Project 3: Gradient-based Polynomial Minimizer

Contents

1 Polynomials **1**

2 The steepest descent algorithm **2**
2.1 Stopping criteria . . . 2
2.2 Step direction . . . 3
2.3 Line search . . . 3

3 Program description **4**

4 Menu options **4**

5 Error messages **5**

6 Required elements **5**

7 Helpful hints **6**

New concepts: Classes, objects, time-keeping, file reading, Collections classes (ArrayList, TreeSet – a sorted HashSet, and HashMap), optimization algorithms

1 Polynomials

You should know what a polynomial is at this point, but just in case you've forgotten, a polynomial is any mathematical expression constructed from variables and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents. For example, a univariate polynomial may take the form:

$$f(x) = x^2 + -8x + 12 \tag{1}$$

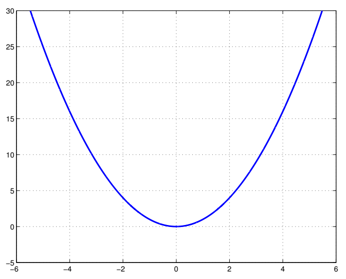
We would say that this **polynomial** has degree 2 and consists of 3 **terms** being summed (x^2, $-8x$, and 12).

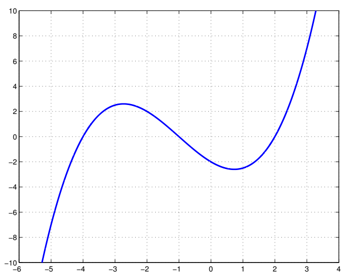
The univariate example (1) is one-dimensional (that is, variable x is a scalar), but polynomials can have any number of variables, and those variables can be multiplied together in any combination. Figure 1 shows some examples of polynomials, and the bottom row is just two univariate polynomials (the same form as (1))—one of x_1 and one of x_2—added together. More generally we can express multivariate polynomials with any number of variables:

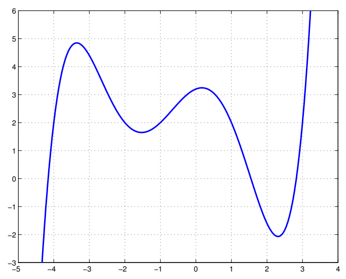
$$f(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_1^3x_2^2x_3 + -4x_1x_2 + -8x_1x_2x_3 + -x_2 + 1 \tag{2}$$

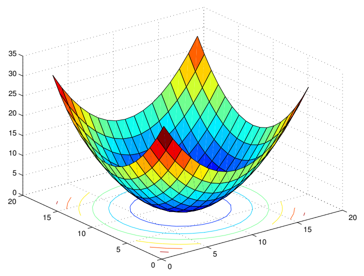
Of course it does not matter whether our variables are $x_1, x_2, x_3, \ldots$ or $x, y, z, \ldots$.

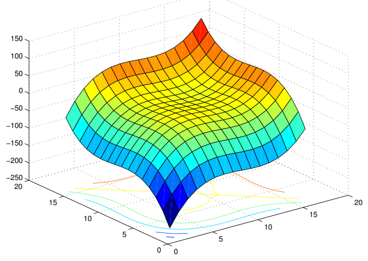
In this project, we will try to minimize multivariate polynomials like (2). As you can see from the examples in Figure 1, minimization will not be easy. Local minima may be present, and the problem may be unbounded (the polynomial goes to $-\infty$). So, we will content ourselves with finding a local 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.


