HOMEWORK SET 2

ΓΑΛΑΝΗΣ ΑΧΙΛΛΕΑΣ ΑΛΕΞΑΝΔΡΟΣ ΒΑΣΙΛΕΙΟΣ - 02941 and ΓΑΛΑΝΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΟΡΕΣΤΗΣ ΒΑΣΙΛΕΙΟΣ - 03074

Νευρο-Ασαφής Υπολογιστική 2023-24
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας, Βόλος
{acgalanis, kogalanis}@uth.gr

Περιεχόμενα

1 PROBLEM-01

– **Conjugate Gradient**
Given the function:

$$F(w) = w_1^2 + w_2^2 + (0.5w_1 + w_2)^2 + (0.5w_1 + w_2)^4$$

1st Iteration:

- The gradient is:

$$\nabla F(x_0) = \left[\frac{\partial F}{\partial w_1}(x_0), \frac{\partial F}{\partial w_2}(x_0)\right]^T$$

$$= \left[2w_1 + 0.5 \cdot w_1 + w_2 + 2 \cdot (0.5 \cdot w_1 + w_2)^3, 2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3\right]^T$$

For the initial guess $x_0 = [3, 3]^T$, the gradient is calculated as:

$$\nabla F(x_0) = \left[2 \cdot 3 + 0.5 \cdot 3 + 3 + 2 \cdot (0.5 \cdot 3 + 3)^3, 2 \cdot 3 + 2(0.5 \cdot 3 + 3) + 4(0.5 \cdot 3 + 3)^3\right]^T$$

$$\nabla F(x_0) = \left[2 \cdot 3 + 4.5 + 2 \cdot (4.5)^3, 2 \cdot 3 + 2 \cdot 4.5 + 4 \cdot 91.125\right]^T$$

$$\nabla F(x_0) = [6 + 4.5 + 2 \cdot 91.125, 6 + 9 + 364.5]^T$$

$$\nabla F(x_0) = [10.5 + 182.25, 6 + 9 + 364.5]^T$$

$$\nabla F(x_0) = [192.75, 379.5]^T$$

- Thus the search direction is the negative gradient:

$$s_0 = -\nabla F(x_0) = [-192.75, -379.5]^T$$

- Now we want to find the Hessian matrix H, which consists of all the second-order partial derivatives of F:

$$H = \begin{bmatrix} \frac{\partial^2 F}{\partial w_1^2} & \frac{\partial^2 F}{\partial w_1 \partial w_2} \\ \frac{\partial^2 F}{\partial w_2 \partial w_1} & \frac{\partial^2 F}{\partial w_2^2} \end{bmatrix}$$

First, we calculate the partial derivatives with respect to w_1:

$$\frac{\partial F}{\partial w_1} = 2w_1 + 0.5 \cdot w_1 + w_2 + 2 \cdot (0.5 \cdot w_1 + w_2)^3$$

$$\frac{\partial^2 F}{\partial w_1^2} = 2 + 0.5 + 3 \cdot (0.5 \cdot w_1 + w_2)^2$$

$$\frac{\partial^2 F}{\partial w_1^2} = 2.5 + 3 \cdot (0.5w_1 + w_2)^2$$

Next, we calculate the partial derivatives with respect to w_2:

$$\frac{\partial F}{\partial w_2} = 2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3$$

$$\frac{\partial^2 F}{\partial w_2^2} = 2 + 2 + 12(0.5w_1 + w_2)^2$$

$$\frac{\partial^2 F}{\partial w_2^2} = 4 + 12(0.5w_1 + w_2)^2$$

Finally, the mixed partial derivatives:

$$\frac{\partial^2 F}{\partial w_1 \partial w_2} = \frac{\partial}{\partial w_2}\left(2w_1 + 0.5 \cdot w_1 + w_2 + 2 \cdot (0.5 \cdot w_1 + w_2)^3\right)$$

$$\frac{\partial^2 F}{\partial w_1 \partial w_2} = 0 + 0 + 1 + 6 \cdot (0.5w_1 + w_2)^2$$

$$\frac{\partial^2 F}{\partial w_1 \partial w_2} = 1 + 6 \cdot (0.5w_1 + w_2)^2$$

Since the second-order partial derivatives are continuous, we have $\frac{\partial^2 F}{\partial w_1 \partial w_2} = \frac{\partial^2 F}{\partial w_2 \partial w_1}$. Thus, the Hessian matrix is:

$$H = \begin{bmatrix} 2.5 + 3(0.5w_1 + w_2)^2 & 1 + 6(0.5w_1 + w_2)^2 \\ 1 + 6(0.5w_1 + w_2)^2 & 4 + 12(0.5w_1 + w_2)^2 \end{bmatrix}$$

Given $w_1 = 3$ and $w_2 = 3$, the Hessian matrix H becomes:

$$H\big|_{w_1=3, w_2=3} = \begin{bmatrix} 2.5 + 3(0.5 \cdot 3 + 3)^2 & 1 + 6(0.5 \cdot 3 + 3)^2 \\ 1 + 6(0.5 \cdot 3 + 3)^2 & 4 + 12(0.5 \cdot 3 + 3)^2 \end{bmatrix}$$

Simplifying the terms inside the matrix:

$$H\big|_{w_1=3, w_2=3} = \begin{bmatrix} 2.5 + 3(4.5)^2 & 1 + 6(4.5)^2 \\ 1 + 6(4.5)^2 & 4 + 12(4.5)^2 \end{bmatrix}$$

$$H\big|_{w_1=3, w_2=3} = \begin{bmatrix} 2.5 + 3 \cdot 20.25 & 1 + 6 \cdot 20.25 \\ 1 + 6 \cdot 20.25 & 4 + 12 \cdot 20.25 \end{bmatrix}$$

$$H\big|_{w_1=3, w_2=3} = \begin{bmatrix} 2.5 + 60.75 & 1 + 121.5 \\ 1 + 121.5 & 4 + 243 \end{bmatrix}$$

$$H\big|_{w_1=3, w_2=3} = \begin{bmatrix} 63.25 & 122.5 \\ 122.5 & 247 \end{bmatrix}$$

- The step size λ_0 is calculated as:

$$\lambda_0 = -\frac{\nabla F(w_0)^T s_0}{s_0^T H s_0}$$

Substituting the values we have:

$$\lambda_0 = -\frac{\begin{bmatrix}192.75\ 379.5\end{bmatrix}\begin{bmatrix}-192.75\\ -379.5\end{bmatrix}}{\begin{bmatrix}-192.75\ -379.5\end{bmatrix}\begin{bmatrix}63.25 & 122.5\\ 122.5 & 247\end{bmatrix}\begin{bmatrix}-192.75\\ -379.5\end{bmatrix}}$$

Calculating the numerator and the denominator:

$$\lambda_0 = -\frac{-181172.81}{55844313.0375}$$

Thus, the step size λ_0 is approximately:

$$\lambda_0 \approx 0.0032$$

- Using the update rule:

$$x^{(1)} = x^{(0)} + \lambda_0 s_0$$

$$x^{(1)} = \begin{bmatrix}3\\ 3\end{bmatrix} + 0.0032\begin{bmatrix}-192.75\\ -379.5\end{bmatrix}$$

$$x^{(1)} = \begin{bmatrix}3\\ 3\end{bmatrix} + \begin{bmatrix}-0.625\\ -1.21\end{bmatrix}$$

$$x^{(1)} = \begin{bmatrix}3 - 0.6168\\ 3 - 1.21\end{bmatrix}$$

$$x^{(1)} = \begin{bmatrix}2.383\\ 1.79\end{bmatrix}$$

- The gradient at the new point x_1 is computed as:

$$\nabla F(x_1) = \left[\frac{\partial F}{\partial w_1}(x_1), \frac{\partial F}{\partial w_2}(x_1)\right]^T$$

For the new point $x_1 = [2.383, 1.79]^T$, the gradient is calculated as:

$$\nabla F(x_1) = \Bigg[2\cdot(2.383) + 0.5\cdot(2.383) + (1.79)$$
$$+ 2\cdot(0.5\cdot(2.383) + (1.79))^3,$$
$$2\cdot(1.79) + 2\,(0.5\cdot(2.383) + (1.79))$$
$$+ 4\,(0.5\cdot(2.383) + (1.79))^3\Bigg]^T$$

$$= [60.754, 115.558]^T$$

- The new search direction s_1 is given by:

$$s_1 = -\nabla F(x_1) + \omega_1 s_0$$

- The formula for ω_1 is:

$$\omega_1 = \frac{\nabla F(x_1)^T \nabla F(x_1)}{\nabla F(x_0)^T \nabla F(x_0)}$$

  Substituting the values into the formula:

$$\omega_1 = \frac{(60.754)^2 + (115.558)^2}{(192.75)^2 + (379.5)^2}$$

$$\omega_1 = \frac{3691.04 + 13353.65}{37152.5625 + 144020.25}$$

$$\omega_1 = \frac{17044.69}{181172.812}$$

$$\omega_1 \approx 0.094$$

- Thus, the new search direction s_1 can be calculated as:

$$s_1 \approx \begin{bmatrix} -60.754 \\ -115.558 \end{bmatrix} + 0.094 \begin{bmatrix} -192.75 \\ -379.5 \end{bmatrix}$$

$$s_1 \approx \begin{bmatrix} -78.87 \\ -151.231 \end{bmatrix}$$

2nd Iteration:

- Given the new gradient at x_1 and the new search direction s_1:

$$\nabla F(x_1) = \begin{bmatrix} 60.754 \\ 115.558 \end{bmatrix}, \quad s_1 = \begin{bmatrix} -78.87 \\ -151.231 \end{bmatrix}$$

- The new Hessian matrix H is:

$$H\big|_{x_1=2.383, x_1=1.79} = \begin{bmatrix} 2.5 + 3 \cdot (0.5 \cdot 2.383 + 1.79)^2 & 1 + 6 \cdot (0.5 \cdot 2.383 + 1.79)^2 \\ 1 + 6 \cdot (0.5 \cdot 2.383 + 1.79)^2 & 4 + 12 \cdot (0.5 \cdot 2.383 + 1.79)^2 \end{bmatrix} = \begin{bmatrix} 29.168 & 54.336 \\ 54.336 & 110.672 \end{bmatrix}$$

- The new step size λ_1 is calculated using the formula:

$$\lambda_1 = -\frac{\nabla F(x_1)^T s_1}{s_1^T H s_1}$$

Substituting the values into the formula:

$$\lambda_1 = -\frac{\begin{bmatrix}60.754 \; 115.558\end{bmatrix}\begin{bmatrix}-78.87\\-151.231\end{bmatrix}}{\begin{bmatrix}-78.87 \; -151.231\end{bmatrix}\begin{bmatrix}29.168 & 54.336\\54.336 & 110.672\end{bmatrix}\begin{bmatrix}-78.87\\-151.231\end{bmatrix}}$$

Therefore, the new step size λ_1 is:

$$\lambda_1 \approx 0.0055$$

- Using the update rule for the next iteration:

$$x^{(2)} = x^{(1)} + \lambda_1 s_1$$

Given that $x^{(1)} = \begin{bmatrix}2.383\\1.79\end{bmatrix}$, $\lambda_1 = 0.0055$, and $s_1 = \begin{bmatrix}-78.87\\-151.231\end{bmatrix}$, we calculate $x^{(2)}$:

$$x^{(2)} = \begin{bmatrix}2.383\\1.79\end{bmatrix} + 0.0055\begin{bmatrix}-78.87\\-151.231\end{bmatrix}$$

$$x^{(2)} = \begin{bmatrix}1.949\\0.958\end{bmatrix}$$

- The gradient at the new point x_2 is computed as:

$$\nabla f(x_2) = \left[\frac{\partial F}{\partial w_1}(x_2), \frac{\partial F}{\partial w_2}(x_2)\right]^T$$

$$\begin{aligned}\nabla F(x_2) = &[2 \cdot 1.949 + 0.5 \cdot 1.949 + 0.958\\ &+2 \cdot (0.5 \cdot 1.949 + 0.958)^3, 2 \cdot 0.958 + 2(0.5 \cdot 1.949 + 0.958)\\ &+4(0.5 \cdot 1.949 + 0.958)^3]^T\\ =&\begin{bmatrix}20.264\\34.649\end{bmatrix}\end{aligned}$$

- The new search direction s_2 is given by:

$$s_2 = -\nabla F(x_2) + \omega_2 s_1$$

- The formula for ω_2 is:

$$\omega_2 = \frac{\nabla F(x_2)^T \nabla F(x_2)}{\nabla F(x_1)^T \nabla F(x_1)}$$

Substituting the values into the formula:

$$\omega_2 = \frac{(20.264)^2 + (34.649)^2}{(60.754)^2 + (115.558)^2}$$

$$\omega_2 = \frac{410.629 + 1200.553}{3691.04 + 13353.65}$$

$$\omega_2 = \frac{1611.182}{17044.69}$$

$$\omega_2 \approx 0.094$$

- Thus, the new search direction s_3 can be calculated as:

$$s_2 \approx \begin{bmatrix} -20.264 \\ -34.649 \end{bmatrix} + 0.094 \begin{bmatrix} -78.87 \\ -151.231 \end{bmatrix}$$

$$s_2 \approx \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}$$

3rd Iteration:

- Given the new gradient at x_2 and the new search direction s_2:

$$\nabla F(x_2) = \begin{bmatrix} 20.264 \\ 34.649 \end{bmatrix}, \quad s_2 = \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}$$

- The new Hessian matrix H is:

$$H\big|_{x_1=1.949, x_1=0.958} = \begin{bmatrix} 2.5 + 3 \cdot (0.5 \cdot 1.949 + 0.958)^2 & 1 + 6 \cdot (0.5 \cdot 1.949 + 0.958)^2 \\ 1 + 6 \cdot (0.5 \cdot 1.949 + 0.958)^2 & 4 + 12 \cdot (0.5 \cdot 1.949 + 0.958)^2 \end{bmatrix} = \begin{bmatrix} 13.703 & 23.407 \\ 23.407 & 48.814 \end{bmatrix}$$

- The new step size λ_2 is calculated using the formula:

$$\lambda_2 = -\frac{\nabla F(x_2)^T s_2}{s_2^T H s_2}$$

Substituting the values into the formula:

$$\lambda_2 = -\frac{\begin{bmatrix} 20.264 & 34.649 \end{bmatrix} \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}}{\begin{bmatrix} -27.677 & -48.864 \end{bmatrix} \begin{bmatrix} 13.703 & 23.407 \\ 23.407 & 48.814 \end{bmatrix} \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}}$$

Therefore, the new step size λ_2 is:

$$\lambda_2 \approx 0.011$$

- Using the update rule for the next iteration:

$$x^{(3)} = x^{(2)} + \lambda_2 s_2$$

Given that $x^{(2)} = \begin{bmatrix} 1.949 \\ 0.958 \end{bmatrix}$, $\lambda_2 = 0.009$, and $s_2 = \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}$, we calculate $x^{(3)}$:

$$x^{(3)} = \begin{bmatrix} 1.949 \\ 0.958 \end{bmatrix} + 0.011 \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}$$

$$x^{(3)} = \begin{bmatrix} 1.644 \\ 0.42 \end{bmatrix}$$

- The gradient at the new point x_3 is computed as:

$$\nabla f(x_3) = \left[\frac{\partial F}{\partial w_1}(x_3), \frac{\partial F}{\partial w_2}(x_3) \right]^T$$

$$\begin{aligned} \nabla F(x_3) = &[2 \cdot 1.644 + 0.5 \cdot 1.644 + 0.42 \\ &+2 \cdot (0.5 \cdot 1.644 + 0.42)^3, 2 \cdot 0.42 + 2(0.5 \cdot 1.644 + 0.42) \\ &+4(0.5 \cdot 1.644 + 0.42)^3]^T \\ = &\begin{bmatrix} 8.361 \\ 10.987 \end{bmatrix} \end{aligned}$$

- The new search direction s_3 is given by:

$$s_3 = -\nabla F(x_3) + \omega_3 s_2$$

- The formula for ω_3 is:

$$\omega_3 = \frac{\nabla F(x_3)^T \nabla F(x_3)}{\nabla F(x_2)^T \nabla F(x_2)}$$

Substituting the values into the formula:

$$\omega_3 = \frac{(8.361)^2 + (10.987)^2}{(20.264)^2 + (34.649)^2}$$

$$\omega_3 = \frac{69.906 + 120.714}{410.629 + 1200.553}$$

$$\omega_3 = \frac{190.62}{1611.182}$$

$$\omega_3 \approx 0.118$$

- Thus, the new search direction s_3 can be calculated as:

$$s_3 \approx \begin{bmatrix} -8.361 \\ -10.987 \end{bmatrix} + 0.118 \begin{bmatrix} -27.677 \\ -48.864 \end{bmatrix}$$

$$s_3 \approx \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}$$

4th Iteration:

- Given the new gradient at x_3 and the new search direction s_3:

$$\nabla F(x_3) = \begin{bmatrix} 8.361 \\ 10.987 \end{bmatrix}, \quad s_3 = \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}$$

- The new Hessian matrix H is:

$$H\big|_{x_1=1.644, x_1=0.42} = \begin{bmatrix} 2.5 + 3 \cdot (0.5 \cdot 1.644 + 0.42)^2 & 1 + 6 \cdot (0.5 \cdot 1.644 + 0.42)^2 \\ 1 + 6 \cdot (0.5 \cdot 1.644 + 0.42)^2 & 4 + 12 \cdot (0.5 \cdot 1.644 + 0.42)^2 \end{bmatrix} = \begin{bmatrix} 7.127 & 10.255 \\ 10.255 & 22.510 \end{bmatrix}$$

- The new step size λ_3 is calculated using the formula:

$$\lambda_3 = -\frac{\nabla F(w_3)^T s_3}{s_3^T H s_3}$$

Substituting the values into the formula:

$$\lambda_3 = -\frac{\begin{bmatrix} 8.361 & 10.987 \end{bmatrix} \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}}{\begin{bmatrix} -11.626 & -16.752 \end{bmatrix} \begin{bmatrix} 7.127 & 10.255 \\ 10.255 & 22.510 \end{bmatrix} \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}}$$

Therefore, the new step size λ_3 is:

$$\lambda_3 \approx 0.024$$

- Using the update rule for the next iteration:

$$x^{(4)} = x^{(3)} + \lambda_3 s_3$$

Given that $x^{(3)} = \begin{bmatrix} 1.644 \\ 0.42 \end{bmatrix}$, $\lambda_3 = 0.024$, and $s_3 = \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}$, we calculate $x^{(4)}$:

$$x^{(4)} = \begin{bmatrix} 1.644 \\ 0.42 \end{bmatrix} + 0.024 \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}$$

$$x^{(4)} = \begin{bmatrix} 1.364 \\ 0.017 \end{bmatrix}$$

- The gradient at the new point x_4 is computed as:

$$\nabla f(x_4) = \left[\frac{\partial F}{\partial w_1}(x_4), \frac{\partial F}{\partial w_2}(x_4) \right]^T$$

$$\begin{aligned} \nabla F(x_4) = &[2 \cdot 1.364 + 0.5 \cdot 1.364 + 0.017 \\ &+2 \cdot (0.5 \cdot 1.364 + 0.017)^3, 2 \cdot 0.017 + 2(0.5 \cdot 1.364 + 0.017) \\ &+4(0.5 \cdot 1.364 + 0.017)^3]^T \\ = &\begin{bmatrix} 4.11 \\ 2.798 \end{bmatrix} \end{aligned}$$

- The new search direction s_4 is given by:

$$s_4 = -\nabla F(x_4) + \omega_4 s_3$$

- The formula for ω_4 is:

$$\omega_4 = \frac{\nabla F(x_4)^T \nabla F(x_4)}{\nabla F(x_3)^T \nabla F(x_3)}$$

Substituting the values into the formula:

$$\omega_4 = \frac{(4.11)^2 + (2.798)^2}{(8.361)^2 + (10.987)^2}$$

$$\omega_4 = \frac{16.892 + 7.828}{69.906 + 120.714}$$

$$\omega_4 = \frac{24.72}{190.62}$$

$$\omega_4 \approx 0.129$$

- Thus, the new search direction s_4 can be calculated as:

$$s_4 \approx \begin{bmatrix} -4.11 \\ -2.798 \end{bmatrix} + 0.129 \begin{bmatrix} -11.626 \\ -16.752 \end{bmatrix}$$

$$s_4 \approx \begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}$$

5th Iteration:

- Given the new gradient at w_4 and the new search direction s_4:

$$\nabla F(x_4) = \begin{bmatrix} 4.11 \\ 2.798 \end{bmatrix}, \quad s_4 = \begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}$$

- The new Hessian matrix H is:

$$H\big|_{x_1=1.364, x_1=0.017} = \begin{bmatrix} 2.5 + 3 \cdot (0.5 \cdot 1.364 + 0.017)^2 & 1 + 6 \cdot (0.5 \cdot 1.364 + 0.017)^2 \\ 1 + 6 \cdot (0.5 \cdot 1.364 + 0.017)^2 & 4 + 12 \cdot (0.5 \cdot 1.364 + 0.017)^2 \end{bmatrix} = \begin{bmatrix} 3.965\ 3.931 \\ 3.931\ 9.863 \end{bmatrix}$$

- The new step size λ_4 is calculated using the formula:

$$\lambda_4 = -\frac{\nabla F(x_4)^T s_4}{s_4^T H s_4}$$

Substituting the values into the formula:

$$\lambda_4 = -\frac{\begin{bmatrix} 4.11\ 2.798 \end{bmatrix} \begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}}{\begin{bmatrix} -5.609\ -4.959 \end{bmatrix} \begin{bmatrix} 3.965\ 3.931 \\ 3.931\ 9.863 \end{bmatrix} \begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}}$$

Therefore, the new step size λ_4 is:

$$\lambda_2 \approx 0.063$$

- Using the update rule for the next iteration:

$$x^{(5)} = x^{(4)} + \lambda_4 s_4$$

Given that $x^{(4)} = \begin{bmatrix} 1.364 \\ 0.017 \end{bmatrix}$, $\lambda_4 = 0.063$, and $\begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}$, we calculate $x^{(4)}$:

$$x^{(4)} = \begin{bmatrix} 1.364 \\ 0.017 \end{bmatrix} + 0.063 \begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}$$

$$x^{(4)} = \begin{bmatrix} 1.010 \\ -0.295 \end{bmatrix}$$

- The gradient at the new point x_5 is computed as:

$$\nabla f(x_5) = \left[\frac{\partial F}{\partial w_1}(x_5), \frac{\partial F}{\partial w_2}(x_5) \right]^T$$

$$\begin{aligned} \nabla F(w_5) = & [2 \cdot 1.010 + 0.5 \cdot 1.010 + (-0.295) \\ & +2 \cdot (0.5 \cdot 1.010 + (-0.295))^3, 2 \cdot (-0.295) + 2(0.5 \cdot 1.010 + (-0.295)) \\ & +4(0.5 \cdot 1.010 + (-0.295))^3]^T \\ = & \begin{bmatrix} 2.248 \\ -0.132 \end{bmatrix} \end{aligned}$$

- The new search direction s_5 is given by:

$$s_5 = -\nabla F(x_5) + \omega_5 s_4$$

- The formula for ω_5 is:

$$\omega_5 = \frac{\nabla F(x_5)^T \nabla F(x_5)}{\nabla F(x_4)^T \nabla F(x_4)}$$

Substituting the values into the formula:

$$\omega_5 = \frac{(2.248)^2 + (-0.132)^2}{(4.11)^2 + (2.798)^2}$$

$$\omega_5 = \frac{5.053 + 0.017}{16.892 + 7.828}$$

$$\omega_5 = \frac{5.070}{24.72}$$

$$\omega_5 \approx 0.205$$

- Thus, the new search direction s_5 can be calculated as:

$$s_5 \approx \begin{bmatrix} -2.248 \\ 0.132 \end{bmatrix} + 0.205 \begin{bmatrix} -5.609 \\ -4.959 \end{bmatrix}$$

$$s_5 \approx \begin{bmatrix} -3.397 \\ -0.884 \end{bmatrix}$$

Observations regarding the convergence of the algorithm are as follows:

1. The convergence towards the minimum $w = (0,0)^T$ appears to be steady but not rapid. Each update makes positive progress towards the minimum, but the rate of approach is moderate.
2. The magnitude of changes in the components of w decreases with each iteration, which is characteristic of the Conjugate Gradient method as it approaches minimum.
3. Given the nature of the function and the starting point far from the minimum, the observed rate of convergence might be considered slow, but it is expected for non-quadratic functions, especially those with higher-order terms.
4. For a more comprehensive analysis of the convergence rate, one would typically consider additional metrics such as the values of the function at each iteration, the norm of the gradient, or the rate of decrease of the function values.

The observed convergence pattern underscores the importance of considering the characteristics of the function and the role of the starting point when assessing the performance of Conjugate Gradient method.

