March 2022

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

MATHS ASSIGNMENT

(Question - 1)

a) Ans:- Given, A is a symmetric matrix

$$B = A^T A = A^2 \quad \ldots\ldots (i)$$

Considering $a\vec{x}$ as an eigen vector which isn't equal to 0

$$\lambda \vec{x} \neq 0$$

$$A\vec{x} = \lambda\vec{x} \quad \ldots\ldots (ii)$$

need λ^2 in a Eigen vector

$\therefore$ Square $A^2\vec{x}$

$A(A\vec{x}) = A(\lambda\vec{x}) = \lambda(A\vec{x})$ from eqn (ii)

$\lambda(\lambda\vec{x}) = \lambda^2\vec{x}$

So λ^2 is a Eigen value of A^2

But we eqn (i) $A^2 = B$

$\therefore \lambda$ is a eigen value of B

b) Ans:- Considering an eg. $A = \begin{bmatrix} 2 & 3 & 6 \\ 3 & 4 & 5 \\ 6 & 5 & 9 \end{bmatrix}$

a Symmetric matrix

$$B = A^T A = \begin{bmatrix} 2 & 3 & 6 \\ 3 & 4 & 5 \\ 6 & 5 & 9 \end{bmatrix} \begin{bmatrix} 2 & 3 & 6 \\ 3 & 4 & 5 \\ 6 & 5 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 4+9+36 & 6+12+30 & 12+15+54 \\ 6+12+30 & 9+16+25 & 18+20+45 \\ 12+15+54 & 18+20+45 & 36+25+81 \end{bmatrix}$$

$$= \begin{bmatrix} 49 & 48 & 81 \\ 48 & 50 & 89 \\ 81 & 89 & 142 \end{bmatrix}$$

i.e B, so if u transpose this vector, u get the same vector, so we can say

B is a symmetric matrix.

c) To construct a matrix C that has the same eigenvalues as A and satisfies $C^TC = A$, we can use the eigenvalue decomposition of A, $A = U^TAU$.

~~Let $C = U^T\sqrt{A}$~~. Let $C = U^T\sqrt{\Lambda}U$, where $\sqrt{\Lambda}$ is the matrix obtained by taking the

square root of each element of Λ.

then we have:

$C^TC = (U\sqrt{\Lambda}U)^T(U\sqrt{\Lambda}U) = U\sqrt{\Lambda}^T U^T U\sqrt{\Lambda}U$

$= U\Lambda U^T = A$

Thus, C has the desired property $C^TC = A$.

To show that C is symmetric, we need to show that $C^T = C$. Substituting the definition of C, we get:

$C^T = (U\sqrt{\Lambda}U)^T = U\sqrt{\Lambda}^T U^T = U\sqrt{\Lambda}U = C$

Therefore, C is a symmetric matrix.

Since C has the same eigenvalues as A and is symmetric, we can use C to define the "square root" of a matrix A as $A^{\frac{1}{2}} = C$. This definition is motivated by the fact that if A and C are both symmetric and $C^TC = A$, then C can be thought of as the square root of A, in the same way that the square root of a number a is a number b such that $b^2 = a$.

(Question - 2)

② Given the function $f(x,y) = (x-1)^4 + (y-4)^2$

a) The gradient of any function is equal to the vector with its partial derivatives i.e.

$$\nabla f(x,y) = \begin{bmatrix} \frac{df}{dx} \\ \frac{df}{dy} \end{bmatrix}$$

Finding the partial derivatives

$$\frac{df}{dx} = 4(x-1)^3 \cdot (1) + 0$$

$$\frac{df}{dx} = 4(x-1)^3$$

$$\frac{df}{dy} = 0 + 2 \cdot (y-4)^1 \cdot (1)$$

$$\frac{df}{dy} = 2(y-4)$$

$$Df(x,y) = \begin{bmatrix} 4(x-1)^3 \\ 2(y-4) \end{bmatrix}$$

b) The Critical values are,

Ans $\frac{df}{dx} = 0$, $\frac{df}{dy} = 0$

i.e. $4(x-1)^3 = 0$ $\quad 2(y-4) = 0$

$(x-1)^3 = 0$ $\quad y - 4 = 0$

$x^3 - 3x(x-1) - 1^3 = 0$ $\quad y = 4$

$x^3 - 3x^2 + 3x - 1 = 0$

$x(x^2 - 3x + 3) = 1$

$x^2 - 3x + 3 = 1$

$x^2 - 3x + 2 = 0$

$(x-2)(x-1) = 0$

$x = 2, x = 1$

(Question-3)

a) $f(x) = 2\cos(x) + x^3 + e^x$

Ans $G(y) = \int_0^y (2\cos x + x^3 + e^x)\,dx$

$= \left[2\sin x + \frac{x^4}{4} + e^x\right]_0^y$

$= \left[2\sin y + \frac{y^4}{4} + e^y\right] - \left[2\sin(0) + \frac{0}{4} + e^0\right]$

