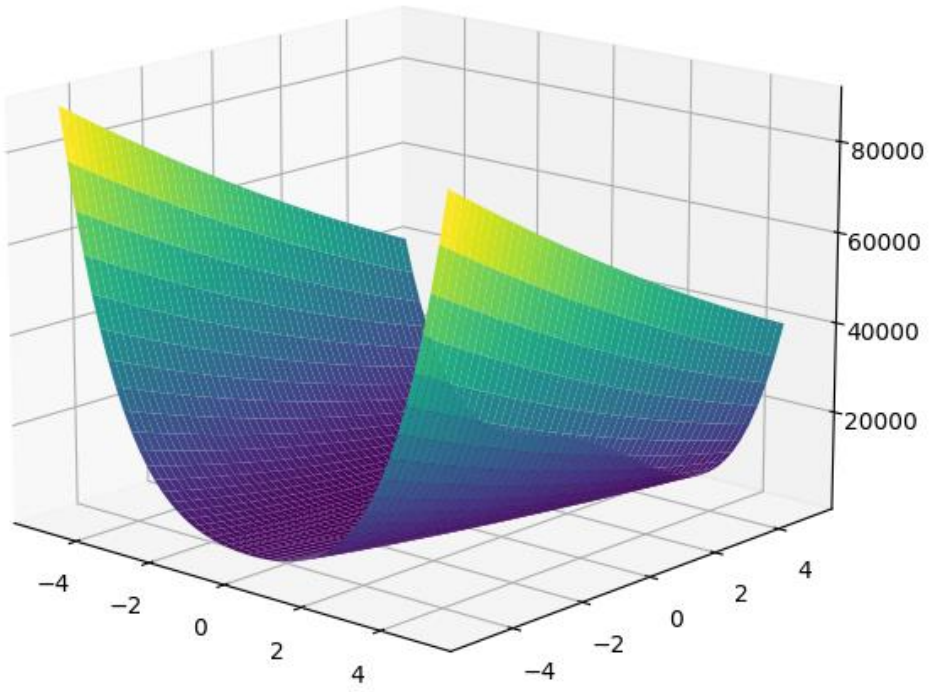
Some answers are written in the notebook submitted separately.

1.

a) Refer to notebook

b) Refer to notebook

```
import numpy as np

import matplotlib.pyplot as plt

def func(x,y):
    return 100*(y-x**2)**2 + (1-x)**2

# create two one-dimensional grids using linspace
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)

# combine the two one-dimensional grids into one two-dimensional grid
X, Y = np.meshgrid(x,y)

# evaluate the function at each element of the two-dimensional grid
Z = func(X, Y)

# create plot
fig = plt.figure(figsize=(7,7))
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
plt.show()
```



c)

k=0, [-1.2 1.]

k=1, [-1.1752809   1.38067416]

k=2, [0.76311487 -3.17503385]

k=3, [0.76342968 0.58282478]

k=4, [0.99999531 0.94402732]

k=5, [0.9999957 0.99999139]

k=6, [1. 1.]

k=7, [1. 1.]

```
import numpy as np

start = np.array([-1.2,1])
v = start

def nab(x, y):
    return np.array([-400*(y-(x**2))*x - 2*(1-x), 200*(y-x**2)])

def hes(x, y):
    return np.array([[1200*(x**2)-400*y+2, -400*x], [-400*x, 200]])

new_nab = nab(v[0], v[1])

print(f'k=0, {v}')

iteration = 1
while not np.linalg.norm(new_nab, ord=2) <= 10**(-6):
    new_nab = nab(v[0], v[1])
    new_hes = hes(v[0], v[1])
    v = v - np.linalg.inv(new_hes) @ new_nab
    print(f'k={iteration}, {v}')
    iteration += 1
```

2.a) Refer to notebook

b) Refer to notebook

c) Refer to notebook

d)

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


df=pd.read_csv('songs.csv', sep=',', header=0)


# I
df.drop(columns=['Artist Name', 'Track Name', 'key', 'mode', 'time_signature',
'instrumentalness'], inplace=True)
```

```
# II
df.drop(df[(df.Class != 5) & (df.Class != 9)].index, inplace=True)
df['Class'].replace([5, 9], [1, 0], inplace=True)


# III
df.dropna(axis=0, inplace=True)


# IV
X_train, X_test, Y_train, Y_test = train_test_split(df.drop(columns='Class'), df['Class'],
test_size=0.3, random_state=23)