– **Gradient Descent**
Given the function:

$$F(w) = w_1^2 + w_2^2 + (0.5w_1 + w_2)^2 + (0.5w_1 + w_2)^4$$

1st Iteration:

- The gradient is:

$$\nabla F(w_0) = \left[2w_1 + 0.5 \cdot w_1 + w_2 + 2 \cdot (0.5 \cdot w_1 + w_2)^3, 2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3\right]^T$$

For the initial guess $w_0 = [3,3]^T$, the gradient is calculated as:

$$\nabla F(w_0) = [192.75, 379.5]^T$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \, \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For the initial guess $w_0 = [3,3]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|\nabla f(\mathbf{w_0})\| = \sqrt{\left(2\cdot 3+0.5\cdot 3+3+2\cdot(0.5\cdot 3+3)^3\right)^2+\left(2\cdot 3+2\cdot(0.5\cdot 3+3)+4\cdot(0.5\cdot 3+3)^3\right)^2}$$

$$\|\nabla F(w_0)\| = \sqrt{192.75^2+379.5^2}$$
$$\|\nabla F(w_0)\| = \sqrt{181172.812}$$
$$\|\nabla F(w_0)\| \approx 425.644$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_0)}{\|\nabla f(\mathbf{w}_0)\|} = \frac{1}{425.644}\begin{bmatrix}-192.75\\-379.5\end{bmatrix} = \begin{bmatrix}-0.453\\-0.891\end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(1)} = w^{(0)} + \lambda\frac{-\nabla f(\mathbf{w}_0)}{\|\nabla f(\mathbf{w}_0)\|}$$

Given that $w^{(0)} = \begin{bmatrix}3\\3\end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_0)}{\|\nabla f(\mathbf{w}_0)\|} = \begin{bmatrix}-0.453\\-0.891\end{bmatrix}$, we calculate $w^{(1)}$:

$$w^{(1)} = \begin{bmatrix}3\\3\end{bmatrix} + 1\begin{bmatrix}-0.453\\-0.891\end{bmatrix}$$

$$w^{(1)} = \begin{bmatrix}3\\3\end{bmatrix} - \begin{bmatrix}0.453\\0.891\end{bmatrix}$$

$$w^{(1)} = \begin{bmatrix}2.547\\2.109\end{bmatrix}$$

2nd Iteration:

- The gradient at the new point w_1 is computed as:

$$\nabla f(w_1) = \left[\frac{\partial F}{\partial w_1}(w_1), \frac{\partial F}{\partial w_2}(w_1)\right]^T$$

$$\begin{aligned}\nabla F(w_1) =& [2\cdot 2.547+0.5\cdot 2.547+2.109\\ &+2\cdot(0.5\cdot 2.547+2.109)^3, 2\cdot 2.109+2(0.5\cdot 2.547+2.109)\\ &+4(0.5\cdot 2.547+2.109)^3]^T\\ =& \begin{bmatrix}85.876\\165.783\end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{\left(\nabla F(w)\right)^T \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{\left(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3\right)^2 + \left(2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3\right)^2}$$

For $w_1 = [2.547, 2.109]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_1})\| = (\left(2 \cdot 2.547 + 0.5 \cdot 2.547 + 2.109 + 2 \cdot (0.5 \cdot 2.547 + 2.109)^3\right)^2 + \left(2 \cdot 2.109 + 2 \cdot (0.5 \cdot 2.547 + 2.109) + 4 \cdot (0.5 \cdot 2.547 + 2.109)^3\right)^2)^{1/2}$$

$$\|\nabla F(w_1)\| = \sqrt{85.876^2 + 165.783^2}$$

$$\|\nabla F(w_1)\| = \sqrt{34858.69}$$

$$\|\nabla F(w_1)\| \approx 186.704$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_1)}{\|\nabla f(\mathbf{w}_1)\|} = \frac{1}{186.704} \begin{bmatrix} -85.876 \\ -165.783 \end{bmatrix} = \begin{bmatrix} -0.459 \\ -0.887 \end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(2)} = w^{(2)} + \lambda \frac{-\nabla f(\mathbf{w}_1)}{\|\nabla f(\mathbf{w}_1)\|}$$

Given that $w^{(1)} = \begin{bmatrix} 2.547 \\ 2.109 \end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_1)}{\|\nabla f(\mathbf{w}_1)\|} = \begin{bmatrix} -0.459 \\ -0.887 \end{bmatrix}$, we calculate $w^{(2)}$:

$$w^{(2)} = \begin{bmatrix} 2.547 \\ 2.109 \end{bmatrix} + 1 \begin{bmatrix} -0.459 \\ -0.887 \end{bmatrix}$$

$$w^{(2)} = \begin{bmatrix} 2.547 \\ 2.109 \end{bmatrix} - \begin{bmatrix} 0.459 \\ 0.887 \end{bmatrix}$$

$$w^{(2)} = \begin{bmatrix} 2.088 \\ 1.222 \end{bmatrix}$$

3rd Iteration:

- The gradient at the new point w_2 is computed as:

$$\nabla f(w_2) = \left[\frac{\partial F}{\partial w_1}(w_2), \frac{\partial F}{\partial w_2}(w_2)\right]^T$$

$$\begin{aligned}\nabla F(w_2) = &[2 \cdot 2.088 + 0.5 \cdot 2.088 + 1.222 \\ &+2 \cdot (0.5 \cdot 2.088 + 1.222)^3, 2 \cdot 1.222 + 2(0.5 \cdot 2.088 + 1.222) \\ &+4(0.5 \cdot 2.088 + 1.222)^3\big]^T \\ = &\begin{bmatrix} 29.712 \\ 53.517 \end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \, \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_2 = [2.088, 1.222]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_2})\| = \big(2 \cdot 2.088 + 0.5 \cdot 2.088 + 1.222 + 2 \cdot (0.5 \cdot 2.088 + 1.222)^3\big)^2 + \big(2 \cdot 1.222 + 2 \cdot (0.5 \cdot 2.088 + 1.222) + 4 \cdot (0.5 \cdot 2.088 + 1.222)^3\big)^2)^{1/2}$$

$$\|\nabla F(w_2)\| = \sqrt{29.712^2 + 53.517^2}$$

$$\|\nabla F(w_2)\| = \sqrt{3746.872}$$

$$\|\nabla F(w_2)\| \approx 61.211$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_2)}{\|\nabla f(\mathbf{w}_2)\|} = \frac{1}{61.211} \begin{bmatrix} -29.712 \\ -53.517 \end{bmatrix} = \begin{bmatrix} -0.485 \\ -0.874 \end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(3)} = w^{(3)} + \lambda \frac{-\nabla f(\mathbf{w}_2)}{\|\nabla f(\mathbf{w}_2)\|}$$

Given that $w^{(2)} = \begin{bmatrix} 2.088 \\ 1.222 \end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_2)}{\|\nabla f(\mathbf{w}_2)\|} = \begin{bmatrix} -0.485 \\ -0.874 \end{bmatrix}$, we calculate $w^{(3)}$:

$$w^{(3)} = \begin{bmatrix} 2.088 \\ 1.222 \end{bmatrix} + 1 \begin{bmatrix} -0.485 \\ -0.874 \end{bmatrix}$$

$$w^{(3)} = \begin{bmatrix} 2.088 \\ 1.222 \end{bmatrix} - \begin{bmatrix} 0.485 \\ 0.874 \end{bmatrix}$$

$$w^{(3)} = \begin{bmatrix} 1.603 \\ 0.348 \end{bmatrix}$$

4th Iteration:

- The gradient at the new point w_3 is computed as:

$$\nabla f(w_3) = \left[\frac{\partial F}{\partial w_1}(w_3), \frac{\partial F}{\partial w_2}(w_3)\right]^T$$

$$\begin{aligned}\nabla F(w_3) = &[2 \cdot 1.603 + 0.5 \cdot 1.603 + 0.348 \\ &+2 \cdot (0.5 \cdot 1.603 + 0.348)^3, 2 \cdot 0.348 + 2(0.5 \cdot 1.603 + 0.348) \\ &+4(0.5 \cdot 1.603 + 0.348)^3\big]^T \\ = &\begin{bmatrix} 7.393 \\ 9.070 \end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \, \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_3 = [1.603, 0.348]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_3})\| = \big(2 \cdot 1.603 + 0.5 \cdot 1.603 + 0.348 + 2 \cdot (0.5 \cdot 1.603 + 0.348)^3\big)^2 + \big(2 \cdot 0.348 + 2 \cdot (0.5 \cdot 1.603 + 0.348) + 4 \cdot (0.5 \cdot 1.603 + 0.348)^3\big)^2)^{1/2}$$

$$\|\nabla F(w_3)\| = \sqrt{7.393^2 + 9.070^2}$$
$$\|\nabla F(w_3)\| = \sqrt{136.921}$$
$$\|\nabla F(w_3)\| \approx 11.701$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_3)}{\|\nabla f(\mathbf{w}_3)\|} = \frac{1}{11.701}\begin{bmatrix} -7.393 \\ -9.070 \end{bmatrix} = \begin{bmatrix} -0.631 \\ -0.775 \end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(4)} = w^{(3)} + \lambda \frac{-\nabla f(\mathbf{w}_3)}{\|\nabla f(\mathbf{w}_3)\|}$$

Given that $w^{(3)} = \begin{bmatrix} 1.603 \\ 0.348 \end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_3)}{\|\nabla f(\mathbf{w}_3)\|} = \begin{bmatrix} -0.631 \\ -0.775 \end{bmatrix}$, we calculate $w^{(4)}$:

$$w^{(4)} = \begin{bmatrix} 1.603 \\ 0.348 \end{bmatrix} + 1 \begin{bmatrix} -0.631 \\ -0.775 \end{bmatrix}$$

$$w^{(4)} = \begin{bmatrix} 1.603 \\ 0.348 \end{bmatrix} - \begin{bmatrix} 0.631 \\ 0.775 \end{bmatrix}$$

$$w^{(4)} = \begin{bmatrix} 0.972 \\ -0.427 \end{bmatrix}$$

5th Iteration:

- The gradient at the new point w_4 is computed as:

$$\nabla f(w_4) = \left[\frac{\partial F}{\partial w_1}(w_4), \frac{\partial F}{\partial w_2}(w_4)\right]^T$$

$$\begin{aligned}\nabla F(w_4) = &[2 \cdot 0.972 + 0.5 \cdot 0.972 + (-0.427)\\ &+2 \cdot (0.5 \cdot 0.972 + (-0.427))^3, 2 \cdot (-0.427) + 2(0.5 \cdot 0.972 + (-0.427))\\ &+4(0.5 \cdot 0.972 + (-0.427))^3]^T\\ = &\begin{bmatrix} 2.003 \\ -0.735 \end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \, \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_4 = [0.972, -0.427]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_4})\| = \big(2 \cdot 0.972 + 0.5 \cdot 0.972 + (-0.427) + 2 \cdot (0.5 \cdot 0.972 + (-0.427))^3\big)^2 +$$
$$\big(2 \cdot (-0.427) + 2 \cdot (0.5 \cdot 0.972 + (-0.427)) + 4 \cdot (0.5 \cdot 0.972 + (-0.427))^3\big)^2)^{1/2}$$

$$\|\nabla F(w_4)\| = \sqrt{2.003^2 + (-0.735)^2}$$
$$\|\nabla F(w_4)\| = \sqrt{4.552}$$
$$\|\nabla F(w_4)\| \approx 2.133$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_4)}{\|\nabla f(\mathbf{w}_4)\|} = \frac{1}{2.133}\begin{bmatrix} -2.003 \\ 0.735 \end{bmatrix} = \begin{bmatrix} -0.939 \\ 0.344 \end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(5)} = w^{(4)} + \lambda \frac{-\nabla f(\mathbf{w}_4)}{\|\nabla f(\mathbf{w}_4)\|}$$

Given that $w^{(4)} = \begin{bmatrix} 0.972 \\ -0.427 \end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_4)}{\|\nabla f(\mathbf{w}_4)\|} = \begin{bmatrix} -0.939 \\ 0.344 \end{bmatrix}$, we calculate $w^{(5)}$:

$$w^{(5)} = \begin{bmatrix} 0.972 \\ -0.427 \end{bmatrix} + 1\begin{bmatrix} -0.939 \\ 0.344 \end{bmatrix}$$
$$w^{(5)} = \begin{bmatrix} 0.972 \\ -0.427 \end{bmatrix} + \begin{bmatrix} -0.939 \\ 0.344 \end{bmatrix}$$
$$w^{(5)} = \begin{bmatrix} 0.033 \\ -0.083 \end{bmatrix}$$

6th Iteration:

- The gradient at the new point w_5 is computed as:

$$\nabla f(w_5) = \left[\frac{\partial F}{\partial w_1}(w_5), \frac{\partial F}{\partial w_2}(w_5)\right]^T$$

$$\begin{aligned}\nabla F(w_5) = &[2 \cdot 0.033 + 0.5 \cdot 0.033 + (-0.083) \\ &+2 \cdot (0.5 \cdot 0.033 + (-0.083))^3, 2 \cdot (-0.083) + 2(0.5 \cdot 0.033 + (-0.083)) \\ &+4(0.5 \cdot 0.033 + (-0.083))^3]^T \\ = &\begin{bmatrix}-0.001 \\ 0.300\end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_5 = [0.033, -0.083]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_5})\| = \big(2 \cdot 0.033 + 0.5 \cdot 0.033 + (-0.083) + 2 \cdot (0.5 \cdot 0.033 + (-0.083))^3\big)^2 + \big(2 \cdot (-0.083) + 2 \cdot (0.5 \cdot 0.033 + (-0.083)) + 4 \cdot (0.5 \cdot 0.033 + (-0.083))^3\big)^2)^{1/2}$$

$$\|\nabla F(w_5)\| = \sqrt{(-0.001)^2 + 0.300^2}$$
$$\|\nabla F(w_5)\| = \sqrt{0.000001 + 0.09}$$
$$\|\nabla F(w_5)\| = \sqrt{0.09}$$
$$\|\nabla F(w_5)\| \approx 0.3$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_5)}{\|\nabla f(\mathbf{w}_5)\|} = \frac{1}{0.3}\begin{bmatrix}0.001 \\ -0.3\end{bmatrix} = \begin{bmatrix}0.003 \\ -1\end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(6)} = w^{(5)} + \lambda\frac{-\nabla f(\mathbf{w}_5)}{\|\nabla f(\mathbf{w}_5)\|}$$

Given that $w^{(5)} = \begin{bmatrix}0.033 \\ -0.083\end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_5)}{\|\nabla f(\mathbf{w}_5)\|} = \begin{bmatrix}-0.003 \\ -1\end{bmatrix}$, we calculate $w^{(6)}$:

$$w^{(6)} = \begin{bmatrix}0.033 \\ -0.083\end{bmatrix} + 1\begin{bmatrix}0.003 \\ -1\end{bmatrix}$$

$$w^{(6)} = \begin{bmatrix} 0.033 \\ -0.083 \end{bmatrix} + \begin{bmatrix} 0.003 \\ -1 \end{bmatrix}$$

$$w^{(6)} = \begin{bmatrix} 0.036 \\ -1.083 \end{bmatrix}$$

7th Iteration:

- The gradient at the new point w_6 is computed as:

$$\nabla f(w_6) = \left[\frac{\partial F}{\partial w_1}(w_6), \frac{\partial F}{\partial w_2}(w_6)\right]^T$$

$$\begin{aligned}\nabla F(w_6) = &[2 \cdot 0.036 + 0.5 \cdot 0.036 + (-1.083) \\ &+2 \cdot (0.5 \cdot 0.036 + (-1.083))^3, 2 \cdot (-1.083) + 2(0.5 \cdot 0.036 + (-1.083)) \\ &+4(0.5 \cdot 0.036 + (-1.083))^3]^T \\ = &\begin{bmatrix} -3.408 \\ -9.127 \end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \, \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_6 = [0.036, -1.083]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_6})\| = \big(2 \cdot 0.036 + 0.5 \cdot 0.036 + (-1.083) + 2 \cdot (0.5 \cdot 0.036 + (-1.083))^3\big)^2 + \\ \big(2 \cdot (-1.083) + 2 \cdot (0.5 \cdot 0.036 + (-1.083)) + 4 \cdot (0.5 \cdot 0.036 + (-1.083))^3\big)^2)^{1/2}$$

$$\|\nabla F(w_6)\| = \sqrt{(-3.408)^2 + (-9.127)^2}$$

$$\|\nabla F(w_6)\| = \sqrt{11.614 + 83.302}$$

$$\|\nabla F(w_6)\| = \sqrt{94.916}$$

$$\|\nabla F(w_6)\| \approx 9.742$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_6)}{\|\nabla f(\mathbf{w}_6)\|} = \frac{1}{9.742} \begin{bmatrix} 3.408 \\ 9.127 \end{bmatrix} = \begin{bmatrix} 0.349 \\ 0.936 \end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(7)} = w^{(6)} + \lambda \frac{-\nabla f(\mathbf{w}_6)}{\|\nabla f(\mathbf{w}_6)\|}$$

Given that $w^{(6)} = \begin{bmatrix} 0.036 \\ -1.083 \end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_6)}{\|\nabla f(\mathbf{w}_6)\|} = \begin{bmatrix} 0.349 \\ 0.936 \end{bmatrix}$, we calculate $w^{(7)}$:

$$w^{(7)} = \begin{bmatrix} 0.036 \\ -1.083 \end{bmatrix} + 1 \begin{bmatrix} 0.349 \\ 0.936 \end{bmatrix}$$

$$w^{(7)} = \begin{bmatrix} 0.036 \\ -1.083 \end{bmatrix} + \begin{bmatrix} 0.349 \\ 0.936 \end{bmatrix}$$

$$w^{(7)} = \begin{bmatrix} 0.385 \\ -0.147 \end{bmatrix}$$

8th Iteration:

- The gradient at the new point w_7 is computed as:

$$\nabla f(w_7) = \left[\frac{\partial F}{\partial w_1}(w_7), \frac{\partial F}{\partial w_2}(w_7) \right]^T$$

$$\begin{aligned} \nabla F(w_7) = &[2 \cdot 0.385 + 0.5 \cdot 0.385 + (-0.147) \\ &+2 \cdot (0.5 \cdot 0.385 + (-0.147))^3, 2 \cdot (-0.147) + 2(0.5 \cdot 0.385 + (-0.147)) \\ &+4(0.5 \cdot 0.385 + (-0.147))^3\big]^T \\ = &\begin{bmatrix} 0.815 \\ -0.202 \end{bmatrix} \end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_7 = [0.385, -0.147]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_7})\| = \big(2 \cdot 0.385 + 0.5 \cdot 0.385 + (-0.147) + 2 \cdot (0.5 \cdot 0.385 + (-0.147))^3\big)^2 + \big(2 \cdot (-0.147) + 2 \cdot (0.5 \cdot 0.385 + (-0.147)) + 4 \cdot (0.5 \cdot 0.385 + (-0.147))^3\big)^2)^{1/2}$$

$$\|\nabla F(w_7)\| = \sqrt{0.815^2 + (-0.202)^2}$$
$$\|\nabla F(w_7)\| = \sqrt{0.705}$$

$$\|\nabla F(w_7)\| \approx 0.839$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_7)}{\|\nabla f(\mathbf{w}_7)\|} = \frac{1}{0.839}\begin{bmatrix}-0.385\\0.147\end{bmatrix} = \begin{bmatrix}-0.458\\0.175\end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(8)} = w^{(7)} + \lambda\frac{-\nabla f(\mathbf{w}_7)}{\|\nabla f(\mathbf{w}_7)\|}$$

Given that $w^{(7)} = \begin{bmatrix}0.385\\-0.147\end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_7)}{\|\nabla f(\mathbf{w}_7)\|} = \begin{bmatrix}-0.458\\0.175\end{bmatrix}$, we calculate $w^{(8)}$:

$$w^{(8)} = \begin{bmatrix}0.385\\-0.147\end{bmatrix} + 1\begin{bmatrix}-0.458\\0.175\end{bmatrix}$$

$$w^{(8)} = \begin{bmatrix}0.385\\-0.147\end{bmatrix} + \begin{bmatrix}-0.458\\0.175\end{bmatrix}$$

$$w^{(8)} = \begin{bmatrix}-0.073\\0.028\end{bmatrix}$$

9th Iteration:

- The gradient at the new point w_8 is computed as:

$$\nabla f(w_8) = \left[\frac{\partial F}{\partial w_1}(w_8), \frac{\partial F}{\partial w_2}(w_8)\right]^T$$

$$\begin{aligned}\nabla F(w_8) = &[2\cdot(-0.073) + 0.5\cdot(-0.073) + 0.028\\ &+2\cdot(0.5\cdot(-0.073)+0.028)^3, 2\cdot 0.028 + 2(0.5\cdot(-0.073)+0.028)\\ &+4(0.5\cdot(-0.073)+0.028)^3]^T\\ =&\begin{bmatrix}-0.154\\0.038\end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T\,\nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1+w_2)^3)^2 + (2w_2 + 2(0.5w_1+w_2) + 4(0.5w_1+w_2)^3)^2}$$

For $w_8 = [-0.073, 0.028]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$\|f(\mathbf{w_8})\| = \big(2\cdot(-0.073) + 0.5\cdot(-0.073) + 0.028 + 2\cdot(0.5\cdot(-0.073)+0.028)^3\big)^2 +$
$\big(2\cdot 0.028 + 2\cdot(0.5\cdot(-0.073)+0.028) + 4\cdot(0.5\cdot(-0.073)+0.028)^3\big)^2\big)^{1/2}$

$$\|\nabla F(w_8)\| = \sqrt{(-0.154)^2 + 0.038^2}$$

$$\|\nabla F(w_8)\| = \sqrt{0.025}$$

$$\|\nabla F(w_8)\| \approx 0.158$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_8)}{\|\nabla f(\mathbf{w}_8)\|} = \frac{1}{0.158}\begin{bmatrix}0.154\\-0.038\end{bmatrix} = \begin{bmatrix}0.974\\-0.24\end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(9)} = w^{(8)} + \lambda\frac{-\nabla f(\mathbf{w}_8)}{\|\nabla f(\mathbf{w}_8)\|}$$

Given that $w^{(8)} = \begin{bmatrix}-0.073\\0.028\end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_8)}{\|\nabla f(\mathbf{w}_8)\|} = \begin{bmatrix}0.974\\-0.24\end{bmatrix}$, we calculate $w^{(9)}$:

$$w^{(9)} = \begin{bmatrix}-0.073\\0.028\end{bmatrix} + 1\begin{bmatrix}0.974\\-0.24\end{bmatrix}$$

$$w^{(9)} = \begin{bmatrix}-0.073\\0.028\end{bmatrix} + \begin{bmatrix}0.974\\-0.24\end{bmatrix}$$

$$w^{(9)} = \begin{bmatrix}0.901\\-0.212\end{bmatrix}$$

10th Iteration:

- The gradient at the new point w_9 is computed as:

$$\nabla f(w_9) = \left[\frac{\partial F}{\partial w_1}(w_9), \frac{\partial F}{\partial w_2}(w_9)\right]^T$$

$$\begin{aligned}\nabla F(w_8) = &[2\cdot 0.901 + 0.5\cdot 0.901 + (-0.212)\\ &+2\cdot(0.5\cdot 0.901 + (-0.212))^3, 2\cdot(-0.212) + 2(0.5\cdot 0.901 + (-0.212))\\ &+4(0.5\cdot 0.901 + (-0.212))^3\big]^T\\ = &\begin{bmatrix}2.067\\0.107\end{bmatrix}\end{aligned}$$

- Thus $\|\nabla f(\mathbf{w})\|$ is calculated:

$$\|\nabla F(w)\| = \sqrt{(\nabla F(w))^T \, \nabla F(w)}$$

$$\|\nabla F(w)\| = \sqrt{(2w_1 + 0.5w_1 + w_2 + 2(0.5w_1 + w_2)^3)^2 + (2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3)^2}$$

For $w_9 = [0.901, (-0.212)]^T$ $\|\nabla f(\mathbf{x})\|$ is calculated:

$$\|f(\mathbf{w_9})\| = \big(2 \cdot 0.901 + 0.5 \cdot 0.901 + (-0.212) + 2 \cdot (0.5 \cdot 0.901 + (-0.212))^3 + 2 \cdot (-0.212 + 2(0.5 \cdot 0.901$$

$$\|\nabla F(w_9)\| = \sqrt{2.067^2 + 0.107^2}$$

$$\|\nabla F(w_9)\| = \sqrt{4.283}$$

$$\|\nabla F(w_9)\| \approx 2.069$$

Direction of descending:

$$\frac{-\nabla f(\mathbf{w}_8)}{\|\nabla f(\mathbf{w}_8)\|} = \frac{1}{2.069} \begin{bmatrix} -2.067 \\ -0.107 \end{bmatrix} = \begin{bmatrix} -0.999 \\ -0.051 \end{bmatrix}$$

- Using the update rule for the next iteration:

$$w^{(10)} = w^{(9)} + \lambda \frac{-\nabla f(\mathbf{w}_9)}{\|\nabla f(\mathbf{w}_9)\|}$$

Given that $w^{(9)} = \begin{bmatrix} 0.901 \\ -0.212 \end{bmatrix}$, $\lambda = 1$, and $\frac{-\nabla f(\mathbf{w}_9)}{\|\nabla f(\mathbf{w}_9)\|} = \begin{bmatrix} -0.999 \\ -0.051 \end{bmatrix}$, we calculate $w^{(10)}$:

$$w^{(10)} = \begin{bmatrix} 0.901 \\ -0.212 \end{bmatrix} + 1 \begin{bmatrix} -0.999 \\ -0.051 \end{bmatrix}$$

$$w^{(10)} = \begin{bmatrix} 0.901 \\ -0.212 \end{bmatrix} - \begin{bmatrix} 0.999 \\ 0.051 \end{bmatrix}$$

$$w^{(10)} = \begin{bmatrix} -0.098 \\ -0.263 \end{bmatrix}$$

We observe that in iteration 5 and 8 we reach close to the minimum point $[0, 0]$ and we can end the method there.

2 PROBLEM-02

```
import numpy as np

def f(x):
  w1, w2 = x[0], x[1]
  return w1**2 + w2**2 + (0.5*w1 + w2)**2 + (0.5*w1 + w2)**4

def gradient_f(x):
  w1, w2 = x[0], x[1]
  df_dw1 = 2*w1 + 0.5*w1 + w2 + 2*(0.5*w1 + w2)**3
  df_dw2 = 2*w2 + 2*(0.5*w1 + w2) + 4*(0.5*w1 + w2)**3
  return np.array([df_dw1, df_dw2])

def hessian_f(x):
  w1, w2 = x[0], x[1]
  d2f_dw1w1 = 2.5 + 3*(0.5*w1 + w2)**2
  d2f_dw1w2 = 1 + 6*(0.5*w1 + w2)**2
  d2f_dw2w1 = d2f_dw1w2  # Hessian is symmetric
  d2f_dw2w2 = 4 + 12*(0.5*w1 + w2)**2
  return np.array([[d2f_dw1w1, d2f_dw1w2], [d2f_dw2w1, d2f_dw2w2
    ]])

def newtons_method(x0, max_iter, tol):
  x_k = np.array(x0)

  for i in range(max_iter):
    grad_k = np.array(gradient_f(x_k))
    hess_k = np.array(hessian_f(x_k))

    # Step 2: Compute the minimizing direction
    s_k = -np.linalg.solve(hess_k, grad_k)

    # Step 3: Minimize f along this direction (line search)
    # For a quadratic function, lambda_k can be chosen as 1
    lambda_k = 1

    # Step 4: Update the point
    x_kplus1 = x_k + lambda_k * s_k

    # Print x value and gradient value
    print(f"x value: {x_kplus1}")
    print(f"gradient value: {grad_k}")
    print(f"Iteration: {i}\n")

    # Step 5: Check convergence criterion
    if np.linalg.norm(x_kplus1 - x_k, ord=2) < tol:
      print(f"Converged at iteration {i}")
      return x_kplus1
    x_k = x_kplus1

  print("Maximum iterations reached without convergence.")
  return x_k

# Usage
initial_guess = [3.0, 3.0]
max_iterations = 100
tolerance = 1e-12
optimal_point = newtons_method(initial_guess, max_iterations,
    tolerance)

print("Optimal point found:", optimal_point)
```

Applying the Newton's method to the function $F(w) = w_1^2 + w_2^2 + (0.5w_1 + w_2)^2 + (0.5w_1 + w_2)^4$, the algorithm converges to the minimum at the point $\mathbf{x}^* = (0, 0)$ after 8 iterations.

3 PROBLEM-03

The initial weights and biases are chosen to be:

$$w^1(0) = -3, \quad b^1(0) = 2, \quad w^2(0) = -1, \quad b^2(0) = -1.$$

An input/target pair is given to be:

$$p = 1, \quad t = 0,$$

and the parameter of Leaky ReLU is equal to 0.001 and learning rate $\alpha = 1$.

