AE6310: Optimization for the Design of Engineered Systems

Assignment 1

Hao Chen

Problem 1. Compute the standard form of the following quadratic functions. Sketch them and find any minimizers.

a) $f(x_1, x_2) = {x_1}^2 + x_1 x_2 + {x_2}^2 - x_1 - x_2$

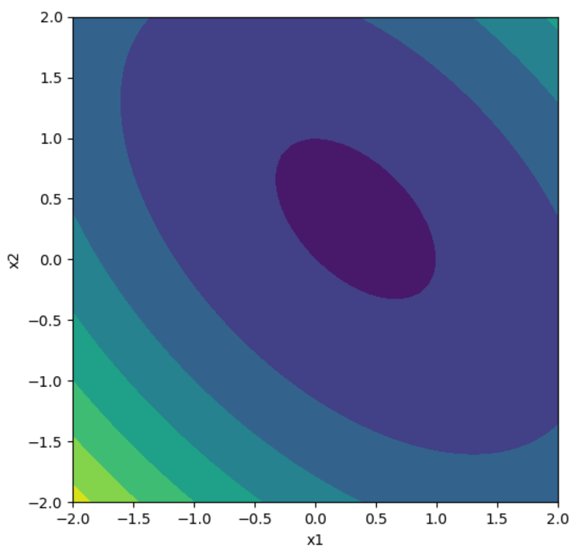
We know that the standard form is

$$f(x) = \frac{1}{2} x^T A x + x^T b + c$$

So, the standard form of this equation is

$$\boxed{f(x) = \frac{1}{2}[x_1 \quad x_2]\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [x_1 \quad x_2]\begin{bmatrix} -1 \\ -1 \end{bmatrix}}$$

Sketch



To find the minimizer, we have

$$\frac{\partial f}{\partial x_1} = 2x_1 + x_2 - 1 \text{ and } \frac{\partial f}{\partial x_2} = x_1 + 2x_2 - 1$$

The gradient $\nabla f = \begin{bmatrix} 2x_1 + x_2 - 1 \\ x_1 + 2x_2 - 1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, we get $x^* = \begin{bmatrix} 1/3 \\ 1/3 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ is also positive definite because $det|2| > 0$ and $det\begin{vmatrix} 2 & 1 \\ 1 & 2 \end{vmatrix} > 0$. Thus, the minimizer is

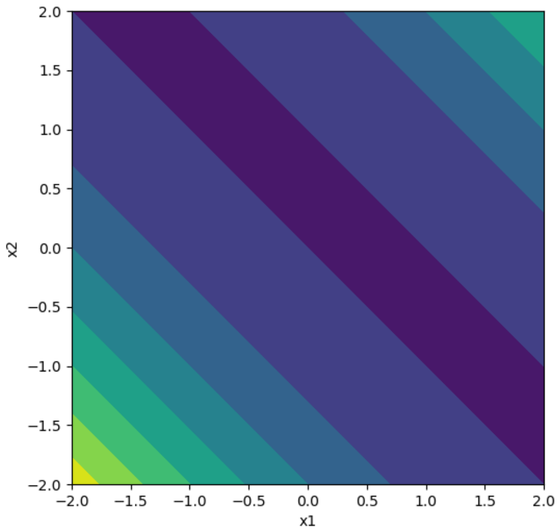
$$\boxed{x^* = \begin{bmatrix} 1/3 \\ 1/3 \end{bmatrix}}$$

b) $f(x_1, x_2) = {x_1}^2 + 2x_1x_2 + {x_2}^2 - x_1 - x_2$

The standard form of this equation is

$$\boxed{f(x) = \frac{1}{2}[x_1 \quad x_2]\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [x_1 \quad x_2]\begin{bmatrix} -1 \\ -1 \end{bmatrix}}$$

Sketch



To find the minimizer, we have

$$\frac{\partial f}{\partial x_1} = 2x_1 + 2x_2 - 1 \text{ and } \frac{\partial f}{\partial x_2} = 2x_1 + 2x_2 - 1$$

The gradient $\nabla f = \begin{bmatrix} 2x_1 + 2x_2 - 1 \\ 2x_1 + 2x_2 - 1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, we get $x^* = \begin{bmatrix} 1/4 \\ 1/4 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$ is a positive semi-definite with eigenvalues $\lambda_1 = 4$ and $\lambda_2 = 0$.

We know the eigenvector for $\lambda_2 = 0$ is $v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $b = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$, thus $b^T v_2 = 0$.

So, we know there are $\boxed{\textbf{infinite}}$ minimizers along the line

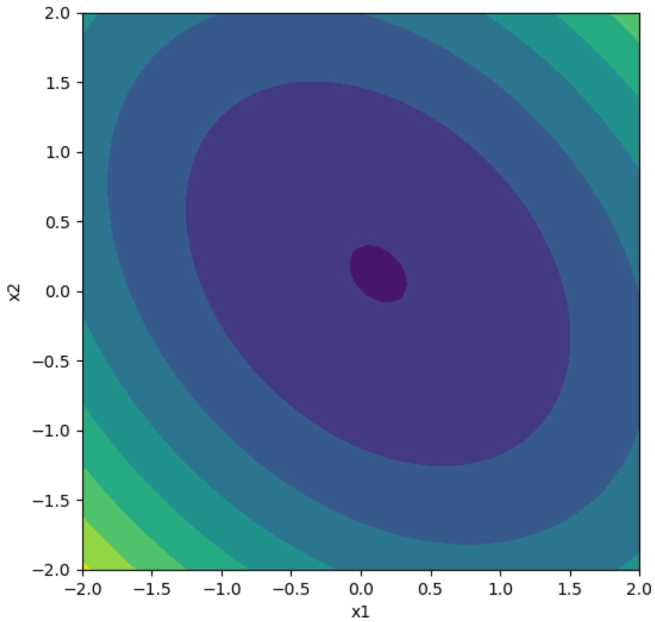
$$\boxed{x_1^* + x_2^* = \frac{1}{2}}$$

c) $f(x_1, x_2) = 3{x_1}^2 + 2x_1x_2 + 3{x_2}^2 - x_1 - x_2$

The standard form of this equation is

$$\boxed{f(x) = \frac{1}{2}[x_1 \quad x_2]\begin{bmatrix} 6 & 2 \\ 2 & 6 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [x_1 \quad x_2]\begin{bmatrix} -1 \\ -1 \end{bmatrix}}$$

Sketch



To find the minimizer, we have

$$\frac{\partial f}{\partial x_1} = 6x_1 + 2x_2 - 1 \text{ and } \frac{\partial f}{\partial x_2} = 2x_1 + 6x_2 - 1$$

The gradient $\nabla f = \begin{bmatrix} 6x_1 + 2x_2 - 1 \\ 2x_1 + 6x_2 - 1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 6 & 2 \\ 2 & 6 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, we get $x^* = \begin{bmatrix} 1/8 \\ 1/8 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 6 & 2 \\ 2 & 6 \end{bmatrix}$ is also positive definite because $det|6| > 0$ and $det\begin{vmatrix} 6 & 2 \\ 2 & 6 \end{vmatrix} > 0$. Thus, the minimizer is

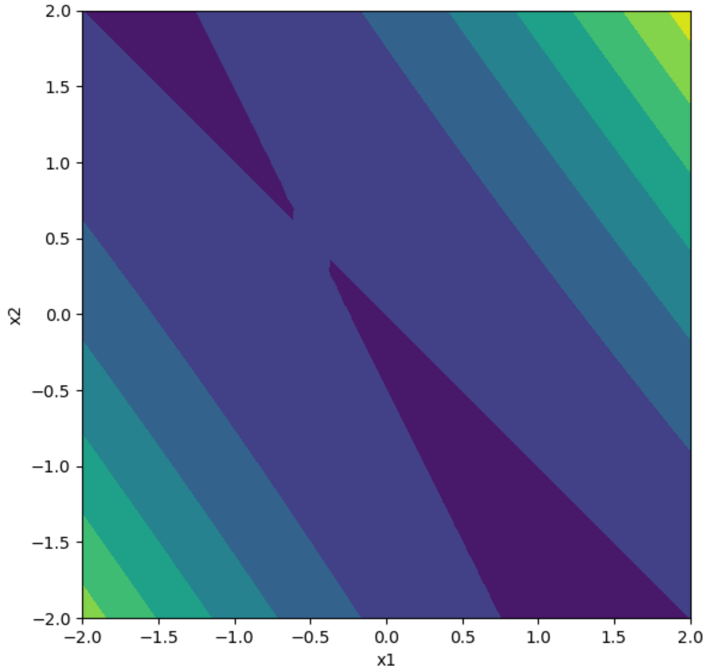
$$\boxed{x^* = \begin{bmatrix} 1/8 \\ 1/8 \end{bmatrix}}$$

d) $f(x_1, x_2) = 4{x_1}^2 + 6x_1x_2 + 2{x_2}^2 + x_1 + x_2$

The standard form of this equation is

$$\boxed{f(x) = \frac{1}{2}\begin{bmatrix} x_1 & x_2 \end{bmatrix}\begin{bmatrix} 8 & 6 \\ 6 & 4 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} x_1 & x_2 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}}$$

