Vectors

1. (10 points) Given the following vectors:

$$p = \begin{bmatrix} 5 \\ -1 \\ 1 \end{bmatrix}, \qquad q = \begin{bmatrix} -4 \\ -2 \\ 7 \end{bmatrix}, \qquad r = \begin{bmatrix} -1 \\ -2 \\ 3 \end{bmatrix}$$

Determine the following **by hand** (remember to show your work):

a. $p + 2q$

b. $p \cdot r$ and $r \cdot p$ where "$\cdot$" denotes the dot product

a. $p + 2q = \begin{bmatrix} 5 \\ -1 \\ 1 \end{bmatrix} + 2 \cdot \begin{bmatrix} -4 \\ -2 \\ 7 \end{bmatrix} = \begin{bmatrix} -3 \\ -5 \\ 15 \end{bmatrix}$

b. $p \cdot r = \begin{bmatrix} 5 \\ -1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ -2 \\ 3 \end{bmatrix} = -5 + 2 + 3 = 0$

$r \cdot p = \begin{bmatrix} -1 \\ -2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ -1 \\ 1 \end{bmatrix} = -5 + 2 + 3 = 0$

c. $q \times r$ and $r \times q$ where "$\times$" denotes the cross product

d. $||p||$, and $||q||$ where $||\cdot||$ denotes the Euclidean norm of a vector

e. distance between the tips of p and q

$$q \times r = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ -4 & -2 & 7 \\ -1 & -2 & 3 \end{bmatrix} = \vec{i} \cdot \left[(-2)\cdot(3) - (7)\cdot(-2)\right] - \vec{j} \cdot \left[(-4)(3) - (7)(-1)\right] + \vec{k} \cdot \left[(-4)\cdot(-2) - (-2)\cdot(-1)\right]$$

$$= \vec{i}(8) - \vec{j}(-5) + \vec{k}(6) = \begin{bmatrix} 8 \\ 5 \\ 6 \end{bmatrix}$$

$r \times q = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ -1 & -2 & 3 \\ -4 & -2 & 7 \end{bmatrix} = \vec{i}\cdot\left[(-1)(-2)-(-2)(-4)\right] - \vec{j}\left[(-1)(7)-(3)(-4)\right] + \vec{k}\left[(-1)(-2)-(-2)(-4)\right]$

$= \vec{i}\cdot(-8) - \vec{j}\,(5) + \vec{k}\cdot(-6) = \begin{bmatrix} -8 \\ -5 \\ -6 \end{bmatrix}$

d. $\|p\| = \sqrt{5^2+1^2+1^2} = 3\sqrt{3}$

$\|q\| = \sqrt{(-4)^2+(-2)^2+7^2} = \sqrt{69}$

e: $d = \sqrt{(5-(-4))^2 + (-1-(-2))^2 + (1-7)^2}$

$= \sqrt{9^2+1^2+6^2} = \sqrt{118}$

2. (10 points) Find all k such that $p = \begin{bmatrix} -2 \\ 1 \\ -k \end{bmatrix}$ and $q = \begin{bmatrix} 2 \\ -3k \\ -k \end{bmatrix}$ are orthogonal **by hand**.

$\vec{p}\cdot\vec{q} = 0 = -2\cdot 2 + (-3k)\cdot 1 + (-k)(-k)$

$= k^2 - 3k - 4 = (k+1)(k-4)$

so $k_1 = -1 \quad k_2 = 4$

$k_1 = -1$

$\vec{p}_1 = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \quad \vec{q}_1 = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \quad \vec{p}_1\cdot\vec{q}_1 = -4+3+1 = 0$

$k_2 = 4$

$\vec{p}_2 = \begin{bmatrix} -2 \\ 1 \\ -4 \end{bmatrix} \quad \vec{q}_2 = \begin{bmatrix} 2 \\ -12 \\ -4 \end{bmatrix} \quad \vec{p}_2\cdot\vec{q}_2 = -4-12+16 = 0$

3. (5 points) Partition the following matrix into submatrices (i.e. find W, X, Y, and Z) **by hand**:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix}$$

where $W \in \mathbb{R}^{2\times 1}$ and $Z \in \mathbb{R}^{2\times 3}$.

