Lab 3 : Convex Optimisation

Gradient Descent

Write the code following the instructions to obtain the desired results

Import all the required libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Find the value of x at which $f(x)$ is minimum :

1. Find x analytically
2. Write the update equation of gradient descent
3. Find x using gradient descent method

Example 1 : $f(x) = x^2 + x + 2$

Analytical :

$$\frac{d}{dx}f(x) = 2x + 1 = 0$$

$$\frac{d^2}{dx^2}f(x) = 2 \ (Minima)$$

$$x = -\frac{1}{2} \ (analytical \ solution)$$

Gradient Descent Update equation :

$$x_{init} = 4$$

$$x_{updt} = x_{old} - \lambda(\frac{d}{dx}f(x)|x = x_{old})$$

$$x_{updt} = x_{old} - \lambda(2x_{old} + 1)$$

Gradient Descent Method :

Follow the below steps and write your code in the block below

1. Generate x, 1000 data points from -10 to 10
2. Generate and Plot the function $f(x) = x^2 + x + 2$
3. Initialize the starting point (x_{init}) and learning rate (λ)
4. Use Gradient descent algorithm to compute value of x at which the function $f(x)$ is minimum
5. Also vary the learning rate and initialisation point and plot your observations

Logic for gradient descent

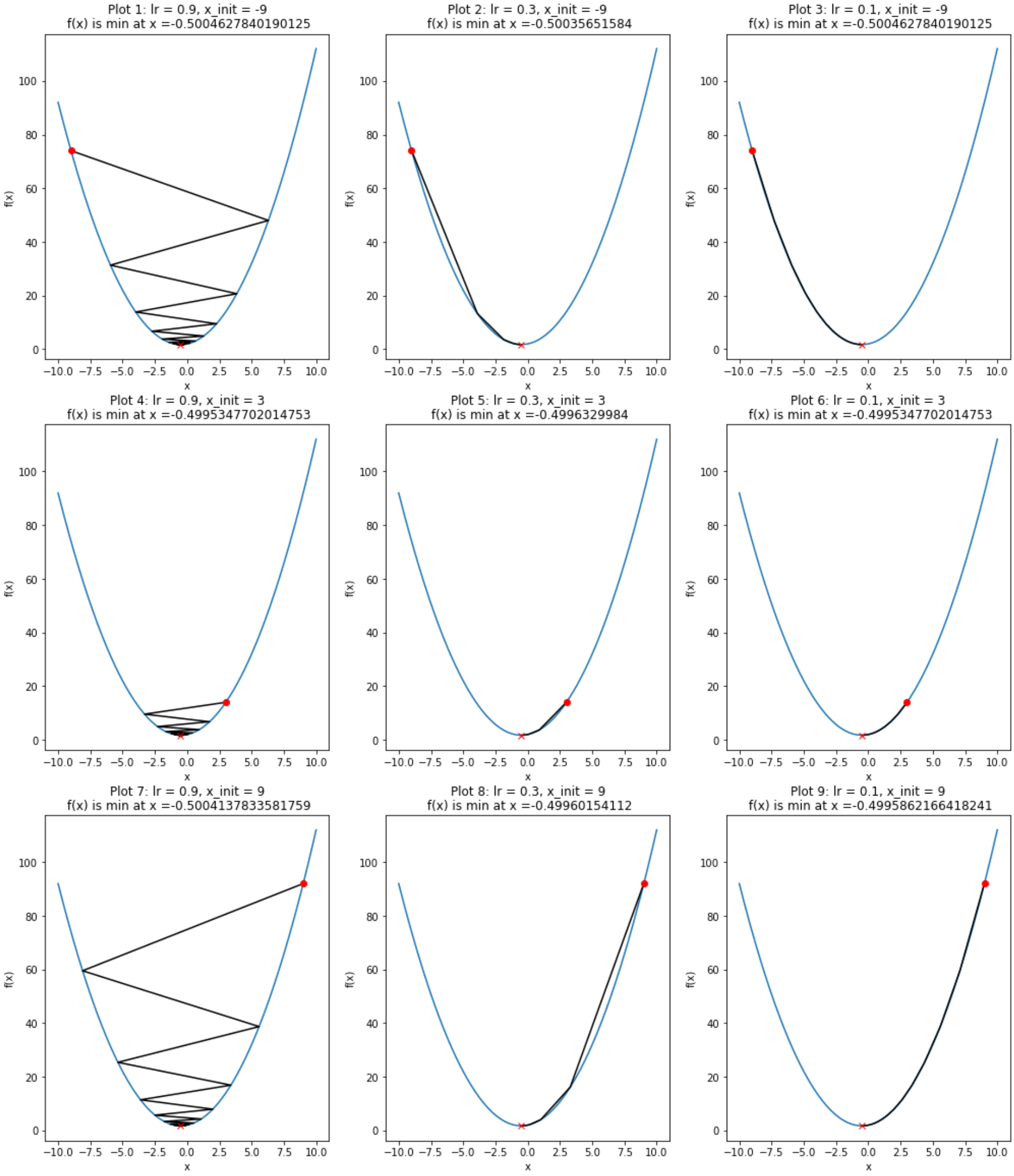
For functions with 1 variable

For a function $f(x)$, minimum/maximum is at a point where $\frac{df(x)}{dx} = 0$. If $f(x)$ is convex, then we get minimum.

In the code, we randomly initialize a starting point x_{init} and we update it according to the formula given above $x_{updt} = x_{old} - \lambda(\frac{d}{dx}f(x)|x = x_{old})$ ($x_{old} = x_{init}$ for first update). As we keep updating, we keep getting closer and closer to $\frac{df(x)}{dx} = 0$. Ideally we would want $\frac{df(x)}{dx} = 0$, but keeping a strict condition of getting $\frac{df(x)}{dx} = 0$ may make the code run without converging. So, we take a small boundary of (-0.001,0.001) around 0 and stop iteration when $\frac{df(x)}{dx}$ is within this boundary.

In [2]:

```
## Write your code here
# defining f(x)
def f (x):
    return x*x+x+2
# defining df(x)/dx
def dfdx (x):
    return 2*x+1
# creating x, 1000 data points from -10 to 10
x = np.linspace(-10,10,1000);
# creating f(x) from x
fx = f(x);
# array of different initial points
x_init_arr = [-9,3,9];
# array of different learning rates
lr_arr = [0.9,0.3,0.1];
# count variable for subplots
i=1
# fixing figure sixe
plt.figure(figsize=(17,20))
# looping over x_init array and lr array
for x_init in x_init_arr:
    for lr in lr_arr:
        # subplot
        plt.subplot(3,3,i)
        i+=1 # incrementing for next subplot
        x_hist = np.array([]); # history array to store all values of x in each update
        f_hist = np.array([]); # history array to store all values of f(x) in each update
        x_hist = np.append(x_hist,x_init); # appending values to array
        f_hist = np.append(f_hist,f(x_init)); # appending values to array
        # loop to get updates till convergence
        # logic is explained in markdown above
        while (dfdx(x_hist[-1])>=0.001 or dfdx(x_hist[-1])<=-0.001):
            x_old = x_hist[-1]; # old x before update
            x_up = x_old - lr*dfdx(x_old); # updating x
            x_hist = np.append(x_hist,x_up); # appending updated x to history array
            f_hist = np.append(f_hist,f(x_up)); # appending updated f(x) to history array
        # plotting y=f(x)
        plt.plot(x,fx)
        # plotting history of updates
        plt.plot(x_hist,f_hist,'k')
        # plotting optimal value
        plt.plot(x_hist[-1],f_hist[-1],color='r',marker='x')
        # plotting initial values
        plt.plot(x_hist[0],f_hist[0],color='r',marker='o')
        # titles and labels
        plt.title("Plot "+str(i-1)+": lr = "+str(lr)+", x_init = "+str(x_init)
                  +"\n f(x) is min at x ="+str(x_hist[-1]))
        plt.xlabel("x")
        plt.ylabel("f(x)")
```



Example 2 : $f(x) = xsinx$

Analytical : Find solution analytically

Gradient Descent Update equation : Write Gradient descent update equations

Gradient Descent Method :

Follow the below steps and write your code in the block below

1. Generate x, 1000 data points from -10 to 10
2. Generate and Plot the function $f(x)$ = x^2 + x + 2
3. Initialize the starting point (x_{init}) and learning rate (λ)
4. Use Gradient descent algorithm to compute value of x at which the function $f(x)$ is minimum
5. Also vary the learning rate and initialisation point and plot your observations

