Project 2

This assessment is marked out of 100 and comprises 20% of the final course mark.

Due by 4pm on Thursday December 2nd, to be submitted on Learn.

Academic misconduct

The assessment is summative in nature. You are expected to be aware of and abide by University policies on academic misconduct.

- School of Mathematics academic misconduct advice and policies
- Academic Services academic misconduct information

This is an individual assignment - do not copy the work of another student or show your own work to another student.

If you use any resources (e.g. textbooks or websites), then you must include appropriate references in your solutions. Course materials do not need to be referenced, but you should clearly state which results you are using.

Code commentary

Your code should be extensively commented, with the functionality of each line of code explained with a comment. This is to test your understanding of the code you have written. Up to half of the marks associated with the coding part of a question may be deducted for a missing, incomplete, or inaccurate code commentary.

The following provides an example of the expected level of commenting.

In [ ]:
```
def LU(A):

    # Find dimension of A
    n = A.shape[0]

    # Initialise L=I, U=A
    L = np.eye(n)
    U = np.copy(A)

    for k in range(n - 1): # loop over columns 1 to n-1
        for j in range(k + 1, n): # loop over rows k+1 to n
            L[j, k] = U[j, k] / U[k ,k] # compute the multiplier l_jk
            U[j, k:] = U[j, k:] - L[j, k] * U[k, k:] # subtract a multiple of
                                                     # below the diagonal in

    return L, U # return the LU factorisation of A
```

Code efficiency

To obtain full marks, your code should be *efficient* in the sense of avoiding unnecessary artihmetic operations and having low computational cost.

Output

Your code must generate and display all relevant output when run. Rerun your code cells after editing your code, to make sure that the output is updated and displayed when you submit your notebook.

Re-using code and built-in functions

You can re-use your own code from previous workshops. You can also use the model solutions for workshop exercises posted on Learn, and Jupyter notebooks from lecture material posted on Learn. You do NOT need to comment re-used code, but you should clearly indicate from where you have taken the code.

You may use any built-in Python functions as required.

Written exercises

You can enter your answers to theoretical questions, i.e. the discussion in question 1.2, questions 2.1, 2.2, 2.5, 3.2 and 3.3, in the Markdown cells provided in this notebook. To start editing the cell, press shift+enter or double click on it. You can use basic Latex. To render the cell, press shift+enter or run.

Alternatively, you can submit your hand-written and scanned answers to the theoretical questions as a pdf on Learn, alongside this notebook.

To obtain full marks, you should provide a clear and justified argument written in full sentences.

Question 1

Let $\mathbf{A} \in \mathbb{R}^{\mathbf{m}\times\mathbf{n}}$, with $m \geq n$ and $rank(\mathbf{A}) = n$, and $\mathbf{b} \in \mathbb{R}^m$. Consider the following algorithm to solve the least squares problem of finding $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ that minimises $\|\mathbf{Ax} - \mathbf{b}\|_\mathbf{2}$.

Algorithm LSQ-PI

Input: $\mathbf{A} \in \mathbb{R}^{\mathbf{m}\times\mathbf{n}}$, *with* $m \geq n$ *and* $rank(\mathbf{A}) = n$, *and* $\mathbf{b} \in \mathbb{R}^{\mathbf{m}}$

Output: $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ *that minimises* $\|\mathbf{Ax} - \mathbf{b}\|_\mathbf{2}$

1. Compute the pseudo-inverse $\mathbf{A}^\dagger$
2. Compute $\mathbf{x} = \mathbf{A}^\dagger\mathbf{b}$ using Algorithm MV (matrix-vector multiplication)

In lectures, we defined the pseudo-inverse $\mathbf{A}^\dagger = (\mathbf{A}^\mathrm{T}\mathbf{A})^{-\mathbf{1}}\mathbf{A}^\mathrm{T}$.

Let $\mathbf{A} = \mathbf{U\Sigma V}^\mathrm{T}$ be a singular value decomposition. In Workshop week 10, we will see that $\mathbf{A}^\dagger = \mathbf{V\Sigma}^\dagger\mathbf{U}^\mathrm{T}$, where $\mathbf{\Sigma}^\dagger \in \mathbb{R}^{n\times m}$ is obtained by taking the reciprocal of the diagonal elements of $\mathbf{\Sigma}$, and then transposing the matrix. Under the assumptions in this question, this is equivalent to $\mathbf{A}^\dagger = \mathbf{V\Sigma}_\mathbf{1}^{-\mathbf{1}}\mathbf{U_1}^\mathrm{T}$, where $\mathbf{A} = \mathbf{U_1\Sigma_1V}^\mathrm{T}$ is a reduced singular value decomposition.

So we can consider two different versions of Algorithm LSQ-PI:

1. Computing the pseudo-inverse in step 1 as $\mathbf{A}^\dagger = (\mathbf{A}^\mathrm{T}\mathbf{A})^{-1}\mathbf{A}^\mathrm{T}$, i.e. by finding $(\mathbf{A}^\mathrm{T}\mathbf{A})^{-1}$ and multiplying it by $\mathbf{A}^\mathrm{T}$.
2. Computing the pseudo-inverse in step 1 as $\mathbf{A}^\dagger = \mathbf{V}\mathbf{\Sigma}_1^{-1}\mathbf{U_1}^\mathrm{T}$.