$W \in \mathbb{R}^{2\times 1}$ $Z \in \mathbb{R}^{2\times 3}$

So: $X \in \mathbb{R}^{2\times 3}$ $Y \in \mathbb{R}^{2\times 1}$

$$\Rightarrow W = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad X = \begin{bmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \end{bmatrix}$$

$$Y = \begin{bmatrix} 9 \\ 13 \end{bmatrix} \quad Z = \begin{bmatrix} 10 & 11 & 12 \\ 14 & 15 & 16 \end{bmatrix}$$

4. (10 points) Perform the following matrix multiplication **by hand**:

$$\begin{bmatrix} 3 & -1 & -3 \\ -1 & 0 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & -1 \\ 0 & -2 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}$$

$\mathbb{R}^{2\times3} \cdot \mathbb{R}^{3\times3} \cdot \mathbb{R}^{3\times1} = \mathbb{R}^{2\times1}$

$$= \begin{bmatrix} 3\times0+(-1)\times(-1)+(-3)\times0, & 3\times1+(-1)\cdot0+(-3)(-2), & 3\times2+(-1)(-1)+(-3)(-1) \\ -1\cdot0+0\cdot(-1)+2\cdot0, & -1\cdot1+0\cdot0+2\cdot(-2), & -1\cdot2+(0)(-1)+2\cdot(-1) \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 9 & 10 \\ 0 & -5 & -4 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}$$

$$= \begin{bmatrix} 1\times2+9\times1+10\times(-2) \\ 0\times2+1\times(-5)+(-4)\times(-2) \end{bmatrix} = \begin{bmatrix} -9 \\ 3 \end{bmatrix}$$

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
```

for homework 1, we will solve the following problems using numpy.

Problem 5.

Solve the following systems of equations using numpy. Recall that there can be a unique solution, no solution, or infinitely many solutions.

a. $\begin{pmatrix} 0 & 0 & -1 \\ 4 & 1 & 1 \\ -2 & 2 & 1 \end{pmatrix} x = \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}$

b. $\begin{pmatrix} 0 & -2 & 6 \\ -4 & -2 & -2 \\ 2 & 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ -2 \\ 0 \end{pmatrix}$

c. $\begin{pmatrix} 2 & -2 \\ -4 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ -2 \end{pmatrix}$

In [17]:
```
def check_solution(A, b):
    det_A = np.linalg.det(A) # determinant of A

    if det_A != 0:
        x_sol = np.linalg.solve(A, b)
        return 'The solution of the system is:{}'.format(x_sol)

    else:
        aug_matrix = np.hstack((A, b.reshape(-1, 1))) # augmented matrix
        rank_A = np.linalg.matrix_rank(A)  # calculate the rank of A
        # check if the system has infinitely many solutions
        rank_aug = np.linalg.matrix_rank(aug_matrix)

        if rank_A == rank_aug:
            return "infinitely many solutions"
        else:
            return "No solution"
# a.
# Define the matrices
A_1 = np.array([[0, 0, -1], [4, 1, 1], [-2, 2, 1]])
b_1 = np.array([3, 1, 1])
# Solve the system
print('-------------------------------------------------')
print(check_solution(A_1, b_1))

# b.
# Define the matrices
A_2 = np.array([[0, -2, 6], [-4, -2, -2], [2, 1, 1]])
b_2 = np.array([1, -2, 0])
# Solve the system
print('-------------------------------------------------')
print(check_solution(A_2, b_2))

# c.
# Define the matrices
A_3 = np.array([[2, -2], [-4, 3]])
b_3 = np.array([3, -2])
```

```
# Solve the system
print('-------------------------------------------------')
print(check_solution(A_3, b_3))
```