(a) x^2


(b) $\frac{1}{4}x^3 + \frac{3}{4}x^2 - \frac{3}{2}x - 2$


(c) $\frac{1}{20}(x+4)(x+2)(x+1)(x-1)(x-3)+2$


(d) $x_1^2 + x_2^2$


(e) $\frac{1}{4}x_1^3 + \frac{3}{4}x_1^2 - \frac{3}{2}x_1 - 2 + \frac{1}{4}x_2^3 + \frac{3}{4}x_2^2 - \frac{3}{2}x_2 - 2$

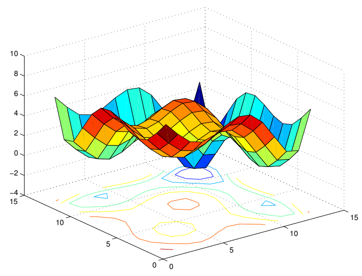

(f) $\frac{1}{20}(x_1+4)(x_1+2)(x_1+1)(x_1-1)(x_1-3) + 2 + \frac{1}{20}(x_2+4)(x_2+2)(x_2+1)(x_2-1)(x_2-3)+2$

Figure 1: Polynomial examples

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 (see Figure 2). We repeat the process over and over until we have reached some stopping criteria.

Defining $\mathbf{d}^{(k)}$ as the chosen direction in iteration k ($\mathbf{d}^{(k)} = (d_1, \ldots, d_n)^\top$ is a vector with a unique direction for each variable x_i) and the step size as α_k, this iterative process is described as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)}$$

As you can see, the two primary components of directional search methods are the step direction and the step length, and they are essentially the only things that vary from one directional search method to another.

2.1 Stopping criteria

Two common stopping criteria are

- Maximum number of iterations reached
- Sufficient closeness to a local minimum

How can we tell if we are sufficiently close to a local minimum? Just look at the gradient $\nabla f(\mathbf{x})$: If the gradient is close to zero, then we are probably pretty close to a local minimum. Since we have a vector of variables ($\mathbf{x}$) and not just a scalar, the gradient will be a vector, and so we look at the size of gradient (defined by its norm) to see how close we are. Formally, we are close enough to a local minimum when

$$\|\nabla f(\mathbf{x})\|_2 \leq \epsilon$$

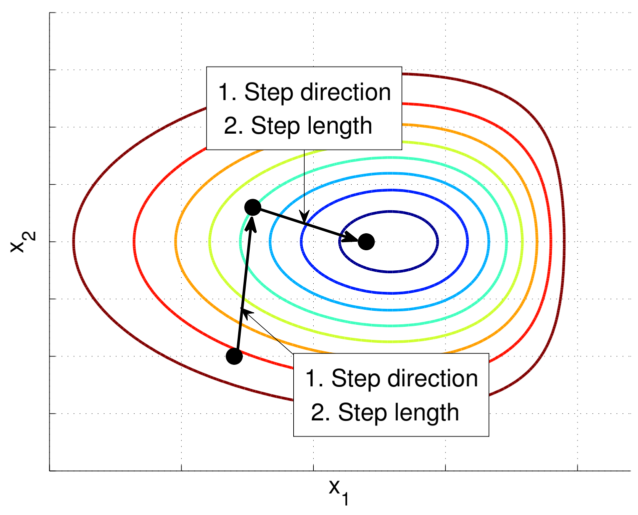


Figure 2: Directional search illustration

where ϵ (called the tolerance) is a small value and $\|\cdot\|_2$ is the ℓ_2 norm, defined as

$$\|\mathbf{y}\|_2 = \sqrt{y_1^2 + \ldots + y_n^2}$$

Recall from linear algebra that the gradient of function $f(\mathbf{x})$ is calculated as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \partial f(\mathbf{x})/\partial x_1 \\ \vdots \\ \partial f(\mathbf{x})/\partial x_n \end{bmatrix}$$

In this project, the steepest descent algorithm will stop when either the maximum number of iterations is reached or $\|\nabla f(\mathbf{x})\|_2 \leq \epsilon$.

2.2 Step direction

In steepest descent, the goal is to move as quickly as possible to the closest local minimum. Recall that the first derivative of a one-dimensional function tells us the slope of the function. If we want to get to the closest local minimum, we should just follow the slope; more specifically, follow the negative of the slope. For example, Figure 1a shows $f(x) = x^2$, which has derivative $f'(x) = 2x$ and minimum at $x = 0$. If $x = 4$, then $f'(4) = 8$, so our direction should be -8, meaning that we should reduce x to get closer to the minimum.

