart with an underscore are ignored when legend() is called with no argument.

Original signal



```
Aliased signal
```



explanation:
I chose to use a sine wave signal as the input
to the discrete Fourier transform (DFT). A sine
wave is a periodic signal with a single sinusoidal
component at a single frequency. This makes it easy to see how the DFT
represents the frequency components of the signal.

I also chose a sampling rate of 1000 Hz and a signal frequency of 500 Hz for the original signal. This means that the Nyquist frequency, which is the highest frequency that can be accurately represented by the DFT, is 500 Hz. This allows us to clearly see how aliasing occurs when the frequency of the signal is increased to 3500 Hz, which is above the Nyquist frequency.

Aliasing occurs in the DFT when the input signal has frequency components that are higher than the Nyquist frequency. In this case, the DFT will "wrap around" these high frequency components and represent them as lower frequency components. This can be seen in the plot of the magnitude of the DFT coefficients, where the coefficient for the aliased signal is different from the coefficient for the original signal at the Nyquist frequency.

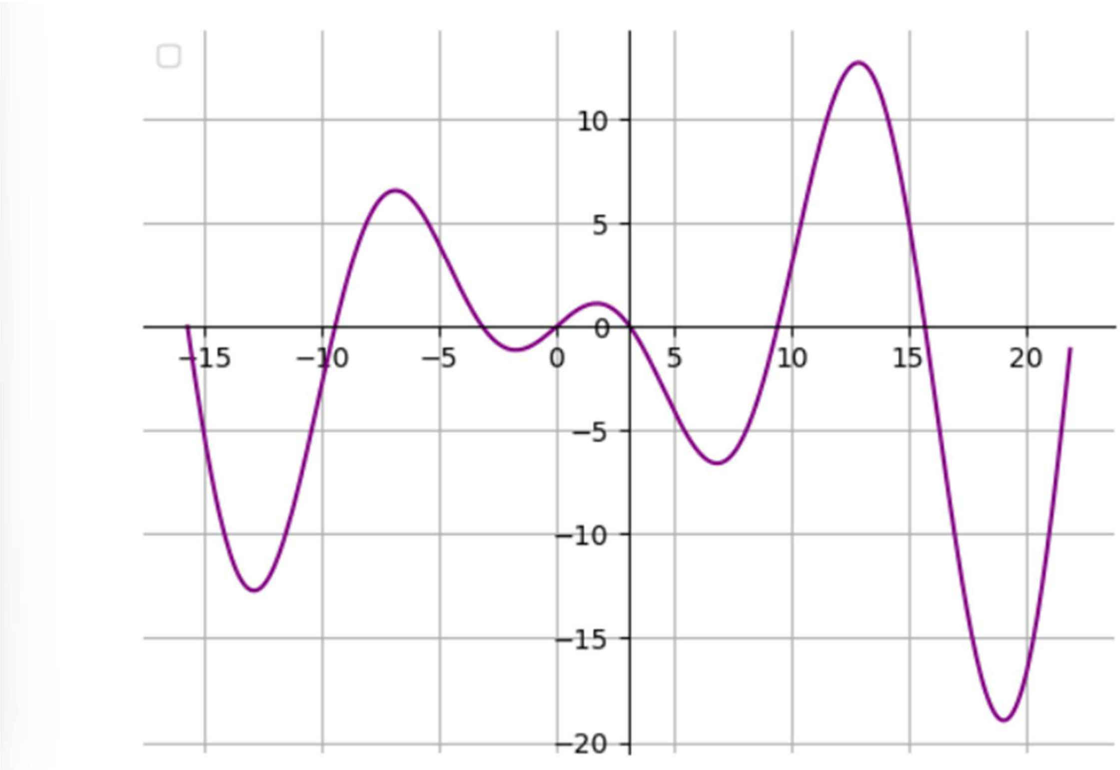
In general, it is important to choose an appropriate sampling rate that is at least twice the highest frequency of the input signal in order to avoid aliasing in the DFT. This is known as the Nyquist-Shannon sampling theorem.

Question 3

a) code:

```
pip install numpy
pip install matplotlib

import numpy as np
import matplotlib.pyplot as plt

# array of x values from -5π to 7π (step size = 0.1)
x = np.arange(-5*np.pi, 7*np.pi, 0.1)

# Calculating the y values for the function
y = x * np.cos(x/2)

# function to format the graph
def format():
  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)
  ax.spines['left'].set_position('center')
  ax.spines['bottom'].set_position('zero')
  ax.spines['right'].set_color('none')
  ax.spines['top'].set_color('none')
  ax.xaxis.set_ticks_position('bottom')
  ax.yaxis.set_ticks_position('left')

format()
plt.plot(x, y, color="purple")
format_2()
```

output:



b)

$f(x)$ = cos(π/2) + (x - π/2) cos'(π/2) + (x - π/2)^2 cos''(π/2) / 2! + (x - π/2)^3 cos'''(π/2) / 3! + ..

$f(x)$ = 1 + (x - π/2)^2 / 2! - (x - π/2)^4 / 4! + (x - π/2)^6 / 6! - ...

code:

```
import math

result = 0
# calculating the taylor series expansion for 6 terms
for i in range(6):
    term = (-1)**i / math.factorial(2*i) * (x - math.pi/2)**(2*i)
    result = result+term

# converting the expansion to a string
str = "cos(x) ≈ "
for i in range(6):
    if i > 0:
        str += " + "
    str += "({} / {}) * (x - {})^{}".format((-1)**i, math.factorial(2*i), math.pi/2, 2*i)

print(str)
```

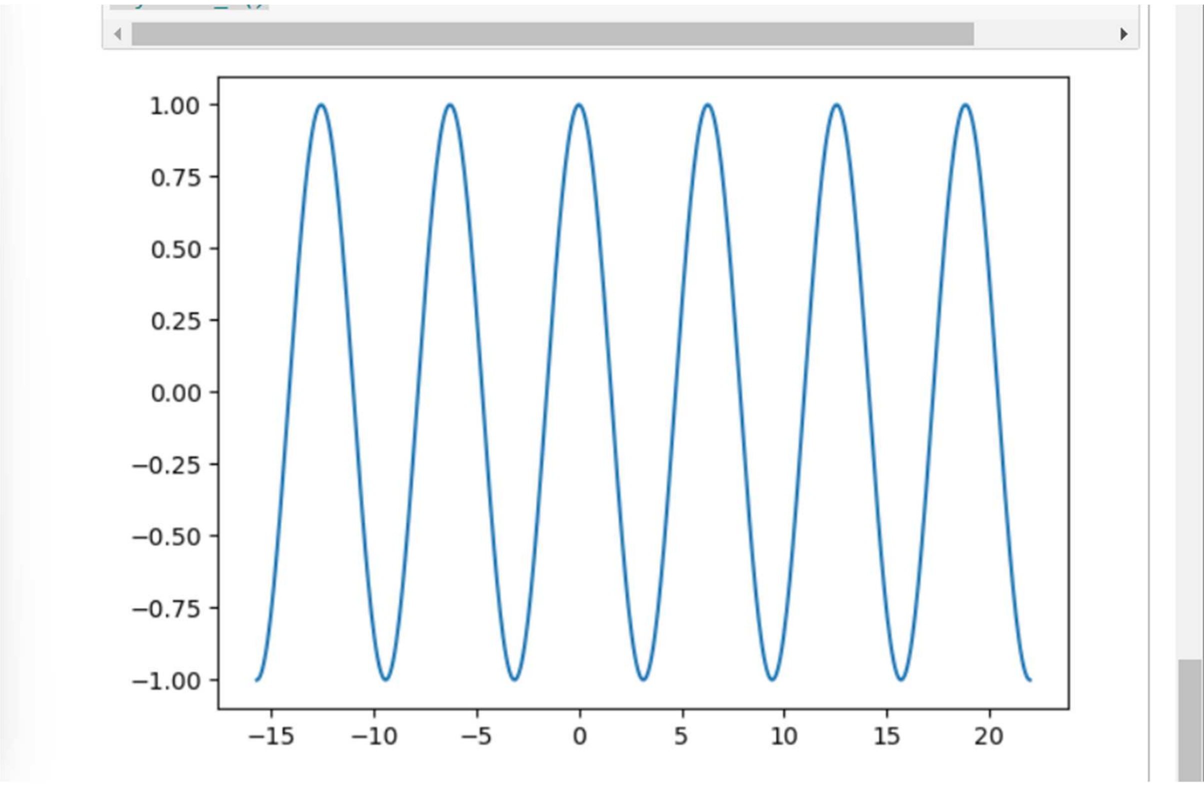
output:

```
cos(x) ≈ (1 / 1) * (x - 1.5707963267948966)^0 + (-1 / 2) * (
x - 1.5707963267948966)^2 + (1 / 24) * (x - 1.5707963267948966)^4 + (-1
/ 720) * (x - 1.5707963267948966)^6 + (1 / 40320) * (x - 1.570796326794
8966)^8 + (-1 / 3628800) * (x - 1.5707963267948966)^10
```

