Optimisation Coursework 1

By Sahil Singh

Note: all code is deferred to the appendix.

Question 1: Mean Squared Error (MSE)

The bivariate linear regression is the objective function, which is the mean squared error, expressed as, $J(w_1, w_2, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(h\left(x_1^{(i)}, x_2^{(i)}\right) - y^{(i)}\right)^2$. Where $h\left(x_1^{(i)}, x_2^{(i)}\right) = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b$, $y^{(i)}$ is the measurement for the given point for $\left(x_1^{(i)}, x_2^{(i)}\right)$.

Normal Equation:

The linear least squares is $\min_{\substack{\boldsymbol{w}\in\mathbb{R}^{\mathbf{n}} \\ \mathrm{b}\in\mathbb{R}}}\{J(\boldsymbol{w}, b) = \mathbf{w}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{w} - 2\mathbf{b}^{\mathrm{T}}\mathbf{A}\mathbf{w} + ||\mathbf{b}||^2\}$ so, when $\boldsymbol{\nabla} J(\boldsymbol{x}) = 0$ you get $(\mathbf{A}^{\mathrm{T}}\mathbf{A})\,\mathbf{x}_{\{\mathrm{LS}\}} = \mathbf{A}^{\mathrm{T}}\mathbf{b}$ which is our normal equation. $(\mathbf{A}^{\mathrm{T}}\mathbf{A})$ is invertible $\Leftrightarrow (\mathbf{A}^{\mathrm{T}}\mathbf{A})$ is positive definite. Then you have the global minimum such that $\mathbf{x}_{\{\mathrm{LS}\}} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b}$. However, if $(\mathbf{A}^{\mathrm{T}}\mathbf{A})$ is not invertible then you have $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is symmetric where it has r strictly positive eigenvalues. Where the remaining $n - r$ eigenvalues equal to 0. So diagonalising it results in $(\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1} = \mathbf{U}\mathbf{D}^{-1}\mathbf{U}^{\mathrm{T}} = \sum_{i=1}^{r}\frac{(\boldsymbol{u}_i\boldsymbol{u}_i^T)}{d_i}$, where the $\mathbf{U}$ is the $(n \times n)$ orthogonal matrix where its columns are eigenvectors of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ and the $D^{-1} := \mathrm{diag}\left(\frac{1}{d_1}, \ldots, \frac{1}{d_r}, 0, 0, \ldots\right)$ the inverse of the diagonal matrix . Used to solve the normal equation. Which is $\mathbf{x}_{\{\mathrm{LS}\}} = \mathbf{U}\mathbf{D}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{b}$. The normal for the objective function is $\boldsymbol{\nabla} J(\boldsymbol{w}, b) = 0$. The explicit form:

- $\frac{\partial J}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m}\left(h\left(x_1^{(i)}, x_2^{(i)}\right) - y^{(i)}\right)x_1^{(i)}$
- $\frac{\partial J}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m}\left(h\left(x_1^{(i)}, x_2^{(i)}\right) - y^{(i)}\right)x_2^{(i)}$
- $\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(h\left(x_1^{(i)}, x_2^{(i)}\right) - y^{(i)}\right)$

The Gradient Descent:

The objective is to find the optimal solution of $\min\{\mathrm{J}(\mathbf{w}, \mathrm{b}): \mathbf{x} \in \mathbb{R}^{\mathrm{n}}\}$ where $\mathbf{w}^{\mathrm{k+1}} = \mathbf{w}^{\mathrm{k}} + \mathrm{t}^{\mathrm{k}}\boldsymbol{d}^k$ where k is a positive integer. Our descent direction $\boldsymbol{d}^k = -\boldsymbol{\nabla} J(\boldsymbol{w}, b)$ if it doesn't equal to zero and t^{k} is the time step. Where $\mathbf{w} = \mathbf{w}(\mathrm{w}_1, \mathrm{w}_2, \mathrm{b})$ optimally is chosen to be the steepest descent direction. The explicit form:

- $w_1^{k+1} = w_1^k - t^k\frac{\partial J}{\partial w_1}$
- $w_2^{k+1} = w_2^k - t^k\frac{\partial J}{\partial w_2}$
- $b^{k+1} = b^k - t^k\frac{\partial J}{\partial b}$

Newton's Methods:

For the Newtons method when finding the stationary point $\boldsymbol{w}^*$and b^* satisfying $\boldsymbol{\nabla} J(\boldsymbol{w}^*, b^*) = 0$ since $J(\boldsymbol{w}^*, b^*)$ is twice continuously differentiable over $\mathbb{R}^{\mathrm{n}}$ and we can compute the Hessian $H = \boldsymbol{\nabla}^2 J(\boldsymbol{w}, b)$ such that $\mathbf{w}^{\mathrm{k+1}} = \mathbf{w}^{\mathrm{k}} - \left(\boldsymbol{\nabla}^2 J(\mathbf{w}^{\mathrm{k}}, b^k)\right)^{-1}\boldsymbol{\nabla} J(\mathbf{w}^{\mathrm{k}}, b^k)$, the solution is only well defined for the Hessian being positive definite at $(\mathbf{w}^{\mathrm{k}}, b^k)$. The Hessian expressed in the explicit form is:

- $\boldsymbol{H} = \frac{1}{m}\begin{pmatrix} \sum_{i=0}^{m}\left(x_1^i\right)^2 & \sum_{i=0}^{m} x_1^i\, x_2^i & \sum_{i=0}^{m} x_1^i \\ \sum_{i=0}^{m} x_1^i\, x_2^i & \sum_{i=0}^{m}\left(x_2^i\right)^2 & \sum_{i=0}^{m} x_2^i \\ \sum_{i=0}^{m} x_1^i & \sum_{i=0}^{m} x_2^i & m \end{pmatrix}$

The Hessian is invertible since proposition 2 from lecture notes 2 page 5 tells you that if the Hessian is 3×3 symmetric matrix. Then the Hessian is positive definite if and only if $\mathrm{D}_1(\mathrm{H}) > 0, \mathrm{D}_2(\mathrm{H}) > 0, \mathrm{D}_3(\mathrm{H}) > 0$. For Newton's method to work it would require Hessian to be positive definite. Also, you can use the fact if the Rank(H) is identical to the number of rows/columns then the Hessian is invertible. Which is what I presented in the code below in the appendix. As it can also be seen there is no linear combination of row 1 and row 2 to get row 3 and vice versa. Since it is a 3×3 matrix can use the conventional method.

- $\boldsymbol{H^{-1}} = \frac{\boldsymbol{m}}{\mathbf{det}(\boldsymbol{H})}\boldsymbol{Adj}(\boldsymbol{H})$

When implementing the gradient descent method, using the data from the CVS file where the fixed number of iterations equal to 100, the initial guess $\mathbf{x}^0 = (0,0,0)$, using the step size of $t = 0.1$ we get the output **w** = $(0.552, 0.832, -2.386 \times 10^{-16})$. I have given the answers on this report to the 3 decimal place.

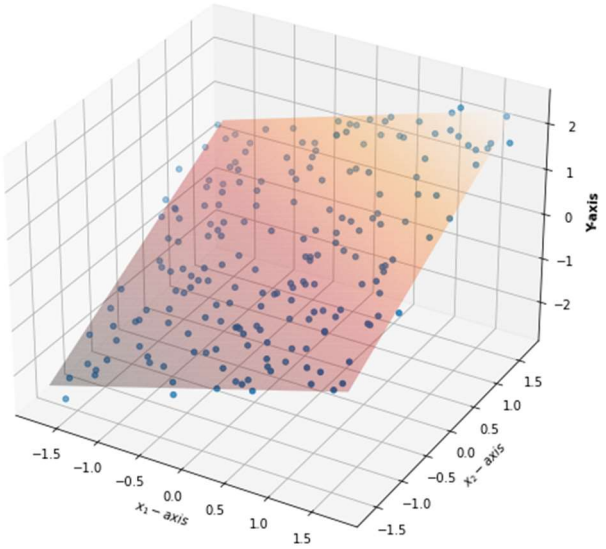


Figure 1: Bivariate Regression Plot of the variables x_1, x_2 and y

This shows the positive linear correlation between the variables x_1 with y and x_2 with y. While the variable x_1 and x_2 have a negative correlation with each other.

When we run a code where we try to investigate the log scale against number of iterations in linear scale. In the first run it plots the cost n log scale against max number of iterations for step size $t = [0.01, 0.1, 0.9, 1.9, 1.92]$. where we get the following graph on which $t = 0.01$ shows a linear descent for the cost n log scale with its iteration. Implying that it the cost of n is less the higher the iteration up until 100. While $t = 0.1$ follows an exponential decay curve to a steady state solution of 0. Showing the choice of $t = 0.1$ is better than 0.01. However, the $t = 0.9$ has a constant n log scale cost. For all the number of iterations. Which is also the lowest in cost, so the best choice would be to choose $t = 0.9$ as its gives the lowest cost from the beginning till the 100th iteration and this can also be seen for higher iterations.