Logic for gradient descent

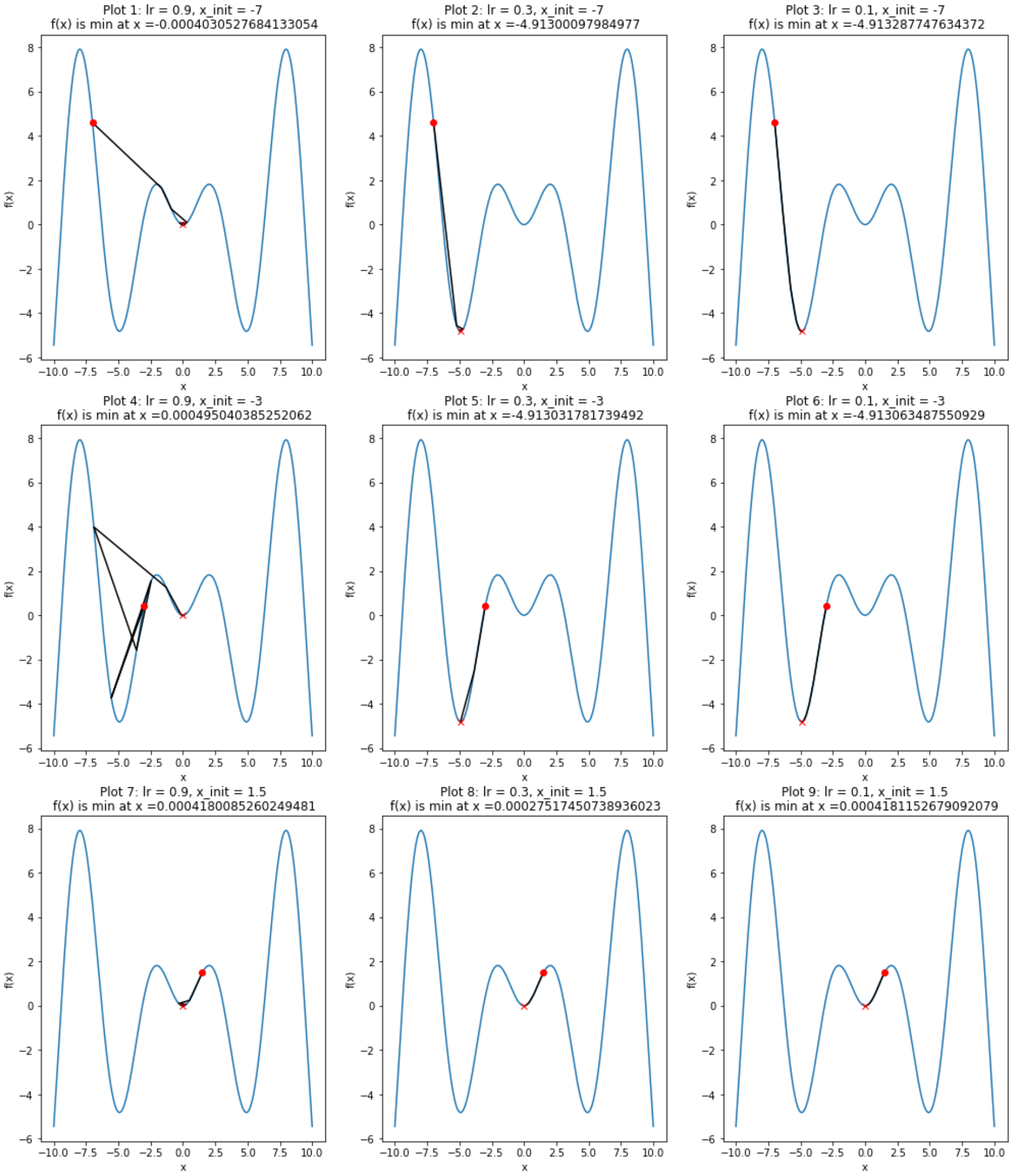
For functions with 1 variable

For a function $f(x)$, minimum/maximum is at a point where $\frac{df(x)}{dx} = 0$. If $f(x)$ is convex, then we get minimum.

In the code, we randomly initialize a starting point x_{init} and we update it according to the formula given above $x_{updt} = x_{old} - \lambda(\frac{d}{dx}f(x)|x = x_{old})$ ($x_{old} = x_{init}$ for first update). As we keep updating, we keep getting closer and closer to $\frac{df(x)}{dx} = 0$. Ideally we would want $\frac{df(x)}{dx} = 0$, but keeping a strict condition of getting $\frac{df(x)}{dx} = 0$ may make the code run without converging. So, we take a small boundary of (-0.001,0.001) around 0 and stop iteration when $\frac{df(x)}{dx}$ is within this boundary.

In [3]:

```
## Write your code here
# defining f(x)
def f (x):
    return x*np.sin(x)
# defining df(x)/dx
def dfdx (x):
    return np.sin(x)+x*np.cos(x)
# creating x, 1000 data points from -10 to 10
x = np.linspace(-10,10,1000);
# creating f(x) from x
fx = f(x);
# array of different initial points
x_init_arr = [-7,-3,1.5];
# array of different learning rates
lr_arr = [0.9,0.3,0.1];
# count variable for subplots
i=1
# fixing figure sixe
plt.figure(figsize=(17,20))
# looping over x_init array and lr array
for x_init in x_init_arr:
    for lr in lr_arr:
        # subplot
        plt.subplot(3,3,i)
        i+=1 # incrementing for next subplot
        x_hist = np.array([]); # history array to store all values of x in each update
        f_hist = np.array([]); # history array to store all values of f(x) in each update
        x_hist = np.append(x_hist,x_init); # appending  initial value to array
        f_hist = np.append(f_hist,f(x_init)); # appending initial value to array
        # loop to get updates till convergence
        # logic is explained in markdown above
        while (dfdx(x_hist[-1])>=0.001 or dfdx(x_hist[-1])<=-0.001):
            x_old = x_hist[-1]; # old x before update
            x_up = x_old - lr*dfdx(x_old); # updating x
            x_hist = np.append(x_hist,x_up); # appending updated x to history array
            f_hist = np.append(f_hist,f(x_up)); # appending updated f(x) to history array
        # plotting y=f(x)
        plt.plot(x,fx)
        # plotting history of updates
        plt.plot(x_hist,f_hist,'k')
        # plotting optimal value
        plt.plot(x_hist[-1],f_hist[-1],color='r',marker='x')
        # plotting initial values
        plt.plot(x_hist[0],f_hist[0],color='r',marker='o')
        # titles and labels
        plt.title("Plot "+str(i-1)+": lr = "+str(lr)+", x_init = "+str(x_init)
                  +"\n f(x) is min at x ="+str(x_hist[-1]))
        plt.xlabel("x")
        plt.ylabel("f(x)")
```



Find the value of x and y at which $f(x, y)$ is minimum :

Example 1 : $f(x, y) = x^2 + y^2 + 2x + 2y$

Gradient Descent Method :

Follow the below steps and write your code in the block below

1. Generate x and y, 1000 data points from -10 to 10
2. Generate and Plot the function $f(x, y) = x^2 + y^2 + 2x + 2y$
3. Initialize the starting point (x_{init},y_{init}) and learning rate (λ)
4. Use Gradient descent algorithm to compute value of x and y at which the function $f(x, y)$ is minimum
5. Also vary the learning rate and initialisation point and plot your observations