Question 1.1

Implement the two versions of Algorithm LSQ-PI in the code cell below. The inputs to your functions should be $\mathbf{A}$ and $\mathbf{b}$, and the output should be the minimiser $\mathbf{x}$.Test your code on the example given at the bottom of the code cell, to which the solution is $x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

[12 marks]

In [2]:
```
import numpy as np

def LSQ_PI_1(A, b):
    ''' Uses A transpose A inverse and A transpose
    to find the pseudo inverse of A to return the minimum x value of ||Ax - b
    ------------------
    Inputs: A = mxn matrix m>=n
    b = m dimensional vector
    Outputs: x = n dimensional vector that minimises ||Ax - b||2.
    '''
    AT = A.T # Find transpose of A
    pA_inv = np.linalg.solve(AT@A, AT) # Calculate pseudo inverse of A
    x = pA_inv@b # Find x

    return x

def LSQ_PI_2(A, b):
    ''' Uses the SVD of A to find the pseudo inverse of A
     to return the minimum x value of ||Ax - b||2.
    ------------------
    Inputs: A = mxn matrix m>=n
    b = m dimensional vector
    Outputs: x = n dimensional vector that minimises ||Ax - b||2.
    '''
    U, Sig, VT = np.linalg.svd(A, full_matrices = False) # Find reduced SVD d

    Sig_inv = np.ones(len(Sig))/Sig # Find the diagonal values of sigma inver
    pA_inv = VT.T@np.diag(Sig_inv)@U.T # Matrix multiply V with sigma inverse

    x = pA_inv@b # Find x
    return x

# Test your code on a small example
b = np.array([0., 1., 2.])
A = np.array([[1., -0.5],[1., 0.],[1., 0.5]])
x1 = LSQ_PI_1(A, b)
x2 = LSQ_PI_2(A, b)
print(x1)
print(x2)
```

```
[1. 2.]
[1. 2.]
```

Question 1.2

Investigate the performance of the two versions of Algorithm LSQ-PI in terms of the error in the computed solution as a function of n (where $\mathbf{A} \in \mathbb{R}^{2n\times n}$) and $\kappa_2(\mathbf{A})$.

Include any tests that you run in the code cell below and display your results. Do you notice any trends in the results? How do you explain the difference in accuracy of the two versions of the algorithm?

[15 marks]

In [3]:

```
import numpy as np
def randsvd(m, n, kappa):
    '''Taken from week 7 jupyter notebook for lectures.
    Generates a random mxn matrix with kappa value kappa.'''
    s = np.zeros(n)
    S = np.zeros((m,n))
    for i in range(n):
        beta = kappa**(1/(n-1))
        s[i] = beta**(-i)
    S[:n,:n] = np.diag(s)

    def haar(n):
        A = np.random.randn(n, n)
        Q, R = np.linalg.qr(A)

        for i in range(n):
            if R[i, i] < 0:
                Q[:, i] *= -1
        return Q

    A = haar(m) @ S @ haar(n).T
    return A

# Define number of kappa and n values to test
a = 20
c = 9

# Initialise arrays of kappa, kappa squared, 2-norm of x and n values
kappa = np.zeros(a)
norm_xsol = np.zeros(c)
n_vals = np.array(np.zeros(c), dtype = int)

for k in range(a): #Generate range of kappa values
    kappa[k] = (4*k + 2)**5

for i in range(c): #Generate range of n values
    n_vals[i] = int((i + 4)**3)

for n in range(c): # Caluclate 2-norms of x for each n
    xsol = np.ones(n_vals[n])
    norm_xsol[n] = np.linalg.norm(xsol, 2)
```

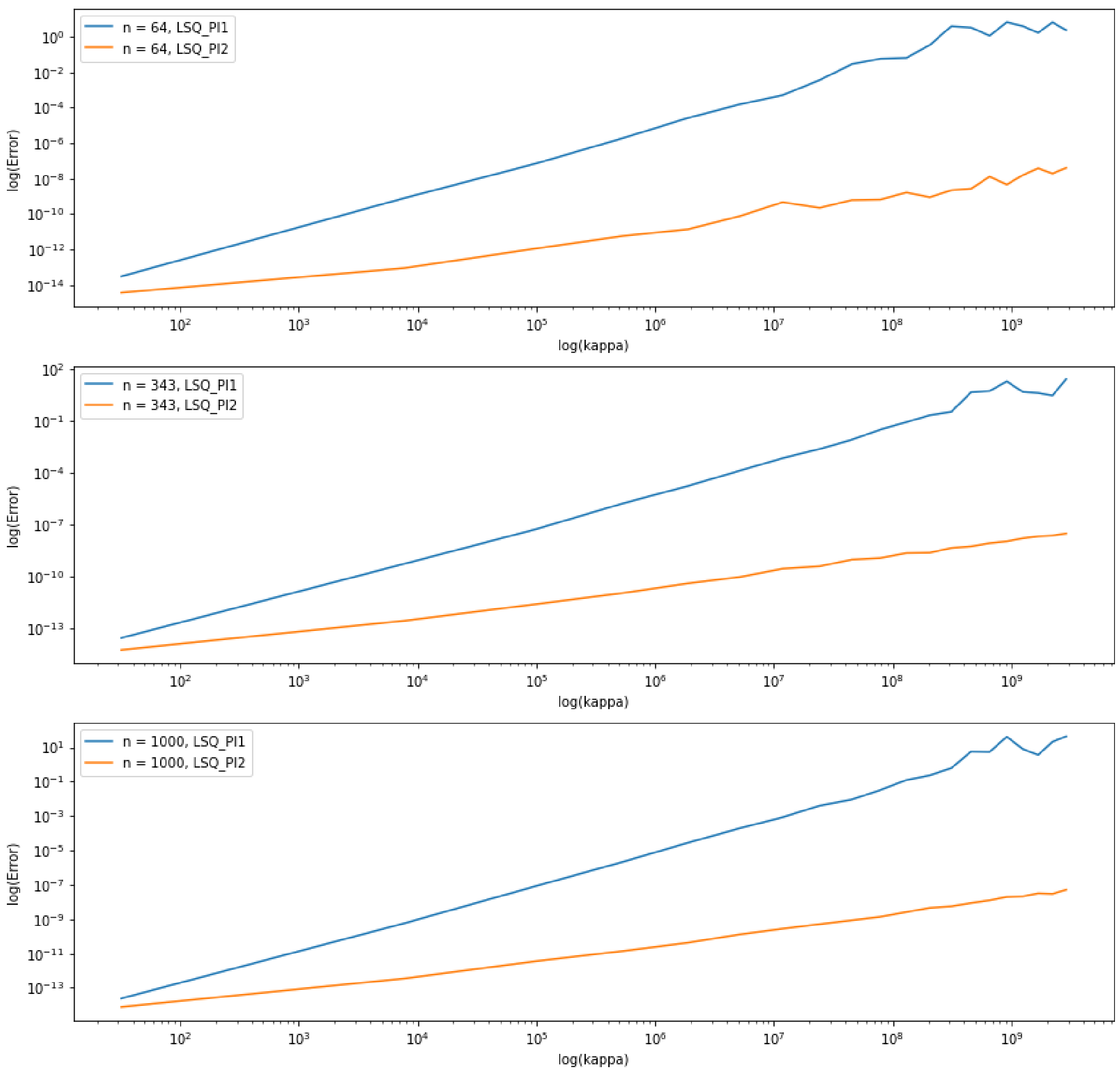
In [4]:

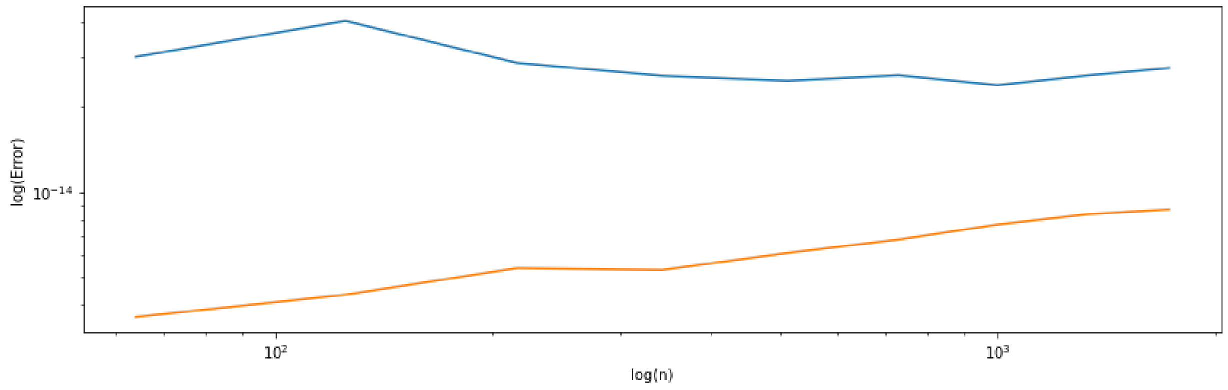
```
import numpy as np
import matplotlib.pyplot as plt
# Initialise matrices of errors for each method
errors1 = np.zeros((a, c))
errors2 = np.zeros((a, c))

for k in range(a): # Loop over kappa values
    for i in range(c): #Loop over 3 2nxn matrices
        n = n_vals[i]
        # Call n values and generate random matrix
        A = randsvd(2*n, n, kappa[k])

        # Define solution
        xsol = np.ones(n)
        # Calculate input b
        b = A@xsol

        # Find approximate solution
        x1 = LSQ_PI_1(A, b)
        x2 = LSQ_PI_2(A, b)

        # Save normalised errors in matrices
        errors1[k][i] = np.linalg.norm(xsol-x1,2)/norm_xsol[i]
        errors2[k][i] = np.linalg.norm(xsol-x2,2)/norm_xsol[i]
```

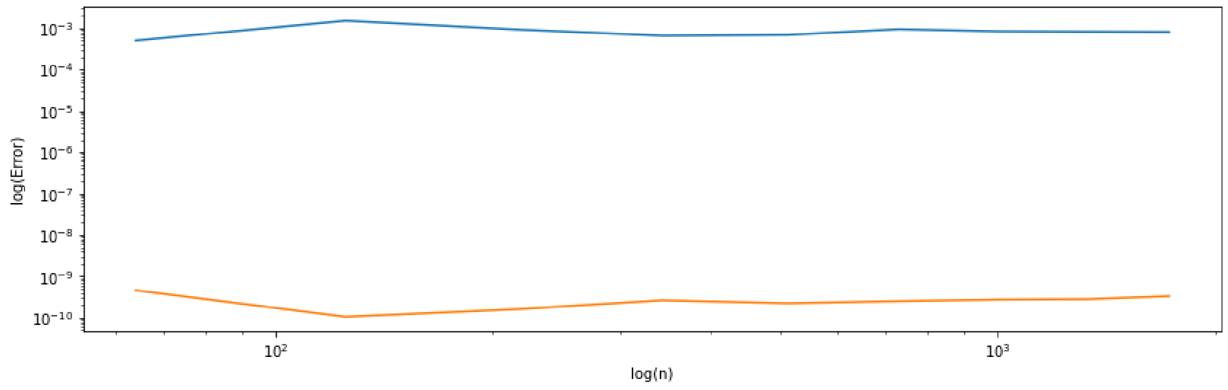
In [11]:

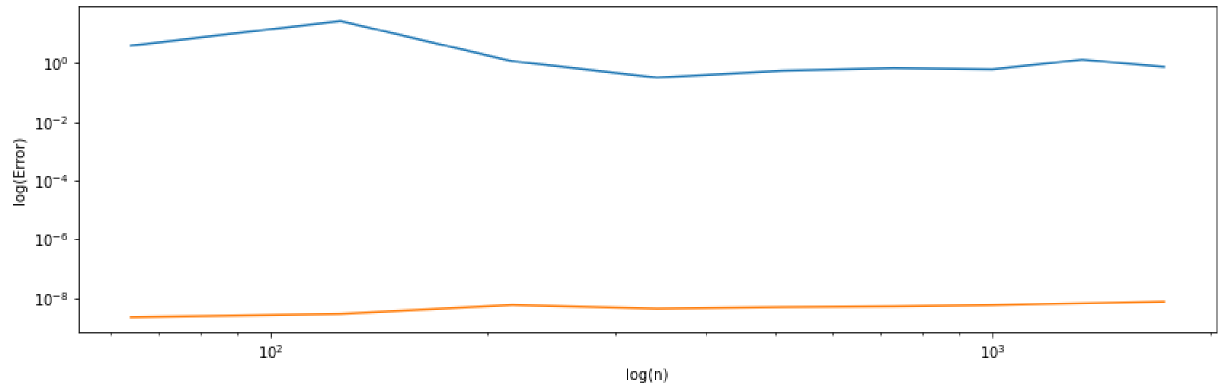
```
#Set up figure for plots of errors
fig, ax = plt.subplots(3, figsize = (14, 14))
fig2, ax2 = plt.subplots(3, figsize = (14, 14))
fig.suptitle('Error comparison LSQ SVD and pseudo inverse')

for i in range(3): # Loop over each n, plotting errors against condition numb
    ax[i].plot(kappa, errors1[:,3*i ], label = f'n = {n_vals[3*i]}, LSQ_PI1')
    ax[i].plot(kappa, errors2[:,3*i ], label = f'n = {n_vals[3*i]}, LSQ_PI2')
    ax[i].set_xscale('log')
    ax[i].set_yscale('log')
    ax[i].set_xlabel('log(kappa)')
    ax[i].set_ylabel('log(Error)')

    ax2[i].plot(n_vals, errors1[6*i,:], label = f'k = {kappa[i]}, LSQ_PI1')
    ax2[i].plot(n_vals, errors2[6*i,:], label = f'k = {kappa[i]}, LSQ_PI2')
    ax2[i].set_xscale('log')
    ax2[i].set_yscale('log')
    ax2[i].set_xlabel('log(n)')
    ax2[i].set_ylabel('log(Error)')

    ax[i].legend()
plt.show()
```

Error comparison LSQ SVD and pseudo inverse









Answer to Q1.2 See attached PDF.

Question 2

This question concerns polynomial regression. Suppose we have a set of m observation points $\{a_i, b_i\}_{i=1,\dots,m}$. As in the lecture notes, we can fit a polynomial of degree $n-1$ to the data by choosing the coefficients $\mathbf{x}^{\mathrm{LS}} \in \mathbb{R}^n$ to minimise $\|\mathbf{Ax} - \mathbf{b}\|_2^2$, where $\mathbf{A} \in \mathbb{R}^{m\times n}$ is the Vandermonde matrix with entries $a_{ij} = (a_i)^{j-1}, i = 1,\dots,m, j = 1,\dots,n$, and $\mathbf{b} = [b_1, b_2, \dots, b_m]^T$.

As we saw in Exercise 1 in Computer lab week 9, the polynomials fitted in this way often lead to *overfitting*:

A more complex model is not necessarily a better model. Overfitting occurs when a model is complex enough to fit very closely to a particular set of data points, but does not generalise well, and will not be suitable to predict future observations. Instead of capturing the general relationship between input and output in a dataset, an overfitted model describes the noise in the observations.

A method to avoid overfitting is *regularisation*. Instead of minimising $\|\mathbf{Ax}-\mathbf{b}\|_2^2$, we seek to minimise the new cost function

$$\|\mathbf{Ax}-\mathbf{b}\|_2^2+\mu\|\mathbf{x}\|_2^2,$$

where $\mu>0$ is a regularisation parameter. The idea is to introduce a penalty term, so that the polynomial coefficients $\mathbf{x}$ (particularly the high-order coefficients) remain relatively small.

Denote by $\mathbf{x}^{\mathrm{regLS}}$ the minimiser of $\|\mathbf{Ax}-\mathbf{b}\|_2^2+\mu\|\mathbf{x}\|_2^2$.

Question 2.1

Show that $\mathbf{x}^{\mathrm{regLS}}$ solves the regularised normal equations

$$(\mathbf{A}^{\mathsf{T}}\mathbf{A}+\mu\mathbf{I})\mathbf{x}^{\mathrm{regLS}}=\mathbf{A}^{\mathsf{T}}\mathbf{b}.$$

[6 marks]

Answer to Q2.1

See attached PDF of written answers.

