Assignment 1 - Vanshita Sharma Kumar

Assignment 1: Optimization

Goal: Get familiar with gradient-based and derivative-free optimization by implementing these methods and applying them to a given function.

In this assignment we are going to learn about **gradient-based** (GD) optimization methods and **derivative-free optimization** (DFO) methods. The goal is to implement these methods (one from each group) and analyze their behavior. Importantly, we aim at noticing differences between these two groups of methods.

Here, we are interested in minimizing the following function:

$$f(\mathbf{x}) = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$$

in the domain $\mathbf{x} = (x_1, x_2) \in [-100, 100]^2$ (i.e., $x_1 \in [-100, 100]$, $x_2 \in [-100, 100]$).

In this assignemnt, you are asked to implement:

1. The gradient-descent algorithm.
2. A chosen derivative-free algorithm. *You are free to choose a method.*

After implementing both methods, please run experiments and compare both methods. Please find a more detailed description below.

1. Understanding the objective

Please run the code below and visualize the objective function. Please try to understand the objective function, what is the optimum (you can do it by inspecting the plot).

If any code line is unclear to you, please read on that in numpy or matplotlib docs.

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

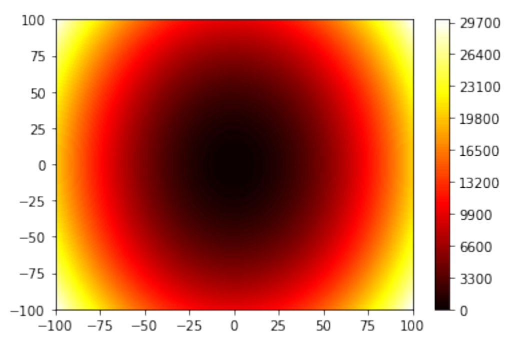
In [2]:
```
# PLEASE DO NOT REMOVE!
# The objective function.
def f(x):
    return x[:,0]**2 + 2*x[:,1]**2 -0.3*np.cos(3.*np.pi*x[:,0])-0.4*np.cos(4.*np.pi*x[:,1])+0.7
```

In [3]:
```
# PLEASE DO NOT REMOVE!
# Calculating the objective for visualization.
def calculate_f(x1, x2):
  f_x = []
  for i in range(len(x1)):
    for j in range(len(x2)):
      f_x.append(f(np.asarray([[x1[i], x2[j]]])))

  return np.asarray(f_x).reshape(len(x1), len(x2))
```

In [4]:
```
# PLEASE DO NOT REMOVE!
# Define coordinates
x1 = np.linspace(-100., 100., 400)
x2 = np.linspace(-100., 100., 400)

# Calculate the objective
f_x = calculate_f(x1, x2).reshape(len(x1), len(x2))
```

In [5]:
```
# PLEASE DO NOT REMOVE!
# Plot the objective
plt.contourf(x1, x2, f_x, 100, cmap='hot')
plt.colorbar()
```

Out[5]:
```
<matplotlib.colorbar.Colorbar at 0x7fe51aeafa60>
```



2. The gradient-descent algorithm

First, you are asked to implement the gradient descent (GD) algorithm. Please take a look at the class below and fill in the missing parts.

NOTE: Please pay attention to the inputs and outputs of each function.

NOTE: To implement the GD algorithm, we need a gradient with respect to $\mathbf{x}$ of the given function. Please calculate it on a paper and provide the solution below. Then, implement it in an appropriate function that will be further passed to the GD class.

Question 1 (0-1pt): What is the gradient of the function $f(\mathbf{x})$?

Answer:

gradient of f(x1) = 2x1 + 0.9sin(3pi*x1)pi

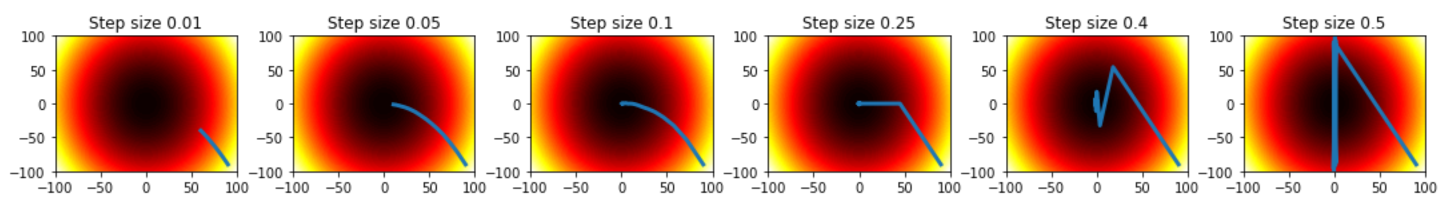
gradient of f(x2) = 4x2 + 1.6sin(4pi*x2)pi

In [6]:
```
#=========
# GRADING:
# 0
# 0.5pt - if properly implemented and commented well
#=========
# Implement the gradient for the considered f(x).
def grad(x):
  #------
  # PLEASE FILL IN:
  # ...

    x1 = 2*x[:,0]+0.3*np.sin(3.*np.pi*x[:,0])*3*np.pi
    x2 = 4*x[:,1]+0.4*np.sin(4*np.pi*x[:,1])*4*np.pi

    grad = np.concatenate((x1, x2),axis=None)


    return grad
```