$G(y) = 2\sin y + \frac{y^4}{4} + e^y - 1$ Ans

b) Need to prove now that the ~~not~~ differentiation of the answer we got gets back the ~~equation~~ question or not

$$G(x) = f(x)$$

$$G'(x) = \frac{d}{dx}\left(2\sin x + \frac{x^4}{4} + e^x - 1\right)$$

$$= 2\cos x + \frac{1}{4}(4x^3) + e^x - 0$$

$$= 2\cos x + x^3 + e^x$$

$\therefore G'(x) = f(x)$ [PROVED]

(Question - 5)

5a) All possible combinations of the Avg values of BR and BC make up the result space (sample space) for this problem. Since there are three boxes BR can have one of three potential

values (B1, B2 or B3) and BC can also have one of three possible values (B1, B2 or B3). The 9 potential outcomes are therefore represented by the BR and BC combinations in the following ways:

(B1, B1),
(B1, B2),
(B1, B3),
(B2, B1),
(B2, B2),
(B2, B3).
(B3, B1),
(B3, B2),
(B3, B3).

These scenarios all have an equal chance of occurring and each one reflects a potential pairing of the values of BR & BC. For instance, result (B2, B3) depicts the scenario where the box containing the money is B2 and the selected box is B3 and it has the same likelihood of occurring as any of the other possibilities. It's vital to remember that all the possible combinations of BR and BC are included in the sample space ~~whether or not~~ regardless of whether they are mutually exclusive

(i.e. cannot occur at the same time).
Due to the possibility of the same box being
both the box with the money and the
chosen box in this situation the values of
BR and BC are not mutually exclusive. As
a result, some possibilities in the outcome
space, such as (B1, B1) or (B2, B2) which
depict scenarios in which the box containing
the money is also the chosen box, are
not possible. But for the result space
to accurately reflect all potential pairings
of BR and BC, all possible outcomes are
presented.

5b) All potential outcomes as well as
their probabilities must be taken
into account to determine the expected
reward. There is only one box containing
money therefore picking it has a ~~chance~~
of probability of 1/3. While choosing one
of the empty box has a probability of 2/3.
∴ Expected reward:
$E[R] = (1/3) * £X + (2/3) * 0 = £X/3$.
Now we will be calculating variance of
the reward by evaluating
the squared difference between each

April 2022

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

outcome and the expected reward, and sum these differences over all outcomes and then divide by the total no. of outcomes

~~var(R) = [((Y3) * EX - (EX/3)]^2 + [(2/3) * 0 - (EX/3)^2]~~ =

$$\text{var}(R) = [((Y3) * \pounds X - (\pounds X/3))^2 + ((2/3) * 0 - (\pounds X/3))^2]/2$$

$$= [(\pounds X/3)^2 + (0 - (\pounds X/3))^2]/2$$

$$= (\Sigma X/3)^2/2$$

$$= (\pounds X)^2/18$$

The variance of the reward is $(\pounds X^2)/18$.

5c) Using 'stick' strategy the expected value of the reward is $\pounds X/3$, since probability of choosing the box with the money initially is $1/3$. The variance of the reward is $\pounds X^2/9$, since variance of a Bernoulli random variable (such as the outcome of picking one of the boxes) is $p(1-p)$, where p is the probability of success. In this case p is equal to $1/3$.

Using 'swap' strategy the expected value

of the reward is £0, since the expected value of the reward from the initially selected box is £X/3, the same as the 'stick' strategy. The expected value of the reward from the box ~~of her~~ that swap's is also £X/3 since it has a 1/3 probability of having the money. The variance of the reward with the 'swap' technique, on the other hand, is 0 since the variations of the reward from the initially selected box and the one switch to cancel out because they are both £X²/9. This is due to the fact that the variance of the sum of two random variables equals the total of their variances, and these two random variables have opposite signs (one is positive and the other one is negative). As a result, the variance of the sum is zero.

