CPSC 552 – Data Minining - Assignment #5

Data Compression, Linear Least Square Estimation via Pseudo-Inverse using Singular Value Decomposition

In this assignment, you will program the Singular Value Decomposition and apply it to image compression. The Singular Value Decomposition (SVD) has important applications in Data Sciences. It factorizes the data matrix into a product of three matrices, i.e.,

$$A = U \sum V$$

Where U contains Eigen Vectors of AA^T, $\sum$ is a diagonal matrix containing the Eigen values of the *A* matrix, and V contains the transpose of the Eigen vectors of $A^T A$.

The higher the Eigen value, more information is contained in the projection of the data onto the corresponding Eigen vector for that Eigen value. Thus a common application of the SVD technique is to reduce the amount of data using a few of the highest Eigen values of *A* in the decomposition via the SVD. When we reduce the number of Eigen values to e.g., k, and the corresponding Eigen vectors in the decomposition of the matrix, the resulting approximation for *A* appears as:

$$= \sum_{i=1}^{k} \sigma_i u_i v_i^T = \underbrace{\left[u_1, \cdots, u_k \right]}_{k} \text{diag}\{\sigma_1, \cdots, \sigma_k\} \left[\begin{array}{c} v_1 \\ \vdots \\ v_k \end{array} \right] \Bigg\} k$$

Thus above approximation can be used to compress data, e.g., images, or the attention matrix in Transformer based NLP architectures.

Another application of the SVD technique is in computing the pseudo inverse, also known as the Moore-Penrose inverse. This pseudo inverse is needed in fitting polynomials to experimental data.

You will program the different applications of SVD in this assignment.

Example: Find the SVD of A, $U\Sigma V^T$, where $A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$.

First we compute the singular values σ_i by finding the eigenvalues of AA^T.

$$AA^T = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}.$$

The characteristic polynomial is $det(AA^T - \lambda I) = \lambda^2 - 34\lambda + 225 = (\lambda - 25)(\lambda - 9)$, so the singular values are $\sigma_1 = \sqrt{25} = 5$ and $\sigma_2 = \sqrt{9} = 3$.

Now we find the right singular vectors (the columns of V) by finding an orthonormal set of eigenvectors of $A^T A$. It is also possible to proceed by finding the left singular vectors (columns of U) instead. The eigenvalues of $A^T A$ are 25, 9, and 0, and since $A^T A$ is symmetric we know that the eigenvectors will be orthogonal.

For $\lambda = 25$, we have

$$A^T A - 25I = \begin{pmatrix} -12 & 12 & 2 \\ 12 & -12 & -2 \\ 2 & -2 & -17 \end{pmatrix}$$

which row-reduces to $\begin{pmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$. A unit-length vector in the kernel of that matrix is $v_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{pmatrix}$.

For $\lambda = 9$ we have $A^TA - 9I = \begin{pmatrix} 4 & 12 & 2 \\ 12 & 4 & -2 \\ 2 & -2 & -1 \end{pmatrix}$ which row-reduces to $\begin{pmatrix} 1 & 0 & -\frac{1}{4} \\ 0 & 1 & \frac{1}{4} \\ 0 & 0 & 0 \end{pmatrix}$.

A unit-length vector in the kernel is $v_2 = \begin{pmatrix} 1/\sqrt{18} \\ -1/\sqrt{18} \\ 4/\sqrt{18} \end{pmatrix}$.

For the last eigenvector, we could compute the kernel of A^TA or find a unit vector perpendicular to v_1 and v_2. To be perpendicular to $v_1 = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ we need $-a = b$. Then the condition that $v_2^T v_3 = 0$ becomes $2a/\sqrt{18} + 4c/\sqrt{18} = 0$ or $-a = 2c$. So $v_3 = \begin{pmatrix} a \\ -a \\ -a/2 \end{pmatrix}$ and for it to be unit-length we need $a = 2/3$ so $v_3 = \begin{pmatrix} 2/3 \\ -2/3 \\ -1/3 \end{pmatrix}$.