# V
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)


# VI
print('first row X_train:',X_train[0][0:3])
print('last row X_train:',X_train[-1][0:3])
print('first row X_test:',X_test[0][0:3])
print('last row X_test:',X_test[-1][0:3])
print('first row Y_train:',Y_train.iloc[0])
print('last row Y_train:',Y_train.iloc[-1])
print('first row Y_test:',Y_test.iloc[0])
print('last row Y_test:',Y_test.iloc[-1])
```

first row X_train: [-0.93555843 0.67519298 1.3849985]

last row X_train: [-1.13301479 -1.09458877 0.96702449]

first row X_test: [-0.29382524 1.36005105 0.26306826]

last row X_test: [-0.29382524 -1.05390413 -1.34833155]

first row Y_train: 0

last row Y_train: 1

first row Y_test: 0

last row Y_test: 1

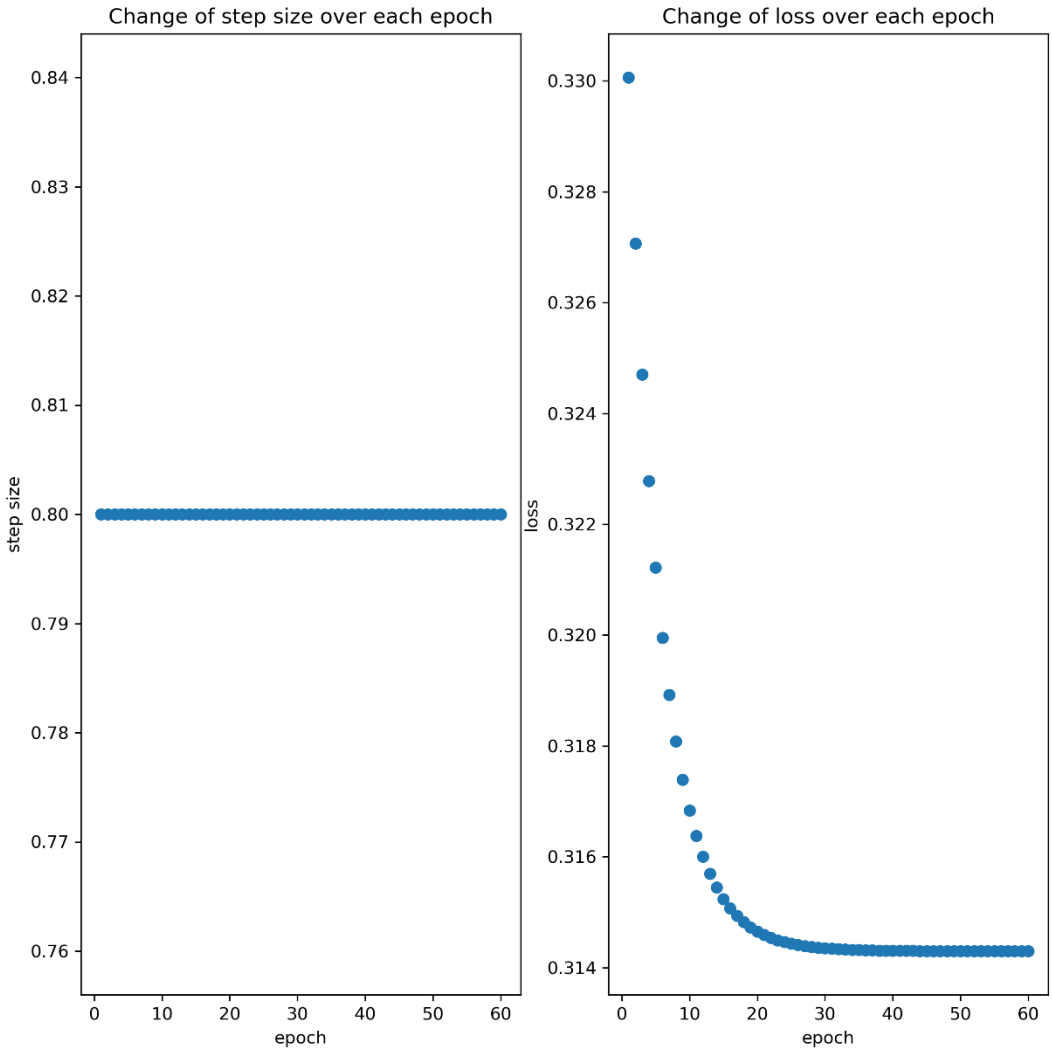
e)

final train loss: 0.3142969702921616

test loss: 0.31750699444279507