Logic for gradient descent

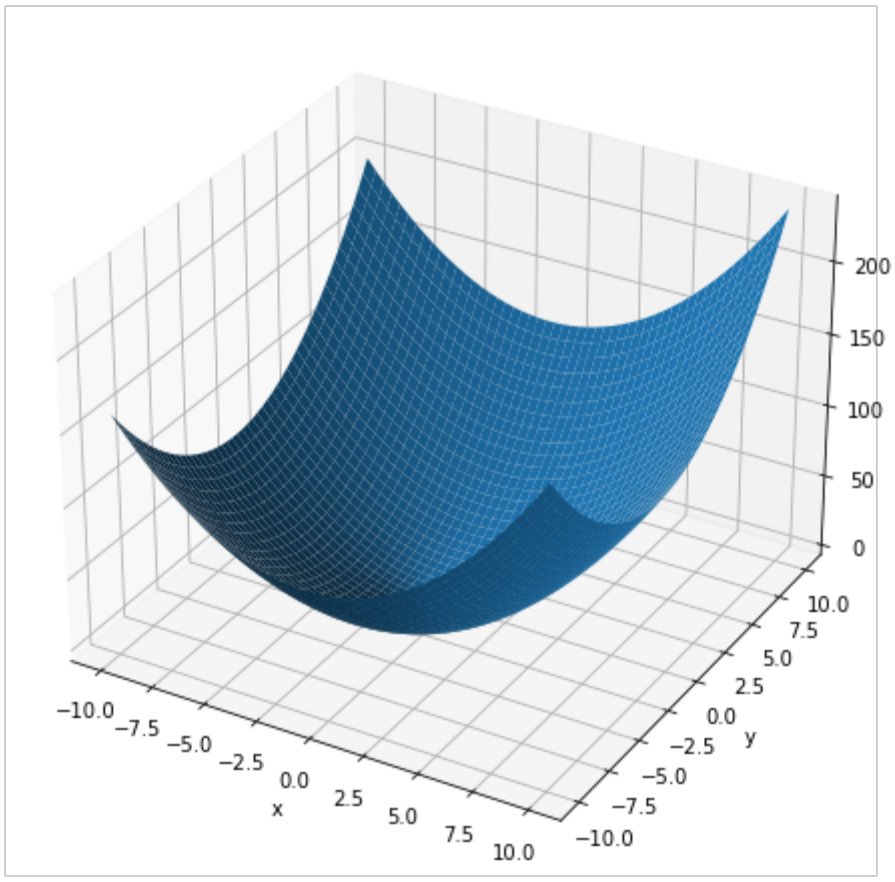
For functions with m variables

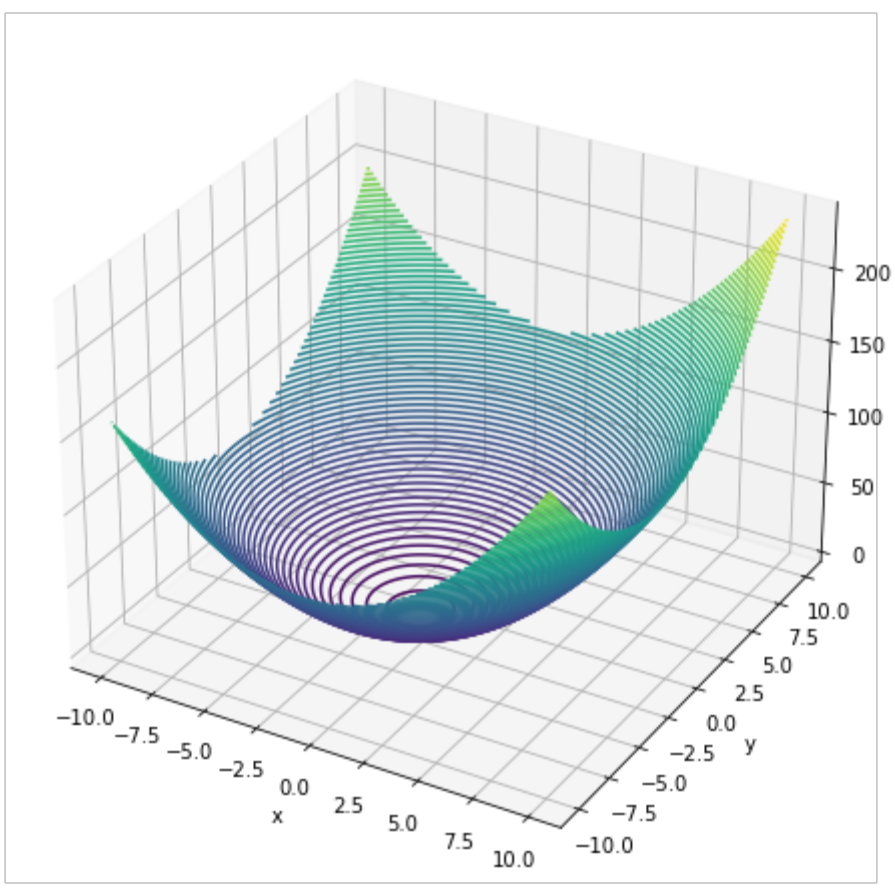
For a multivariate function $f(x_1, x_2, \ldots, x_n)$, minimum/maximum is at a point where $\frac{\partial f}{\partial x_i} = 0 \, \forall \, i = 1, 2, \ldots, n$. If $f(x_1, x_2, \ldots, _n)$ is convex, then we get minimum.

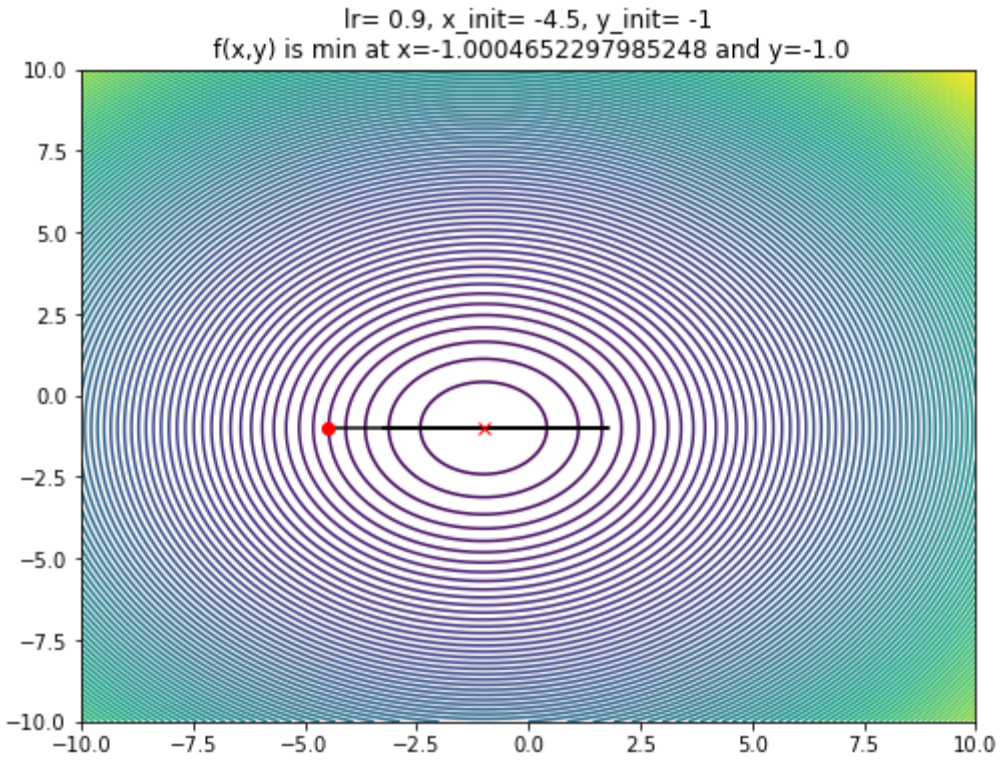
In the code, we randomly initialize starting points $x_{i,init} \, \forall \, i = 1, 2, \ldots, n$ and we update it according to the formula given above $x_{i,updt} = x_{i,old} - \lambda \left(\frac{\partial f}{\partial x_i} f(x_1, x_2, \ldots, x_n) \mid x_i = x_{i,old} \right)$ [$x_{i,old} = x_{i,init}$ for first update]. As we keep updating, we keep getting closer and closer to $\frac{\partial f}{\partial x_i} = 0$. Ideally we would want $\frac{\partial f}{\partial x_i} = 0$, but keeping a strict condition of getting $\frac{\partial f}{\partial x_i} = 0$ may make the code run without converging. So, we take a small boundary of (-0.001,0.001) around 0 and stop iteration when $\frac{\partial f}{\partial x_i}$ is within this boundary.

