ANA670 Applied Optimization
Assignment Module Two

First Name	**Randall**
Last Name	**Crawford**
ID#	**901012194**
Email Address	**rcc_0149@yahoo.com**

Table of Contents

How to submit your Assignment

After filling all the parts in this file, please follow the following steps.

1) Add your name and ID to the first page.
2) Save the file in the original format (Docx or Doc)

   (please do not convert to other file formats e.g. PDF, ZIP, RAR, …).

3) Rename the file as

   *YOUR **First** Name - YOUR **Last** Name - ID* – ANA670 – ***HW2**.docx*

   Example: John - Smith - ID - ANA670 – HW2.docx

4) Upload the file and submit it (only using BrightSpace)

P1 [Optional - 10 points extra credit]

Use a penalty function to solve the following problem of Gill-Murray Wright type:

minimize $f(x) = 100(x - b)^2 + \pi$

subject to $g(x) = x - a \geq 0$

where $a > b$ is a given value.

(Not a coding problem, solve symbolically)

Solution

Using the penalty function method, the constraint needs to be incorporated into the objective function by adding a penalty term that penalizes violations of the constraint. For the inequality constraint $g(x) \geq 0$, a penalty term that is active only when $g(x) < 0$ is necessary. A common choice is to use the squared violation multiplied by a penalty parameter $\mu > 0$.

A. Penalized Function Approach

The penalized objective function:

$P(x, \mu) = f(x) + \mu \cdot \max(0, -g(x))^2$

Since $g(x) = x - a$, the penalty term is:

$\max(0, -g(x)) = \max(0, -(x - a)) = \max(0, a - x)$

Thus, the penalized objective function becomes:

$P(x, \mu) = 100(x - b)^2 + \pi + \mu \cdot \max(0, a - x)^2$

B. Interpretation

If $x \geq a$, then $\max(0, a - x) = 0$, and the penalty term is zero, so $P(x, \mu) = f(x) = 100(x - b)^2 + \pi$.

If $x < a$, then $\max(0, a - x) = a - x$, and the penalty term $\mu(a - x)^2$ is added, which increases the objective value to discourage violation of the constraint.

C. Minimization

To minimize $P(x, \mu)$, take the derivative of $P(x, \mu)$ with respect to x. The penalty parameter μ is typically chosen to be large to enforce the constraint, and is iteratively increased to approximate the constrained minimum as $\mu \rightarrow \infty$.

Two cases need to be considered based on the value of x relative to a:

Case 1: $x \geq a$

$P(x, \mu) = 100(x - b)^2 + \pi$

Derivative:

$dP/dx = 200(x - b)$

Set derivative equal to zero to find critical points:

$200(x - b) = 0$
$x = b$

Since $a > b$ and $x = b$ cannot satisfy $x \geq a$, the solution path must reside in Case 2.

Case 2: $x < a$

$P(x, \mu) = 100(x - b)^2 + \pi + \mu(a - x)^2$

Derivative:

$dP/dx = 200(x - b) - 2\mu(a - x)$

Set derivative equal to zero to find critical points:

$200(x - b) - 2\mu(a - x) = 0$
$200x - 200b - 2\mu a + 2\mu x = 0$
$(200 + 2\mu)x = 200b + 2\mu a$
$x = (200b + 2\mu a)/ (200 + 2\mu) = (100b + \mu a)/ (100 + \mu)$

As $\mu \rightarrow \infty$, $x \rightarrow a$ (since the term μa dominates), which aligns with the constraint boundary.

D. Solution

The solution depends on the behavior as μ becomes large:

The minimum of the unconstrained problem $100(x - b)2 + \pi$ occurs at $x = b$.

Since the constraint $x \geq a$ is active (because the unconstrained minimum $x = b$ is less than a), the feasible region is $x \geq a$, and the minimum will occur at the boundary $x = a$ (since f(x) increases as x moves away from b). The penalty method ensures that as $\mu \rightarrow \infty$, the solution approaches the constrained minimum.

For $a > b$, the optimal solution to the constrained problem is $x^* = a$. Substitute $x = a$ into the objective function: $f(x^*) = 100(a - b)^2 + \pi$

This is the minimum value achievable while satisfying the constraint $x \geq a$. The penalty method confirms this by driving x toward a as μ increases, penalizing any $x < a$.

P2 [20 points]

Employ the BFGS method in Python so as to find the global maximum of Easom's benchmark function as written as a function of one variable:

$$f(x) = \cos(x)e^{-(x-\pi)^2}, \qquad x \in [-5, 5].$$

Investigate the effect of starting from different initial solutions, say, on the extrema of the domain: x_0 = -5 and x_0 = 5. Hint: You can write your own or use the BFGS implementation included with the **scipy** package; if you use that, you will need to import **numpy** as well.

Your Code

```
import numpy as np
from scipy.optimize import minimize
import datetime
import sys

# Define Easom's benchmark function as a single variable (minimize -f to find
maximum)
def easom_function(x):
    return -np.cos(x) * np.exp(-((x - np.pi)**2))  # Negate to maximize f(x)

def find_global_maximum(initial_guess):

    try:
        initial = float(initial_guess)
        bounds = [(-5, 5)]  # Enforce domain x ∈ [-5, 5]
        result = minimize(
            easom_function,
            initial,
            method='L-BFGS-B',  # Use L-BFGS-B to handle bounds
            bounds=bounds,
            options={'disp': True}
        )
        optimal_value = -result.fun  # Negate back to get the maximum of f(x)
        optimal_param = result.x[0]
        message = (f"Global maximum found at: f({optimal_param:.6f}) =
{optimal_value:.6f}\n"
                   f"Number of iterations: {result.nit}\n"
                   f"Success: {result.success}")
        return optimal_value, optimal_param, message
    except ValueError as e:
        return None, None, f"Error: Invalid input - {str(e)}"
    except Exception as e:
        return None, None, f"Error: Optimization failed - {str(e)}"
```

```
def main():
    print("Easom's Function Global Maximum Finder with L-BFGS-B")
    print(f"Current date and time: {datetime.datetime.now().strftime('%Y-%m-%d
%H:%M:%S')} CDT")
    print("Domain: x ∈ [-5, 5]")

    # Part 1: Find the global maximum with a reasonable initial guess (e.g., x
= 0)
    print("\nPart 1: Finding the global maximum")
    initial_guess_global = 0
    opt_value, opt_param, message = find_global_maximum(initial_guess_global)
    print("Results for global maximum:")
    print(message)

    # Part 2: Investigate effect of starting from different initial solutions
    print("\nPart 2: Investigating effect of solutions including extrema,
recognizing the global minimum equals pi.")
    initial_guesses = [-5, 0, 1.3, 4, 5]
    results = {}

    for x0 in initial_guesses:
        print(f"\nTesting with initial guess x0 = {x0}")
        opt_value, opt_param, message = find_global_maximum(x0)
        results[x0] = (opt_value, opt_param, message)
        print("Optimization Results:")
        print(message)

    print("\nSummary of Results for Initial Solutions:")
    for x0 in initial_guesses:
        val, param, _ = results[x0]
        print(f"Initial guess x0 = {x0}: Maximum at x = {param:.6f}, f(x) =
{val:.6f}")

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("\nProgram terminated by user")
        sys.exit(0)
```

Used Wolfram Alpha to view a graph for this function, and determined that the global minimum resided at x = π, f(x) = -1. Values for x were chosen based on that graph and minimum. The global maximum (0.009006) was common to many x values forming more than one basin between the extrema -5 and 5.

Run the code and insert the result in the following box.

