Assignment 1

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

In [2]:
```
# PLEASE DO NOT REMOVE!
# The objective function.
def f(x):
    return (
        x[:, 0] ** 2
        + 2 * x[:, 1] ** 2
        - 0.3 * np.cos(3.0 * np.pi * x[:, 0])
        - 0.4 * np.cos(4.0 * np.pi * x[:, 1])
        + 0.7
    )
```

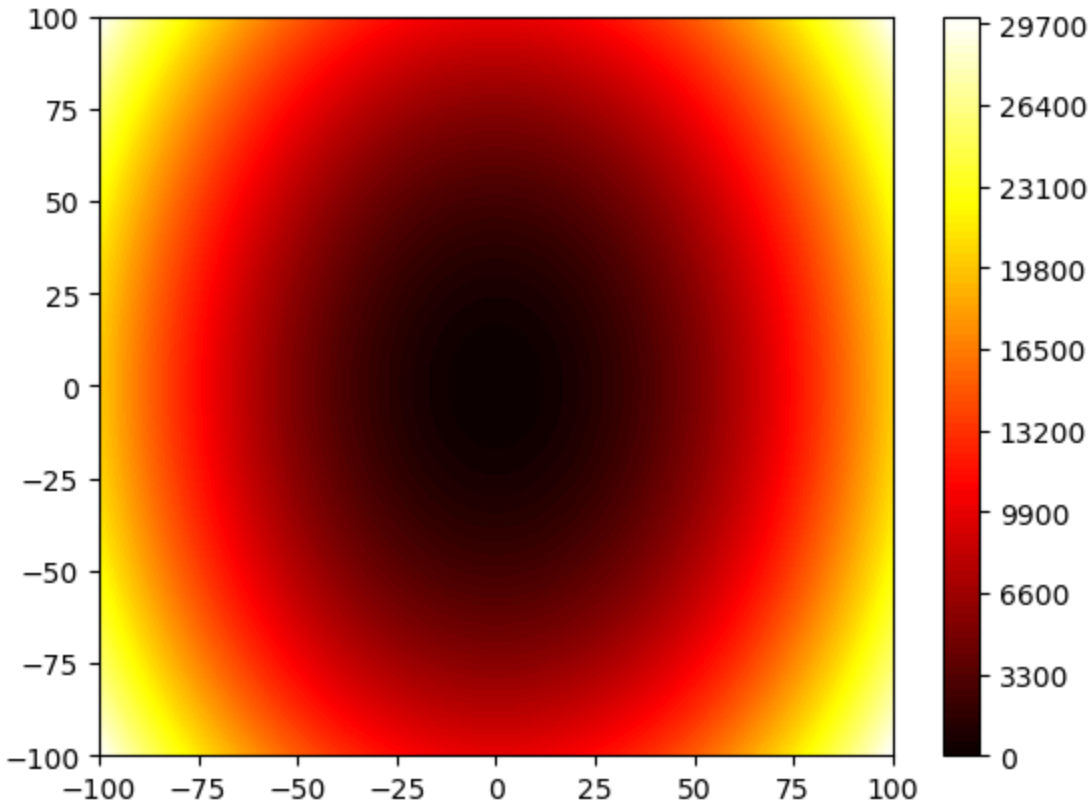
In [3]:
```
# PLEASE DO NOT REMOVE!
# Calculating the objective for visualization.
def calculate_f(x1, x2):
    f_x = []
    for i in range(len(x1)):
```

```
        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(-100.0, 100.0, 400)
x2 = np.linspace(-100.0, 100.0, 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 0x112a82f50>
```



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 $f(\mathbf{x})$? Please fill below both the mathematical expression and within the code.

Answer:

$$\nabla_{\mathbf{x}_1} f(\mathbf{x}) = 2x_1 + 0.9 * pi * sin(3 * pi * x_1)$$
$$\nabla_{\mathbf{x}_2} f(\mathbf{x}) = 4x_2 + 1.6 * pi * sin(4 * pi * x_2)$$