```
-------------------------------------------------
The solution of the system is:[ 0.4  2.4 -3. ]
-------------------------------------------------
No solution
-------------------------------------------------
The solution of the system is:[-2.5 -4. ]
```

Problem 6.

Given the following matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} -2 & -2 \\ 4 & -3 \end{pmatrix}$$

Use numpy to calculate the following. a. $A + 2B$ b. AB and BA c. A^T, transpose of A d. B^2 e. $A^T B^T$ and $(AB)^T$ f. $det(A)$, determinant of A g. B^{-1}, inverse of B

In [18]:
```
# Define the matrices
A = np.array([[1, 2], [3, -1]])
B = np.array([[-2, -2], [4, -3]])

# a
A_plus_2B = A + 2*B
print('-------------------------------------------------')
print('A + 2B is:\n', A_plus_2B)

# b
A_dot_B = np.dot(A, B)
B_dot_A = np.dot(B, A)
print('-------------------------------------------------')
print('AB is:\n', A_dot_B)
print('BA is:\n', B_dot_A)

# c
A_transpose = A.T
print('-------------------------------------------------')
print('A^T is:\n', A_transpose)

# d
B_squared = np.dot(B, B)
print('-------------------------------------------------')
print('B^2 is:\n', B_squared)

# e
A_dot_B_transpose = np.dot(A.T, B.T)
A_dot_B_dot_transpose = np.dot(A, B).T
print('-------------------------------------------------')
print('A^TB^T is:\n', A_dot_B_transpose)
print('(AB)^T is:\n', A_dot_B_dot_transpose)

# f
det_A = np.linalg.det(A)
print('-------------------------------------------------')
print('det(A) is:', det_A)
```

```
# g
B_inverse = np.linalg.inv(B)
print('-------------------------------------------------')
print('B^-1 is:\n', B_inverse)
```

```
-------------------------------------------------
A + 2B is:
 [[-3 -2]
 [11 -7]]
-------------------------------------------------
AB is:
 [[  6  -8]
 [-10  -3]]
BA is:
 [[-8 -2]
 [-5 11]]
-------------------------------------------------
A^T is:
 [[ 1  3]
 [ 2 -1]]
-------------------------------------------------
B^2 is:
 [[ -4  10]
 [-20   1]]
-------------------------------------------------
A^TB^T is:
 [[-8 -5]
 [-2 11]]
(AB)^T is:
 [[  6 -10]
 [ -8  -3]]
-------------------------------------------------
det(A) is: -7.000000000000001
-------------------------------------------------
B^-1 is:
 [[-0.21428571  0.14285714]
 [-0.28571429 -0.14285714]]
```

Problem 7.

A rotation matrix R is defined by the following sequence of basic rotations:

i. A rotation of π/2 about the z-axis ii. A rotation of −π/5 about the new y-axis iii. A rotation of π about the new z-axis

In [19]:
```
# Define the rotation matrix R
R = np.eye(3)
T_1 = np.array([[np.cos(np.pi/2), -np.sin(np.pi/2), 0],
                [np.sin(np.pi/2), np.cos(np.pi/2), 0],
                [0, 0, 1]])
T_2 = np.array([[np.cos(-np.pi/5), 0, np.sin(-np.pi/5)],
                [0, 1, 0],
                [-np.sin(-np.pi/5), 0, np.cos(-np.pi/5)]])
T_3 = np.array([[np.cos(np.pi), -np.sin(np.pi), 0],
                [np.sin(np.pi), np.cos(np.pi), 0],
                [0, 0, 1]])
R_final = np.dot(np.dot(T_1, T_2),T_3)
print('The rotation matrix R is:\n', R_final.round(4))

# Verify that the rotation matrix R is a valid rotation matrix
```

```
# Determinant equals 1: The determinant of a valid rotation matrix must be 1
det_R = np.linalg.det(R_final)
print('The determinant of R is:', det_R)
# Orthogonality: The columns (or rows) of the rotation matrix must be orthogonal to each other, meaning that the dot product of any two distinct columns (or rows) must be zero.
print('The columns of R are orthogonal:', np.allclose(np.dot(R_final.T, R_final), np.eye(3)))
```