Results

```
Easom's Function Global Maximum Finder with L-BFGS-B
Current date and time: 2025-06-11 20:59:44 CDT
Domain: x ∈ [-5, 5]

Part 1: Finding the global maximum.
Results for global maximum:
Global maximum found at: f(1.304989) = 0.009006
Number of iterations: 3
Success: True

Part 2: Investigating solutions including extrema, recognizing the global minimum is at x = π.

Testing with initial guess x0 = -5
Optimization Results:
Global maximum found at: f(-5.000000) = 0.000000
Number of iterations: 0
Success: True

Testing with initial guess x0 = 0
Optimization Results:
Global maximum found at: f(1.304989) = 0.009006
Number of iterations: 3
Success: True

Testing with initial guess x0 = 1.3
Optimization Results:
Global maximum found at: f(1.304994) = 0.009006
Number of iterations: 3
Success: True

Testing with initial guess x0 = 4
Optimization Results:
Global maximum found at: f(4.978189) = 0.009006
Number of iterations: 6
Success: True

Testing with initial guess x0 = 5
Optimization Results:
Global maximum found at: f(4.978190) = 0.009006
Number of iterations: 4
Success: True

Summary of results for investigated solutions:
Initial guess x0 = -5: Maximum at x = -5.000000, f(x) = 0.000000
Initial guess x0 = 0: Maximum at x = 1.304989, f(x) = 0.009006
Initial guess x0 = 1.3: Maximum at x = 1.304994, f(x) = 0.009006
Initial guess x0 = 4: Maximum at x = 4.978189, f(x) = 0.009006
Initial guess x0 = 5: Maximum at x = 4.978190, f(x) = 0.009006
```

P3 [30 points]

- Update your code from the previous assignment dealing with the gaussian blob, as follows. Read the new input file (two_clusters.txt) attached to the assignment)
- Initial conditions: start with the two centroids:
  - (x=1, y=10)
  - (x=4, y=14)
- Compute the value of the objective function (which is the sum of the squared Euclidean distances of the each point associated within a cluster to its associated cluster center)

Your Code

```
import numpy as np

# Step 1: Load the data from two_clusters.txt
data = np.genfromtxt('two_clusters.txt', delimiter=',')  # Load the file as a
2D array.

# Extract x and y columns.
x = data[:, 0]  # First column is x.
y = data[:, 1]  # Second column is y.

# Step 2: Compute the required statistics.
# 1. Maximum in the x direction
max_x = np.max(x)

# 2. Minimum in the x direction
min_x = np.min(x)

# 3. Maximum in the y direction
max_y = np.max(y)

# 4. Minimum in the y direction
min_y = np.min(y)

# 5. Sum of the squared Euclidean distances from each required point.
# Squared Euclidean distance formula: (x_i - x)^2 + (y_i - y)^2

#5a. (x = 1, y = 10)
a_distances = ((x - 1) ** 2) + ((y - 10) ** 2)
sum_a_distances = np.sum(a_distances)

#5b. (x = 4, y = 14)
b_distances = ((x - 4) ** 2) + ((y - 14) ** 2)
sum_b_distances = np.sum(b_distances)
```

```
# Step 3: Print the results.
print(f"1. Maximum in the x direction: {max_x:,.3f}")
print(f"2. Minimum in the x direction: {min_x:,.3f}")
print(f"3. Maximum in the y direction: {max_y:,.3f}")
print(f"4. Minimum in the y direction: {min_y:,.3f}")
print(f"5a. Sum of the squared Euclidean distances to (1, 10):
{sum_a_distances:,.3f}")
print(f"5b. Sum of the squared Euclidean distances to (4, 14):
{sum_b_distances:,.3f}")
```

Run the code and insert the result in the following box.

Results

1. Maximum in the x direction: 4.846
2. Minimum in the x direction: -0.069
3. Maximum in the y direction: 14.300
4. Minimum in the y direction: 10.163
5a. Sum of the squared Euclidean distances to (1, 10): 7,956.347
5b. Sum of the squared Euclidean distances to (4, 14): 7,314.530

P4 [45 points]

Consider $f(x_1, x_2) = x_1^2 - x_1 x_2 + 3x_2^2 + 5$ (I suggest plotting it with Wolfram Alpha to get a sense of what it looks like) and let the initial point for optimization be $x = (2, 2)$. Set up the initial values and perform 3 iterations of Steepest Descent including step size α updates. An "iteration" is computation of the loop body of the loop on **line 7 of Algorithm 1** in the attached handout *"Notes on the Steepest Descent Method"*

This is not a coding exercise; just show each step, with each arithmetic operation the algorithm takes in 3 iterations, annotated with explanations in English demonstrating your understanding and so I can follow what you are doing.

Solution

Let us minimize the function:

$f(x_1, x_2) = x_1^2 - x_1 x_2 + 3x_2^2 + 5$, where $(x_1, x_2) \in [-2, 2] \times [-2, 2]$.

Using the steepest descent method, we will start from an initial point $x^{(0)} = (2, 2)^T$.

We know that the gradient is

$$\nabla f = (\partial f/\partial x_1, \partial f/\partial x_2) = (2x_1 - x_2, -x_1 + 6x_2)^T,$$

therefore $\nabla f(x^{(0)}) = (2, 10)^T$. In the first iteration, we have $x^{(1)} = x^{(0)} - \alpha_0 \begin{vmatrix} 2 \\ 10 \end{vmatrix}$

The step size α_0 should be chosen such that $f(x^{(1)})$ is at the minimum, which means that

$$f(\alpha_0) = (2 - 2\alpha_0)^2 - (2 - 2\alpha_0)(2 - 10\alpha_0) + 3(2 - 10\alpha_0)^2 + 5$$

should be minimized. This becomes an optimization problem for a single independent variable α_0. All the techniques for univariate optimization problems such as Newton's method can be used to find α_0. We can also obtain the solution by setting the derivative equal to zero.

Used the power rule, product rule, and chain rule standard differentiation rules:

$$df/d\alpha_0 = (-8 + 8\alpha_0) + (24 - 40\alpha_0) + (-120 + 600\alpha_0) + 0 = 568\alpha_0 - 104$$

Set the derivative equal to zero:

$$568\alpha_0 - 104 = 0$$
$$\alpha_0 = 104/568 \approx 0.1831$$

$$x^{(1)} = x^{(0)} - \alpha_0 \begin{vmatrix} 2 \\ 10 \end{vmatrix}$$

$$x^{(1)} = (2, 2)^T - 0.1831(2, 10)^T \approx (1.6338, 0.169)^T$$

At the second iteration, we have

$$x^{(1)} \approx (1.6338, 0.169)^T$$

$$\nabla f(x^{(1)}) = (3.0986, -0.6198)^T,\ x^{(2)} = x^{(1)} - \alpha_1 \begin{vmatrix} 3.0986 \\ -0.6198 \end{vmatrix}$$

The step size α_1 should be chosen such that $f(x^{(2)})$ is at the minimum, which means that

$$f(\alpha_1) = (1.6338 - 3.0986\alpha_1)^2 - (1.6338 - 3.0986\alpha_1)(0.169 + 0.6198\alpha_1) + 3(0.169 + 0.6198\alpha_1)^2 + 5$$

Find the derivative:

$$df/d\alpha_1 = (19.2076\alpha_1 - 10.1297) + (1.9212\alpha_1 + 1.4323) + (2.3051\alpha_1 + 0.6285) + 0 = 23.4339\alpha_1 - 8.0689$$

Set the derivative equal to zero:

$$23.4339\alpha_1 - 8.0689 = 0$$
$$\alpha_1 = 8.0689/23.4339 \approx 0.3443$$

$$x^{(2)} = x^{(1)} - \alpha_1 \begin{vmatrix} 3.0986 \\ -0.6198 \end{vmatrix}$$

$$x^{(2)} = (1.6338, 0.169)^T - 0.3443(3.0986, -0.6198)^T \approx (0.5669, 0.3824)^T$$

At the third iteration, we have

$$x^{(2)} \approx (0.5669, 0.3824)^T$$

$\nabla f(x^{(2)}) = (0.7514, 1.7275)^T$, $x^{(3)} = x^{(2)} - \alpha_2 \begin{vmatrix} 0.7514 \\ 1.7275 \end{vmatrix}$