Sketch



To find the minimizer, we have

$$\frac{\partial f}{\partial x_1} = 8x_1 + 6x_2 + 1 \text{ and } \frac{\partial f}{\partial x_2} = 6x_1 + 4x_2 + 1$$

The gradient $\nabla f = \begin{bmatrix} 8x_1 + 6x_2 + 1 \\ 6x_1 + 4x_2 + 1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 8 & 6 \\ 6 & 4 \end{bmatrix}$

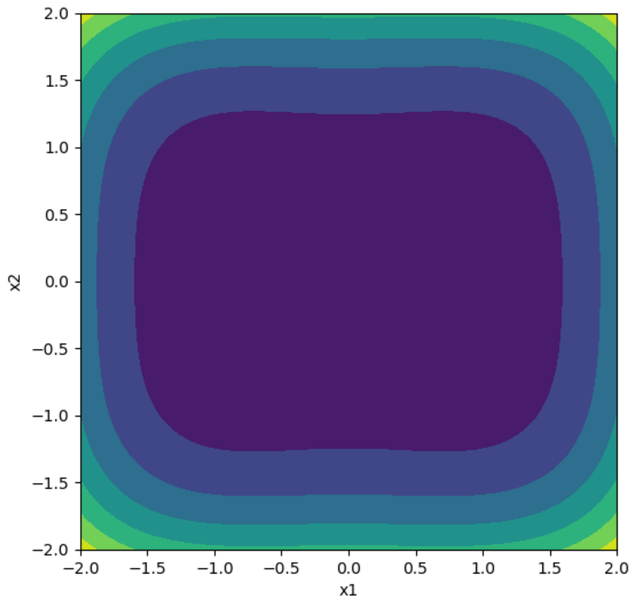
Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, we get $x^* = \begin{bmatrix} -1/2 \\ 1/2 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 8 & 6 \\ 6 & 4 \end{bmatrix}$ is indefinite with eigenvalues $\lambda_1 = 2(3 + \sqrt{10}) > 0$ and $\lambda_2 = 2(3 - \sqrt{10}) < 0$.

So, there is $\boxed{\textbf{no minimizer}}$. But there is a saddle point at $\begin{bmatrix} -1/2 \\ 1/2 \end{bmatrix}$ as shown in the figure above.

Problem 2. Plot the following functions and find and characterize their critical points. Justify your answers.

a) $f(x_1, x_2) = {x_1}^4 + {x_2}^4 + 1 - {x_1}^2 + {x_2}^2$

plot



To find the critical points, we have

$$\frac{\partial f}{\partial x_1} = 4{x_1}^3 - 2x_1 \text{ and } \frac{\partial f}{\partial x_2} = 4{x_2}^3 + 2x_2$$

The gradient $\nabla f = \begin{bmatrix} 4{x_1}^3 - 2x_1 \\ 4{x_2}^3 + 2x_2 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 12{x_1}^2 - 2 & 0 \\ 0 & 12{x_2}^2 + 2 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and we only consider the scenario that both x_1 and x_2 are real numbers. We get $\boldsymbol{x}^* = \begin{bmatrix} 0, \frac{1}{\sqrt{2}}, or - \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$.

$\boxed{\textbf{When } \boldsymbol{x}^* = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}}$,

The Hessian $H(x) = \begin{bmatrix} -2 & 0 \\ 0 & 2 \end{bmatrix}$, so it is indefinite, this critical point is a $\boxed{\textbf{saddle point}}$.

$\boxed{\textbf{When } \boldsymbol{x}^* = \begin{bmatrix} \frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \\ \mathbf{0} \end{bmatrix}}$,

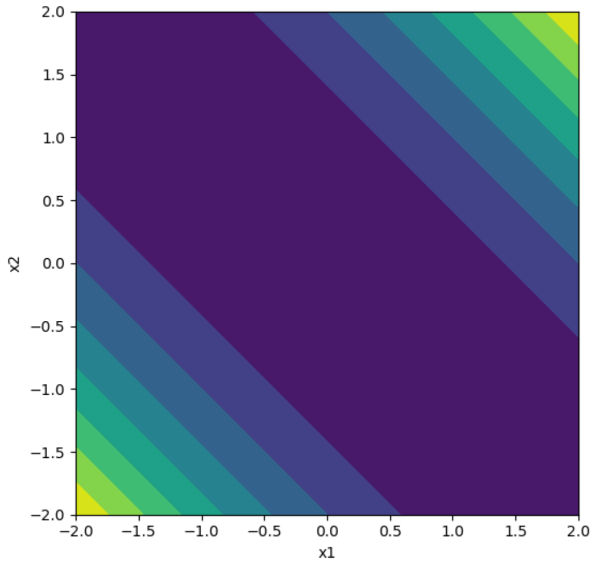
The Hessian $H(x) = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix}$, so it is positive definite, this critical point is a $\boxed{\textbf{minimizer}}$.

$\boxed{\textbf{When } \boldsymbol{x}^* = \begin{bmatrix} -\frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \\ \mathbf{0} \end{bmatrix}}$,

The Hessian $H(x) = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix}$, so it is positive definite, this critical point is a $\boxed{\textbf{minimizer}}$.

b) $f(x_1, x_2) = {x_1}^2 + {x_2}^2 + 2x_1x_2$

plot



To find the critical points, we have

$$\frac{\partial f}{\partial x_1} = 2x_1 + 2x_2 \text{ and } \frac{\partial f}{\partial x_2} = 2x_2 + 2x_1$$

The gradient $\nabla f = \begin{bmatrix} 2x_1 + 2x_2 \\ 2x_2 + 2x_1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, we get $x_1^* + x_2^* = 0$. We know the Hessian $H(x) = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$ is positive semi-definite with eigenvalues $\lambda_1 = 4$ and $\lambda_2 = 0$. The eigenvector for $\lambda_2 = 0$ is $v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, thus $b^T v_2 = 0$.

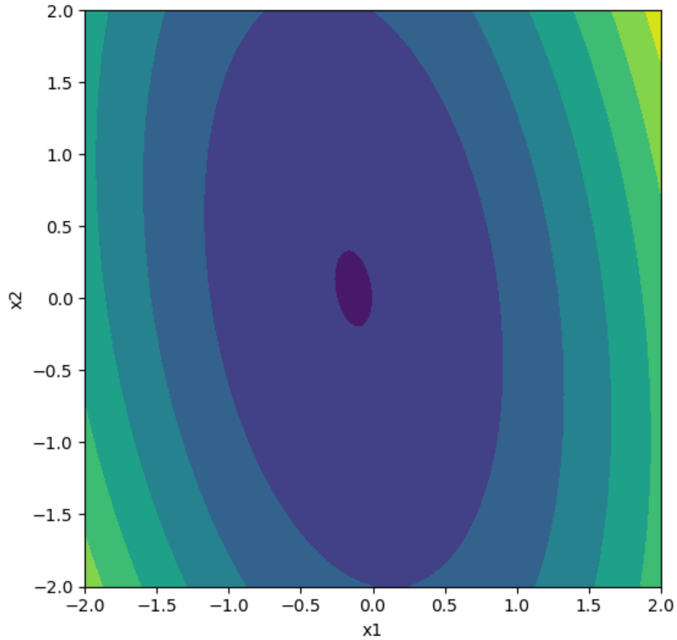
So, we know there are infinite critical points along the line

$$\boxed{\boldsymbol{x_1^* + x_2^* = 0}}$$

They are all $\boxed{\textbf{minimizers}}$.

c) $f(x_1, x_2) = 4{x_1}^2 + {x_2}^2 + x_1x_2 + x_1$

plot



To find the critical points, we have

$$\frac{\partial f}{\partial x_1} = 8x_1 + x_2 + 1 \quad \text{and} \quad \frac{\partial f}{\partial x_2} = 2x_2 + x_1$$

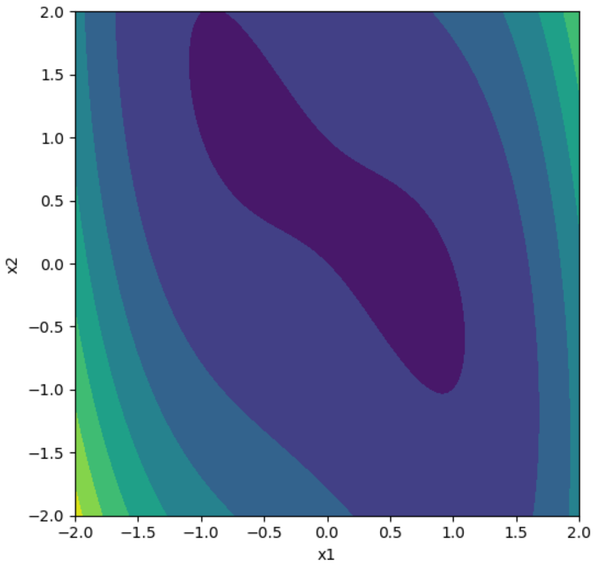
The gradient $\nabla f = \begin{bmatrix} 8x_1 + x_2 + 1 \\ 2x_2 + x_1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 8 & 1 \\ 1 & 2 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, we get $x^* = \begin{bmatrix} -\frac{2}{15} \\ \frac{1}{15} \end{bmatrix}$. We also know the Hessian $H(x) = \begin{bmatrix} 8 & 1 \\ 1 & 2 \end{bmatrix}$ is positive definite because $det|8| > 0$ and $det\begin{vmatrix} 8 & 1 \\ 1 & 2 \end{vmatrix} > 0$.