1st Iteration:

Feed-Forward Phase:

$n^1 = w^1 \cdot p + b^1 = ((-3) \cdot (1) + 2 = -1$

$$\begin{aligned} a^1 &= \text{Swish}(w^1 \cdot p + b^1) \\ &= \frac{w^1 \cdot p + b^1}{1 + e^{-(w^1 \cdot p + b^1)}} \\ &= \frac{-3 \cdot 1 + 2}{1 + e^{-(-3 \cdot 1 + 2)}} \\ &= \frac{-1}{1 + e^{3 \cdot 1 - 2}} \\ &= \frac{-1}{1 + e^1} \\ &= -0.269 \end{aligned}$$

since $n^2 = w^2 \cdot a^1 + b^2 = ((-1) \cdot (-0.269) + (-1) = -0.731 < 0$

$$\begin{aligned} a^2 &= \text{LReLU}(w^2 \cdot a^1 + b^2) \\ &= 0.001 \cdot (w^2 \cdot a^1 + b^2) \\ &= 0.001 \cdot ((-1) \cdot (-0.269) + (-1)) \\ &= -0.000731 \end{aligned}$$

Compute Error:

$$\text{error} = e = t - \text{output} = 1 + 0.000731 = 1.000731$$

Backpropagation:

$$\begin{aligned} s^2 &= -2 \cdot (F^2)'(n^2) \cdot (t - a) \\ &= -2 \cdot 0.001 \cdot 1.000731 \\ &= -0.002 \end{aligned}$$

$$\begin{aligned} s^1 &= (F^1)' \cdot (W^2)^T s^2 \\ &= \frac{1 + e^{-n^1} + n^1 \cdot e^{-n^1}}{(1 + e^{-n^1})^2} \cdot (W^2)^T s^2 \\ &= \frac{1 + e^1 + (-1) \cdot e^1}{(1 + e^1)^2} \cdot (-1) \cdot (-0.002) \\ &= 0.00014 \end{aligned}$$

Update weights and biases:

$$\begin{aligned} W^2(1) &= W^2(0) - \alpha \cdot s^2 \cdot (a^1)^T = -1 - 1 \cdot (-0.002) \cdot (-0.269) = -1.000538 \approx -1 \\ b^2(1) &= b^2(0) - \alpha \cdot s^2 = -1 - 1 \cdot (-0.002) = -0.998 \approx -1 \\ W^1(1) &= W^1(0) - \alpha \cdot s^1 \cdot (p)^T = -3 - 1 \cdot 0.00014 \cdot 1 = -3.00014 \approx -3 \\ b^1(1) &= b^1(0) - \alpha \cdot s^1 = 2 - 1 \cdot 0.00014 = 1.99986 \approx 2 \end{aligned}$$

2nd Iteration:

Feed-Forward Phase:

$$\begin{aligned} a^1 &= \text{Swish}(w^1 \cdot p + b^1) \\ &= \frac{w^1 \cdot p + b^1}{1 + e^{-(w^1 \cdot p + b^1)}} \\ &= \frac{-3 \cdot 1 + 2}{1 + e^{-(-3 \cdot 1 + 2)}} \\ &= \frac{-1}{1 + e^{3 \cdot 1 - 2}} \\ &= \frac{-1}{1 + e^1} \\ &= -0.269 \end{aligned}$$

since $n^2 = w^2 \cdot a^1 + b^2 = ((-1) \cdot (-0.269) + (-1) = -0.731 < 0$

$$\begin{aligned} a^2 &= \text{LReLU}(w^2 \cdot a^1 + b^2) \\ &= 0.001 \cdot (w^2 \cdot a^1 + b^2) \\ &= 0.001 \cdot ((-1) \cdot (-0.269) + (-1)) \\ &= -0.000731 \end{aligned}$$

Compute Error:

$$\text{error} = e = t - \text{output} = 1 + 0.000731 = 1.000731$$

Backpropagation:

$$\begin{aligned} s^2 &= -2 \cdot (F^2)'(n^2) \cdot (t - a) \\ &= -2 \cdot 0.001 \cdot 1.000731 \\ &= -0.002 \end{aligned}$$

$$\begin{aligned} s^1 &= (F^1)' \cdot (W^2)^T s^2 \\ &= \frac{1 + e^{-n^1} + n^1 \cdot e^{-n^1}}{(1 + e^{-n^1})^2} \cdot (W^2)^T s^2 \\ &= \frac{1 + e^1 + (-1) \cdot e^1}{(1 + e^1)^2} \cdot (-1) \cdot (-0.002) \\ &= 0.00014 \end{aligned}$$

Update weights and biases:

$$W^2(1) = W^2(0) - \alpha \cdot s^2 \cdot (a^1)^T = -1 - 1 \cdot (-0.002) \cdot (-0.269) = -1.000538 \approx -1$$
$$b^2(1) = b^2(0) - \alpha \cdot s^2 = -1 - 1 \cdot (-0.002) = -0.998 \approx -1$$
$$W^1(1) = W^1(0) - \alpha \cdot s^1 \cdot (p)^T = -3 - 1 \cdot 0.00014 \cdot 1 = -3.00014 \approx -3$$
$$b^1(1) = b^1(0) - \alpha \cdot s^1 = 2 - 1 \cdot 0.00014 = 1.99986 \approx 2$$

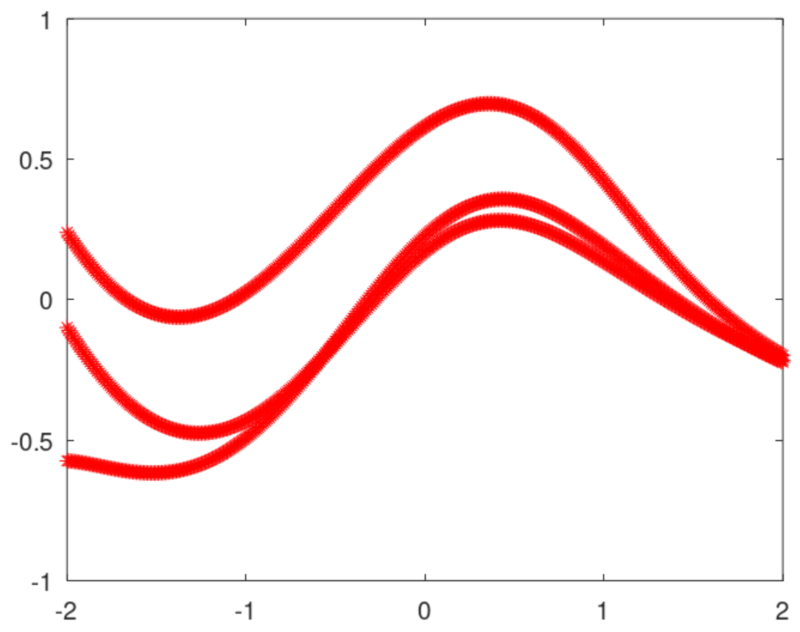
4 PROBLEM-04

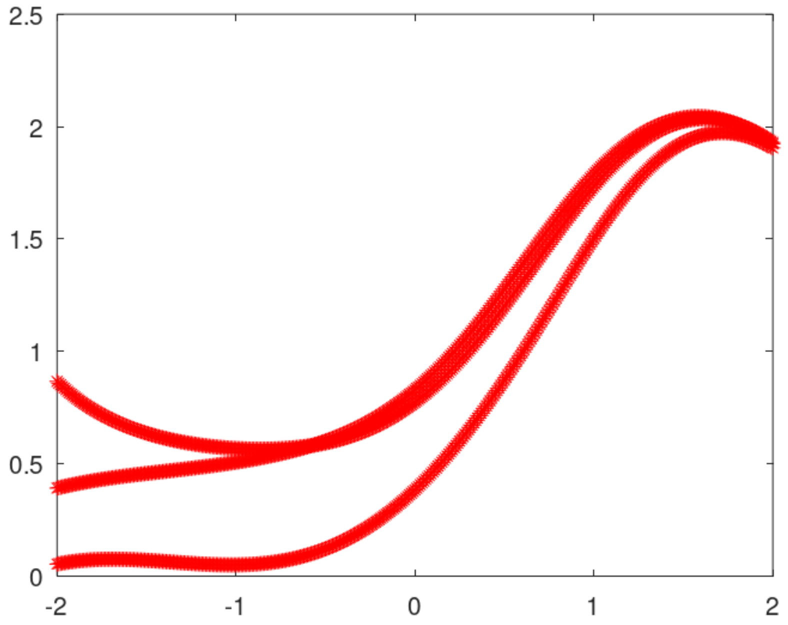
We have 3 plots for each hidden layer architecture. The first one is for 2 neurons in 1 hidden layer. The second one is for 10 neurons in 1 hidden layer. The second one is for 12 neurons in 1 hidden layer. For each layer, we are going to plot:

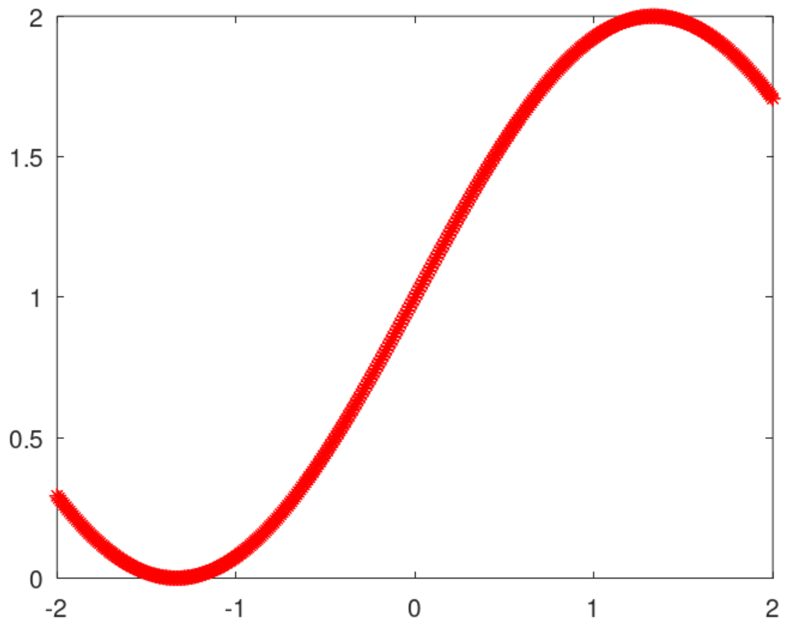
1. The error of the approximation.
2. The approximation according to the input function.
3. The real value of the function.

4.1 2 neurons in 1 hidden layer

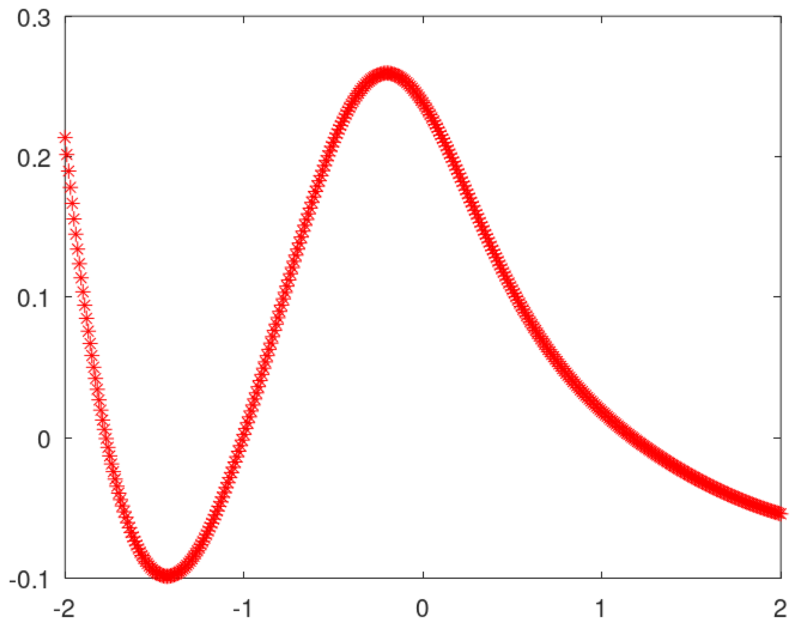
- a = 0.01

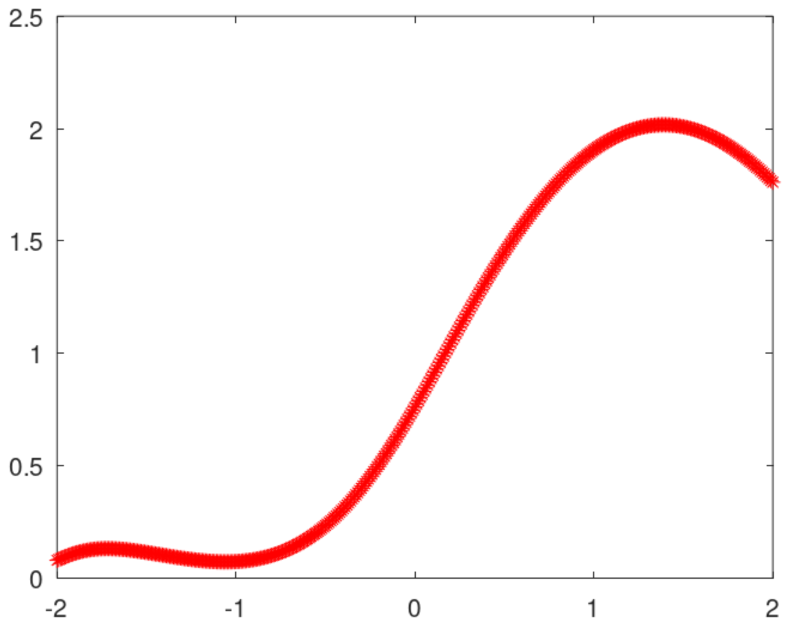


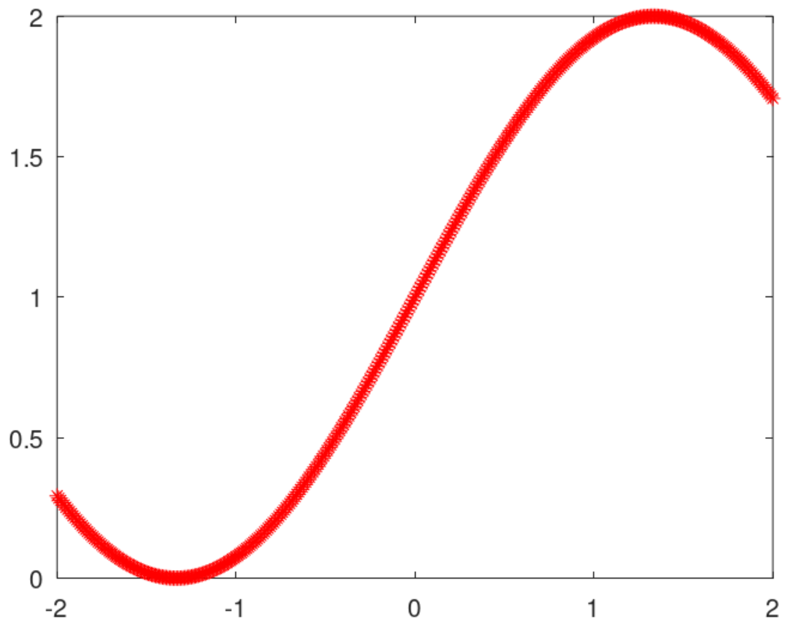

2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



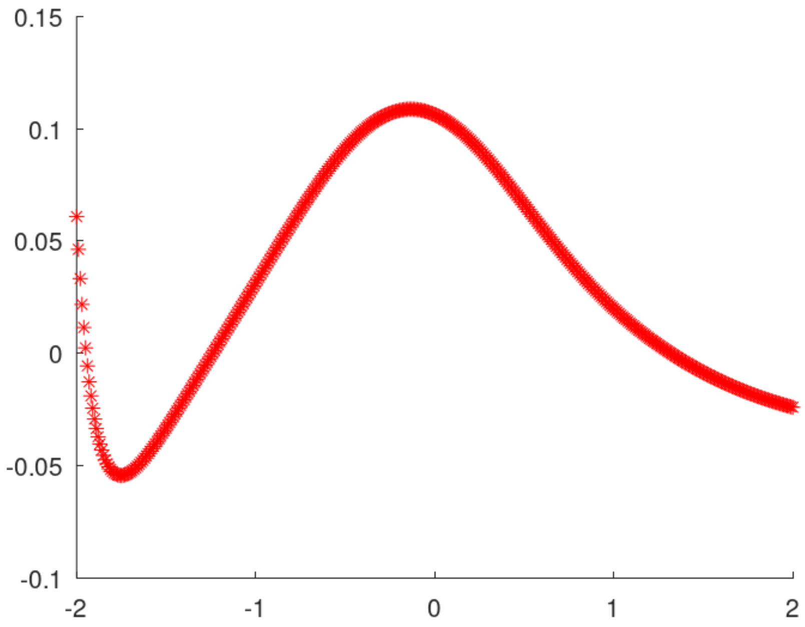
- a = 0.05


0.3
0.2
0.1
0
-0.1
-2
-1
0
1
2

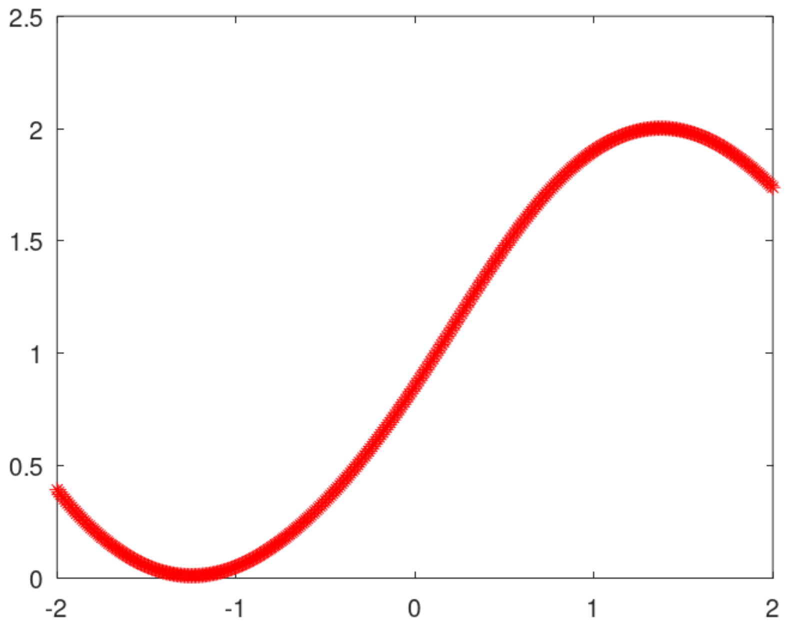

2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

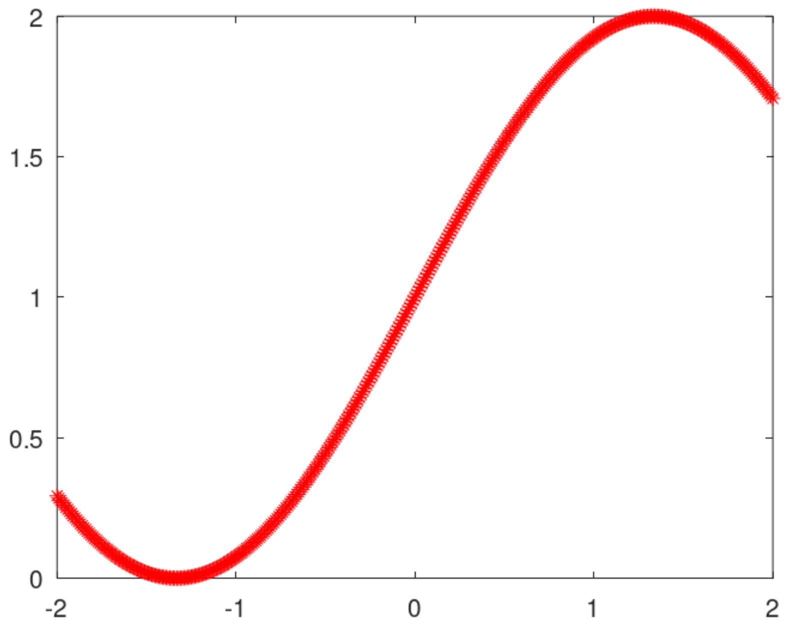


– a = 0.1

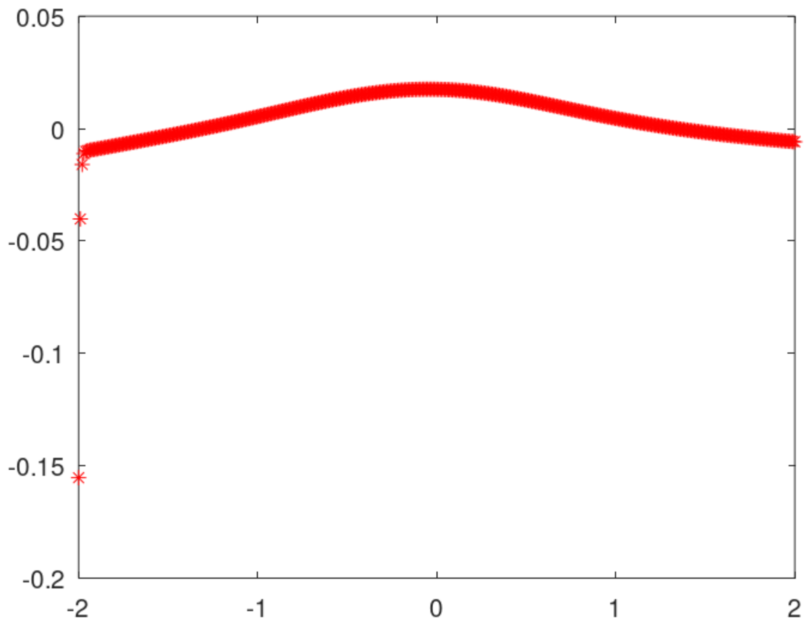


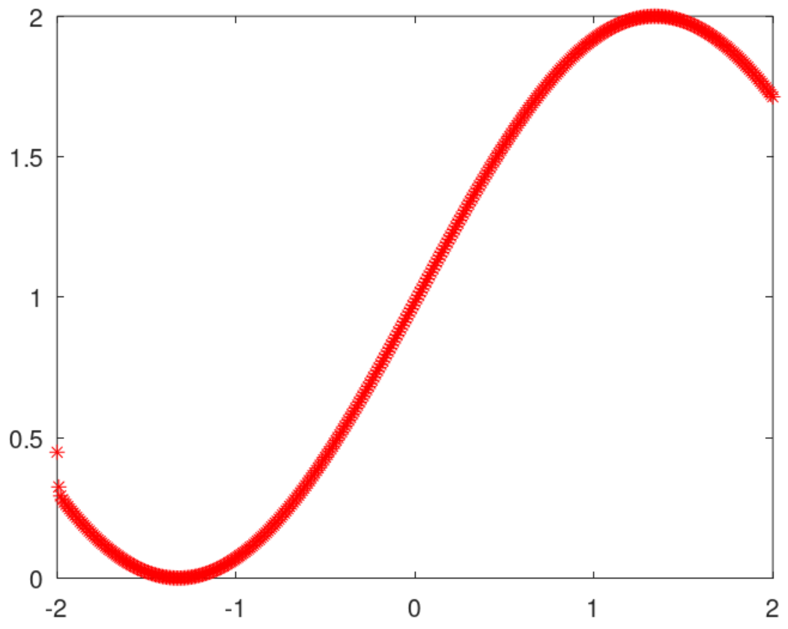
0.15
0.1
0.05
0
-0.05
-0.1
-2
-1
0
1
2


2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

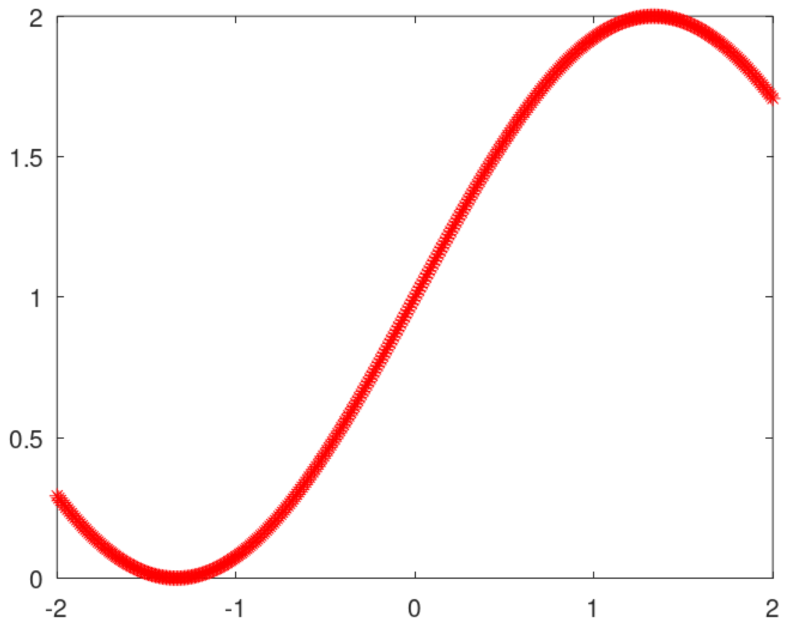


- a = 0.5

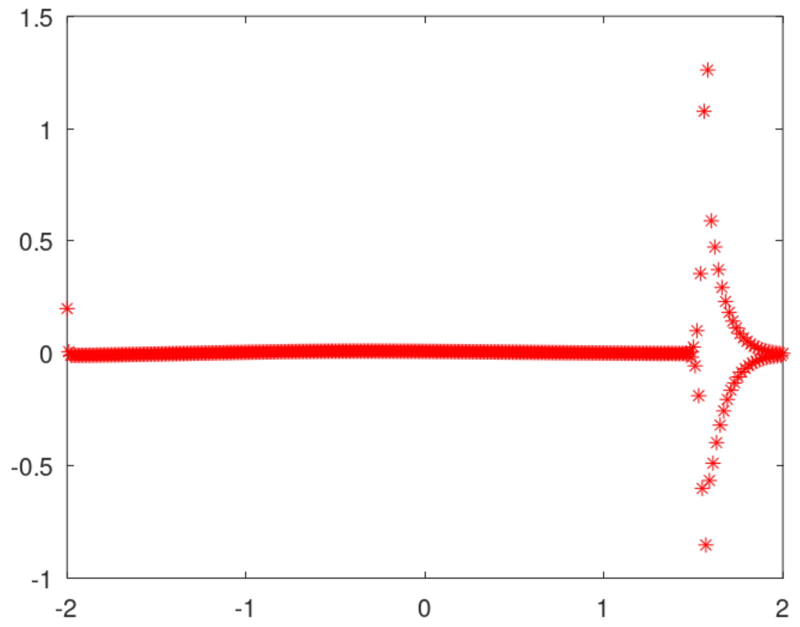

0.05
0
-0.05
-0.1
-0.15
-0.2
-2
-1
0
1
2



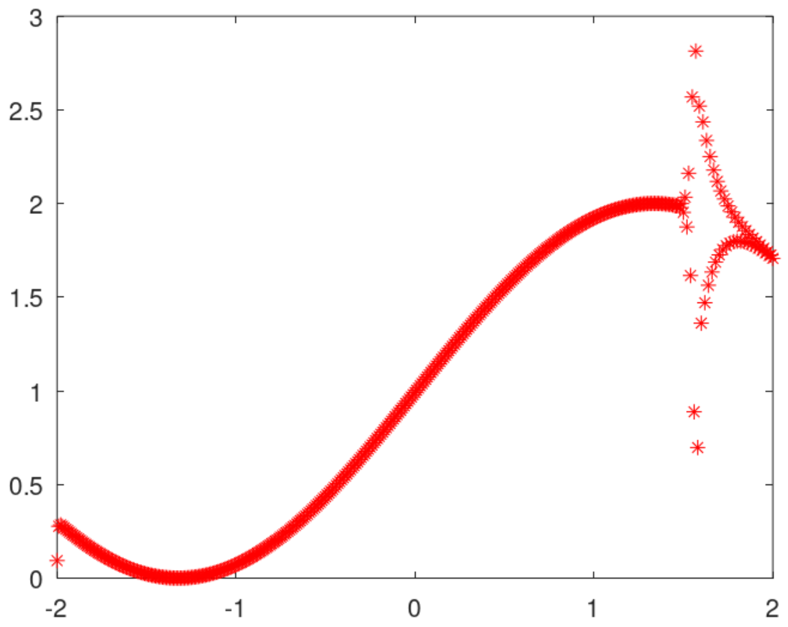
2
1.5
1
0.5
0
-2
-1
0
1
2



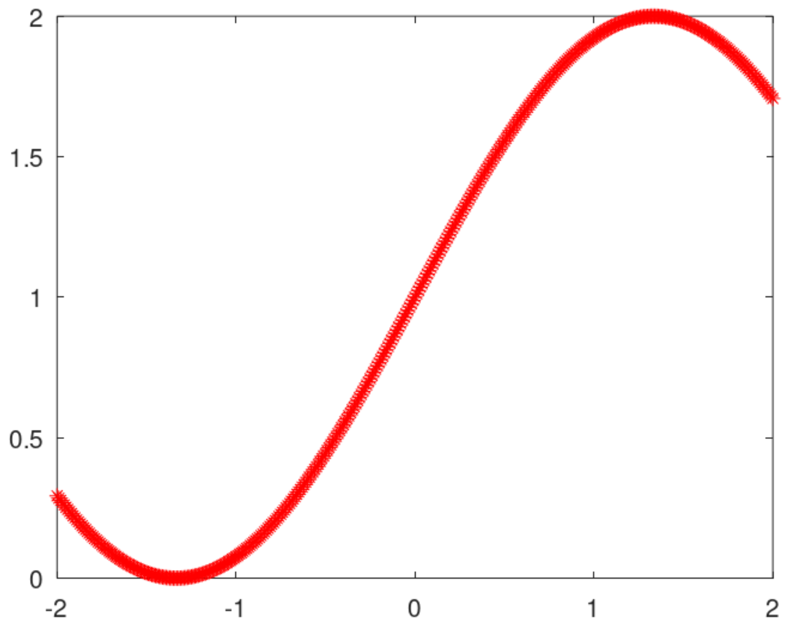
– a = 1



1.5
1
0.5
0
-0.5
-1
-2
-1
0
1
2



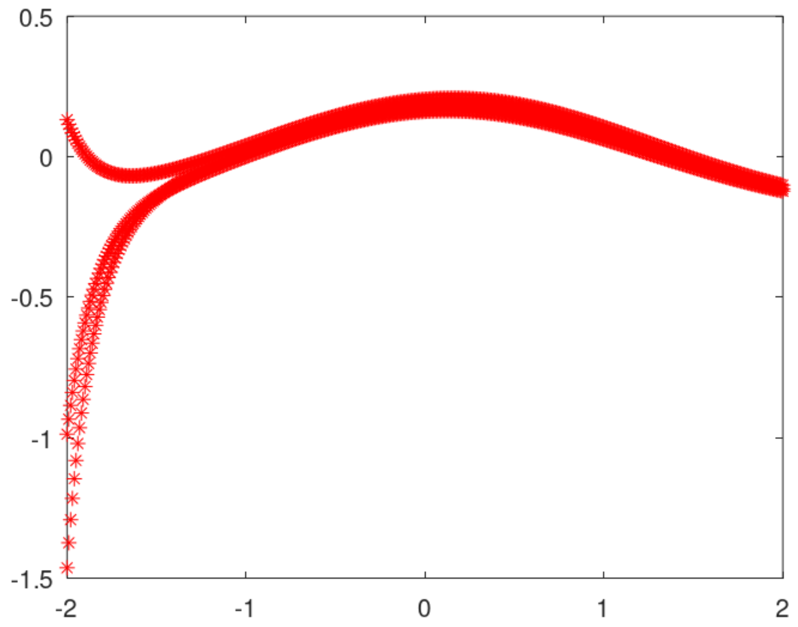
3
2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