The step size α_2 should be chosen such that $f(x^{(3)})$ is at the minimum, which means that

$f(\alpha_2) = (0.5669 - 0.7514\alpha_2)^2 - (0.5669 - 0.7514\alpha_2)(0.3824 - 1.7275\alpha_2) + 3(0.3824 - 1.7275\alpha_2)^2 + 5$

Find the derivative:

$df/d\alpha_2 = (1.1293\alpha_2 - 0.8522) + (-1.2976\alpha_2 - 0.0311) + (17.9046\alpha_2 - 3.9645) + 0 = 17.7363\alpha_2 - 4.8478$

Set the derivative equal to zero:

$17.7363\alpha_2 - 4.8478 = 0$
$\alpha_2 = 4.8478/17.7363 \approx 0.2733$

$x^{(3)} = x^{(2)} - \alpha_2 \begin{vmatrix} 0.7514 \\ 1.7275 \end{vmatrix}$

$x^{(3)} = (0.5669, 0.3824)^T - 0.2733(0.7514, 1.7275)^T \approx (0.3615, -0.0897)^T$

Iterations will need to continue until a prescribed tolerance is met or the first partial derivatives are equal to zero.

$2x_1 - x_2 = 0$ and $-x_1 + 6x_2 = 0$

Algebraically looking at those equations, it appears that only $(0, 0)^T$ would potentially satisfy them both.

P5 [15 points]

Plot some level sets of the following three functions:

- $f_1(x_1, x_2) = (x_1 - x_2)^2$
- $f_2(x_1, x_2) = x_1^2 - 3x_2^2$
- $f_3(x_1, x_2, x_3) = x_1^2 + 5x_2^2 + 3x_3^2$

for the range of $x_i \in [-10, 10]$ for all i. You only need to plot the level sets of the function roughly taking integer 13 values, i.e. $f(x) = 1, -1, 0$ etc. Small numerical errors are allowed, we only care about the shapes of the curves. (OK to use **contourplot** in Wolfram Alpha for this. Level set interactive demonstration here under "level curve" section.)

Your Code

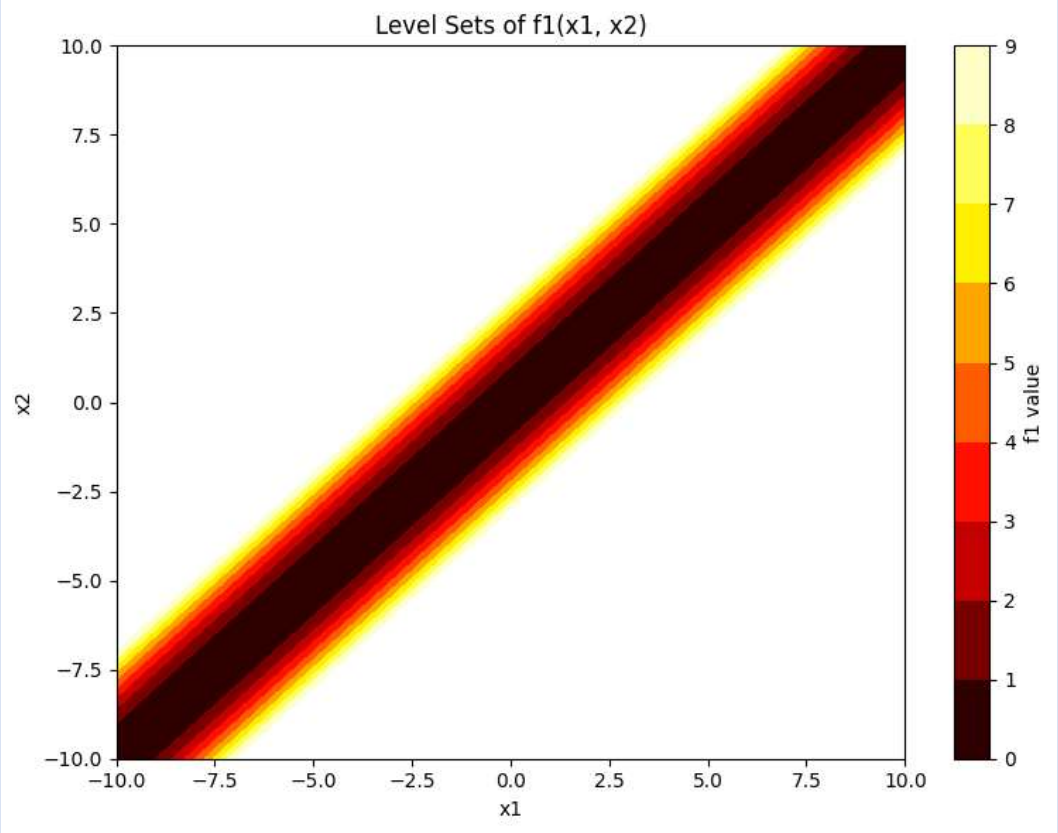
```
import numpy as np
import matplotlib.pyplot as plt

# Create a grid
x1 = np.linspace(-10, 10, 100)
x2 = np.linspace(-10, 10, 100)
X1, X2 = np.meshgrid(x1, x2)

# Function 1: f1(x1, x2) = (x1 - x2)^2
Z1 = (X1 - X2)**2

# Function 2: f2(x1, x2) = x1^2 - 3*x2^2
Z2 = X1**2 - 3 * X2**2

# Plot for f1
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z1, levels=np.arange(0, 10, 1), cmap='hot')
plt.title('Level Sets of f1(x1, x2)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.colorbar(label='f1 value')
plt.tight_layout()
plt.show()

# Plot for f2
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z2, levels=np.arange(-10, 5, 1), cmap='plasma')
plt.title('Level Sets of f2(x1, x2)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.colorbar(label='f2 value')
plt.tight_layout()
plt.show()

# List of x3 values from -10 to 10, selecting 13 evenly spaced values
x3_values = np.linspace(-10, 10, 13)

# Loop through x3 values
for x3 in x3_values:
    # Calculate Z3 for the current x3 using Function 3: f3(x1, x2, x3)
    Z3 = X1**2 + 5 * X2**2 + 3 * x3**2
```

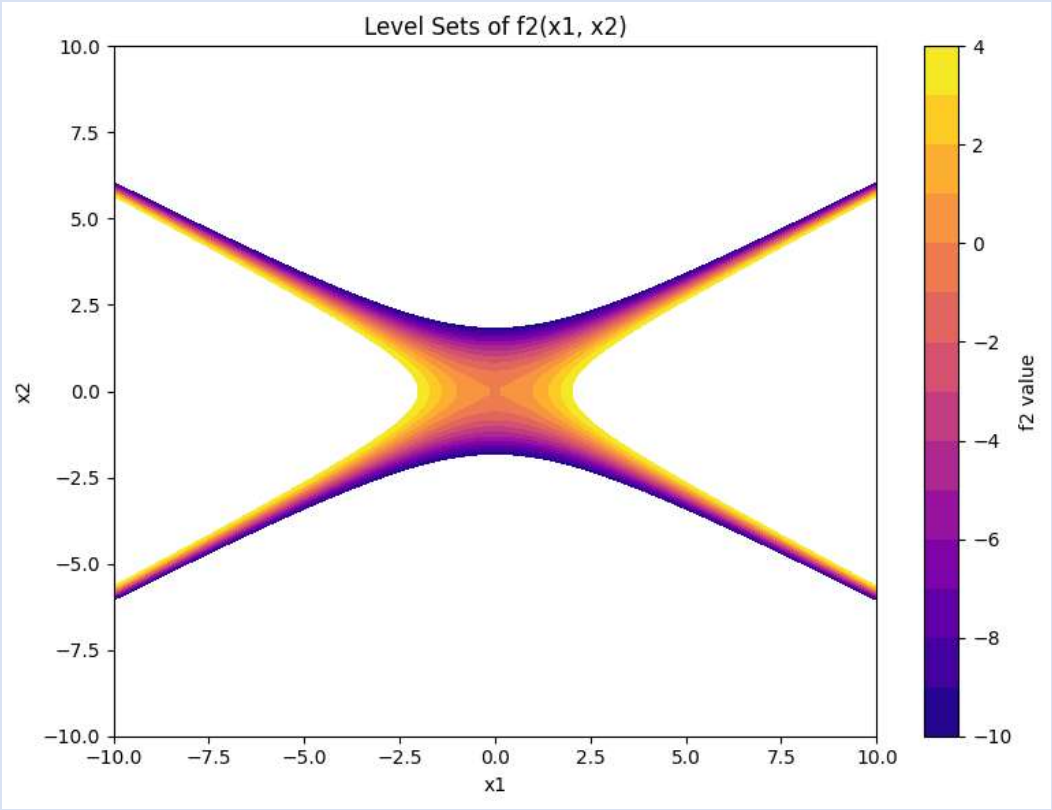
```
    # Create a single contour plot
    plt.figure(figsize=(8, 6))
    cf = plt.contourf(X1, X2, Z3, levels=np.linspace(0, 900, 13), cmap='magma')
    plt.contour(X1, X2, Z3, levels=np.linspace(0, 900, 13), colors='black',
linewidths=0.5)

    # Labels and title
    plt.title(f'Level Sets of f3(x1, x2, x3) with x3 at {x3:.2f}')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.colorbar(cf, label='f3 value')
    plt.tight_layout()
    plt.show()
```