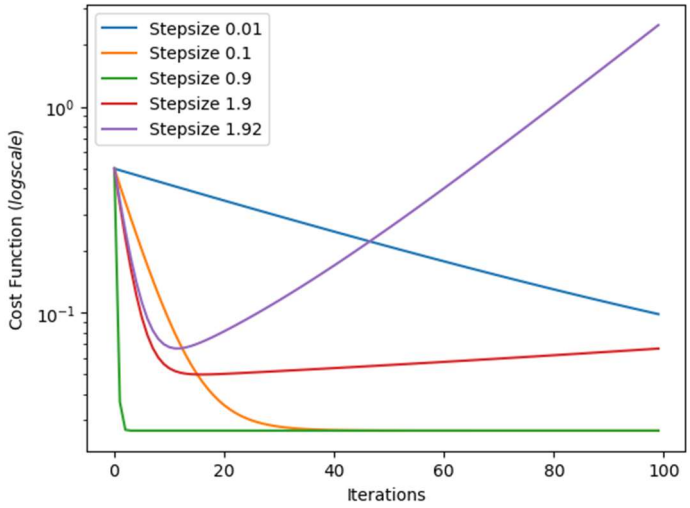


Figure 2: Plot of the cost function (log scale) vs number of iteration (100)

In the second category of step size are $[1.9,1.92]$ where both these step sizes demonstrate to us the decay of cost function up until the 10th iteration which is when $t = 1.9$ starts off with a high cost. However, it reduces bellow to 10^{-1}but is slowly growing after the 10th iteration similarly with $t = 1.92$. However, it's growth is not slow but rapid where the cost function is increasing as number of iteration increases. So, the conclusion for the best step size to be used for the mean square estimator is 0.9 as it has the lowest cost.

However, by changing the number of iterations from 100 to 200, we can see the intersection of time step $t = 1.9$ and $t = 0.01$ where, just before 125 iterations the cost function for 0.01 and 1.9 cross as seen in figures 2.2 and you have lower cost for step size 0.01. So, from observations the best step size would be 0.9.

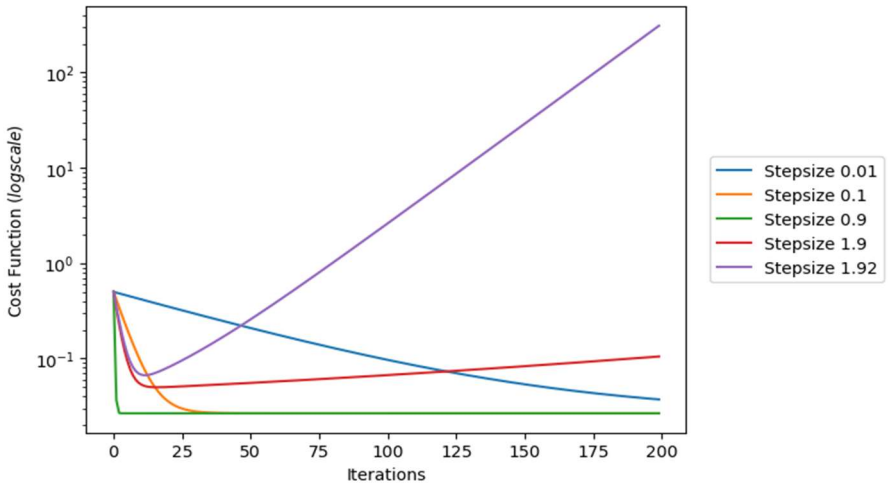


Figure 2.2: Plot of the cost function (log scale) vs number of iteration (200)

Question 2: Average High Temperature at Nottingham

Table 1: Average High Temperatures at Nottingham

Month	Temperature (°C)
Jan	7
Feb	8
Mar	10
Apr	13
May	17
Jun	19
Jul	22
Aug	21
Sep	18
Oct	14
Nov	10
Dec	8