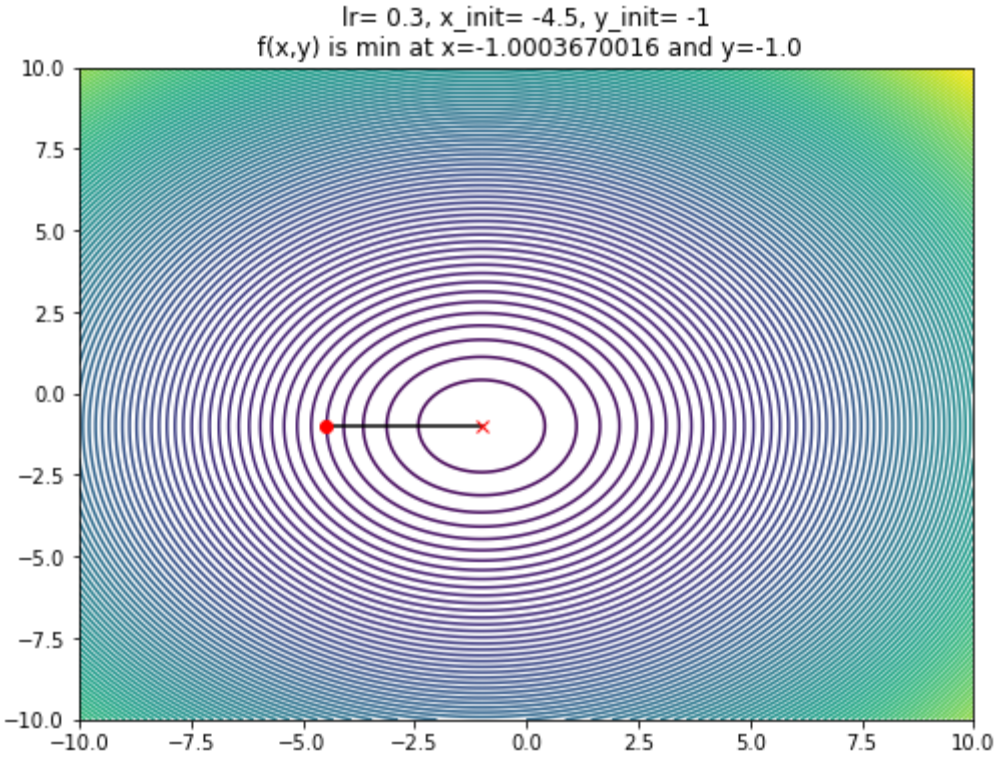
In [4]:

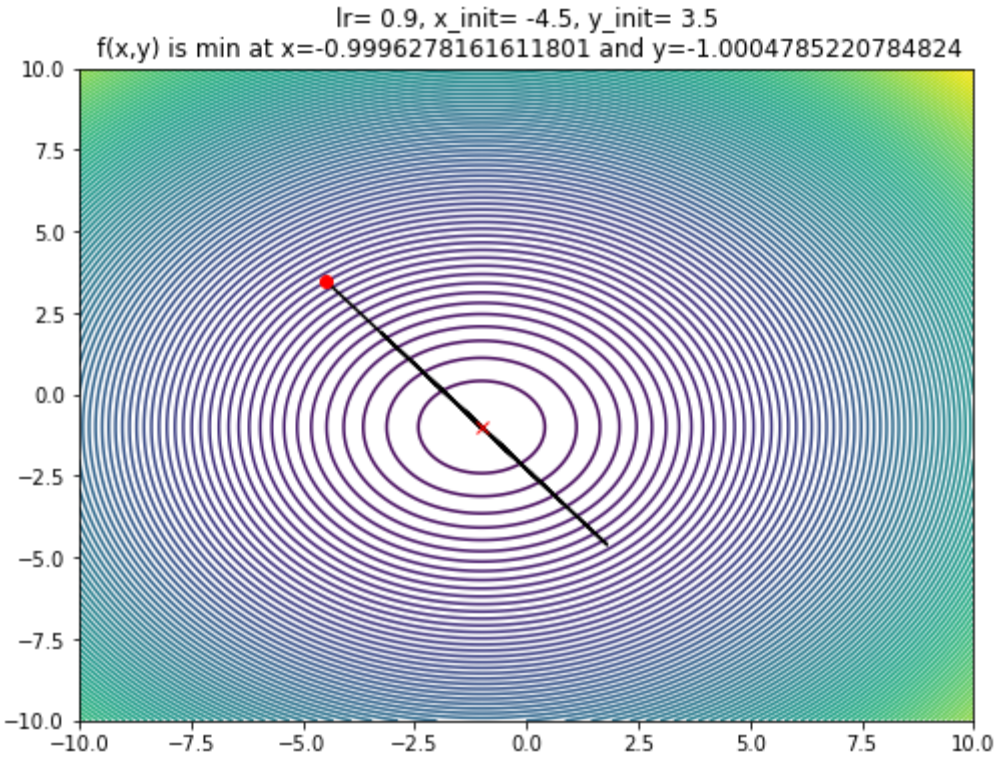
```
## Write your code here (Ignore the warning)
# defining f(x,y)
def f (x,y):
    return x**2+y**2+2*x+2*y
# defining partial differtiations
def dfdx_x (x,y):
    return 2*x+2
def dfdx_y (x,y):
    return 2*y+2
# Generating x and y, 1000 data points from -10 to 10
x = np.linspace(-10,10,1000);
y = np.linspace(-10,10,1000);
# meshgrid for plotting 3D plots
X,Y = np.meshgrid(x,y);
# generating f(x,y)
fxy = f(X,Y);
# plotting Surface plot
plt.figure(figsize=(8,8))
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, fxy)
ax.set_xlabel('x');
ax.set_ylabel('y');
ax.set_zlabel('z');
#plotting contour 3D plot
plt.figure(figsize=(8,8))
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, fxy,80)
ax.set_xlabel('x');
ax.set_ylabel('y');
ax.set_zlabel('z');
# array of initial values of x ,y
x_init_arr = [-4.5,1];
y_init_arr = [-1,3.5];
# array of different values of learning rate
lr_arr = [0.9,0.3];
# looping over inital points and learning rates
for x_init in x_init_arr:
    for y_init in y_init_arr:
        for lr in lr_arr:

            x_hist = np.array([]); # history array to store all values of x in each update
            y_hist = np.array([]); # history array to store all values of y in each update
            x_hist = np.append(x_hist,x_init); # appending initial value to array
            y_hist = np.append(y_hist,y_init); # appending initial value to array
            # loop to get updates till convergence
            # logic is explained in markdown above
            while (dfdx_x(x_hist[-1],y_hist[-1])>=0.001 or dfdx_x(x_hist[-1],y_hist[-1])<=-0.001
                   or dfdx_y(x_hist[-1],y_hist[-1])>=0.001 or dfdx_y(x_hist[-1],y_hist[-1])<=-0.001):
                x_old = x_hist[-1] # old x before update
                y_old = y_hist[-1] # old y before update
                x_up = x_old - lr*dfdx_x(x_old,y_old); # updating x
                y_up = y_old - lr*dfdx_y(x_old,y_old); # updating y
                x_hist = np.append(x_hist,x_up); # appending updated x to history array
                y_hist = np.append(y_hist,y_up); # appending updated y to history array
            # defining figure and size
            plt.figure(figsize=(8,6))
            # plotting 2D contour plots
            plt.contour(X, Y, fxy,100);
            # plotting history of updates
            plt.plot(x_hist,y_hist,'k')
            # plotting optimal value
            plt.plot(x_hist[-1],y_hist[-1],color='r',marker='x')
            # plotting initial value
            plt.plot(x_hist[0],y_hist[0],color='r',marker='o')
            # title
            plt.title("lr= "+str(lr)+", x_init= "+str(x_init)+", y_init= "+str(y_init)
                      +"\n f(x,y) is min at x="+str(x_hist[-1])+" and y="+str(y_hist[-1]))
```