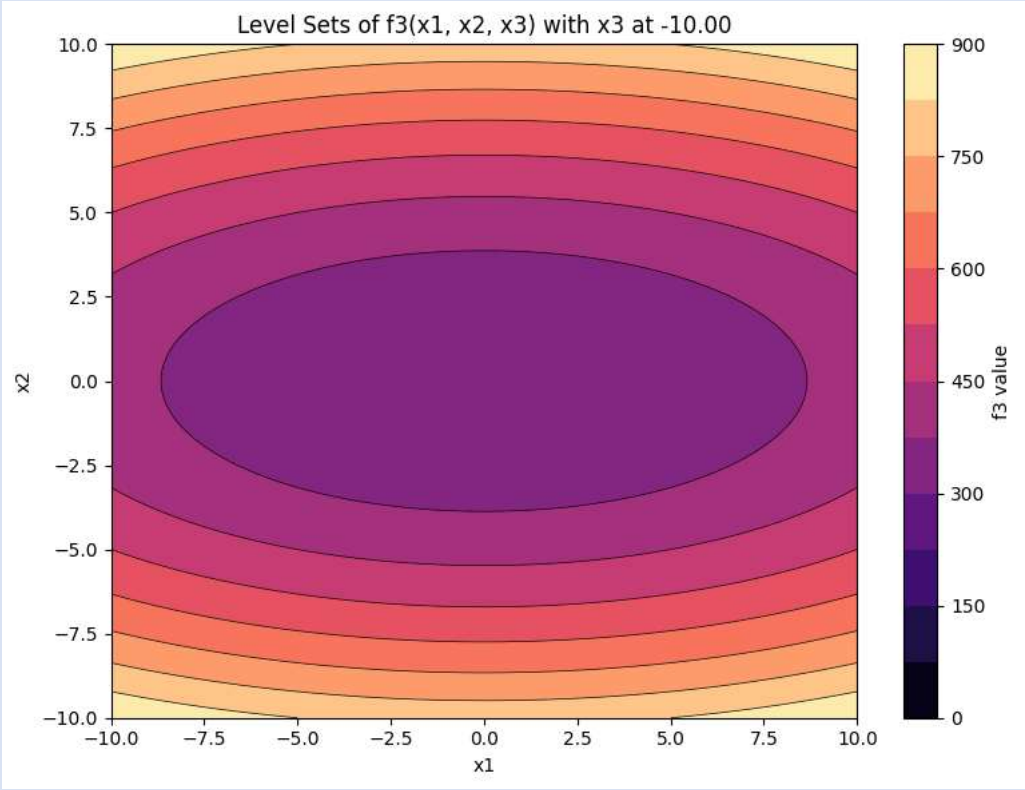
Used $x_i \in [-10, 10]$ and Python/Wolfram Alpha 3D plots to set specific levels for each function. Used relevant integers for f1 and f2 levels, but used 13 evenly spaced floats for the f3 levels.

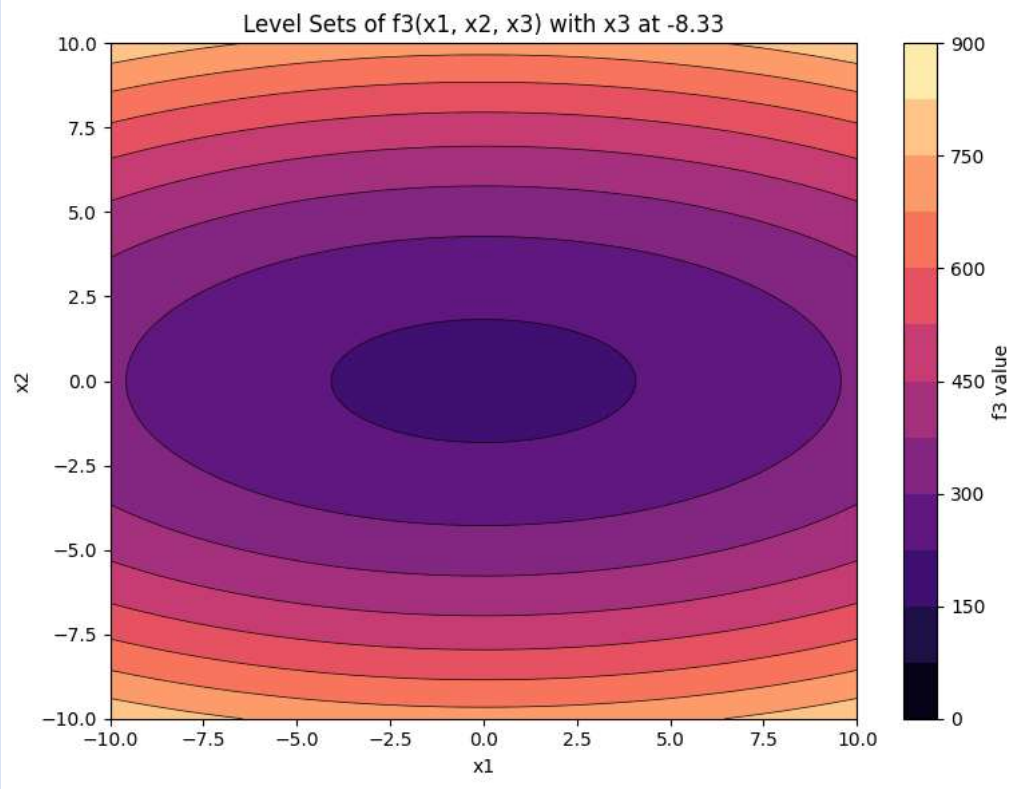
Insert an image of your plot here.