Question 2.2

Let $\mathbf{A}=\mathbf{U_1}\mathbf{\Sigma_1}\mathbf{V}^{\mathsf{T}}$ be the reduced SVD of $\mathbf{A}$. Using Question 2.1, or otherwise, show that $\mathbf{x}^{\mathrm{regLS}}$ satisfies

$$\mathbf{x}^{\mathrm{regLS}}=\mathbf{VSU_1}^{\mathsf{T}}\mathbf{b},$$

where the matrix $\mathbf{S}\in\mathbb{R}^{n\times n}$ is diagonal with diagonal entries given by

$$s_{ii}=\frac{\sigma_i}{\sigma_i^2+\mu},\qquad i=1,\dots,n,$$

with σ_i the singular values of $\mathbf{A}$.

[10 marks]

Answer to Q2.2

See PDF.

Question 2.3

Implement a Python function called LSQ_SVD_reg, which computes $\mathbf{x}^{\mathrm{regLS}}$ using the expression found in Question 2.2. The inputs to your function should be $\mathbf{A}$, $\mathbf{b}$ and μ, and the output should be the minimiser $\mathbf{x}^{\mathrm{regLS}}$. Test your code on the example given at the bottom of the code cell, to which the solution is $x=\begin{bmatrix}0.6\\0.4\end{bmatrix}$.

[8 marks]

```
In [35]: import numpy as np
```

```
def LSQ_SVD_reg(A, b, mu):
    '''Returns a regularised least squares value of x.
    -------------------------------
    Inputs: A = mxn vandermonde matrix
    b = array of observed values for each data point
    mu = regularisation constant.
    Outputs: x = least squares estimate, array of coefficients of
    n-1 degree polynomial, approximating the data.'''
    # Reduced singular value decomposition
    U, Sig, VT = np.linalg.svd(A, full_matrices = False)

    # Redefine singular values using regularisation constant
    Sig_mu = (mu*np.ones(len(Sig)) + Sig*Sig)
    S = np.diag(Sig/Sig_mu)

    # Use expression derives in Q2.2 to find x
    x = VT.T@S@U.T@b
    return x

# Test your code on a small example
b = np.array([0., 1., 2.])
A = np.array([[1., -0.5],[1., 0.],[1., 0.5]])
mu = 2.0
x = LSQ_SVD_reg(A, b,mu)
print(x)
```

```
[0.6 0.4]
```

Question 2.4

We now want to test the effect of regularisation on a particular observed data set. Implement the following in Python in the code cell below, using the code from Exercise 1 in Computer lab week 9 (or otherwise):

(i) Download the file *data_points_30.npy* from Learn, and load it into an array using np.load(). This is a 30x2 array containing a set of 30 data points $\{a_i, b_i\}_{i=1,...,30}$ with the a_i in the first column and the b_i in the second column. Plot the data points.

(ii) Set up the Vandermonde matrix $\mathbf{A}$ of size $30 \times n$ for $n = 2, 4, 6, 8, 10$, and compute the (un-regularised) polynomial coefficients $\mathbf{x}^{\text{LS}}$ for the polynomial of degree $n-1$. On the same graph as the data points, plot your fitted polynomials over the interval $[0, 6]$.

(ii) Use your function *LSQ_SVD_reg* from Question 2.3 to compute the the (regularised) polynomial coefficients $\mathbf{x}^{\text{LSreg}}$ for the polynomial of degree $n-1$ for $n = 2, 4, 6, 8, 10$, for $\mu = 5, 0.5, 0.01$, and 10^{-5}. For each value of μ, produce the same plot as in (i)-(ii).

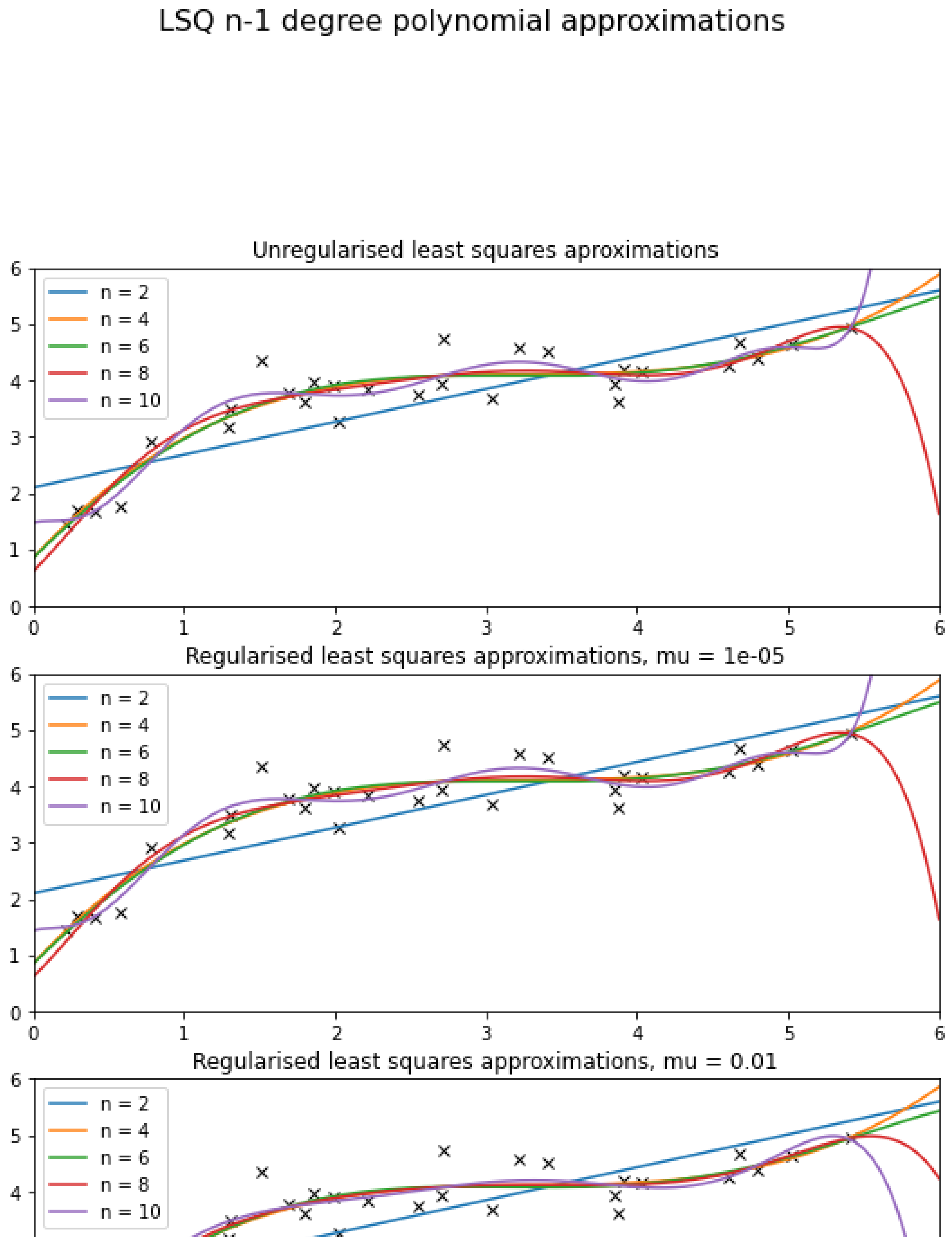
You should end up with 5 plots showing the data points in *data_points_30.npy* and the fitted polynomials for $n = 2, 4, 6, 8, 10$: 1 for the standard minimiser $\mathbf{x}^{\text{LS}}$, and 4 for the regularised minimiser $\mathbf{x}^{\text{LSreg}}$ (1 for each of the 4 values of μ).

