ORI391 & CSE 393: Course Project

The University of Texas at Austin Due: April 25, 2024

Project Overview

The project for this course involves the implementation of a package of line search and trust region methods for solving unconstrained optimization problems. This document describes the algorithms that you are required to implement and further guidelines that you are to follow. It also describes a set of problems that you are asked to solve and the expectations of a written report that you are asked to submit with your code. The coding exercises assigned throughout the semester will be part of the complete package. You will be required to implement more algorithms and options, allowing flexibility for users of your code. All requirements are described in this document.

Coding Guidelines

Upon completion of your Matlab or Python software package, a user should be able to run any of your implemented algorithms with various sets of inputs using the following command:

$$[x, f] = \text{optSolver-TeamName(problem,method,options)}.$$

Here, optSolver-TeamName is the name of your software.

Outputs:

- x: the final iterate produced by an algorithm.
- f: the function value at the final iterate produced by an algorithm.

Inputs:

- $problem$: The minimal requirements for the problem $struct$ are $problem.x0$ the starting point and $problem.name$ the problem name. Failure to pass either should return an $error$. Note, associated with each problem, you should have functions that compute the function, gradient, and Hessian given a point x.
- $method$: The minimal requirement for the method $struct$ is a method name $method.name$. Of course, all methods have their own associated parameters and options. These should be set to default values if a user choses not to specify them. For example, if one wants to run $GradientDescent$ with a fixed step size, but fails to specify a step size, the code should set the step size to a default value. The following methods will be investigated in this project:
  1. $GradientDescent$, with Armijo backtracking line search
  2. $GradientDescentW$, with Wolfe line search

3. $ModifiedNewton$, with Armijo backtracking line search
4. $ModifiedNewtonW$, with Wolfe line search
5. $NewtonCG$, with Armijo backtracking line search
6. $NewtonCGW$, with wolfe line search
7. $BFGS$, BFGS quasi-Newton with Armijo backtracking line search
8. $BFGSW$, BFGS quasi-Newton with Wolfe line search
9. DFP, DFP[1] quasi-Newton with backtracking line search
10. $DFPW$, DFP quasi-Newton with Wolfe line search
11. $LBFGS$, L-BFGS quasi-Newton with Armijo backtracking line search
12. $LBFGSW$, L-BFGS quasi-Newton with Wolfe line search

- $options$: The minimal requirement for the options $struct$ is an empty struct options=[]. All options can be set to default values. Remember, there are a lot of options; e.g., termination tolerances, maximum number of iterations, line search constants, etc). That being said, your code should allow a user the specify the following options:

  1. $term - tol$: optimality tolerance
  2. $max - iterations$: iteration limit
  3. c_1 and c_2: line search parameters
  4. η: Newton-CG tolerance $\|r_k\| \leq \eta\|\nabla f(x_k)\|$
  5. m: L-BFGS memory size

  Preferably, every constant in your code should be an input that the user can specify, if they so choose.

Coding Requirements

- It is a strict **requirement that your code must be well-commented**. A good rule of thumb is that we should be able to follow all of the steps in your code without looking at any of the code at all! We should be able to follow everything simply by reading the in-line comments. This will help us provide partial grading.

- You should augment the output of your code to return the quantities required for investigating the performance of the algorithms. For example, you may want to return the number of iterations required to get a specific tolerance or a flag indicating why an algorithm failed. Of course, there are many other quantities you may want to return. Think about this carefully before running your experiments.

[1] see Section 6.1 eq (6.13) and (6.15) in the Numerical Optimization textbook for the DFP update formula

Test Problems

The problems that you are asked to solve with your various algorithms are posted on the course site. Please see the zip file at Project/problems.zip. A description of each problem is provided in the function files themselves.You should use these files as test problems to check that your code is working correctly, but it is also a wise idea to create your own test problems so that your code solves more than these problems! Note, for some problems we have provided the function, gradient and Hessian computations.

Project Deliverables

The final deliverable for the project are a written report and your software package. There are two mandatory submissions as part of the project (Phase I and Phase II).

Phase I (due: April 02, 20% of grade)

The deliverables for Phase I are as follows:

- Please form a team of 2-3 students and set a team name.
- Decide which big question (see below) you will investigate and describe how you intend to accomplish your goal(s). Be as precise as possible.
- Report should be short (2 paragraphs, around or less than half a page). Be as specific as possible.