Thus, there is one critical point: $\boxed{\boldsymbol{x}^* = \begin{bmatrix} -\frac{\mathbf{2}}{\mathbf{15}} \\ \frac{\mathbf{1}}{\mathbf{15}} \end{bmatrix}}$, this critical point is also a $\boxed{\textbf{minimizer}}$.

d) $f(x_1, x_2) = {x_1}^4 + {x_2}^2 + 2x_1x_2 - x_1 - x_2$

plot



To find the critical points, we have

$$\frac{\partial f}{\partial x_1} = 4{x_1}^3 + 2x_2 - 1 \text{ and } \frac{\partial f}{\partial x_2} = 2x_2 + 2x_1 - 1$$

The gradient $\nabla f = \begin{bmatrix} 4{x_1}^3 + 2x_2 - 1 \\ 2x_2 + 2x_1 - 1 \end{bmatrix}$ and the Hessian $H(x) = \begin{bmatrix} 12{x_1}^2 & 2 \\ 2 & 2 \end{bmatrix}$

Let $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and we only consider the scenario that both x_1 and x_2 are real numbers. We get $\boldsymbol{x}^* = \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{2} - \frac{1}{\sqrt{2}} \end{bmatrix} or \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{2} + \frac{1}{\sqrt{2}} \end{bmatrix}$.

$\boxed{\textbf{When } \boldsymbol{x}^* = \begin{bmatrix} \mathbf{0} \\ \frac{\mathbf{1}}{\mathbf{2}} \end{bmatrix}}$,

The Hessian $H(x) = \begin{bmatrix} 0 & 2 \\ 2 & 2 \end{bmatrix}$, so it is positive semi-definite, this critical point is a $\boxed{\textbf{saddle point}}$.

$\boxed{\textbf{When } \boldsymbol{x}^* = \begin{bmatrix} \frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \\ \frac{\mathbf{1}}{\mathbf{2}} - \frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \end{bmatrix}}$,

The Hessian $H(x) = \begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}$, so it is positive definite, this critical point is a $\boxed{\textbf{minimizer}}$.

$\boxed{\textbf{When } \boldsymbol{x}^* = \begin{bmatrix} -\frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \\ \frac{\mathbf{1}}{\mathbf{2}} + \frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \end{bmatrix}}$,

The Hessian $H(x) = \begin{bmatrix} 6 & 2 \\ 2 & 2 \end{bmatrix}$, so it is positive definite, this critical point is a $\boxed{\textbf{minimizer}}$.

Problem 3. Consider the function $f(x) = x(1-x)^2(x-3)$ on the interval $x \in [-1,4]$. From the point $x = 0$, find a descent direction.

a) We know that $\frac{\partial f(x)}{\partial x} = 4x^3 - 15x^2 + 14x - 3$, which gives $\nabla f(0) = -3$. Thus, we can pick the steepest descent direction, $p = -\nabla f(0)/\|\nabla f(0)\| = 1$.