Figure 3: The average temperature in Nottingham in degrees Celsius

The function $f(t; \boldsymbol{x}) = x_1 \sin(x_2\, t + x_3) + x_4$ has a cyclical pattern, if we were to put the x_1 in context it should be the amplitude of our sin function (maximum high from the stationary point). It will control how high and low our temperature can go. Due to the maximum temperature being 22 degrees Celsius, while the lowest temperature being 7 degrees Celsius, so based on this information I choose to pick the middle point which is 7.5 degrees Celsius. For the parameter x_2 is the frequency of our periodic function, this should just influence the distance between the two temperature points in our plot in figure 4. So, the choice for it is clearly using the formula $x_2 = frequency = \frac{2\pi}{period} = \frac{2\pi}{12} = \frac{\pi}{6}$. The parameter x_3 is the phase shift of the wave function, so if our function is going to be shifted to the left or right to match the original temperature points with the predicted curve, I choose $\mathrm{x}_3 = \frac{-\pi}{2}$ to help shift the sin function to the right by $\frac{\pi}{2}$ which should be 3 months. The final parameter x_4 is how much we move our sin function up and down (vertical shift), so it should be within the range of our max and min of temperature. It is safe to guess for 15 degrees Celsius, which is difference between max(T) and min(T).

The average temperature from January to December in Nottingham is given in figure 3. For which we are given the cyclical pattern of the temperature suggested in the model in the form $f(t;\ \boldsymbol{x}) = x_1 \sin(x_2\, t + x_3) + x_4$ where the vector $\boldsymbol{x} = (x_1, x_2, x_3, x_4)^{\mathrm{T}}$ which is constant is to be determined for the purpose of being able to find the relationship between the temperature and the months. The vector $\boldsymbol{x}$ is determined by solving for $\min_{\boldsymbol{x}\in\mathbb{R}^4}\left\{ \sum_{k=1}^{12} (f(k;\ \boldsymbol{x}) - T_k)^2 \right\}$. To resolve this problem, we approached it in two ways:

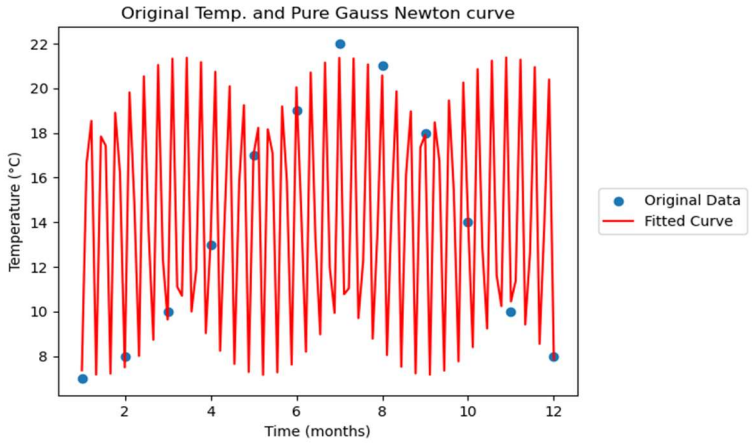
- Pure Gauss-Newton method, which is expressed as $J(\boldsymbol{x_k})$ is our Jacobian matrix which is constructed using the $F(\boldsymbol{x_k}) = \begin{bmatrix} f(t_1; \boldsymbol{x_k}) - T_1 \\ f(t_2; \boldsymbol{x_k}) - T_2 \\ \vdots \\ f(t_{12}; \boldsymbol{x_k}) - T_{12} \end{bmatrix}$ where T_k is the Temperature for each month from the table in figure 3.
  This is because we want to minimize the distance between the points our model predicts and the actual temperature, so the problem becomes $\min_{\boldsymbol{x}\in\mathbb{R}^4} \|F(\boldsymbol{x_k})\|^2$. This helps solve the non-linear least squares

problem, its just an iterative approach like the gradient decent which at the end generate our sequence $\boldsymbol{x_k}$ for $k = 1,2,\ldots$ which should converge (which will be addressed later in the report). This just transitions into $\mathbf{x}_{k+1} = \underset{\boldsymbol{x}\in\mathbb{R}^4}{\operatorname{argmin}}\|J(\boldsymbol{x}_k)\boldsymbol{x} - \boldsymbol{b}_k\|^2$, $\mathbf{b}_k = J(\boldsymbol{x}_k)\boldsymbol{x} - F(\boldsymbol{x_k})$ as refered in <u>lecture notes 5 page 3</u> this is reduced down into a linear least squares problem. Which is ~~just~~