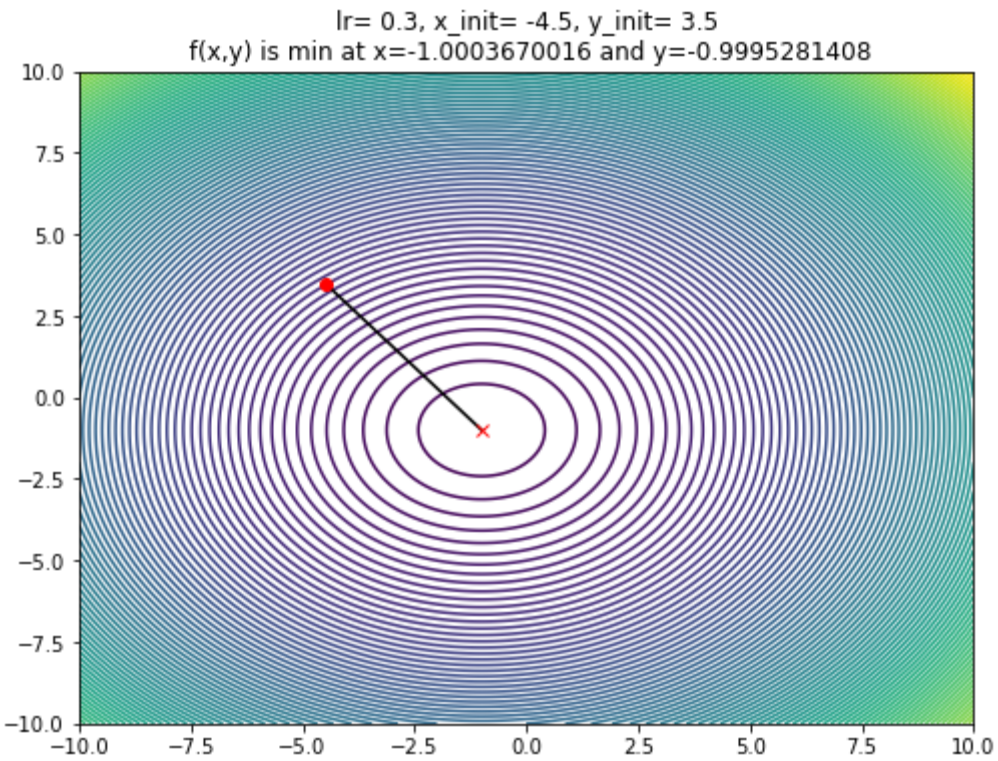

200
150
z
100
50
0
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

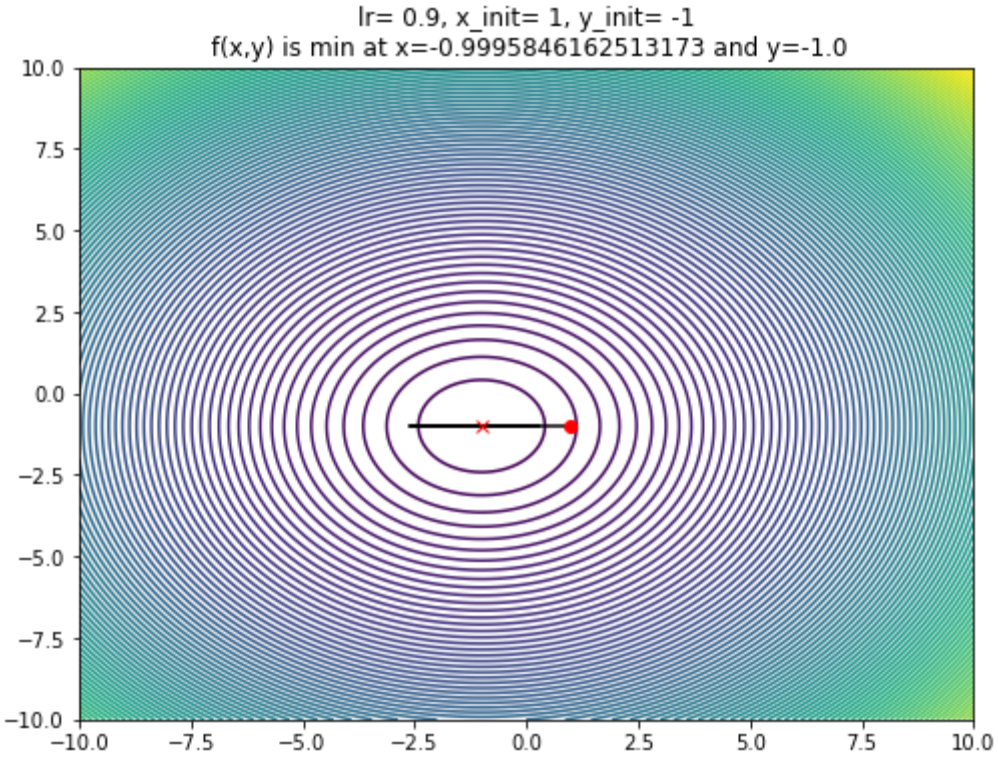

200
150
z
100
50
0
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

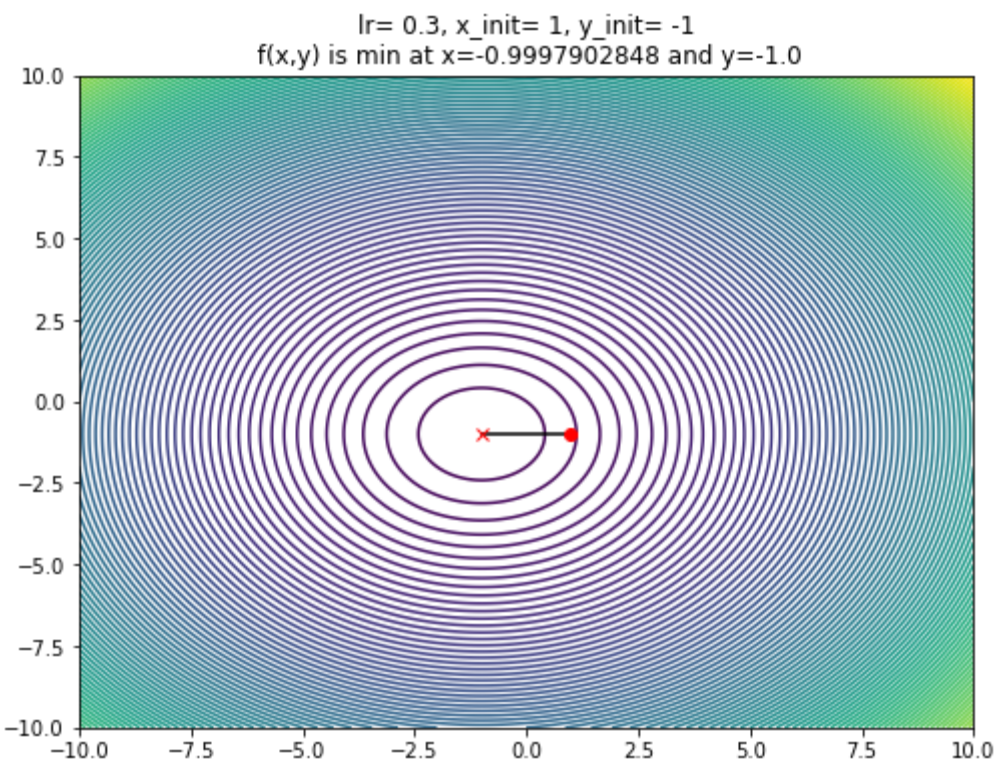

lr= 0.9, x_init= -4.5, y_init= -1
f(x,y) is min at x=-1.0004652297985248 and y=-1.0
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

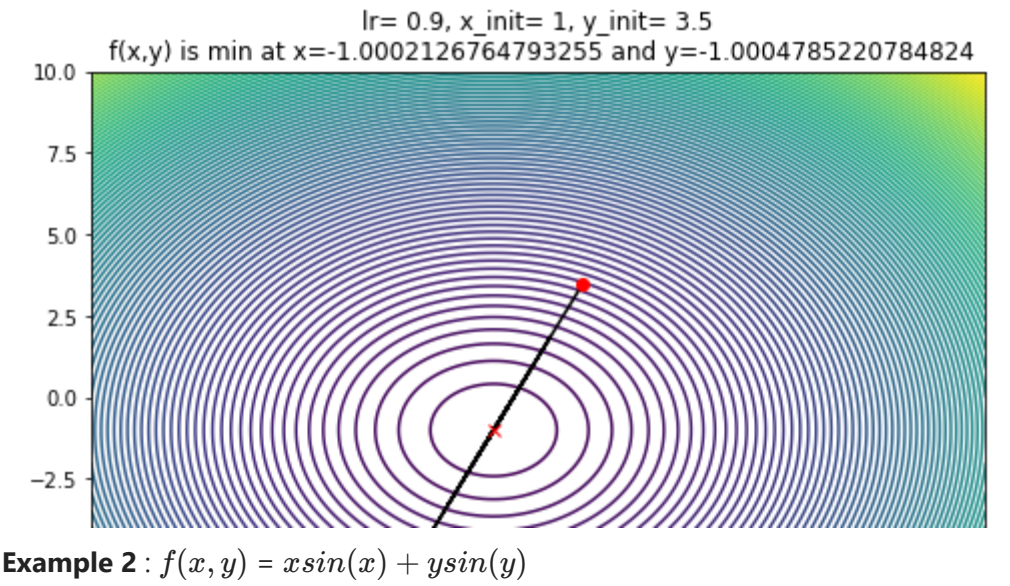

lr= 0.3, x_init= -4.5, y_init= -1
f(x,y) is min at x=-1.0003670016 and y=-1.0
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


lr= 0.9, x_init= -4.5, y_init= 3.5
f(x,y) is min at x=-0.9996278161611801 and y=-1.0004785220784824
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


lr= 0.3, x_init= -4.5, y_init= 3.5
f(x,y) is min at x=-1.0003670016 and y=-0.9995281408
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0