c)

```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

x = sym.symbols('x')
eq = sym.cos(x)

ms=np.empty(60,dtype=object) # 60
xrange=np.linspace(-5*np.pi,7*np.pi,500)
y=np.zeros([61,500]) # 61

ms[0] = eq.subs(x,0)
# print(ms[0])
f = sym.lambdify(x,ms[0],'numpy')
y[0,:] = f(xrange)
for n in range(1,60):
  ms[n] = ms[n-1]+(eq.diff(x,n).subs(x,np.pi/2)*((n-np.pi/2)**2)/(np.math.factorial(n)))
#   print((n+1),".",ms[n])
  f=sym.lambdify(x,ms[n],'numpy')
  y[n,:] = f(xrange)

f = sym.lambdify(x,eq,'numpy')
y[60,:]=f(xrange)
# plt.plot(xrange,y[0,:])
# plt.plot(xrange,y[4,:])
# plt.plot(xrange,y[9,:])
# format()
plt.plot(xrange,y[60,:])
# plt.legend(["1","5","10","func"])
plt.show()
# format_2()
```

output:



d)

```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

x = sym.symbols('x')
eq = sym.cos(x)

ms=np.empty(1000,dtype=object) # 60
xrange=np.linspace(20,-20,500)
y=np.zeros([1001,500]) # 61

ms[0] = eq.subs(x,0)
f = sym.lambdify(x,ms[0],'numpy')
y[0,:] = f(xrange)
for n in range(1,1000):
   ms[n] = ms[n-1]+(eq.diff(x,n).subs(x,np.pi/6)*((np.pi/6)**n)/(np.math.factorial(n)))
   f=sym.lambdify(x,ms[n],'numpy')
   y[n,:] = f(xrange)

f = sym.lambdify(x,eq,'numpy')
y[1000,:]=f(xrange)

# plt.plot(xrange,y[0,:])
# plt.plot(xrange,y[4,:])
# plt.plot(xrange,y[9,:])
```

```
# plt.plot(xrange,y[14,:])
# plt.plot(xrange,y[19,:])
# plt.plot(xrange,y[24,:])
# plt.plot(xrange,y[1000,:])
# # plt.legend(["1","5","10","15","20","25","1000"])
# plt.show()

actual_value = np.pi/3*np.cos(np.pi/6)
error = abs(ms[999] - actual_value)
print("actual value: ",actual_value)
print("absolute error: ",error)
```

output:

```
actual value:  0.9068996821171089
absolute error:  0.272925085901547
```

explanation:
The absolute error gives us a measure of how
close the approximation is to the actual value.
A smaller absolute error indicates a better approximation.
Since the absolute error in this case is 0.2729, we can say that it is close to the actual value.

Question 4:

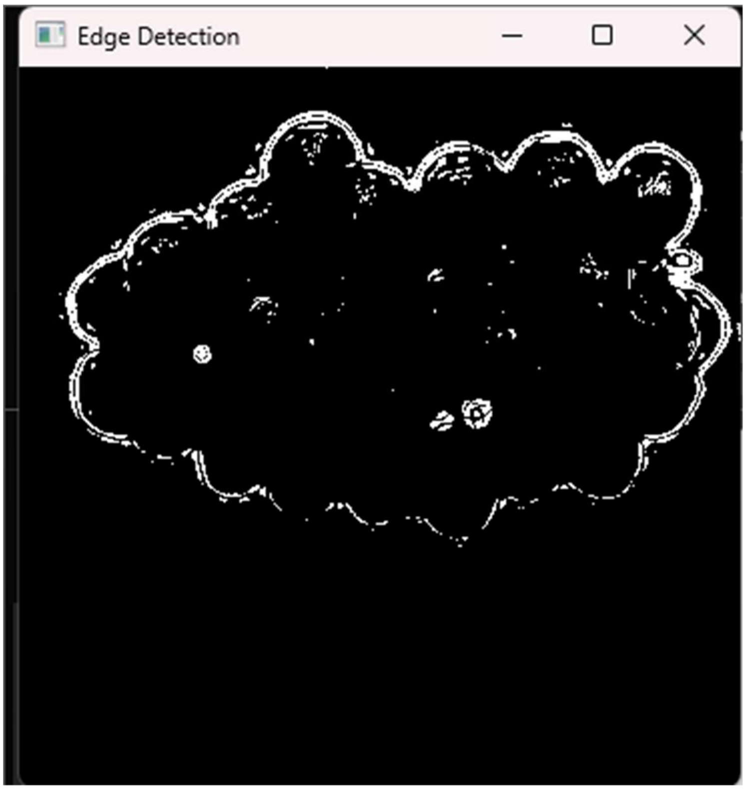
a)

code:

```
pip install numpy matplotlib
pip install opencv-python

import cv2
import numpy as np

# Loading the image into an array using OpenCV
image = cv2.imread('fruit.jpg')
# converting to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Applying a high pass filter to the image to enhance the edges
# performing 2D FT
ft = np.fft.fft2(gray)

# Shifting the zero-frequency component
# to the center of the spectrum
fshift = np.fft.fftshift(ft)

# Setting up the the low frequency components to 0
rows, cols = gray.shape
```

```
crow, ccol = rows // 2, cols // 2
fshift[crow-30:crow+30, ccol-30:ccol+30] = 0

# Appling inverse FT to convert the
# filtered image into the spatial domain
filtered_image = np.fft.ifftshift(fshift)
filtered_image = np.fft.ifft2(filtered_image)
filtered_image = np.abs(filtered_image)

# creating a binary image (filtere_image --> binary image)
#(the edges are white and the background is black in that image)
_, thresh = cv2.threshold(filtered_image, 15, 255, cv2.THRESH_BINARY)

# Displaying the image
cv2.imshow('Edge Detection', thresh)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

output:



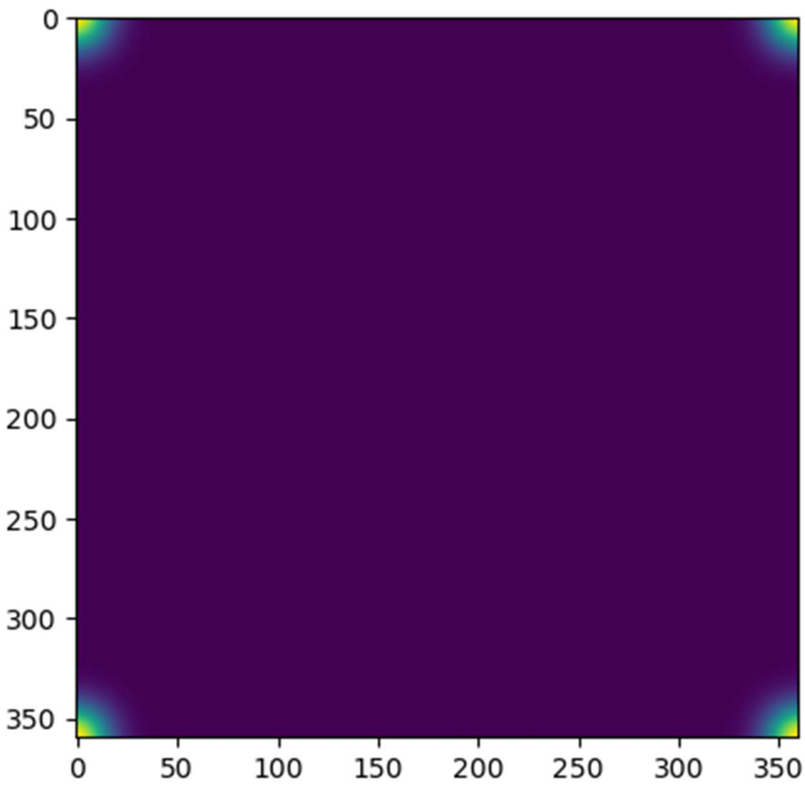
b)

code:

```
import matplotlib.image as mpimg
import scipy.fftpack as sfft
import scipy.signal as signal
import matplotlib.pyplot as plt
```

```
# reads the image
image = mpimg.imread("fruit.jpg")

#creates a Gaussian filter kernel (a two-dimensional array)
gaussian_filter_kernel = np.outer(signal.gaussian(360, 5), signal.gaussian(360, 5))

# performs a 2D fft on the kernal
# converts kernal from the spatial domain to the frequency domain
converted_kernal = sfft.fft2(sfft.ifftshift(gaussian_filter_kernel))  #freq domain kernel
# display freq domain kernal
plt.imshow(np.abs(converted_kernal))
plt.show()
```

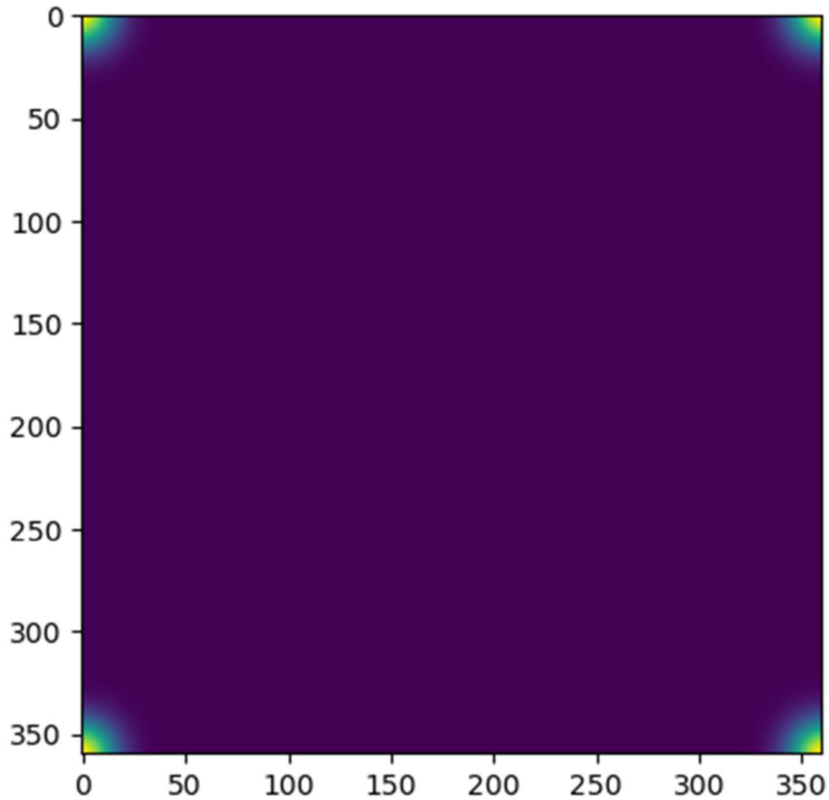
output:



code:

```
# performs a 2D fft on the image
fft_on_image = sfft.fft2(image)
# display freq domain image
plt.imshow(np.abs(converted_kernal))
plt.show()
```

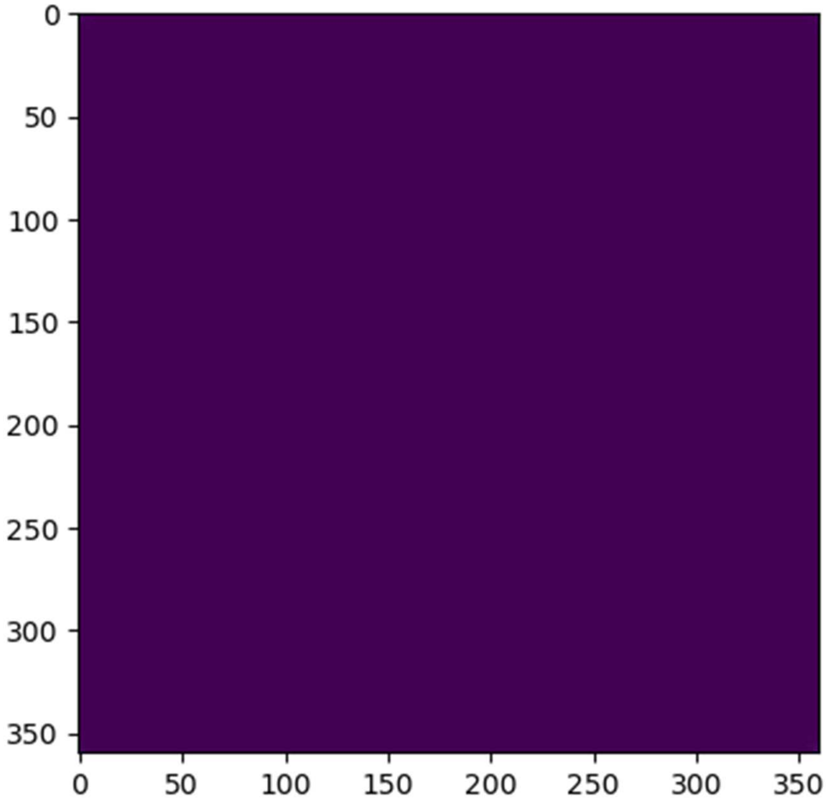
output:



code:

```
# applies the blur filter to the image
blur_filter_image = fft_on_image*converted_kernal
# displays the blurred frequency domain image
plt.imshow(np.abs(blur_filter_image))
plt.show()
```

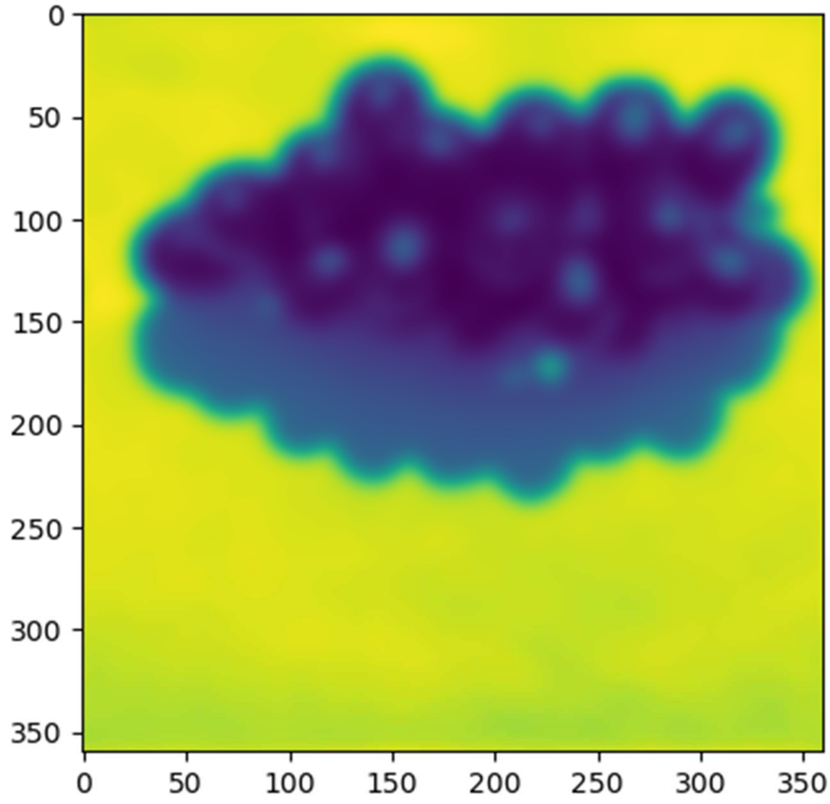
output:



code:

```
# performs an inverse two-dimensional FFT on the blurred frequency domain image
# converts the image back to the spatial domain,
# resulting in a blurred version of the original image
image2 = sfft.ifft2(blur_filter_image)
plt.imshow(np.abs(image2))
plt.show()
```

output:

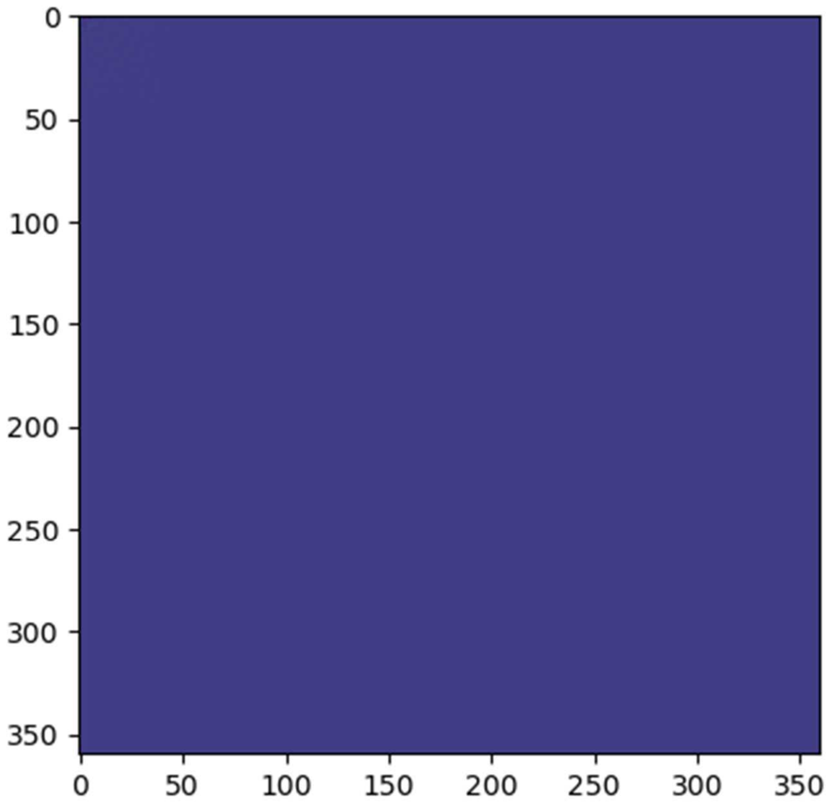


c)

code:

```
import matplotlib.image as mpimg
import scipy.fftpack as sfft

# reads the image
image = mpimg.imread("fruit.jpg")

# performs a 2D discrete cosine transform (DCT) on the image
# converts image from the spatial domain to the frequency domain
# done by decomposing the image into a series of cosine functions
# (each represents a different frequency component of the image)
freq_image = sfft.dct((sfft.dct(image,norm='ortho')).T,norm='ortho')
# display freq domain image
plt.imshow(freq_image)
plt.show()
```

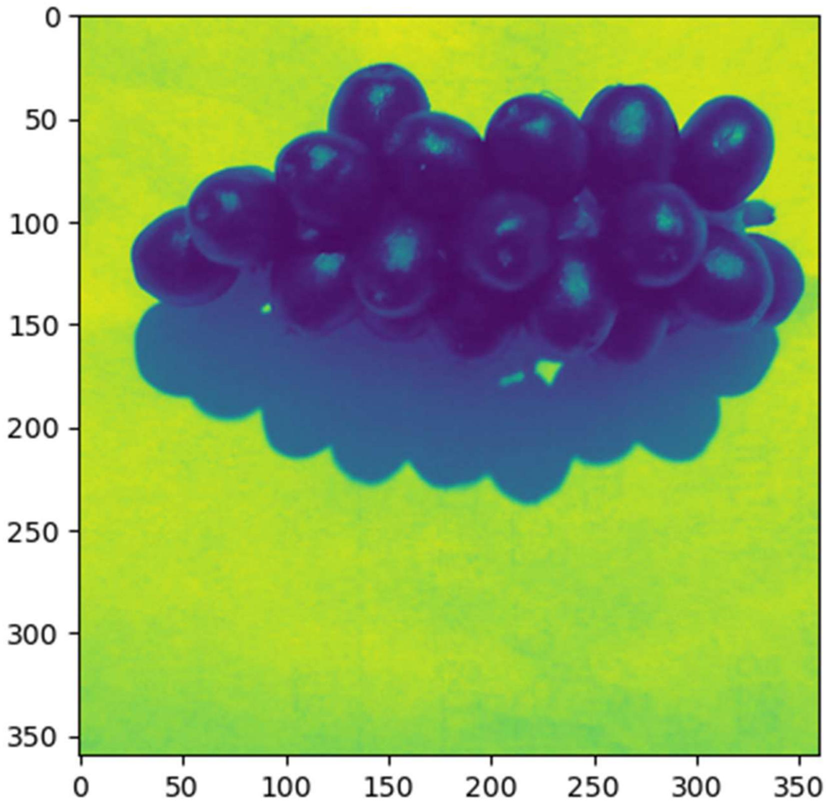
output:



code:

```
# performs an inverse 2D DCT on the freq domain image
# converts image back to the spatial domain
image_2 = sfft.idct((sfft.idct(freq_image,norm='ortho')).T,norm='ortho')
# display
plt.imshow(image_2)
plt.show()
```

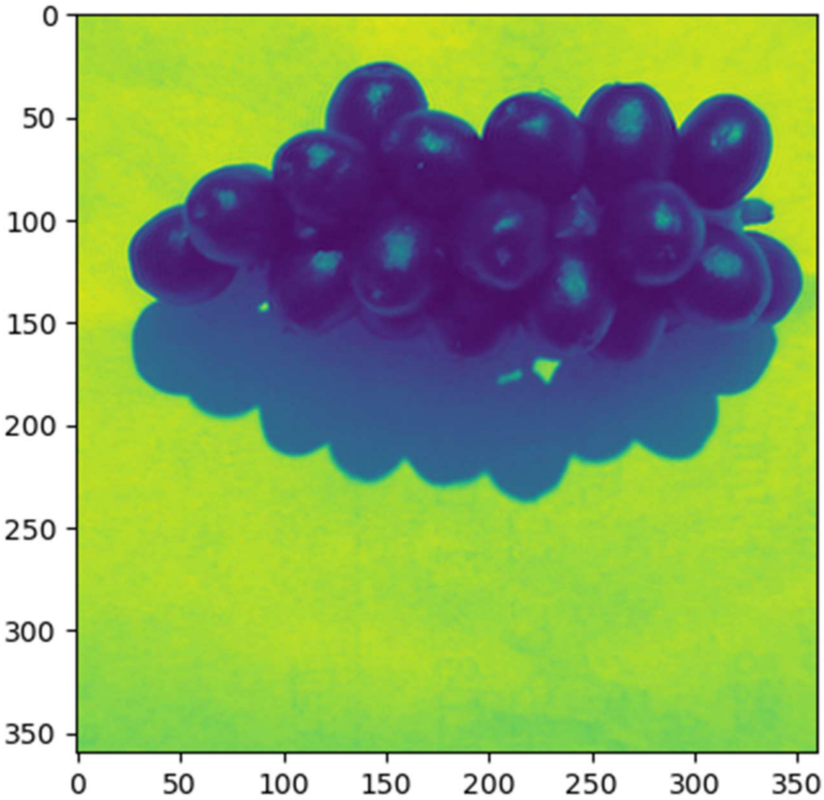
output:



code:

```
#Removing high frequency components
#creates a new image (is a copy of original freq domain image, but with the high freq components set to 0)
freq_image_2 = np.zeros((360,360))
freq_image_2[:240,:240] = freq_image[:240,:240]
# transform back to the spatial domain using the inverse 2D DCT
image_2 = sfft.idct((sfft.idct(freq_image_2,norm='ortho')).T,norm='ortho')
# display
plt.imshow(image_2)
plt.show()
```

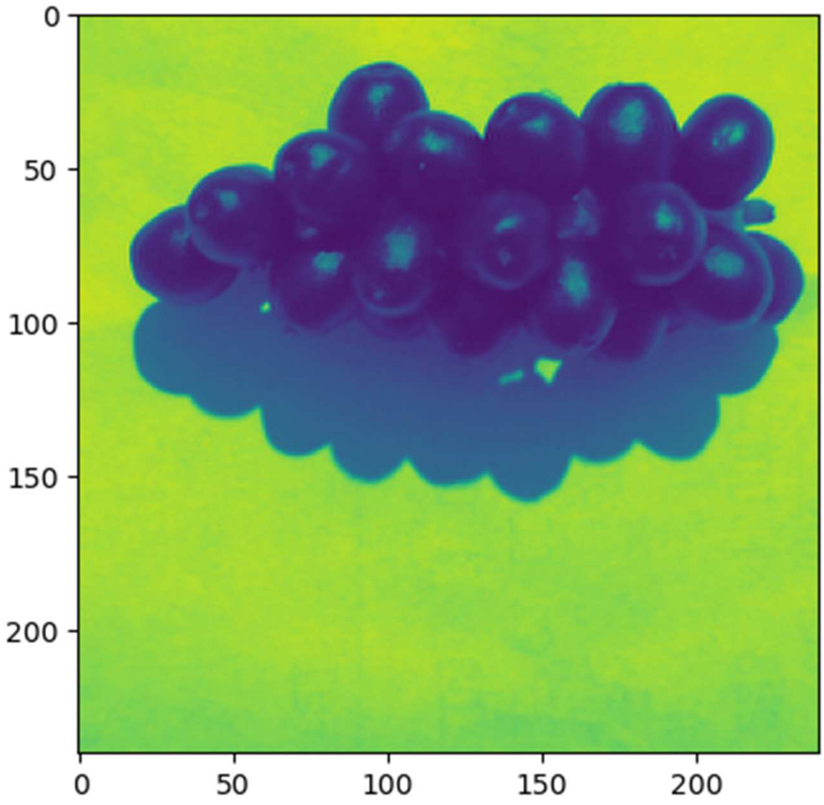
output:



code:

```
#Scaling
#creates a new image
#  that is a copy of the original frequency domain image
# but with the lower right corner of the image removed
freq_image_3 = freq_image[0:240,0:240]
#transforme back to the spatial domain using the inverse 2D DCT
image_2 = sfft.idct((sfft.idct(freq_image_3,norm='ortho')).T,norm='ortho')
plt.imshow(image_2)
plt.show()
```

output:



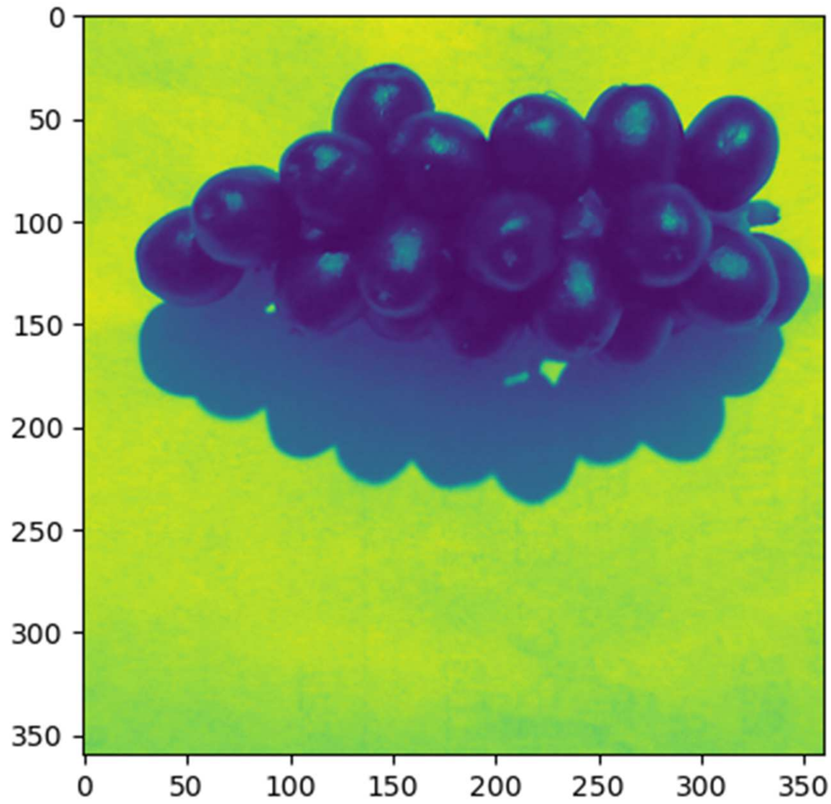
d)

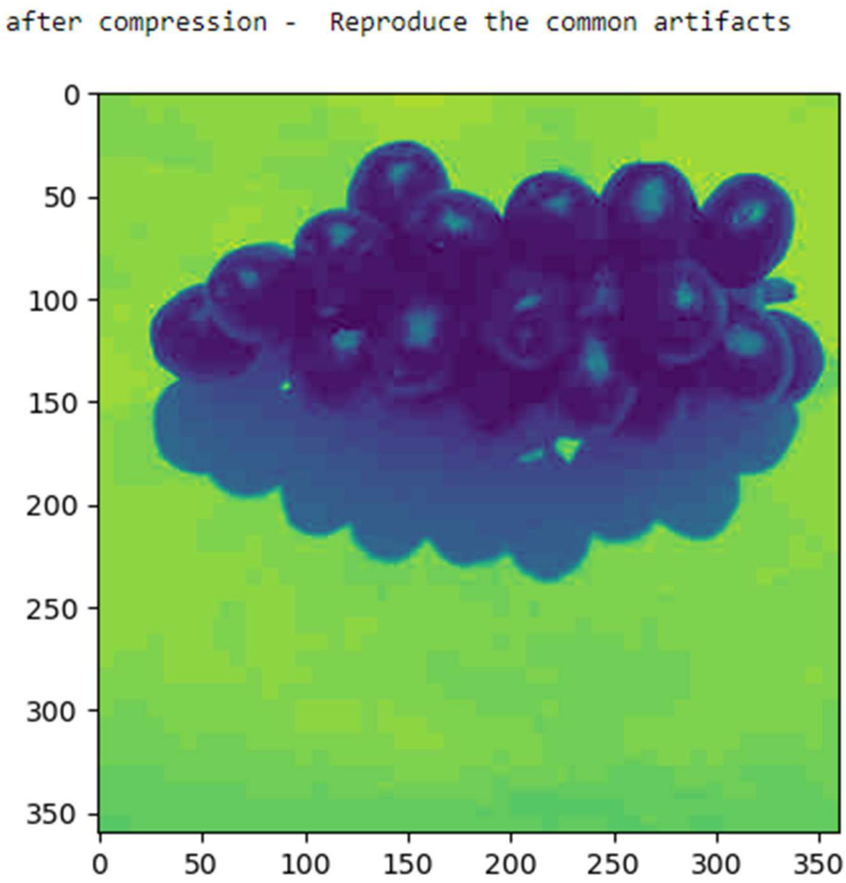
```
from PIL import Image

# load the image
image = Image.open("fruit.jpg")

# Save the image using lossy compression (set for the low quality)
image.save("compressed.jpg", "JPEG", quality=20)

# Convert the image to a NumPy array
image_array = np.array(image)

print("Before compression")
# Plot the image
plt.imshow(image_array)
plt.show()

import matplotlib.pyplot as plt
from PIL import Image

# load the compressed image
image = Image.open("compressed.jpg")

# Convert the image to a NumPy array
image_array = np.array(image)
```

```
print("after compression -  Reproduce the common artifacts")
# Plot the image
plt.imshow(image_array)
plt.show()
```

output:

Before compression





Question 5:

a) and

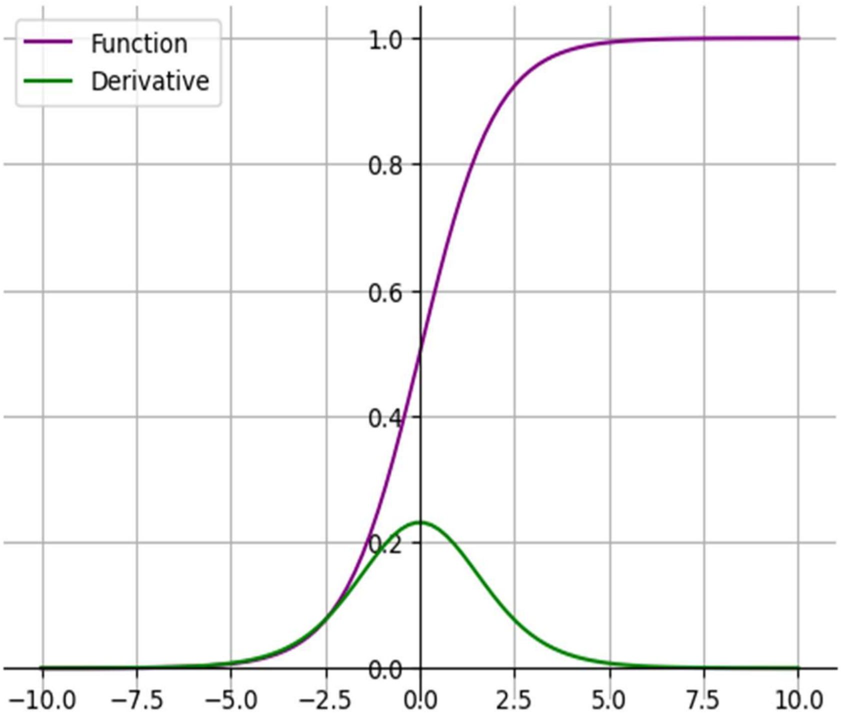
b)

code:

```
pip install sympy

import matplotlib.pyplot as plt
import numpy as np
from scipy.misc import derivative
import sympy as sym
from sympy import pi
import math

# function
def function(x):
  return 1/(1+math.e**(-x))

# function to find the derivative (by x)
def dev_function(x):
  return derivative(function, x)
```

```
# x axis intervals
y = np.linspace(-10,10,100)

# function to format the graph
def format():
   fig = plt.figure()
   ax = fig.add_subplot(1, 1, 1)
   ax.spines['left'].set_position('center')
   ax.spines['bottom'].set_position('zero')
   ax.spines['right'].set_color('none')
   ax.spines['top'].set_color('none')
   ax.xaxis.set_ticks_position('bottom')
   ax.yaxis.set_ticks_position('left')

format()
#plotting the function
plt.plot(y, function(y), color='purple',label='Function')
plt.plot(y, dev_function(y), color='green', label='Derivative')

def format_2():
   # formatting
   plt.legend(loc='upper left')
   plt.grid(True)
format_2()
```

output:



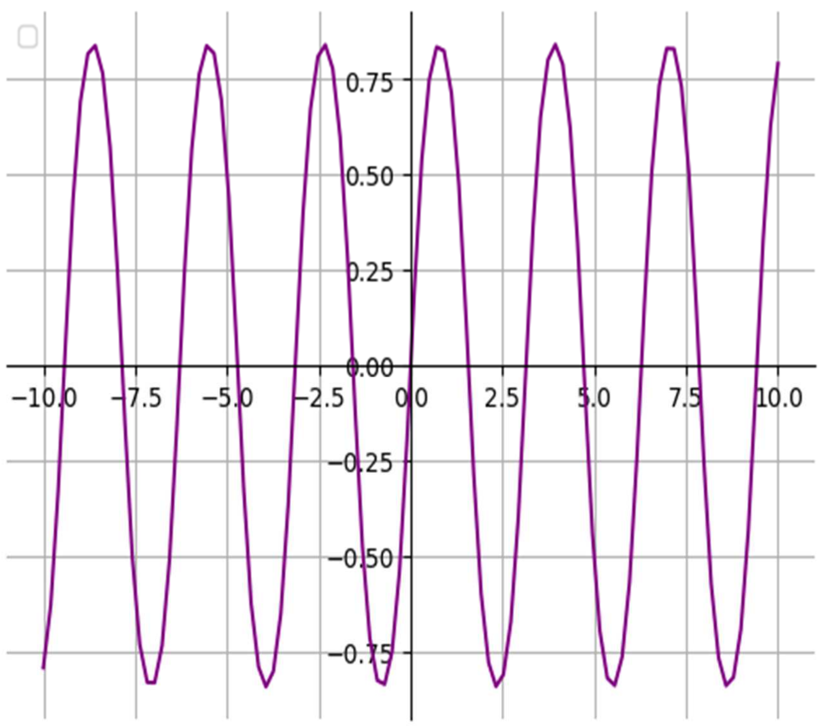
c)

a.

code:

```
def function1(x):
  return np.sin(np.sin(2*x))
format()
#plotting the function
plt.plot(y, function1(y), color='purple')
format_2()
```

output:

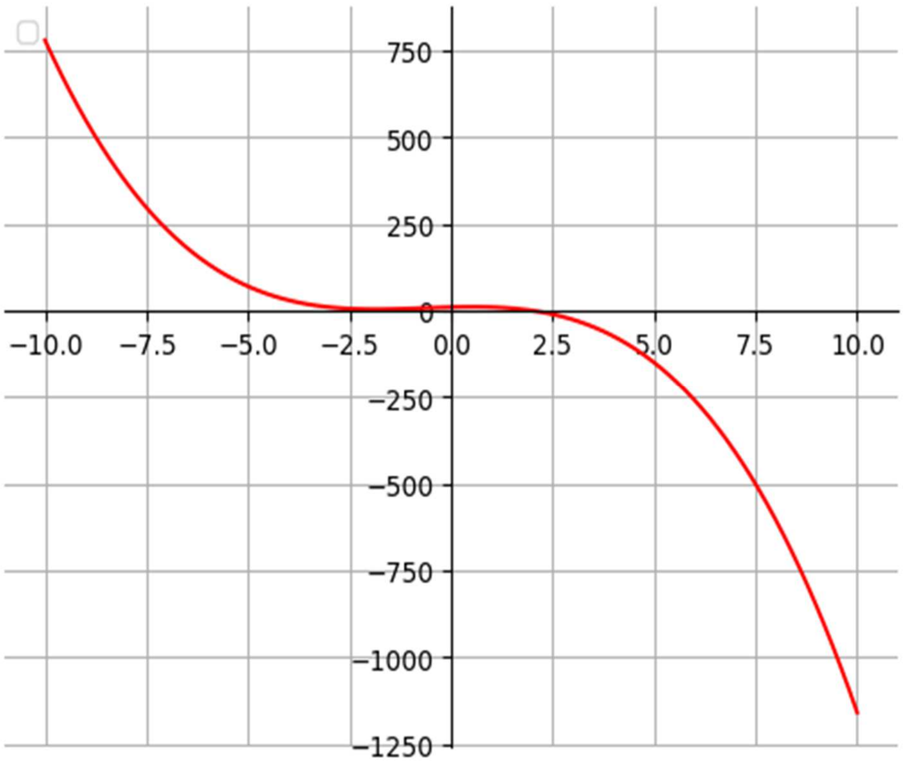


b.

code:

```
def function2(x):
  return -x**3-2*x**2+3*x+10
format()
#plotting the function
plt.plot(y, function2(y), color='red')
format_2()
```

output:

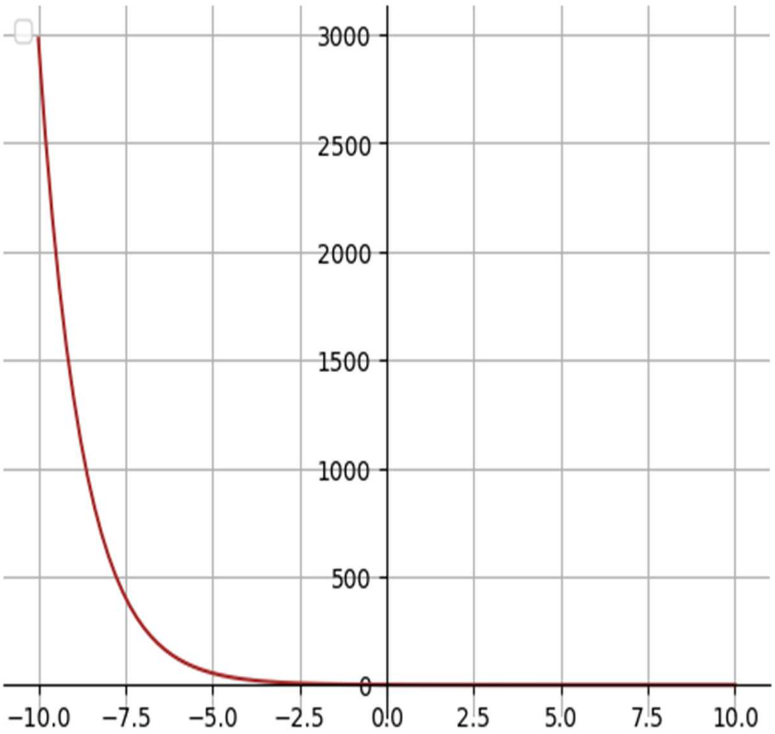


c.

code:

```
def function3(x):
   return np.exp(-0.8*x)
format()
#plotting the function
plt.plot(y, function3(y), color='brown')
format_2()
```

output:

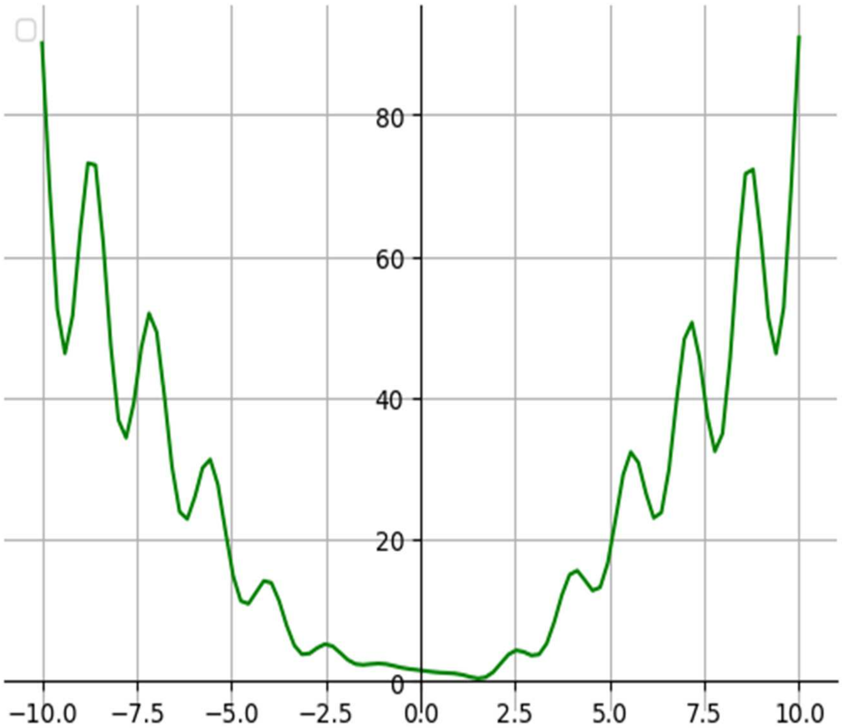


d.

code:

```
def function4(x):
   return x**2*np.cos(np.cos(2*x))-2*np.sin(np.sin(x-math.pi/3))
format()
#plotting the function
plt.plot(y, function4(y), color='green')
format_2()
```

output:

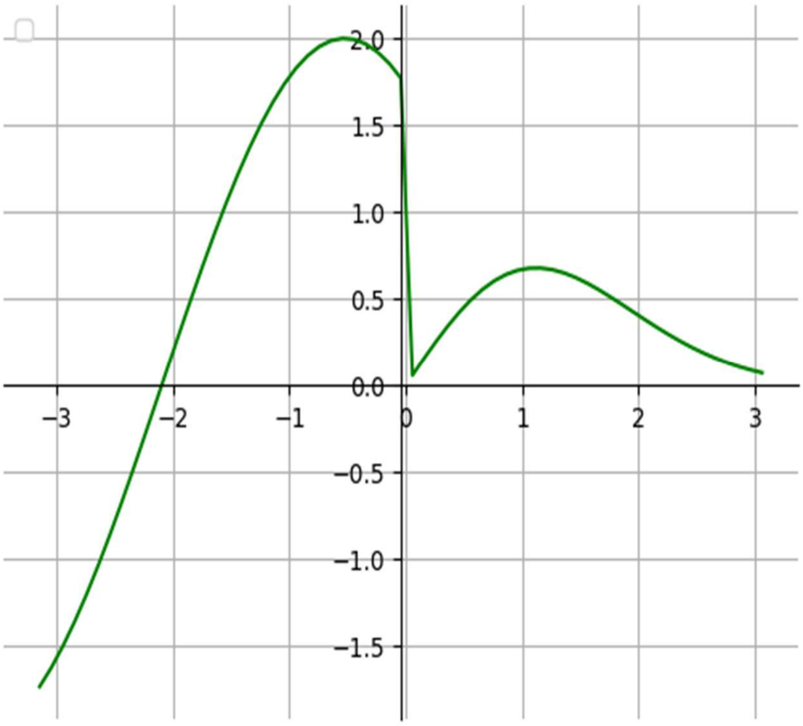


e.

code:

```
# range of x values , step size = 0.1
x1 = np.arange(-np.pi, np.pi, 0.1)

# Calculating y values
y1 = np.where(x1 < 0, 2*np.cos(x1 + np.pi/6), x1*np.exp(-0.4*x1**2))

# Plot the function
format()
plt.plot(x1, y1,color='green')
format_2()
```

output:



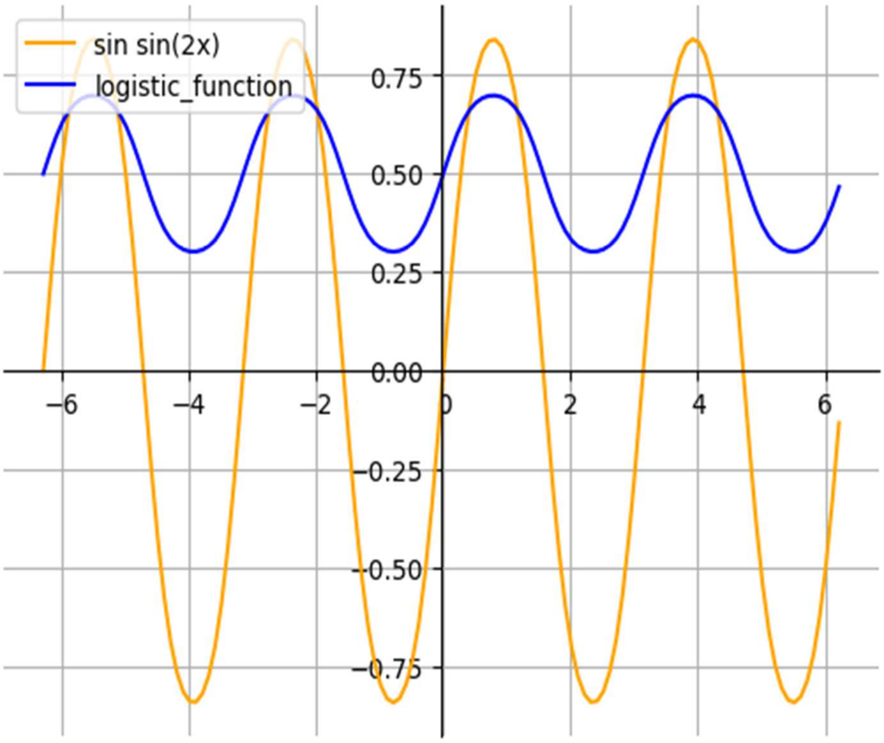
d)

a.

code:

```
# range of x values
x2 = np.arange(-2*np.pi, 2*np.pi, 0.1)

# Calculating the y values
y2 = np.sin(np.sin(2*x2))

# appling logstic function
def logistic_function(y):
  return 1 / (1 + np.exp(-y))

format()
plt.plot(x2, y2, color="orange", label="sin sin(2x)")
plt.plot(x2, logistic_function(y2), color="blue", label="logistic_function")
format_2()
```

output:

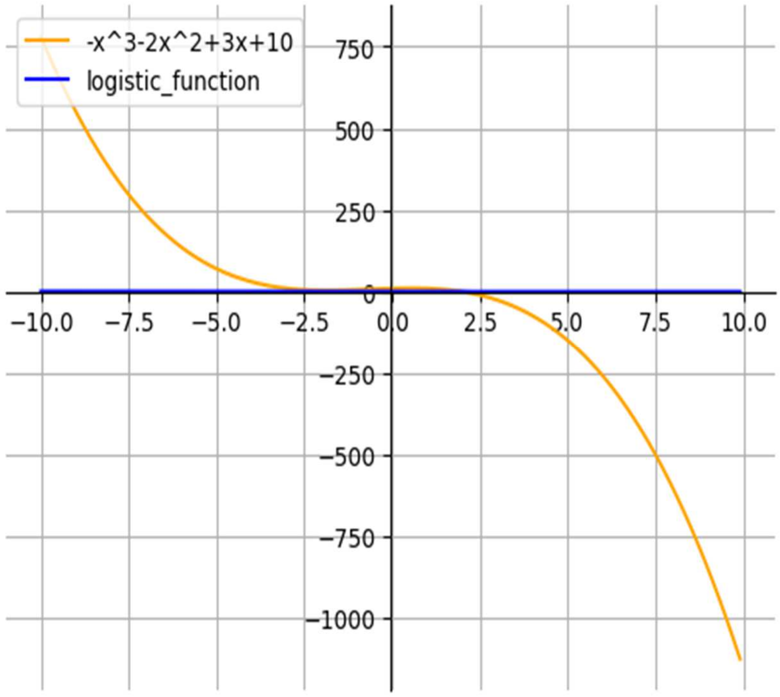


b.

code:

```
# range of x values
x3 = np.arange(-10, 10, 0.1)

# Calculating the y values
y3 = -x3**3 - 2*x3**2 + 3*x3 + 10

format()
plt.plot(x3, y3, color="orange", label="-x^3-2x^2+3x+10")
plt.plot(x3, logistic_function(y3),  color="blue", label="logistic_function")
format_2()
```

output:

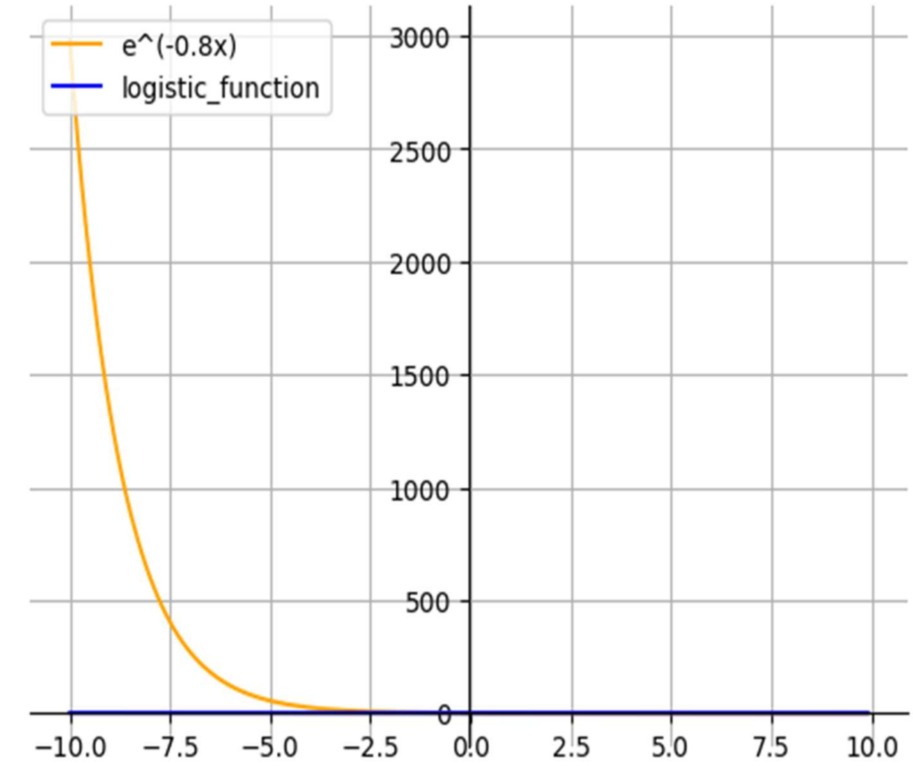


c.

code:

```
# Calculating the y values
y4 = np.exp(-0.8*x3)

format()
plt.plot(x3, y4, color="orange", label="e^(-0.8x)")
plt.plot(x3, logistic_function(y4),color="blue", label="logistic_function")
format_2()
```

output:

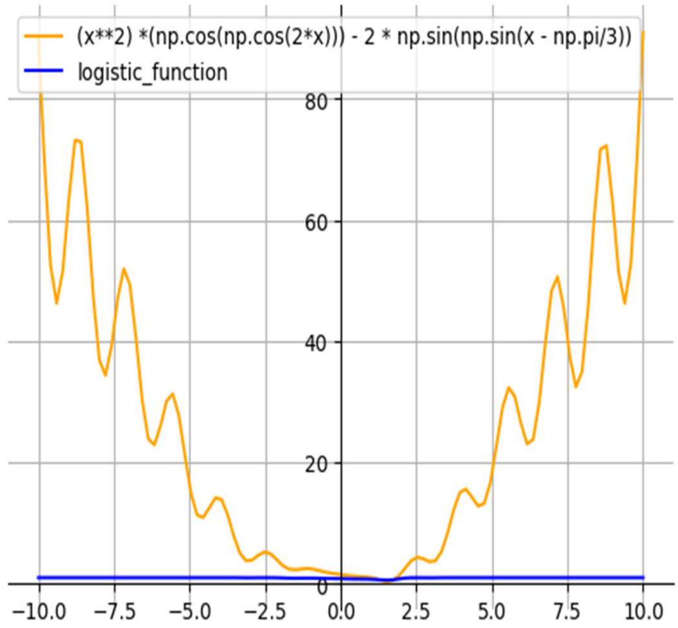


d.

code:

```
x4 = np.linspace(-10, 10, 100)
# Calculating the y values
y5 = (x4**2) *(np.cos(np.cos(2*x4))) - 2 * np.sin(np.sin(x4 - np.pi/3))

format()
plt.plot(x4, y5, color="orange", label="(x**2) *(np.cos(np.cos(2*x))) - 2 *
np.sin(np.sin(x - np.pi/3))")
plt.plot(x4, logistic_function(y5),color="blue", label="logistic_function")
format_2()
```

output:

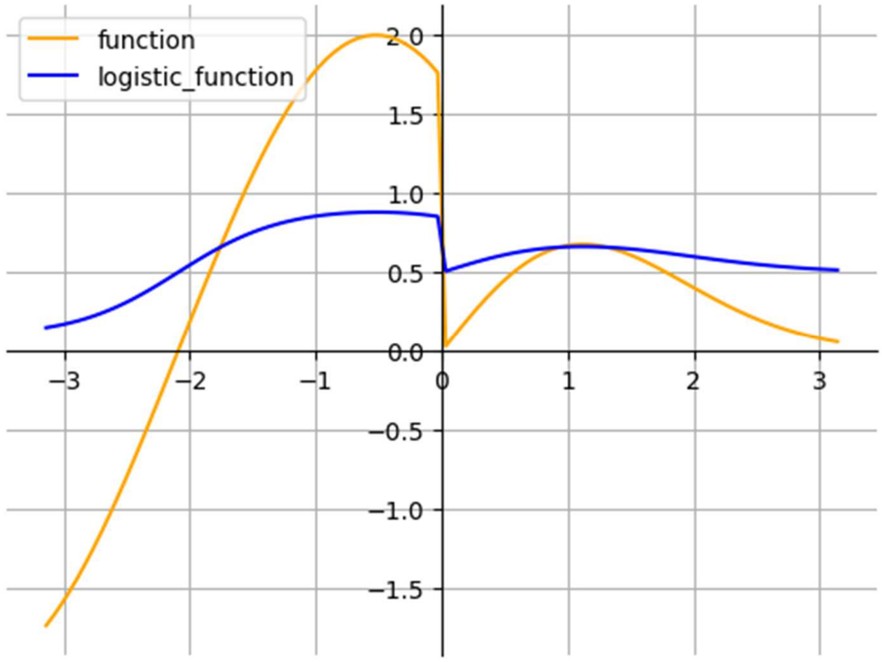


e.

code:

```
#range of x values
x5 = np.linspace(-np.pi, np.pi, 100)

# Calculating the values of g(x) for each x
y6 = np.where(x5 < 0, 2 * np.cos(x5 + np.pi/6), x5 * np.exp(-0.4*x5**2))

format()
plt.plot(x5, y6, color="orange", label="function")
plt.plot(x5, logistic_function(y6),color="blue", label="logistic_function")
format_2()
```

output:


function
logistic_function
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3