```
The rotation matrix R is:
 [[-0.      1.     -0.    ]
 [-0.809  -0.     -0.5878]
 [-0.5878 -0.      0.809 ]]
The determinant of R is: 1.0
The columns of R are orthogonal: True
```

Problem 8.

You would like to point a mobile robot so that it is looking at a target point. The camera of the robot is aligned with x in the robot's coordinate frame. The robot is assumed to be on flat ground (z = 0). The robot's pose in the world frame is:

$$T_r^0 = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 1.7 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 2.1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

a. The target point is at p = [−0.4,0.9,0]' in the world frame. Assuming the robot stays at its current position, calculate the vector v in the world frame that the camera should align with.

In [20]:
```
# Calculate the vector v in the world frame that the camera should align with
T_r0 = np.array([[np.sqrt(2)/2, np.sqrt(2)/2, 0, 1.7],
                 [-np.sqrt(2)/2, np.sqrt(2)/2, 0, 2.1],
                 [0, 0, 1, 0],
                 [0,0,0,1]])

p = np.array([-0.4, 0.9, 0]).T

T_position = T_r0[:3,3]
v = (p - T_position)/np.linalg.norm(p - T_position)
print('The vector v in the world frame that the camera should align with is:', v)
```

```
The vector v in the world frame that the camera should align with is: [-0.86824314 -0.49613894  0.        ]
```

b. Use v to calculate the desired pose of the robot, again assume the robot does not change position. The robot must remain upright on the ground; i.e. its z axis must not change.

In [21]:
```
# Calculate the desired pose of the robot
Pose_Matrix = np.zeros((4,4))
Pose_Matrix[:,0] = np.append(v.T,0)
Pose_Matrix[:,2] = [0,0,1,0]
# The y-axis can be computed as the cross product of the z-axis and the x-axis
# Mind the order of the cross product!
Pose_Matrix[:,1][:3] = np.cross(Pose_Matrix[:,2][:3], Pose_Matrix[:,0][:3])
Pose_Matrix[:,3] = T_r0[:,3]
print('The desired pose of the robot is:\n', Pose_Matrix)
```

