# Project 5: Polynomial local minimizer v3

## Contents



**New concepts:** Inheritance, recursion

## 1 Overview

In the previous projects, we built a nice framework for testing parameter variations in the steepest descent algorithm. In this project, we will expand the framework to also examine how different line searches affect the performance of our algorithm. The remainder of this section summarizes the description of the polynomial functions and steepest descent algorithm from the previous project.

### 1.1 Polynomials

In this project, we will try to minimize polynomials of the form SimplePolynomialSum:

$$f(x_1,\ldots,x_n)=\sum_{j=1}^{n}\left(c_{jd}x_j^d+c_{j,d-1}x_j^{d-1}+\ldots+c_{j2}x_j^2+c_{j1}x_j+c_{j0}\right) \qquad \text{(SimplePolynomialSum)}$$

Minimization will not be easy. Local minima may be present, and the problem may be unbounded (the polynomial goes to $-\infty$). So, we will content ourselves with finding a local minimum. In the real world, we are often happy to accept a local minimum as a solution when the objective function is too difficult to globally optimize.

We will test on several polynomials, providing the user with both per-polynomial performance and statistical summaries for the following metrics:

- Final objective function value
- Closeness to local minimum ($\ell_2$ norm of the gradient at the final solution)
- Number of iterations needed for the algorithm to run
- Computation time needed for the algorithm to run

### 1.2 The steepest descent algorithm

The steepest descent algorithm is part of a class of algorithms called *directional search* methods. Basically, we start at some feasible point, pick a direction to move, pick an distance to move in the direction, and then move to a new point by a taking a step along the selected distance. We repeat the process over and over until we have reached some stopping criteria.

In steepest descent, the direction is the negative gradient ($-\nabla f(\mathbf{x})$) at the current point $\mathbf{x}$, and in the previous project, we made the simplifying choice to always use the same step size; we will improve on that simplification in this project. Thus, our iterative process is described as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha_k \nabla f(\mathbf{x}^{(k)})$$

In this project, the steepest descent algorithm will stop when either the maximum number of iterations is reached or $\|\nabla f(\mathbf{x})\|_2 \leq \epsilon$, where $\|\nabla f(\mathbf{x})\|_2$ is $\ell_2$ norm of the gradient at $\mathbf{x}$ and $\epsilon$ is some value near zero.

The starting point of algorithm is an appropriate-sized vector with all the same values in each element, i.e., $\mathbf{x}^{(0)} = (x_0, x_0, \ldots, x_0)^\top$, and the user can specify the value of $x_0$ in the algorithm parameters.

## 2 Program description

We have already learned that a line search is the search for the step size $\alpha_k$ in iteration $k$. The line search has a significant ability to impact the performance of a directional search algorithm, both in terms of time and solution quality. If we take the optimal step size in each iteration, we will need far fewer iterations to converge to a local minimum, which saves time. However, finding the optimal step size may be so time-consuming as to cost us more total time than if we used simpler inexact line search methods.

In this project, we will look at three line search methods:

1. Fixed step size (inexact)
2. Armijo line search (inexact)
3. Golden section search (exact)

### 2.1 Fixed step size

This approach is the same approach that we used in previous projects: $\alpha_k = \alpha$. This method is as fast as a line search can be, but may result in us taking too many iterations if the step size is too small (which takes time), or always stepping over the local minimum if the step size is too big, and thus never improving.

### 2.2 Armijo line search

An Armijo line search finds the first step size that ensures that $f(\mathbf{x})$ decreases a "sufficient" amount. The definition of "sufficient" decrease is a step size $\alpha_k$ such that

$$f(\mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)}) \leq f(\mathbf{x}^{(k)}) + \alpha_k \beta [\nabla f(\mathbf{x}^{(k)})]^\top \mathbf{d}^{(k)}$$

where $\mathbf{d}^{(k)}$ is the direction and $\beta$ is a user-defined scalar in $(0, 1)$. Since $\mathbf{d}^{(k)} = -\nabla f(\mathbf{x}^{(k)})$ in our steepest descent algorithm, we can re-write this condition as

$$\begin{aligned} f(\mathbf{x}^{(k)} - \alpha_k \nabla f(\mathbf{x}^{(k)}) &\leq f(\mathbf{x}^{(k)}) - \alpha_k \beta [\nabla f(\mathbf{x}^{(k)})]^\top \nabla f(\mathbf{x}^{(k)}) \\ &\leq f(\mathbf{x}^{(k)}) - \alpha_k \beta \|\nabla f(\mathbf{x}^{(k)})\|_2^2 \end{aligned} \qquad \text{(ArmijoDecrease)}$$

where $\|\nabla f(\mathbf{x}^{(k)})\|_2^2$ is the $\ell_2$ norm of the gradient squared. How convenient that we already have a function to calculate the $\ell_2$ norm of the gradient!