5)a) In this case, the expected value of
Ans:- the reward for the 'swap' strategy is N/k times the reward amount because the selected box has a N/K probability of being a reward box. The variance of the reward for the

'swap' strategy is $N(N-k)/k^2$ times the reward amount squared. This is because the variance of a Bernoulli random variable (the reward) is $p(1-p)$, where p is the probability of success - (here, N/k).

(Question - 6)

a) In the absence of predators, the population of prey will increase exponentially at a pace determined by the birth rate a. Because there is no factor limiting the prey population, it will continue to increase until it is constrained by other factors like resources or competition. This is sensible behaviour because it is consistent with how populations of creatures frequently behave in the lack of predators or other limiting forces.

Ans $N(t) = aN(t)(1 - N(t)K)$ where $K > 1$ is a parameter

b) The change in the birth rate inserts a density-dependent factor into the model, which implies that as prey population density of prey increases,

the pace at which population grows decreases. This is a more realistic picture of population dynamics, as resources and space are often restricted in nature and cannot sustain endless population increase.

The maximum value that the prey population could take after starting from an initial population of 1 is K. This is because as the population density approaches K, the growth rate of the population approaches zero. The environment's carrying capacity for the population of prey can be interpreted as the parameter K. It stands for the largest population size that the ecosystem can tolerate.

c) P is equal to $\gamma 2/\gamma 1$. It represents the relative growth rate of species 2 compared to species 1. If value of P is greater than 1, then species 2 has a higher growth rate than species 1. If value of P is less than 1 it signifies species 1 has a higher growth rate

two species 2. If ρ is equal to 1, then it implies that both the species have same growth rate.

g) The Jacobian matrix for this system of differential eqns is given by:

Ans:-

$$J = [[f'(v1), f'(v2)], [g'(v1), g'(v2)]]$$

where $f(v_1)$ and $g(v_2)$ are the derivatives of v_1 and v_2 respectively with respect to t.

For calculating Jacobian matrix first we need to find derivatives of v_1 and v_2 with respect to t. From given differential eqns, we have:

~~$v1'(t) = v1(t) * (1 - v1(t) - a12 * v2(t))$~~

$$v_1'(t) = v1(t) * (1 - v1(t) - a12 * v2(t)) v2'(t)$$

$$= \underset{(rho)}{\rho} * v2(t) * (1 - v2(t) - a21 * v_1(t))$$

then, we can plug these expressions

into the formula for the Jacobian matrix to obtain:

$$J = [[1 - 2 * u1(t) - a12 * u2(t), -a12 * u1(t)],$$

$$[-a21 * u2(t), f - \underset{(\partial u_2)}{\partial} * u2(t) - a21 * \underset{(u_1)}{u_1(t)}]]$$

This is the Jacobian matrix for the given system of differential equations.

b) The Jacobian matrix for the first fixed point is:

[[-0.9, 1.1]: [1.1, -0.9]]

The trace of this matrix is -1.8 and the determinant is -0.99.

The Jacobian matrix for the second fixed point is:

[[-1.1, 0.9], [0.9, -1.1]]

The trace of this matrix is -2.2 and the determinant is -0.99.

The product of the eigenvalues (i.e.

the determinant of the Jacobian matrix) and the sum of the eigenvalues (i.e., the trace of the Jacobian matrix) both need to be negative for both fixed points to be stable. We can see from the calculations above that both of these requirements are met. Both fixed points are therefore stable.

The stability requirement is that the eigenvalue product (determinant) and the sum of the eigenvalues (trace) ~~which together make up the~~ both need to be negative 0. Thus, a12 and a21 both must be less than 1. If either of them is greater than 1, then one of the eigenvalues will be positive,

10 Sunday and the fixed point will be unstable.

MATHS ASSIGNMENT CODING QUESTIONS

QUESTION 1 D ANSWER

This is an algorithm which given a symmetric matrix A , builds its corresponding C .

