

4 Assignment 4

4.1 Finding Critical Points and Local/Global Minimizers

4.1.A

Consider the following function $f : \mathbb{R}^n \rightarrow \mathbb{R}$:

$$f(\mathbf{x}) = x_1 \sin(x_2) \quad (1)$$

We would like to find the critical points of this function. To begin, we will compute the first partial derivatives of the function:

$$f_{x_1} = \frac{\partial f}{\partial x_1} = \sin(x_2) \quad (2)$$

$$f_{x_2} = \frac{\partial f}{\partial x_2} = x_1 \cos(x_2) \quad (3)$$

A critical point $\mathbf{x}^* \in \mathbb{R}^n$ of the function will exist where both partial derivatives are 0. Thus, we will need to find:

$$\mathbf{x}^* = \begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} \quad (4)$$

Such that:

$$f_{x_1}(x_1^*, x_2^*) = f_{x_2}(x_1^*, x_2^*) = 0 \quad (5)$$

Thus, we have the following system of equations which, once solved, will give us our critical point \mathbf{x}^* :

$$\sin(x_2^*) = 0 \quad (6)$$

$$x_1^* \cos(x_2^*) = 0 \quad (7)$$

Consider first the case where:

$$x_2^* = \frac{\pi}{2}(2k - 1), \quad k \in \mathbf{Z} \quad (8)$$

As this would satisfy (7). Subbing this into (6) we get:

$$\sin(x_2^*) = \sin\left(\frac{\pi}{2}(2k - 1)\right) = \pm 1 \neq 0 \quad (9)$$

invalidating our initial guess for x_2^* . Consider instead $x_2^* = k\pi, k \in \mathbf{Z}$ which would satisfy (6). Subbing this into (7) we get:

$$x_1^* \cos(x_2^*) = x_1^* \cos(k\pi) = x_1^*(\pm 1) \quad (10)$$

And so we can set $x_1^* = 0$ to satisfy (7) with $x_2^* = k\pi$. This means our critical points are:

$$\mathbf{x}^* = \begin{bmatrix} 0 \\ k\pi \end{bmatrix}, \quad k \in \mathbf{Z} \quad (11)$$

To check if any of these critical points are local or global minimizers, we will compute the Hessian of the function, compute its eigenvalues, and then substitute our critical points into the eigenvalues. If all eigenvalues are non-negative, the Hessian is positive semi-definite, meaning the critical point is a local minimum. The Hessian of $f(\mathbf{x}) = x_1 \sin(x_2)$ and its eigenvalues, computed with Wolfram Alpha, are as follows:

$$H = \begin{bmatrix} 0 & \cos(x_2) \\ \cos(x_2) & -x_1 \sin(x_2) \end{bmatrix} \quad (12)$$

$$\lambda_1 = \frac{-\sqrt{2}}{4} \sqrt{(x_1^2(-\cos(2x_2)) + x_1^2 + 4\cos(2x_2) + 4) - 2x_1 \sin(x_2)} \quad (13)$$

$$\lambda_2 = \frac{\sqrt{2}}{4} \sqrt{(x_1^2(-\cos(2x_2)) + x_1^2 + 4\cos(2x_2) + 4) - 2x_1 \sin(x_2)} \quad (14)$$

Substituting our \mathbf{x}^* into λ_1 , we get:

$$\lambda_1 = \frac{-\sqrt{2}}{4} \sqrt{0 + 0 + 4\cos(2k\pi + 4)} - 0 \quad (15)$$

And since the cosine of any even multiple of π is $+1$, the term under the root is positive, and λ_1 is negative. Thus, not all eigenvalues of the Hessian are non-negative and the Hessian is not positive semi-definite, thus our critical point \mathbf{x}^* is not a local minimizer. And since \mathbf{x}^* is not a local minimum, it certainly is not a global minimum.

4.1.B

Consider the following function $f : \mathbb{R}^n \rightarrow \mathbb{R}$:

$$f(\mathbf{x}) = \frac{x_1}{1 + x_1^2 + x_2^2} \quad (16)$$

We would like to find the critical points of this function. To begin, we will compute the first partial derivatives of the function:

$$f_{x_1} = \frac{\partial f}{\partial x_1} = \frac{-x_1^2 + x_2^2 + 1}{(1 + x_1^2 + x_2^2)^2} \quad (17)$$

$$f_{x_2} = \frac{\partial f}{\partial x_2} = \frac{-2x_1 x_2}{(1 + x_1^2 + x_2^2)^2} \quad (18)$$

A critical point $\mathbf{x}^* \in \mathbb{R}^n$ of the function will exist where both partial derivatives are 0. Thus, we will need to find:

$$\mathbf{x}^* = \begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} \quad (19)$$