Now, all we have to do is find an $\alpha$ value in $(0, 1]$ that satisfies the ArmijoDecrease condition. First, set $\alpha = 1$ and see if the equation is satisfied. If not, multiply $\alpha$ by a user-specified value $\tau \in (0, 1)$ (i.e., $\alpha = \tau\alpha$) and try again. Repeat the process until the ArmijoDecrease condition is satisfied. The **entire** steepest descent algorithm quits if the Armijo line search fails to find a suitable $\alpha$ after a user-specified $K$ iterations.

Essentially, we start with a "large" step size, and keep moving backward until we have sufficient improvement in our function, and we say we have converged if sufficient improvement isn't there.

## 2.3 Golden section search

Golden section search (GSS) is a simple and effective method of finding the minimum (or maximum) of a 1D function where the function has exactly one minimum (or maximum) in the interval to be explored. In our case, we want the minimum of $f(\mathbf{x}^{(k)} - \alpha\nabla f(\mathbf{x}^{(k)})]$ *with respect to* $\alpha$ to give us the optimal step size $\alpha$; the interval is the minimum and maximum allowable step size.

### 2.3.1 Overview

GSS is based on the golden ratio

$$\varphi = (1 + \sqrt{5})/2 \approx 1.618 \qquad \text{(GoldenRatio)}$$

which is simply three points $a, b, c$ in interval $[a, b]$ that satisfy the golden ratio, as illustrated in Figure 1a. We use $\varphi$ to calculate $c$:

$$c = a + \frac{b - a}{\varphi}$$

We call these points satisfying the golden ratio "triplet" $(a, c, b)$.

How is that useful to us? Well, we want to find the minimum of $f(\alpha)$ over an interval $[a, b]$; we will find it using triplet $(a, c, b)$. We find out which sub-interval ($[a, c]$ or $[c, b]$) the minimum is in, then we find a new golden ratio point in that sub-interval and repeat the process until the interval being examined is very small.

How do we figure out whether our minimum is in the left interval or the right interval? Let's use Figure 1b to demonstrate. Since there is only one minimum in the interval $[a, b]$, $f_c$ must be smaller than $f_a$ and $f_b$. We sample a point $y$ in the larger interval (in this case, the left interval, $[a, c]$), where

$$y = a + \frac{c - a}{\varphi}$$

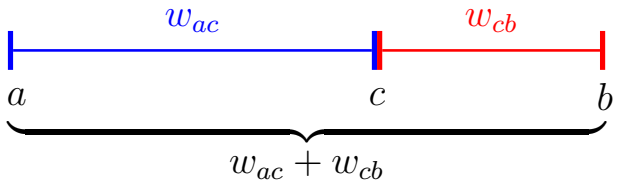
We then check if $f(y)$ is less than the function value at the endpoints of its interval, i.e., $f_a$ and $f_c$. Say $f(y) = f_{y2}$, which is less than both $f_a$ and $f_b$. Then, the minimum must be in the interval $[a, c]$, and we repeat the process using the triplet $(a, y, c)$, which satisfies the golden ratio since we calculated $y$ the same way we calculated $c$. Conversely, if $f(y) = f_{y1}$ (which is larger than $f_c$), the minimum must be in interval $[c, b]$, so we repeat the process using the triplet $(y, c, b)$, which also satisfies the golden ratio.

Alternatively, if our triplet $(a, c, b)$ has the larger interval on the right (see Figure 1c), we calculate $y$ as

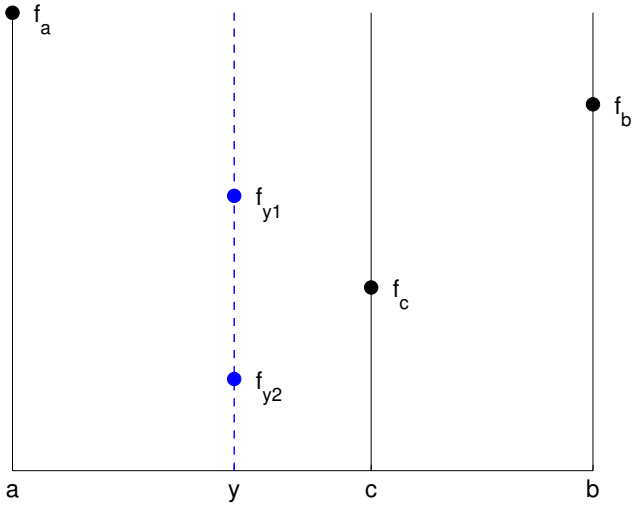
$$y = b - \frac{b - c}{\varphi}$$

The rest of the process is the same.