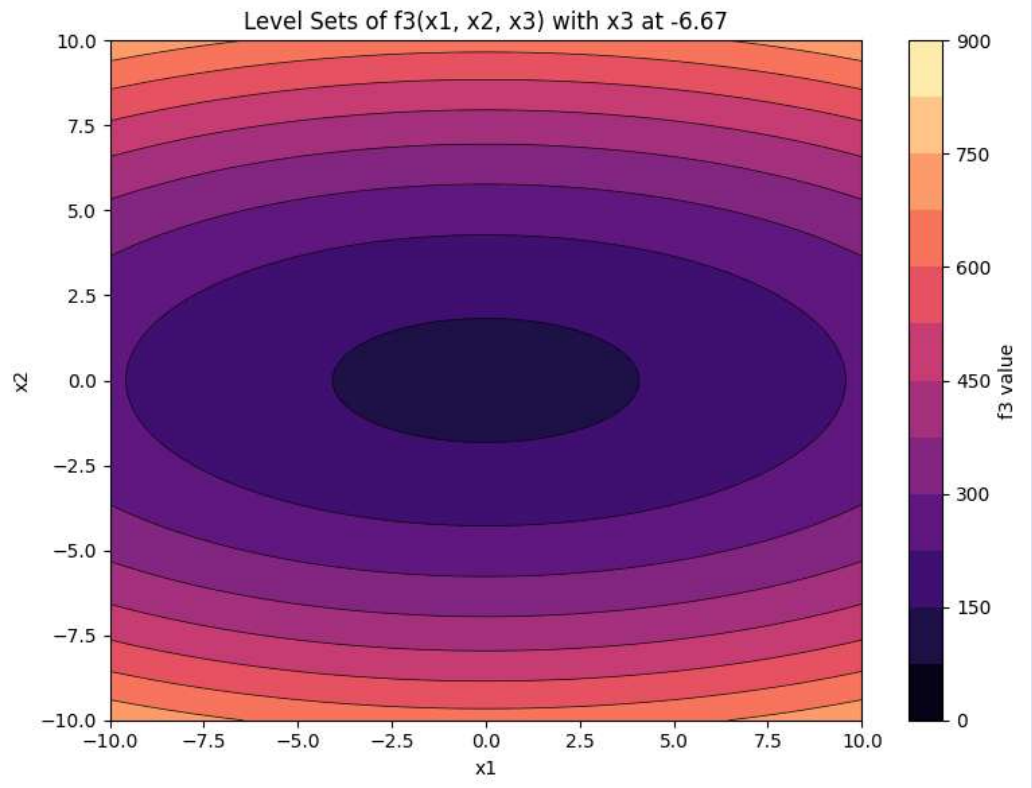
Plots

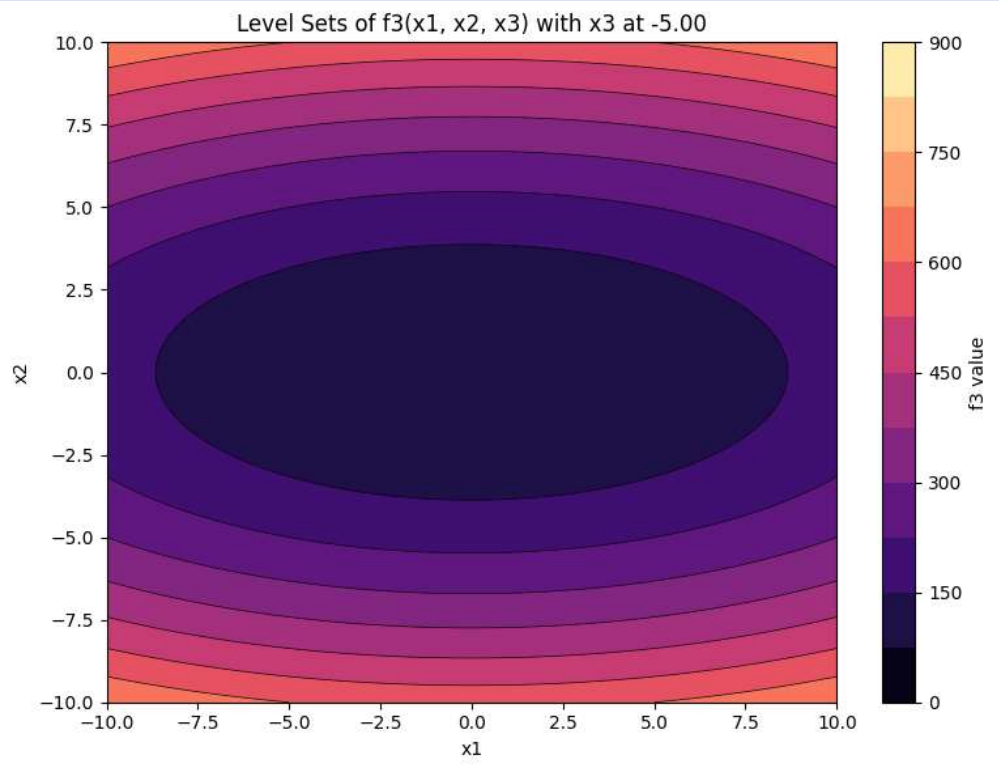


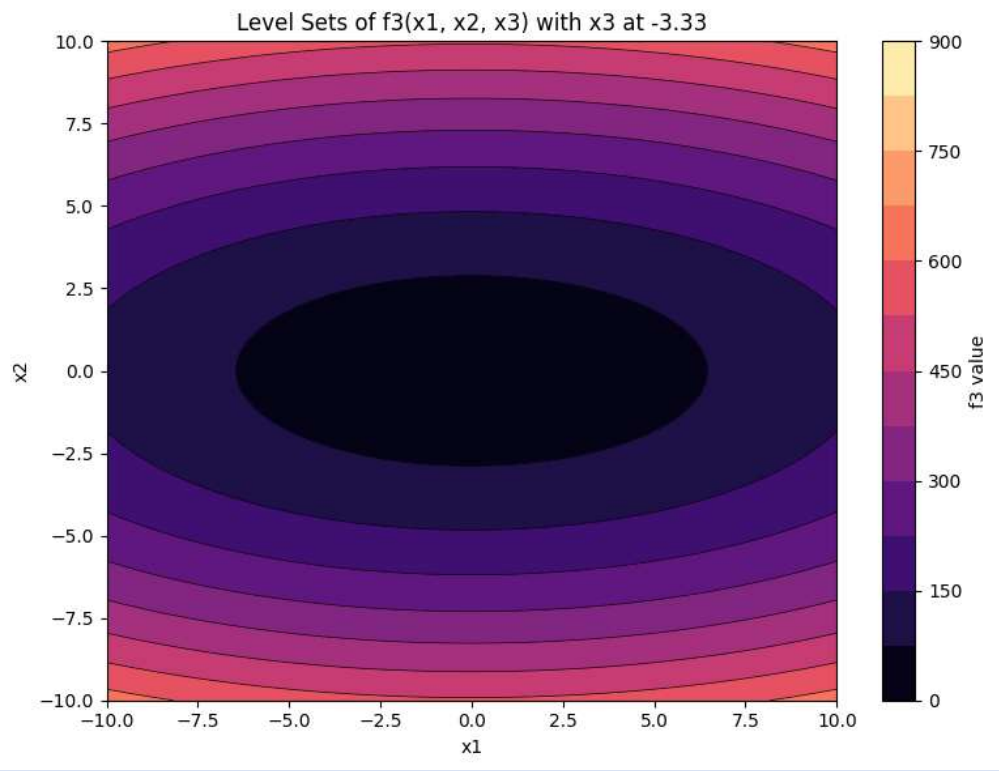

Level Sets of f2(x1, x2)
x1
x2
f2 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
4
2
0
−2
−4
−6
−8
−10