The first derivative of $f(x)$ is,

$$f'(x) = 4x^3 - 15x^2 + 14x - 3$$

The function value and the first derivative are

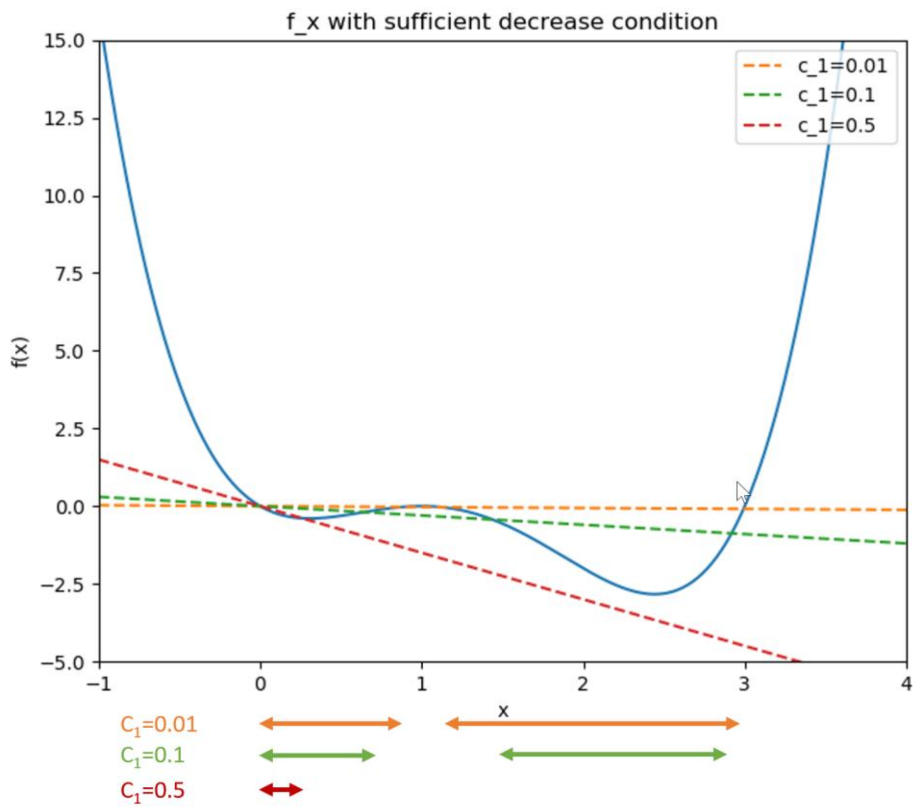
$$f(0) = 0 \text{ and } f'(0) = -3$$

The first Wolfe condition (Armijo condition) is

$$x(1-x)^2(x-3) \leq f(0) + c_1 x f'(0)$$

$$\Rightarrow \boxed{x(1-x)^2(x-3) \leq -3c_1 x}$$

Plot for the sufficient decrease condition

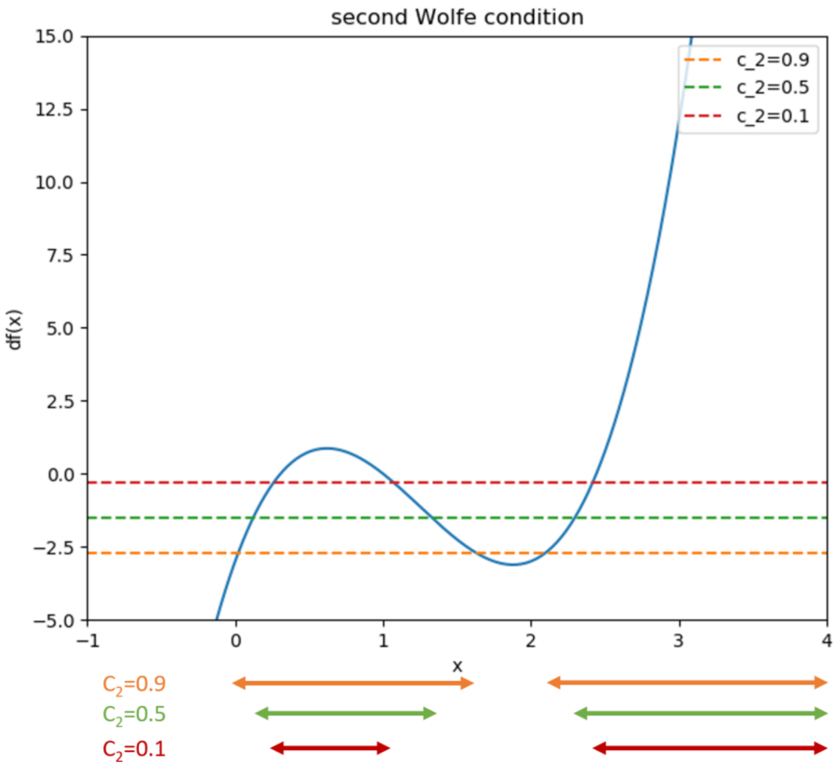


b) The second Wolfe condition is

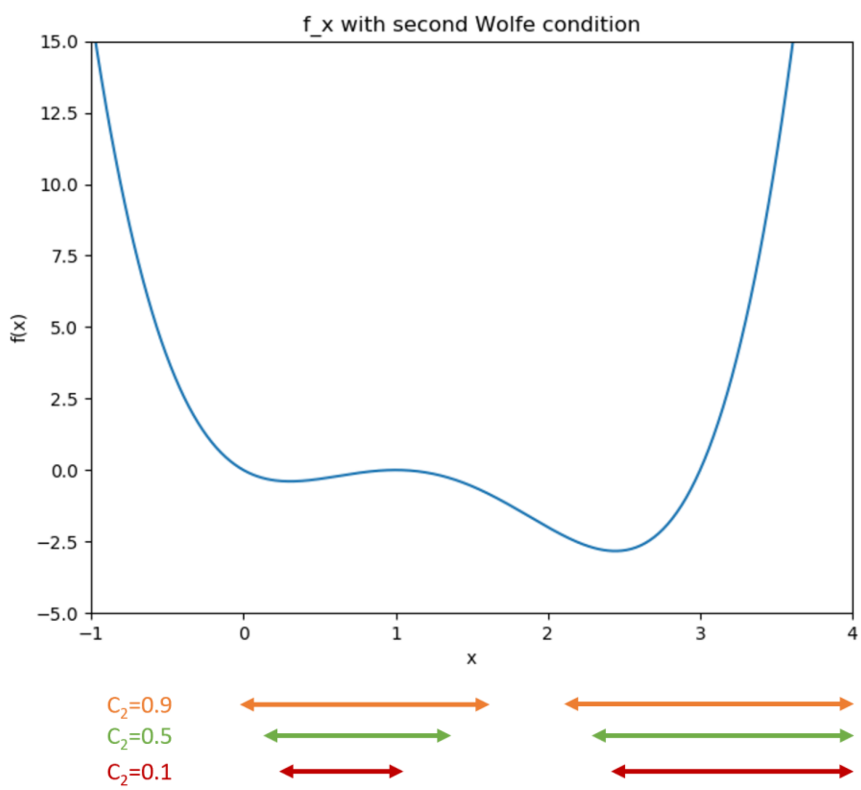
$$4x^3 - 15x^2 + 14x - 3 \geq c_2 f'(0)$$

$$\Rightarrow \boxed{4x^3 - 15x^2 + 14x - 3 \geq -3c_2}$$

To know the interval, we need a plot for the $f'(x)$, as shown below



Hence, plot for the function and the intervals that satisfy the second Wolfe condition

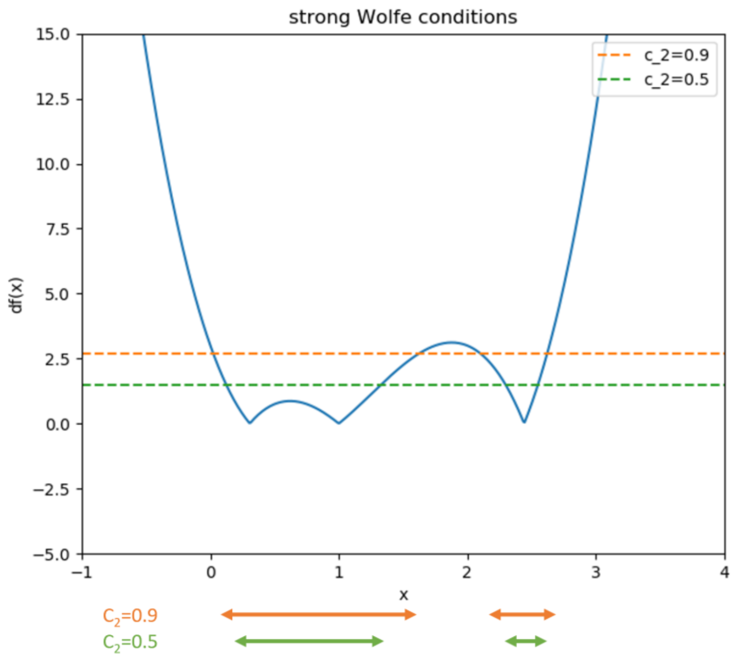


c) The strong Wolfe conditions are

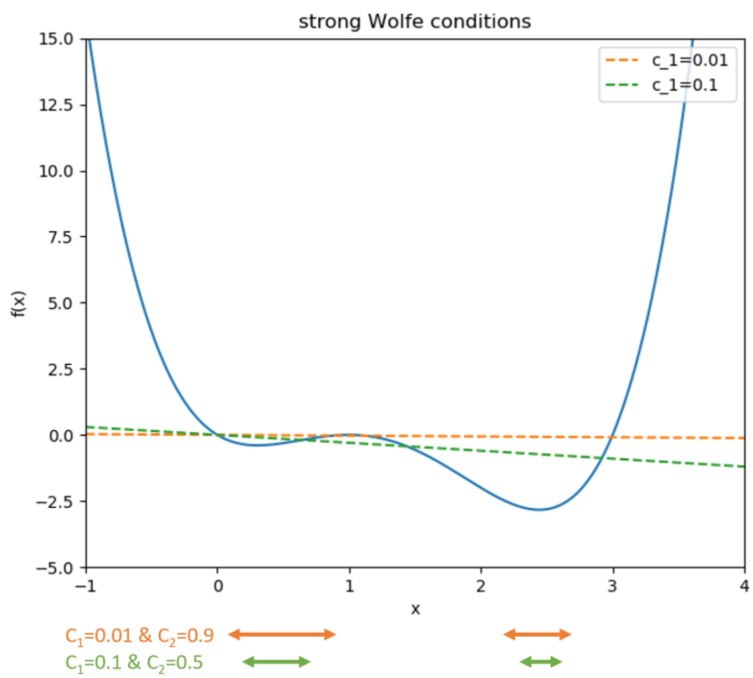
$$\boxed{x(1-x)^2(x-3) \le -3c_1 x}$$

$$\boxed{|4x^3 - 15x^2 + 14x - 3| \le 3c_2}$$

We first plot for the $f'(x)$ to check the second conditions



Hence, plot for the function and the intervals that satisfy the strong Wolfe conditions



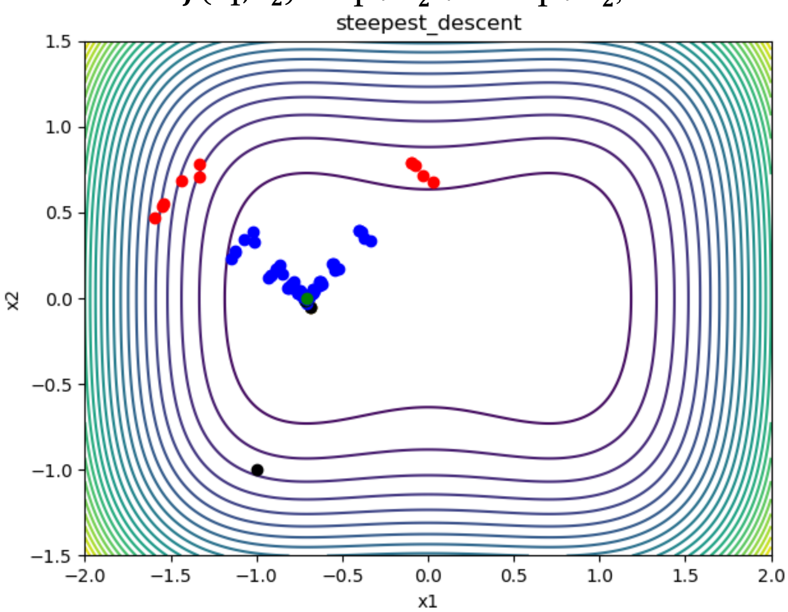
Problem 4. The following question is based on the following two functions:

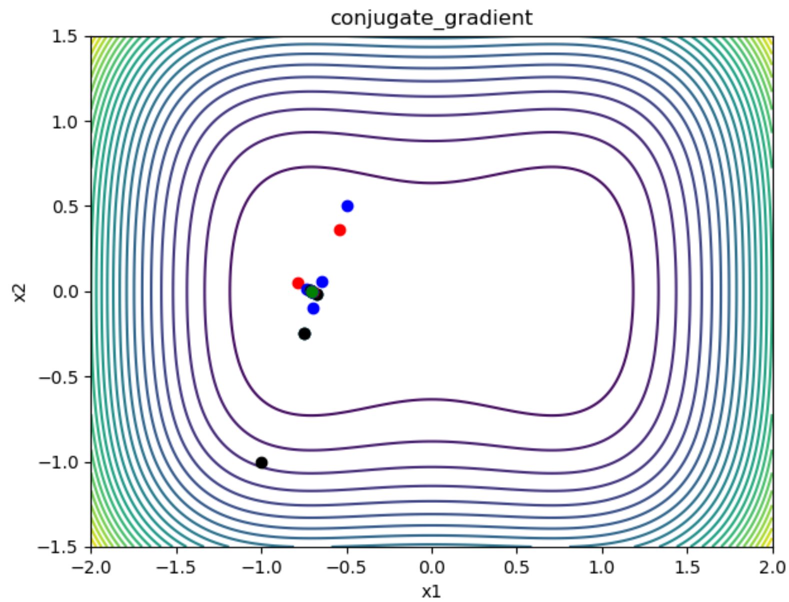
$$f(x_1, x_2) = x_1^4 + x_2^4 + 1 - x_1^2 + x_2^2$$

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

a) We use the strong Wolfe line search for both functions and [-1, -1] as the common starting points.