```
import numpy as np
from sklearn.metrics import log_loss
import matplotlib.pyplot as plt
data = __import__('2d')

def sigmoid(x):
    # logistic sigmoid
    return np.exp(-np.logaddexp(0, -x))

def loss(gamma, X, y, lam):
    # gamma has first coordinate = beta0 = intercept, and second coordinate = beta
    norm_beta_sq = np.linalg.norm(gamma[1:], ord=2)**2
    z = np.dot(X, gamma[1:]) + gamma[0]
    sig_z = sigmoid(z)
    return lam * log_loss(y, sig_z, normalize=True) + 0.5 * norm_beta_sq

def nab_loss(gamma, X, y, lam):
    n = X.shape[0]

    summ = np.zeros(p+1)
    for i in range(0,n):
        summ += (y[i] - sigmoid(np.dot(gamma, np.insert(X[i], 0, 1)))) * np.insert(X[i], 0, 1)

    return np.insert(gamma[1:], 0, 0) - (lam / n) * summ

p = data.X_train.shape[1]
gamma = np.zeros(p + 1)
lam = 0.5
alpha = 1
a = 0.5
b = 0.8

epochs_lim = 60
epochs = np.arange(1, epochs_lim+1)
step_sizes = np.full(epochs_lim, -1.0)
losses = np.full(epochs_lim, -1.0)

for ep in range(1, epochs_lim+1):
    cur_nab_loss = nab_loss(gamma, data.X_train, data.Y_train, lam)
    cur_loss = loss(gamma, data.X_train, data.Y_train, lam)

    if loss(gamma - alpha * cur_nab_loss, data.X_train, data.Y_train, lam) > cur_loss - a *
alpha * np.linalg.norm(cur_nab_loss, ord=2)**2:
        alpha = alpha * b

    step_sizes[ep-1] = alpha
```

```
    # Update equation
    gamma = gamma - alpha * cur_nab_loss

    losses[ep-1] = loss(gamma, data.X_train, data.Y_train, lam)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10,10))

def plot(x, y, xlabel, ylabel, title, loc):
    axes[loc].scatter(x, y)
    axes[loc].set_title(title)
    axes[loc].set_xlabel(xlabel)
    axes[loc].set_ylabel(ylabel)

plot(epochs, step_sizes, 'epoch', 'step size', 'Change of step size over each epoch', 0)
plot(epochs, losses, 'epoch', 'losses', 'Change of losses over each epoch', 1)

print('final train loss:', losses[-1])
print('test loss:', loss(gamma, data.X_test, data.Y_test, lam))

plt.savefig("2e.png", dpi=300)
plt.show()
```


Change of step size over each epoch
step size
epoch
0.84
0.83
0.82
0.81
0.80
0.79
0.78
0.77
0.76
0
10
20
30
40
50
60
Change of loss over each epoch
loss
epoch
0.330
0.328
0.326
0.324
0.322
0.320
0.318
0.316
0.314

f) I'm unable to make this work. The inverse of hessian keeps blowing up to inf in numpy when doing update for newton's method. Code is given below. If you set epochs_lim = 3, you will see that log loss of train (and test) data blows very quickly to inf.

```
import numpy as np
from sklearn.metrics import log_loss
import matplotlib.pyplot as plt
data = __import__('2d')


def sigmoid(x):
    # logistic sigmoid
    return np.exp(-np.logaddexp(0, -x))


def loss(gamma, X, y, lam):
    # gamma has first coordinate = beta0 = intercept, and second coordinate = beta
    norm_beta_sq = np.linalg.norm(gamma[1:], ord=2)**2
    z = np.dot(X, gamma[1:]) + gamma[0]
    sig_z = sigmoid(z)
    return lam * log_loss(y, sig_z, normalize=True) + 0.5 * norm_beta_sq


def nab_loss(gamma, X, y, lam):
    n = X.shape[0]

    summ = np.zeros(p+1)
    for i in range(0,n):
        summ += (y[i] - sigmoid(np.dot(gamma, np.insert(X[i], 0, 1)))) * np.insert(X[i], 0, 1)