The process for a multi-dimensional $\mathbf{x}$ is the same. Since a gradient is just a collection of first derivatives (slopes) for every variable, if we are at point $\mathbf{x}$, the direction with the most decrease in every variable is just the negative gradient, $-\nabla f(\mathbf{x})$.

Note that because we are aiming for the closest local minimum, our choice of starting point ($\mathbf{x}^{(0)}$) may significantly impact what local minimum we find. But, there is no way of knowing an appropriate starting point in advance; we just have to try things out. A common starting point is something very close to zero, e.g., $\mathbf{x}^{(0)} = (0.05, \ldots, 0.05)^\top$.

2.3 Line search

Normally, we would try to optimize the step size α each iteration by performing what is called a *line search* (an optimization of f as a function of α), but we will be easy for now and just fix α to one value for whole algorithm. If we make α too big, we may continually step over the local minimum and never converge; however, if we make α too small, we may take such small steps that our algorithm moves too slowly to be practical. There's no easy way to find a happy middle ground; we just have to try out different values and see what works for our particular problem.

3 Program description

In this project, you will implement the steepest descent algorithm to find local minima of polynomials of the form (2). You will display the performance of the algorithm for the following metrics:

- Final objective function value
- Closeness to local minimum (ℓ_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

Important Notes: Most input-output statements, polynomial parsing, and a compilable skeleton of all classes in the project are provided for you. This project should require much less focus on output formatting than previous projects. You have to implement reading from a file, symbolic differentiation, and gradient descent optimization. In addition, you have to get used to working with ArrayLists, TreeSets – a sorted variant of a HashSet, and HashMaps. This will take some getting used to, but these are invaluable tools in modern Java programming. For all of the Java "Collections" classes as well as other classes you are encountering for the first time (e.g., StringBuilder), please refer to their Java documentation (just Google for "Java" and the class name) to understand what functionality these classes offer.

While I have provided a lot of code for you, I would expect that you can write all code on your own – so please read through and understand it – you may see a snippet on an exam.

4 Menu options

All menu functionality has been implemented for you in `Pro3_utorid.java` **except** for implementation of method calls in other classes and the first option **R**, where you have to implement file reading.

- `R - Read polynomial function`
  The user enters a filename where the polynomial can be read. Polynomial parsing has been implemented for you in the `Polynomial` class constructor.
- `F - View polynomial function`
  The polynomial function is printed to the console. This functionality and the implementation of `toString()` for the `Polynomial` class has been done for you.
- `S - Set steepest descent parameters`
  The user enters values for each of the steepest descent parameters: tolerance ϵ, maximum number of iterations, step size α, and starting point ($\mathbf{x}^{(0)}$ for all variables in the polynomial). Aside from method calls in other classes, this functionality has been done for you.
- `P - View steepest descent parameters`
  The steepest descent parameters are printed to the console. This functionality has been done for you.
- `M - Minimize polynomial by gradient descent`
  Run the steepest descent algorithm on the polynomial. You need to implement the functionality in the `Minimizer` class.
- `Q - Quit`
  Quit the program.

5 Error messages

All output and error messages have been provided for you in the code. Our test cases for this project will focus on correct functionality of the minimizer rather than error checking for pathological cases.

6 Required elements

Your project must use all methods and members provided in the starter code – definitions should not be changed. You can have more classes, members, and methods 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.**

Below I comment specifically on methods you have to implement – please see the starter code for each class to see *additional members and methods* that have already been implemented for you. **Important Note:** These classes are listed in the order that you should complete them – many classes have their own `main()` methods for local testing of class functionality.

class `VectorUtils`: Performs operations on vectors stored as HashMaps. See the `main()` method for an explanation and test cases with expected output.