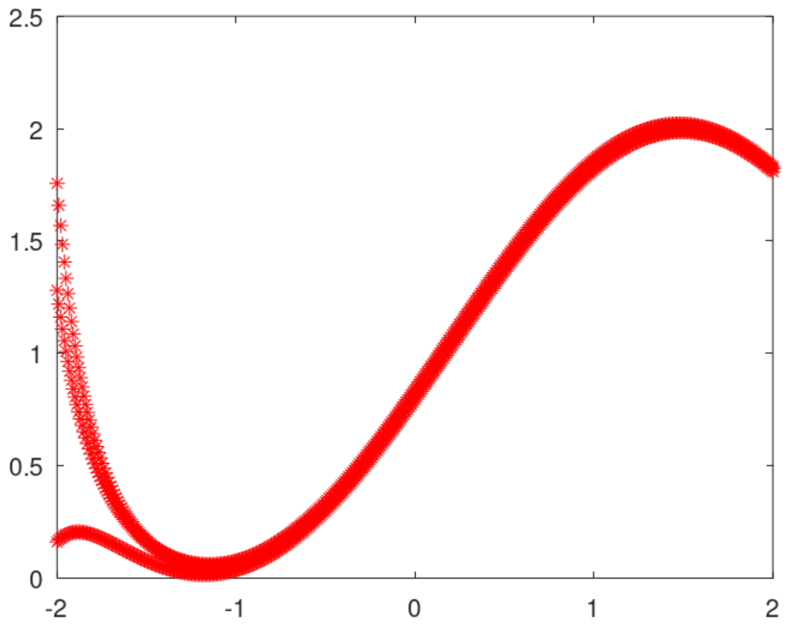


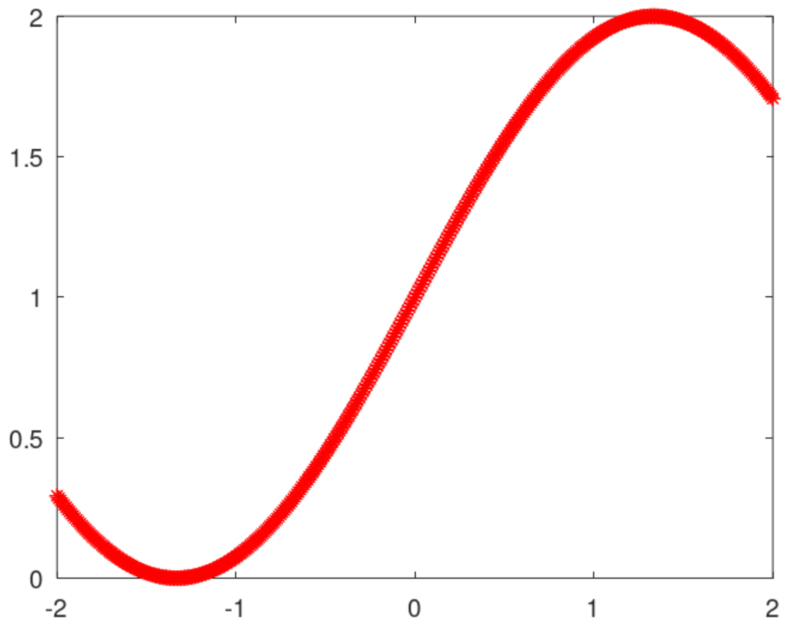
```
% Initialize network parameters
Max = 0.5;
Min = -Max;
interval = Max - Min;
lr = 0.01;
epochs = 1000;
global k;
k = 1;

% Initialize Network Function
function [W1, B1, W2, B2] = initialize_network(Min, interval)
    W1(:, 1) = Min + rand(1, 2) * interval;
    B1(:, 1) = Min + rand(1, 2) * interval;
    W2(1, :) = (Min + rand(1, 2) * interval)';
    B2(1) = Min + rand(1, 1) * interval;
end

% Train Network Function
function [W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr,
    epochs, interval)
    global k;
    while true
        for p = -2:0.01:2
            % Feed-forward
            [A1, A2] = forward_pass(W1, B1, W2, B2, p);

            % Calculate error
            error = 1 + sin(p * (3 * pi / 8)) - A2(k);

            % Backpropagation and update weights
            [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2,
    A1, A2, lr, p, error);

            % Plotting
            plot_data(p, error, A2);
            k = k + 1;
        end
        if abs(error) < 0.1 || k > epochs
            save_plots();
            return;
        end
    end
end

function y = logsig(n1)
    global k;
    y(:, k) = 1 ./ (1 + exp(-n1));
end

function y = relu(n2)
    y = max(0, n2);
end

% Forward Pass Function
function [A1, A2] = forward_pass(W1, B1, W2, B2, p, A1, A2)
    global k;
    n1 = (W1(:, k) .* p + B1(:, k));
    A1 = logsig(n1);
    n2 = W2(k, :) * A1(:, k) + B2(1);
    A2(k) = relu(n2);
end

% Backpropagation Function
function [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2, A1,
    A2, lr, p, error)
    global k;
    if(A2(k) > 0)
        derivative_2 = 1;
    else
        derivative_2 = 0;
    endif
    derivative = arrayfun(@(x) (1 - A1(x, k)) * A1(x, k), 1:2);
    R = diag(derivative);
    s2 = -2 * derivative_2 * error;
    s1 = R * (W2(k, :)' * s2);
    W1(:, k + 1) = W1(:, k) - lr * s1 * p;
    B1(:, k + 1) = B1(:, k) - lr * s1;
    W2(k + 1, :) = W2(k, :) - lr * s2 * A1(:, k)';
    B2(k + 1) = B2(k) - lr * s2;
end

% Plotting Function
function plot_data(p, error, A2)
    global k;
    figure(1); hold on; plot(p, error, '*r');
    figure(2); hold on; plot(p, A2(k), '*r');
    figure(3); hold on; plot(p, 1 + sin(p * (3 * pi / 8)), '*r')
    ;
end

% Save Plots Function
function save_plots()
    figure(1); print -dpng 'error_plot.png';
    figure(2); print -dpng 'approximation_plot.png';
    figure(3); print -dpng 'real_value_plot';
end

% Initialize weights and biases
[W1, B1, W2, B2] = initialize_network(Min, interval);
% Training the network
[W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr, epochs,
    interval);
```