In [1]:
```
import numpy as np
def Build_C(A):
    eigenvalues, eigenvectors = np.linalg.eig(A) # Computes the eigenvalue decomposition of A
    sqrt_Lambda = np.sqrt(np.clip(np.diag(eigenvalues), 0, np.inf)) # Computes the matrix sqrt_Lambda by taking the square root of each element of Lambda
    C = eigenvectors.T @ sqrt_Lambda @ eigenvectors # Setting C = U.T @ sqrt_Lambda @ U
    return C
```

In [2]:
```
# Testing the algorithm using an example.
A = np.array([[2, 3, 6], [3, 4, 5],[6, 5, 9]]) # symmetric matrix A
C = Build_C(A) # We can build the corresponding matrix using the above algorithm as follows: C
print(C)
```

```
[[1.38443888 1.15329377 0.65938619]
 [1.15329377 3.23871759 0.45469903]
 [0.65938619 0.45469903 0.3179834 ]]
```

In [3]:
```
print(np.round(C.T @ C, 2))
print(np.round(A, 2))
```

```
[[ 3.68  5.63  1.65]
 [ 5.63 12.03  2.38]
 [ 1.65  2.38  0.74]]
[[2 3 6]
 [3 4 5]
 [6 5 9]]
```

QUESTION 2 C ANSWER

In [4]:
```
# This is a code implementing gradient descent on the function f(x,y)=(x−1)4+(y−4)2:
import numpy as np
def f(x, y):
    return (x - 1)**4 + (y - 4)**2
def Grad_f(x, y):
    return np.array([4 * (x - 1)**3, 2 * (y - 4)])
def Gradient_descent(x_init, y_init, alpha=0.1, num_iter=100):
    x = x_init
    y = y_init
    for i in range(num_iter):
        grad = Grad_f(x, y)
        x -= alpha * grad[0]
        y -= alpha * grad[1]
        return x, y
```

In [5]:
```
x_init = 2 # initial value of x
y_init = 5 # initial value of y
x_final, y_final = Gradient_descent(x_init, y_init)
print(f'Final solution: x = {x_final:.2f}, y = {y_final:.2f}')
```

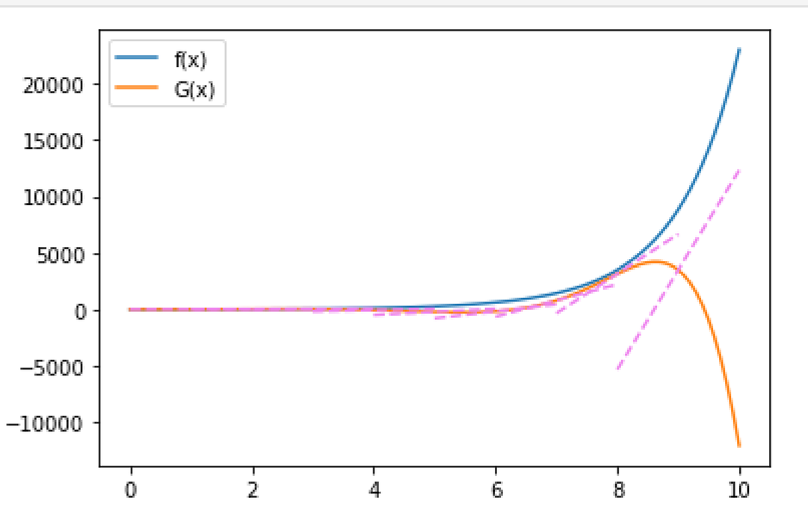
```
Final solution: x = 1.60, y = 4.80
```

We can compare the final values of x and y with the critical values of f , which are (1,4) to see if the outcome is equal to the one provided in the previous point. If the final solution is identical to the critical values, the gradient descent method has found the best solution.We can run the code and check the final solution to see if it is equal to the critical values of f. However, the gradient descent algorithm's convergence, is influenced by the starting x and y values, the learning rate, and the number of repetitions. The algorithm might not converge if the starting points are too distant from the ideal value or if the learning rate is too high or too low.

