ISE 417: Nonlinear Optimization, Spring 2015
Project Description

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. A LaTeX template has been posted on Course Site for you to use to write your project report.

The coding exercises that have been available all semester asked you to implement steepest descent and Newton methods with various types of line searches. They also asked you to implement a trust-region-CG method. Rather than submit each of these algorithms as separate pieces of code, you are asked to *combine* all of these algorithms into a single software package. You will also 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 software package, a user should be able to run any of your implemented algorithms with various sets of inputs using the following command:

```
>> x = packagename('problem',x,'algorithm',i);
```

Here, `packagename` is the name of your software. You can call it something descriptive, such as `optsolver`, or you can be more creative. Below is a description of the inputs. The input/output `x` should be the initial/final iterate, the latter of which will hopefully be a stationary point, or even a local minimizer.

- `'problem'`: A problem will be specified by having the function, gradient, and Hessian evaluations coded into a file. For example, if the Rosenbrock function is coded in `rosenbrock.m`, then it should be minimized with your code by replacing `'problem'` with `'rosenbrock'`. Similar to the `rosenbrock.m` file provided previously on Course Site (along with the code for the exercise on line search methods), a call to this function with option 0 should return a function value. Option 1 should return a gradient value and option 2 should return a Hessian value.
- `'algorithm'`: The algorithm to be run will be specified by the user through this input. You should structure your code so that the following options are valid:
  - `steepestbacktrack`, steepest descent with backtracking line search;
  - `steepestwolfe`, steepest descent with Wolfe line search;
  - `newtonbacktrack`, Newton's method with backtracking line search;
  - `newtonwolfe`, Newton's method with Wolfe line search;
  - `trustregioncg`, trust region method with CG subproblem solver;
  - `sr1trustregioncg`, SR1 quasi-Newton trust region method with CG subproblem solver;
  - `bfgsbacktrack`, BFGS quasi-Newton method with backtracking line search;
  - `bfgswolfe`, BFGS quasi-Newton method with Wolfe line search.
- `i`: Input parameter values will be specified by the user through the *structure* `i`. You create a structure in Matlab whenever you define a variable with a dot (i.e., `.`) in its name. For example, your code *must* allow the user to specify the maximum number of iterations (call it `maxiter`) and the optimality tolerance for the gradient (call it `opttol`). Then, a particular call to your program may be the following:

```
>> i.maxiter = 1e+3;
>> i.opttol = 1e-6;
>> x = optsolve(‘rosenbrock’,x,‘newtonwolfe’,i);
```

Note that the structure `i` includes both parameters, and so both can be passed through your code simply by passing `i`. The termination check in your code may now look like the following:

```
if k > i.maxiter || norm(g) <= i.opttol*g0, break; end;
```

(Here, `g0` is assumed to be $\max\{1, \|\nabla f(x_0)\|\}$ so that an appropriate relative stopping criterion is used.) A complete list of parameters that you are required to allow the user to specify is the following. You are allowed to define more parameters, if you wish, but at least all of these are necessary. Any values that the user does not specify should be set to a default value of your choice. (In other words, a user should not be forced to specify any of these values in order to run your code. If they choose not to set a value, then your code must still run—with the default values that you have specified.)

- `opttol`: optimality tolerance, i.e., the value such that if $\|\nabla f(x_k)\|$ is less than `i.opttol*g0`, then your code should terminate as an (approximate) solution has been found;
- `maxiter`: iteration limit, i.e., the value such that if k is more than `i.maxiter`, then your code should terminate as the iteration limit has been reached;
- `c1ls`, `c2ls`: parameters for the Armijo and curvature line search conditions;
- `c1tr`, `c2tr`: parameters for the trust region radius update;
- `cgopttol`: CG optimality tolerance;
- `cgmaxiter`: CG iteration limit;
- `sr1updatetol`: tolerance for SR1 Hessian approximation updates; see (6.26) in N&W;
- `bfgsdamptol`: tolerance for BFGS Hessian approximation updates; see (18.14) in N&W.

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

As the intent of this project is for you to create a software package, you should not repeat code. In particular, it is **not** acceptable if you create an `if` statement in your main program and write separate code for each algorithm, such as in the following (i.e., **the following is bad code!**):

```
if strcmp('algorithm','steepestbacktrack')==1

 // all the code for steepest descent with a backtracking line search...

elseif strcmp('algorithm','steepestwolfe')==1

  // all the code for steepest descent with a Wolfe line search...

elseif ...
```

The problem with coding like this is that you'll have to repeat certain procedures many times, which often leads to errors (bugs); e.g., every case above is going to require a termination check, even though the same termination check should be used for each algorithm. Thus, instead of structuring your main program in this manner, you should structure it so that no matter which algorithm is being run, all calls will follow the same series of steps. Indeed, it would be preferable to structure the main program as a series of generic steps (similar to pseudocode), with all of the details relegated to subroutines, as in the following (it is assumed here that within the `stepacceptance` function, either a line search or trust region procedure will be run):

```
[d,D] = searchdirection(g,H,i);
[a,f] = stepacceptance(p,x,f,d,D,i);
```

Your code need not look exactly like this, but this is the main idea of how to structure your code.