Such that:

$$f_{x_1}(x_1^*, x_2^*) = f_{x_2}(x_1^*, x_2^*) = 0 \quad (20)$$

We know the denominator of the function will always be non-negative, given that it is squared, so we will focus on setting the numerator of the partial derivatives to 0. Thus, we have the following system of equations which, once solved, will give us our critical point \mathbf{x}^* :

$$-x_1^{*2} + x_2^{*2} + 1 = 0 \quad (21)$$

$$-2x_1^* x_2^* = 0 \quad (22)$$

To solve, first consider the case where $x_1^* = 0$ which satisfies (22). Subbing this into (21), we get:

$$0 + x_2^{*2} + 1 = 0 \quad (23)$$

$$x_2^* = \sqrt{-1} \quad (24)$$

Since we define the inputs of f to be real, this is not a valid value for x_2^* . We try again with $x_2^{*2} = 0$ which satisfies (22). Subbing into (21), we get:

$$-x_1^* = \sqrt{1} = \pm 1 \quad (25)$$

And so, our critical points for f are:

$$\mathbf{x}_a^* = \begin{bmatrix} +1 \\ 0 \end{bmatrix}, \quad \mathbf{x}_b^* = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \quad (26)$$

To check if any of these critical points are local or global minimizers, we will compute the Hessian of the function, compute its eigenvalues, and then substitute our critical points into the eigenvalues. If all eigenvalues are non-negative, the Hessian is positive semi-definite, meaning the critical point is a local minimum. The Hessian of $f(\mathbf{x}) = x_1 \sin(x_2)$ and its eigenvalues, computed with Wolfram Alpha, are as follows:

$$H = \begin{bmatrix} \frac{8x^3}{(x^2+y^2+1)^3} - \frac{6x}{(x^2+y^2+1)^2} & \frac{8x^2y}{(x^2+y^2+1)^3} - \frac{2y}{(x^2+y^2+1)^2} \\ \frac{8x^2y}{(x^2+y^2+1)^3} - \frac{2y}{(x^2+y^2+1)^2} & \frac{8xy^2}{(x^2+y^2+1)^3} - \frac{2x}{(x^2+y^2+1)^2} \end{bmatrix} \quad (27)$$

$$\lambda_1 = \frac{2 \left(-\sqrt{x_1^6 + 3x_1^4x_2^2 - 2x_1^4 + 3x_1^2x_2^4 + x_1^2 + x_2^6 + 2x_2^4 + x_2^2} - 2x_1 \right)}{(x_1^2 + x_2^2 + 1)^3} \quad (28)$$

$$\lambda_2 = \frac{2 \left(\sqrt{x_1^6 + 3x_1^4x_2^2 - 2x_1^4 + 3x_1^2x_2^4 + x_1^2 + x_2^6 + 2x_2^4 + x_2^2} - 2x_1 \right)}{(x_1^2 + x_2^2 + 1)^3} \quad (29)$$

We would like to check, now, to see if any of our critical points are local or global minimizers. Substituting \mathbf{x}_a^* into λ_1 , we get:

$$\lambda_1 = \frac{2 \left(-\sqrt{1+0-2+0+1+0+0+0+0} - 2 \right)}{(1+0+1)^3} = \frac{2(-2)}{8} = -\frac{1}{2} \quad (30)$$

Since λ_1 is negative at \mathbf{x}_a^* , \mathbf{x}_a^* is not a local minimum of f . Let's try substituting \mathbf{x}_b^* into λ_1 and λ_2 :

$$\lambda_1 = \frac{2 \left(-\sqrt{1+0-2+0+1+0+0+0+0} + 2 \right)}{(1+0+1)^3} = \frac{2(+2)}{8} = \frac{1}{2} \quad (31)$$

$$\lambda_2 = \frac{2 \left(\sqrt{1+0-2+0+1+0+0+0+0} + 2 \right)}{(1+0+1)^3} = \frac{2(+2)}{8} = \frac{1}{2} \quad (32)$$

And since both eigenvalues of the Hessian of f are positive at \mathbf{x}_b^* , the Hessian is positive semi-definite at that critical point which we know means the point \mathbf{x}_b^* is a local minimum of f . Since the denominator of f grows faster than the numerator, the function is decreasing the further away we move from the origin, approaching 0 in every direction. This means there are no edge points of the domain of the function which could be a global minimum, and so the critical point and local minimum we found, \mathbf{x}_b^* is the global minimum of f .

4.2 Properties of Convex Functions

4.2.A Sum of 2 Convex Functions

We would like to show that the sum of two convex functions is also convex. We begin with the following convex functions:

$$f(\mathbf{x}), \quad f : \mathbb{R}^n \rightarrow \mathbb{R} \quad (33)$$

$$g(\mathbf{x}), \quad g : \mathbb{R}^n \rightarrow \mathbb{R} \quad (34)$$

Each defined by the definition of a convex function:

$$f(t\mathbf{x} + (1-t)\mathbf{x}') \leq tf(\mathbf{x}) + (1-t)f(\mathbf{x}') \quad (35)$$

$$g(t\mathbf{x} + (1-t)\mathbf{x}') \leq tg(\mathbf{x}) + (1-t)g(\mathbf{x}') \quad (36)$$