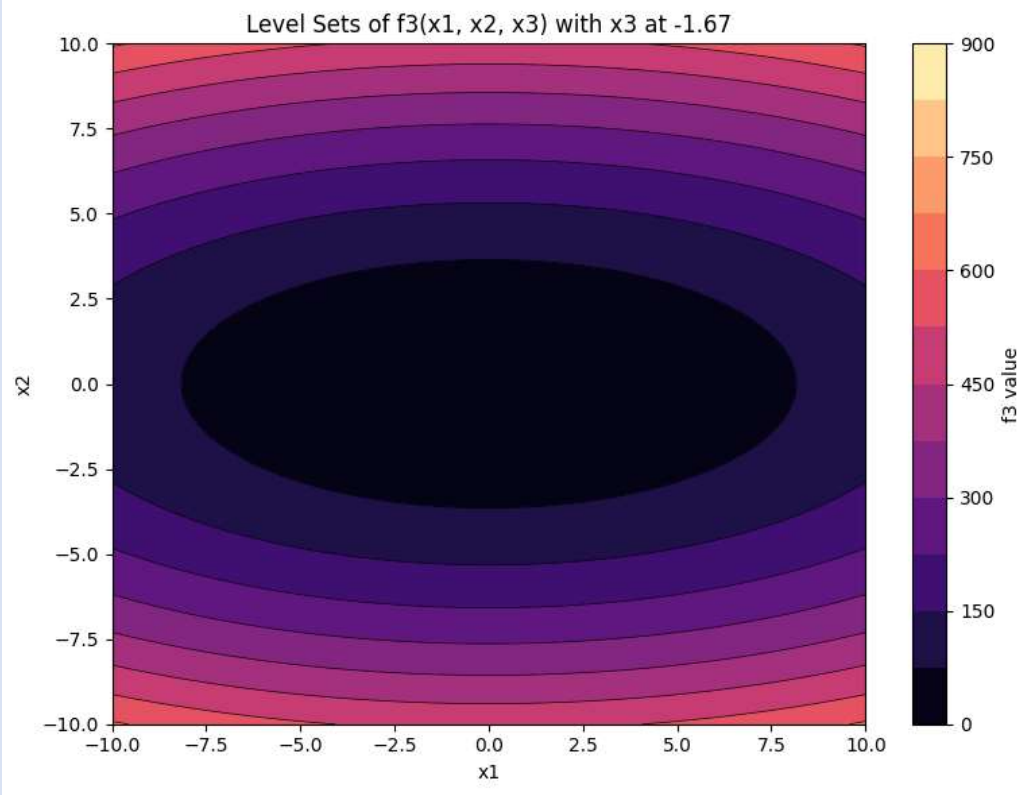

Level Sets of f3(x1, x2, x3) with x3 at -10.00
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

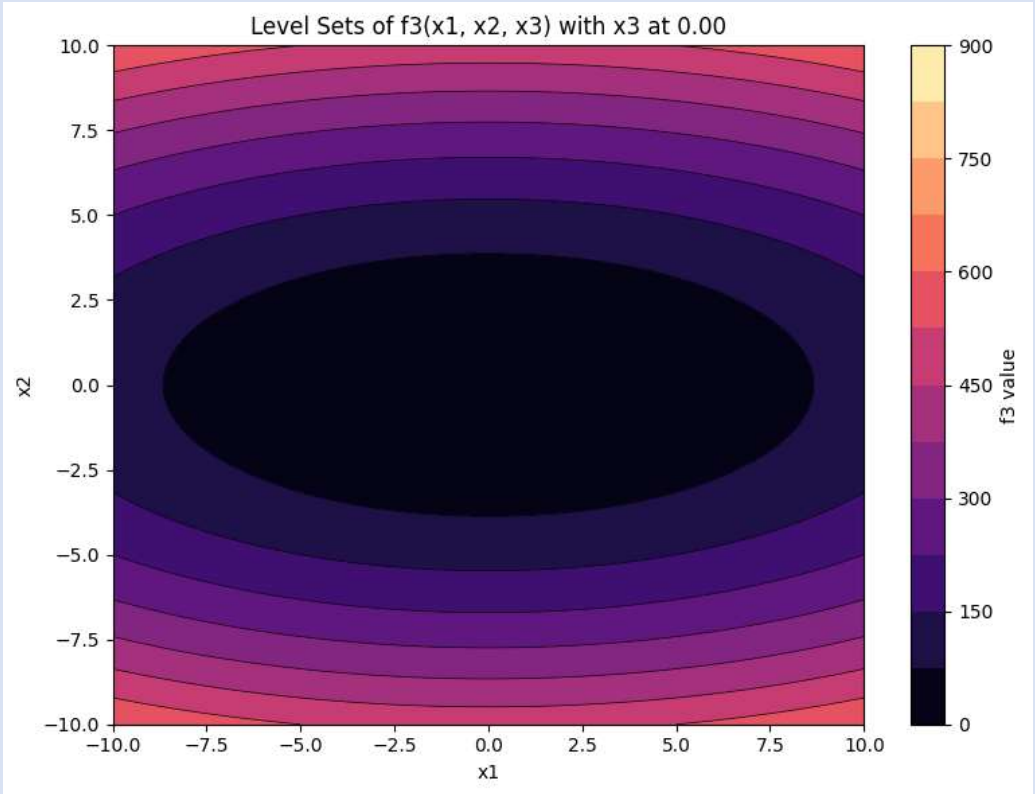

Level Sets of f3(x1, x2, x3) with x3 at -8.33
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

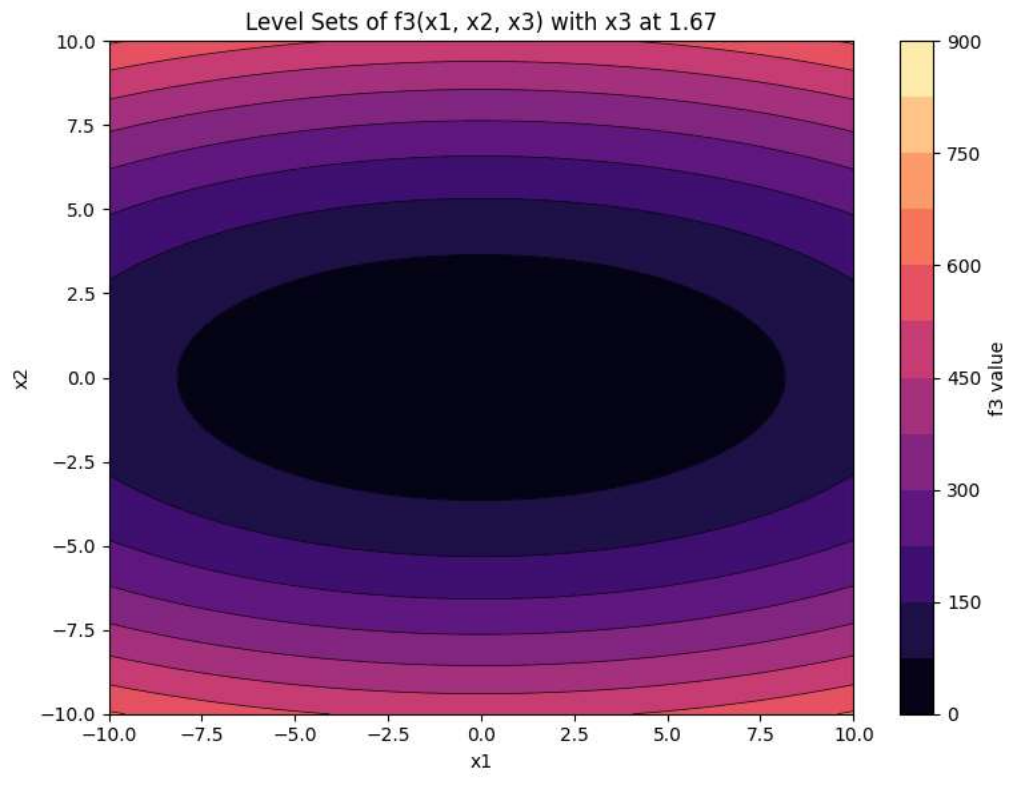

Level Sets of f3(x1, x2, x3) with x3 at -6.67
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