4.2 10 neurons in 1 hidden layer

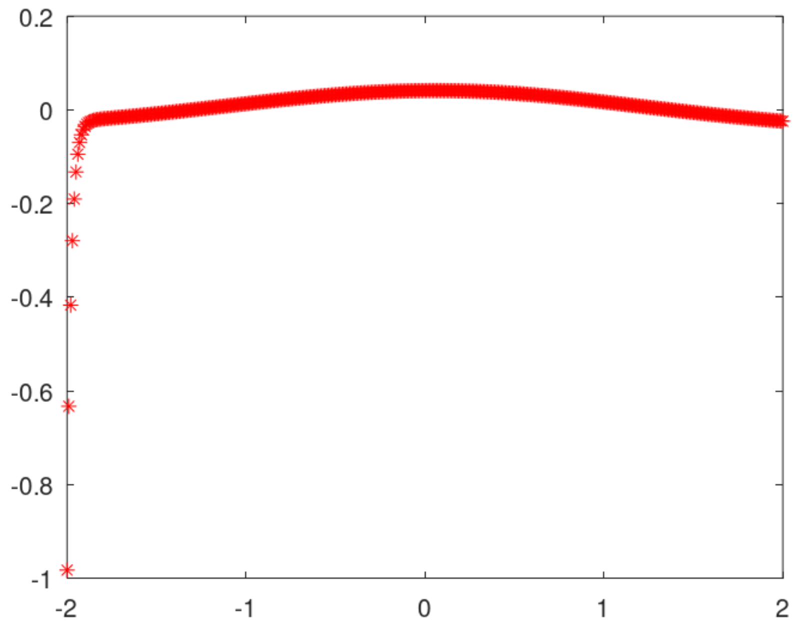
- a = 0.01

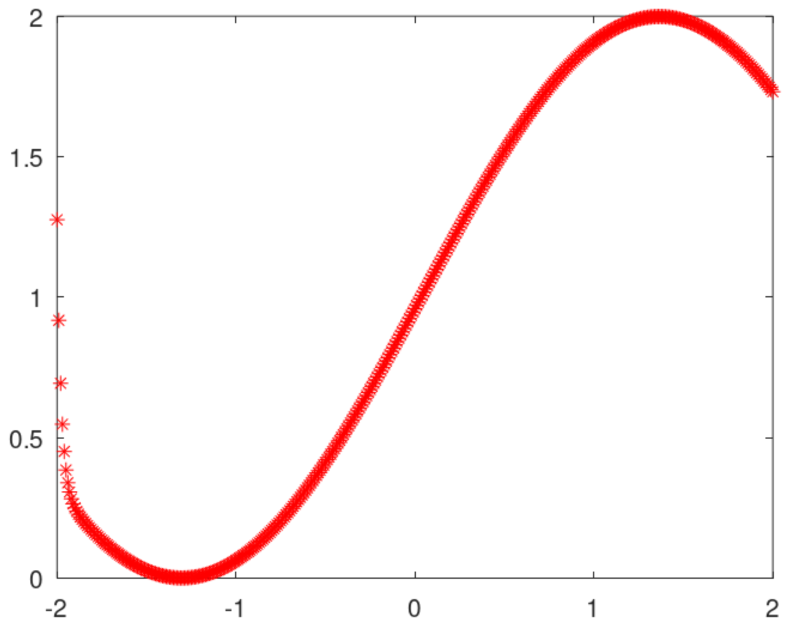

0.5
0
-0.5
-1
-1.5
-2
-1
0
1
2


2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

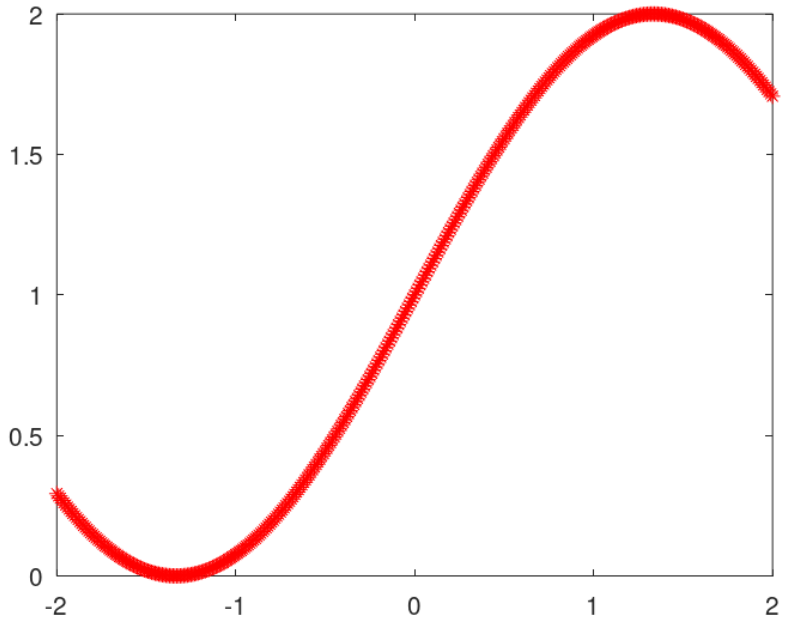


– a = 0.05

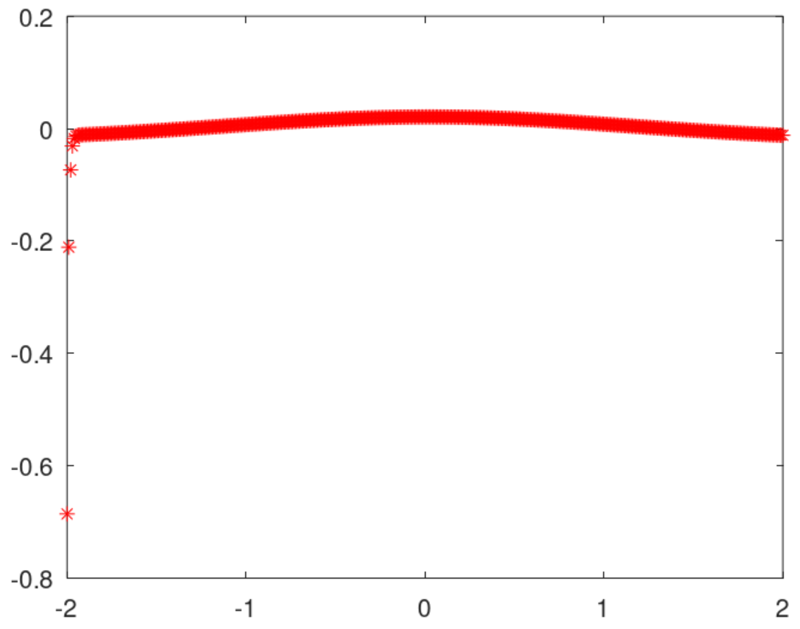

0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-2
-1
0
1
2

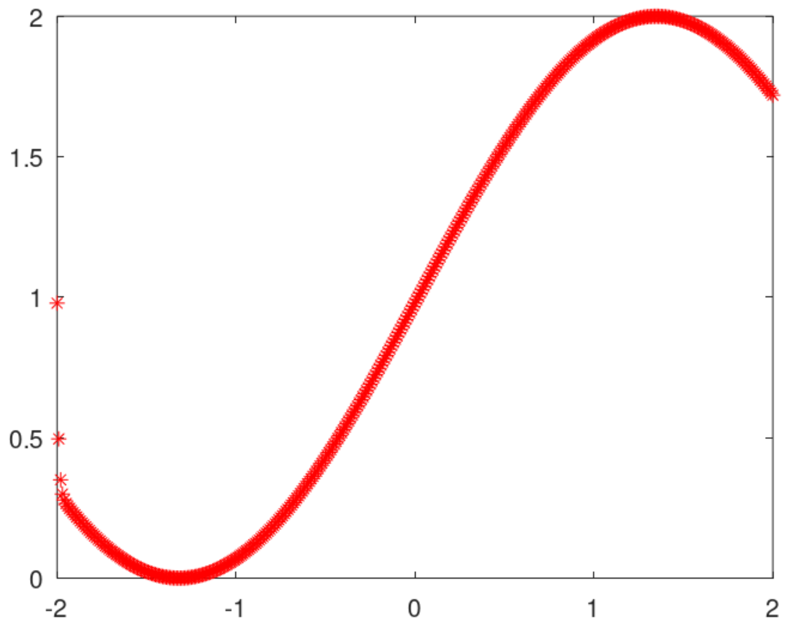


2
1.5
1
0.5
0
-2
-1
0
1
2

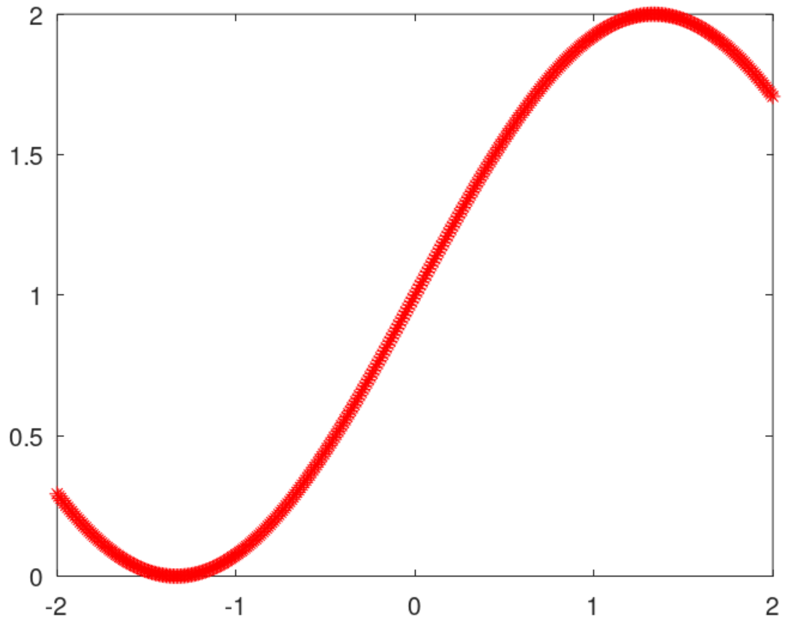


– a = 0.1

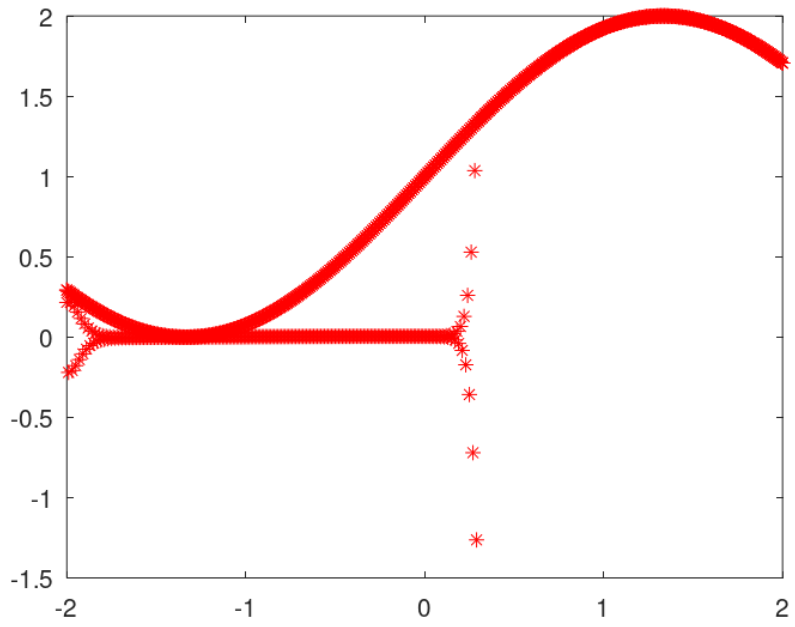

0.2
0
-0.2
-0.4
-0.6
-0.8
-2
-1
0
1
2

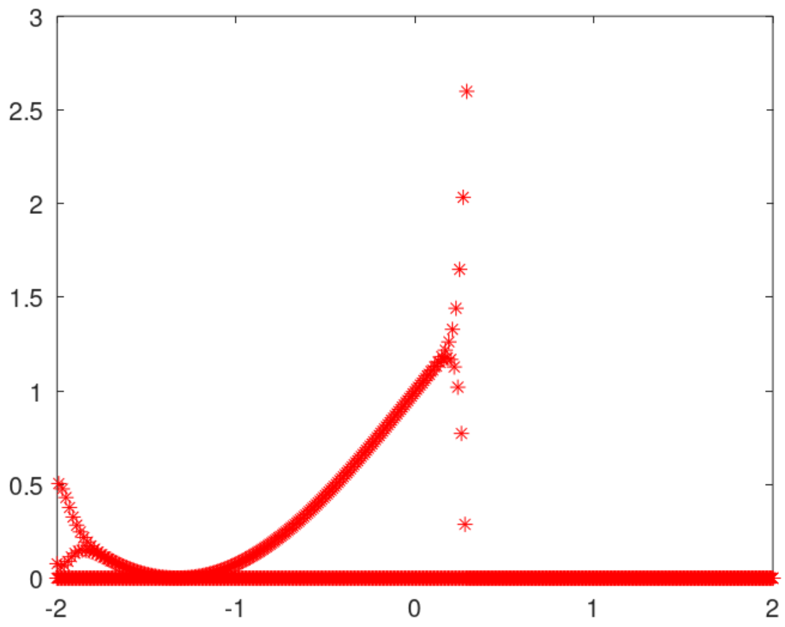


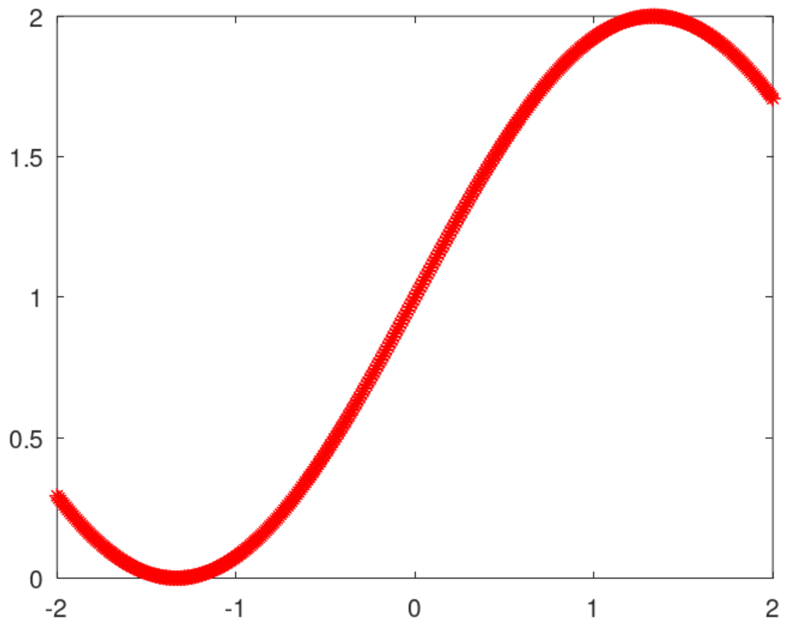
2
1.5
1
0.5
0
-2
-1
0
1
2



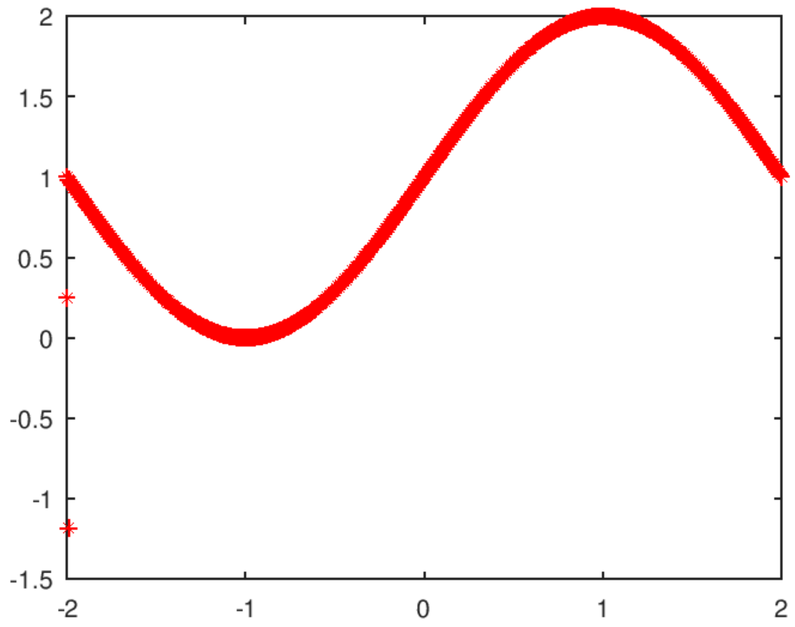
- a = 0.5

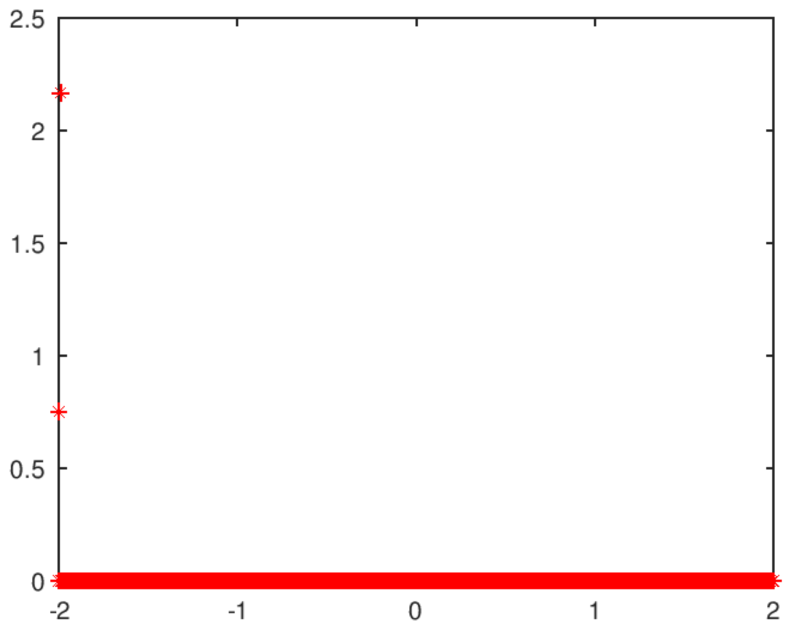

2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1
0
1
2

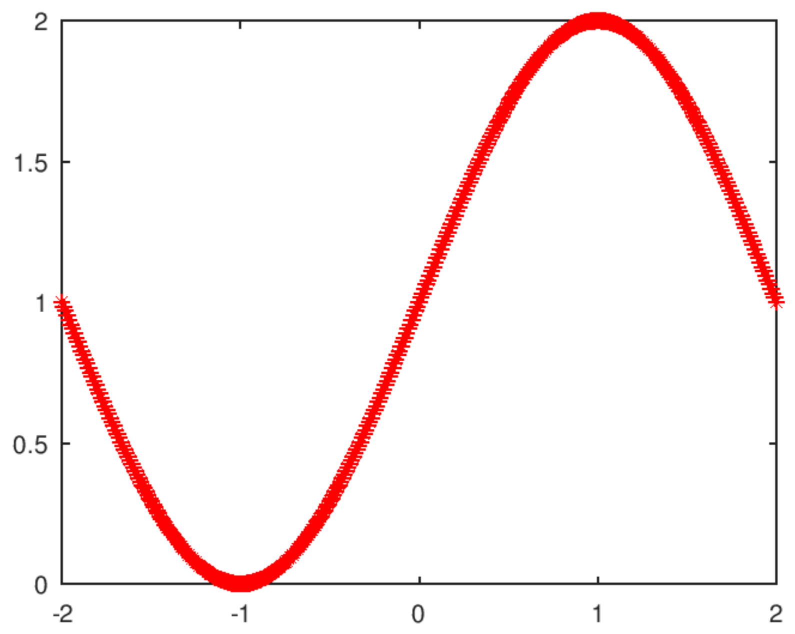

3
2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



– a = 1


2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1
0
1
2

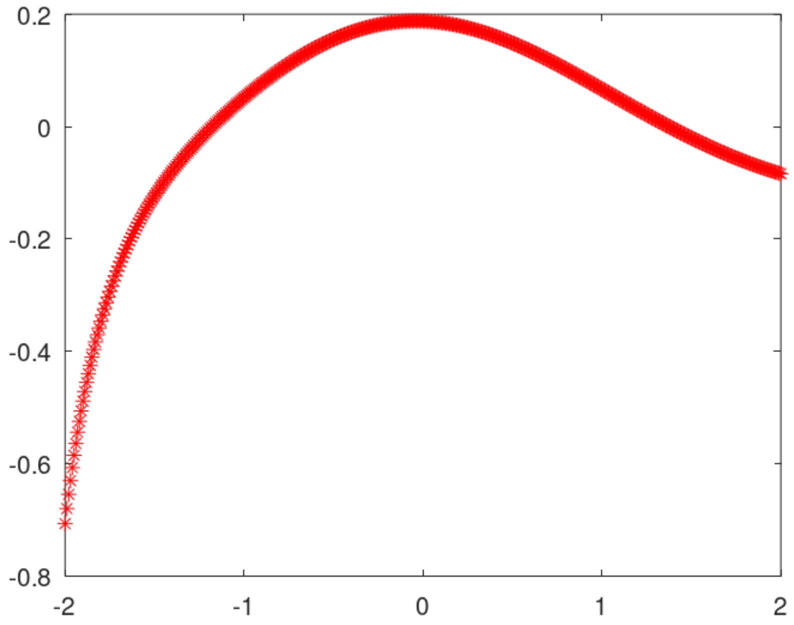

2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

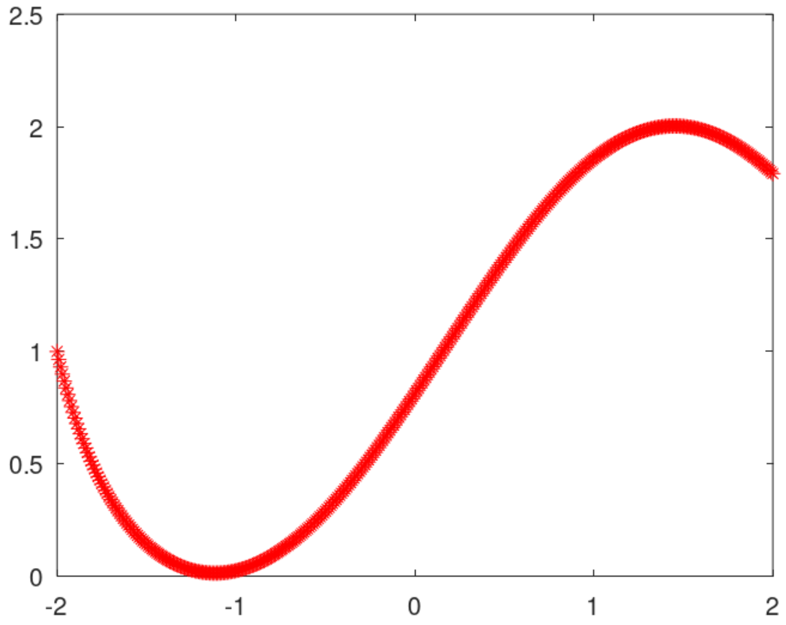


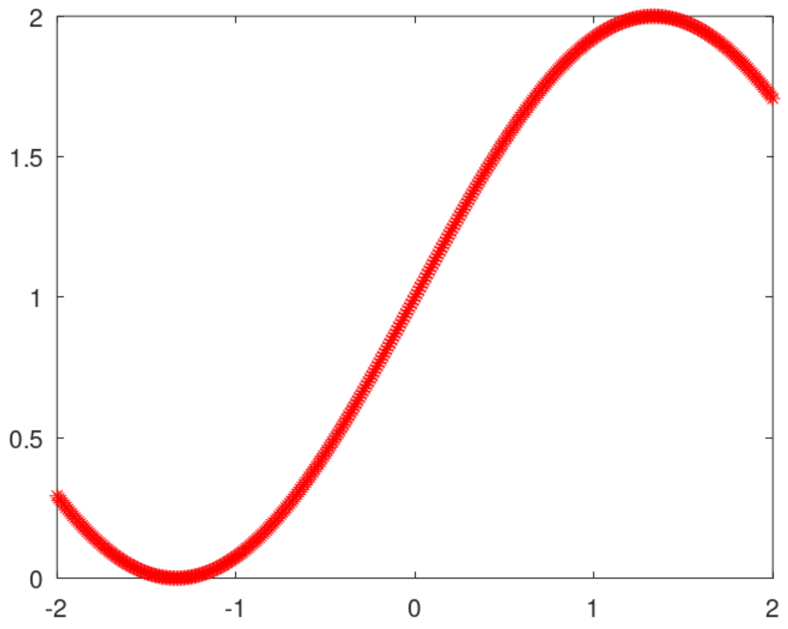
```
% Initialize network parameters
Max = 0.5;
Min = -Max;
interval = Max - Min;
lr = 0.01;
epochs = 1000;
global k;
k = 1;

% Initialize Network Function
function [W1, B1, W2, B2] = initialize_network(Min, interval)
    W1(:, 1) = Min + rand(1, 10) * interval;
    B1(:, 1) = Min + rand(1, 10) * interval;
    W2(1, :) = (Min + rand(1, 10) * interval)';
    B2(1) = Min + rand(1, 1) * interval;
end

% Train Network Function
function [W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr,
    epochs, interval)
    global k;
    while true
        for p = -2:0.01:2
            % Feed-forward
            [A1, A2] = forward_pass(W1, B1, W2, B2, p);

            % Calculate error
            error = 1 + sin(p * (3 * pi / 8)) - A2(k);

            % Backpropagation and update weights
            [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2,
    A1, A2, lr, p, error);

            % Plotting
            plot_data(p, error, A2);
            k = k + 1;
        end
        if abs(error) < 0.1 || k > epochs
            save_plots();
            return;
        end
    end
end

function y = logsig(n1)
    global k;
    % Directly update A1 inside the function might not work as
    expected without returning it
    y(:, k) = 1 ./ (1 + exp(-n1));
end

function y = relu(n2)
    y = max(0, n2);
end

% Forward Pass Function
function [A1, A2] = forward_pass(W1, B1, W2, B2, p, A1, A2)
    global k;
    n1 = (W1(:, k) .* p + B1(:, k));
    A1 = logsig(n1);
    n2 = W2(k, :) * A1(:, k) + B2(1);
    A2(k) = relu(n2);
end

% Backpropagation Function
function [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2, A1,
    A2, lr, p, error)
    global k;
    if(A2(k) > 0)
        derivative_2 = 1;
    else
        derivative_2 = 0;
    endif
    derivative = arrayfun(@(x) (1 - A1(x, k)) * A1(x, k), 1:10);
    R = diag(derivative);
    s2 = -2 * derivative_2 * error;
    s1 = R * (W2(k, :)' * s2);
    W1(:, k + 1) = W1(:, k) - lr * s1 * p;
    B1(:, k + 1) = B1(:, k) - lr * s1;
    W2(k + 1, :) = W2(k, :) - lr * s2 * A1(:, k)';
    B2(k + 1) = B2(k) - lr * s2;
end

% Plotting Function
function plot_data(p, error, A2)
    global k;
    figure(1); hold on; plot(p, error, '*r');
    figure(2); hold on; plot(p, A2(k), '*r');
    figure(3); hold on; plot(p, 1 + sin(p * (3 * pi / 8)), '*r')
    ;
end

% Save Plots Function
function save_plots()
    figure(1); print -dpng 'error_plot.png';
    figure(2); print -dpng 'approximation_plot.png';
    figure(3); print -dpng 'real_value_plot';
end

% Initialize weights and biases
[W1, B1, W2, B2] = initialize_network(Min, interval);
% Training the network
[W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr, epochs,
    interval);
```