$$J(\boldsymbol{x_k}, \boldsymbol{t}) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \frac{\partial F_1}{\partial x_3} & \frac{\partial F_1}{\partial x_4} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \frac{\partial F_2}{\partial x_3} & \frac{\partial F_2}{\partial x_3} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial F_{12}}{\partial x_1} & \frac{\partial F_{12}}{\partial x_2} & \frac{\partial F_{12}}{\partial x_3} & \frac{\partial F_{12}}{\partial x_4} \end{bmatrix}, \quad \begin{aligned} \frac{\partial F_k}{\partial x_1} &= \sin(\boldsymbol{x_2}t_k + \boldsymbol{x_3}) \\ \frac{\partial F_k}{\partial x_2} &= \boldsymbol{x_1}t_k\cos(\boldsymbol{x_2}t_k + \boldsymbol{x_3}) \\ \frac{\partial F_k}{\partial x_3} &= \boldsymbol{x_1}\cos(\boldsymbol{x_2}t_k + \boldsymbol{x_3}) \\ \frac{\partial F_k}{\partial x_4} &= \mathbf{1} \end{aligned}$$

$$J(\boldsymbol{x_k}, \boldsymbol{t}) = \begin{bmatrix} \sin(\boldsymbol{x_2}t_1 + \boldsymbol{x_3}) & \boldsymbol{x_1}t_1\cos(\boldsymbol{x_2}t_1 + \boldsymbol{x_3}) & \boldsymbol{x_1}\cos(\boldsymbol{x_2}t_1 + \boldsymbol{x_3}) & 1 \\ \sin(\boldsymbol{x_2}t_2 + \boldsymbol{x_3}) & \boldsymbol{x_1}t_2\cos(\boldsymbol{x_2}t_2 + \boldsymbol{x_3}) & \boldsymbol{x_1}\cos(\boldsymbol{x_2}t_2 + \boldsymbol{x_3}) & 1 \\ \vdots & \vdots & \vdots & \vdots \\ \sin(\boldsymbol{x_2}t_{12} + \boldsymbol{x_3}) & \boldsymbol{x_1}t_{12}\cos(\boldsymbol{x_2}t_{12} + \boldsymbol{x_3}) & \boldsymbol{x_1}\cos(\boldsymbol{x_2}t_{12} + \boldsymbol{x_3}) & 1 \end{bmatrix}$$

This just further reduces down to $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \left(J(\boldsymbol{x_k}, t)^{\mathrm{T}}J(\boldsymbol{x_k}, t)\right)^{-1}\nabla g(\boldsymbol{x_k}, t)$ where the $\nabla g(\boldsymbol{x_k}, \boldsymbol{t}) = J(\boldsymbol{x_k}, \boldsymbol{t})^{\mathrm{T}}F(\boldsymbol{x_k})$ and $\boldsymbol{t} = [1,2,3,\ldots,12]$ which represents months. So, when we finally implement the pure Gauss-Newton metho d with $\mathbf{x_0} = (5, 1, 0, 10)$ with the tolerance being for $\|\nabla g(x_{k+1})\| \leq 10^{-6}$, as <u>shown in lecture notes 5 page 3,</u> you g et the output $\boldsymbol{x_k} = (-7.11, -37.15, 289.80, 14.27)$ at $k = 21$



<u>Figure 4</u>: Curve plot for the Pure Gauss Newton method with original data

- The second methd which was used is Damped Gauss-Newton method. For this method everything is the same. However, we are not implementing a step size which helps dampen our function. As seen in figure 4 the plot's frequency is too high which results in the curve showing spike in temperature down to 7 degrees Celsius in May and June, which is uncommon. Hence by implementing the $t_k = \min\{1, \frac{1}{\|\nabla g(x_k)\|}\}$. When this is done, we get the $x_k = (7.11, 0.55, 3.92, 14.27)$ at $k = 183$ where the values for $\boldsymbol{x}$ as much more appropriate as it gives us a new and much more improved curve plot in figure 5. Although there are months such as 7(July) which are do not align closely with the curve. However, we do observe that this is a superior model in comparison to the Pure Gauss Newton method, my justification is that frequency which is x_2 must be greater than zero. The damped method does have slower convergence however due to the implementation of the

step size where we pick between 1 and the inverse of the norm of gradient g we get a much more refined iteration where it prevents our final solution from overshooting like the pure method.