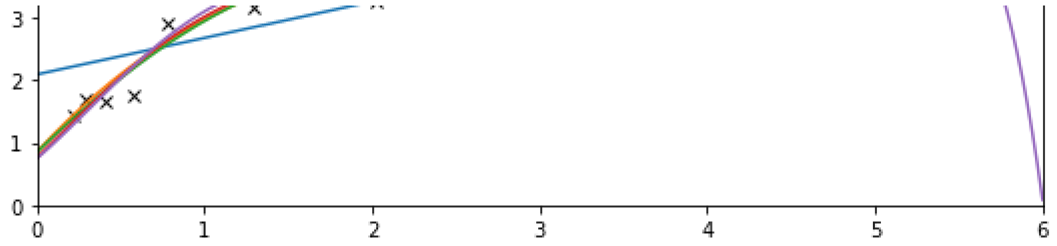
[8 marks]

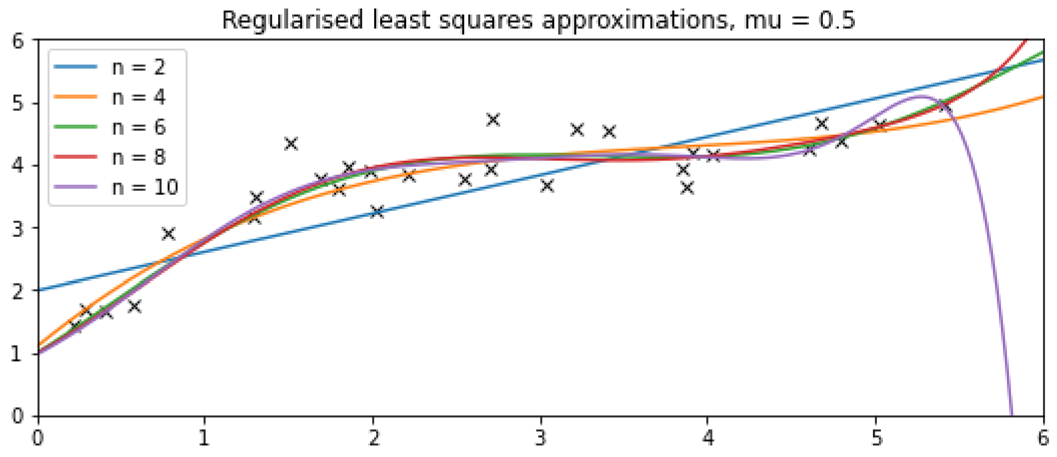
In [36]:
```
# Load the data
M = np.load('data_points_30.npy copy')
a = M[:, 0]
b = M[:, 1]
m = M.shape[0]
```