4.3 12 neurons in 1 hidden layer

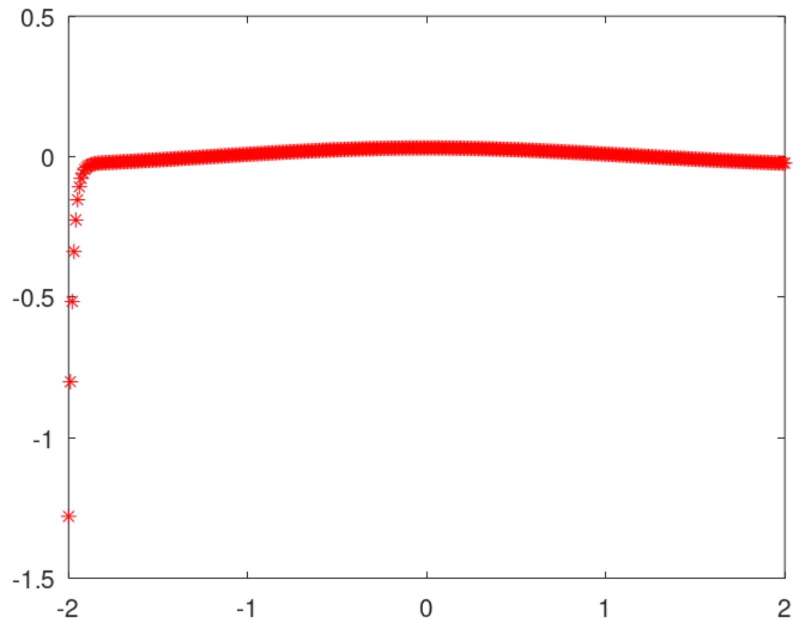
– a = 0.01

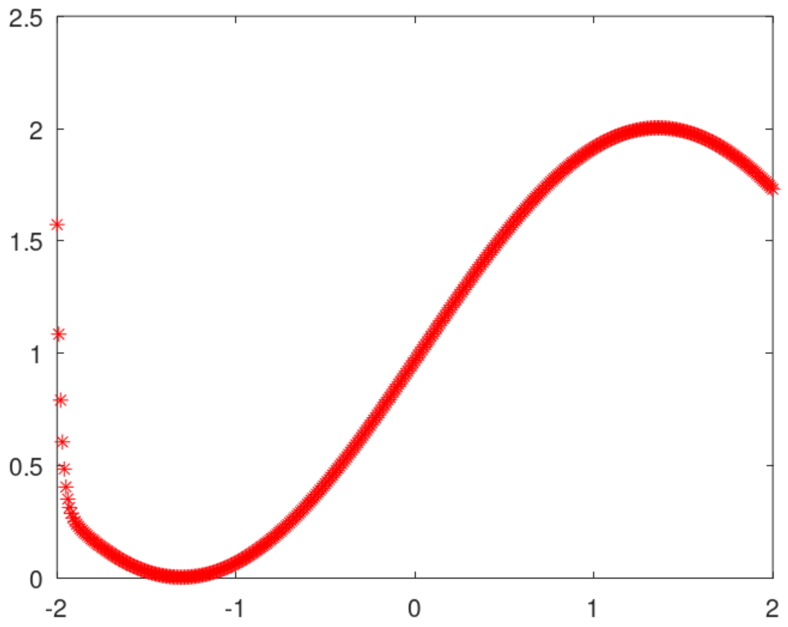

0.2
0
-0.2
-0.4
-0.6
-0.8
-2
-1
0
1
2

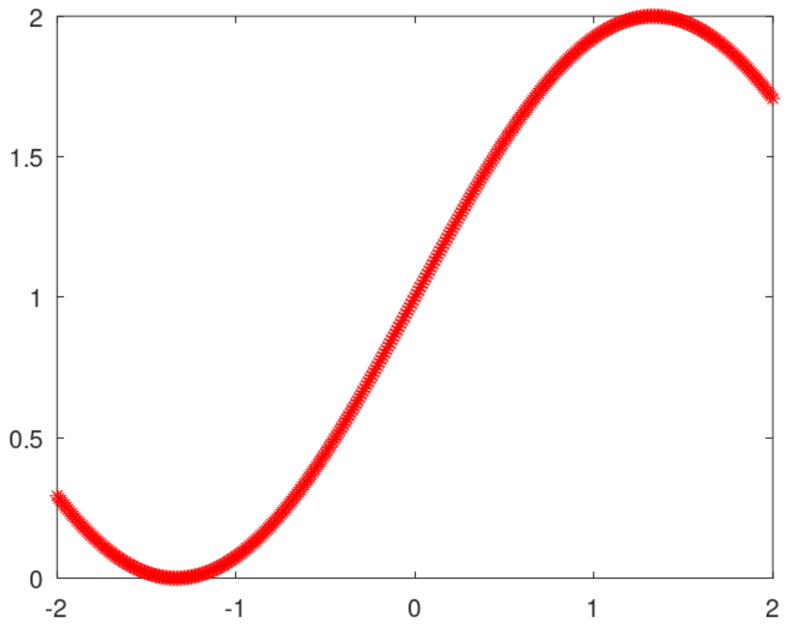

2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



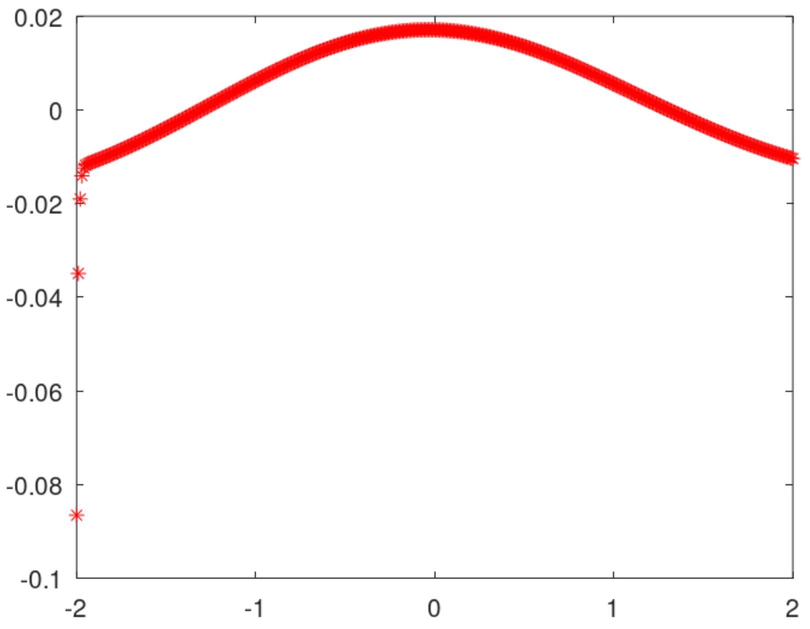
- a = 0.05

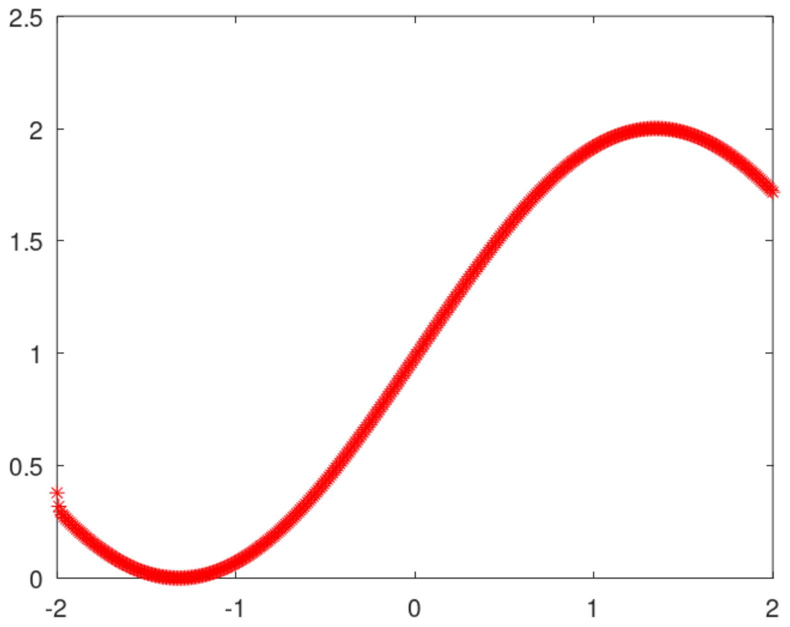

0.5
0
-0.5
-1
-1.5
-2
-1
0
1
2

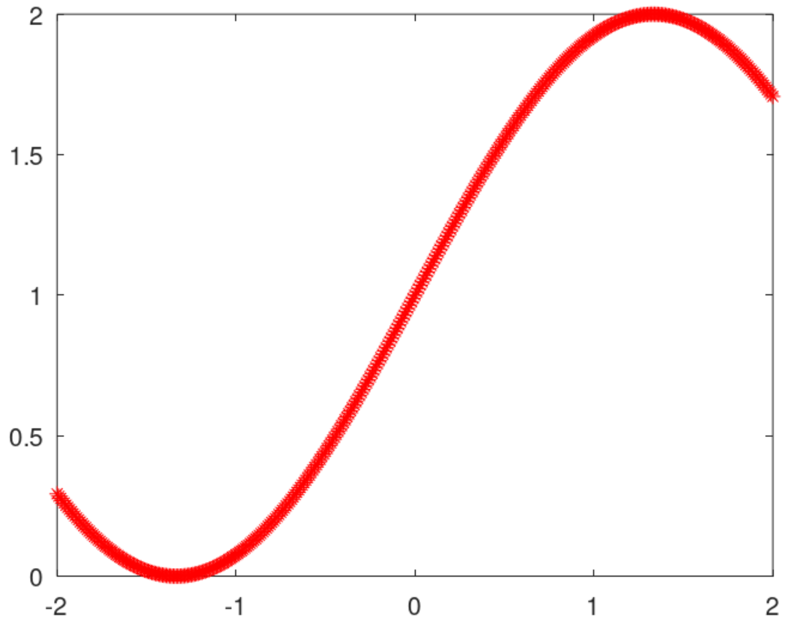

2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



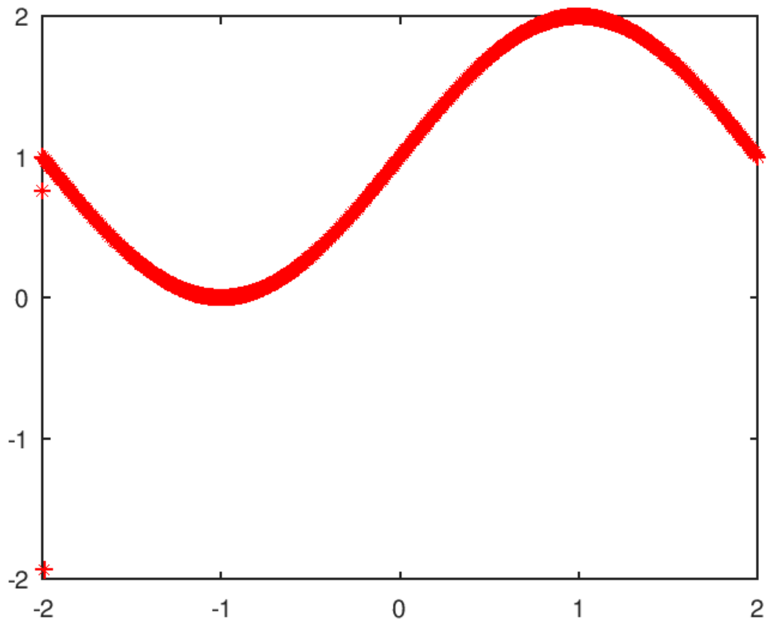
- a = 0.1

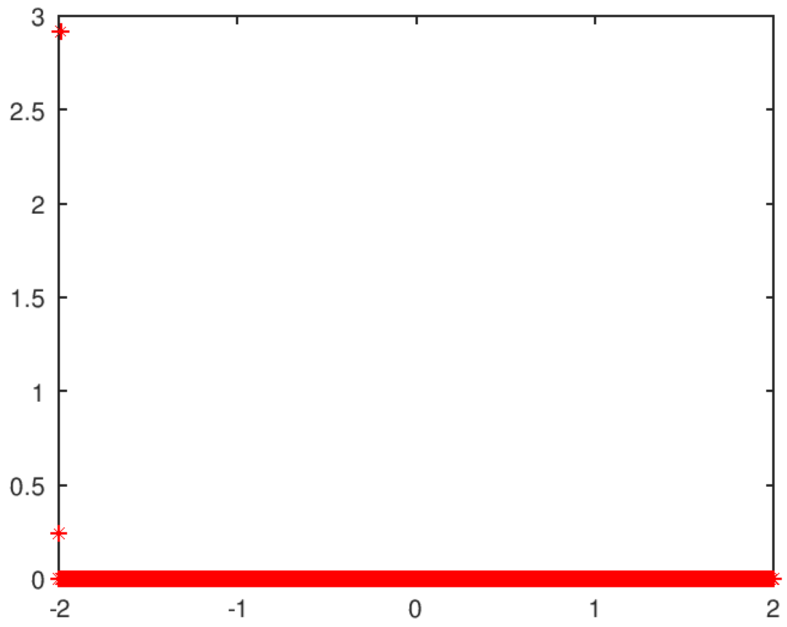

0.02
0
-0.02
-0.04
-0.06
-0.08
-0.1
-2
-1
0
1
2

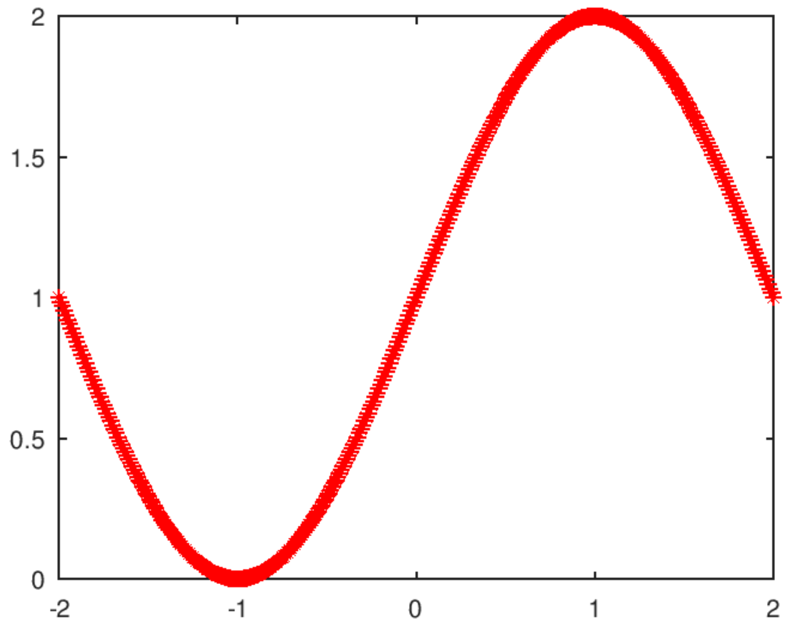

2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



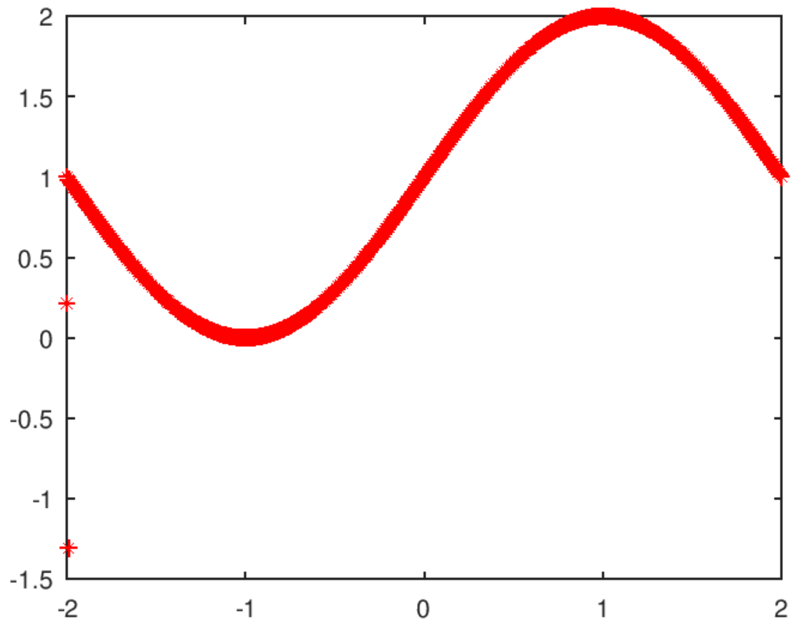
– a = 0.5

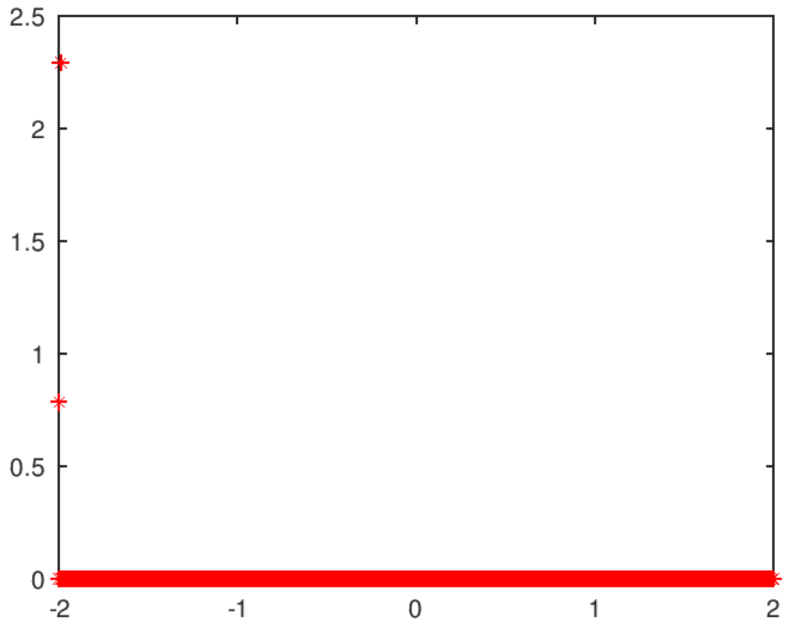

2
1
0
-1
-2
-2
-1
0
1
2

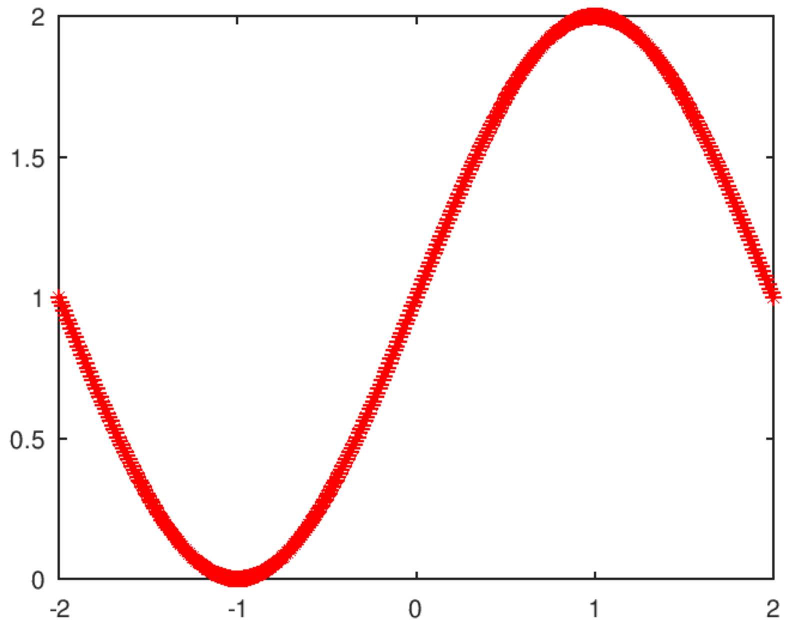

3
2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



– a = 1


2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1
0
1
2


2.5
2
1.5
1
0.5
0
-2
-1
0
1
2



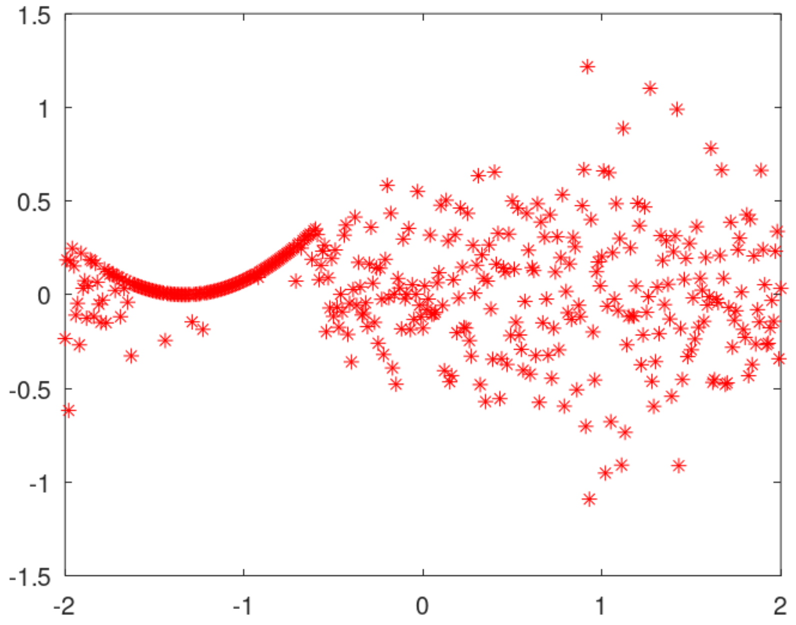
```
% Initialize network parameters
Max = 0.5;
Min = -Max;
interval = Max - Min;
lr = 0.01;
epochs = 1000;
global k;
k = 1;

% Initialize Network Function
function [W1, B1, W2, B2] = initialize_network(Min, interval)
    W1(:, 1) = Min + rand(1, 12) * interval;
    B1(:, 1) = Min + rand(1, 12) * interval;
    W2(1, :) = (Min + rand(1, 12) * interval)';
    B2(1) = Min + rand(1, 1) * interval;
end

% Train Network Function
function [W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr,
    epochs, interval)
    global k;
    while true
        for p = -2:0.01:2
            % Feed-forward
            [A1, A2] = forward_pass(W1, B1, W2, B2, p);

            % Calculate error
            error = 1 + sin(p * (3 * pi / 8)) - A2(k);

            % Backpropagation and update weights
            [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2,
    A1, A2, lr, p, error);

            % Plotting
            plot_data(p, error, A2);
            k = k + 1;
        end
        if abs(error) < 0.1 || k > epochs
            save_plots();
            return;
        end
    end
end

function y = logsig(n1)
    global k;
    y(:, k) = 1 ./ (1 + exp(-n1));
end

function y = relu(n2)
    y = max(0, n2); % This is a correct implementation for relu
end

% Forward Pass Function
function [A1, A2] = forward_pass(W1, B1, W2, B2, p, A1, A2)
    global k;
    n1 = (W1(:, k) .* p + B1(:, k));
    A1 = logsig(n1);
    n2 = W2(k, :) * A1(:, k) + B2(1);
    A2(k) = relu(n2);
end

% Backpropagation Function
function [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2, A1,
    A2, lr, p, error)
    global k;
    if(A2(k) > 0)
        derivative_2 = 1;
    else
        derivative_2 = 0;
    endif
    derivative = arrayfun(@(x) (1 - A1(x, k)) * A1(x, k), 1:12);
    R = diag(derivative);
    s2 = -2 * derivative_2 * error;
    s1 = R * (W2(k, :)' * s2);
    W1(:, k + 1) = W1(:, k) - lr * s1 * p;
    B1(:, k + 1) = B1(:, k) - lr * s1;
    W2(k + 1, :) = W2(k, :) - lr * s2 * A1(:, k)';
    B2(k + 1) = B2(k) - lr * s2;
end

% Plotting Function
function plot_data(p, error, A2)
    global k;
    figure(1); hold on; plot(p, error, '*r');
    figure(2); hold on; plot(p, A2(k), '*r');
    figure(3); hold on; plot(p, 1 + sin(p * (3 * pi / 8)), '*r')
    ;
end

% Save Plots Function
function save_plots()
    figure(1); print -dpng 'error_plot.png';
    figure(2); print -dpng 'approximation_plot.png';
    figure(3); print -dpng 'real_value_plot';
end

% Initialize weights and biases
[W1, B1, W2, B2] = initialize_network(Min, interval);
% Training the network
[W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr, epochs,
    interval);
```

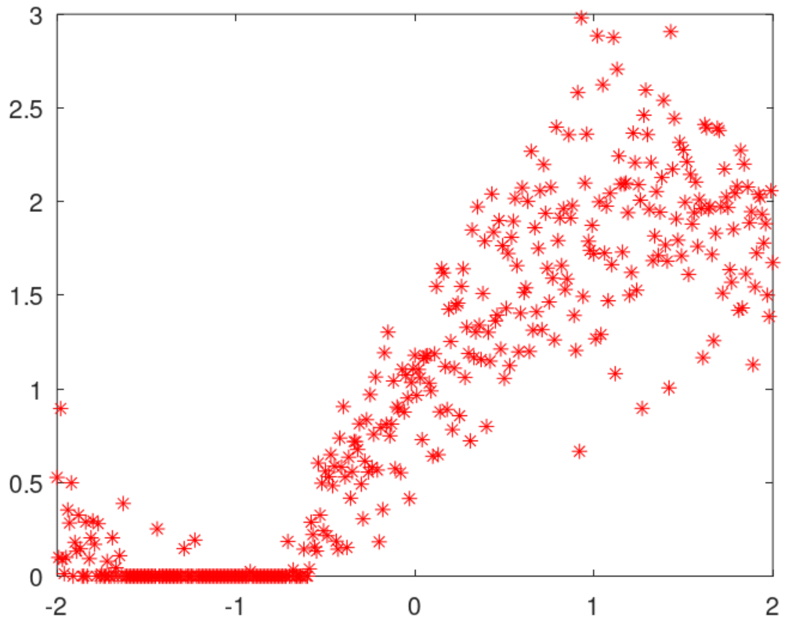
4.4 Comments:

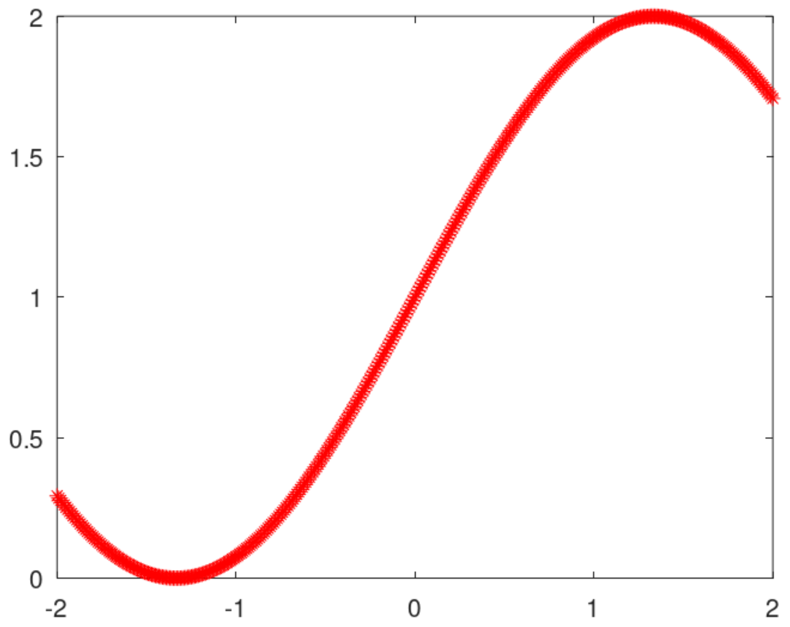
- As the learning rate and the number of neurons increase, the approximation plot tends to diverge from the plot of the real values. This indicates that a higher learning rate and increased neuron count may lead to less accurate approximations of the target function within the neural network model. There is a golden balance however between learning rate and number of neurons. Increased number of neurons needs smaller learning rate to make accurate approximations.
- For the 2 neuron network smaller learning rates (0.01, 0.05) seem to not be beneficial for the model. However by increasing the learning rate (0.1, 0.5, 1) the model converges and approximates the function. For the 10 neurons network smaller learning rates (0.01, 0.05 and 0.1) are more useful for the model's convergence than bigger learning rates (0.5, 1). The same thing occurs for the 12 neuron network with the difference that it does not converge as well as the 10 neuron network for learning rate = 0.01.
- The conclusions occurred were based on the error plot which is essentially the difference between the real value of the function and the approximated value of the neural network.

5 PROBLEM-05

5.1 $\theta = 0.15$



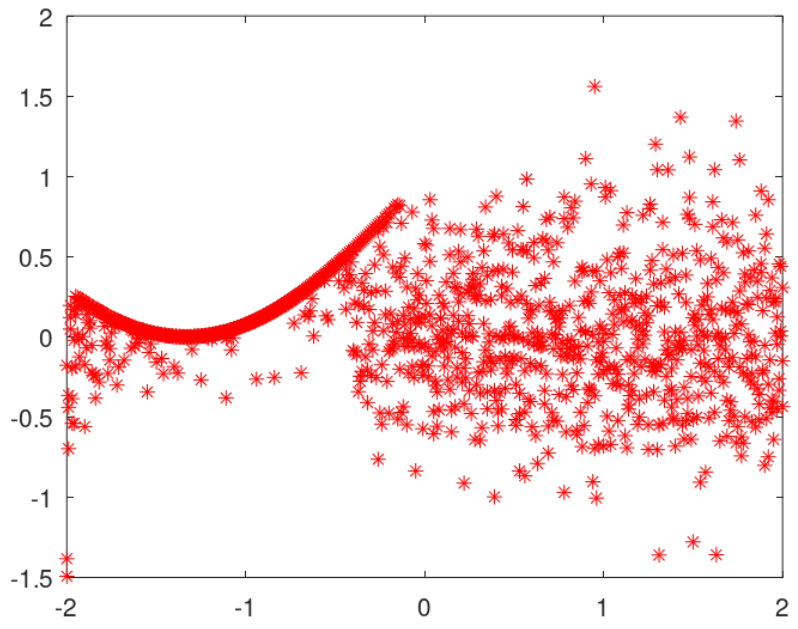

3
2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

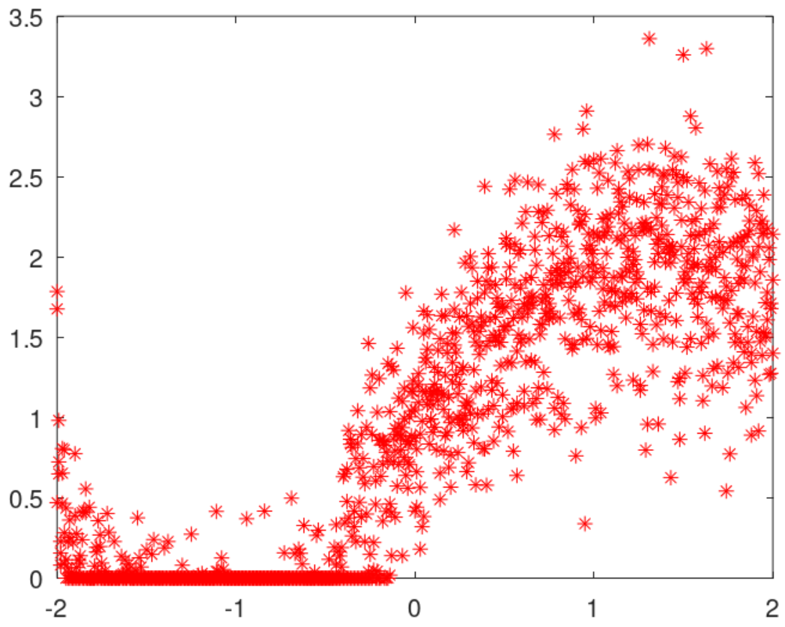


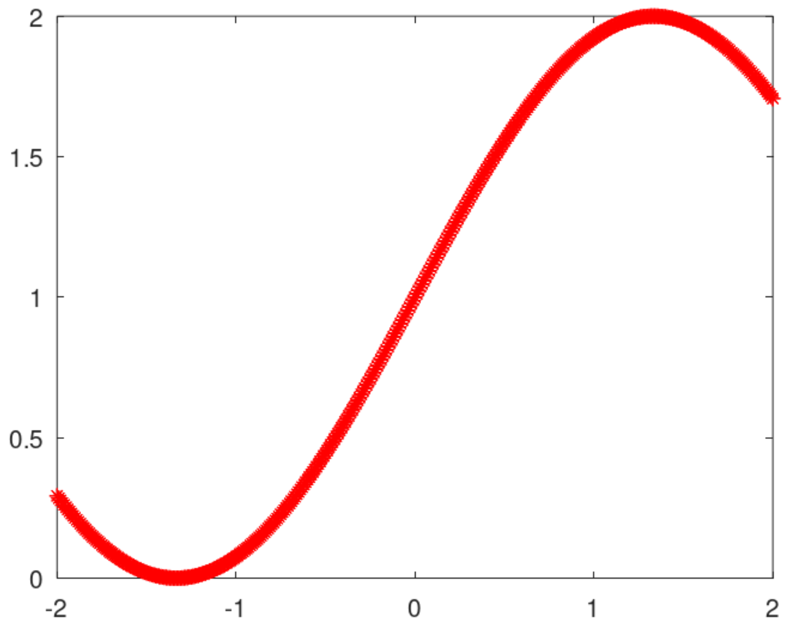
```
% Initialize network parameters
Max = 0.5;
Min = -Max;
interval = Max - Min;
lr = 0.1;
epochs = 1000;
global k;
k = 1;
global theta;
theta = 0.15; # Dropout probability

% Initialize Network Function
function [W1, B1, W2, B2] = initialize_network(Min, interval)
    W1(:, 1) = Min + rand(1, 12) * interval;
    B1(:, 1) = Min + rand(1, 12) * interval;
    W2(1, :) = (Min + rand(1, 12) * interval)';
    B2(1) = Min + rand(1, 1) * interval;
end

% Train Network Function
function [W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr,
    epochs, interval)
    global k;
    while true
```

```
        for p = -2:0.01:2
            % Feed-forward

            [A1, A2] = forward_pass(W1, B1, W2, B2, p);

            % Calculate error
            error = 1 + sin(p * (3 * pi / 8)) - A2(k);

            % Backpropagation and update weights
            [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2,
    A1, A2, lr, p, error);

            % Plotting
            plot_data(p, error, A2);
            k = k + 1;
        end
        if abs(error) < 0.1 || k > epochs
            save_plots();
            return;
        end
    end
end

function y = logsig(n1)
    global k;
    y(:, k) = 1 ./ (1 + exp(-n1));
end

function y = relu(n2)
    y = max(0, n2);
end

% Forward Pass Function
function [A1, A2] = forward_pass(W1, B1, W2, B2, p, A1, A2)
    global k;
    global theta;
    dropout_mask = (rand(12, 1) > theta);
    n1 = (W1(:, k) .* p + B1(:, k));
    A1 = logsig(n1);
    A1_dropout = A1(:, k) .* dropout_mask;
    n2 = W2(k, :) * A1_dropout + B2(1);
    A2(k) = relu(n2);
end

% Backpropagation Function
function [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2, A1,
    A2, lr, p, error)
    global k;
    if(A2(k) > 0)
        derivative_2 = 1;
    else
        derivative_2 = 0;
    endif
    derivative = arrayfun(@(x) (1 - A1(x, k)) * A1(x, k), 1:12);
    R = diag(derivative);
    s2 = -2 * derivative_2 * error;
    s1 = R * (W2(k, :)' * s2);
    W1(:, k + 1) = W1(:, k) - lr * s1 * p;
    B1(:, k + 1) = B1(:, k) - lr * s1;
    W2(k + 1, :) = W2(k, :) - lr * s2 * A1(:, k)';
    B2(k + 1) = B2(k) - lr * s2;
end

% Plotting Function
function plot_data(p, error, A2)
    global k;
    figure(1); hold on; plot(p, error, '*r');
    figure(2); hold on; plot(p, A2(k), '*r');
    figure(3); hold on; plot(p, 1 + sin(p * (3 * pi / 8)), '*r')
    ;
end

% Save Plots Function
function save_plots()
    figure(1); print -dpng 'error_plot.png';
    figure(2); print -dpng 'approximation_plot.png';
    figure(3); print -dpng 'real_value_plot';
end

% Initialize weights and biases
[W1, B1, W2, B2] = initialize_network(Min, interval);
% Training the network
[W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr, epochs,
    interval);
```

5.2 $\theta = 0.25$



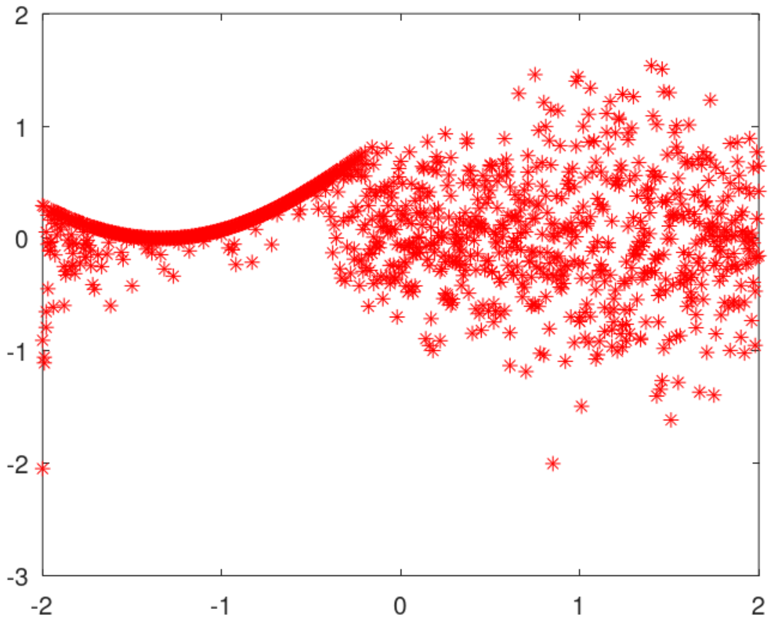

3.5
3
2.5
2
1.5
1
0.5
0
-2
-1
0
1
2

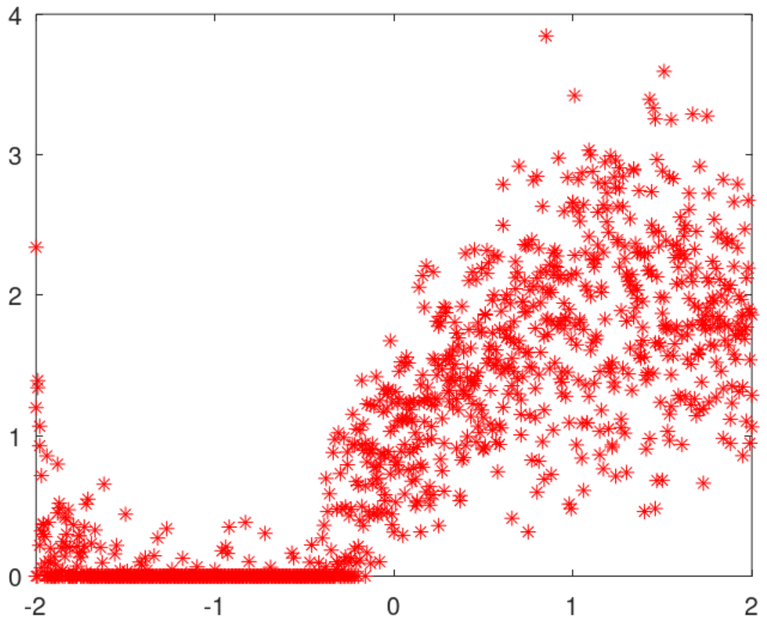


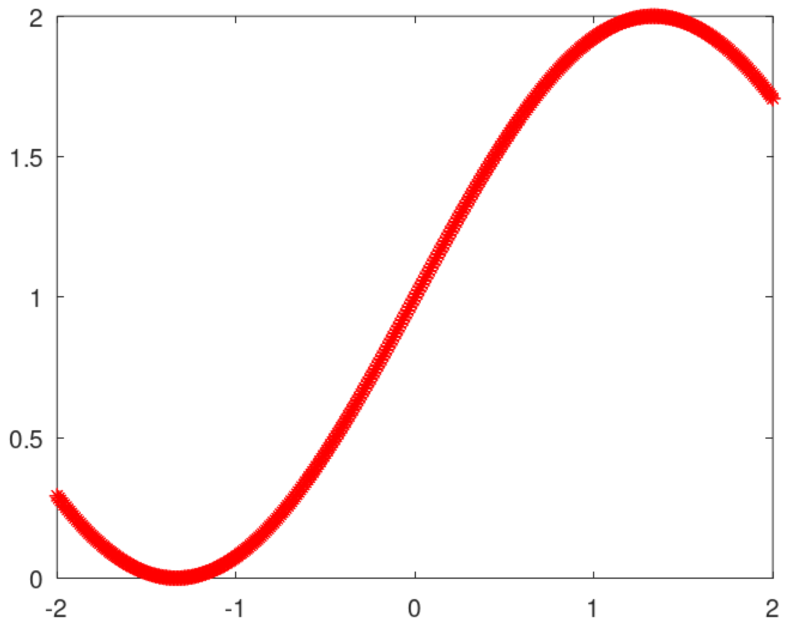
```
% Initialize network parameters
Max = 0.5;
Min = -Max;
interval = Max - Min;
lr = 0.1;
epochs = 1000;
global k;
k = 1;
global theta;
theta = 0.25; # Dropout probability

% Initialize Network Function
function [W1, B1, W2, B2] = initialize_network(Min, interval)
    W1(:, 1) = Min + rand(1, 12) * interval;
    B1(:, 1) = Min + rand(1, 12) * interval;
    W2(1, :) = (Min + rand(1, 12) * interval)';
    B2(1) = Min + rand(1, 1) * interval;
end

% Train Network Function
function [W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr,
    epochs, interval)
    global k;
    while true
```

```
        for p = -2:0.01:2
            % Feed-forward

            [A1, A2] = forward_pass(W1, B1, W2, B2, p);

            % Calculate error
            error = 1 + sin(p * (3 * pi / 8)) - A2(k);

            % Backpropagation and update weights
            [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2,
    A1, A2, lr, p, error);

            % Plotting
            plot_data(p, error, A2);
            k = k + 1;
        end
        if abs(error) < 0.1 || k > epochs
            save_plots();
            return;
        end
    end
end

function y = logsig(n1)
    global k;
    y(:, k) = 1 ./ (1 + exp(-n1));
end

function y = relu(n2)
    y = max(0, n2);
end

% Forward Pass Function
function [A1, A2] = forward_pass(W1, B1, W2, B2, p, A1, A2)
    global k;
    global theta;
    dropout_mask = (rand(12, 1) > theta);
    n1 = (W1(:, k) .* p + B1(:, k));
    A1 = logsig(n1);
    A1_dropout = A1(:, k) .* dropout_mask;
    n2 = W2(k, :) * A1_dropout + B2(1);
    A2(k) = relu(n2); % Using relu function directly
end

% Backpropagation Function
function [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2, A1,
    A2, lr, p, error)
    global k;
    if(A2(k) > 0)
        derivative_2 = 1;
    else
        derivative_2 = 0;
    endif
    derivative = arrayfun(@(x) (1 - A1(x, k)) * A1(x, k), 1:12);
    R = diag(derivative);
    s2 = -2 * derivative_2 * error;
    s1 = R * (W2(k, :)' * s2);
    W1(:, k + 1) = W1(:, k) - lr * s1 * p;
    B1(:, k + 1) = B1(:, k) - lr * s1;
    W2(k + 1, :) = W2(k, :) - lr * s2 * A1(:, k)';
    B2(k + 1) = B2(k) - lr * s2;
end

% Plotting Function
function plot_data(p, error, A2)
    global k;
    figure(1); hold on; plot(p, error, '*r');
    figure(2); hold on; plot(p, A2(k), '*r');
    figure(3); hold on; plot(p, 1 + sin(p * (3 * pi / 8)), '*r')
    ;
end

% Save Plots Function
function save_plots()
    figure(1); print -dpng 'error_plot.png';
    figure(2); print -dpng 'approximation_plot.png';
    figure(3); print -dpng 'real_value_plot';
end

% Initialize weights and biases
[W1, B1, W2, B2] = initialize_network(Min, interval);
% Training the network
[W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr, epochs,
    interval);
```

5.3 $\theta = 0.35$




4
3
2
1
0
-2
-1
0
1
2



```
% Initialize network parameters
Max = 0.5;
Min = -Max;
interval = Max - Min;
lr = 0.1;
epochs = 1000;
global k;
k = 1;
global theta;
theta = 0.35; # Dropout probability

% Initialize Network Function
function [W1, B1, W2, B2] = initialize_network(Min, interval)
    W1(:, 1) = Min + rand(1, 12) * interval;
    B1(:, 1) = Min + rand(1, 12) * interval;
    W2(1, :) = (Min + rand(1, 12) * interval)';
    B2(1) = Min + rand(1, 1) * interval;
end

% Train Network Function
function [W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr,
    epochs, interval)
    global k;
    while true
```

```
        for p = -2:0.01:2
            % Feed-forward

            [A1, A2] = forward_pass(W1, B1, W2, B2, p);

            % Calculate error
            error = 1 + sin(p * (3 * pi / 8)) - A2(k);

            % Backpropagation and update weights
            [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2,
    A1, A2, lr, p, error);

            % Plotting
            plot_data(p, error, A2);
            k = k + 1;
        end
        if abs(error) < 0.1 || k > epochs
            save_plots();
            return;
        end
    end
end

function y = logsig(n1)
    global k;
    y(:, k) = 1 ./ (1 + exp(-n1));
end

function y = relu(n2)
    y = max(0, n2);
end

% Forward Pass Function
function [A1, A2] = forward_pass(W1, B1, W2, B2, p, A1, A2)
    global k;
    global theta;
    dropout_mask = (rand(12, 1) > theta);
    n1 = (W1(:, k) .* p + B1(:, k));
    A1 = logsig(n1);
    A1_dropout = A1(:, k) .* dropout_mask;
    n2 = W2(k, :) * A1_dropout + B2(1);
    A2(k) = relu(n2); % Using relu function directly
end

% Backpropagation Function
function [W1, B1, W2, B2] = backpropagation(W1, B1, W2, B2, A1,
    A2, lr, p, error)
    global k;
    if(A2(k) > 0)
        derivative_2 = 1;
    else
        derivative_2 = 0;
    endif
    derivative = arrayfun(@(x) (1 - A1(x, k)) * A1(x, k), 1:12);
    R = diag(derivative);
    s2 = -2 * derivative_2 * error;
    s1 = R * (W2(k, :)' * s2);
    W1(:, k + 1) = W1(:, k) - lr * s1 * p;
    B1(:, k + 1) = B1(:, k) - lr * s1;
    W2(k + 1, :) = W2(k, :) - lr * s2 * A1(:, k)';
    B2(k + 1) = B2(k) - lr * s2;
end

% Plotting Function
function plot_data(p, error, A2)
    global k;
    figure(1); hold on; plot(p, error, '*r');
    figure(2); hold on; plot(p, A2(k), '*r');
    figure(3); hold on; plot(p, 1 + sin(p * (3 * pi / 8)), '*r')
    ;
end

% Save Plots Function
function save_plots()
    figure(1); print -dpng 'error_plot.png';
    figure(2); print -dpng 'approximation_plot.png';
    figure(3); print -dpng 'real_value_plot';
end

% Initialize weights and biases
[W1, B1, W2, B2] = initialize_network(Min, interval);
% Training the network
[W1, B1, W2, B2] = train_network(W1, B1, W2, B2, lr, epochs,
    interval);
```

5.4 Comments:

– For all dropout rates, the network seems to not converge as it iterates through the input range. The introduction of dropout, seem to reduce our network's accuracy because it is reducing the capacity during training.
– Compared to the previous exercise without it, the introduction of dropout result in a noisier approximation. This is also demonstrated by the error plot which is the difference between the real value of the function and the approximated value.
– If we reduced the learning rate by significant margin (< 0.001) the neural networks would present signs of convergence. That is because weight updates become smaller and lead to more stable convergence.