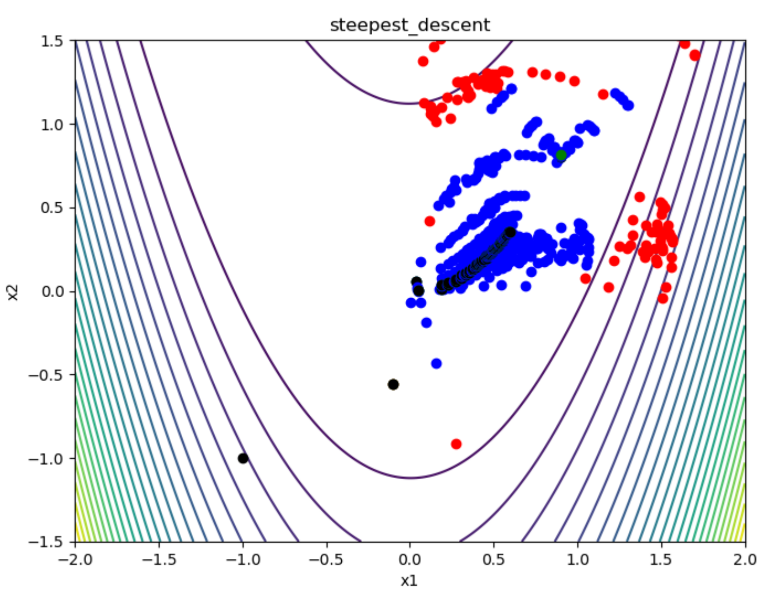
For $\boldsymbol{f(x_1, x_2) = x_1^4 + x_2^4 + 1 - x_1^2 + x_2^2}$,

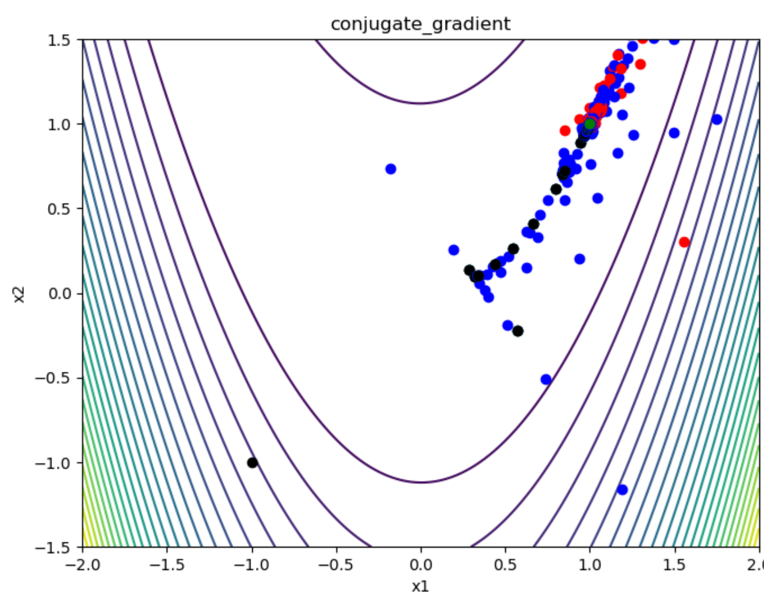




In this case, both algorithms converge within the maximal iterations.

For $\boldsymbol{f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2}$,





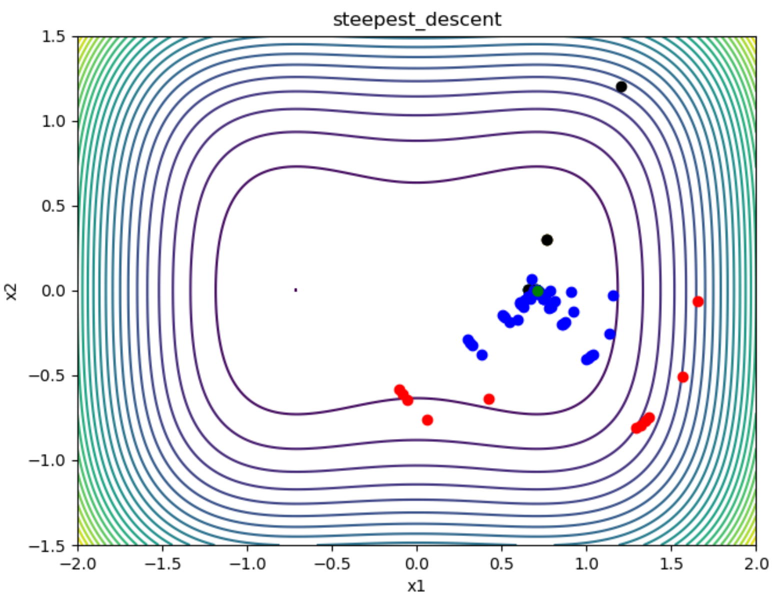
In this case, within maximal iterations, the steepest descent method failed to converge but the conjugate gradient method converges successfully.

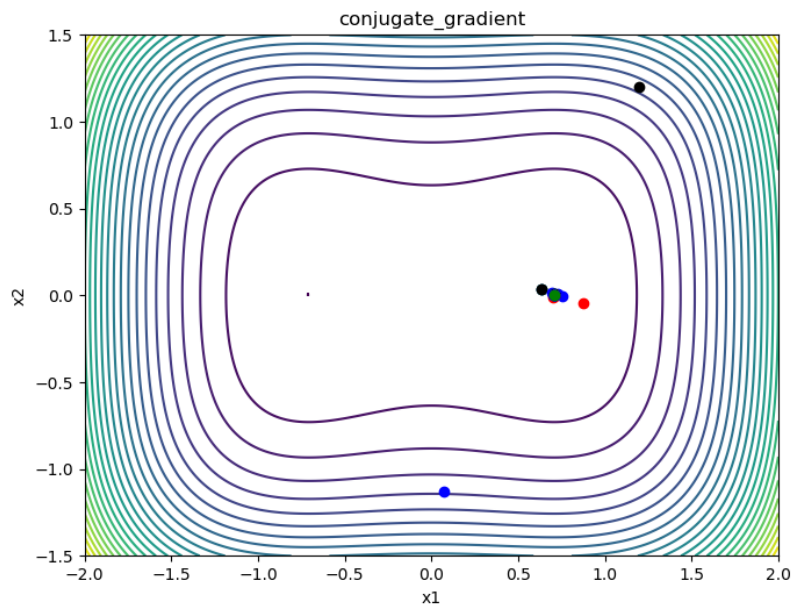
b) Compare the performance of these two algorithms from different starting points. Comment on the difference in performance you observe.

First, based on the figures shown in part a), we can find that the conjugate gradient method converges faster than the steepest descent method. If we change the starting points, the results are shown as below.

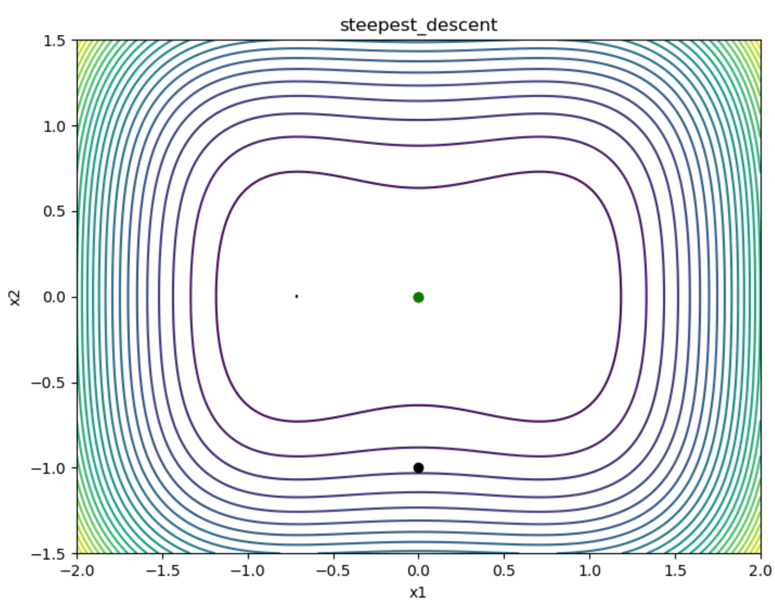
For $\boldsymbol{f(x_1, x_2) = x_1^4 + x_2^4 + 1 - x_1^2 + x_2^2}$,