Example 2 : $f(x, y) = xsin(x) + ysin(y)$

Gradient Descent Method :

Follow the below steps and write your code in the block below

1. Generate x and y, 1000 data points from -10 to 10
2. Generate and Plot the function $f(x, y) = xsin(x) + ysin(y)$
3. Initialize the starting point (x_{init},y_{init}) and learning rate (λ)
4. Use Gradient descent algorithm to compute value of x and y at which the function $f(x, y)$ is minimum
5. Also vary the learning rate and initialisation point and plot your observations

Logic for gradient descent

For functions with m variables

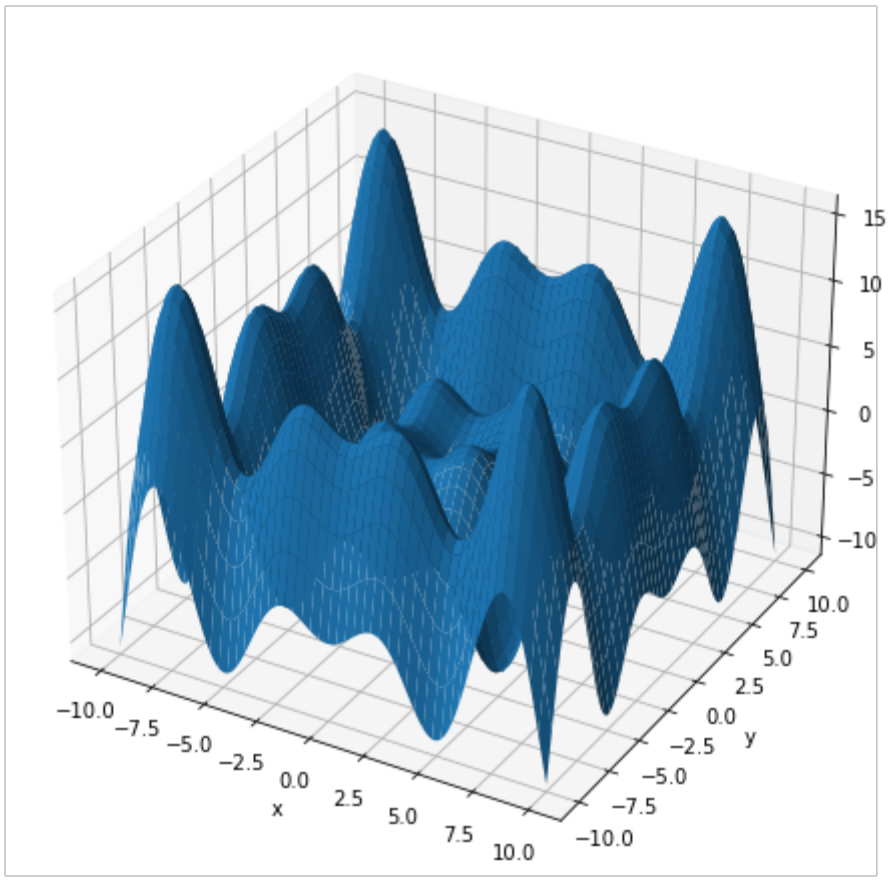
For a multivariate function $f(x_1, x_2, \ldots, x_n)$, minimum/maximum is at a point where $\frac{\partial f}{\partial x_i} = 0 \, \forall \, i = 1, 2, \ldots, n$. If $f(x_1, x_2, \ldots, _n)$ is convex, then we get minimum.

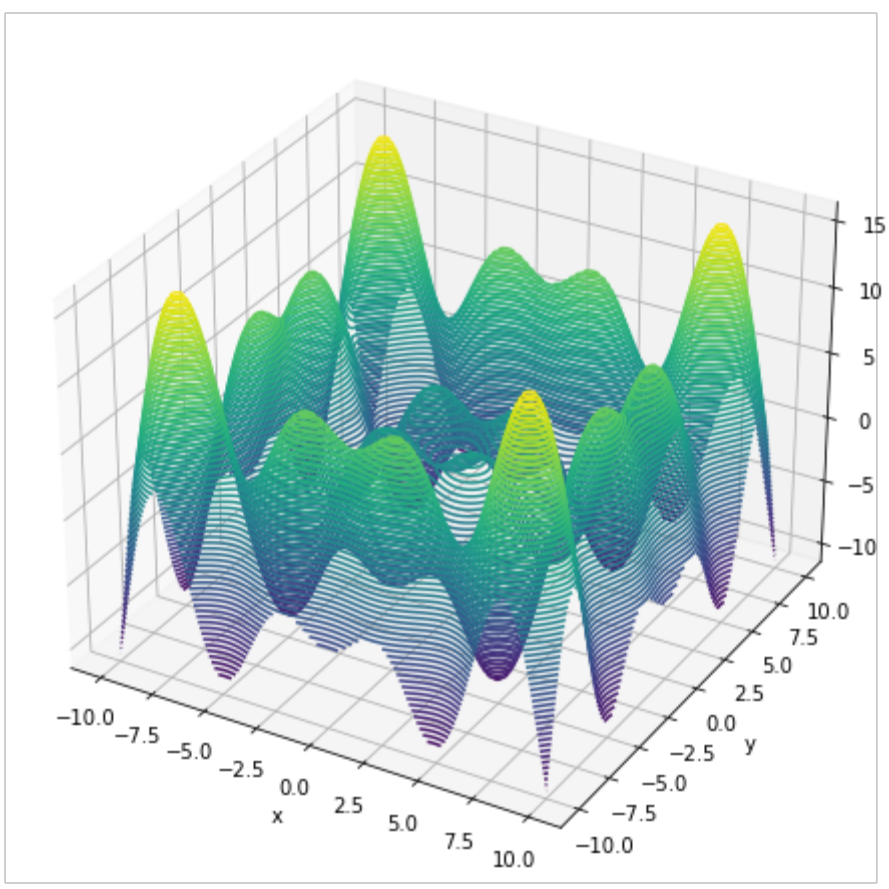
In the code, we randomly initialize starting points $x_{i,init} \, \forall \, i = 1, 2, \ldots, n$ and we update it according to the formula given above

$x_{i,updt} = x_{i,old} - \lambda \left(\frac{\partial f}{\partial x_i} f(x_1, x_2, \ldots, x_n) \mid x_i = x_{i,old}\right)$ [$x_{i,old} = x_{i,init}$ for first update]. As we keep updating, we keep getting closer and closer to $\frac{\partial f}{\partial x_i} = 0$. Ideally we would want $\frac{\partial f}{\partial x_i} = 0$, but keeping a strict condition of getting $\frac{\partial f}{\partial x_i} = 0$ may make the code run without converging. So, we

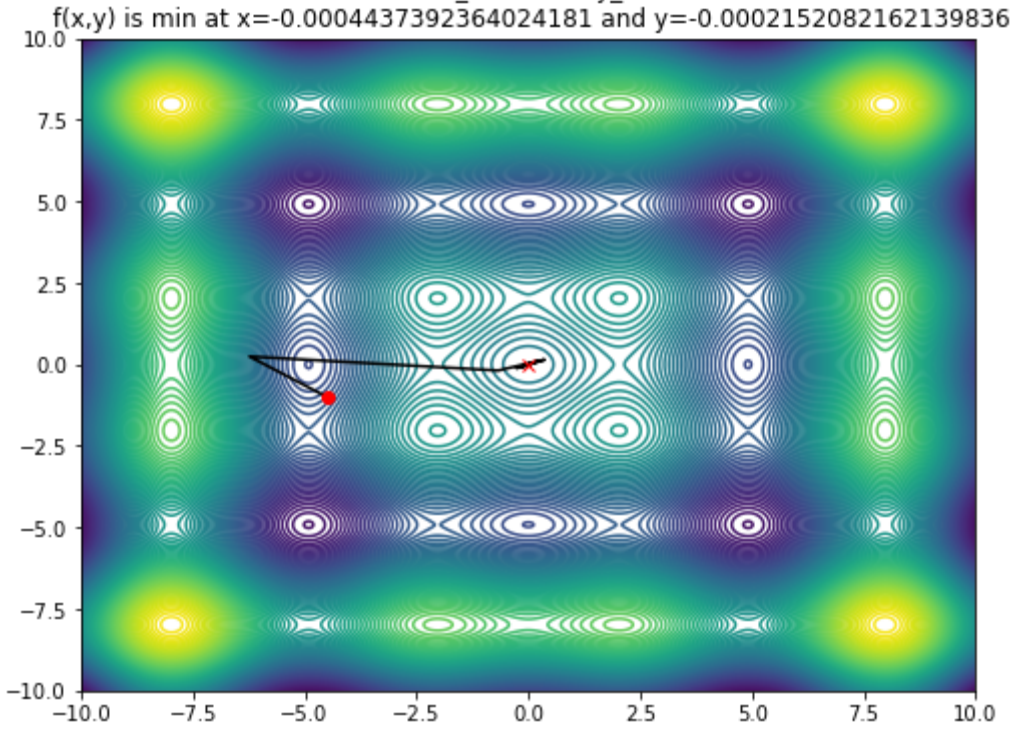
In [5]:

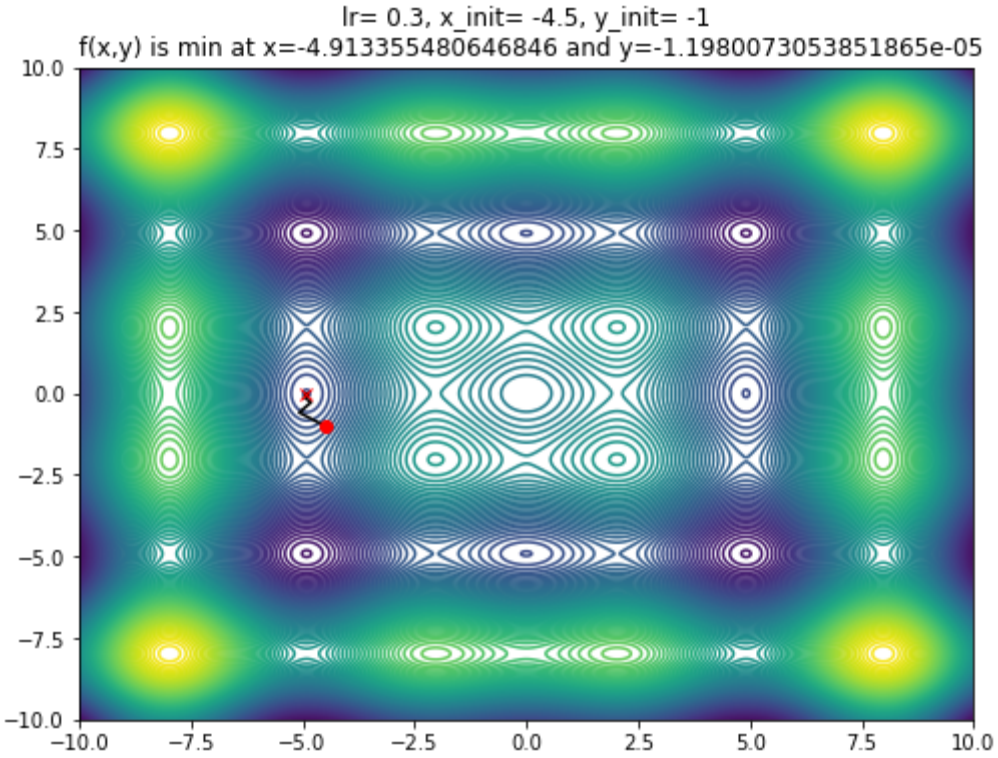
```
## Write your code here (Ignore the warning)
# defining f(x,y)
def f (x,y):
    return x*np.sin(x)+y*np.sin(y)
# defining partial differtiations
def dfdx_x (x,y):
    return np.sin(x)+x*np.cos(x)
def dfdx_y (x,y):
    return np.sin(y)+y*np.cos(y)
# Generating x and y, 1000 data points from -10 to 10
x = np.linspace(-10,10,1000);
y = np.linspace(-10,10,1000);
# meshgrid for plotting 3D plots
X,Y = np.meshgrid(x,y);
# generating f(x,y)
fxy = f(X,Y);
# plotting Surface plot
plt.figure(figsize=(8,8))
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, fxy)
ax.set_xlabel('x');
ax.set_ylabel('y');
ax.set_zlabel('z');
#plotting contour 3D plot
plt.figure(figsize=(8,8))
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, fxy,80)
ax.set_xlabel('x');
ax.set_ylabel('y');
ax.set_zlabel('z');
# array of initial values of x ,y
x_init_arr = [-4.5,1];
y_init_arr = [-1,3.5];
# array of different values of learning rate
lr_arr = [0.9,0.3];
# looping over inital points and learning rates
for x_init in x_init_arr:
    for y_init in y_init_arr:
        for lr in lr_arr:

            x_hist = np.array([]); # history array to store all values of x in each update
            y_hist = np.array([]); # history array to store all values of y in each update
            x_hist = np.append(x_hist,x_init); # appending initial value to array
            y_hist = np.append(y_hist,y_init); # appending initial value to array
            # loop to get updates till convergence
            # logic is explained in markdown above
            while (dfdx_x(x_hist[-1],y_hist[-1])>=0.001 or dfdx_x(x_hist[-1],y_hist[-1])<=-0.001
                   or dfdx_y(x_hist[-1],y_hist[-1])>=0.001 or dfdx_y(x_hist[-1],y_hist[-1])<=-0.001):
                x_old = x_hist[-1] # old x before update
                y_old = y_hist[-1] # old y before update
                x_up = x_old - lr*dfdx_x(x_old,y_old); # updating x
                y_up = y_old - lr*dfdx_y(x_old,y_old); # updating y
                x_hist = np.append(x_hist,x_up); # appending updated x to history array
                y_hist = np.append(y_hist,y_up); # appending updated y to history array
            # defining figure and size
            plt.figure(figsize=(8,6))
            # plotting 2D contour plots
            plt.contour(X, Y, fxy,100);
            # plotting history of updates
            plt.plot(x_hist,y_hist,'k')
            # plotting optimal value
            plt.plot(x_hist[-1],y_hist[-1],color='r',marker='x')
            # plotting initial value
            plt.plot(x_hist[0],y_hist[0],color='r',marker='o')
            # title
            plt.title("lr= "+str(lr)+", x_init= "+str(x_init)+", y_init= "+str(y_init)
                      +"\n f(x,y) is min at x="+str(x_hist[-1])+" and y="+str(y_hist[-1]))
```