In [7]:
```
#=========
# GRADING:
# 0
# 0.5pt if properly implemented and commented well
#=========
# Implement the gradient descent (GD) optimization algorithm.
# It is equivalent to implementing the step function.
class GradientDescent(object):
    def __init__(self, grad, step_size=0.1):
        self.grad = grad
        self.step_size = step_size

    def step(self, x_old):
    #------
    # PLEASE FILL IN:
    # ...
        x_new = x_old-step_size*grad(x_old)
    #------
        return x_new
```

In [8]:
```
# PLEASE DO NOT REMOVE!
# An auxiliary function for plotting.
def plot_optimization_process(ax, optimizer, title):
  # Plot the objective function
    ax.contourf(x1, x2, f_x, 100, cmap='hot')

  # Init the solution
    x = np.asarray([[90., -90.]])
    x_opt = x
  # Run the optimization algorithm
    for i in range(num_epochs):
        x = optimizer.step(x)
        x_opt = np.concatenate((x_opt, x), 0)

    ax.plot(x_opt[:,0], x_opt[:,1], linewidth=3.)
    ax.set_title(title)
```

In [9]:
```
# PLEASE DO NOT REMOVE!
# This piece of code serves for the analysis.
# Running the GD algorithm with different step sizes
num_epochs = 20 # the number of epochs
step_sizes = [0.01, 0.05, 0.1, 0.25, 0.4, 0.5] # the step sizes

# plotting the convergence of the GD
fig_gd, axs = plt.subplots(1,len(step_sizes),figsize=(15, 2))
fig_gd.tight_layout()

for i in range(len(step_sizes)):
  # take the step size
    step_size = step_sizes[i]
  # init the GD
    gd = GradientDescent(grad, step_size=step_size)
  # plot the convergence
    plot_optimization_process(axs[i], optimizer=gd, title='Step size ' + str(gd.step_size))
```



Question 2 (0-0.5pt): Please analyze the plots above and comment on the behavior of the gradient-descent for different values of the step size.

Answer: x_new is the product of the gradient and step_size. We can see the progress to be slow for step_sizes 0.01 and 0.05, unfortunately, these step_sizes fail to reach the global minima point even though the algorithm is progressing in the right direction. At step_sizes 0.1 and 0.25, the line reaches the minima quickly but soon escalates in step_size 0.4 and 0.5. This jump from 0.25 to 0.5 shows that the algorithm keeps oscillating around the minima, where it has the potential of never reaching the minima again, hence the optimum step_size would be around 0.1 and 0.25.

Question 3 (0-0.5pt): Can we do something about the step size equal 0.01? What about the step size equal 0.5?

Answer: We can make a few altercations to our code so that at step_sizes 0.01 and 0.5 can reach the global minimum. For step sizes 0.01 we can change the number of epochs to 150 or more. Because the gradient reduces as we reach closer to global minima, the number of iterations which is required to reach the goal also increases exponentially.

For step size 0.5 we can alter the formula by dividing "self.step_size*gradient[:,0]" by the absolute value of "gradient[:,0]”. This implies that the coordinates will be changed by 0.5 for each epoch in the direction of the gradient. We can also divide "self.step_size*gradient[:,0]" by 4 so the algorithm moves at a fewer distance at each iteration.

3. The derivative-free optimization

In the second part of this assignment, you are asked to implement a derivative-free optimziation (DFO) algorithm. Please notice that you are free to choose any DFO method you wish. Moreover, you are encouraged to be as imaginative as possible! Do you have an idea for a new method or combine multiple methods? Great!

Question 4 (0-0.5-1-1.5-2-2.5-3pt): Please provide a description (a pseudocode) of your DFO method here.

NOTE (grading): The more complex the method, the higher the score! Please keep it in mind during developing your algorithm. TAs will also check whether the pseudocode is correct.

Answer:

Create DFO class

1. Initialise object with attributes:

   obj_fun: Objective function

   step_size: Number of steps taken

   min_val: setting the bounds to minimum value of -5

   max_val: setting the bounds to maximum value of 5

1. Create step function(self, x_old):

   candidate = x_old + the uniform distribution for values of x * the step_size

   evaluate candidate = obj_fun(candidate)

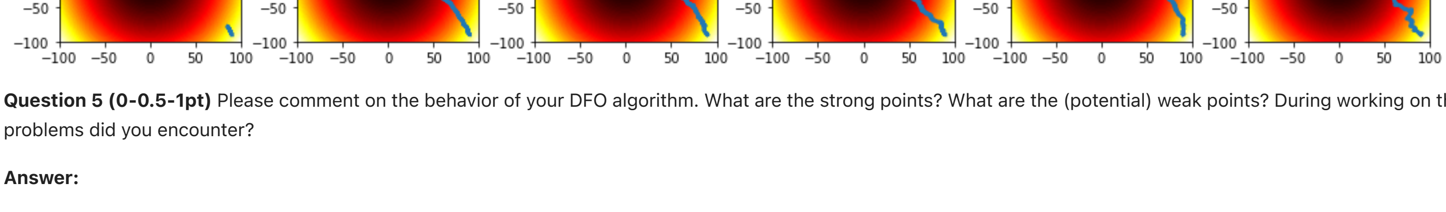
   if the candidate evaluation <= obj_fun(x_old)

