.

1.Consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined as follows,

$$f(x_1, x_2) = -2x_1^2 + x_1x_2^2 + 4x_1^4.$$

Find all the stationary points of this function, and for each such point say whether thepoint is a local minimum/local maximum/saddle point, with reasoning.

Consider the following update rule,

$$x_{k+1} := x_k - \frac{0.1}{k+1}\nabla f(x_k).$$

Starting at x_0, run the iterate until you reach a point x_{k*} such that $\|x_{k*} - x_{k-1}\|_2 \leq 10^{-10}$. Report the value of k^* when the starting point is (i) $x_0 = [0.7, 0.005]$, (ii) $x_0 = [0.3, 0.005]$, (iii) $x_0 = [-0.1, 0.005]$, and (iv) $x_0 = [-0.7, 0.005]$.

2.Backtracking. Backtracking is a form of inexact line search in which a step size is determined at each step which satisfies the Armino-Goldstein condition. Given constants $\alpha, \beta \in (0, 1)$, at each step of the algorithm, if the current point is $x \in \mathbb{R}^d$, the direction of line search is chosen as $u = -\nabla f(x)$, and for determining the step size, an initial step size $t = 1$ is chosen and is repeatedly updated as $t \leftarrow \beta t$ until $f(x + tu) \leq f(x) + \alpha t \nabla f(x)^T u$ and then x is updated as $x \leftarrow x + tu$. Once the update distance $\|tu\|_2$ for the point x becomes less than ϵ during any epoch, the algorithm is stopped.

The following is the pseudocode for implementing inexact line search with backtracking:

```
i ← 0
u ← −∇f(x)
t ← 1
while i < maxIterations do
    if f(x + tu) ≤ f(x) + αt∇f(x)^T u then
        return x + tu
    end if
    if ||tu||_2 < ϵ then
        return x + tu
    end if
    t ← βt
    i ← i + 1
end while
```

You are given oracles for $f : \mathbb{R} \to \mathbb{R}$, ∇f. Instructions to call the oracle through the script are as follows:

- Choose the oracles required for your setup from the concerned directory.
- For Python users, use the following instructions to call the executable: (Please copy the below carefully and do debugging checks to ensure you're reading the output correctly)
  - someVar = subprocess.run(["full path of the oracle", "<SRNo.>,x"], stdout=subprocess.PIPE).stdout.decode("utf-8")
    For eg., subprocess.run(["./getGradients linux.exe","12345,10"], stdout=subprocess.PIPE).stdout.decode("utf-8")
  - It will return the string "$f(x)$, $\nabla f(x)$" into someVar. For eg., someVar might look like the string "1.25,-1,05".
  - Convert someVar to appropriate floating point numbers.
- For MATLAB, use the following instructions to call the executable:
  - [status,cmdout] = system("full path of the oracle <SRNo.>,x")
    For eg., [status,cmdout] = system("./getGradients windows.exe 12345,10").
  - It will return the string "$f(x)$, $\nabla f(x)$" into cmdout. For eg., cmdout might look like the string "1.25,-1,05".
  - Convert cmdout to appropriate floating point numbers.
- In case using Linux or Mac ensure that you add chmod permissions for executing the .exe file.
- For mac use the following instructions:
  - Run chmod 777 <executable name>
  - You may need to do: Preferences > Security and Privacy > General and allow the executable (app) to run (click "allow").

Now use the provided oracles to implement the following two algorithms:

(a) Gradient descent with inexact line search using backtracking: Use $\alpha = 0.5$, $\beta = 0.1$, $\epsilon = 10^{-3}$ and *maxIterations* = 30. Start from the point $x_0 = 1.0$ and stop the update when you have reached x_k such that $\|\nabla f(x_k)\|_2 \leq 10^{-3}$.

i. (6 points) Write down the value of $f(x)$ after 1, 10, and 20 iterations of gradient descent. If the algorithm has converged before the given iteration, write -1 for the corresponding iteration.

ii. (3 points) Also report the final values of $x, f(x), \nabla f(x)$ after gradient descent terminates.

(b) Heavy Ball Method:

In [Polyak, 1964] Polyak has analysed the convergence of multi-step methods, where the update x_{k+1} depends on multiple previous iterates $x_k, x_{k-1}, \ldots, x_{k-m}$, where $0 < m < k$. According to this, any update rule for x_{k+1} that depends only on x_k is a one-step update. In this exercise we will notice the faster convergence of a two-step update method, also known as the "heavy ball method", compared to the one-step gradient descent update in part (a). This method is inspired by interpretations from Physics, i.e., if each iterate x_k is imagined to be a heavy ball, i.e., an object with mass, it possesses a momentum from the previous update, $x_k - x_{k-1}$, which can be used in our favor. The full momentum update is:

$$x_{k+1} := x_k - \gamma \nabla f(x_k) + \eta \underbrace{(x_k - x_{k-1})}_{\text{momentum}}.$$

Here η is a tunable hyperparameter. Usually the value of η is chosen to be in $(0, 1)$ to scale down the momentum. This update allows us to control oscillations in each update as well as accelerate the update in the regions with low curvature, hence achieving faster convergence rates.

With $\gamma = 0.25$ and $\eta = 0.5$, start from the point $x_0 = 1.0$ and stop the update when you have reached x_k such that $\|\nabla f(x_k)\|_2 \leq 10^{-3}$.

i. (6 points) Write down the value of $f(x)$ after 1, 3, and 5 iterations of the Heavy ball method. If the algorithm has converged before the given iteration, write -1 for the corresponding iteration.

ii. (3 points) Also report the final values of $x, f(x), \nabla f(x)$ after the heavy ball method terminates.