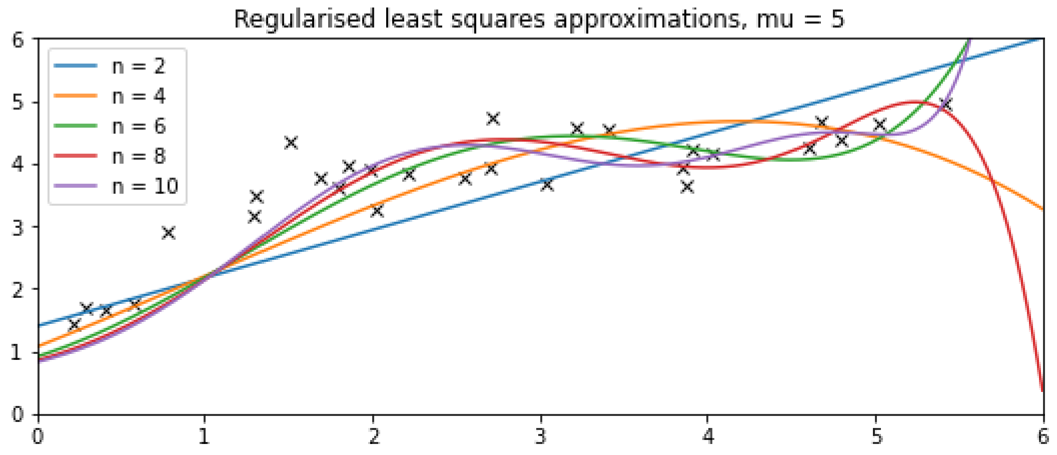
```
# Initialise plot
fig, ax = plt.subplots(5, figsize=(9, 20))
fig.suptitle('LSQ n-1 degree polynomial approximations', fontsize=16)
x_plot = np.linspace(0, 6, 200)

mu_vals = [0, 10**(-5), 0.01 ,0.5, 5 ]
# mu = 0 is equivalent to LSQ_SVD (unregularised)

for r in range(len(mu_vals)): # Loop over mu value, generating plot for each
    ax[r].plot(a, b, 'kx')
    print(f'\n Polynomial coefficients for mu = {mu_vals[r]}')

    for n in range(1, 6):# Plot all n-1 degree polynomials for each mu
        # Form Vandermonde matrix
        A = np.array([[i**k for k in range(2*n)] for i in a], dtype=float)

        # Compute the minimiser x
        x = LSQ_SVD_reg(A, b, mu_vals[r])
        print(f' n = {2*n}: {x}')
        # Plot the polynomials
        p_plot = np.polyval(x[::-1], x_plot)
        ax[r].plot(x_plot, p_plot, label='n = {}'.format(2*n))
        ax[r].set_xlim([0, 6])
        ax[r].set_ylim([0, 6])
        ax[r].legend()

        # Generate appropriate plot titles
        if r != 0:
            ax[r].set_title(f'Regularised least squares approximations, mu =

        else:
            ax[r].set_title('Unregularised least squares aproximations')
plt.show()
```

```
 Polynomial coefficients for mu = 0
 n = 2: [2.09693807 0.58331644]
 n = 4: [ 0.84309863  2.92031637 -0.87568152  0.0881553 ]
 n = 6: [ 0.84054741  2.65389791 -0.44164567 -0.1405294   0.04841341 -0.003570
6 ]
 n = 8: [ 0.61044909  2.57265399  2.2100051  -4.04396842  2.34849783 -0.662675
02
  0.0915045  -0.00493114]
 n = 10: [ 1.47134351e+00  7.67004161e-01 -6.55509717e+00  2.37976912e+01
 -2.89047046e+01  1.73823070e+01 -5.82168375e+00  1.10616538e+00
 -1.11621110e-01  4.64938000e-03]

 Polynomial coefficients for mu = 1e-05
 n = 2: [2.09693577 0.58331707]
 n = 4: [ 0.84310716  2.92029491 -0.87567164  0.0881541 ]
 n = 6: [ 0.8405557   2.65380922 -0.44153343 -0.14058129  0.04842354 -0.003571
3 ]
 n = 8: [ 0.6108555   2.57286337  2.20526322 -4.0371014   2.34447447 -0.661524
16
  0.0913447  -0.00492252]
 n = 10: [ 1.43357430e+00  8.38982070e-01 -6.16586799e+00  2.26176603e+01
 -2.76058079e+01  1.66407719e+01 -5.58032554e+00  1.06098975e+00
 -1.07091654e-01  4.46083494e-03]

 Polynomial coefficients for mu = 0.01
 n = 2: [2.09464278 0.58395089]
 n = 4: [ 0.85152806  2.89907226 -0.86589339  0.08697001]
 n = 6: [ 0.84841147  2.57205854 -0.33904908 -0.18764478  0.05756094 -0.004206
62]
 n = 8: [ 7.60450894e-01  2.58925389e+00  5.87925205e-01 -1.61174327e+00
  9.03819182e-01 -2.46272309e-01  3.33987892e-02 -1.78498291e-03]
 n = 10: [ 7.44649948e-01  2.33700306e+00  8.83195562e-01 -4.27043918e-01
 -1.26381040e+00  1.20029838e+00 -4.52195371e-01  8.54726124e-02
 -7.96452261e-03  2.87591190e-04]

 Polynomial coefficients for mu = 0.5
 n = 2: [1.98896126 0.6130424 ]
 n = 4: [ 1.09483682  2.20688381 -0.53881687  0.0469427 ]
 n = 6: [ 0.99489392  1.6422819   0.48675419 -0.4521282   0.09166274 -0.005621
74]
 n = 8: [ 9.89590361e-01  1.47862341e+00  4.87455095e-01 -3.58746082e-02
 -2.77939266e-01  1.21377653e-01 -1.93933267e-02  1.09988617e-03]
 n = 10: [ 9.68398351e-01  1.43696192e+00  5.60220250e-01  1.17513215e-01
 -2.66405619e-01 -1.56870251e-01  1.92858489e-01 -6.51019720e-02
  9.49130813e-03 -5.17233623e-04]

 Polynomial coefficients for mu = 5
 n = 2: [1.39649112 0.7691144 ]
 n = 4: [ 1.06688903  1.05133297  0.11307236 -0.03784386]
 n = 6: [ 0.90964762  0.85366176  0.49784544 -0.04841596 -0.05141277  0.007975
56]
 n = 8: [ 0.8499622   0.73215868  0.4859245   0.19347141 -0.11192586 -0.029846
52
  0.01530359 -0.00143253]
 n = 10: [ 8.22075398e-01  6.77284922e-01  4.64365033e-01  2.61193851e-01
  6.56047229e-03 -1.27535229e-01  1.58809081e-02  1.19265319e-02
 -3.34068911e-03  2.43456596e-04]
```









Question 2.5

Discuss your plots in Question 2.4. How does regularisation affect the fitted polynomials in terms of their fit to the data and their usefulness for extrapolation? How does regularisation affect the coefficient values? What happens when μ is too large or too small, and can you explain why this happens? Which value of μ produces the best results in your opinion?