Where $t \in [0, 1]$ and x, x' are in the domain of f and g (they have the same domain, \mathbb{R}^n). We will also define the sum of f and g as:

$$h(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x}) \quad (37)$$

We would like to show that $h(\mathbf{x})$ is convex; ie. we would like to show:

$$h(t\mathbf{x} + (1-t)\mathbf{x}') \leq th(\mathbf{x}) + (1-t)h(\mathbf{x}') \quad (38)$$

We begin by writing the sum of f and g , by definition:

$$f(t\mathbf{x} + (1-t)\mathbf{x}') + g(t\mathbf{x} + (1-t)\mathbf{x}') \leq tf(\mathbf{x}) + (1-t)f(\mathbf{x}') + tg(\mathbf{x}) + (1-t)g(\mathbf{x}') \quad (39)$$

Factoring out the ts and $(1-t)s$ on the left hand side and rearranging, we get:

$$f(t\mathbf{x} + (1-t)\mathbf{x}') + g(t\mathbf{x} + (1-t)\mathbf{x}') \leq t(f(\mathbf{x}) + g(\mathbf{x})) + (1-t)(f(\mathbf{x}') + g(\mathbf{x}')) \quad (40)$$

Substituting the definition of $h(\mathbf{x})$ (37) into (40), we get:

$$h(t\mathbf{x} + (1-t)\mathbf{x}') \leq th(\mathbf{x}) + (1-t)h(\mathbf{x}') \quad (41)$$

which is (38), the inequality which shows $h(\mathbf{x})$ is convex by the definition of convexity. Since f and g are convex and as we have shown h is convex directly using the definition of convexity, we have shown, in the general case, that the sum of two convex functions is convex.

4.2.B Norm of Convex Functions

Consider the function:

$$\mathbf{Ax} - \mathbf{b}, \quad \mathbf{A} \in \mathbb{R}^{n,n}, \quad \mathbf{b} \in \mathbb{R}^n \quad (42)$$

And consider the norm operator:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2} \quad (43)$$

We would like to show that:

$$\|\mathbf{Ax} - \mathbf{b}\|_2 \quad (44)$$

is convex in \mathbf{x} . We begin by showing $\mathbf{Ax} - \mathbf{b}$ is convex itself to begin with. Then we can show that the norm operator (43) preserves convexity. And finally, we will show that the composition of two functions is also convex. First, define $g : \mathbb{R}^n \rightarrow \mathbb{R}^n$:

$$g(\mathbf{x}) = \mathbf{Ax} - \mathbf{b}, \quad \mathbf{A} \in \mathbb{R}^{n,n}, \quad \mathbf{b} \in \mathbb{R}^n \quad (45)$$

g is the image under an affine map, and as such, is convex if its domain is convex. The domain of g is the set $S = \mathbb{R}^n$. Since the set of reals, \mathbb{R} , is convex, the domain of g is convex, and so, by this, we know g is also be convex. Next, we will show the norm operator is a convex function. Consider next the function $f : \mathbb{R}^n \rightarrow \mathbb{R}$:

$$f(\mathbf{x}) = \|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2} \quad (46)$$

Consider also the p -norm operator, which we know is convex, defined by the following:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n x_i^p \right)^{1/p} \quad (47)$$

We can see that f is simply a special case of the p -norm, where $p = 2$. And thus, we conclude f is also convex. Next, we will use the composition of functions to show that:

$$f(g(\mathbf{x})) = \|\mathbf{Ax} - \mathbf{b}\|_2 \quad (48)$$

is convex. In order for the composition of two functions to be convex, the following conditions must be satisfied:

- a) The dimensionality of f and g must be consistent.
- b) f is non-decreasing.
- c) g is convex.

Since $g : \mathbb{R}^n \rightarrow \mathbb{R}^n$ and $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the dimensions of f and g are consistent. We have already shown that f is convex, thus it is non-decreasing, and we have also shown that g is convex. Thus, we have shown $f(g(\mathbf{x})) = \|\mathbf{Ax} - \mathbf{b}\|_2$ is convex.

4.3

Consider the following functions: Let $\mathbf{C} \in \mathbb{R}^{n,n}$ be fixed and consider the following functions over \mathbb{R}^n :

$$f_{\text{square}}(\mathbf{x}) = \mathbf{x}^\top \mathbf{C} \mathbf{x} \quad (49)$$

and

$$f_{\text{hole}}(\mathbf{x}) = 1 - \exp(\mathbf{x}^\top \mathbf{C} \mathbf{x}). \quad (50)$$

Where $\mathbf{C} \in \mathbb{R}^{n,n}$ is a diagonal matrix with entries:

$$C_{i,i} = 10^{\frac{i-1}{n-1}} \quad (51)$$

4.3.A Gradients

We first compute the gradient of f_{square} :

$$\nabla f_{\text{square}} = \nabla \left[\begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} C_{1,1} & 0 & \cdots & 0 \\ 0 & C_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & C_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \right] \quad (52)$$

$$\nabla f_{\text{square}} = \nabla(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2) = \nabla \sum_{i=1}^n C_{i,i}x_i^2 \quad (53)$$

$$\nabla f_{\text{square}} = \begin{bmatrix} 2C_{1,1}x_1 \\ 2C_{2,2}x_2 \\ \vdots \\ 2C_{n,n}x_n \end{bmatrix} \quad (54)$$