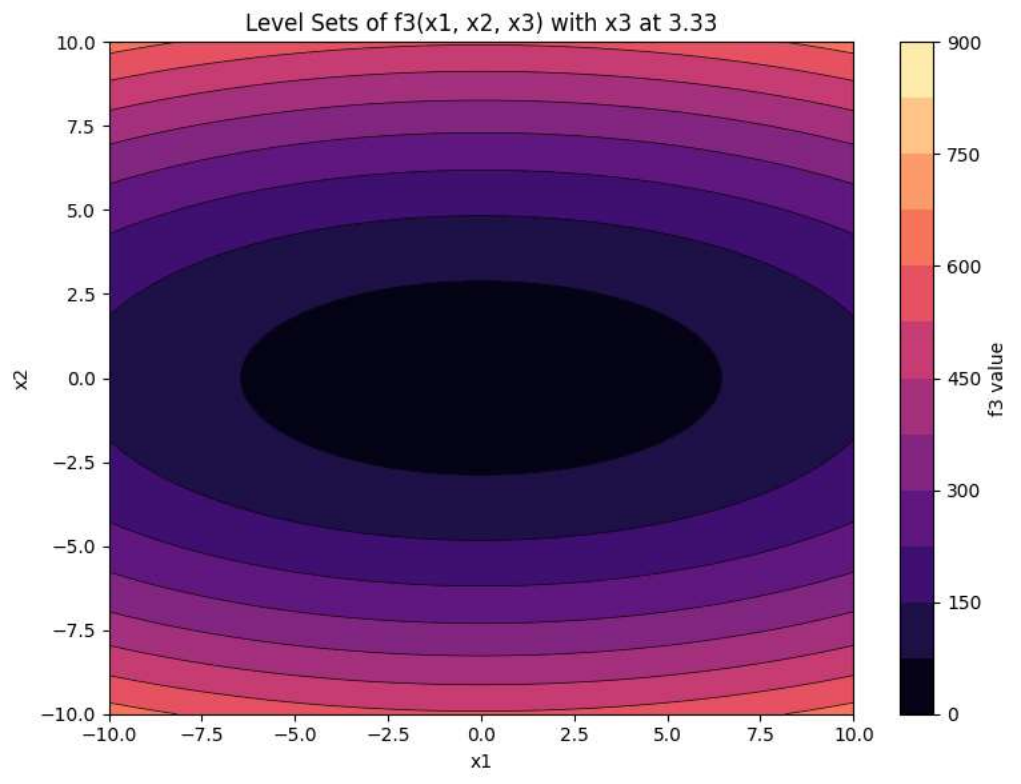

Level Sets of f3(x1, x2, x3) with x3 at -5.00
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

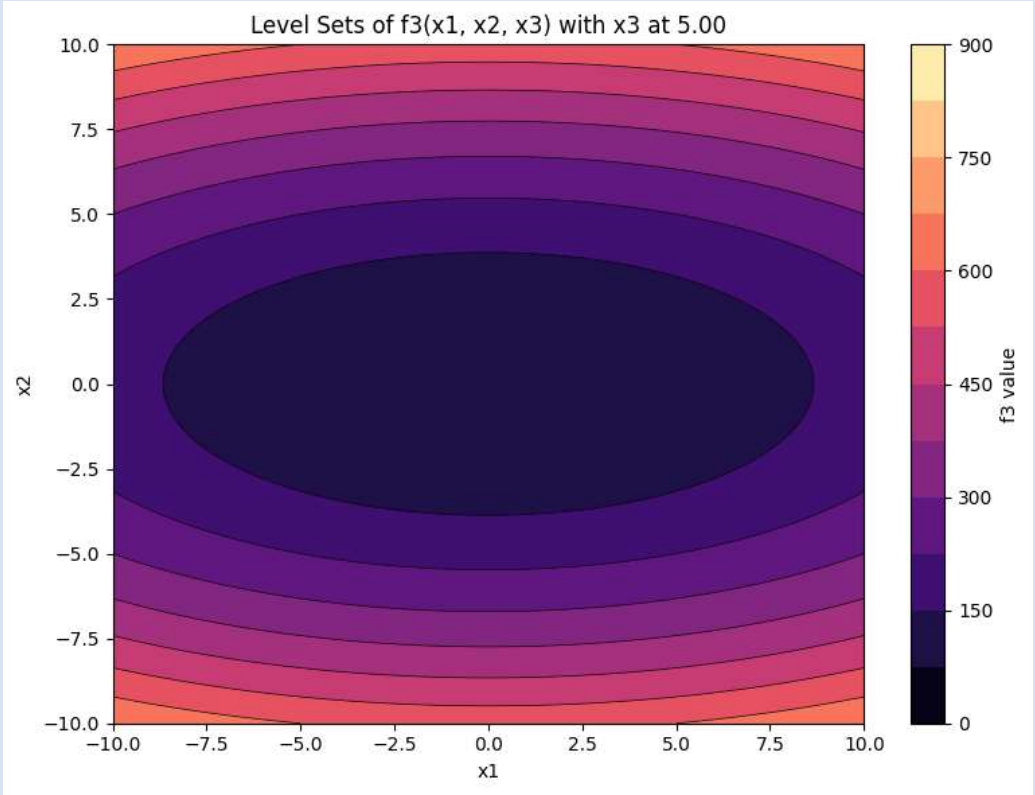

Level Sets of f3(x1, x2, x3) with x3 at -3.33
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

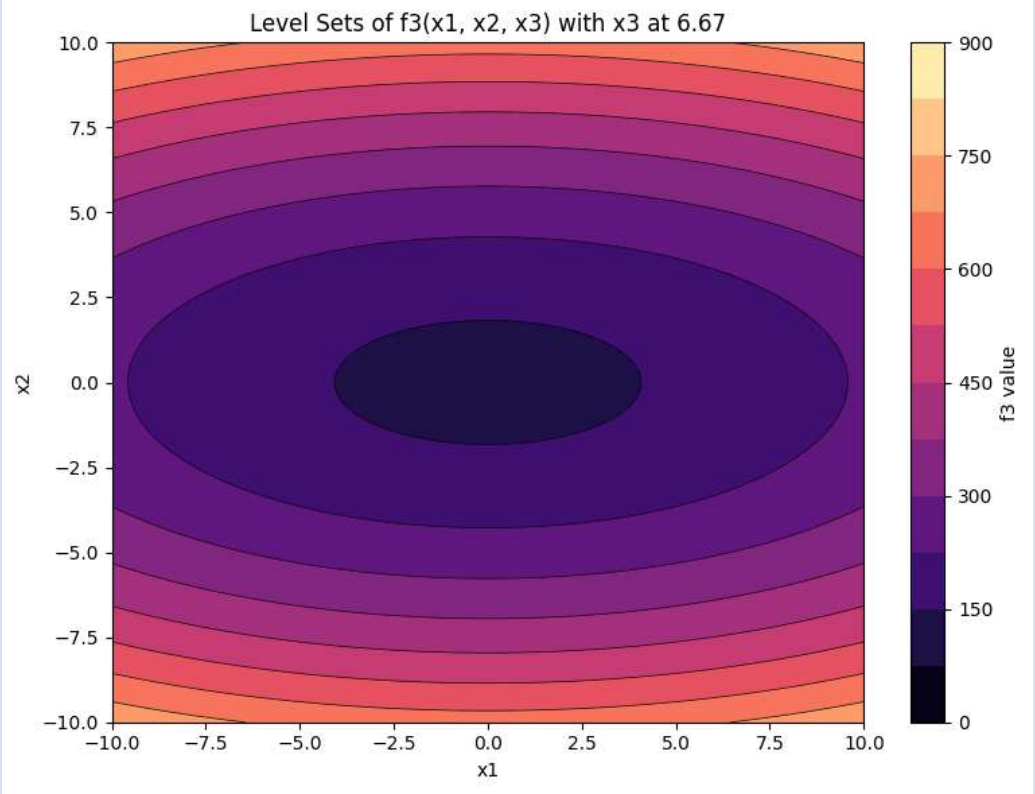

Level Sets of f3(x1, x2, x3) with x3 at -1.67
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