    return np.insert(gamma[1:], 0, 0) - (lam / n) * summ


def hes_loss(gamma, X, y, lam):
    n = X.shape[0]

    summ = np.zeros((p+1, p+1))
    for i in range(0,n):
        # appending the extra 1 as first element of each x datapoint
        xi = np.insert(X[i], 0, 1)
        summ += sigmoid(np.dot(gamma, xi)) * (1 - sigmoid(np.dot(gamma, xi))) * np.outer(xi,
np.transpose(xi))

    return (lam / n) * summ


p = data.X_train.shape[1]
gd_gamma = np.zeros(p + 1)
newt_gamma = np.zeros(p + 1)
lam = 0.5
alpha = 1
a = 0.5
b = 0.8
```

```
epochs_lim = 60
epochs = np.arange(1, epochs_lim+1)
step_sizes = np.full(epochs_lim, -1.0)
gd_losses = np.full(epochs_lim, -1.0)
newt_losses = np.full(epochs_lim, -1.0)


# Log regression with gradient descent
for ep in range(1, epochs_lim+1):
    cur_nab_loss = nab_loss(gd_gamma, data.X_train, data.Y_train, lam)
    cur_loss = loss(gd_gamma, data.X_train, data.Y_train, lam)

    if loss(gd_gamma - alpha * cur_nab_loss, data.X_train, data.Y_train, lam) > cur_loss - a *
alpha * np.linalg.norm(cur_nab_loss, ord=2)**2:
        alpha = alpha * b

    step_sizes[ep-1] = alpha

    # Update equation
    gd_gamma = gd_gamma - alpha * cur_nab_loss

    gd_losses[ep-1] = loss(gd_gamma, data.X_train, data.Y_train, lam)


# Log regression with newton's method
for ep in range(1, epochs_lim+1):
    cur_nab_loss = nab_loss(newt_gamma, data.X_train, data.Y_train, lam)
    cur_loss = loss(newt_gamma, data.X_train, data.Y_train, lam)

    if loss(newt_gamma - alpha * cur_nab_loss, data.X_train, data.Y_train, lam) > cur_loss - a
* alpha * np.linalg.norm(cur_nab_loss, ord=2)**2:
        alpha = alpha * b

    step_sizes[ep-1] = alpha

    # Update equation
    newt_gamma = newt_gamma - alpha * (np.linalg.inv(hes_loss(newt_gamma, data.X_train,
data.Y_train, lam)) @ cur_nab_loss)

    newt_losses[ep-1] = loss(newt_gamma, data.X_train, data.Y_train, lam)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10,10))

axes[0].scatter(epochs, step_sizes)
axes[0].set_xlabel('epoch')
axes[0].set_ylabel('step size')
axes[0].set_title('Change of step size over each epoch')

axes[1].scatter(epochs, gd_losses, color='red', label='gradient descent')
axes[1].scatter(epochs, newt_losses, color='blue', label='newton\'s method')
axes[1].set_xlabel('epoch')
axes[1].set_ylabel('loss')
axes[1].set_title('Comparing GD and newtown\'s method on log-loss')
```

```
axes[1].legend(loc='upper right')

print('final train loss:', loss(newt_gamma, data.X_train, data.Y_train, lam))
print('test loss:', loss(newt_gamma, data.X_test, data.Y_test, lam))