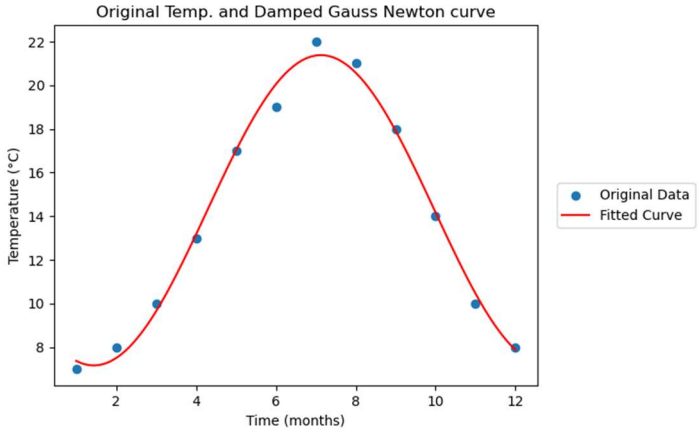


Figure 5: Curve plot for the Damped Gauss Newton method with original data

However as seen in previous modules(Scientific Computation) we know through polynomial interpolation; you can fit multiple values for $\boldsymbol{x}$ such as seen in the above example for Pure and Damped Gauss-Newton methods. Both fit the model as all the original data points are covered by the respective values for $\boldsymbol{x}$**.** However as mentioned above, only one is valued in application due to the definition of x_2 being the frequency which must be strictly positive. One of the reasons we do see the big discrepancies in the values for x_1 and x_3 is due to the implementation of the step size in your iteration for damped. Furthermore, when we change the initial starting point to $\boldsymbol{x_0} = (100,100,100,100)$ we see both Pure and Damped Gauss-Newton methods converge to $x_k = (7.11,99.98,99.76,12.27)$ for 2 decimal places. For pure $k = 8$, while the damped $k = 87285$ both producing reasonable plots in figure 6 but the fastest one in this case is the pure method taking up less computational time. This is an alternative converging point; however, this is still poor choice of parameters due to the application of the parameter. It is not common to see in January average temperature spike to be above 20°C. As the model is supposed to curve out the average temperature change in Nottingham throughout the year.

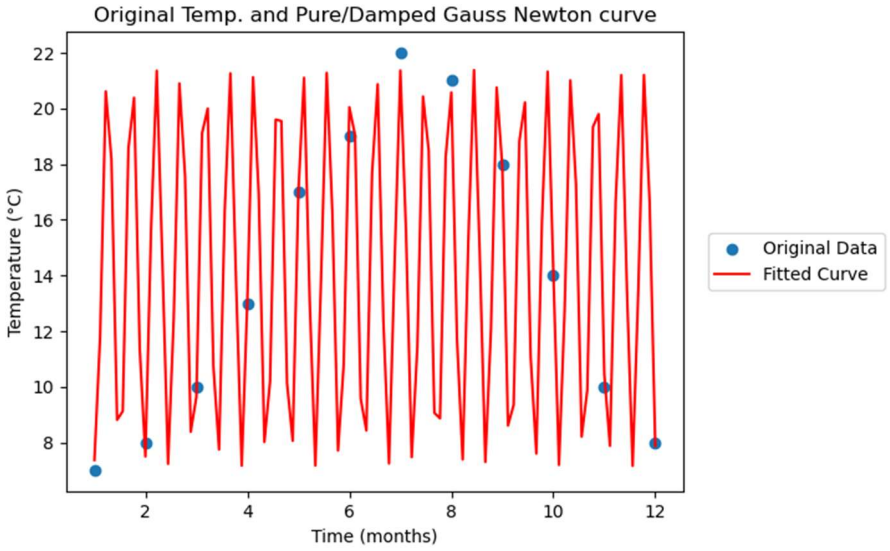


Figure 6: Curve plot for the Pure/Damped Gauss Newton method with original at initial $x_0 = (100,100,100,100)$