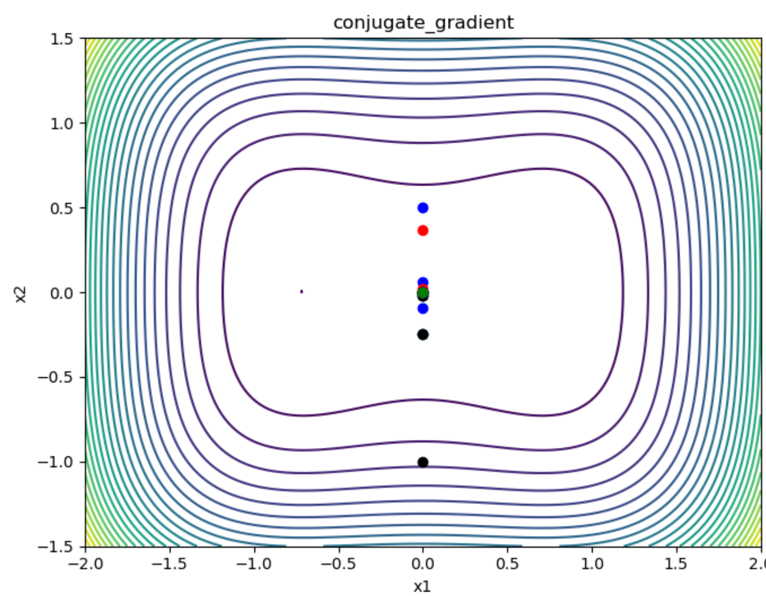
starting point $\boldsymbol{x_0} = [1.2, 1.2]$





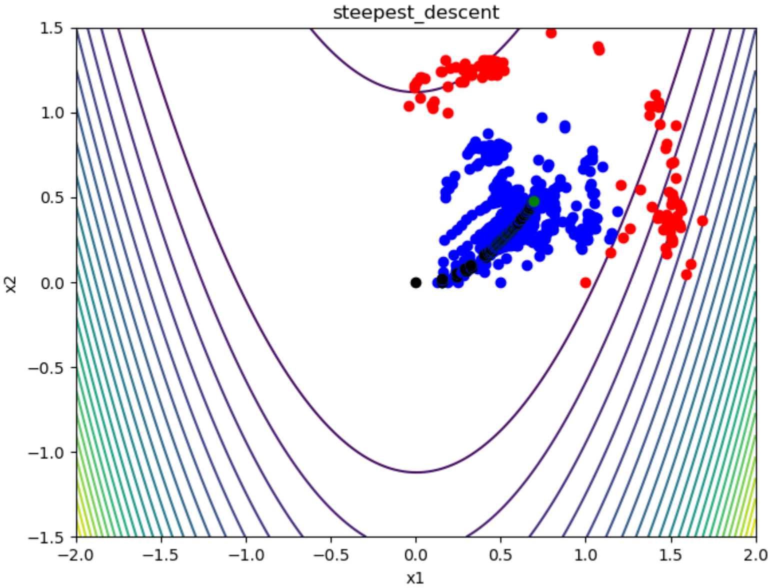
starting point $\boldsymbol{x_0} = [0, -1]$

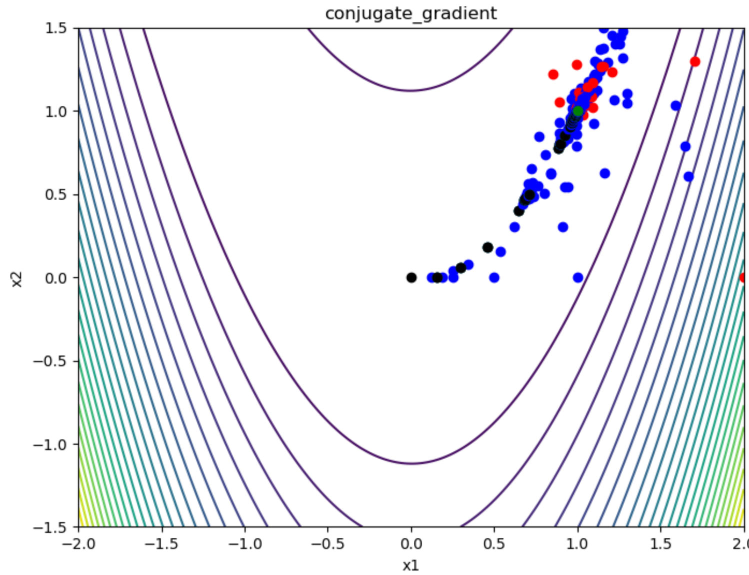




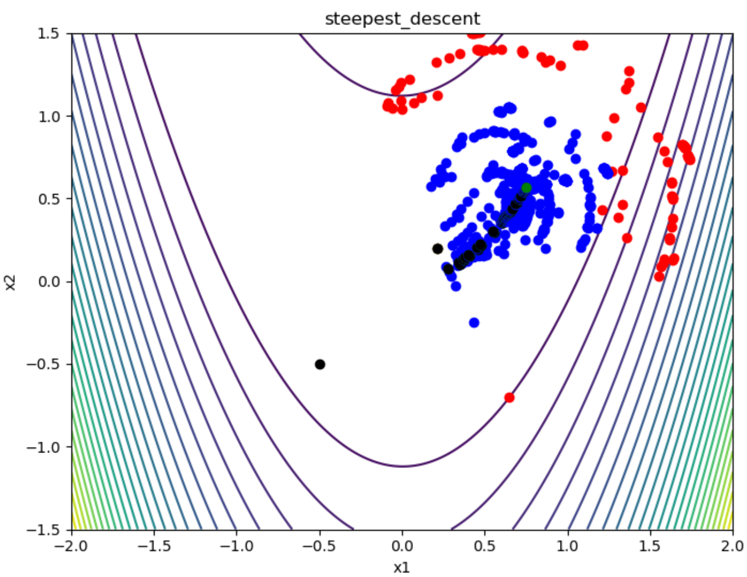
For $\boldsymbol{f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2}$,

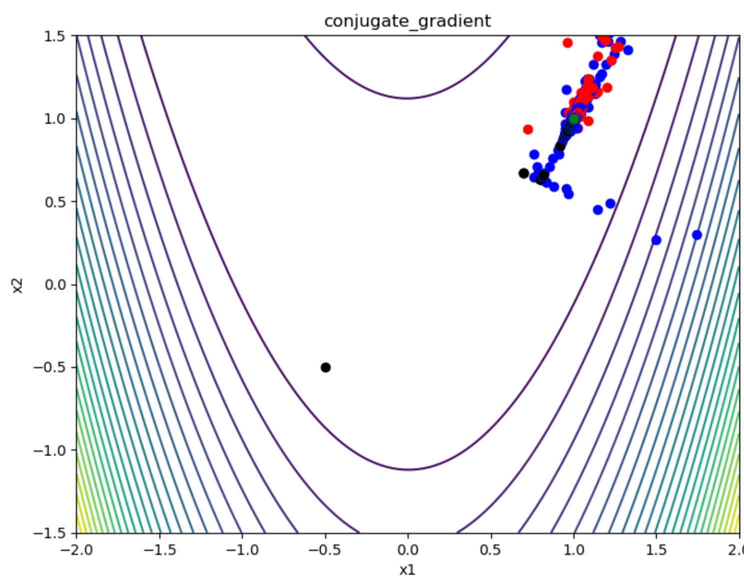
<u>starting point $\boldsymbol{x_0} = [0,0]$</u>





<u>starting point $\boldsymbol{x_0} = [-0.5, -0.5]$</u>





For all tested cases shown above, we can see that <u>generally, the conjugate gradient method converges much faster than the steepest gradient method</u>. However, for some special scenario, for example the first function with the starting point $\boldsymbol{x_0} = [0, -1]$, both methods converge to a saddle point rather than the minimizers. Moreover, the steepest descent method converges faster because the saddle point is right on the opposite direction of the starting point's gradient.

Appendix for Python Code

Prob 1 & 2

```
# Hao Chen AE 6310 HW 1 Prob. 1 & 2 plot

import numpy as np
import matplotlib.pylab as plt


def contour_plot(fobj):
    # Generate the data for a contour plot
    n = 200
    x1 = np.linspace(-2, 2, n)
    x2 = np.linspace(-2, 2, n)
    X1, X2 = np.meshgrid(x1, x2)
    f = np.zeros((n, n))

    # Query the function at the specified locations
    for i in range(n):
        for j in range(n):
            f[i, j] = fobj([X1[i, j], X2[i, j]])

    fig, ax = plt.subplots(1, 1)
    ax.contourf(X1, X2, f)
    ax.set_aspect('equal', 'box')
    fig.tight_layout()
    plt.xlabel('x1')
    plt.ylabel('x2')

    return

# Prob 1, a)
def f1(x):
    return x[0] ** 2 + x[0]*x[1] + x[1] ** 2 - x[0] - x[1]

# Prob 1, b)
def f2(x):
    return x[0] ** 2 + 2*x[0]*x[1] + x[1] ** 2 - x[0] - x[1]

# Prob 1, c)
def f3(x):
    return 3*x[0] ** 2 + 2*x[0]*x[1] + 3*x[1] ** 2 - x[0] - x[1]

# Prob 1, d)
def f4(x):
    return 4*x[0] ** 2 + 6*x[0]*x[1] + 2*x[1] ** 2 + x[0] + x[1]

# Prob 2, a)
def f5(x):
    return x[0] ** 4 + x[1] ** 4 + 1 - x[0] ** 2 + x[1] ** 2

# Prob 2, b)
def f6(x):
    return x[0] ** 2 + x[1] ** 2 + 2*x[0]*x[1]

# Prob 2, c)
def f7(x):
    return 4 * x[0] ** 2 + x[1] ** 2 + x[0] * x[1] + x[0]

# Prob 2, d)
def f8(x):
    return x[0] ** 4 + x[1] ** 2 + 2 * x[0] * x[1] - x[0] - x[1]


funcs = [f1, f2, f3, f4, f5, f6, f7, f8]

for fobj in funcs:
    contour_plot(fobj)
```

```
plt.show()
```