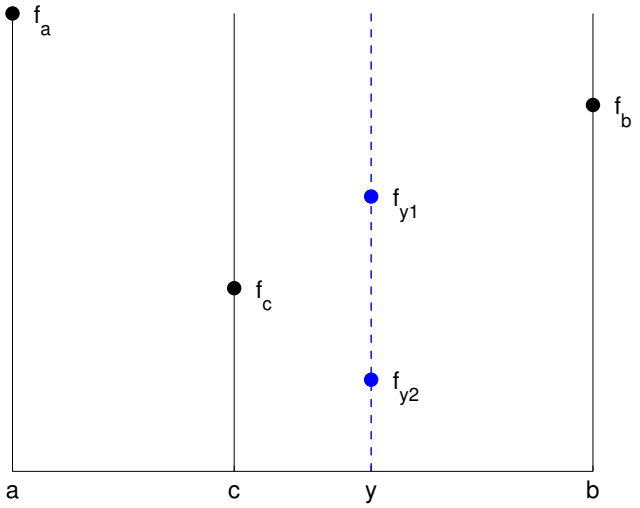
As we are applying GSS to a line search, we need to get from the user $a$ (the minimum step size) and $b$ (the maximum step size). We will continue iterating until the size of the interval is less than a user-specified $\delta$ value, at which point we return the midpoint of the interval.




(a) The golden ratio represents $\varphi = w_{ac}/w_{cb} = (w_{ac} + w_{cb})/w_{ac}$.




(b) Golden section search where the left interval is larger; $y = a + (c - a)/\varphi$




(c) Golden section search where the right interval is larger; $y = b - (b - c)/\varphi$

Figure 1: Relationship of the golden ratio $\varphi$ to the golden section search method

### 2.3.2 Special considerations

By now, you may be wondering about the validity of the assumption that there is exactly one minimum in the function in the initial interval. We have no way of guaranteeing that we always pick a large enough maximum step size to capture a minimum, and given the nature of our polynomials, there may be multiple minima in the interval $[a, b]$. So, we need a way to identify these situations and handle them. At any iteration in GSS, if the function value of the golden ratio point in the interval ($c$ in Figure 1) is larger than the function value at either endpoint, then there is no minimum or there are multiple minima in the interval. We have two possible scenarios:

1. $f_a \geq f_b$ ($f$ improves with the largest step size), so pick the right endpoint (the largest step size).
2. $f_a < f_b$ ($f$ worsens with the largest step size), so pick the left endpoint (the smallest step size).

These rules are imperfect, but they are quick and easy to implement.

# 3 Menu options

- `L - Load polynomials from file`
  Read in polynomials from a user-specified file.
- `F - View polynomial functions`
  Print the polynomial functions to the console.
- `C - Clear polynomial functions`
  Clear all loaded polynomial functions.
- `S - Set steepest descent parameters`
  The user enters values for each of the steepest descent parameters for each of the algorithm variations (fixed line search, Armijo line search, golden section search) sequentially.
- `P - View steepest descent parameters`
  The steepest descent parameters for each algorithm variation are printed to the console.
- `R - Run steepest descent algorithms`
  Run each steepest descent algorithm variation on all polynomials.
- `D - Display algorithm performance`
  Display the algorithm performance for each individual polynomial and the statistical summary for each algorithm variation.
- `X - Compare average algorithm performance`
  Compare the average performance of each algorithm variation on the three metrics (norm of the gradient, number of iterations needed, computation time needed) and identify the best performer in each category and the overall best performer. Ties in best performance are broken in this order: fixed, Armijo, GSS.
- `Q - Quit`
  Quit the program.

# 4 Error messages

- `ERROR: Invalid menu choice!`
  when the user enters an invalid menu choice.
- `ERROR: No polynomial functions are loaded!`
  when the user attempts to solve or print the polynomial without entering a polynomial function.

- `ERROR: File not found!`
  when the user enters a file that does not exist.
- `ERROR: Inconsistent dimensions in polynomial <n>!`
  when the dimensions are inconsistent in the $n$th polynomial in the file being read
- `ERROR: Results do not exist for all line searches!`
  when the user attempts to view algorithm results when results do not exist for the current polynomials and all current algorithm variation parameters.
- `ERROR: Input must be an integer in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of an `int`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of an `int`, then "`-infinity`" is displayed instead of the actual LB value.
- `ERROR: Input must be a real number in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of a `double`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of a `double`, then "`-infinity`" is displayed instead of the actual LB value.
- `Armijo line search did not converge!`
  when the maximum number of Armijo line search iterations is reached.