Conclusion

To conclude this question, we find that there is more than one solution for our parameter x as shown above, however not all solutions are valid due to the properties of $x = (x_1, x_2, x_3, x_4)$ where clearly x_2 must be greater than zero due to it being the frequency. It is quite possible that for future test we run the same algorithm within constrained boundaries to allow for a unique convergence. Otherwise in an unconstrained optimisation problem we can have infinitely many initial points and get possibly infinitely many solutions which is not optimal. Which is another reason why we get big difference in values for the damped and pure method as it is unconstrained. Furthermore, it is possible to implement the regularised least squares to our pure method to prevent overfitting which is what we see in figure 4 to help since we have only 12 data points, however as we know $\mathrm{x_{RLS}} = \left(\mathrm{A^T A} + \lambda \mathrm{D^T D}\right)^{-1} \mathrm{A^T b}$, where if λ is too big can cause our iteration to shoot off and not be accurate as this parameter allows the solution to come out clean and smooth. But since the damped method does demonstrate the best curve at the initial point (5,1,0,10) it may be best to leave this for one. Finally, we also must understand that the Gauss-Newton method is a linear approximation which means its best suited for when we know our solution should be within a boundary, so our initial guess is not too far off from our optimal solution. Which means its sensitive to our initial guess and give us other solutions. An alternative method could possibly be that we use Quadrative approximation which is Newton's method. Due to the fact we have linearised our sinusoidal function at each iteration it would not give us the most accurate solutions at each iteration, while on the other hand the quadratic approximation is capturing more information at each iteration for a stronger solution. However not everything good in life comes free and easy, the newton's method would require us to compute the inverse of the Hessian at each iteration which is very demanding in time space complexity as it is $O(n^3)$, furthermore it requires the hessian must be positive definite. Given all this information it may be worth trying to use the Newton's method but as we know the Gauss newton method $J(\boldsymbol{x_k}, \boldsymbol{t})^\top J(\boldsymbol{x_k}, \boldsymbol{t})$ would primarily dominate as mentioned in the lecture 5 page 5. So even though you would get a faster convergence at the cost of taking up lots of time and energy it would be best to go with the damped Gauss-Newton method.

Appendix (code)

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Read the CSV f i l e

data = pd.read_csv ('bivariate_linear_regression_data.csv')

Question 1.a

#Initialise

X1 = data ['X1']

X2 = data ['X2']

y = data ['y']

# Normalize the f e at u r e s and t a r g e t

X1 = (X1 - np.mean(X1 ) ) / np.std(X1 )

X2 = (X2 - np.mean(X2 ) ) / np.std(X2 )
```

```
y = ( y -  np.mean(y ) ) /  np.std(y  )
#Hessian
m = len(X1)
H = (1/m)*np.array([
   [np.sum(X1**2), np.sum(X1 * X2), np.sum(X1)],
   [np.sum(X1 * X2), np.sum(X2**2), np.sum(X2)],
   [np.sum(X1)   , np.sum(X2)     , m         ]])
#Check
rank_H = np.linalg.matrix_rank(H)
if rank_H == H.shape[0]:
   print("True the matrix H invertible")
else:
   Print("False the matrix H invertible")
Question 1B and C
def gradient_descent(X1,X2,y,t,iterations,w_0):
      w1 = w_0[0]
      w2 = w_0[2]
      b  = w_0[1]
      m = len(y)
      for _ in range(iterations):
         #Calculate the predicted values
         h = w1*X1 + w2*X2 + b
         #W1,W2,b
         w1 -= t*1/(m) * np.sum((h - y) * X1)
         w2 -= t*1/(m) * np.sum((h - y) * X2)
         b  -= t*1/(m) * np.sum( h - y)
      return w1,w2,b
   w_0 = [0,0,0]
   iterations = 100
   t = 0.1
   gradient_descent(X1,X2,y,t,iterations,w_0)
   Question 1D
```

```
def plot_B_R(X1,X2,y,t,iterations):
    w1, w2, b = gradient_descent(X1,X2,y,t,iterations,w_0)
    # Creating figure
    fig = plt.figure(figsize=(16, 9))
    ax = fig.add_subplot(111, projection='3d')
    # Scatter plot for data points
    ax.scatter3D(X1, X2, y, marker='o', label='$(x_1, x_2)$')
    # Decision boundary
    x1_range = np.linspace(min(X1), max(X1), 100)
    x2_range = np.linspace(min(X2), max(X2), 100)
    x1_vals, x2_vals = np.meshgrid(x1_range, x2_range)
    y_vals = w1 * x1_vals + w2 * x2_vals + b
    ax.plot_surface(x1_vals, x2_vals, y_vals, alpha=0.3, cmap='gist_heat',label='Decision Boundary')
    ax.set_xlabel('$x_1-axis$', fontweight='bold')
    ax.set_ylabel('$x_2-axis$', fontweight='bold')
    ax.set_zlabel('Y-axis', fontweight='bold')
    plt.show()
plot_B_R(X1,X2,y,t,iterations)
```

Question 1E and F

```
def gradient_descent_cost(X1,X2,y,t,iterations,w_0):
    m = len(y)
    w1 = w_0[0]
    w2 = w_0[2]
    b  = w_0[1]
    cost = []
    for i in range(iteration):
        h = w1 * X1 + w2 * X2 + b
        grad_w1 = (1/m) * np.sum((h - y)*X1)
        grad_w2 = (1/m) * np.sum((h - y)*X2)
        grad_b = (1/m) * np.sum(h - y)
#gradient descent
        w1 -= t * grad_w1
```