Next we compute the gradient of f_{hole} :

$$\nabla f_{\text{hole}} = \nabla \left(1 - \exp \left(\begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} C_{1,1} & 0 & \cdots & 0 \\ 0 & C_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & C_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \right) \right) \quad (55)$$

$$\nabla f_{\text{hole}} = \nabla(1 - \exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2)) = \nabla(1 - \exp(\sum_{i=1}^n C_{i,i}x_i^2)) \quad (56)$$

$$\nabla f_{\text{hole}} = \begin{bmatrix} -2C_{1,1}x_1 \exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2) \\ -2C_{2,2}x_2 \exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2) \\ \vdots \\ -2C_{n,n}x_n \exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2) \end{bmatrix} \quad (57)$$

Rewriting with sum notation:

$$\nabla f_{\text{hole}} = \begin{bmatrix} -2C_{1,1}x_1 \exp(\sum_{i=1}^n C_{i,i}x_i^2) \\ -2C_{2,2}x_2 \exp(\sum_{i=1}^n C_{i,i}x_i^2) \\ \vdots \\ -2C_{n,n}x_n \exp(\sum_{i=1}^n C_{i,i}x_i^2) \end{bmatrix} \quad (58)$$

4.3.B Hessians

Now, we would like to find the Hessian matrix for f_{square} and f_{hole} . We begin with f_{square} :

$$\nabla^2 f_{square} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_n} \\ \frac{\partial f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_n \partial x_1} & \frac{\partial f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \quad (59)$$

Starting with the (1, 1) element:

$$\frac{\partial^2 f}{\partial x_1^2} = \frac{\partial}{\partial x_1} \frac{\partial f}{\partial x_1} = \frac{\partial}{\partial x_1} (2x_1 C_{1,1}) = 2C_{1,1} \quad (60)$$

Generalizing to all diagonal entries:

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{\partial}{\partial x_i} \frac{\partial f}{\partial x_i} = \frac{\partial}{\partial x_i} (2x_i C_{i,i}) = 2C_{i,i} \quad (61)$$

For the off-diagonal entries, since the first partial derivative is a function of only x_i and constants, taking a second derivative with respect to any $x \neq x_i$ will result in a derivative of 0. Thus, every element of the Hessian of f_{square} which lies off the diagonal is 0. And so:

$$\nabla^2 f_{square} = \begin{bmatrix} 2C_{1,1} & 0 & \cdots & 0 \\ 0 & 2C_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 2C_{n,n} \end{bmatrix} \quad (62)$$

Now, we will find the Hessian matrix for f_{hole} :

$$\nabla^2 f_{hole} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_n} \\ \frac{\partial f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_n \partial x_1} & \frac{\partial f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \quad (63)$$

Starting with the (1, 1) element:

$$\frac{\partial^2 f}{\partial x_1^2} = \frac{\partial}{\partial x_1} \frac{\partial f}{\partial x_1} = \frac{\partial}{\partial x_1} [-2C_{1,1}x_1 \exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2)] \quad (64)$$

$$\frac{\partial^2 f}{\partial x_1^2} = (-2C_{1,1})(1 + 2C_{1,1}x_1^2)(\exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2)) \quad (65)$$

Rewriting with sum notation:

$$\frac{\partial^2 f}{\partial x_1^2} = (-2C_{1,1})(1 + 2C_{1,1}x_1^2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) \quad (66)$$

And generalizing for all diagonal entries:

$$\frac{\partial^2 f}{\partial x_i^2} = (-2C_{i,i})(1 + 2C_{i,i}x_i^2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) \quad (67)$$

Now looking at the off-diagonal entries, starting with $(1, 2)$:

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = \frac{\partial}{\partial x_1} \frac{\partial f}{\partial x_2} = \frac{\partial}{\partial x_1} [-2C_{2,2}x_2 \exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2)] \quad (68)$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = (-4C_{1,1}C_{2,2}x_1x_2)(\exp(C_{1,1}x_1^2 + C_{2,2}x_2^2 + \cdots + C_{n,n}x_n^2)) \quad (69)$$

Rewritten with sum notation:

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = (-4C_{1,1}C_{2,2}x_1x_2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) \quad (70)$$

Thus, the Hessian matrix for f_{hole} is:

$$\nabla^2 f_{hole} = \begin{bmatrix} (-2C_{1,1})(1 + 2C_{1,1}x_1^2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) & (-4C_{1,1}C_{2,2}x_1x_2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) & \cdots & (-4C_{1,1}C_{n,n}x_1x_n)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) \\ (-4C_{1,1}C_{2,2}x_1x_2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) & (-2C_{2,2})(1 + 2C_{2,2}x_2^2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) & \cdots & (-4C_{2,2}C_{n,n}x_2x_n)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) \\ \vdots & \vdots & \ddots & \vdots \\ (-4C_{1,1}C_{n,n}x_1x_n)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) & (-4C_{2,2}C_{n,n}x_2x_n)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) & \cdots & (-2C_{n,n})(1 + 2C_{n,n}x_n^2)(\exp(\sum_{i=1}^n C_{i,i}x_i^2)) \end{bmatrix} \quad (71)$$