# 5 Required elements

Your project must use the each of the elements described in this section. You can have more functions, variables, and objects if you want, but you must at least use the elements described in this section.

**Failure to correctly implement any of the elements in this section may result in a zero on the project.**

## 5.1 Variables

- An `ArrayList` of `Polynomial` objects, as defined in Section 5.2 (an `ArrayList` will be much easier than an array given the unknown quantity of polynomials that will be loaded into the program)
- A single `SDFixed` object, as defined in Section 5.2
- A single `SDArmijo` object, as defined in Section 5.2
- A single `SDGSS` object, as defined in Section 5.2
- Global `public static BufferedReader` as **the only** `BufferedReader` object in the entire program for reading user input
  - Although your programs will compile and run just fine on your own computer with multiple `BufferedReader` objects, they will not run correctly on a web server with multiple `BufferedReader` objects. Since your programs will be graded on a web server, please don't have more than one `BufferedReader` for reading input from the console.

## 5.2 Classes

You are required to write the following classes:

1. `Polynomial` class to store all the polynomial data
2. `SteepestDescent` class that provides the steepest descent algorithm template (basically the same as the class used in the previous project)

3. `SDFixed` class that inherits from the `SteepestDescent` class and runs the fixed line search variation of the algorithm described in Section 2.1

4. `SDArmijo` class that inherits from the `SteepestDescent` class and runs the Armijo line search variation of the algorithm described in Section 2.2

5. `SDGSS` class that inherits from the `SteepestDescent` class and runs the GSS line search variation of the algorithm described in Section 2.3

You can write more classes, and you can add to these classes, but you must at least implement the classes defined in this section.

```java
public class Polynomial {
  private int n;              // no. of variables
  private int degree;         // degree of polynomial
  private double[][] coefs;   // coefficients

  // constructors
  public Polynomial()
  public Polynomial(int n, int degree, double[][] coefs)

  // getters
  public int getN()
  public int getDegree()
  public double[][] getCoefs()

  // setters
  public void setN(int a)
  public void setDegree(int a)
  public void setCoef(int j, int d, double a)

  // other methods
  public void init()                      // init member arrays to correct size
  public double f(double[] x)             // calculate function value at point x
  public double[] gradient(double[] x)    // calculate gradient at point x
  public double gradientNorm(double[] x) // calculate norm of gradient at point x
  public boolean isSet()                  // indicate whether polynomial is set
  public void print()                     // print out the polynomial
}
```

Polynomial_Template.java

```java
public class SteepestDescent {
  private double eps;             // tolerance
  private int maxIter;            // maximum number of iterations
  private double x0;              // starting point
  private ArrayList<double[]> bestPoint; // best point found for all polynomials
  private double[] bestObjVal;    // best obj fn value found for all polynomials
  private double[] bestGradNorm;  // best gradient norm found for all polynomials
  private long[] compTime;        // computation time needed for all polynomials
  private int[] nIter;            // no. of iterations needed for all polynomials
  private boolean resultsExist;   // whether or not results exist

  // constructors
  public SteepestDescent()
  public SteepestDescent(double eps, int maxIter, double x0)

  // getters
  public double getEps()
```

```
  public int getMaxIter()
  public double getX0()
  public double[] getBestObjVal()
  public double[] getBestGradNorm()
  public double[] getBestPoint(int i)
  public int[] getNIter()
  public long[] getCompTime()
  public boolean hasResults()

  // setters
  public void setEps(double a)
  public void setMaxIter(int a)
  public void setX0(double a)
  public void setBestObjVal(int i, double a)
  public void setBestGradNorm(int i, double a)
  public void setBestPoint(int i, double[] a)
  public void setCompTime(int i, long a)
  public void setNIter(int i, int a)
  public void setHasResults(boolean a)

  // other methods
  public void init(ArrayList<Polynomial> P)  // init member arrays to correct size
  public void run(int i, Polynomial P)       // run the steepest descent algorithm
  public double lineSearch(Polynomial P, double[] x)  // find the next step size
  public double[] direction(Polynomial P, double[] x) // find the next direction
  public boolean getParamsUser()      // get parameters from user, return success
  public void print()                 // print algorithm parameters
  public void printStats()            // print statistical summary of results
  public void printAll()              // print final results for all polynomials
  public void printSingleResult(int i, boolean rowOnly) // print final result for
      one polynomial, column header optional
}
```