1

So at this point we know that

$$A = U\Sigma V^T = U \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}.$$

Finally, we can compute U by the formula $\sigma u_i = Av_i$, or $u_i = \frac{1}{\sigma} Av_i$. This gives $U = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$. So in its full glory the SVD is:

$$A = U\Sigma V^T = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}.$$

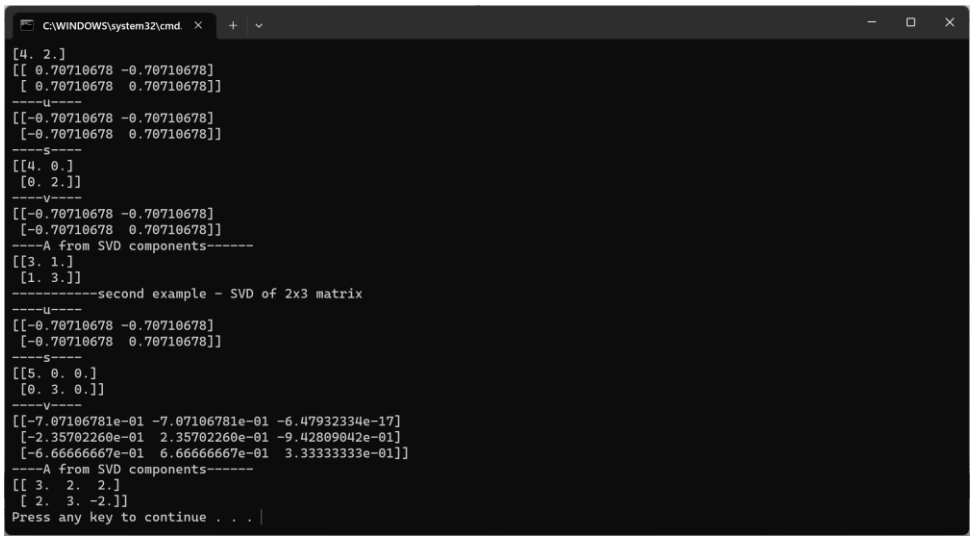
Problem #1: Use the numpy linalg libraray's svd function to compute the SVD decomposition of $\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$ and $\begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix}$,

Solution: Create a Python application called SVD. Type the following code in SVD.py.

```
import sys
import numpy as np

def compute_svd(A):
    u,s,v = np.linalg.svd(A)
    print("----u----")
```

```
    print(u)
    print("----s----")
    s = np.diag(s)
    if (s.shape[1] != A.shape[1]):
        # stack zero columns in the diagonal matrix
        num_zero_cols = A.shape[1] - s.shape[1]
        sz = np.zeros((s.shape[0],A.shape[1]))
        sz[:,:-1] = s  # extra columns of zeros
        s = sz
    print(s)
    print("----v----")
    print(v)
    Asvd = np.dot(np.dot(u,s),v)
    print("----A from SVD components------")
    print(Asvd)

def main():
    # compute Eigen values, Eigen Vectors, SVD for
    #[[3,1],
    # [1,3]]
    Alist = [[3,1],[1,3]]
    A = np.asarray(Alist, dtype=float) # convert list to numpy array
    eigen_vals = np.linalg.eigvals(A)
    print(eigen_vals)
    eigenvs, eigen_vecs = np.linalg.eig(A)
    print(eigen_vecs)
    compute_svd(A)

    # compute SVD of [[3,2,2],[2,3,-2]]
    print("-----------second example - SVD of 2x3 matrix")
    A2list = [[3,2,2],[2,3,-2]]
    A2 = np.asarray(A2list, dtype=float)
    compute_svd(A2)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

If you run the above program, your output will appear as:



Problem #2: Use top *n* Eigen values in the SVD decomposition of an image. Experiment with different values of *n* to see the effect on the reconstruction of the image and the compression percentage achieved.

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \ldots + \sigma_n \mathbf{u}_n \mathbf{v}_n^T$$

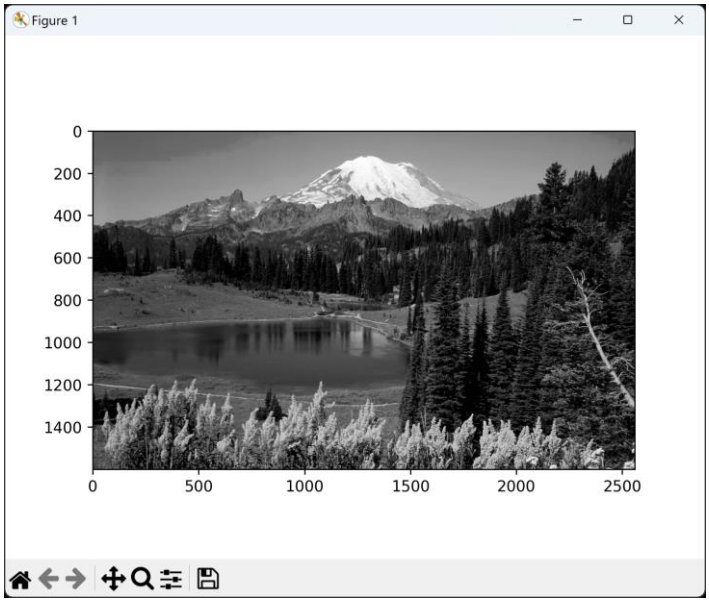
Solution: Add a Python module file to the Python SVD project you created in problem 1 called SVDImage compression.py with the following code in it.