To make the entire matrix fit on the page, we can rewrite the Hessian matrix for f_{hole} as follows by factoring out the sum term multiplying each entry:

$$\nabla^2 f_{hole} = \left[\exp \sum_{i=1}^n C_{i,i}x_i^2 \right] \begin{bmatrix} (-2C_{1,1})(1 + 2C_{1,1}x_1^2) & (-4C_{1,1}C_{2,2}x_1x_2) & \cdots & (-4C_{1,1}C_{n,n}x_1x_n) \\ (-4C_{1,1}C_{2,2}x_1x_2) & (-2C_{2,2})(1 + 2C_{2,2}x_2^2) & \cdots & (-4C_{2,2}C_{n,n}x_2x_n) \\ \vdots & \vdots & \ddots & \vdots \\ (-4C_{1,1}C_{n,n}x_1x_n) & (-4C_{2,2}C_{n,n}x_2x_n) & \cdots & (-2C_{n,n})(1 + 2C_{n,n}x_n^2) \end{bmatrix} \quad (72)$$

4.3.C Graphs

Here is a plot of f_{square} on the interval $[-1,1]^2$ and the associated code:

```
1 x = np.linspace(-1, 1, 100)
2 y = np.linspace(-1, 1, 100)
3 X, Y = np.meshgrid(x, y)
4 Z = np.zeros_like(X)
5
6 for i in range(X.shape[0]):
7     for j in range(X.shape[1]):
8         Z[i, j] = f_square(np.array([X[i, j], Y[i, j]]))
9
10 fig = plt.figure()
11 ax = fig.add_subplot(111, projection='3d')
12 ax.plot_surface(X, Y, Z)
13 ax.set_xlabel('x_1')
14 ax.set_ylabel('x_2')
15 ax.set_title('f_square(x_1, x_2)')
16 plt.show()
```

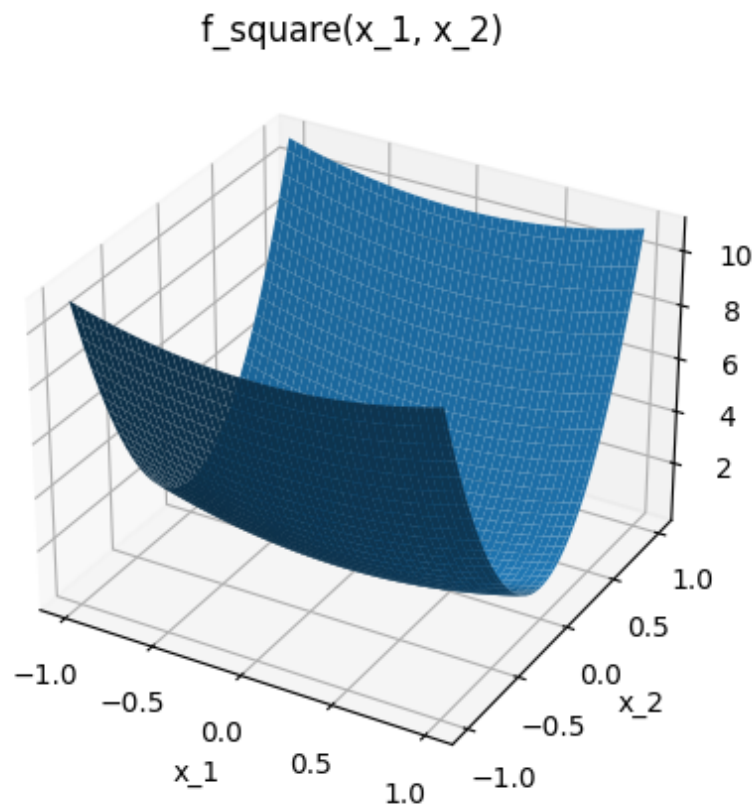


Figure 1:

And here is a plot of f_{hole} on the same interval:

```

1 x = np.linspace(-1, 1, 100)
2 y = np.linspace(-1, 1, 100)
3 X, Y = np.meshgrid(x, y)
4 Z = np.zeros_like(X)
5
6 for i in range(X.shape[0]):
7     for j in range(X.shape[1]):
8         Z[i, j] = f_hole(np.array([X[i, j], Y[i, j]]))
9
10 fig = plt.figure()
11 ax = fig.add_subplot(111, projection='3d')
12 ax.plot_surface(X, Y, Z)
13 ax.set_xlabel('x_1')
14 ax.set_ylabel('x_2')
15 ax.set_title('f_hole(x_1, x_2)')
16 plt.show()