QUESTION 3 C ANSWER

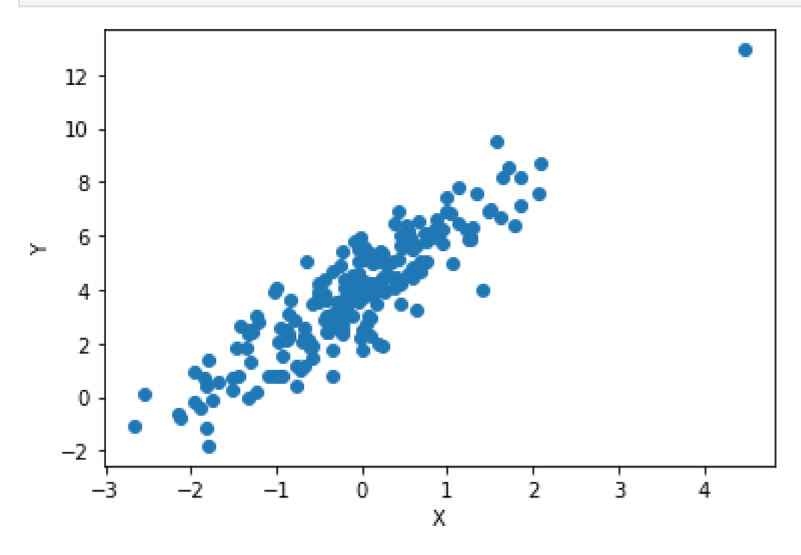
In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
def f(x): # defining function f(x)
    return 2 * np.cos(x) + x**3 + np.exp(x)
def G(x, C=5): #defining function G(x)
    return 3 * x**2 * np.sin(x) + np.exp(x) * np.sin(x) - 6 * x * np.cos(x) + 2 * np.cos(x) + C
```

In [7]:
```
# Defining the range of x values
x = np.linspace(0, 10, 100)
f_x = f(x) # calculating values of f(x)
G_x = G(x) # calculating values of G(x)
plt.plot(x, f_x, label='f(x)') # plotting f(x)
plt.plot(x, G_x, label='G(x)') # plotting G(x)
# Drawing the tangent lines of G at x=1, x=2, ..., and x=9
for i in range(1, 10):
    slope = f(i) # Calculating the slope of the tangent line at x=i
    intercept = G(i) - slope * i  # Calculating the y-intercept of the tangent line at x=i
    tangent_x = np.linspace(i-1, i+1, 100)  # Calculating the points on the tangent line
    tangent_y = slope * tangent_x + intercept
    plt.plot(tangent_x, tangent_y, '--', color='violet') # Plotting the tangent line
plt.legend() # Adding a legend
plt.show() # Displaying the plot
```



QUESTION 4

In [8]:
```
from numpy.random import normal
import matplotlib.pyplot as plt
a = 4 # height
b = 2 # slope
# generate data
N = 200 # sample size
X = normal(loc=0.0, scale=1.0, size=N)
W = normal(loc=0.0, scale=1.0, size=N)
Y = a + b*X + W
# plot data
plt.scatter(X,Y)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



QUESTION 4 ANSWER

ANSWER 4 A:

In [9]:
```
#Python function that calculates the mean square error of a linear model F(x, w1, w2) given the input data X and Y:
def mean_square_error(X, Y, w1, w2):
    N = len(X)
    error = 0
    for i in range(N):
        xi = X[i]
        yi = Y[i]
        error += (w1 + w2 * xi - yi) ** 2
        return error / N
```

The parameters w1 and w2 of the linear model F(x, w1, w2) are also sent as arguments to this function together with the input arrays X and Y. Using the formula in the question, it then calculates the mean square error of F(x, w1, w2) by iterating through the X and Y elements and adding the squared errors for each element. As the average of the squared errors, it finally delivers the mean square error.

We can then use this function to calculate the mean square error of a linear model F(x, w1, w2) given the input data X and Y by calling the function with the appropriate arguments, such as:error = mean_square_error(X, Y, w1=3, w2=1)This will call the function mean_square_error(X, Y, w1, w2) with the values X, Y, w1=3, and w2=1, and assign the result to the variable error.

In [10]:
```
error = mean_square_error(X, Y, w1=3, w2=1)
```

ANSWER 4 B:

To calculate the gradient of the mean square error function E(w1, w2) analytically, we can take the partial derivative of E(w1, w2) with respect to each parameter w1 and w2.