```
        w2 -= t * grad_w2
        b -= t * grad_b
#cost
        MSE_cost = (1 / (2 * m)) * np.sum((h - y) ** 2)
        cost.append(MSE_cost)

    return w1, w2, b, cost

#parameter

iteration = 200
stepsizes = [0.01, 0.1, 0.9,1.9,1.92]

#plot

for _ in stepsizes:
    w1, w2, b, cost = gradient_descent_cost(X1,X2,y,_,iterations,w_0)
    plt.plot(range(iteration), cost, label=f'Stepsize {_}')

plt.xlabel('Iterations')
plt.ylabel('Cost Function $(log scale)$')
plt.yscale('log')
plt.legend(loc='center left', bbox_to_anchor=(1.04, 0.5), borderaxespad=0)
plt.show()

Question 2C and F
# Data
t = np.arange(1,12+1)
temp = np.array([7,8,10,13,17,19,22,21,18,14,10,8])

# Initial
x0 = np.array([5,1,0,10])
```

```
TOL = 10**(-6)
#n = 10*6

#Function f
def f(x,t):
    #t is time represented in months 1-12
    return x[0]*np.sin(x[1]*t + x[2]) + x[3]

#Jacobian
def Jocobian(x,t):
    Jocobian = np.zeros((len(t),len(x)))

    for k in range(0,len(t)):
        Jocobian[k,0] =          np.sin(t[k]*x[1] +x[2])
        Jocobian[k,1] = t[k]*x[0]*np.cos(t[k]*x[1] +x[2])
        Jocobian[k,2] =      x[0]*np.cos(t[k]*x[1] +x[2])
        Jocobian[k,3] = 1
    return Jocobian

def gradient(x,temp,t):
    F = f(x,t) - temp
    J = Jocobian(x,t)
    grad = (J.T)@F
    return grad

def Hessian(x,t):
    J = Jocobian(x,t)
    Hessian = (J.T)@J
    return Hessian
#Gaus-newtton and the Damping method
def gauss_newtton_or_Damp(x0,temp,t,TOL,Damp):
    i = 0
```

```
#Tolerance
    while np.linalg.norm(gradient(x0,temp,t)) >= TOL:
        directional_deriv = np.linalg.solve(Hessian(x0, t),-gradient(x0,temp,t))
#Damping
        if Damp == True:
            x0 = x0 + directional_deriv * min(1, 1 / np.linalg.norm(gradient(x0,temp,t)))
            i+=1
#Pure Gauss-Newtton
        else:
            x0 = x0 + directional_deriv
            i+=1
    print("The for method below the iteration stops at:",i)
    return x0


print("\n###################################")
print("The Gause-N method gives:", gauss_newtton_or_Damp(x0,temp,t,TOL,Damp=False))
print("\n###################################")
print("The Damping Gause-N method gives:", gauss_newtton_or_Damp(x0,temp,t,TOL,Damp=True))
print("\n###################################")
def fit(x, temp, t):
    plt.scatter(t, temp, label='Original Data')

    # Time values for the curve
    time_fit = np.linspace(min(t), max(t), 100)

    # Compute values
    temp_fit = f(x, time_fit)

    plt.plot(time_fit, temp_fit, label='Fitted Curve', color='red')
    plt.xlabel('Time (months)')
    plt.ylabel('Temperature (°C)')
    plt.title('Original Temp. and Pure/Damped Gauss Newton curve')
```

```
    plt.legend(loc='center left', bbox_to_anchor=(1.04, 0.5), borderaxespad=0)
    plt.show()

fit(gauss_newtton_or_Damp(x0, temp, t, TOL, Damp=False), temp, t)
fit(gauss_newtton_or_Damp(x0, temp, t, TOL, Damp=True), temp, t)

x0 = np.array([100,100,100,100])
TOL = 10**(-6)
print("\n##################################")
fit(gauss_newtton_or_Damp(x0, temp, t, TOL, Damp=False), temp, t)
print("The Gause-N method gives:", gauss_newtton_or_Damp(x0,temp,t,TOL,Damp=False))
print("\n##################################")
fit(gauss_newtton_or_Damp(x0, temp, t, TOL, Damp=True), temp, t)
print("The Damping Gause-N method gives:", gauss_newtton_or_Damp(x0,temp,t,TOL,Damp=True))
print("\n##################################")
```