```

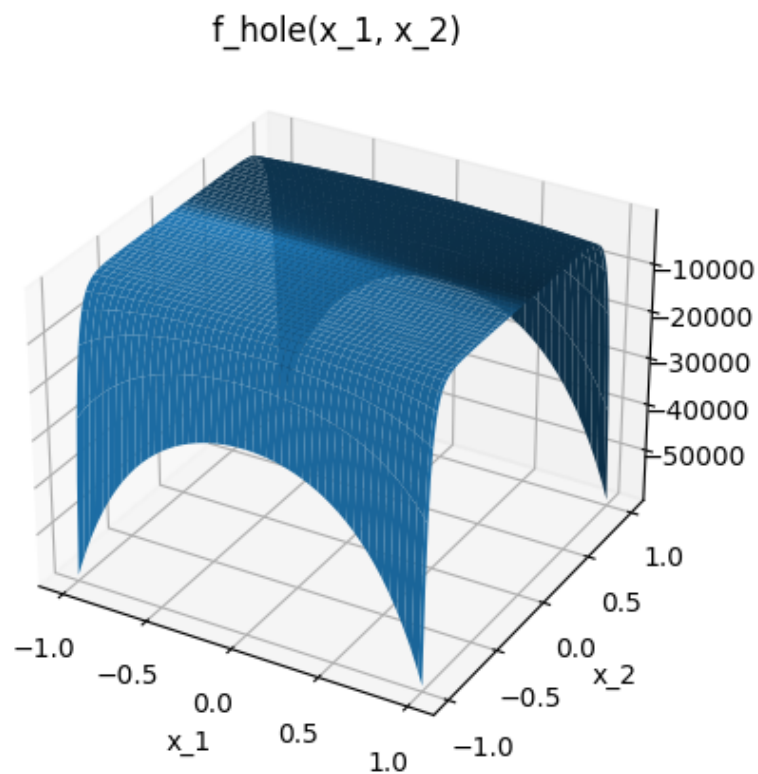


Figure 2:

4.3.D Gradient Descent

$$x_{k+1} = x_k + \alpha \nabla f_{\text{square}}(x_k) \quad (73)$$

Here, I've chosen $\alpha = -0.09$ as this value seemed to be the largest possible step the system could take without diverging or exhibiting unpredictable behaviour. The following is an implementation of Gradient Descent used to optimize f_{square} , along with a plot of $f_{\text{square}}(x_k)$, and the raw outputs of the algorithm:

```

1      # Gradient Descent: fsquare
2
3  def gradient_f_square(x):
4      return 2 * np.matmul(C, x)
5
6  x0 = np.array([1, 1])
7  alpha = -0.09
8  num_iterations = 15
9
10 x_k = x0
11 x_history = []
12 t0 = 0
13 t = []
14 # loop
15 time_start_ov = time.time()
16 for _ in range(num_iterations):
17     time_start_it = time.time()
18     gradient = gradient_f_square(x_k)
19     x_kp1 = x_k + alpha * gradient
20     x_k = x_kp1
21     x_history.append(x_kp1)
22     t.append(t0)
23     t0 += 1
24     time_end_it = time.time()
25
26 time_end_ov = time.time()
27
28
29 time_it = time_end_it - time_start_it
30 time_ov = time_end_ov - time_start_ov
31 print("Time per iteration (15):", time_it)
32 print("Overall time (15):", time_ov)
33
34
35 x = []
36 x_history = np.array(x_history)
37 for i in range(len(x_history)):
38     x.append(f_square(x_history[i]))
39
40 plt.figure()
41 plt.plot(t, x, marker='o')
42 plt.xlabel('iteration')
43 plt.ylabel('f_square(x_1, x_2)')
44 plt.title('f_square Gradient Descent')
45 plt.grid(True)
46 plt.show()