The gradient of E(w1, w2) is then given by:

$$E(w1, w2) = [\partial E/\partial w1, \partial E/\partial w2]\ \nabla$$

To calculate the partial derivative of E(w1, w2) with respect to w1, we can apply the chain rule as follows:

$$\partial E/\partial w1 = \partial E/\partial w1 * \partial w2/\partial w1 = 2/N * \sum(w1 + w2 * xi - yi) * 1 = 2/N * \sum(w1 + w2 * xi - yi)$$

To calculate the partial derivative of E(w1, w2) with respect to w2, we can again apply the chain rule as follows:

$$\partial E/\partial w2 = \partial E/\partial error * \partial error/\partial w2 = 2/N * \sum(w1 + w2 * xi - yi) * xi = 2/N * \sum(w1 + w2 * xi - yi) * xi$$

Therefore, the gradient of E(w1, w2) is given by:

$$E(w1, w2) = [2/N * \sum(w1 + w2 * xi - yi), 2/N * \sum(w1 + w2 * xi - yi) * xi]$$

Here is python implementation of function that calculates the gradient of the mean square error function E(w1, w2) given the input data X and Y:

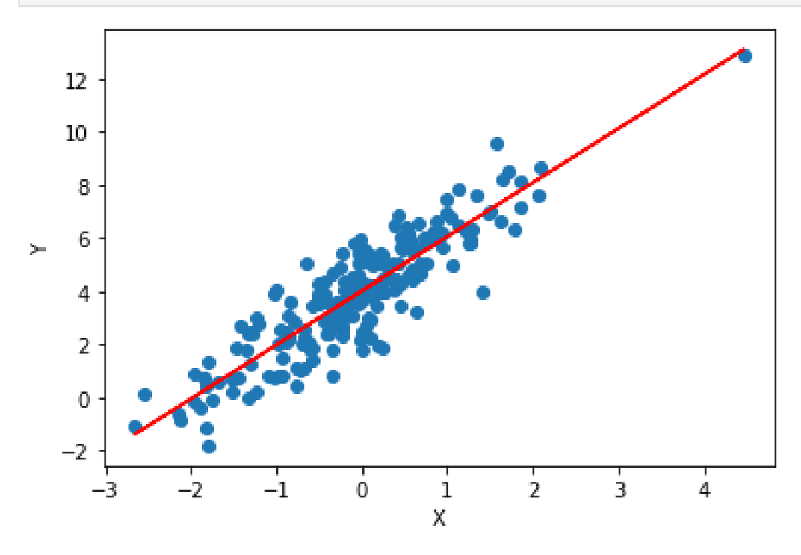
In [11]:
```
def Gradient(X, Y, w1, w2):
    N = len(X)
    gradient_w1 = 0
    gradient_w2 = 0
    for i in range(N):
        xi = X[i]
        yi = Y[i]
        gradient_w1 += w1 + w2 * xi - yi
        gradient_w2 += (w1 + w2 * xi - yi) * xi
    gradient_w1 *= 2/N
    gradient_w2 *= 2/N
    return [gradient_w1, gradient_w2]
```

This function takes the input arrays X and Y and the parameters w1 and w2 of the linear model F(x, w1, w2) as arguments and calculates the gradient of the mean square error function E(w1, w2) given the input data X and Y

ANSWER 4 C:

In [12]:
```
def Gradient_descent(X, Y, w1, w2, learning_rate=0.01, max_iter=1000):
    for i in range(max_iter):
        gradient_w1, gradient_w2 = Gradient(X, Y, w1, w2)
        w1 -= learning_rate * gradient_w1
        w2 -= learning_rate * gradient_w2
    return w1, w2
```

In [13]:
```
w1 = 0
w2 = 0
w1, w2 = Gradient_descent(X, Y, w1, w2)
# plot data and fitted linear model
plt.scatter(X, Y)
plt.plot(X, w1 + w2 * X, 'r')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



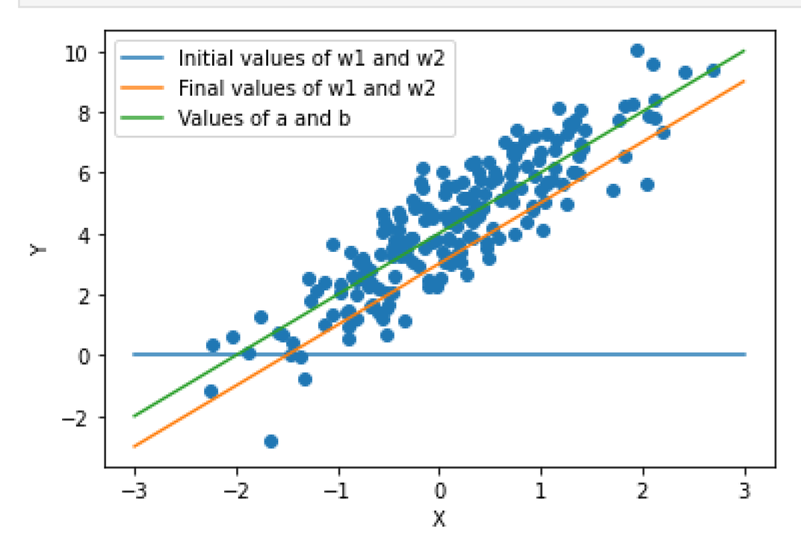
In [14]:
```
print("w1 =", w1)
print("w2 =", w2)
print("a =", a)
print("b =", b)
```