```
The desired pose of the robot is:
 [[-0.86824314  0.49613894  0.          1.7       ]
 [-0.49613894 -0.86824314  0.          2.1       ]
 [ 0.          0.          1.          0.        ]
 [ 0.          0.          0.          1.        ]]
```

c. Prove that the rotation component of the pose meets all conditions for being a valid rotationmatrix.

In [22]:
```
# Prove that the rotation component of the pose meets all conditions for being a valid rotation matrix
R = Pose_Matrix[:3,:3]
print('The rotation matrix R is:\n', R)
# Determinant equals 1: The determinant of a valid rotation matrix must be 1
det_R = np.linalg.det(R)
print('The determinant of R is:', det_R)
# Orthogonality: The columns (or rows) of the rotation matrix must be orthogonal to each other, meaning that the dot product of any two distinct columns (or rows) must be zero.
print('The columns of R are orthogonal:', np.allclose(np.dot(R.T, R), np.eye(3)))
```

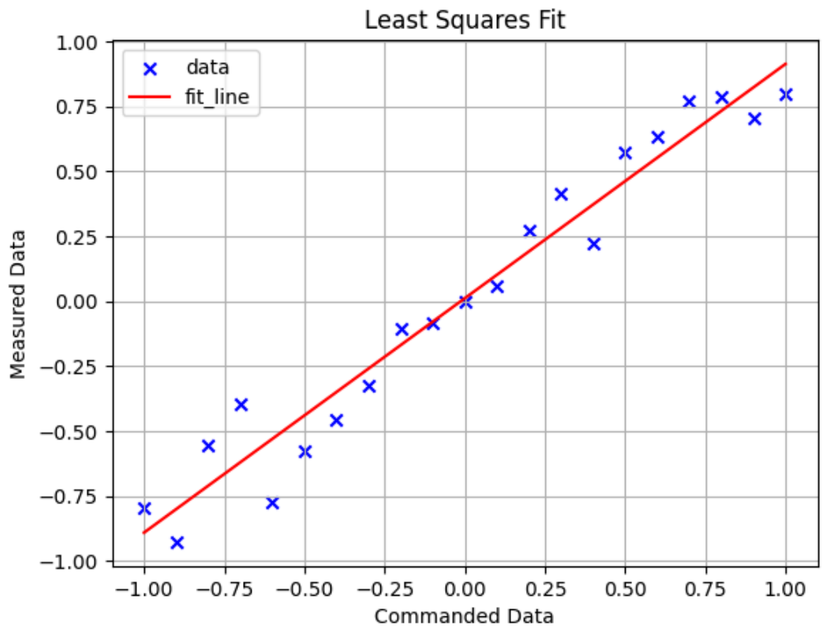
```
The rotation matrix R is:
 [[-0.86824314  0.49613894  0.        ]
 [-0.49613894 -0.86824314  0.        ]
 [ 0.          0.          1.        ]]
The determinant of R is: 0.9999999999999999
The columns of R are orthogonal: True
```

Problem 9.

You are testing one of the joints of a new robot arm and you notice there is some error when moving to a target position. To investigate, you command the joint to move through a series of positions and measure where the joint moves for each command. The recorded data isin calibration.txt, where the first column is the commanded position and the second column is the measured position. You see that there is significant error in the commanded vs. measured positions. Note: You are strongly advised to consult the linear algebra book, section 13.1 for this problem, especially part (c). a.Use the psuedo-inverse to perform a least-squares fit of a line to the data. Produce a plot showing the data as blue xs and the line you computed in red and include it in the pdf. Include the parameters of the line and the sum of squared error of the fit in your pdf. Submit your code as leastsquares.py to Gradescope. When we run the code, it should produce the plot and print out the parameter values and the sum of squared errors.

In [23]:
```
# for Probelem a)
# Load data from file
data = np.loadtxt('calibration.txt')
Commanded_data = data[:,0]
Measured_data = data[:,1]
# Pseudo-inverse
A = np.vstack([Commanded_data, np.ones(Commanded_data.shape[0])]).T
ATA = np.dot(A.T, A)
#A_pseudo_inv = np.linalg.inv(ATA) @ A.T
A_pseudo_inv = np.linalg.pinv(A)

# Least-squares fit
parameters = np.dot(A_pseudo_inv, Measured_data)

slope = parameters[0]  # slope of the line
intercept = parameters[1]  # intercept of the line

# calculate fitted values
fitted_values = slope * Commanded_data + intercept

# Sum of Squared Errors
squared_errors = (Measured_data - fitted_values) ** 2
sum_of_squared_errors = np.sum(squared_errors)

plt.scatter(Commanded_data, Measured_data, color='blue', label='data', marker='x')
plt.plot(Commanded_data, fitted_values, color='red', label='fit_line')
plt.xlabel('Commanded Data')
plt.ylabel('Measured Data')
plt.legend()
plt.title('Least Squares Fit')
plt.grid()
plt.show()
```

```
print('The Slope is:', slope)
print('The Intercept is:', intercept)
print('The Sum of Squared Errors is:', sum_of_squared_errors)
```