```

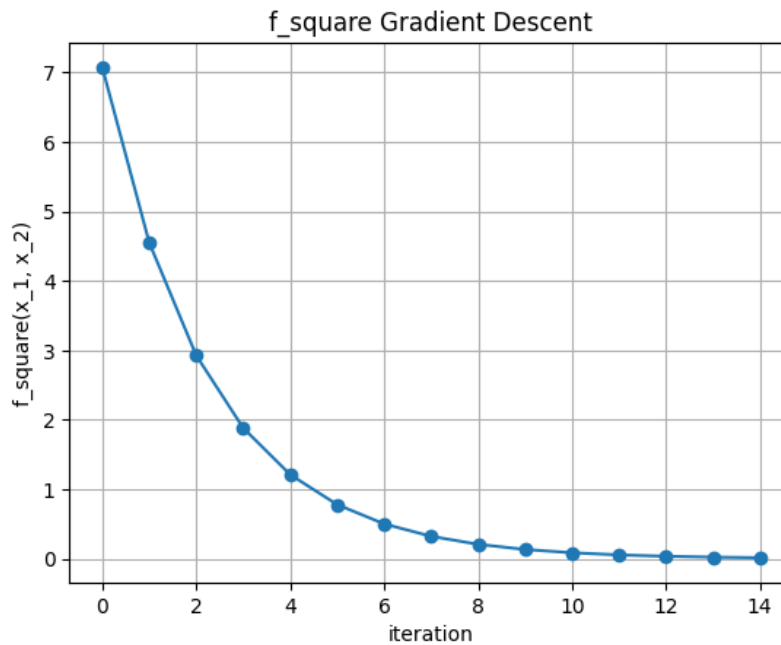


Figure 3: Gradient Descent of f_{square} starting at $(1, 1)$, 15 Iterations

Raw outputs: $\mathbf{x}_k = [x_1, x_2]$ for each iteration k :

```

1 [[ 0.82      -0.8      ]
2 [ 0.6724     0.64     ]
3 [ 0.551368   -0.512    ]
4 [ 0.45212176  0.4096   ]
5 [ 0.37073984 -0.32768  ]
6 [ 0.30400667  0.262144  ]
7 [ 0.24928547 -0.2097152 ]
8 [ 0.20441409  0.16777216]
9 [ 0.16761955 -0.13421773]
10 [ 0.13744803  0.10737418]
11 [ 0.11270739 -0.08589935]
12 [ 0.09242006  0.06871948]
13 [ 0.07578445 -0.05497558]
14 [ 0.06214325  0.04398047]
15 [ 0.05095746 -0.03518437]]

```

The following is an implementation of Gradient Descent used to optimize f_{hole} , along with its outputs:

```

1  # Gradient Descent: fhole
2
3  def gradient_f_hole(x):
4      return -2 * np.matmul(C, x) * np.exp(f_square(x))
5
6  x0 = np.array([1, 1])
7  alpha = 0.01
8  num_iterations = 99
9
10 x_k = x0
11 x_history = []
12 t0 = 0
13 t = []
14 # loop
15 for _ in range(num_iterations):
16     gradient = gradient_f_hole(x_k)
17     x_kp1 = x_k + alpha * gradient
18     x_k = x_kp1
19     x_history.append(x_kp1)
20     t.append(t0)
21     t0 += 1
22
23 x = []
24 x_history = np.array(x_history)
25 for i in range(len(x_history)):
26     x.append(f_hole(x_history[i]))
27
28 print(t)
29 print(x_history)
30 print(x)

```

```

1  [[ -1196.4828343  -11973.82834304]
2   [
3   [
4   [
5   [
6   [
7   [
8   [
9   [
10  [
11  [
12  [
13  [
14  [
15  [
16  [
17  [
18  [

```

the algorithm appears to diverge after only a few iterations. This is because f_{hole} has no minimum, and the value is overflowing as the algorithm is trying to input smaller and smaller values.

4.3.E Newton's Method

Below is an implementation of Newton's Method being used to optimize f_{square} , along with a plot of $f_{\text{square}}(x_k)$ at each iteration:

```

1      # Newton's Method: fsquare
2
3  def hessian_f_square(x):
4      return 2 * C
5
6  x0 = np.array([1, 1])
7  num_iterations = 2
8
9  x_k = x0
10 x_history = [x_k]
11 t = [0]
12 t0 = 0
13
14 # loop
15 time_start_ov = time.time()
16 for _ in range(num_iterations):
17     time_start_it = time.time()
18     gradient = gradient_f_square(x_k)
19     hessian = hessian_f_square(x_k)
20     x_kp1 = x_k - np.matmul(np.linalg.inv(hessian), gradient)
21     x_k = x_kp1
22     x_history.append(x_kp1)
23     t0 += 1
24     t.append(t0)
25     time_end_it = time.time()
26
27 time_end_ov = time.time()
28
29 time_it = time_end_it - time_start_it
30 time_ov = time_end_ov - time_start_ov
31 print("Time per iteration:", time_it)
32 print("Overall time:", time_ov)
33
34 x = []
35 x_history = np.array(x_history)
36 for i in range(len(x_history)):
37     x.append(f_square(x_history[i]))
38
39 plt.figure()
40 plt.plot(t, x, marker='o')
41 plt.xlabel('iteration')
42 plt.ylabel('f_hole(x_1, x_2)')
43 plt.title('f_hole Newton\'s Method')
44 plt.grid(True)
45 plt.show()