```
import sys
import numpy as np
import cv2
import matplotlib.pyplot as plt

def svd_compression(img, num_components=10):
    u,s,v = np.linalg.svd(img)
    print(img.shape)
    uc = u[:, :num_components]
    sc = s[:num_components]
    vc = v[:num_components, :]
    orig_size = img.shape[0] * img.shape[1]
    compressed_size = uc.shape[0]*uc.shape[1] + sc.shape[0] + vc.shape[0] *
vc.shape[1]
    print("storage needed for orig data =", orig_size)
    print("storage needed for compressed data =", compressed_size, " percentage of
orig size=",compressed_size/orig_size*100,"%")
    compressed_img = np.matrix(u[:, :num_components]) * np.diag(s[:num_components])
* np.matrix(v[:num_components, :])
    plt.imshow(compressed_img, cmap='gray')
    plt.show()
    print(compressed_img.shape)

def main():
    image_filename = 'd:/PythonAM3/Images/scenery1.jpg'
    img = cv2.imread(image_filename,0)
    if img is None:
        print('could not open or find the image: ', image_filename)
        exit(0)
    #cv2.imshow('Original Image', img)
    #cv2.waitKey()
    plt.imshow(img, cmap='gray')
    plt.show()
    num_components = 50
    svd_compression(img, num_components)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

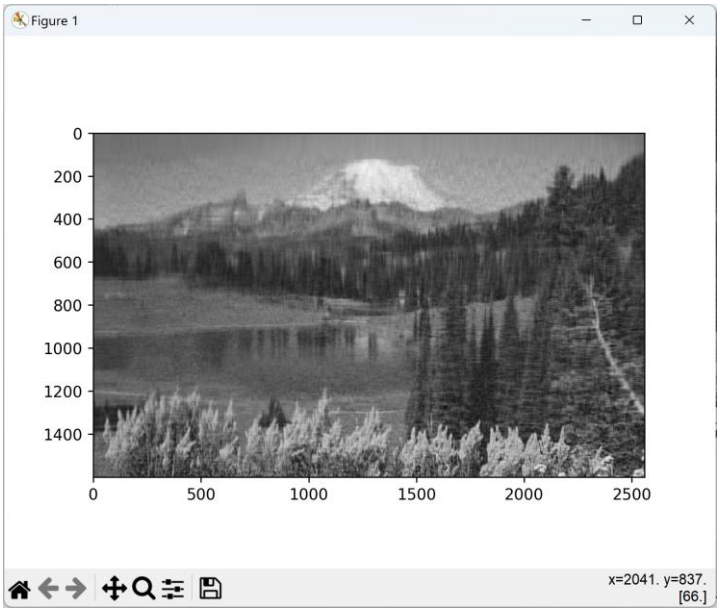
The num_components variable chooses the number of Eigen values and the corresponding Eigen vectors to keep. Experiment with different values ranging from 10 to a few hundred to see the effect on the quality of the reconstructed image from the SVD components and the compression percentage with respect to the original image.

