Golden Section & Steepest Descent: **Report**

by Kerim U. Kizil

Due: 28-Feb

1 About the Submission

1.1 Content

Submission folder consists of five different elements:

- **01_Report.pdf:** This is the document that is currently being read. It is intended as the main and guiding document of the submission.
- **02_Code_Initialization.java:** This file includes the Java code for the initialization part of the Homework. It implements the *golden section algorithm* to find the unconstrained minimum of a simple univariate convex function: $\theta(x) = (x-2)^2$.
- **03_Code_Adaptation.java:** This file includes the Java code for the adaptation part of the Homework. The code in *02_Code_Initialization.java* is modified to solve an "optimal stepsize selection" problem. The code also includes the experimental case where we define a function $f : \mathbb{R}^n \to as\mathbb{R}$ as $f(x) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$ to make trials on the coded golden section algorithm.
- **04_Code_SteepestDescent.java:** This file includes the code for the Steepest Descent implementation. I've run the tests for different initial points on this code.
- **05_SteepestDescent_DetailedOutput.txt:** This file includes a detailed algorithmic output for one of the specific runs of the coded Steepest Descent algorithm.

1.2 Testing the Code

The grader can easily test my code on an online Java compiler such as: `https://repl.it/languages/java10`. This would eliminate the need for setting up a Java environment on the grader's PC. The given website can take up to a minute before it starts to run my code.

1.3 Organization of the Report

The remainder of the report is organized as four sections. In the section *Golden Section Search*, I give my observations about the role of inputs (namely, interval lower bound, upper bound and precision values) on the outcome of the algorithm. Next section is called *Optimal Stepsize Selection.* In this section I run my code with some appropriate choices of algorithm parameters and interval of uncertainty (IoU) using the provided x and d combinations. I discuss several observations of these experiments. In the section *Steepest Descent*, I extend my line search algorithm into a gradient descent code. I report several aspects of this implementation. The report is concluded with the section *Conclusions & Remarks* where I also report some more observations that I find interesting.

2 Golden Section Search

The coded Golden Section Search algorithm can be found in the file *02_Code_Initialization.java*. The algorithm is coded as a function and is called with 3 different inputs, as given below in *Code Box 1*.

Code Box 1: Calling the `goldenSearch` Function

```
public static void main(String[] args) {
        goldenSearch(0.000001, -5000, 5000);
    }
```

The function `goldenSearch` takes precision value, interval lower bound and upper bound as parameters, respectively. Golden Section Search is applied to the univariate function $\theta(x) = (x-2)^2$. This function coded as `theta` in the code, as given below in *Code Box 2*.

Code Box 2: Defining the `theta` Function

```
public static double theta(double x){
        return Math.pow((x-2), 2);
    }
```

Below are the outputs of this algorithm with different combinations of inputs, for the same `theta` function defined in *Code Box 2*.

For `goldenSearch(0.000001, -5000, 5000)`, we have:

```
The optimal x value is between: [a*, b*] = [1.999999501, 2.000000432]
f(a*) = 0.000000000
f(b*) = 0.000000000
The code found the solution at step 32.0.
```

For `goldenSearch(0.1, -5000, 5000)`, we have:

```
The optimal x value is between: [a*, b*] = [1.934984462, 2.031433219]
f(a*) = 0.004227020
f(b*) = 0.000988047
The code found the solution at step 17.0.
```

Notice that, a bigger precision value led to a bigger final interval of uncertainty. We also observe that algorithm stops earlier when we make the precision value bigger. Let's now experiment with the initial lower and upper bounds.

For `goldenSearch(0.1, -3, 3)`, we have:

```
The optimal x value is between: [a*, b*] = [1.966744388, 2.015528100]
f(a*) = 0.001105936
f(b*) = 0.000241122
The code found the solution at step 8.0.
```

Observe that when we feed the algorithm a smaller initial IoU, algorithm stops way earlier.

3 Optimal Stepsize Selection

I made the necessary adjustments to my code to solve an "optimal stepsize selection" problem. The revised code can be found in the file named *03_Code_Adaptation.java.* I implemented my code for the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined as $f(x) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$. The code for defining the function `f` in Java is given below in *Code Box 3*.

Code Box 3: Defining the `f` Function

```java
public static double f(double[] input){
        return Math.pow((input[0] - 2), 4) +
            Math.pow((input[0] - 2*input[1]), 2);
    }
```

Given a point $x \in \mathbb{R}^2$ and a nonzero direction vector $d \in \mathbb{R}^n$, my code solves for $min_{\lambda \in [a,b]} \theta(\lambda)$ where $\theta(\lambda) = f(x + \lambda d)$.

I run my code using the given x and d combinations. I choose 0.000001 as the precision value. -5000 and 5000 are given to the algorithm as lower and upper bounds of IoU. The results are summarized below:

For $x = (0, 0)^T$ and $d = (1, 1)^T$, we have:

```
Optimal stepsize is between: [a, b] = [1.1648754, 1.1648793]
f(x + d*a) = 1.8433476
x + d*a = [1.1648754, 1.1648754]
f(x + d*b) = 1.8433476
x + d*b = [1.1648793, 1.1648793]
The code found the solution at step 32.0
```