- `public static HashMap<String,Double> sum(HashMap<String,Double> x, HashMap<String,Double> y)`
  Produces the sum of two vectors (assuming they have the same variables).
- `public static HashMap<String,Double> scalarMult(double scalar, HashMap<String,Double> x)`
  Performs an elementwise multiply of a vector with a scalar value.
- `public static double computeL2Norm(HashMap<String,Double> x)`
  Computes the ℓ_2 norm of a vector as defined earlier.

class `Term`: Implements an individual term of a polynomial.

- `public void setCoef(double coef)`
  Sets the coefficient class member.
- `public double getCoef()`
  Returns the coefficient class member.
- `public double evaluate(HashMap<String, Double> assignments) throws Exception`
  Evaluate this term for the given variable assignments. If a term contains a variable that is not assigned, an Exception should be thrown (exact text does not matter).
- `public Term differentiate(String var)`
  Provide the symbolic form resulting from differentiating this term w.r.t. var. One can assume that each variable name only appears one in the list `_vars` being multiplied together in this term.

class `Polynomial`: Implements a multi-term polynomial (sum of constituent terms). See the `main()` method for examples of expected functionality and test cases with expected output.

- `public double evaluate(HashMap<String, Double> assignments) throws Exception`
  Evaluate this polynomial for the given variable assignments. If a polynomial contains a variable that is not assigned, an Exception should be thrown (exact text does not matter).
- `public Polynomial differentiate(String var)`
  Provide the symbolic form resulting from differentiating this polynomial w.r.t. var.

class `Minimizer`: Implements a gradient descent minimizer for a polynomial. See the `main()` method for an example of expected functionality and a test case with expected output.

- Numerous "getters" and "setters" of class members as outlined in the comments.

- `public void minimize(Polynomial p) throws Exception`
  Implements the main gradient descent algorithm described in this handout. The `Polynomial evaluate()` method may generate Exceptions which should not be caught since this method passes these Exceptions on up to its calling method.

class `Pro3_utorid`: Implements the main user interaction loop. This class and its functionality (including input-output formatting) is almost completely implemented for you.

- `private static Polynomial readPolynomial() throws Exception`
  Asks the user for a filename, retrieves first line, and makes a new Polynomial with it. Anything that goes wrong is thrown here as an Exception (exact text does not matter) and caught and handled by the calling method.

7 Helpful hints

- It will take a long time to understand the complete functionality of all the classes provided. Begin by coding the classes **in the order they are listed above** and try to **reproduce the functionality** as expected in the `main()` method test cases, which will also help you understand what each class does.
- Note that a lot of numerical output is not formatted (i.e., just using `System.out.println`) and, in particular, HashMaps and TreeSets are often printed to the screen directly with `System.out.println(o)` where o is a HashMap or TreeSet object. This is not necessarily user-friendly, but it will save you a lot of time with formatting details.
- Here is an example of how to keep time in milliseconds (ms) in Java:

```java
long start = System.currentTimeMillis();  // Get current time
// do something ...
long elapsedTime = System.currentTimeMillis()-start; // Get elapsed time in ms
```

- **Don't worry if your computation times are not exactly the same as the solution's times.** The computation time is not only dependent on the algorithm, but also on the coding style: efficiency of loops, efficiency of data structures, etc. Computation time will not be compared when grading.
- **File reading note:** When running chk_pro and grading, the system will assume the file is in the same directory as your .class files. If your Eclipse installation does not treat the directory with the class files as the "working directory" (i.e., it claims "file not found") then you can change the working directory in Eclipse by (1) right-click on the class name, (2) select Run-as→Run configurations, (3) change to the Arguments tab, (4) change the Working directory listed at the bottom.
- **Symbolic differentiation and String equality:** You can assume that each variable only occurs once in a term. Each term can be differentiated and the results summed. When differentiating a term, it will be critical to determine whether the variable being differentiated occurs in the term, which will most likely be done with a String equality check. For **Strings `a` and `b`**, make sure you use **`a.equals(b)` to test String equality**, i.e., `a == b` only tests whether the memory addresses of `a` and `b` are equal.