SteepestDescent_Template.java

```
public class SDFixed extends SteepestDescent {
  private double alpha;    // fixed step size

  // constructors
  public SDFixed()
  public SDFixed(double alpha)

  // getters
  public double getAlpha()

  // setters
  public void setAlpha(double a)

  // other methods
  public double lineSearch(Polynomial P, double [] x)  // fixed step size
  public boolean getParamsUser() // get algorithm parameters from user
  public void print()            // print parameters
}
```

SDFixed_Template.java

```
public class SDArmijo extends SteepestDescent {
  private double maxStep; // Armijo max step size
  private double beta;    // Armijo beta parameter
  private double tau;     // Armijo tau parameter
```

```
  private int K;          // Armijo max no. of iterations

  // constructors
  public SDArmijo()
  public SDArmijo(double maxStep, double beta, double tau, int K)

  // getters
  public double getMaxStep()
  public double getBeta()
  public double getTau()
  public int getK()

  // setters
  public void setMaxStep(double a)
  public void setBeta(double a)
  public void setTau(double a)
  public void setK(int a)

  // other methods
  public double lineSearch(Polynomial P, double [] x)  // Armijo line search
  public boolean getParamsUser() // get algorithm parameters from user
  public void print()            // print parameters
}
```

SDArmijo_Template.java

```
public class SDGSS extends SteepestDescent {
  private final double _PHI_ = (1. + Math.sqrt(5))/2.;
  private double maxStep;    // Armijo max step size
  private double minStep;    // Armijo beta parameter
  private double delta;      // Armijo delta parameter

  // constructors
  public SDGSS()
  public SDGSS(double maxStep, double minStep, double delta)

  // getters
  public double getMaxStep()
  public double getMinStep()
  public double getDelta()

  // setters
  public void setMaxStep(double a)
  public void setMinStep(double a)
  public void setDelta(double a)

  // other methods
  public double lineSearch(Polynomial P, double [] x)  // step size from GSS
  public boolean getParamsUser() // get algorithm parameters from user
  public void print()            // print parameters
  private double GSS(double a, double b, double c, double [] x, double [] dir,
      Polynomial P)
}
```

SDGSS_Template.java

### 5.3 Functions

Function prototypes and short descriptions are provided below. It is your job to determine exactly what should happen inside each function, though it should be clear from the function and argument names and function descriptions.

- `public static void displayMenu()`
  Display the menu.
- `public static boolean loadPolynomialFile(ArrayList<Polynomial> P)`
  Load the polynomial function details from a user-specified file.
- `public static void getAllParams(SDFixed SDF, SDArmijo SDA, SDGSS SDG)`
  Get algorithm parameters from the user for each algorithm variation.
- `public static void printAllParams(SDFixed SDF, SDArmijo SDA, SDGSS SDG)`
  Print the parameters for all the algorithm variations.
- `public static void runAll(SDFixed SDF, SDArmijo SDA,SDGSS SDG,ArrayList<Polynomial> P)`
  Run all algorithm variations on all loaded polynomials.
- `public static void printPolynomials(ArrayList<Polynomial> P)`
  Print all the polynomial functions currently loaded.
- `public static void printAllResults(SDFixed SDF, SDArmijo SDA, SDGSS SDG, ArrayList<Polynomial> P)`
  Print the detailed results and statistics summaries for all algorithm variations.
- `public static void compare(SDFixed SDF, SDArmijo SDA, SDGSS SDG)`
  Compare the performance of the algorithms and pick winners.
- `public static int getInteger(String prompt, int LB, int UB)`
  Get an integer in the range [LB, UB] from the user. Prompt the user repeatedly until a valid value is entered.
- `public static double getDouble(String prompt, double LB, double UB)`
  Get a real number in the range [LB, UB] from the user. Prompt the user repeatedly until a valid value is entered.

## 6 Helpful hints

- All the hints from the previous steepest descent project are still useful.
- Implement GSS using recursion. It's the easiest way! All you have to do in the GSS method is find the new triplet we want to examine, then call the GSS method for that triplet. Return the midpoint of the interval when the interval is sufficiently small.
- Since we have to display a row of performance metric averages in two places (in printing the statistics summary and in doing the comparison), it would save time to write a method for the `SteepestDescent` class that just prints the row of averages. Note that the row title (i.e., "Averages" or the line search name) is different in those two situations, so you will want to be able to handle different row titles to print out.