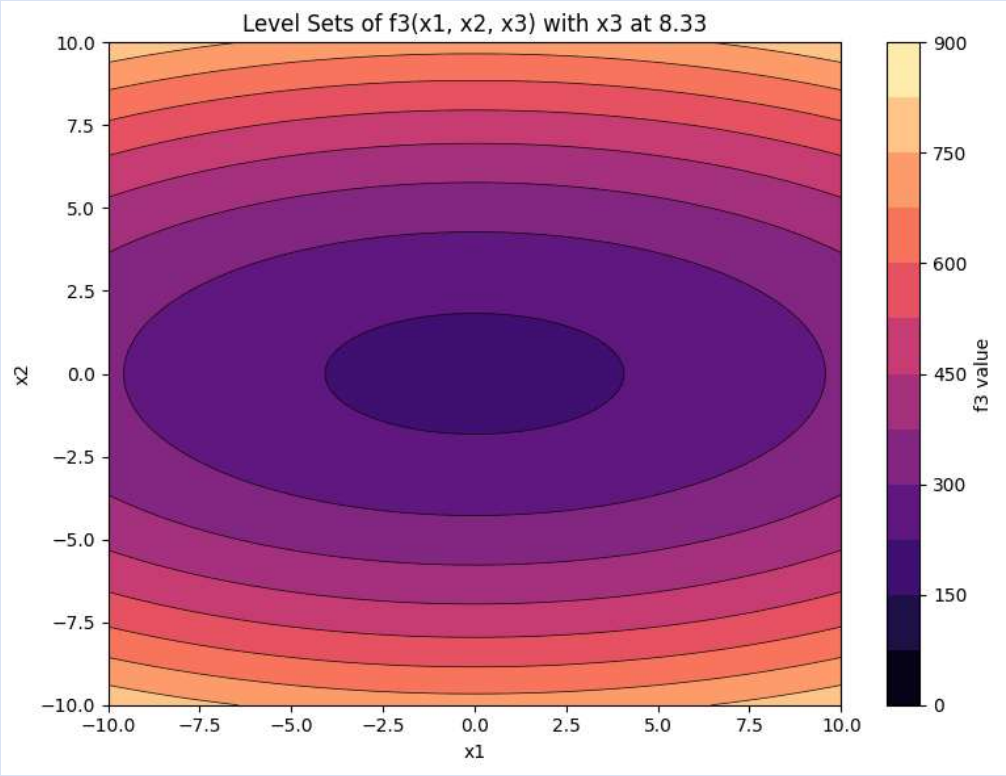

Level Sets of f3(x1, x2, x3) with x3 at 0.00
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

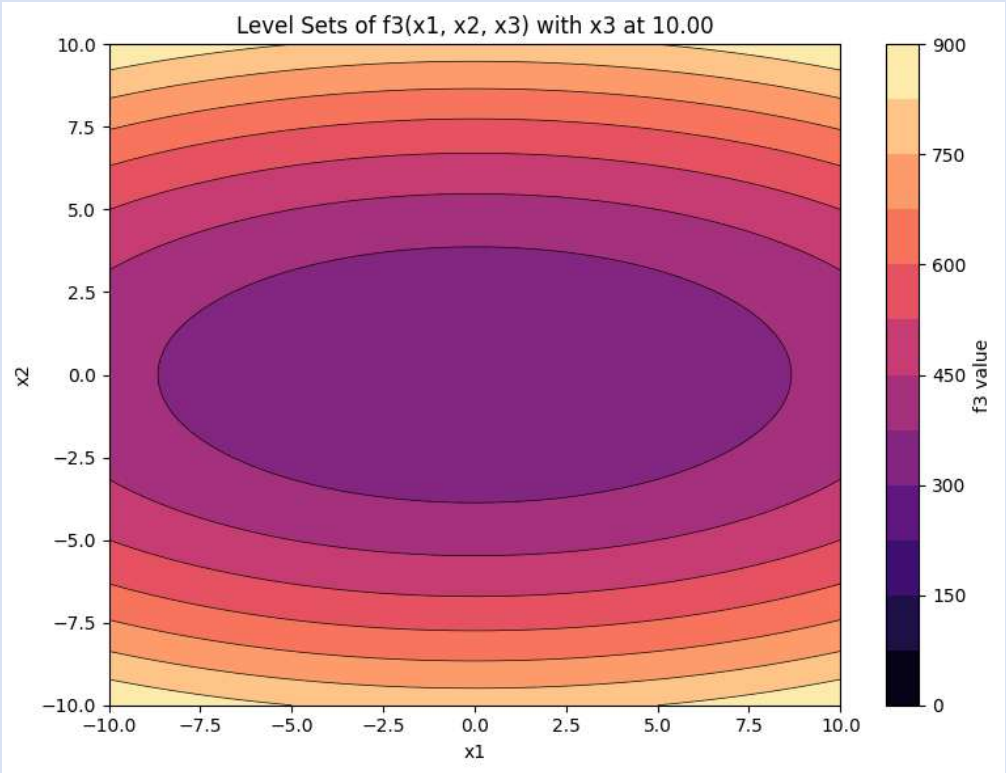

Level Sets of f3(x1, x2, x3) with x3 at 1.67
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0


Level Sets of f3(x1, x2, x3) with x3 at 3.33
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0


Level Sets of f3(x1, x2, x3) with x3 at 5.00
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0


Level Sets of f3(x1, x2, x3) with x3 at 6.67
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0


Level Sets of f3(x1, x2, x3) with x3 at 8.33
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0


Level Sets of f3(x1, x2, x3) with x3 at 10.00
x1
x2
f3 value
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
900
750
600
450
300
150
0

P6 [15 points]

Show that $f(x_i) = \sum_{i=1}^{n} ix_i^2$

for any *n* is convex. This is not a coding problem; this can be clearly established with mathematical reasoning (Hint: what about the second derivative in each dimension?)

Solution

Step 1: Understand Convexity

A function f: $\mathbb{R}^n \rightarrow \mathbb{R}$ is convex if its Hessian matrix is positive semi-definite. Since f is a sum of terms each depending on a single variable x_i, we need to analyze the second derivatives with respect to each x_i.

Step 2: Compute the First and Second Derivatives

The function is $f(x_i) = \sum_{i=1}^{n} ix_i^2$, where x = ($x_1$, x_2, …, x_n).

First partial derivative with respect to x_i:

$$\partial f/\partial x_i = \partial/\partial x_i(ix_i^2 + \sum_{j \neq i} jx_j^2) = 2ix_i$$

The derivative of ix_i^2 with respect to x_i is $2ix_i$, and the terms involving x_j for $j \neq i$ are constant with respect to x_i.

Second partial derivative with respect to x_i:

$$\partial^2 f/\partial x_i^2 = \partial/\partial x_i(2ix_i) = 2i$$

For $i \neq j$, the mixed partial derivative is:

$$\partial^2 f/\partial x_i \partial x_j = \partial/\partial x_j(2ix_i) = 0$$

, because x_i does not depend on x_j.

Step 3: Analyze the Hessian Matrix

The Hessian matrix H of f is:

$$H = \begin{vmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 & \dots & \partial^2 f/\partial x_1 \partial x_n \\ \partial^2 f/\partial x_2 \partial x_1 & \partial^2 f/\partial x_2^2 & \dots & \partial^2 f/\partial x_2 \partial x_n \\ \vdots & \vdots & \vdots & \vdots \\ \partial^2 f/\partial x_n \partial x_1 & \partial^2 f/\partial x_n \partial x_2 & \dots & \partial^2 f/\partial x_n^2 \end{vmatrix} = \begin{vmatrix} 2 & 0 & \dots & 0 \\ 0 & 4 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 2n \end{vmatrix}$$

This is a diagonal matrix where the i-th diagonal element is 2i. The eigenvalues of this matrix are the diagonal elements 2, 4, …, 2n , all of which are positive since $i \geq 1$. A diagonal matrix with positive diagonal entries is positive definite.

Step 4: Conclusion

Since the Hessian matrix is positive definite (all eigenvalues are positive), the function $f(x_i) = \sum_{i=1}^{n} ix_i^2$ is convex for any n, because the second derivative 2i in each dimension is positive for all $i \geq 1$.