Prob 3

```
# Hao Chen AE 6310 HW 1 Prob. 3 plot

from numpy import *
import matplotlib.pyplot as plt

x = linspace(-1, 4, 500)
f_x = x*((1-x)**2)*(x-3)
df_x = 4*x**3-15*x**2+14*x-3
# Prob. 3 part a)
wolfe1_c1_1 = -3 * 0.01 * x
wolfe1_c1_2 = -3 * 0.1 * x
wolfe1_c1_3 = -3 * 0.5 * x

# Prob. 3 part b)
wolfe2_c2_1 = -3 * 0.9 *x/x  # the *x/x term is simply to uniform the dimension
wolfe2_c2_2 = -3 * 0.5 *x/x
wolfe2_c2_3 = -3 * 0.1 *x/x

# Prob. 3 part c)
ab_df_x = abs(4*x**3-15*x**2+14*x-3)
wolfeS2_c2_1 = 3 * 0.9 *x/x
wolfeS2_c2_2 = 3 * 0.5 *x/x

# Prob. 3 part a)
plt.figure(1)
plt.plot(x, f_x)  # plotting f_x

plt.plot(x, wolfe1_c1_1, '--', label="c_1=0.01")
plt.plot(x, wolfe1_c1_2, '--',  label="c_1=0.1")
plt.plot(x, wolfe1_c1_3, '--',  label="c_1=0.5")
plt.legend(loc="upper right")
plt.xlim(-1, 4)
plt.ylim(-5, 15)
plt.title("f_x with sufficient decrease condition")
plt.xlabel("x")
plt.ylabel("f(x)")

# Prob. 3 part b)
plt.figure(2)
plt.plot(x, df_x)  # plotting df_x

plt.plot(x, wolfe2_c2_1, '--', label="c_2=0.9")
plt.plot(x, wolfe2_c2_2, '--',  label="c_2=0.5")
plt.plot(x, wolfe2_c2_3, '--',  label="c_2=0.1")
plt.legend(loc="upper right")
plt.xlim(-1, 4)
plt.ylim(-5, 15)
plt.title("second Wolfe condition")
plt.xlabel("x")
plt.ylabel("df(x)")

plt.figure(3)
plt.plot(x, f_x)  # plotting f_x
plt.xlim(-1, 4)
plt.ylim(-5, 15)
plt.title("f_x with second Wolfe condition")
plt.xlabel("x")
plt.ylabel("f(x)")

# Prob. 3 part c)
plt.figure(4)
plt.plot(x, ab_df_x)  # plotting df_x

plt.plot(x, wolfeS2_c2_1, '--', label="c_2=0.9")
```

```python
plt.plot(x, wolfeS2_c2_2, '--',  label="c_2=0.5")
plt.legend(loc="upper right")
plt.xlim(-1, 4)
plt.ylim(-5, 15)
plt.title("strong Wolfe conditions")
plt.xlabel("x")
plt.ylabel("df(x)")

plt.figure(5)
plt.plot(x, f_x)  # plotting f_x

plt.plot(x, wolfe1_c1_1, '--', label="c_1=0.01")
plt.plot(x, wolfe1_c1_2, '--',  label="c_1=0.1")
plt.legend(loc="upper right")
plt.xlim(-1, 4)
plt.ylim(-5, 15)
plt.title("strong Wolfe conditions")
plt.xlabel("x")
plt.ylabel("f(x)")

plt.show()
```