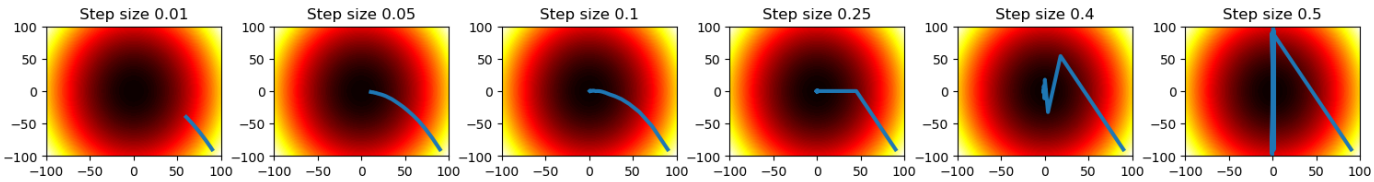
In [6]:
```
# =========
# GRADING:
# 0
# 0.5pt - if properly implemented and commented well
# =========
# Implement the gradient for the considered f(x).
def grad(x):
    #This is the calculated gradient of x1, written using the same style as the objectiv
    x1 =  2*x[0,0] + 0.9 * np.pi * np.sin(3*np.pi*x[0,0])
    #This is the calculated gradient of x2, written using the same style as the objectiv
    x2 =  4*x[0,1] + 1.6 * np.pi * np.sin(4*np.pi*x[0,1])
    #This is the vector of the derivative
    grad = np.asarray([x1, x2])
    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([[90.0, -90.0]])
    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=3.0)
    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. What happens in the small and large step sizes and what is the optimum step size?

Answer: As can be seen when the step size is too small (as can be seen when the stepsize is 0.01) or too large (as can be seen when the stepsize is 0.5 then the global optimum is not found. For the one that do end up finding the minimum, the stepsizes of 0.05 and 0.1 behave similarly, with the gradually curving towards the minimum. In contrast to this the other stepsizes of 0.25 and 0.4 are more erratic in their steps, taking sharp steps before converging on the minimum, with this being more the case with a step size of 0.4 especially.

Question 3 (0-0.5pt): How can we improve the convergence when the step size equals 0.01? What about when the step size equals 0.5?

Answer: For convergence in stepsize of 0.01 we can increase convergence by increasing the number of epochs so that eventually the plot will reach the global optimum. On the other hand for large stepsize we can remedy this by taking the average of the line bouncing between extremes i.e. if the line bounces between values of 100 and -100 then the minimum must be at 0.

Could also just add more epochs.

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): Please keep in mind: start simple, make sure your approach works. You are encouraged to use your creativity and develop more complex approaches that will influence the grading. TAs will also check whether the pseudocode is correct.

Answer: Local Search

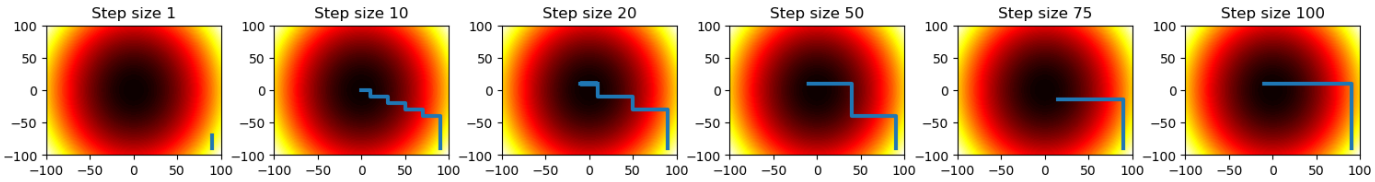
Input: Initialize x with a random solution. Until STOP is called (e.g. number of iterations performed, or adequate fitness reached),repeat the following:

1. Find the best solution for the neighborhood of the current best solution i.e. in the range (0, Current Solution) .
2. Set the new solution as the best current solution
3. Peturb/Add some noise to the best current solution and find the best solution in the range (previous solution, pertrubed solution)

In [19]:
```
#=========
# 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 a
class DFO(object):
    def __init__(self, obj_fun, step_size):
        self.obj_fun = obj_fun
        # PLEASE FILL IN: You will need some other variables
        self.step_size = step_size

    ## PLEASE FILL IN IF NECESSARY
    ## Please remember that for the DFO you may need extra functions.
    #def ...

    # This function MUST be implemented.
    # No additional arguments here!
    def step(self, x_old):
    ## PLEASE FILL IN.
        x_step = [step_size, 0, -step_size, 0, 0]
        y_step = [0, step_size, 0, -step_size, 0]
        next_steps = []
        cor_values = []
        for i in range(len(x_step)):
            next_step = x_old + np.array([x_step[i], y_step[i]])
            next_steps.append(next_step)
        for next_step in next_steps:
            cor_value = self.obj_fun(next_step)
            cor_values.append(cor_value)
        #select neighbor with minimum objective function value
        x_new = next_steps[np.argmin(cor_values)]
        return x_new
```

In [21]:
```
# PLEASE DO NOT REMOVE!
# Running the DFO algorithm with different step sizes
num_epochs = 20 # the number of epochs (you may change it!)

## PLEASE FILL IN
## Here all hyperparameters go.
## Please analyze at least one hyperparameter in a similar manner to the
## step size in the GD algorithm.
# analyzed step size parameter
```

```
step_sizes = [1, 10, 20, 50, 75, 100]

## plotting the convergence of the DFO
## Please uncomment the two lines below, but please provide the number of axes (replace
fig_dfo, axs = plt.subplots(1, len(step_sizes), figsize=(15, 2))
fig_dfo.tight_layout()

# the for-loop should go over (at least one) parameter(s) (replace HERE appriopriately)
# and uncomment the line below
#iterate over step_sizes to get step_sie and then dfo of each one
for x in range(len(step_sizes)):
  ## PLEASE FILL IN
    step_size = step_sizes[x]
    dfo = DFO(f, step_size)
  # plot the convergence
  # please change the title accordingly!
    plot_optimization_process(axs[x], optimizer=dfo, title='Step size '+ str(dfo.step_si
```



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: Iterated Local Search is a DFO that takes elements of random search and optimizes it by making it less likely to get stuck in local optimum. This is accomplished in the 3rd step by adding noise to the previous best solution, thus addinf an element of stochasity making it less likely to be trapped in local optimum. On top of this it is also easy to implement, require (almost) no extra knowledge i.e. calculus and works for both continuous and discrete problem. However one of its weaknesses is that DFO are very slow, especially in comparison with gradient descent.

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: The gradient descent can only work for continuous spaces and smooth and differentiable functions.

The DFO can work for both continuous or discrete spaces.

DFOs also need to evaluate every point, which can be expensive whereas gradient descent does not

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: Due to the fact that it requires almost no additional knowledge and/or calculation and that it is versatile enough to work for both continuous and discrete problems, it is clear that in most cases the

derivative free approach is the easiest to use. However when implementing, it is more cost effective to use gradient descent as opposed to DFO methods