Operation of Dropout as a Generalization Technique:

- Dropout appears to act as a regularizer that prevents overfitting by randomly omitting a subset of neurons during training. This forces the network to learn more robust features that are not reliant on any one neuron.
- The network's performance with dropout suggests that incorporating dropout does not enhance its approximation capabilities. However, the original network did not exhibit signs of overfitting to begin with, and introducing dropout may not yield benefits and could unnecessarily complicate the learning process without improving the network's accuracy.

6 PROBLEM-06

6.1 Learning Rule (Steepest Descent Algorithm)

We define the performance index using the squared error as $F(x) = (t - f(n))^2$, where t is the target and $f(n)$ is the output.

We then calculate the weight updates using the steepest descent algorithm with the update rule $\Delta w_i = -\alpha \frac{\partial F(x)}{\partial w_i}$.(1)

And we compute the derivatives of $F(x)$ with respect to the weights w_1, w_2, and w_{12} as $\frac{\partial F(x)}{\partial w_i} = 2(t - f(n)) \left(-\frac{\partial f(n)}{\partial w_i}\right)$.

Therefore, the key terms we need to compute are

$$\frac{\partial f(n)}{\partial w_i}.$$

To compute these terms we first need to write out the network equation:

$$f(n) = \text{tanh}(w_1 p_1 + w_2 p_2 + w_{12} p_1 p_2 + b)$$

Next we take the derivative of both sides of this equation with respect to the network weights:

$$\frac{\partial f(n)}{\partial w_1} = p_1(1 - \text{tanh}^2(n)),$$

$$\frac{\partial f(n)}{\partial w_2} = p_2(1 - \text{tanh}^2(n)),$$

$$\frac{\partial f(n)}{\partial w_{12}} = p_1 p_2(1 - \text{tanh}^2(n)),$$

$$\frac{\partial f(n)}{\partial b} = (1 - \text{tanh}^2(n)),$$

where $n = w_1 p_1 + w_2 p_2 + w_{12} p_1 p_2 + b$.

so when we calculate all them, we use function (1) and calculate Δw_i

6.2 One Iteration with Initial Values

The network output is computed as:

$$f(n) = \text{tanh}(w_1 p_1 + w_2 p_2 + w_{12} p_1 p_2 + b) = \text{tanh}(-1 \cdot 1 + 1) = \text{tanh}(0) = 0$$

The derivatives of the output $f(n)$ with respect to the weights and bias are:

$$\frac{\partial f(n)}{\partial w_1} = p_1(1 - \text{tanh}^2(n)) = 0$$
$$\frac{\partial f(n)}{\partial w_2} = p_2(1 - \text{tanh}^2(n)) = 1$$
$$\frac{\partial f(n)}{\partial w_{12}} = p_1 p_2(1 - \text{tanh}^2(n)) = 0$$
$$\frac{\partial f(n)}{\partial b} = (1 - \text{tanh}^2(n)) = 1$$

The performance index is:

$$F(x) = (t - f(n))^2 = 0.5625$$

Then we calculate:

$$2(t - f(n)) = 1.5$$

The weight updates are:

$$\Delta w_1 = -2\alpha(t - f(n))(-\frac{\partial f(n)}{\partial w_1}) = 0$$
$$\Delta w_2 = -2\alpha(t - f(n))(-\frac{\partial f(n)}{\partial w_2}) = 1.5$$
$$\Delta w_{12} = -2\alpha(t - f(n))(-\frac{\partial f(n)}{\partial w_{12}}) = 0$$
$$\Delta b = -2\alpha(t - f(n))(-\frac{\partial f(n)}{\partial b}) = 1.5$$

Applying the weight updates, the new values for the weights and bias are:

$$w_1^{\text{new}} = w_1 + \Delta w_1 = 1 - 0 = -1$$
$$w_2^{\text{new}} = w_2 + \Delta w_2 = -1 - (1.5) = -2.5$$
$$w_{12}^{\text{new}} = w_{12} + \Delta w_{12} = 0.5 - 0 = 0.5$$
$$b^{\text{new}} = b + \Delta b = 1 - (1.5) = -0.5$$

Therefore, after one iteration of the learning rule, the updated parameters are $w_2^{\text{new}} = -2.5$ and $b^{\text{new}} = -0.5$.

```
import numpy as np

def tanh(x):
  return np.tanh(x)

def dtanh(x):
  return 1 - np.tanh(x)**2

# Initial parameters from the problem
w1 = 1
w2 = -1
w12 = 0.5
b = 1
p1 = 0
p2 = 1
t = 0.75  # Target output
alpha = 1  # Learning rate

# Compute the network output 'f(n)'
f_n = tanh(w1*p1 + w2*p2 + w12*p1*p2 + b)

# Derivatives of the output f(n) with respect to the weights and
     bias
df_dw1 = p1 * dtanh(f_n)
df_dw2 = p2 * dtanh(f_n)
df_dw12 = p1 * p2 * dtanh(f_n)
df_db = dtanh(f_n)

# Performance index calculation
F_x = (t - f_n)**2

# Derivative of the performance index with respect to the output
     f(n)
dF_df = 2 * (t - f_n)

# Weight updates calculation
Delta_w1 = -alpha * dF_df * df_dw1
Delta_w2 = -alpha * dF_df * df_dw2
Delta_w12 = -alpha * dF_df * df_dw12
Delta_b = -alpha * dF_df * df_db

# New values for the weights and bias
w1_new = w1 + Delta_w1
w2_new = w2 + Delta_w2
w12_new = w12 + Delta_w12
b_new = b + Delta_b


print(w1_new, w2_new, w12_new, b_new)
```

7 PROBLEM-07

First, we define the ReLU and pReLU functions:

$$\begin{aligned}\text{ReLU}(x) &= \max(0, x),\\ \text{pReLU}(x) &= \max(0, x) + \alpha \min(0, x),\end{aligned}$$

where α is a small constant.

Now, let's consider a simple MLP with two layers, where H^1 and H^2 are the outputs of the first and second layers, respectively, and $\mathbf{W}^1$, $\mathbf{W}^2$, $\mathbf{b}^1$, $\mathbf{b}^2$ represent the weights and biases of the two layers:

$$\begin{aligned}H^1 &= \max(0, x) \cdot (xW^1 + b^1)\\ H^2 &= \max(0, H^1) \cdot (H^1W^2 + b^2)\end{aligned}$$

Now, considering the piecewise nature of ReLU, we can express H^2 in terms of x:

$$H^2 = \begin{cases}(H^1W^2 + b^2) & \text{if } x \geq 0\\ b^2 & \text{if } x < 0\end{cases}$$

The function H^1 is piecewise linear because the ReLU activation function is linear for all inputs greater than zero and constant (zero) for all inputs less than or equal to zero. Similarly, H^2 is also a piecewise linear function of H^1. Since the composition of piecewise linear functions is still piecewise linear, the output of the MLP, H^2, is a continuous piecewise linear function of the input x.

To elaborate, for each neuron in the first hidden layer, we have a piecewise linear transformation. This transformation is then propagated to the second layer, where another set of piecewise linear transformations are applied. As a result, the final output of the MLP is a composition of piecewise linear functions, which remains piecewise linear.

Thus, we conclude that an MLP with ReLU or pReLU activations indeed constructs a continuous piecewise linear function.

Now, let's explain why this is also true for pReLU.

$$\begin{aligned}H^1 &= \max(0, x) \cdot (x\mathbf{W}^1 + \mathbf{b}^1) + \alpha \min(0, x) \cdot (x\mathbf{W}^1 + \mathbf{b}^1).\\ H^2 &= \max(0, H^1) \cdot (H^1\mathbf{W}^2 + \mathbf{b}^2) + \alpha \min(0, H^1) \cdot (H^1\mathbf{W}^2 + \mathbf{b}^2).\end{aligned}$$

Now, considering the piecewise nature of ReLU, we can express H^2 in terms of x:

$$H^2 = \begin{cases}(H^1W^2 + b^2) & \text{if } x \geq 0\\ \alpha(H^1W^2 + b^2) & \text{if } x < 0\end{cases}$$

Even with the pReLU activation, both H^1 and H^2 remain piecewise linear because both the max and min parts of pReLU are piecewise linear functions.

Since the composition of piecewise linear functions is still piecewise linear, the output of the MLP, H^2, is a continuous piecewise linear function of the input x.

Thus, we conclude that an MLP with ReLU or pReLU activations indeed constructs a continuous piecewise linear function.

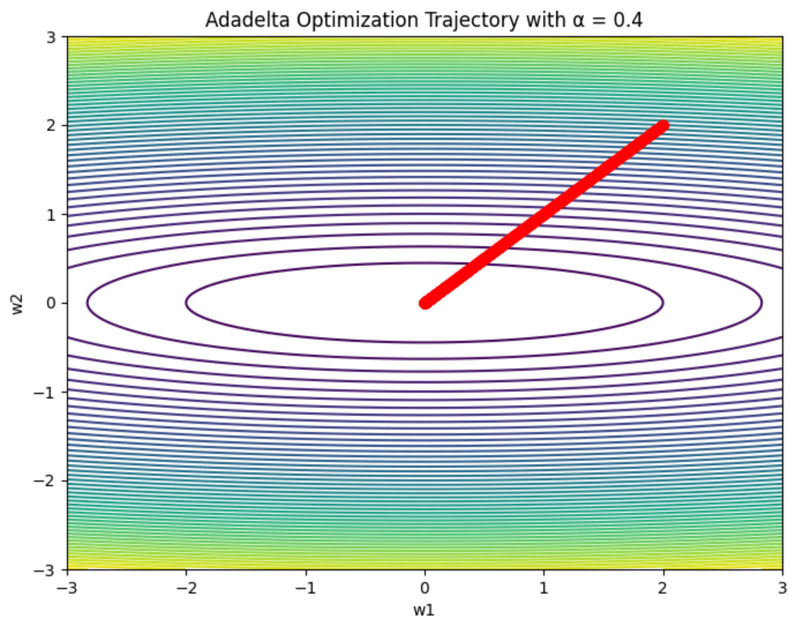
8 PROBLEM-08

Traditionally, AdaDelta does not have a learning rate parameter. Instead, it dynamically adjusts the learning rate based on the history of gradients for each parameter, aiming to improve convergence by considering the window of accumulated gradients to scale the learning steps. This approach allows AdaDelta to adapt over time without the need for manual tuning of a learning rate.

However, incorporating a fixed learning rate into AdaDelta can be seen as a modification to its original formulation, potentially to blend the adaptability of AdaDelta with the simplicity and intuitive control offered by a global learning rate.

8.1 Part 1

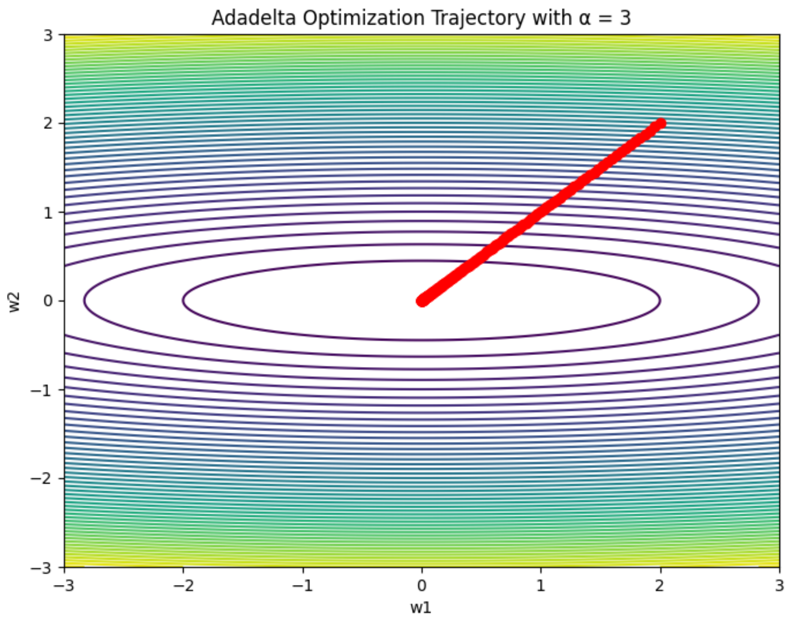
```
import numpy as np
import matplotlib.pyplot as plt

# Function F(w)
def F(w):
  return 0.1 * w[0]**2 + 2 * w[1]**2

# Gradient of F(w)
def grad_F(w):
  return np.array([0.2 * w[0], 4 * w[1]])

# Adadelta optimizer function
def adadelta_optimizer(grad_f, w_init, lr, rho=0.95, epsilon=1e
    -5, max_iter=1000, tol=1e-7):
  w = w_init
  trajectory = [w]
  E_g2 = np.zeros_like(w)
  E_delta_w2 = np.zeros_like(w)

  for i in range(max_iter):
    g = grad_f(w)
    E_g2 = rho * E_g2 + (1 - rho) * g**2
    delta_w = - (np.sqrt(E_delta_w2 + epsilon) / np.sqrt(E_g2 +
    epsilon)) * g
    E_delta_w2 = rho * E_delta_w2 + (1 - rho) * delta_w**2
    w_new = w + lr * delta_w

    # Check for convergence
    if np.linalg.norm(w_new - w) < tol:
      print(f"Converged in {i} iterations")
      w = w_new
      break

    w = w_new
    trajectory.append(w)
  return w, np.array(trajectory)


# Initial parameters
w_init = np.array([2.0, 2.0])  # initial weights
learning_rate = 0.4  # learning rate

# Optimization
w_min, trajectory = adadelta_optimizer(grad_F, w_init,
    learning_rate)

# Plotting
x = np.linspace(-3, 3, 400)
y = np.linspace(-3, 3, 400)
X, Y = np.meshgrid(x, y)
Z = F([X, Y])

plt.figure(figsize=(8, 6))
plt.contour(X, Y, Z, levels=50)
plt.plot(trajectory[:, 0], trajectory[:, 1], color='r', marker='
    o')
plt.title("Adadelta Optimization Trajectory with α = 0.4")
plt.xlabel("w1")
plt.ylabel("w2")
plt.show()

min_value = F(w_min)

# Rresults
print("Minimum weights (w_min):", w_min)
print("Minimum value of the function (F(w_min)):", min_value)
```



We concluded that it Converged in 720 iterations. The minimum weights (w_min) are [1.08124984e-06 3.29263156e-27] (basically [0, 0]) and the minimum value of the function is 1.1691012211955963e-13 (basically 0).

8.2 Part 2

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

# Function F(w)
def F(w):
  return 0.1 * w[0]**2 + 2 * w[1]**2

# Gradient of F(w)
def grad_F(w):
  return np.array([0.2 * w[0], 4 * w[1]])

# Adadelta optimizer function
def adadelta_optimizer(grad_f, w_init, lr, rho=0.95, epsilon=1e
    -5, max_iter=1000, tol=1e-7):
  w = w_init
  trajectory = [w]
  E_g2 = np.zeros_like(w)
  E_delta_w2 = np.zeros_like(w)

  for i in range(max_iter):
    g = grad_f(w)
    E_g2 = rho * E_g2 + (1 - rho) * g**2
    delta_w = - (np.sqrt(E_delta_w2 + epsilon) / np.sqrt(E_g2 +
    epsilon)) * g
    E_delta_w2 = rho * E_delta_w2 + (1 - rho) * delta_w**2
    w_new = w + lr * delta_w

    # Check for convergence
    if np.linalg.norm(w_new - w) < tol:
      print(f"Converged in {i} iterations")
      w = w_new
      break

    w = w_new
    trajectory.append(w)

  return w, np.array(trajectory)


# Initial parameters
w_init = np.array([2.0, 2.0])  # initial weights
learning_rate = 3  # learning rate

# Optimization
w_min, trajectory = adadelta_optimizer(grad_F, w_init,
    learning_rate)

# Plotting
x = np.linspace(-3, 3, 400)
y = np.linspace(-3, 3, 400)
X, Y = np.meshgrid(x, y)
Z = F([X, Y])

plt.figure(figsize=(8, 6))
plt.contour(X, Y, Z, levels=50)
plt.plot(trajectory[:, 0], trajectory[:, 1], color='r', marker='
    o')
plt.title("Adadelta Optimization Trajectory with α = 3")
plt.xlabel("w1")
plt.ylabel("w2")
plt.show()

min_value = F(w_min)

# Results
print("Minimum weights (w_min):", w_min)
print("Minimum value of the function (F(w_min)):", min_value)
```



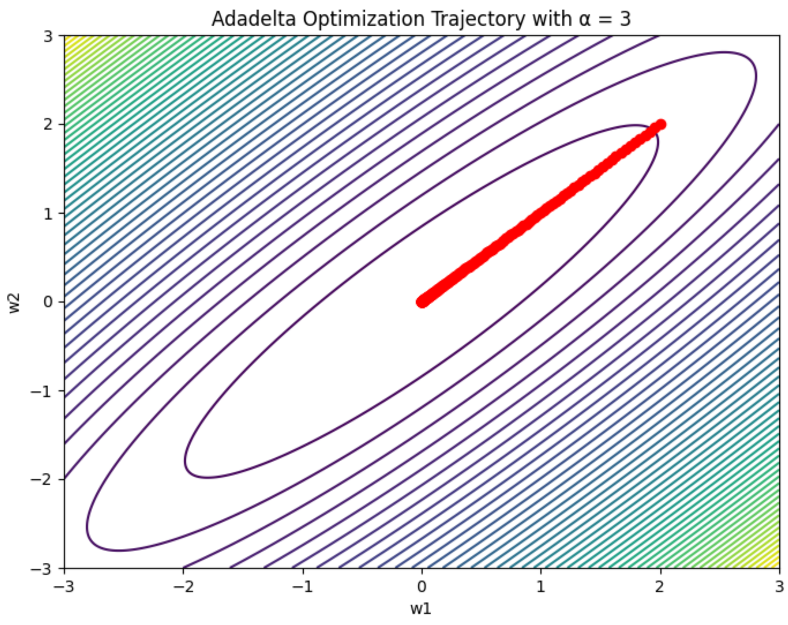
We concluded that it Converged in 199 iterations. The minimum weights (w_min) are [1.64153180e-07 6.89568224e-08] (basically [0, 0] and the minimum value of the function is 1.2204713374471352e-14 (basically 0).

8.3 Part 3

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

# Function F(w)
def F(w):
  return 0.1 * (w[0] + w[1])**2 + 2 * (w[0] - w[1])**2

# Gradient of F(w)
def grad_F(w):
  return np.array([0.2 * (w[0] + w[1]) + 4 * (w[0] - w[1]),
             0.2 * (w[0] + w[1]) - 4 * (w[0] - w[1])])

# Adadelta optimizer function
```

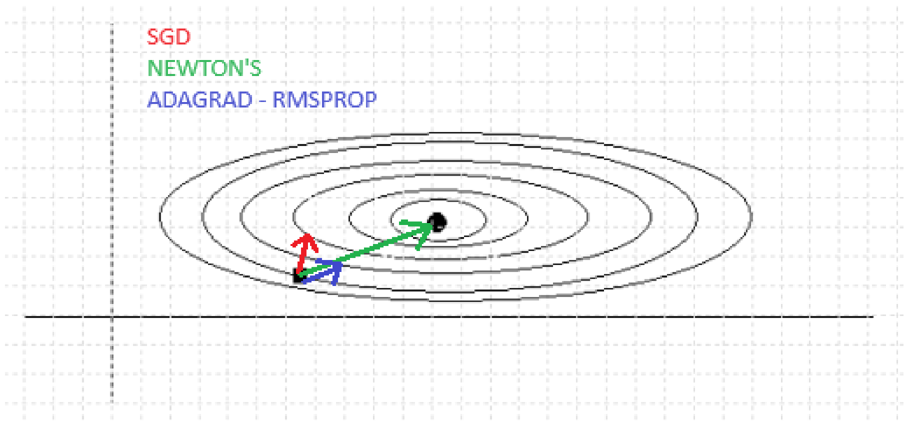
```
def adadelta_optimizer(grad_f, w_init, lr, rho=0.95, epsilon=1e
    -5, max_iter=1000, tol=1e-7):
  w = w_init
  trajectory = [w]
  E_g2 = np.zeros_like(w)
  E_delta_w2 = np.zeros_like(w)

  for i in range(max_iter):
    g = grad_f(w)
    E_g2 = rho * E_g2 + (1 - rho) * g**2
    delta_w = - (np.sqrt(E_delta_w2 + epsilon) / np.sqrt(E_g2 +
    epsilon)) * g
    E_delta_w2 = rho * E_delta_w2 + (1 - rho) * delta_w**2
    w_new = w + lr * delta_w

    # Check for convergence
    if np.linalg.norm(w_new - w) < tol:
      print(f"Converged in {i} iterations")
      w = w_new
      break

    w = w_new
    trajectory.append(w)

  return w, np.array(trajectory)


# Initial parameters
w_init = np.array([2.0, 2.0])  # initial weights
learning_rate = 3  # learning rate

# Optimization
w_min, trajectory = adadelta_optimizer(grad_F, w_init,
    learning_rate)

# Plotting
x = np.linspace(-3, 3, 400)
y = np.linspace(-3, 3, 400)
X, Y = np.meshgrid(x, y)
Z = F([X, Y])

plt.figure(figsize=(8, 6))
plt.contour(X, Y, Z, levels=50)
plt.plot(trajectory[:, 0], trajectory[:, 1], color='r', marker='
    o')
plt.title("Adadelta Optimization Trajectory with α = 3")
plt.xlabel("w1")
plt.ylabel("w2")
plt.show()
```

```
min_value = F(w_min)

# Results
print("Minimum weights (w_min):", w_min)
print("Minimum value of the function (F(w_min)):", min_value)
```



We concluded that it converged in 198 iterations. The minimum weights (w_min) are [1.30130871e-07 1.30130871e-07] (basically [0, 0] and the minimum value of the function is 6.773617470907017e-15 (basically 0). The Adadelta optimizer does not behave differently for the rotated objective function. It converges in almost the same iterations and the values are 0.

9 PROBLEM-09



- **Standard gradient:** This method calculates the gradient of the loss function with respect to the parameters at the current position (black square), and then moves in the opposite direction of the gradient (because we want to minimize the function). The size of the step is determined by the learning rate. On a contour plot, the standard gradient descent would move perpendicularly to the contour lines towards the direction where the function decreases most rapidly. It does not take into account the curvature of the function being minimized thus the arrow not going directly in the minimum.
- **Natural gradient (or Newton's method):** Newton's Method efficiently finds a function's minimum by leveraging the second-order information encapsulated in the Hessian matrix, which contains the second-order partial derivatives of the function. This matrix is a representation of the curvature of the function's surface. By applying the inverse of the Hessian, Newton's Method adjusts the gradient vector, allowing for scaling that reflects the curvature of the function. The unique efficiency of Newton's Method, particularly with quadratic functions, stems from its foundation in the second-order Taylor series expansion. This expansion allows Newton's Method to form a quadratic approximation of the function around the current point. When the function being minimized is itself quadratic, this approximation is exact. Thus, Newton's Method is able to locate the function's minimum in a single iteration because the step calculated from the quadratic approximation directly leads to the stationary point, which is the minimum for a convex quadratic function. By observing the plot we can see that it is derived from a quadratic function. This is why, for quadratic functions with a distinct minimum, Newton's Method achieves convergence in one iteration.
- **Adagrad or RMSprop (assume they have run for a while to accumulate gradient information):** Adagrad and RMSprop are adaptive learning rate methods that adjust the step size for each parameter based on the history of gradients. They reduce

the step size for parameters with large historical gradients to prevent overshooting, while increasing it for parameters with smaller or less frequent gradients, promoting exploration. After having been in operation for some time, these algorithms accumulate substantial gradient information which allows them to take more informed and subtle steps, not perpendicular to the contour lines, but towards the minimum. As they get closer to the minimum, the step size generally becomes smaller because it has accumulated the gradient information.