Prob 4

```python
# Hao Chen AE 6310 HW 1 Prob. 4 plot

from __future__ import print_function
import numpy as np
import matplotlib.pylab as plt


# define functions
def f1(x, linesearch=False, symb='ko'):
    '''If the linesearch flag is true, plot the point'''
    if linesearch:
        plt.plot([x[0]], [x[1]], symb)
    return x[0] ** 4 + x[1] ** 4 + 1 - x[0] ** 2 + x[1] ** 2


# define function gradients
def f1_grad(x):
    return np.array([4*x[0]**3 - 2 * x[0],
                     4*x[1]**3 + 2 * x[1]])


def f2(x, linesearch=False, symb='ko'):
    if linesearch:
        plt.plot([x[0]], [x[1]], symb)
    return 100*(x[1] - x[0]**2)**2 + (1 - x[0])**2


def f2_grad(x):
    return np.array([2 * (x[0] - 1) + 400 * (x[0] ** 2 - x[1]) * x[0],
                     200 * (x[1] - x[0] ** 2)])


def onedim_plot(func, n=250, xhigh=5.0):
    x = np.linspace(0, xhigh, n)
    f = np.zeros(n)
    for i in range(n):
        f[i] = func([x[i]])
    plt.figure()
    plt.plot(x, f, linewidth=2)
    return


def contour_plot(func, n=250, xlow=-2, xhigh=2, ylow=-1.5, yhigh=1.5):
    '''Create a contour plot of the function'''
    x = np.linspace(xlow, xhigh, n)
```

```python
    y = np.linspace(ylow, yhigh, n)
    X, Y = np.meshgrid(x, y)
    f = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            f[i, j] = func([X[i, j], Y[i, j]])

    fig, ax = plt.subplots(1, 1)
    # if func == rosen:
    #     ax.contour(X, Y, f, levels=np.max(f) * np.linspace(0, 1.0, 50) ** 2)
    # else:
    ax.contour(X, Y, f, levels=np.linspace(np.min(f), np.max(f), 25))
    plt.xlabel('x1')
    plt.ylabel('x2')
    ax.set_aspect('equal', 'box')
    fig.tight_layout()

    return


def backtrack(func, grad_func, x, p,
              tau=0.5, alpha=1.0, c1=1e-3, max_iter=100):
    '''
    Given the function pointer and the gradient function pointer,
    find a step length alpha that satisfies the sufficient decrease
    conditions.
    '''

    # Evaluate the function and gradient at the initial point x
    phi0 = func(x, linesearch=True)
    grad0 = grad_func(x)

    # Compute the derivative of the merit function at alpha = 0.0
    dphi0 = np.dot(grad0, p)

    # Check for a descent direction
    if dphi0 >= 0.0:
        return ValueError('Must provide a descent direction')

    for i in range(max_iter):
        # Evaluate the function at the new point
        xp = x + alpha * p
        phi = func(xp, linesearch=True, symb='ro')

        # Check the sufficient decrease condition
        if phi < phi0 + c1 * alpha * dphi0:
            # Evaluate the function again to illustrate the final point
            func(xp, linesearch=True, symb='go')
            return alpha
        else:
            print('Sufficient decrease failed with alpha = %15.8e' % (alpha))
            print('phi(alpha) =                            %15.8e' % (phi))
            print('phi0 + c1*alpha*dphi0 =                 %15.8e' % (phi0 + c1 * alpha * dphi0))

        # Set the alpha values and append one to the list
        alpha = tau * alpha

    # The line search has failed at this point
    return 0.0


def strong_wolfe(func, grad_func, x, pk, c1=1e-3, c2=0.9,
                 alpha=1.0, alpha_max=100.0, max_iters=100,
                 verbose=False):
    '''
    Strong Wolfe condition line search method

    Input:
```

```
    func:      the function pointer
    grad_func: the gradien function pointer
    x:         the design variables
    p:         the search direction
    alpha:     the initial estimate for the step length
    alpha_max: the maximum value of alpha

    returns:
    alpha:     the step length satisfying the strong Wolfe conditions
    '''

    # Compute the function and the gradient at alpha = 0
    fk = func(x, linesearch=True)
    gk = grad_func(x)

    # Compute the dot product of the gradient with the search
    # direction to evaluate the derivative of the merit function
    proj_gk = np.dot(gk, pk)

    # Store the old value of the objective
    fj_old = fk
    proj_gj_old = proj_gk
    alpha_old = 0.0

    for j in range(max_iters):
        # Evaluate the merit function
        fj = func(x + alpha * pk, linesearch=True, symb='ro')

        # Evaluate the gradient at the new point
        gj = grad_func(x + alpha * pk)
        proj_gj = np.dot(gj, pk)

        # Check if either the sufficient decrease condition is
        # violated or the objective increased
        if (fj > fk + c1 * alpha * proj_gk or
                (j > 0 and fj > fj_old)):
            if verbose:
                print('Sufficient decrease conditions violated: interval found')
            # Zoom and return
            return zoom(func, grad_func, fj_old, proj_gj_old, alpha_old,
                        fj, proj_gj, alpha,
                        x, fk, gk, pk, c1=c1, c2=c2, verbose=verbose)

        # Check if the strong Wolfe conditions are satisfied
        if np.fabs(proj_gj) <= c2 * np.fabs(proj_gk):
            if verbose:
                print('Strong Wolfe alpha found directly')
            func(x + alpha * pk, linesearch=True, symb='go')
            return alpha

        # If the line search is vioalted
        if proj_gj >= 0.0:
            if verbose:
                print('Slope condition violated; interval found')
            return zoom(func, grad_func, fj, proj_gj, alpha,
                        fj_old, proj_gj_old, alpha_old,
                        x, fk, gk, pk, c1=c1, c2=c2, verbose=verbose)

        # Record the old values of alpha and fj
        fj_old = fj
        proj_gj_old = proj_gj
        alpha_old = alpha

        # Pick a new value for alpha
        alpha = min(2.0 * alpha, alpha_max)

        if alpha >= alpha_max:
            if verbose:
                print('Line search failed here')
```

```python
            return None

    if verbose:
        print('Line search unsuccessful')
    return alpha


def zoom(func, grad_func, f_low, proj_low, alpha_low,
         f_high, proj_high, alpha_high,
         x, fk, gk, pk, c1=1e-3, c2=0.9, max_iters=100, verbose=False):
    '''
    Zoom function: Locate a value between alpha_low and alpha_high
    that satisfies the strong Wolfe conditions. Remember:
    alpha_low/alpha_high are step lengths yielding the
    lowest/higher values of the merit function.

    input:
    f_low:      the value of f(x) at alpha_low
    proj_low:   the value of the derivative of phi at alpha_low
    alpha_low:  the value of the step at alpha_low
    f_high:     the value of f(x) at alpha_high
    proj_high:  the value of the derivative of phi at alpha_high
    alpha_high: the value of the step at alpha_high
    x:          the value of the design variables at alpha = 0
    fk:         the value of the function at alpha = 0
    gk:         the gradient of the function at alpha = 0
    pk:         the line search direction

    returns:
    alpha:   a step length satisfying the strong Wolfe conditions
    '''

    proj_gk = np.dot(pk, gk)

    for j in range(max_iters):
        # Pick an alpha value using cubic interpolation
        # alpha_j = cubic_interp(alpha_low, f_low, proj_low,
        #                        alpha_high, f_high, proj_high)

        # Pick an alpha value by bisecting the interval
        alpha_j = 0.5 * (alpha_high + alpha_low)

        # Evaluate the merit function
        fj = func(x + alpha_j * pk, linesearch=True, symb='bo')

        # Check if the sufficient decrease condition is violated
        if fj > fk + c1 * alpha_j or fj >= f_low:
            if verbose:
                print('Zoom: Sufficient decrease conditions violated')
            alpha_high = alpha_j
            f_high = fj

            # We need the derivative here for proj_high
            gj = grad_func(x + alpha_j * pk)
            proj_high = np.dot(gj, pk)
        else:
            # Evaluate the gradient of the function and the
            # derivative of the merit function
            gj = grad_func(x + alpha_j * pk)
            proj_gj = np.dot(gj, pk)

            # Return alpha, the strong Wolfe conditions are
            # satisfied
            if np.fabs(proj_gj) <= c2 * np.fabs(proj_gk):
                if verbose:
                    print('Zoom: Wolfe conditions satisfied')
                func(x + alpha_j * pk, linesearch=True, symb='go')
                return alpha_j
            elif verbose:
```

```python
                print('Zoom: Curvature condition violated')

            # Make sure that we have the intervals right
            if proj_gj * (alpha_high - alpha_low) >= 0.0:
                # Swap alpha high/alpha low
                alpha_high = alpha_low
                proj_high = proj_low
                f_high = f_low

            # Swap alpha low/alpha j
            alpha_low = alpha_j
            proj_low = proj_gj
            f_low = fj

    return alpha_j


def cubic_interp(self, x0, m0, dm0, x1, m1, dm1, verbose=False):
    '''
    Return an x in the interval (x0, x1) that minimizes a cubic
    interpolant between two points with both function and
    derivative values.

    This method does not assume that x0 > x1. If the solution is
    not in the interval, the function returns the mid-point.
    '''

    # Compute d1
    d1 = dm0 + dm1 - 3 * (m0 - m1) / (x0 - x1)

    # Check that the square root will be real in the
    # expression for d2
    if (d1 ** 2 - dm0 * dm1) < 0.0:
        if verbose:
            print('Cubic interpolation fail')
        return 0.5 * (x0 + x1)

    # Compute d2
    d2 = np.sign(x1 - x0) * np.sqrt(d1 ** 2 - dm0 * dm1)

    # Evaluate the new interpolation point
    x = x1 - (x1 - x0) * (dm1 + d2 - d1) / (dm1 - dm0 + 2 * d2)

    # If the new point is outside the interval, return
    # the mid point
    if x1 > x0 and (x > x1 or x < x0):
        return 0.5 * (x0 + x1)
    elif x0 > x1 and (x > x0 or x < x1):
        return 0.5 * (x0 + x1)

    return x


def steepest_descent(x0, func, grad_func,
                     c1=1e-3, c2=0.9, eps=1e-6,
                     max_iters=5000, line_search_type='strong Wolfe'):
    '''
    Solve an unconstrained optimization problem using the steepest descent method.

    input:
    x0:         the starting point
    func:       a function pointer to f(x)
    grad_func:  a function pointer to the gradient of f(x)
    c1:         sufficient decrease parameter
    c2:         curvature condition parameter
    eps:        stopping tolerance such that ||grad f(x)|| < eps
    max_iters:  maximum number of iterations before we give up
    '''
    # Make sure we are using a np array
```

```
    x = np.array(x0)

    for i in range(max_iters):
        grad = grad_func(x)

        # Check the norm of the gradient
        if np.sqrt(np.dot(grad, grad)) < eps:
            print('Steepest descent found first-order point\n')
            return x

        pk = -grad / np.sqrt(np.dot(grad, grad))

        if line_search_type == 'strong Wolfe':
            alpha = strong_wolfe(func, grad_func, x, pk, c1=c1, c2=c2)
        else:
            alpha = backtrack(func, grad_func, x, pk, c1=c1)
        x += alpha * pk

    print('Steepest descent failed\n')
    return x


def conjugate_gradient(x0, func, grad_func,
                       c1=1e-3, c2=0.9, eps=1e-6,
                       max_iters=5000, line_search_type='strong Wolfe'):
    '''
    Solve an unconstrained optimization problem using the Fletcher-Reeves
    conjugate gradient method

    input:
    x0:         the starting point
    func:       a function pointer to f(x)
    grad_func:  a function pointer to the gradient of f(x)
    c1:         sufficient decrease parameter
    c2:         curvature condition parameter
    eps:        stopping tolerance such that ||grad f(x)|| < eps
    max_iters:  maximum number of iterations before we give up
    '''
    # Make sure we are using a np array
    x = np.array(x0)
    grad_prev = np.zeros(x.shape)
    p_prev = np.zeros(x.shape)

    for i in range(max_iters):
        grad = grad_func(x)

        # Check the norm of the gradient
        if np.sqrt(np.dot(grad, grad)) < eps:
            print('Conjugate gradient method found a first-order point\n')
            return x

        if i > 0:
            beta = np.dot(grad, grad) / np.dot(grad_prev, grad_prev)
            pk = -grad + beta * p_prev
        else:
            pk = -grad

        # Check if we have a descent direction. If not revert to the
        # steepest descent method.
        if np.dot(pk, grad) >= 0.0:
            pk = -grad

        if line_search_type == 'strong Wolfe':
            alpha = strong_wolfe(func, grad_func, x, pk, c1=c1, c2=c2)
        else:
            alpha = backtrack(func, grad_func, x, pk, c1=c1)
        x += alpha * pk

        # Save the previous information
```

```
        grad_prev[:] = grad[:]
        p_prev[:] = pk[:]

    print('Conjugate gradient method failed to converge\n')
    return x

#################################################
# func definition
func = f1  # f1 or f2
func_grad = f1_grad  # f1_grad or f2_grad
# starting point
x0 = [-1.0, -1.0]
#x0 = [-0.5, -0.5]

# steepest_descent
# plt.figure(1)
# Create a contour plot, f1 or f2
contour_plot(func)

# Perform a strong Wolfe line search
# func input can be "f1, f1_grad" or "f2, f2_grad"
xstar = steepest_descent(x0, func, func_grad, c2=0.1, max_iters=100)
print(xstar)

plt.title("steepest_descent")
plt.xlim(-2, 2)
plt.ylim(-1.5, 1.5)

##################################################
# conjugate_gradient
# plt.figure(2)
# Create a contour plot, f1 or f2
contour_plot(func)

# Perform a strong Wolfe line search
# func input can be "f1, f1_grad" or "f2, f2_grad"
xstar = conjugate_gradient(x0, func, func_grad, c2=0.1, max_iters=100)
print(xstar)

plt.title("conjugate_gradient")
plt.xlim(-2, 2)
plt.ylim(-1.5, 1.5)
plt.show()
```