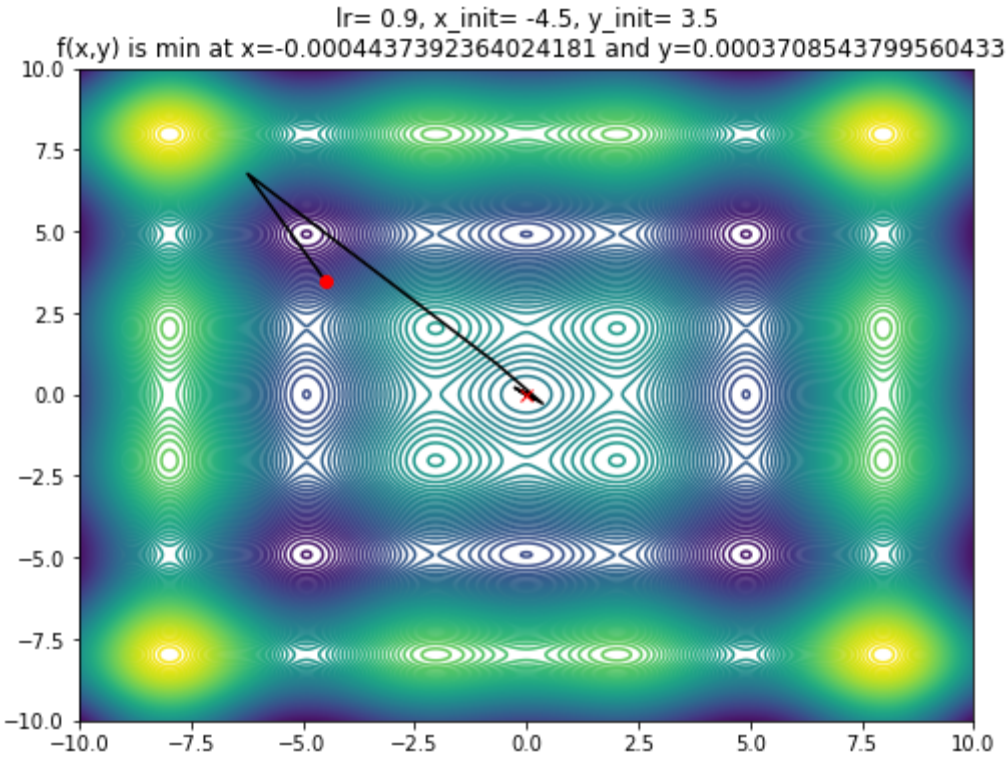


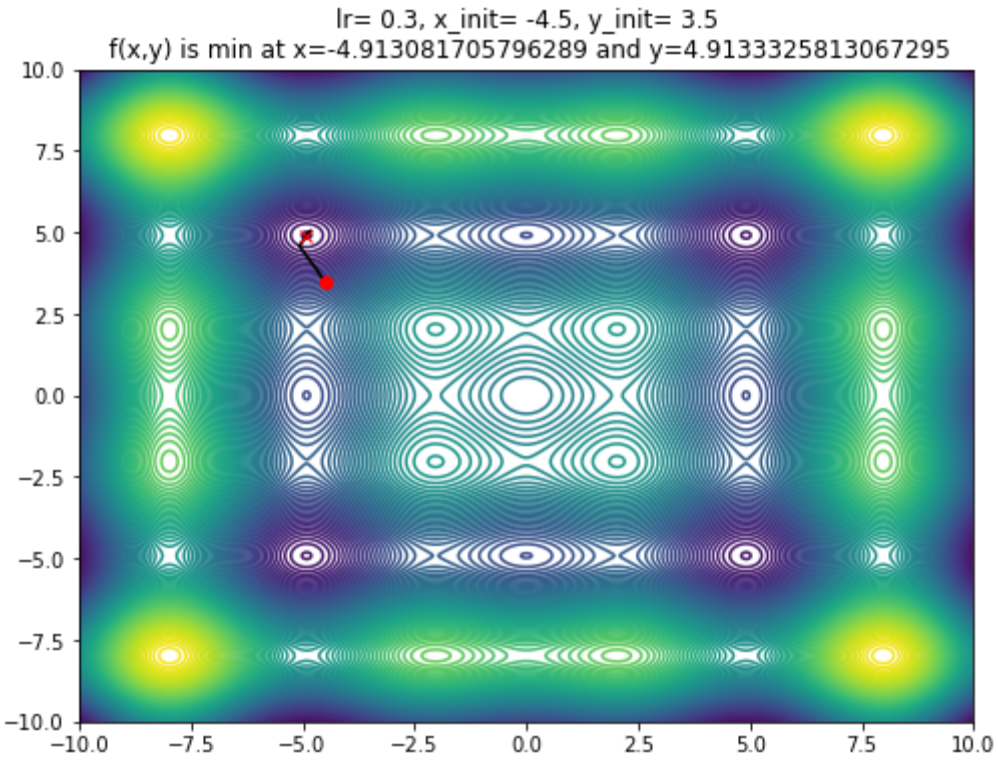


lr= 0.9, x_init= -4.5, y_init= -1
f(x,y) is min at x=-0.0004437392364024181 and y=-0.0002152082162139836

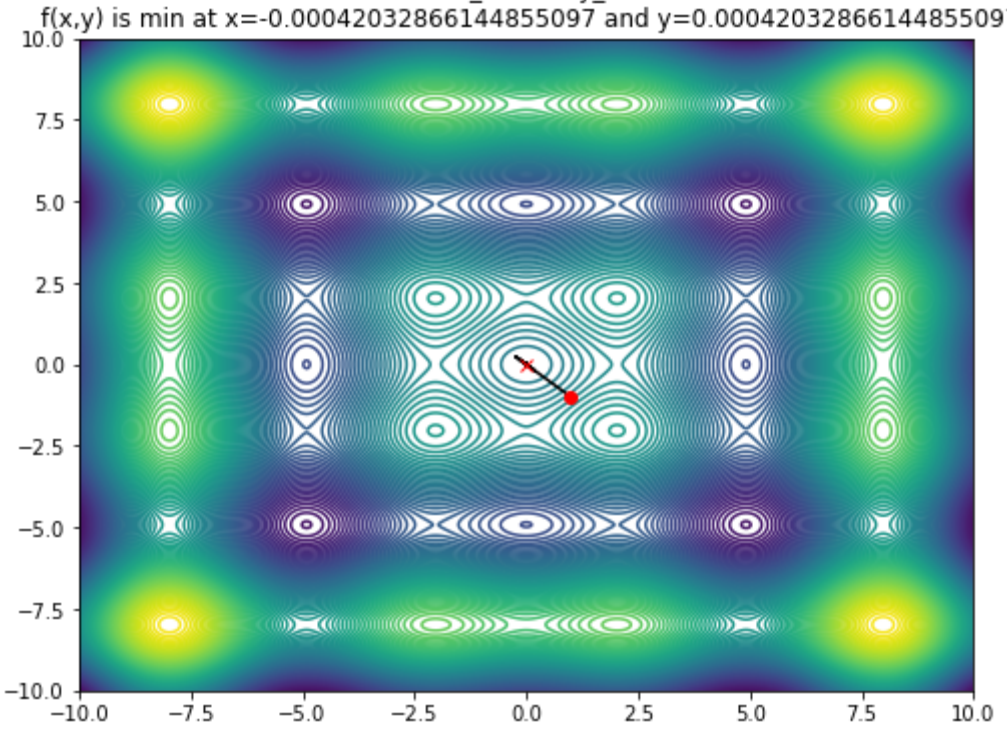



lr= 0.3, x_init= -4.5, y_init= -1
f(x,y) is min at x=-4.913355480646846 and y=-1.1980073053851865e-05
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

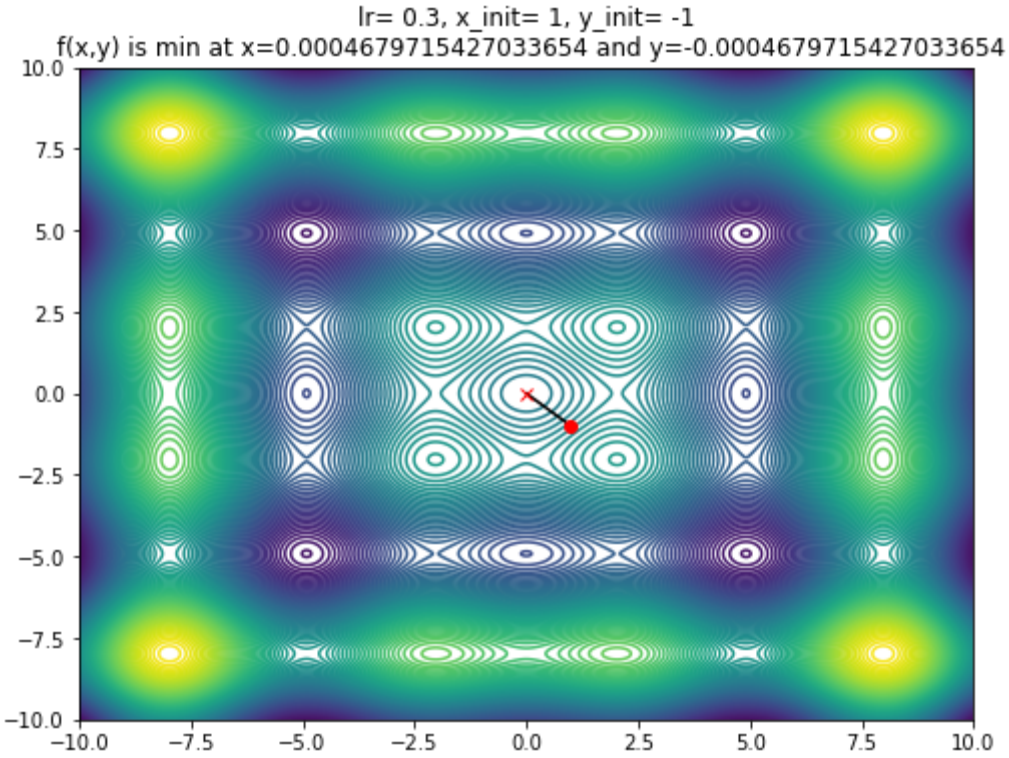

lr= 0.9, x_init= -4.5, y_init= 3.5
f(x,y) is min at x=-0.0004437392364024181 and y=0.0003708543799560433
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


lr= 0.3, x_init= -4.5, y_init= 3.5
f(x,y) is min at x=-4.913081705796289 and y=4.9133325813067295
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

lr= 0.9, x_init= 1, y_init= -1
f(x,y) is min at x=-0.00042032866144855097 and y=0.00042032866144855097



lr= 0.3, x_init= 1, y_init= -1
f(x,y) is min at x=0.0004679715427033654 and y=-0.0004679715427033654



lr= 0.9, x_init= 1, y_init= 3.5
f(x,y) is min at x=-0.00026901028486798584 and y=-0.00046356804965901207