10 PROBLEM-10

For certain values of β and ν, this update rule resembles well-known optimization algorithms:

Gradient Descent: This is the most basic form of optimization. For gradient descent, there is no momentum term; hence, $\beta = 0$ and $\nu = 0$. The update rules simplify to:

$$g_{t+1} \leftarrow \nabla_{\theta_t} L(\theta_t)$$
$$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \nabla_{\theta_t} L(\theta_t)$$

Momentum (or SGD with momentum): This method introduces a momentum term to the gradient, which helps accelerate gradients vectors in the right directions, thus leading to faster converging. It is characterized by $\beta \neq 0$ and $\nu = 0$. The equations are:

$$g_{t+1} \leftarrow \beta \cdot g_t + (1-\beta) \cdot \nabla_{\theta_t} L(\theta_t)$$
$$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot g_{t+1}$$

11 PROBLEM-11

11.1 Part A

We apply the convolution layer in the input image with kernel: $K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ and stride: s = (1, 1), which produces a 4×4 output array:

Output = $\begin{bmatrix} 225 & 258 & 250 & 209 \\ 458 & 566 & 552 & 472 \\ 708 & 981 & 887 & 802 \\ 1000 & 1488 & 1320 & 1224 \end{bmatrix}$

The elements occur like this:

Output[0][0] = $(1{\cdot}20+1{\cdot}35+1{\cdot}35+1{\cdot}29+0{\cdot}46+1{\cdot}44+1{\cdot}16+1{\cdot}25+1{\cdot}21) = 225.$
Output[0][1] = $(1{\cdot}35+1{\cdot}35+1{\cdot}35+1{\cdot}46+0{\cdot}44+1{\cdot}42+1{\cdot}25+1{\cdot}21+1{\cdot}19) = 258.$
Output[0][2] = $(1{\cdot}35+1{\cdot}35+1{\cdot}20+1{\cdot}44+0{\cdot}42+1{\cdot}27+1{\cdot}21+1{\cdot}19+1{\cdot}12) = 250.$

Output[0][3] = $(1\cdot35+1\cdot20+1\cdot0+1\cdot42+0\cdot27+1\cdot0+1\cdot19+1\cdot12+1\cdot0) = 209.$

Output[1][0] = $(1\cdot29+1\cdot46+1\cdot44+1\cdot16+0\cdot25+1\cdot21+1\cdot66+1\cdot120+1\cdot116) = 458.$

Output[1][1] = $(1\cdot46+1\cdot44+1\cdot42+1\cdot25+0\cdot21+1\cdot19+1\cdot120+1\cdot116+1\cdot154) = 566.$

Output[1][2] = $(1\cdot44+1\cdot42+1\cdot42+1\cdot21+0\cdot19+1\cdot19+1\cdot116+1\cdot154+1\cdot114) = 552.$

Output[1][3] = $(1\cdot42+1\cdot42+1\cdot27+1\cdot19+0\cdot19+1\cdot12+1\cdot154+1\cdot114+1\cdot62) = 472.$

Output[2][0] = $(1\cdot16+1\cdot25+1\cdot21+1\cdot66+0\cdot120+1\cdot116+1\cdot74+1\cdot216+1\cdot174) = 708.$

Output[2][1] = $(1\cdot25+1\cdot21+1\cdot19+1\cdot120+0\cdot116+1\cdot154+1\cdot216+1\cdot174+1\cdot252) = 981.$

Output[2][2] = $(1\cdot21+1\cdot19+1\cdot19+1\cdot116+0\cdot154+1\cdot114+1\cdot174+1\cdot252+1\cdot172) = 887.$

Output[2][3] = $(1\cdot19+1\cdot19+1\cdot12+1\cdot154+0\cdot114+1\cdot62+1\cdot252+1\cdot172+1\cdot112) = 802.$

Output[3][0] = $(1\cdot66+1\cdot120+1\cdot116+1\cdot74+0\cdot216+1\cdot174+1\cdot70+1\cdot210+1\cdot170) = 1000.$

Output[3][1] = $(1\cdot120+1\cdot116+1\cdot154+1\cdot216+0\cdot174+1\cdot252+1\cdot210+1\cdot170+1\cdot250) = 1488.$

Output[3][2] = $(1\cdot116+1\cdot154+1\cdot114+1\cdot174+0\cdot252+1\cdot172+1\cdot170+1\cdot250+1\cdot170) = 1320.$

Output[3][3] = $(1\cdot154+1\cdot114+1\cdot62+1\cdot252+0\cdot172+1\cdot112+1\cdot250+1\cdot170+1\cdot110) = 1224.$

```
import numpy as np

# Input image
I = np.array([
  [20, 35, 35, 35, 35, 20],
  [29, 46, 44, 42, 42, 27],
  [16, 25, 21, 19, 19, 12],
  [66, 120, 116, 154, 114, 62],
  [74, 216, 174, 252, 172, 112],
  [70, 210, 170, 250, 170, 110]
])

# Convolution kernel
kernel = np.array([
  [1, 1, 1],
  [1, 0, 1],
  [1, 1, 1]
])

# Convolution operation
def convolve2d(image, kernel, stride):
  kernel_height, kernel_width = kernel.shape
  image_height, image_width = image.shape
  output_height = 1 + (image_height - kernel_height) // stride
  output_width = 1 + (image_width - kernel_width) // stride

  new_image = np.zeros((output_height, output_width))

  for y in range(output_height):
    for x in range(output_width):
      end_y = y * stride + kernel_height
      end_x = x * stride + kernel_width
      if end_y <= image_height and end_x <= image_width:
        new_image[y, x] = np.sum(image[y * stride:end_y, x *
   stride:end_x] * kernel)

  return new_image.astype(int)

# Apply convolution with stride (1, 1)
output_convolution = convolve2d(I, kernel, stride=1)
print(output_convolution)
```

11.2 Part B

We apply the max pooling layer with window shape = (2, 2) and stride: s = (2, 2) in the Output array, which produces a 2 × 2 array:

Output = $\begin{bmatrix} 566 & 552 \\ 1488 & 1320 \end{bmatrix}$

The elements occur like this:

Output[0][0] = $max(225, 258, 458, 566) = 566$.
Output[0][1] = $max(250, 209, 552, 472) = 552$.
Output[1][0] = $max(708, 981, 1000, 1488) = 1488$.
Output[1][1] = $max(887, 802, 1320, 1224) = 1320$.

```
import numpy as np

# Input image
I = np.array([
  [20, 35, 35, 35, 35, 20],
  [29, 46, 44, 42, 42, 27],
  [16, 25, 21, 19, 19, 12],
  [66, 120, 116, 154, 114, 62],
  [74, 216, 174, 252, 172, 112],
  [70, 210, 170, 250, 170, 110]
])

# Convolution kernel
kernel = np.array([
  [1, 1, 1],
  [1, 0, 1],
  [1, 1, 1]
])

# Convolution operation
def convolve2d(image, kernel, stride):
  kernel_height, kernel_width = kernel.shape
  image_height, image_width = image.shape

  output_height = 1 + (image_height - kernel_height) // stride
  output_width = 1 + (image_width - kernel_width) // stride

  new_image = np.zeros((output_height, output_width))

  for y in range(output_height):
    for x in range(output_width):
      end_y = y * stride + kernel_height
      end_x = x * stride + kernel_width
      if end_y <= image_height and end_x <= image_width:
        new_image[y, x] = np.sum(image[y * stride:end_y, x *
    stride:end_x] * kernel)

  return new_image.astype(int)

# Max pooling operation
def max_pooling(image, pool_size, stride):
  # Calculate the output dimensions
  output_height = (image.shape[0] - pool_size) // stride + 1
  output_width = (image.shape[1] - pool_size) // stride + 1

  # Create the output array
  new_image = np.zeros((output_height, output_width))

  # Apply the pooling operation
  for y in range(0, output_height):
    for x in range(0, output_width):
      new_image[y, x] = np.max(image[y * stride:y * stride +
    pool_size, x * stride:x * stride + pool_size])
  return new_image.astype(int)

# Apply convolution with stride (1, 1)
output_convolution = convolve2d(I, kernel, stride=1)
print(output_convolution)

# Apply max pooling with size (2, 2) and stride (2, 2)
output_max_pooling = max_pooling(output_convolution, pool_size
    =2, stride=2)
print(output_max_pooling)
```

11.3 Part C

- We apply the convolution layer in the input image with KernelF1: $K = \begin{bmatrix} -10 & -10 & -10 \\ 5 & 5 & 5 \\ -10 & -10 & -10 \end{bmatrix}$ and stride: s = (1, 1), which produces a 4×4 output array:

  Output = $\begin{bmatrix} -925 & -1040 & -1000 & -845 \\ -3900 & -4895 & -4825 & -4160 \\ -3750 & -5120 & -4650 & -4210 \\ -5200 & -6990 & -6750 & -5920 \end{bmatrix}$

  The elements occur like this:

  Output[0][0] = $(-10 \cdot 20 + -10 \cdot 35 + -10 \cdot 35 + 5 \cdot 29 + 5 \cdot 46 + 5 \cdot 44 + -10 \cdot 16 + -10 \cdot 25 + -10 \cdot 21) = -925.$
  Output[0][1] = $(-10 \cdot 35 + -10 \cdot 35 + -10 \cdot 35 + 5 \cdot 46 + 5 \cdot 44 + 5 \cdot 42 + -10 \cdot 25 + -10 \cdot 21 + -10 \cdot 19) = -1040.$
  Output[0][2] = $(-10 \cdot 35 + -10 \cdot 35 + -10 \cdot 20 + 5 \cdot 44 + 5 \cdot 42 + 5 \cdot 27 + -10 \cdot 21 + -10 \cdot 19 + -10 \cdot 12) = -1000.$
  Output[0][3] = $(-10 \cdot 35 + -10 \cdot 20 + -10 \cdot 0 + 5 \cdot 42 + 5 \cdot 27 + 5 \cdot 0 + -10 \cdot 19 + -10 \cdot 12 + -10 \cdot 0) = -845.$
  Output[1][0] = $(-10 \cdot 29 + -10 \cdot 46 + -10 \cdot 44 + 5 \cdot 16 + 5 \cdot 25 + 5 \cdot 21 + -10 \cdot 66 + -10 \cdot 120 + -10 \cdot 116) = -3900.$
  Output[1][1] = $(-10 \cdot 46 + -10 \cdot 44 + -10 \cdot 42 + 5 \cdot 25 + 5 \cdot 21 + 5 \cdot 19 + -10 \cdot 120 + -10 \cdot 116 + -10 \cdot 154) = -4895.$
  Output[1][2] = $(-10 \cdot 44) + (-10 \cdot 42) + (-10 \cdot 42) + (5 \cdot 21) + (5 \cdot 19) + (5 \cdot 19) + (-10 \cdot 116) + (-10 \cdot 154) + (-10 \cdot 114) = -4825.$
  Output[1][3] = $(-10 \cdot 42) + (-10 \cdot 42) + (-10 \cdot 27) + (5 \cdot 19) + (5 \cdot 19) + (5 \cdot 12) + (-10 \cdot 154) + (-10 \cdot 114) + (-10 \cdot 62) = -4160.$
  Output[2][0] = $(-10 \cdot 16) + (-10 \cdot 25) + (-10 \cdot 21) + (5 \cdot 66) + (5 \cdot 120) + (5 \cdot 116) + (-10 \cdot 74) + (-10 \cdot 216) + (-10 \cdot 174) = -3750.$
  Output[2][1] = $(-10 \cdot 25) + (-10 \cdot 21) + (-10 \cdot 19) + (5 \cdot 120) + (5 \cdot 116) + (5 \cdot 154) + (-10 \cdot 216) + (-10 \cdot 174) + (-10 \cdot 252) = -5120.$
  Output[2][2] = $(-10 \cdot 21) + (-10 \cdot 19) + (-10 \cdot 19) + (5 \cdot 116) + (5 \cdot 154) + (5 \cdot 114) + (-10 \cdot 174) + (-10 \cdot 252) + (-10 \cdot 172) = -4650.$
  Output[2][3] = $(-10 \cdot 19) + (-10 \cdot 19) + (-10 \cdot 12) + (5 \cdot 154) + (5 \cdot 114) + (5 \cdot 62) + (-10 \cdot 252) + (-10 \cdot 172) + (-10 \cdot 112) = -4210.$
  Output[3][0] = $(-10 \cdot 66) + (-10 \cdot 120) + (-10 \cdot 116) + (5 \cdot 74) + (5 \cdot 216) + (5 \cdot 174) + (-10 \cdot 70) + (-10 \cdot 210) + (-10 \cdot 170) = -5200.$
  Output[3][1] = $(-10 \cdot 120) + (-10 \cdot 116) + (-10 \cdot 154) + (5 \cdot 216) + (5 \cdot 174) + (5 \cdot 252) + (-10 \cdot 210) + (-10 \cdot 170) + (-10 \cdot 250) = -6990.$
  Output[3][2] = $(-10 \cdot 116) + (-10 \cdot 154) + (-10 \cdot 114) + (5 \cdot 74) + (5 \cdot 216) + (5 \cdot 174) + (-10 \cdot 70) + (-10 \cdot 210) + (-10 \cdot 170) = -6750.$
  Output[3][3] = $(-10 \cdot 154) + (-10 \cdot 114) + (-10 \cdot 62) + (5 \cdot 216) + (5 \cdot 174) + (5 \cdot 112) + (-10 \cdot 210) + (-10 \cdot 170) + (-10 \cdot 110) = -5920.$

This kernel has positive values in the middle row and negative values in the top and bottom rows. This configuration means that F1 is likely designed to extract horizontal edges of an image. (The negative values will respond strongly to dark regions above bright regions, while the positive middle row will respond to bright regions.)

– We apply the convolution layer in the input image with KernelF2: $K = \begin{bmatrix} 2 & 2 & 2 \\ 2 & -12 & 2 \\ 2 & 2 & 2 \end{bmatrix}$ and stride: s = (1, 1), which produces a 4×4 output array:

$$\text{Output} = \begin{bmatrix} -102 & -12 & -4 & -86 \\ 616 & 880 & 876 & 716 \\ -24 & 570 & -74 & 236 \\ -592 & 888 & -384 & 384 \end{bmatrix}$$

The elements occur like this:

Output[0][0] = $(2 \cdot 20) + (2 \cdot 35) + (2 \cdot 35) + (2 \cdot 29) + (-12 \cdot 46) + (2 \cdot 44) + (2 \cdot 16) + (2 \cdot 25) + (2 \cdot 21) = -102$.
Output[0][1] = $(2 \cdot 35) + (2 \cdot 35) + (2 \cdot 35) + (2 \cdot 46) + (-12 \cdot 44) + (2 \cdot 42) + (2 \cdot 25) + (2 \cdot 21) + (2 \cdot 19) = -12$.
Output[0][2] = $(2 \cdot 35) + (2 \cdot 35) + (2 \cdot 20) + (2 \cdot 44) + (-12 \cdot 42) + (2 \cdot 27) + (2 \cdot 21) + (2 \cdot 19) + (2 \cdot 12) = -4$.
Output[0][3] = $(2 \cdot 35) + (2 \cdot 20) + (2 \cdot 27) + (2 \cdot 42) + (-12 \cdot 27) + (2 \cdot 19) + (2 \cdot 19) + (2 \cdot 12) + (2 \cdot 12) = -86$.
Output[1][0] = $(2 \cdot 29) + (2 \cdot 46) + (2 \cdot 44) + (2 \cdot 16) + (-12 \cdot 25) + (2 \cdot 21) + (2 \cdot 66) + (2 \cdot 120) + (2 \cdot 116) = 616$.
Output[1][1] = $(2 \cdot 46) + (2 \cdot 44) + (2 \cdot 42) + (2 \cdot 25) + (-12 \cdot 21) + (2 \cdot 19) + (2 \cdot 120) + (2 \cdot 116) + (2 \cdot 154) = 880$.
Output[1][2] = $(2 \cdot 44) + (2 \cdot 42) + (2 \cdot 42) + (2 \cdot 21) + (-12 \cdot 19) + (2 \cdot 19) + (2 \cdot 116) + (2 \cdot 154) + (2 \cdot 114) = 876$.
Output[1][3] = $(2 \cdot 42) + (2 \cdot 42) + (2 \cdot 27) + (2 \cdot 19) + (-12 \cdot 19) + (2 \cdot 12) + (2 \cdot 154) + (2 \cdot 114) + (2 \cdot 62) = 716$.
Output[2][0] = $(2 \cdot 16) + (2 \cdot 25) + (2 \cdot 21) + (2 \cdot 66) + (-12 \cdot 120) + (2 \cdot 116) + (2 \cdot 74) + (2 \cdot 216) + (2 \cdot 174) = -24$.
Output[2][1] = $(2 \cdot 25) + (2 \cdot 21) + (2 \cdot 19) + (2 \cdot 120) + (-12 \cdot 116) + (2 \cdot 154) + (2 \cdot 216) + (2 \cdot 174) + (2 \cdot 252) = 570$.
Output[2][2] = $(2 \cdot 21) + (2 \cdot 19) + (2 \cdot 19) + (2 \cdot 116) + (-12 \cdot 154) + (2 \cdot 114) + (2 \cdot 174) + (2 \cdot 252) + (2 \cdot 172) = -74$.
Output[2][3] = $(2 \cdot 19) + (2 \cdot 19) + (2 \cdot 12) + (2 \cdot 154) + (-12 \cdot 114) + (2 \cdot 62) + (2 \cdot 252) + (2 \cdot 172) + (2 \cdot 112) = 236$.
Output[3][0] = $(2 \cdot 66) + (2 \cdot 120) + (2 \cdot 116) + (2 \cdot 74) + (-12 \cdot 216) + (2 \cdot 174) + (2 \cdot 70) + (2 \cdot 210) + (2 \cdot 170) = -592$.
Output[3][1] = $(2 \cdot 120) + (2 \cdot 116) + (2 \cdot 154) + (2 \cdot 216) + (-12 \cdot 174) + (2 \cdot 252) + (2 \cdot 210) + (2 \cdot 170) + (2 \cdot 250) = 888$.
Output[3][2] = $(2 \cdot 116) + (2 \cdot 154) + (2 \cdot 114) + (2 \cdot 174) + (-12 \cdot 252) + (2 \cdot 172) + (2 \cdot 170) + (2 \cdot 250) + (2 \cdot 170) = -384$.

Output[3][3] = $(2 \cdot 154) + (2 \cdot 114) + (2 \cdot 62) + (2 \cdot 252) + (-12 \cdot 172) + (2 \cdot 112) + (2 \cdot 250) + (2 \cdot 170) + (2 \cdot 110) = 384$.

Kernel F2 has a strong negative value in the center with positive values surrounding it. This could be used for detecting a center-surround structure or a small spot. It might act like a sort of ”inverse” Laplacian operator, highlighting regions that are surrounded by similar intensity while reducing the response of isolated points or noise.

– We apply the convolution layer in the input image with KernelF3: $K = \begin{bmatrix} -20 & -10 & 0 & 5 & 10 \\ -10 & 0 & 5 & 10 & 5 \\ 0 & 5 & 10 & 5 & 0 \\ 5 & 10 & 5 & 0 & -10 \\ 10 & 5 & 0 & -10 & -20 \end{bmatrix}$

and stride: s = (1, 1), which produces a 2×2 output array:

Output = $\begin{bmatrix} -2405 & 1000 \\ -120 & 3915 \end{bmatrix}$

The elements occur like this:

Output[0][0] = $(-20 \cdot 20) + (-10 \cdot 35) + (0 \cdot 35) + (5 \cdot 29) + (10 \cdot 46) + (-10 \cdot 44) + (5 \cdot 16) + (10 \cdot 25) + (-20 \cdot 21) = -2405$.
Output[0][1] = $(-20 \cdot 35) + (-10 \cdot 35) + (5 \cdot 35) + (10 \cdot 46) + (5 \cdot 44) + (-10 \cdot 42) + (5 \cdot 25) + (10 \cdot 21) + (-20 \cdot 19) = 1000$.
Output[1][0] = $(-10 \cdot 20) + (0 \cdot 35) + (5 \cdot 35) + (10 \cdot 29) + (5 \cdot 46) + (0 \cdot 44) + (5 \cdot 16) + (10 \cdot 25) + (-10 \cdot 21) = -120$.
Output[1][1] = $(-10 \cdot 35) + (0 \cdot 35) + (5 \cdot 35) + (10 \cdot 46) + (5 \cdot 44) + (0 \cdot 42) + (5 \cdot 25) + (10 \cdot 21) + (-10 \cdot 19) = 3915$.

Kernel F3 seems to be designed to detect a diagonal edge from bottom left to top right. The values gradually increase from the bottom left to the center and then decrease symmetrically. This pattern will respond to gradients or edges that match this diagonal orientation.

```
import numpy as np

# Input image
I = np.array([
  [20, 35, 35, 35, 35, 20],
  [29, 46, 44, 42, 42, 27],
  [16, 25, 21, 19, 19, 12],
  [66, 120, 116, 154, 114, 62],
  [74, 216, 174, 252, 172, 112],
  [70, 210, 170, 250, 170, 110]
])

# Convolution kernel
f1= np.array([
  [-10, -10, -10],
  [5, 5, 5],
  [-10, -10, -10]
])

f2= np.array([
  [2, 2, 2],
  [2, -12, 2],
  [2, 2, 2]
])

f3= np.array([
  [-20, -10, 0, 5, 10],
  [-10, 0, 5, 10, 5],
  [0, 5, 10, 5, 0],
  [5, 10, 5, 0, -10],
  [10, 5, 0, -10, -20]
])

# Convolution operation
def convolve2d(image, kernel, stride):
  kernel_height, kernel_width = kernel.shape
  image_height, image_width = image.shape

  # Correct the calculation of output dimensions
  output_height = 1 + (image_height - kernel_height) //
    stride
  output_width = 1 + (image_width - kernel_width) // stride

  new_image = np.zeros((output_height, output_width))

  for y in range(output_height):
    for x in range(output_width):
      # Adjust the slicing to avoid going out of image bounds
      end_y = y * stride + kernel_height
      end_x = x * stride + kernel_width
      if end_y <= image_height and end_x <= image_width:
        new_image[y, x] = np.sum(image[y * stride:end_y, x *
    stride:end_x] * kernel)

  return new_image.astype(int)

# Apply convolution with stride (1, 1)
output_convolution = convolve2d(I, f1, stride=1)
print(output_convolution)
output_convolution = convolve2d(I, f2, stride=1)
print(output_convolution)
output_convolution = convolve2d(I, f3, stride=1)
print(output_convolution)
```

12 PROBLEM-12

For the first convolutional layer:

- Kernel size: 3
- Number of input channels: 3
- Number of output channels (filters): 4

The number of weights is given by:

$$\text{Number of weights} = \text{Kernel size} \times \text{Number of input channels} \times \text{Number of output channels}$$

$$\text{Number of weights} = 3 \times 3 \times 4 = 36$$

The number of biases is equal to the number of output channels:

$$\text{Number of biases} = \text{Number of output channels}$$

$$\text{Number of biases} = 4$$

For the second convolutional layer:

- Kernel size: 5
- Number of input channels: 4 (output channels of the first layer)
- Number of output channels (filters): 10

The number of weights is given by:

$$\text{Number of weights} = \text{Kernel size} \times \text{Number of input channels} \times \text{Number of output channels}$$

$$\text{Number of weights} = 5 \times 4 \times 10 = 200$$

The number of biases is equal to the number of output channels:

$$\text{Number of biases} = \text{Number of output channels}$$

$$\text{Number of biases} = 10$$

13 PROBLEM-13

13.1 Part-A

To express max(a, b) using only ReLU operations, we can use the property of ReLU which is defined as ReLU(x) = max$(0, x)$
The max of two numbers a and b can be computed as:

$$\max(a, b) = \text{ReLU}(a - b) + b$$

This works because if $a > b$, then $ReLU(a - b)$ will be $a - b$, and adding b gives us a. If $a \leq b$,then $ReLU(a - b)$ will be 0, and thus the result is b.

13.2 Part-B

Max-pooling can be achieved by applying a convolutional operation followed by a ReLU activation function. Here's how it can be done:

1. Perform a convolution over the input tensor using a kernel filled with ones and the appropriate stride to cover the pooling size. This convolution will effectively sum the elements in each pooling region.

2. Duplicate the input tensor into as many channels as there are elements in the pooling region. Each channel will undergo a convolution with a kernel that has a single element set to one and the rest set to zero, effectively picking out individual elements.

3. Apply the ReLU function on the difference between the tensor obtained from step 1 and each of the convolved tensors from step 2. The ReLU function will zero out negative values, which correspond to the non-maximum elements.

4. Add back the individual elements picked out in step 2 to the results of step 3. The ReLU function ensures that only the maximum element from each pooling region contributes to the final result.

13.3 Part-C

An $n \times n$ convolution needs n^2 channels and layer, so we need 4 for 2×2 convolution, 9 for 3×3 convolution.

14 PROBLEM-14

To calculate the total number of weights in the described convolutional neural network (CNN), we consider each layer's structure using the formula (num_filters * filter_size * filter_size * num_channels) + num_filters for a convolution and (num_units * num_inputs) + num_units for a dense layer:

- **Layer 1**: Convolutional layer with 100 5×5 filters.
  Each filter has $5 \times 5 = 25$ weights for a grayscale image.
  Total weights for Layer 1: $100 \times 25 + 100 = 2600$.
- **Layer 2**: Convolutional layer with 100 5×5 filters.
  Each filter has $5 \times 5 = 25$ weights, and there are 100 channels from the previous layer.
  Total weights for Layer 2: $100 \times 25 \times 100 + 100 = 250,100$.
- **Layer 3**: Max pooling layer.
  Max pooling layers do not have weights.
- **Layer 4**: Dense layer with 100 units.
  This layer is fully connected to the output of Layer 3, which is $50 \times 50 \times 100$ nodes after pooling.
  Total weights for Layer 4: $50 \times 50 \times 100 \times 100 + 100 = 25,000,100$.
- **Layer 5**: Dense layer with 100 units.
  Each of the 100 units in Layer 4 is connected to each of the 100 units in Layer 5.
  Total weights for Layer 5: $100 \times 100 + 100 = 10,100$.

- **Layer 6**: Single output unit.
  This output unit is fully connected to the 100 units of Layer 5.
  Total weights for Layer 6: $100 \times 1 = 100$.

Therefore, the total number of weights in the CNN is calculated as follows:

Total Weights = Weights for Layer 1 + Weights for Layer 2 + Weights for Layer 4 + Weights for Layer 5 + Weights for Layer 6

$$\text{Total Weights} = 2600 + 250,100 + 25,000,100 + 10,100 + 100 = 25,263,000$$

15 PROBLEM-15

15.1 Part A

The alternative approach of reading a k + Δ wide strip and computing a Δ-wide output strip, offers several advantages that can lead to more efficient convolution operations:

- **Improved Memory Access Pattern:** By reading a larger strip of input data (k + Δ wide), this approach enhances data reuse. It better utilizes the memory hierarchy and cache, reducing the frequency of memory accesses compared to reading k-wide strips multiple times. This efficiency in memory access can significantly improve performance, particularly for large images or kernels.
- **Reduced Loop Overhead:** Reading larger strips of data in fewer iterations diminishes the loop overhead and branching in the code. This reduction in computational overhead can contribute to overall faster execution times.
- **Enhanced Parallelization:** A wider strip allows for the computation of multiple output values simultaneously. This characteristic is particularly beneficial on modern hardware architectures, such as GPUs or multi-core CPUs, where parallel processing capabilities can be fully exploited. Such parallelism can dramatically speed up convolution operations.

However, selecting an overly large Δ has its drawbacks. A balance must be struck, considering several factors:

- **Memory Constraints:** A larger Δ increases the memory required to store the input strip. In systems with limited memory, this could lead to inefficient use of resources or even out-of-memory errors.
- **Cache Size and Thrashing:** As Δ increases, the risk of cache misses and cache thrashing also rises. This can offset the benefits of increased data reuse if the strip size exceeds the cache capacity.
- **Computational Overhead:** A larger Δ means more data to process per iteration, which can introduce additional computational load. This might be counterproductive, especially if the increase in data processing does not proportionally translate to performance gains.

Finding the optimal Δ is a matter of balancing the increased data reuse and parallelism against the potential drawbacks of higher memory usage and computational complexity. This balance can vary depending on the specific hardware architecture and the nature of the data being processed. Profiling and experimenting with different Δ values is essential to determine the most effective choice.

15.2 Part B

We calculated the average times of the methods on each kernel size and these are the results after 5 experimental runs:

1. **3x3 kernel:**
   - Method 1 (k-wide strip for 1-wide output): 1.253 seconds
   - Method 2 (k + Δ wide strip for Δ-wide output): 0.048 seconds
   - **Observation:** For the smallest kernel size, Method 2 significantly outperforms Method 1. This suggests that there is no overhead associated with handling larger strips in Method 2 when dealing with small kernels.
2. **7x7 kernel:**
   - Method 1 (k-wide strip for 1-wide output): 1.242 seconds
   - Method 2 (k + Δ wide strip for Δ-wide output): 0.108 seconds
   - **Observation:** In the case of the 7x7 kernel, Method 2 also outperforms Method 1. This indicates that the benefits of Method 2, such as improved data reuse and cache efficiency. The wider strip read in Method 2 allows for more effective use of loaded data, reducing redundant memory accesses and potentially enhancing parallelism.
3. **11x11 kernel:**
   - Method 1 (k-wide strip for 1-wide output): 1.240 seconds
   - Method 2 (k + Δ wide strip for Δ-wide output): 0.260 seconds
   - **Observation:** With the 11x11 kernel, the trend observed with the 7x7 kernel continues, with Method 2 outperforming Method 1. This further reinforces the idea that Method 2's approach is more suitable for larger kernels. However, as the size of the strip increases, we observe diminishing returns in performance benefits due to computational overhead, memory constraints, and cache thrashing.

```python
import numpy as np
import time

image = np.random.rand(228, 228)

kernels = {
  "3x3": np.random.rand(3, 3),
  "7x7": np.random.rand(7, 7),
  "11x11": np.random.rand(11, 11)
}

def convolve2d(image, kernel):
  kernel_height, kernel_width = kernel.shape
  image_height, image_width = image.shape
  output_height = image_height - kernel_height + 1
  output_width = image_width - kernel_width + 1
  new_image = np.zeros((output_height, output_width))
  for y in range(output_height):
    for x in range(output_width):
      new_image[y, x] = np.sum(image[y:y+kernel_height, x:x+
    kernel_width] * kernel)
  return new_image

# Calculating sliding window with delta = kernel size
def sliding_window(image, kernel_shape):
  kernelH, kernelW = kernel_shape
  imageH, imageW = image.shape
  h, w = imageH + 1 - kernelH, imageW + 1 - kernelW

  filter1 = np.arange(kernelW) + np.arange(h)[:, np.newaxis]
  intermediate = image[filter1]
  intermediate = np.transpose(intermediate, (0, 2, 1))

  filter2 = np.arange(kernelH) + np.arange(w)[:, np.newaxis]
  intermediate = intermediate[:, filter2]
  final = np.transpose(intermediate, (0, 1, 3, 2))

  return final

execution_times = {}

for kernel_size, kernel in kernels.items():
  start_time = time.time()
  output1 = convolve2d(image, kernel)
  time_method1 = time.time() - start_time

  start_time = time.time()
  prepared_image = sliding_window(image, kernel.shape)
  output2 = np.sum(prepared_image * kernel, axis=(2, 3))
  time_method2 = time.time() - start_time

  execution_times[kernel_size] = {"Method 1": time_method1, "
    Method 2": time_method2}

print(execution_times)
```