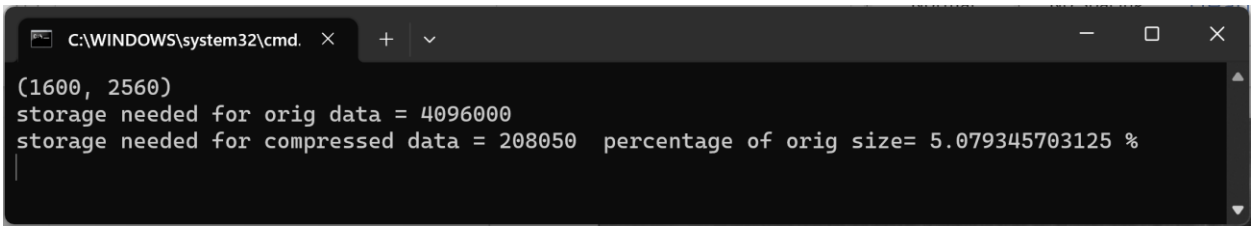
Running the program will show you the original image first, and then once you close the figure, it will show you the reconstructed image from the num_components SVD components.

Original Image appears as:

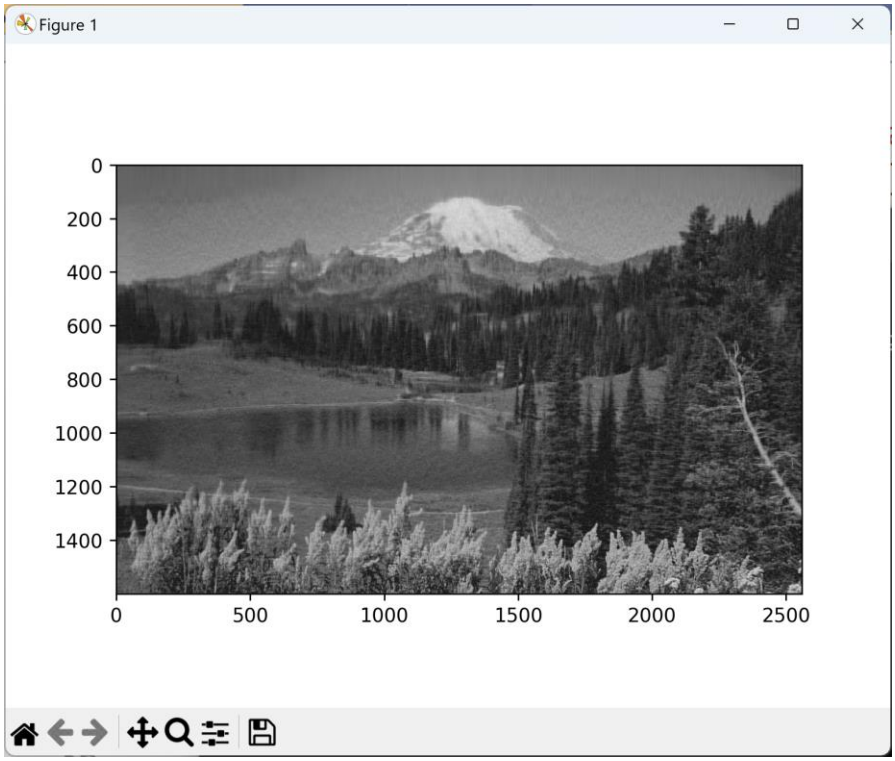


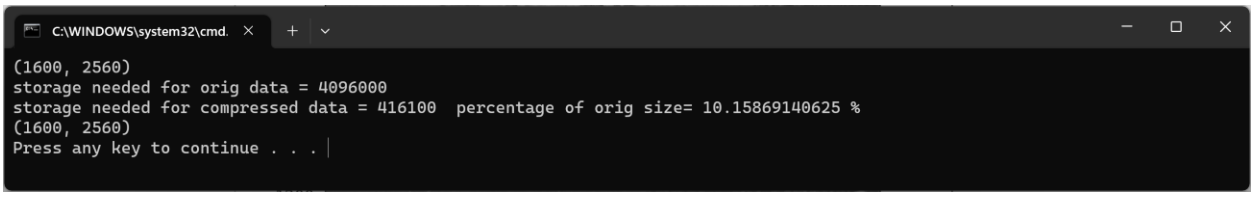
For num_components of 50, the reconstructed image appears as:





If you change the num_components to 100, the reconstructed image appears as:





Problem #3: Use the Pseudo-Inverse approach in linear least square estimation of curve fitting by a third order polynomial to a given set of x and noisy_y data points:

```
x = [0,0.5,1,1.5,2,2.5,3,3.5,4]
poly_coeffs = [2, -9, 8, 5]
ynoise = np.random.normal(0, 2, len(x))
y = poly_coeffs[0]*x**3 + poly_coeffs[1]*x**2 + poly_coeffs[2]*x + poly_coeffs[3]
noisy_y = y + ynoise
```

As you can see, noisy_y is generated by plugging the x values in the equation

$$y = 2x^3 - 9x^2 + 8x^1 + 5$$

And adding a gaussian noise with mean of 0 and standard deviation of 2.

Given the x and the noisy_y, your goal is to estimate the coefficients of the third order polynomial using the pseudo-inverse approach.

$$y = \beta_0 x^3 + \beta_1 x^2 + \beta_2 x^1 + \beta_3$$

For an n polynomial approximation where k data points with x and y values are given, the matrix equation for the n_{th} degree polynomial appears as:

$$\begin{bmatrix} 1 & x_0^1 & x_0^2 & .. & x_0^n \\ 1 & & & & \\ 1 & & & & \\ .. & .. & .. & .. & .. \\ 1 & x_k^1 & x_k^2 & .. & x_k^n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ .. \\ \beta_k \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ .. \\ y_k \end{bmatrix}$$

The above equation can be written in matrix form as:

$$A\beta = y$$

Then the solution for the β coefficients becomes:

$$\beta = A^{-1}y$$

If *A* is not square, which is usually the case, then A^{-1} is the pseudo inverse, which is computed by doing an SVD on *A*, and then the pseudo inverse is computed as:

$$A^{-1} = v^T \sum^{-1} u$$

where $\sum^{-1}$ is reciprocal of the Eigen values (as inverse of a diagonal matrix is the reciprocal of the values in the diagonal).

For example, for an approximation with the third degree polynomial where 9 sets of data points (x,y) pais are given, A will be 9x4, v^T will be 4x4, $\sum^{-1}$ will be 9x4 where the first 4 diagonals will be reciprocal of the Eigen values, and the remaining rows will contain zero values. u will be 9x9 matrix, resulting in A^{-1} being 4x9. Since y in this example is 9x1, $\beta = A^{-1}y$ will give us the four β coefficients describing the polynomial:

$$y = \beta_0 x^3 + \beta_1 x^2 + \beta_2 x^1 + \beta_3$$

that best fits the data i.e., pairs of given (x,y) points. You will see this in the following implementation as we develop the code.

Solution: Add a file called PseudoInverse.py to the project with the following code in it.