plt.savefig("2f.png", dpi=300)
plt.show()
```

g)

Although Newton's method converges much quicker than GD, it does have some disadvantages that makes it not as popular in machine learning. The main disadvantage is that it requires computing the second derivative of the function we want to minimise. The second derivative may often be intractable, or comes with a very high computation cost. The higher computation cost is because whilst computing the first derivative has p values (p = the number of features), the second order derivative has p^2 values.

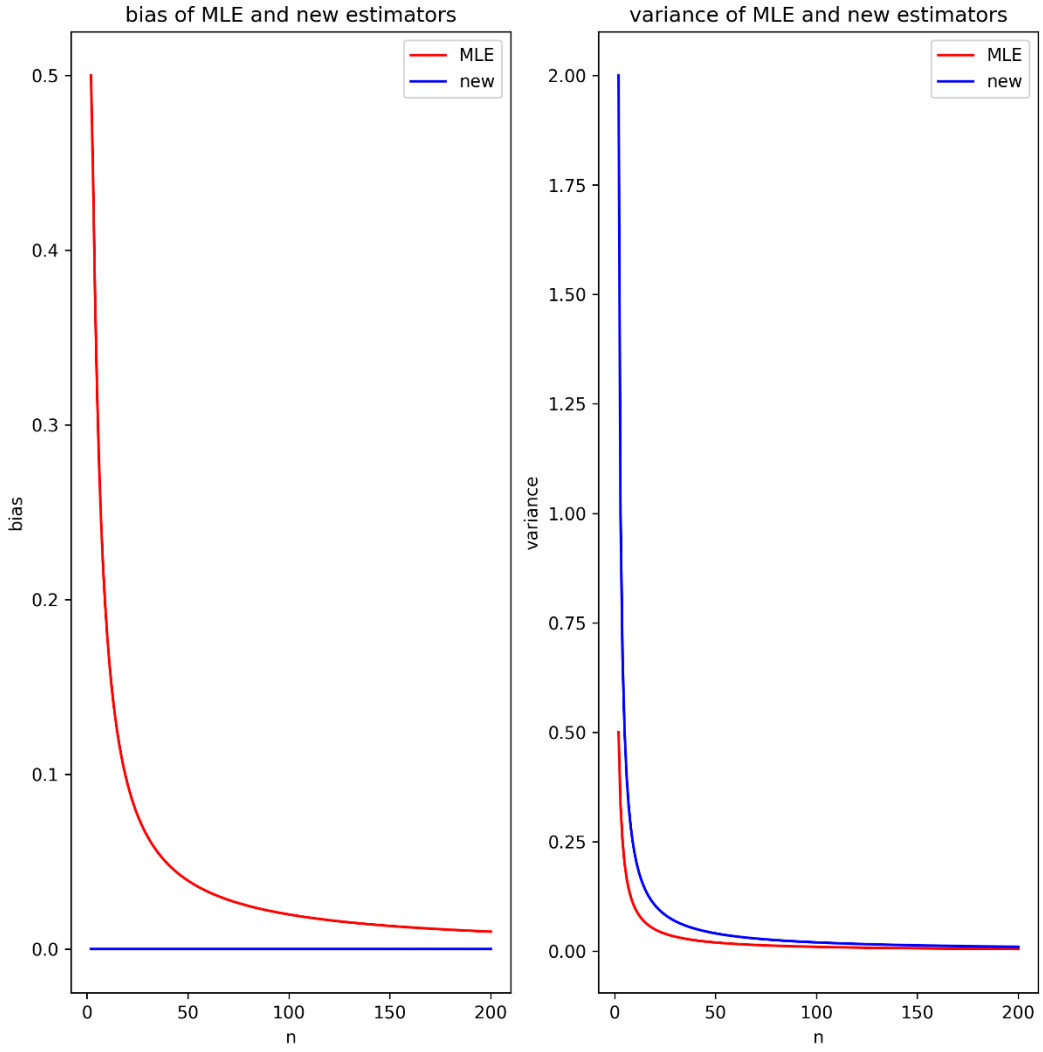
3.a) Refer to notebook

b) Refer to notebook

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

SD = 1

def MLE_bias_estimator(n):
    return (2*SD**4)*(n-1)/n**2

def MLE_var_estimator(n):
    return SD**2/n

def new_bias_estimator(n):
    return 0

def new_var_estimator(n):
    return (2*SD**4)/(n-1)

MLE_bias_estimator = np.vectorize(MLE_bias_estimator)
MLE_var_estimator = np.vectorize(MLE_var_estimator)
new_bias_estimator = np.vectorize(new_bias_estimator)
new_var_estimator = np.vectorize(new_var_estimator)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10,10))

fig.suptitle('Comparing bias and variance of MLE and new estimator')

def plot(xrange, func1, func2, ylabel, title, x):
    axes[x].plot(xrange, func1(xrange), color='red', label='MLE')
```