A few more comments and guidelines:

- The `newtonbacktrack`, `newtonwolfe`, `bfgsbacktrack`, and `bfgswolfe` algorithms all require that the (approximate) Hessian is positive definite in order to ensure that the search direction will be one of descent for f at x_k. For the Newton algorithms, this should be ensured by modifying the Hessian if the eigenvalues are not all positive. Here is a block of code that you may use to perform this modification (assume `H` has been computed as the exact Hessian and `n = length(x)` is the number of variables):

```
xi = 1e-4;
while min(eig(H))<0
  H = H + xi*eye(n);
  xi = 10*xi;
end
```

  (Note that the additions to `H` grow larger within the `while` loop, since otherwise only incrementing the eigenvalues by `1e-4` each time may make the code run for a long time! Also note that computing eigenvalues can be expensive/slow for large problems, but it is fine for our purposes.) For the BFGS quasi-Newton algorithms, rather than performing modifications, you should implement the *damped* BFGS update discussed in class; see pg. 537 of N&W.

- The `sr1trustregioncg` algorithm should be exactly the same as the `trustregioncg` algorithm, except that the Hessians should be approximated using SR1 updates. Recall equation (6.26) in N&W, which imposes a condition under which the update may be skipped.

- It is a strict requirement that **your code must be well-commented**. A good rule of thumb is that I should be able to follow all of the steps in your code without looking at any of the code at all! I should be able to follow everything simply by reading the in-line comments. You can see examples of comments that I have provided in the coding examples that have been available on Course Site.

- You are encouraged to augment the output produced by the code. You should leave all of the quantities currently being printed during every iteration (iteration number, objective value, gradient norm, etc.) in the code that was posted earlier in the course (see `steepestdescent.zip`), but you may decide to print more information. For example, you may want to print the trust region radius during every iteration of a trust region method or the number of CG iterations required in a method that employs CG. That may help in finding errors in your code, since we know that (under certain assumptions) the trust region radius should stay bounded and away from zero, and CG should not require more than n iterations to produce a good solution (though it might take more due to finite precision arithmetic). **One thing that you are required to add to your code is a summary to be printed at the end of the output.** The summary should include a statement of the result (i.e., whether the problem was solved, the iteration limit was reached, etc.), the final objective value, the final norm of the gradient, the number of function evaluations, the number of gradient evaluations, the number of Hessian evaluations, the number of linear systems that needed to be solved, and the CPU time. Matlab reveals CPU time through the `tic` and `toc` commands. Place `tic` as the first line of your main program. This will start the clock. Then, whenever you invoke the command `toc`, its value will be the time (in seconds) elapsed since `tic` was called.

Test problems

Four problems that you are asked to solve with your various algorithms are posted on Course Site (see `project.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!

Project report

You are required to submit a written report with your software containing the following:

- Summarize each algorithm in a few sentences each. Your descriptions should include whether it is a trust region or line search algorithm, how the steps are computed, how nonconvexity is handled, etc.
- Provide a table of default input parameters 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.
- Provide a table of the numbers of iterations, function evaluations, gradient evaluations, and CPU seconds required to solve each of the four 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. Be sure to indicate the starting points you used.
- 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?
- 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?

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. So that I can easily run your code with the default parameters that you have specified, you are required to provide a file that will run all of your algorithms with the default parameters for a specific problem and a specific starting point. In particular, this file should be structured in the following way, where it is assumed that `problem` is the problem name; e.g., the first run will be `>> runner('rosenbrock',x);` for some `x`):

```
function runner(problem,x)

% Set common input parameters for all algorithms
i.opttol = 1e-6;
i.maxiter = 1e+3;

% Set default input parameters for steepest descent with backtracking line search
i.c1 = 0.1;
i.c2 = 0.9;

% Run steepest descent with backtracking line search
x = optsolver(problem,x,'steepestbacktrack',i);

% Set default input parameters for steepest descent with Wolfe line search...
```

Note that I get to choose the starting point when I run your code!

In addition to the provided problems, I also plan to run everyone's code on a "secret" set of problems that I will not provide (at least, not until all projects have been submitted). I will summarize the results and post them on Course Site after the projects have been submitted and graded. A bonus will be awarded to the person who codes the best algorithm for these problems!

Please ask me if you have any questions about my expectations or anything else!

(Finally, note that you are expected to work on the code and report for this project *on your own*. If there is any evidence of sharing code or writing in your report, then there will be *serious* unfortunate consequences.)