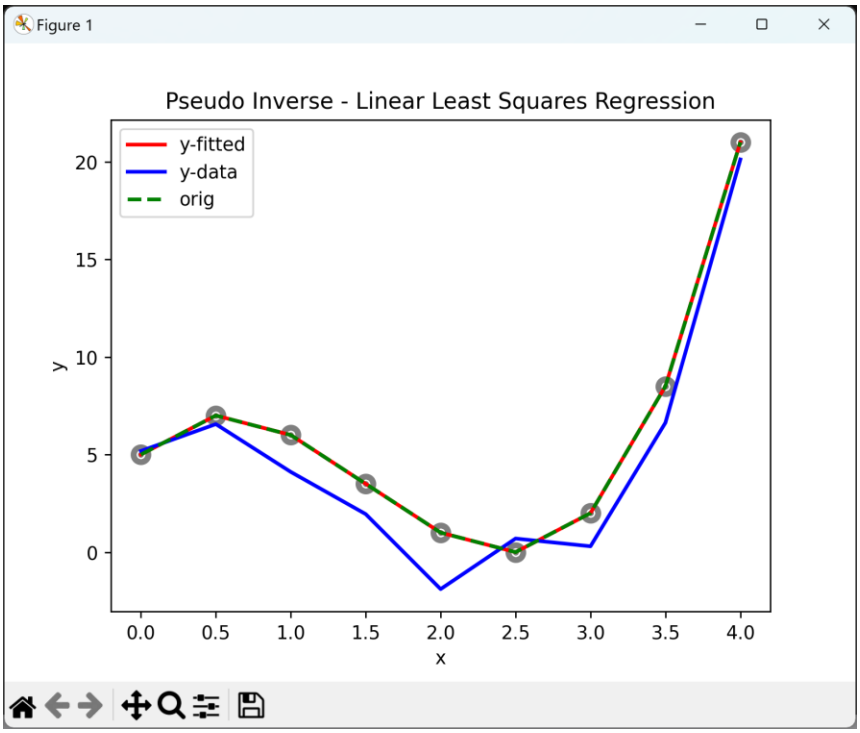
```
import sys
from tkinter.tix import Tree
import numpy as np
import matplotlib.pyplot as plt

def create_polynomial_data(x, poly_coeffs, add_noise):
    ynoise = np.random.normal(0, 2, len(x))
    #print(ynoise)
    y = poly_coeffs[0]*x**3 + poly_coeffs[1]*x**2 + poly_coeffs[2]*x +
poly_coeffs[3]
    if add_noise == True:
        noisy_y = y + ynoise
    else:
        noisy_y = y
    return y, noisy_y
```

```
def plot_data(x, y, y2, yorig):
    area = 10
    colors =['black']
    plt.scatter(x, y, s=area, c=colors, alpha=0.5, linewidths=8)
    plt.title('Pseudo Inverse - Linear Least Squares Regression')
    plt.xlabel('x')
    plt.ylabel('y')
    #plot the fitted line
    line,=plt.plot(x, y, '-', linewidth=2, label="y-fitted")
    line.set_color('red')
    line,=plt.plot(x, y2, '-', linewidth=2, label="y-data")
    line.set_color('blue')
    line,=plt.plot(x, yorig, '--', linewidth=2, label="orig")
    line.set_color('green')
    plt.legend(loc="upper left")
    plt.show()

def solve_by_pseudo_inv(x,y, poly_order):
    A = np.zeros((len(x),poly_order+1))
    for i in range(0,len(x)):
        for j in range(0,poly_order+1):
            A[i,j] = x[i]**(j)
    print(A)
    # compute pseudo inverse of A
    u, s, vt = np.linalg.svd(A)
    # print different components
    print("U: ", u)
    print("Singular Values", s)
    print("V^{T}", vt)


    #inverse of diagonalmatrix is the reciprocal of each element
    s_inv = 1.0 / s
    sz = np.zeros(A.shape)