   `then return candidate`

   otherwsie return x_old

In [10]:
```
#=========
# GRADING: 0-0.5-1-1.5-2pt
# 0
# 0.5pt the code works but it is very messy and unclear
# 1.0pt the code works but it is messy and badly commented
# 1.5pt the code works but it is hard to follow in some places
# 2.0pt the code works and it is fully understandable
#=========
# Implement a derivative-free optimization (DFO) algorithm.
# REMARK: during the init, you are supposed to pass the obj_fun and other objects that are necessary in your method.

class DFO(object):

    def __init__(self, obj_fun, step_size=0.1, min_val=-5, max_val=5):# adjacency_list):
        self.obj_fun = obj_fun
        self.step_size = step_size
        self.min_val= min_val
        self.max_val = max_val

    def step(self, x_old):
        # this a uniform distribution for the points of x, so we can evalue later if candiate is <= x_old
        candidate = x_old + np.random.uniform(low=self.min_val, high=self.max_val, size = x_old.shape[1]) * self.step_size
        # evaluate candidate point
        candidte_eval = self.obj_fun(candidate) # comapring the value found in old f(x) to the new value found in f(x)
        # check if we should keep the new point
        if candidte_eval <= self.obj_fun(x_old):
            return(candidate)
        return x_old
```

In [11]:
```
# PLEASE DO NOT REMOVE!
# Running the DFO algorithm with different step sizes
num_epochs = 1000 # the number of epochs (you may change it!)
step_sizes = [0.01, 0.05, 0.1, 0.25, 0.4, 0.5]

fig_dfo, axs = plt.subplots(1, len(step_sizes), figsize=(15, 2))
fig_dfo.tight_layout()

for i in range(len(step_sizes)):
    # take the step size
    step_size = step_sizes[i]
  # init the DFO class
    dfo = DFO(f, step_size=step_size)
  # plot the convergence
    plot_optimization_process(axs[i], optimizer=dfo, title='Step size ' + str(dfo.step_size))
```



Question 5 (0-0.5-1pt) Please comment on the behavior of your DFO algorithm. What are the strong points? What are the (potential) weak points? During working on the algorithm, what kind of problems did you encounter?

Answer:

The respective code above is the representation of the stochastic hill-climbing algorithm. The code has 3 main steps. The first step is finding our x_new value or the candidate value. To find candidate we have applied the gradient equation of the line. We declare our constant value “c” to be x_old. x_old is the value found by the algorithm supplied. Our “m” is the uniform distribution for the points of x. We find any random point and multiply it with the step_size (the “x” value). After our candidate value is found, in step 2 we evaluate x_new via the “candidate_eval” variable. The candidate_eval implements the value of the candidate in the given objective function. In our third step, the candidate is then compared to the previous point found, which was our x_old. In hill climbing if the new point is less than or equal to the previous point, we take the step further to the new point, this logic is applied in our code with the use of the if-statement, evaluating if we should return candidate or x_old.

While programming the hill-climbing algorithm I had encountered an issue where the point would deviate from the centre, it would often get stuck in local minima as it failed to find the next best point. This issue was then resolved by changing the bounds to individually declaring min and max values, later finding the best points within a declared range of values.

The strengths of the algorithm are that it is simple and straightforward. Because of its simplicity the code requires less time to compute the best possible path, because differentiation of some functions may be costly or unknown Since it moves to other points if the objective function yields the same value.

4. Final remarks: GD vs. DFO

Eventually, please answer the following last question that will allow you to conclude the assignment draw conclusions.

Question 6 (0-0.5pt): What are differences between the two approaches?

Answer: in gradient descend algorithm or in derivative algorithms, it can get stuck in local minima, because the local minima are zero, and requires computation calculation which can be hard to compute, and to calculate the derivative of a function at each new point.

in DFO, we can apply these algorithms easily, they can be used in functions without easy derivatives or non-differentiable functions. DFO also do not get stuck at local minima that often, making them quite useful.

Question 7 (0-0.5): Which of the is easier to apply? Why? In what situations? Which of them is easier to implement in general?

Answer: the DFO algorithms are easier to apply, but when speaking mathematically it depends if the function is differentiable, with derivative-based algorithms it is best to use when you aim to find local minima and maxima, like gradient descend, however they do often get stuck. For functions with just one minima, gradient descent is the best choice. But if the function is non-differentiable or not easily differentiable, gradient descent can't be used. Furthermore, differential functions are also hard to compute because for each new point a gradient must be calculated. In general, DFO is easier to compute and implement, as real-world functions are too complicated.