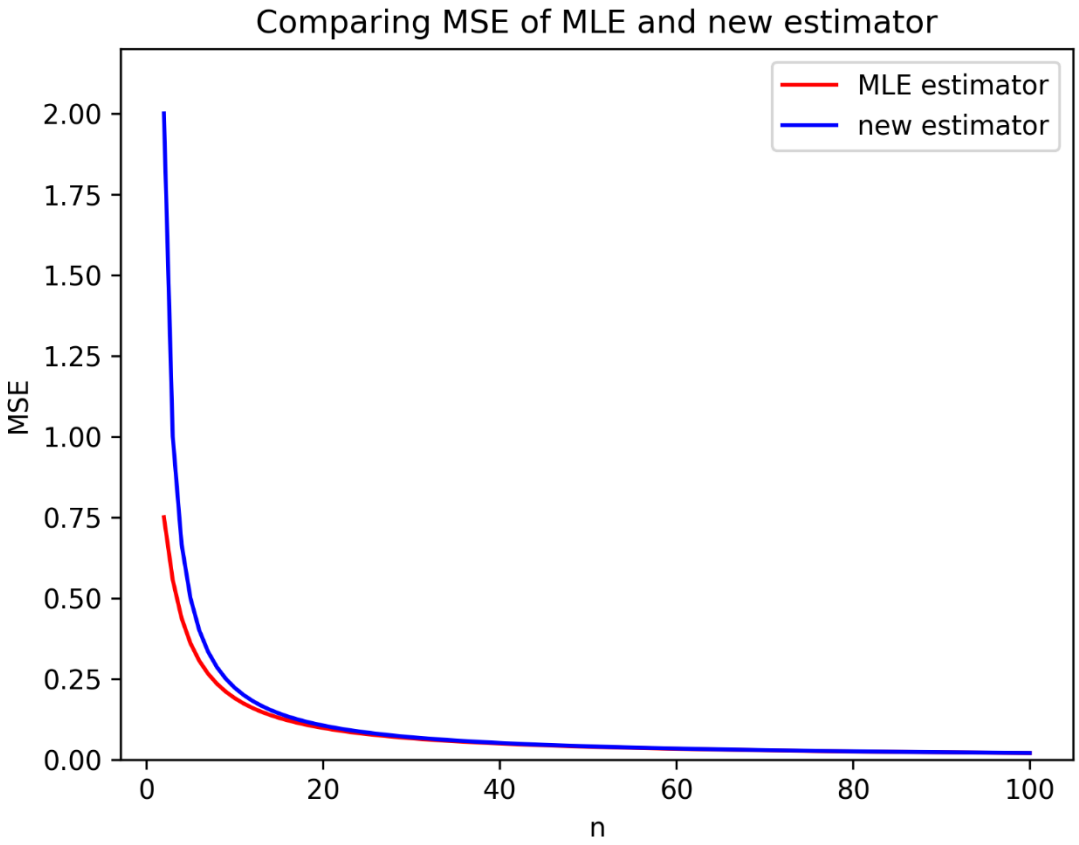
```
    axes[x].plot(xrange, func2(xrange), color='blue', label='new')
    axes[x].set_title(title)
    axes[x].set_xlabel('n')
    axes[x].set_ylabel(ylabel)
    axes[x].legend(loc='upper right')

xrange = np.arange(2, 201, dtype=np.longdouble)

plot(xrange, MLE_bias_estimator, new_bias_estimator, 'bias', 'bias of MLE and new estimators',
0)
plot(xrange, MLE_var_estimator, new_var_estimator, 'variance', 'variance of MLE and new
estimators', 1)

plt.savefig("3b.png", dpi=300)
plt.show()
```

Comparing bias and variance of MLE and new estimator



c) Refer to notebook

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

SD = 1

def MLE_estimator_MSE(n):
    return (2*SD**4/n) - (SD**4/n**2)

def new_estimator_MSE(n):
    return (2*SD**4) / (n-1)

MLE_estimator_MSE = np.vectorize(MLE_estimator_MSE)
new_estimator_MSE = np.vectorize(new_estimator_MSE)

xrange = np.arange(2, 101, dtype=np.longdouble)

plt.plot(xrange, MLE_estimator_MSE(xrange), color='red', label='MLE estimator')
plt.plot(xrange, new_estimator_MSE(xrange), color='blue', label='new estimator')

plt.title('Comparing MSE of MLE and new estimator')
plt.legend(loc='upper right')
plt.xlabel('n')
plt.ylabel('MSE')
plt.ylim(0, 2.2)

plt.savefig("3c.png", dpi=300)
plt.show()
```


Comparing MSE of MLE and new estimator
MLE estimator
new estimator
MSE
n
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20
40
60
80
100