    sz[0:len(s),0:len(s)] = np.diag(s_inv)  # pad rows of zeros
    sinv = sz
    print(sinv)
    # calculate pseudoinverse
    pseudo_inv = np.dot(np.dot(vt.T, sinv.T), u.T)
    print(pseudo_inv) # 9x4 in our example
    beta_coeffs = np.dot(pseudo_inv,y)
    return beta_coeffs

def main():
    x = [0,0.5,1,1.5,2,2.5,3,3.5,4]
    x = np.asarray(x, float)
    poly_coeffs = [2, -9, 8, 5] # 2x^3 - 9x^2 + 8x + 5
    y, noisy_y = create_polynomial_data(x,poly_coeffs, True)  # True means add noise
    print(y)
    plot_data(x, y, noisy_y, y)

    beta_coeffs = solve_by_pseudo_inv(x,noisy_y, 3)
    print('---------beta_coeffs--------')
    print(beta_coeffs)

    # np.flip reverses the coefficient array
    yfitted, noisy_y2 = create_polynomial_data(x,np.flip(beta_coeffs), False) #
false means do not add noise
```

```
    plot_data(x, yfitted, noisy_y, y) # noisy_y was the data we fitted

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Study the above code to see how the pseudo-inverse is calculated from the SVD components and used to determine the coefficients of the polynomial that fit the data. If you run the above program, the output appears as:

The following figure shows the noisy data (in blue) that we will try to fit a polynomial to.



The fitted polynomial (in red) using the pseudo-inverse technique appears as:

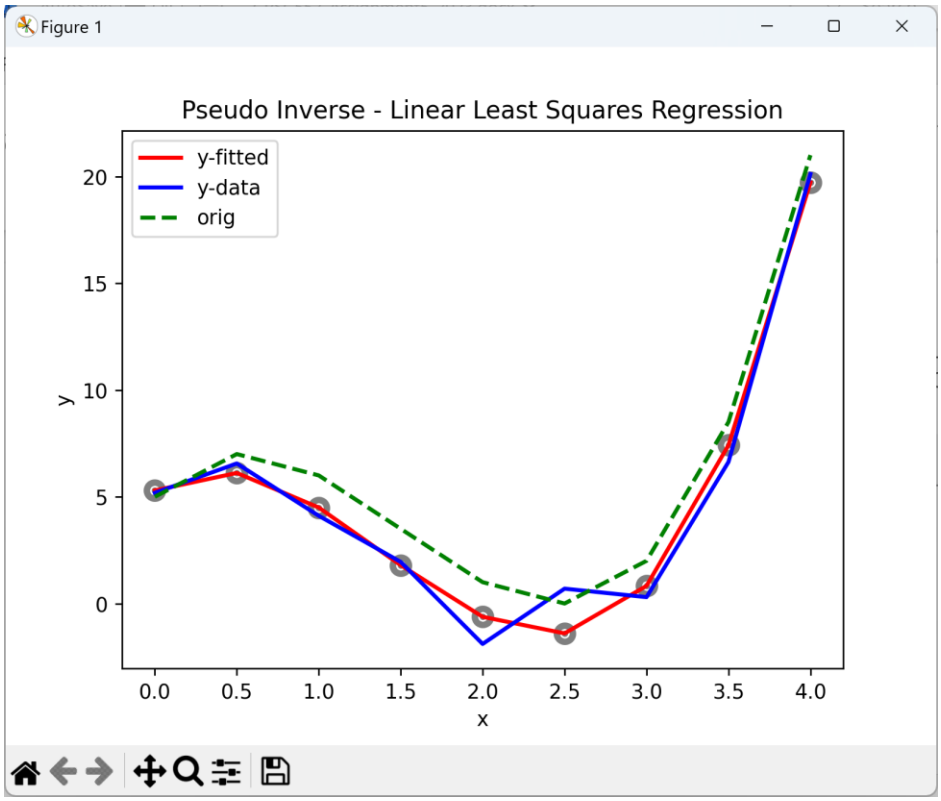


Figure 1
Pseudo Inverse - Linear Least Squares Regression
y-fitted
y-data
orig
20
15
10
5
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x