```

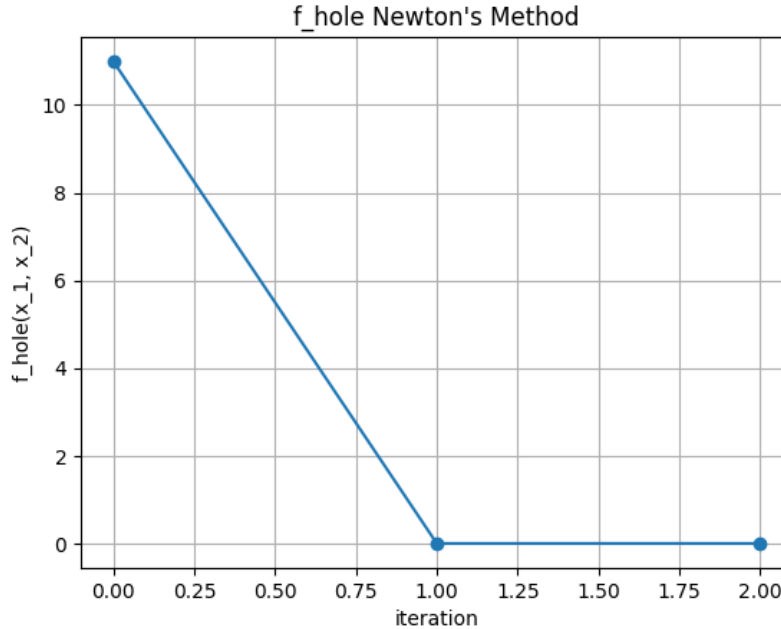


Figure 4: Newton's Method of f_{square} starting at (1, 1), 2 Iterations

We can see the system converges after only one iteration. This is due to the very high convergence rate of Newton's Method. Comparing Newton's Method to Gradient Descent, it took 15 iterations for the Gradient Descent to decently converge, which took 1.07×10^{-5} s per iteration, and 55.2×10^{-5} s overall. On the other hand, to complete 15 iterations, Newton's Method took 2.81×10^{-5} s per iteration and 288×10^{-5} s overall. Alternatively, for the single iteration it took Newton's Method to decently converge, it took 36×10^{-5} s for the one loop iteration, and 77×10^{-5} s overall including overhead. Thus, even though Newton's Method converged in fewer steps, it still took longer than Gradient Descent; however, Newton's Method did achieve higher precision in 2 iterations than Gradient Descent could achieve in 15. We could further compare the two methods by more precisely comparing the precision and accuracy associated with each method.

Below is an implementation of Newton's Method used to optimize f_{hole} , along with a lot of $f_{hole}(x_k)$ against number of iterations:

```

1     # Newton's Method: fhole
2
3 def hessian_f_hole(x):
4     exp_val = np.exp(f_square(x))
5     return -2 * (np.matmul(np.matmul(C, np.identity(2)), C) * exp_val +
6                 2 * np.outer(np.matmul(C, x), np.matmul(C, x)) * exp_val)
7
8 x0 = np.array([1, 1])
9 num_iterations = 10
10
11 x_k = x0
12 x_history = [x_k]
13 t = [0]
14 t0 = 0
15
16 # loop
17 for _ in range(num_iterations):
18     gradient = gradient_f_hole(x_k)
19     hessian = hessian_f_hole(x_k)
20     x_kp1 = x_k - np.matmul(np.linalg.inv(hessian), gradient)
21     x_k = x_kp1
22     x_history.append(x_kp1)
23     t0 += 1
24     t.append(t0)
25
26 x = []
27 x_history = np.array(x_history)
28 for i in range(len(x_history)):
29     x.append(f_hole(x_history[i]))
30
31 plt.figure()
32 plt.plot(t, x, marker='o')
33 plt.xlabel('iteration')
34 plt.ylabel('f_hole(x_1, x_2)')
35 plt.title('f_hole Newton's Method')
36 plt.grid(True)
37 plt.show()

```

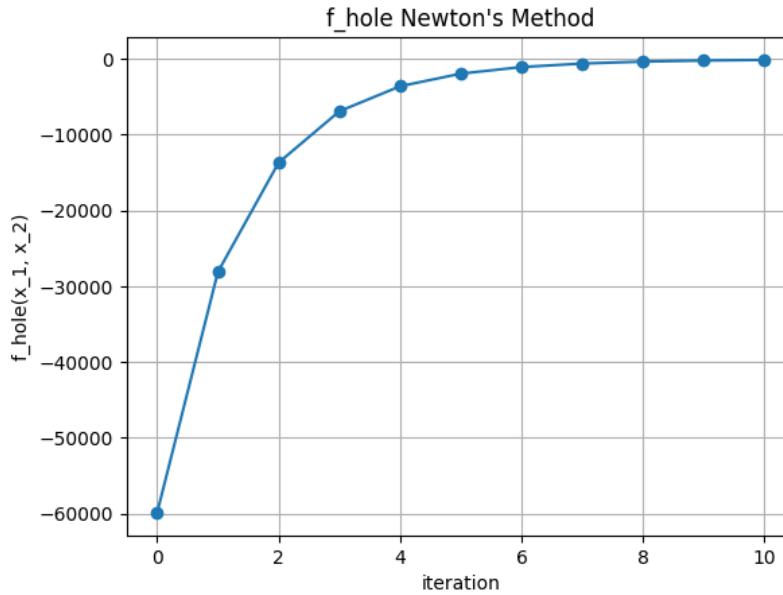



Figure 5: Newton's Method of f_{hole} starting at $(1, 1)$, 10 Iterations

There is not much to compare between the Gradient Descent implementation and the Newton's Method implementation, as the Gradient Descent Method diverged after one iteration and produced no useful solution, while the Newton's Method optimization is converging to 0, as I believe it should, geometrically speaking (looking at the plot in Figure 2). It seems to take at least 10 iteration to converge to a roughly accurate result. The scale in this graph is very large, so simply looking at the graph visually is not enough to determine the precision of this method. With 99 iterations, the value of $f_{\text{hole}}(x_k)$ is $-1.0279200046703352 \times 10^{-7}$, which is extremely close to the global optimum, 0. This is a testament to the precision and fast convergence of Newton's Method.