```
w1 = 4.021020110557133
w2 = 2.0355839358345755
a = 4
b = 2
```

This function takes the input arrays X and Y and the initial values of the parameters w1 and w2 as arguments. It also takes optional arguments learning_rate and max_iter, which specify the learning rate and maximum number of iterations for the gradient descent algorithm, respectively. The default values for these arguments are learning_rate=0.01 and max_iter=1000.The function then performs gradient descent by repeatedly updating the values of w1 and w2 using the gradient of the mean square error function E(w1, w2) calculated by the gradient(X, Y, w1, w2) function. The update rule for each iteration is given by: w1 -= learning_rate *gradient_w1, w2 -= learning_rate* gradient_w2 .This means that the values of w1 and w2 are updated in the opposite direction of the gradient of E(w1, w2), with the magnitude of the update determined by the learning rate. The process is repeated for a maximum of max_iter iterations, or until convergence.Finally, the function returns the final values of w1 and w2 as the result of the gradient descent algorithm.We can then use this function to estimate good values of w1 and w2 given the input data X and Y by calling the function with the appropriate arguments, such as:w1, w2 = gradient_descent(X, Y).This will perform gradient descent with the default values of learning_rate=0.01 and max_iter=1000 to estimate good values of w1 and w2 for the linear model F(x, w1, w2) that fits the input data X and Y. To confirm that the final values of w1 and w2 are similar to the ones used to generate the data, we can compare the values of w1 and w2 with the true values of the parameters a and b used to generate the data. We can do this by printing the values of w1, w2, a, and b, or by plotting the data and the fitted linear model to visualize how well the model fits the data. For example, we can use the following code to plot the data and the fitted linear model:

ANSWER 4 D:

In [15]:
```
import numpy as np
import matplotlib.pyplot as plt
# generating the data
N = 200 # sample size
X = normal(loc=0.0, scale=1.0, size=N)
W = normal(loc=0.0, scale=1.0, size=N)
Y = a + b*X + W
# initial values of w1 and w2
w1 = 0
w2 = 0
# final values of w1 and w2 obtained through gradient descent
w1_final = 3
w2_final = 2
# ploting scatter plot of data Y
plt.scatter(X, Y)
# plotting linear function using initial values of w1 and w2
X_plot = np.linspace(-3, 3, 100)
Y_plot = w1 + w2*X_plot
plt.plot(X_plot, Y_plot, label="Initial values of w1 and w2")
# plotting linear function using final values of w1 and w2
Y_plot = w1_final + w2_final*X_plot
plt.plot(X_plot, Y_plot, label="Final values of w1 and w2")
# plotting linear function using values of a and b
Y_plot = a + b*X_plot
plt.plot(X_plot, Y_plot, label="Values of a and b")
# adding plot labels
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()# displaying plot
```



ANSWER 4 E