Phase II (due April 25, 80% of grade)

You are required to submit a written report with your software package. Your report should include the following:

- Summarize each algorithm in a few sentences each. Your descriptions should include whether it is a Armijo backtracking or Wolfe line search algorithm, how the steps are computed, how nonconvexity is handled, etc. Be descriptive but concise.
- Provide a table of default *options* for your code, which may or may not vary between algorithms. (The optimality tolerance and iteration limit should be $1e-6$ and $1e+3$, respectively, but you are asked to provide default values for the remaining parameters.) From testing your code, you should pick values that you believe to be the best for your algorithms.
- Compare the performance of the algorithms on the given problems.
  1. Provide a table (“Table: Summary of Results") of the numbers of iterations, function evaluations, gradient evaluations, and CPU seconds required to solve each of the twelve (12) given test problems with each of your algorithms. The table should include a row for each problem and a column for each algorithm. If a particular algorithm fails to solve a particular problem, then indicate that.

2. Compare the algorithms using Function vs. iteration and $\|\nabla f(x)\|_2$ vs. iteration plots for each problem. That is, the y-axis of your plot is function values or norm of the gradient (in log scale) and the x-axis is iterations.

- Comment on the results in your output table. Was any algorithm consistently the best? If not, can you guess why some algorithms had trouble with certain problems?
- If you had to choose one algorithm (that balances cost, convergence speed etc.), which algorithm would be your algorithm of choice"? Discuss your decision.
- Investigate one big question thoroughly. (See below for examples.) The idea of this portion of the project is to investigate in depth one aspect of an (or several optimization algorithms). Examples of questions are:
  - Is the curvature condition important in a line search?
  - What are good choices of the line search parameters?
  - What are good choices of the Newton-CG tolerance parameter?
  - How does memory affect the performance of L-BFGS methods?
  - In L-BFGS methods, which pair should be removed at every iteration?
  - How much modifying is too much in Newton's method?

  You may choose to answer one of the questions above or any other question you see fit. Please discuss with the instructor if you have questions.
- Summarize your experience with this project. Would you declare any of your algorithms the winner? Consideration for that distinction should include the algorithm's performance, but also how easy it was to code and how many parameters you needed to tune before it worked well. Indeed, you should comment on your experience coding all of the algorithms and describe your impressions of each. What method would you recommend to an expert coder? What method would you recommend to a user who is not an expert in coding or in nonlinear optimization? How would that depend on whether or not the user is able to code second derivatives for their problem?
- Any other comments...

The list below summarizes the deliverables in terms of the code:

- A complete software package as described above.
- A single script that runs all experiments in "Table: Summary of Results".
- A single script that runs your algorithm of choice on the Rosenbrock function, with your optimal choice of parameters.

Project Grading

- Do not be discouraged if you cannot get all of your algorithms to solve all problems. **It is better to code some of the algorithms correctly than to code all of them poorly.** Your grade for the project will be based on the merits of your (well-commented!) code as well as the clarity of presentation in your report.
- Project grades will be based on the code and your report. Your report will be graded for clarity of presentation, depth of analysis and insights, and the quality of your numerical results. Your code will be evaluated based on efficiency, clarity, and of course comments! Moreover, we might est your code on a “secret" set of problems to evaluate the overall performance.
- *Bonus points:* The 2 teams with the best reports and the 2 teams with the best performing “algorithms of choice" will receive a bonus in their project grades.

Please ask the instructor if you have any questions about the expectations or anything else! Finally, note that you are expected to work on the code and report for this project only with your project team. Any evidence of sharing code or writing in your report will be considered as violation of the Honor code.

Some tips/suggestions for coding/debugging

- Comment your code!
- Printing output is one of the best ways to debug code
- Use simple tests to check all components (no matter how small) of your code
- Start small and simple
- Why are bugs/issues so hard to

  find? Usually, bugs/issues are small/tiny (e.g., a plus sign should have been a minus sign), and hard to find since we do not expect such a simple part of the code to be wrong. From personal experience, bugs are either where you least expect to

  find them or right in front of you in the simplest operations of the code.
- It is very easy to make mistakes when coding derivatives; much easier than getting them right the first time(in my experience). As a debugging tool, you can use techniques from Section 8.1, Numerical Optimization textbook to compute approximations of the derivatives (e.g., finite difference approximations to the derivatives) at different points and compare those to the results of your code.