```
The Slope is: 0.9025793376623379
The Intercept is: 0.011078666666666674
The Sum of Squared Errors is: 0.31059694806528876
```

b. (5 points) Is this least-squares problem underdetermined or overdetermined? Explain your answer.

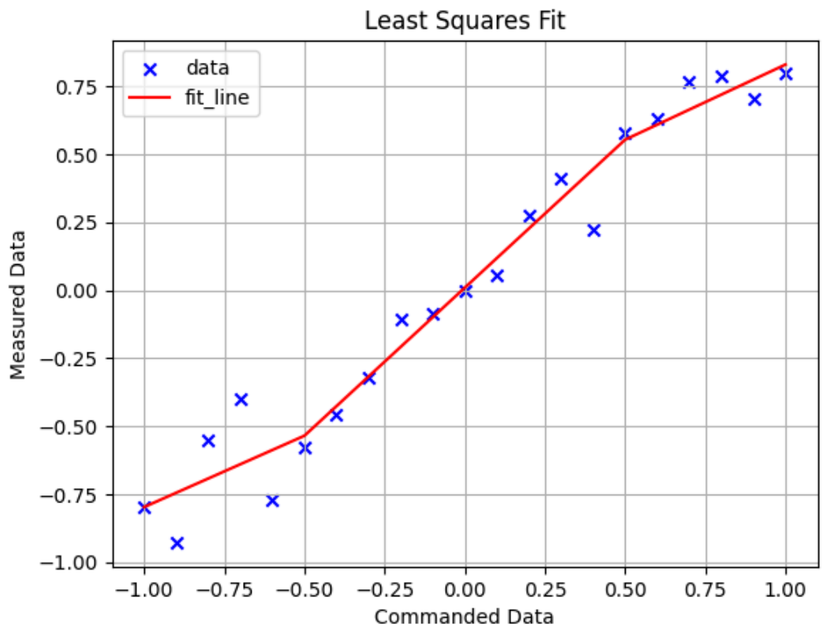
The problem is overdetermined because the number of data points is greater than the number of parameters to be estimated. In this case, we have 21 data points and 2 parameters to estimate (slope and intercept). Therefore, the problem is overdetermined.

c. (15 points) You notice that the error is quite high using a linear fit and you notice that the error seems lower in the range [-0.5, 0.5] than elsewhere. Perform a piece-wise linear least-squares fit to the data using -0.5 and 0.5 as the knotpoints. You should not perform perform multiple least- squares fits to different parts of the data, i.e. you should set up the whole problem as Ax = b and solve for all the parameters simultaneously. Produce a plot in the same format as in (a) and include it in your pdf along with the parameters you computed and the sum of squared errors. Use the parameters you computed in (c) to predict what the measured position will be when you apply the command 0.68. Include the prediction in your pdf. Submit your code as pwleastsquares.py to Gradescope. When we run the code, it should produce the plot and print out the parameter values, the sum of squared errors, and the prediction for 0.68.

In [24]:
```
# Load data from file
data = np.loadtxt('calibration.txt')
# Load data from file
Commanded_data = data[:,0]
Measured_data = data[:,1]

# Piece-wise linear least-squares fit
knotpoints = [-0.5, 0.5]

# Create design matrix
# see testbook section 13.1- piece-wise linear least-squares fit
B = np.zeros((Commanded_data.shape[0], 4))
B[:,0] = 1
B[:,1] = Commanded_data
B[:,2] = np.maximum(Commanded_data - knotpoints[0], 0)
B[:,3] = np.maximum(Commanded_data - knotpoints[1], 0)
```

```
BTB = np.dot(B.T, B)
B_pseudo_inv = np.linalg.pinv(B)

# Least-squares fit
parameters = np.dot(B_pseudo_inv, Measured_data)
print('The parameters are:', parameters)
print(parameters)

# calculate fitted values
fitted_values = np.dot(B, parameters)
#print(fitted_values)

# Sum of Squared Errors
squared_errors = (Measured_data - fitted_values) ** 2
sum_of_squared_errors = np.sum(squared_errors)

plt.scatter(Commanded_data, Measured_data, color='blue', label='data', marker='x')
plt.plot(Commanded_data, fitted_values, color='red', label='fit_line')
plt.xlabel('Commanded Data')
plt.ylabel('Measured Data')
plt.legend()
plt.title('Least Squares Fit')
plt.grid()
plt.show()

print('The Sum of Squared Errors is:', sum_of_squared_errors)

# Predicted value for 0.68
predicted_value = parameters[0] + parameters[1] * 0.68 + parameters[2] * np.maximum(0.68 - knotpoints[0], 0) + parameters[3] * np.maximum(0.68 - knotpoints[1], 0)
print('The predicted value for 0.68 is:', predicted_value)
```

```
The parameters are: [-0.27277026  0.52452193  0.56375211 -0.5361319 ]
[-0.27277026  0.52452193  0.56375211 -0.5361319 ]
```



```
The Sum of Squared Errors is: 0.24330448882121877
The predicted value for 0.68 is: 0.6526284002588031
```