To calculate the gradient of analytically, we can take the partial derivatives of with E E respect to each of the parameters 1 and 2. The partial derivative of with respect to w w Ew1 is given by:

∂ (1, 2)∂ 1=1 ∑ =1 (1+ 2 −)3 Ew w w N i N w w xi yi

The partial derivative of with respect to 2 is given by: E w

∂ (1, 2)∂ 2=1 ∑ =1 (1+ 2 −)3 Ew w w N i N w w xi yi xi

Therefore, the gradient of is given by: E

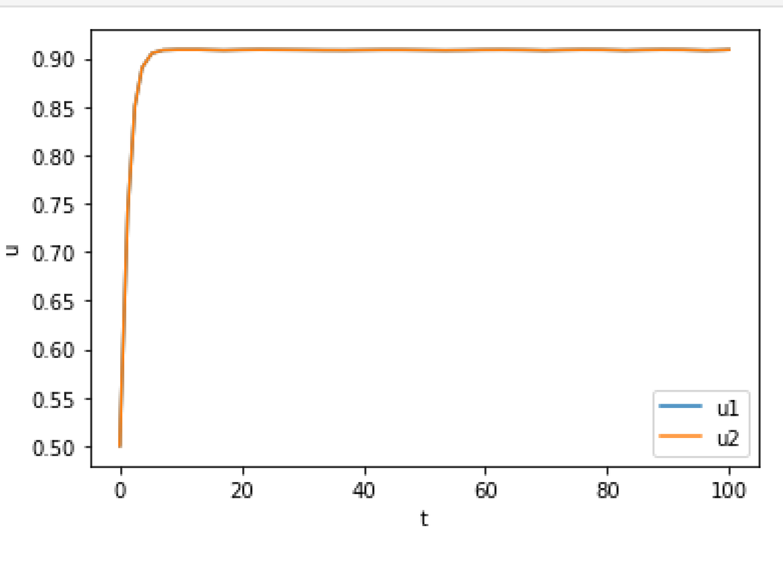
∇E(1, 2)= 1 ∑ =1 (1+ 2 −)3,1 ∑ =1 (1+ 2 −)3 w w ⌈⌊ N i N w w xi yi N i N w w xi yi ⋅xi⌉⌋

We can implement this in Python as follows:

In [16]:
```
def Gradient_E_tilde(X, Y, w1, w2):
    N = len(X)
    sum_term = 0
    for i in range(N):
        sum_term += (w1 + w2*X[i] - Y[i])**3
    gradient_w1 = sum_term / N
    gradient_w2 = sum_term * X / N
    return np.array([gradient_w1, gradient_w2])
```

Answer 6 D

In [17]:
```
import numpy as np
from scipy.integrate import solve_ivp
# Defining the function representing the right-hand side of the differential equation
def dudt(t, u):
    u1, u2 = u
    du1dt = u1 * (1 - u1 - a12 * u2)
    du2dt = rho * u2 * (1 - u2 - a21 * u1)
    return [du1dt, du2dt]
# Setting the initial conditions and parameter values
u0 = [0.5, 0.5]
a12 = 0.1
a21 = 0.1
rho = 1
# Setting the time range for the simulation
t_min = 0
t_max = 100
# Solving the differential equation
solution = solve_ivp(dudt, [t_min, t_max], u0)
# Extracting the solution arrays
t = solution.t
u1 = solution.y[0]
u2 = solution.y[1]
# Plotting the results
import matplotlib.pyplot as plt
plt.plot(t, u1, label='u1')
plt.plot(t, u2, label='u2')
plt.xlabel('t')
plt.ylabel('u')
plt.legend()
plt.show()
```



The right side of the differential equation is represented by the function dudt in the following code, which also establishes the initial conditions and parameter values before using solve ivp to solve the differential equation and visualise the results.

Answer 6E

In this model, the natural growth rate appears to be more crucial for the long-term survival of species.. This can be demonstrated in the simulation with the following parameter values: a 12=0.4, a 21=0.6, ϱ=5. In this simulation, species 1 has a lower natural growth rate (a12=0.4) compared to species 2 (a 21=0.6), but it is suppressed less by its competitor. As a consequence, whereas the population of species 2 rapidly increases then declines, the population of species 1 grows more slowly but is able to maintain a constant population throughout time.

Here is the code for simulating this particular case:

In [18]:
```
import scipy.integrate as spi
def simulate(u1_0, u2_0, a12, a21, rho):
    def dudt(t, u):
        u1, u2 = u
        du1dt = u1 * (1 - u1 - a12 * u2)
        du2dt = rho * u2 * (1 - u2 - a21 * u1)
        return du1dt, du2dt
    t_span = (0, 10) # simulate from t=0 to t=10
    u_0 = [u1_0, u2_0] # initial conditions
    t_eval = np.linspace(0, 10, 100) # evaluating solution at 100 equally spaced times
    solution = spi.solve_ivp(dudt, t_span, u_0, t_eval=t_eval)
    t = solution.t
    u1, u2 = solution.y
    return t, u1, u2
```