[15 marks]

Answer to Q2.5

See PDF.

Question 3

Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be symmetric. Consider the following extension of Algorithm SII (Shifted Inverse Iteration), known as Rayleigh Quotient Iteration.

Algorithm RQI:

Input: $\mathbf{A} \in \mathbb{R}^{n\times n}$ symmetric, $\mathbf{z}^{(0)} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$, $\varepsilon > 0$

Output: $\mathbf{z}^{(k)} \in \mathbb{R}^n$, $\lambda^{(k)} \in \mathbb{R}$, with $\mathbf{z}^{(k)} \approx \mathbf{x}_j$ and $\lambda^{(k)} \approx \lambda_j$ for some $j \in \{1, \ldots, n\}$.

```
1. $k = 0$
2. $\lambda^{(0)} = r_{\mathbf{A}}(\mathbf{z}^{(0)})$
3. While $\|\mathbf{A}\mathbf{z}^{(k)} - \lambda^{(k)}\mathbf{z}^{(k)}\|_2 > \varepsilon$
4.     $k = k + 1$
5.     Solve $\left(\mathbf{A} - \lambda^{(k-1)}\mathbf{I}\right)\mathbf{z}^{(k)} = \mathbf{z}^{(k-1)}$
6.     $\mathbf{z}^{(k)} = \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|_2}$
7.     $\lambda^{(k)} = r_{\mathbf{A}}(\mathbf{z}^{(k)})$
8. End While
```

The main idea of Rayleigh quotient iteration is to replace the shift s in ALgorithm SII with the estimate $\lambda^{(k-1)}$ of the eigenvalue to speed up convergence. If Rayleigh quotient iteration converges, it converges to an eigenvector and eigenvalue of $\mathbf{A}$, but which eigenvector/-value it converges to will depend on the starting guess $\mathbf{z}^{(0)}$.

In line 5 of Algorithm RQI, suppose we want to solve the system using Algorithm GEPP (Gaussian elimination with partial pivoting).

Question 3.1

In the code cell below, write a function *RQI* that implements Algorithm RQI following the pseudo-code above.

Test your code on the example given at the bottom of the code cell, for which the outputs should converge to $\mathbf{x}_2 = -\frac{1}{\sqrt{10}}\begin{bmatrix}3\\1\end{bmatrix} \approx \begin{bmatrix}-0.9486833\\-0.31622776\end{bmatrix}$ and $\lambda_2 = 1$.

[10 marks]

In [37]:
```python
import numpy as np

def RQI(A,z0,eps):
    '''Uses Rayleigh Quotient Iteration to find an eigenvalue and eigenvector
    that the intial guess z0 is closest to within a tolerance of eps.
    '''
    def lamk(zk):
        '''Function which finds the rayleigh quotient of zk'''
        return zk.T@A@zk/(zk.T@zk)

    # Initilise values for iteration
    zk = z0
    lk = lamk(z0)
    norm = np.linalg.norm(A@zk - lk*zk, 2)

    while norm > eps: # Loop until tolerance in convergence is acheived
        # compute next iteration for the eigenvector
        zk_plus = np.linalg.solve(A - lk*np.eye(np.shape(A)[0]), zk) #np.lina
        # Normalise eigenvector guess
        zk = zk_plus/np.linalg.norm(zk_plus, 2)
        # Compute next iteration for the eigenvalue
        lk = lamk(zk)
        # Check convergence to compare against eps in next iteration
        norm = np.linalg.norm(A@zk - lk*zk, 2)

    return lk, zk

# Small test example
A = np.array([[2, -3], [-3,10]], dtype=float)
z0 = np.array([1,1], dtype=float)
eps = 10**(-3)

lk, zk = RQI(A,z0,eps)
print(lk)
print(zk)
```

```
1.0000000000000007
[-0.9486833  -0.31622776]
```

Question 3.2

For the particular $\mathbf{A}$ given in the example in Question 3.1, derive an expression for the condition number $\kappa_2(\mathbf{A} - \lambda^{(k)}\mathbf{I})$ in terms of $\lambda^{(k)}$ and the eigenvalues of $\mathbf{A}$.

You may use without proof that the eigenvalues of $\mathbf{A}$ *are* $\lambda_1 = 11$ *and* $\lambda_2 = 1$.

[6 marks]

Answer to Q3.2

See PDF.

Question 3.3

For the particular $\mathbf{A}$ given in the example in Question 3.1, what happens to the condition number $\kappa_2(\mathbf{A} - \lambda^{(k)}\mathbf{I})$ as $k \to \infty$?

What implications does the behaviour of $\kappa_2(\mathbf{A} - \lambda^{(k)}\mathbf{I})$ as $k \to \infty$ have when Algorithm RQI is implemented in floating point arithmetic?

Do you think you would see the same behaviour of $\kappa_2(\mathbf{A} - \lambda^{(k)}\mathbf{I})$ as $k \to \infty$ for a general symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$? Explain why or why not.

[10 marks]

Answer to Q3.3

See PDF.