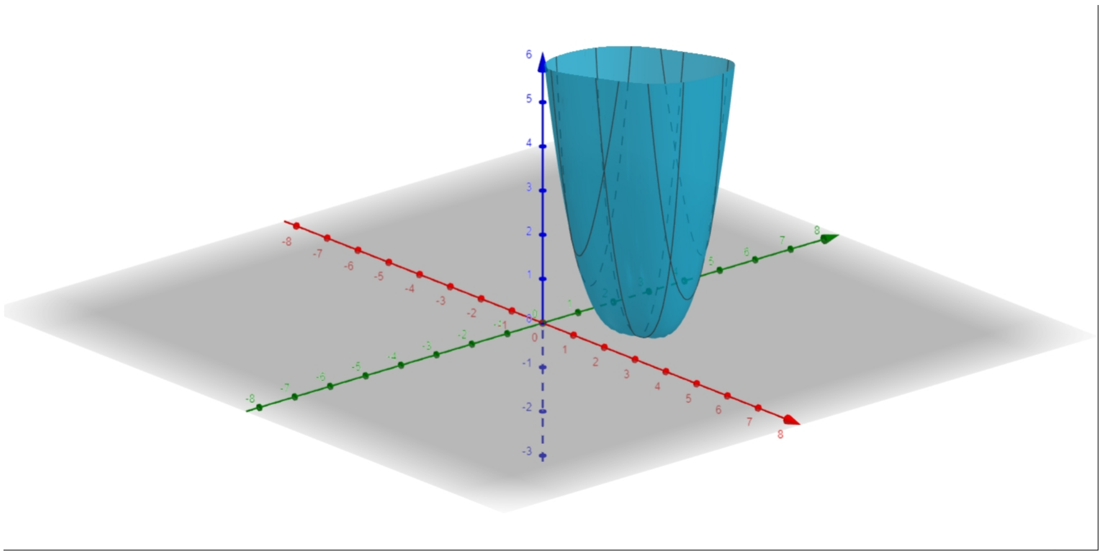

Figure 1: The graph for $f(x) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$.

For $x = (2,2)^T$ and $d = (-1,0)^T$, we have:

```
Optimal stepsize is between: [a, b] = [-0.8351240, -0.8351201]
f(x + d*a) = 1.8433476
x + d*a = [2.8351240, 2.0000000]
f(x + d*b) = 1.8433476
x + d*b = [2.8351201, 2.0000000]
The code found the solution at step 29.0
```

For $x = (2,2)^T$ and $d = (0,-1)^T$, we have:

```
Optimal stepsize is between: [a, b] = [0.9999965, 1.0000029]
f(x + d*a) = 0.0000000
x + d*a = [2.0000000, 1.0000035]
f(x + d*b) = 0.0000000
x + d*b = [2.0000000, 0.9999971]
The code found the solution at step 29.0
```

Observing the solutions for the given points, we see that only the last one converges to the *optimal solution* (by observation). The first two converge to a suboptimal solution. This is because we search the optimal point along a fixed direction. One can see the 3D graph of the function (Figure 1) to get more intuition about why our search along the first two directions doesn't give us the optimal point.

4 Steepest Descent

The coded Steepest Descent algorithm can be found in the file *04_Code_SteepestDescent.java*. For this part of the homework, I included a while loop in the `Main` method. This loop enables us to call the `goldenSearch` function until we meet the condition of $\nabla f(x^k) = 0$. It is important to note that this condition never truly holds. That is, $\nabla f(x^k)$ gets extremely close to 0 as iterations go on, but it never reaches to 0. So, I decided to allow the code to treat $\nabla f(x^k)$ as 0, when every element of $\nabla f(x^k)$ is strictly less than some $\epsilon \in \mathbb{R}$. I decided to choose ϵ as 10^{-6} in our specific problem.

Calculating $\nabla f(x^k)$: I wrote a function `gradientCalc` in Java to calculate $\nabla f(x^k)$ for any given x^k. In doing so, I made use of the limit definition of partial derivatives. For example, for our function $f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$, the coded function follows the facts that:

$$\frac{\partial f}{\partial x_1}(x_1^k, x_2^k) = \lim_{h \to 0} \frac{f(x_1^k + h, x_2^k) - f(x_1^k, x_2^k)}{h}$$

$$\frac{\partial f}{\partial x_2}(x_1^k, x_2^k) = \lim_{h \to 0} \frac{f(x_1^k, x_2^k + h) - f(x_1^k, x_2^k)}{h}$$

In my code, I chose h as 10^{-11} to approximate partial derivatives as closely as possible. In *Code Box 4*, below I present my code for the function `gradientCalc`.

To test my code, I run the algorithm starting from 4 different initial points. The summary table about the experiments are given below:

x^0	$f(x^0)$	x^T	$f(x^T)$	T
$(5, -5)$	306	$(1.994282915, 0.997141426)$	0.000000001	3198
$(-2, 12)$	306	$(1.994363044, 0.997181482)$	0.000000001	7017
$(0, 0)$	16	$(1.994055077, 0.997027504)$	0.000000001	764
$(6, 6)$	292	$(2.005953492, 1.002976780)$	0.000000001	468

In the above table, x^0 is the initial point, x^T is the final point and T is the number of iterations. We observe a convergence to optimal solution in all 4 trials in above table.

05_SteepestDescent_DetailedOutput.txt includes the detailed algorithmic output for the above trial with the initial point of $(6, 6)$.

5 Conclusions & Remarks

This homework was especially beneficial for me to understand the relation between the line search and gradient descent algorithms. I also experienced that Java is not the best programming language to work with arrays. I needed to create my own functions (methods) in my code, e.g. for adding up two arrays together, and multiplying each element of an array with a scalar. I would like to note an interesting observation about the four Steepest Descent trials in §4. For a minimization problem, $f(x_A^0) < f(x_B^0)$ doesn't necessarily imply that $T_A < T_B$, as we see in 3rd & 4th trials.

6 References

I have used GeoGebra 3D Calculator (Graphing) tool for plotting Figure 1.