UNIVERSITÄT BERN

Applied Optimization
Exercise 7 - L-BFGS and Gauss Newton Method

November 10, 2021

Hand-in instructions:

Please hand-in **only one** compressed file named after the following convention: `Exercise`n`-GroupMemberNames.zip`, where n is the number of the current exercise sheet. This file should contain:

- The complete code folder except for the `build` subfolder. Make sure that your code submission compiles on your machine. Zip the code folder.
- A `readme.txt` file containing a description on how you solved each exercise (use the same numbers and titles) and the encountered problems.
- Other files that are required by your `readme.txt` file. For example, if you mention some screenshot images in `readme.txt`, these images need to be submitted too.
- Submit your solutions to ILIAS before the submission deadline.

Secant Equation (1 pts)

The secant equation is:

$$B_{k+1}s_k = y_k,$$

where $s_k = x_{k+1} - x_k, y_k = \nabla f_{k+1} - \nabla f_k$, and B_{k+1} is an approximation of the Hessian $\nabla^2 f_{k+1}$.
Prove that the secant equation is valid to find the next hessian approximation B_{k+1}

Curvature Condition (1 pts)

Show that the curvature condition $s_k^T y_k > 0$ is required for the Quasi-Newton method. Prove that the curvature condition is satisfied when the line search algorithm for the Wolfe conditions is used.

Programming (12 pts)

L-BFGS (4 pts)

The framework of the L-BFGS algorithm that is described in the lecture slides is implemented in the `solve(...)` function in the `Algorithms/LBFGS.hh` file. Understand by yourself the L-BFGS code and complete the functions `two_loop_recursion(...)` and `update_storage(...)`. For the step length, you can use the `backtracking_line_search` in the `LineSearch.hh`. Hint: whenever the back-tracking line search fails to find a step length that meets the curvature condition, you need to skip the update. Make use of the global variables in the class for implementation.

Gauss-Newton Method (6 pts)

In least-squares problems, the objective function f has the following special form:

$$f(x) = \frac{1}{2}\sum_{j=0}^{m} r_j^2(x),$$

where each $r_j^2(x)$ is a smooth function from $\mathbb{R}^n$ to $\mathbb{R}$. We can assemble the individual components r_j to a vector $r : \mathbb{R}^n \to \mathbb{R}^m$, as follows: $r(x) = (r_1(x), r_2(x), ..., r_m(x))^T$. Then f can be expressed as $f(x) = \frac{1}{2}||r(x)||_2^2$. A simple algorithm to solve the least-square problem is the Gauss-Newton method. Instead of solving the standard Newton equations $H\Delta x_k = -\nabla f(x_k)$, we solve the following system to obtain the search direction Δx_k:

$$J_k^T J_k \Delta x_k = -J_k^T r_k.$$

Gauss-Newton approximate the Hessian with $J^T J$, where

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right], j = 0, 1..., m, i = 0, 1, ..., n$$

is the derivative of $f(x)$ called Jacobian which is a $m \times n$ matrix.

The task is to implement the Gauss-Newton method to solve the mass spring system. First, start by implementing the function r_j, represented in code as the usual `eval_f(...)` and `eval_gradient(...)`, for:

- `SpringElement2DLeastSquare.hh`
- `SpringElement2DWithLengthLeastSquare.hh`
- `ConstrainedSpringElementLeastSquare.hh`

Study the code and the comments in those files to help you in this implementation.

The next step is to assemble those elements into a MassSpringSystem, by filling the functions in the new file `Functions/MassSpringProblem2DLeastSquare.hh`. We advise you to follow these steps:

1. Implement `eval_r(...)`, which evaluates all $r_j(x)$ functions and gathers them in the $r(x)$ vector mentioned above.

2. Implement `eval_jacobian`, which evaluates all $\frac{\partial r_j}{\partial x_i}$ and gathers them in the Jacobian matrix, as shown above. Those two first steps are the hardest of this exercise so make sure to thoroughly read the in-code comments.
3. Implement `eval_f(...)`, which computes $f(x)$, as shown above.
4. Implement `eval_gradient(...)`, which computes the gradient $J^T r$.
5. Implement `eval_hessian(...)`, which computes the hessian $J^T J$.

Once the implementation is completed, the standard newton solver can be used to solve the problem. You can run your implementation with multiple random Mass Spring Systems using `GaussNewton/main.cc` and the corresponding executable `GaussNewton`. As usual, we also provide a set of unit-tests, which you can run with the executable `GaussNewton-test`.

Comparison of different methods (2 pts)

With the provided constrained spring elements and the random initialization, use different algorithms, including gradient descent, newton, projected newton, L-BGFS and Gauss Newton to optimize the mass spring system with both spring element types of different dimensions ($5 \times 5, 10 \times 10, 20 \times 20$). For the spring element with length , if solved with standard newton's method, use the spring element class with positive definite hessian, i.e. set the function index to 2. Compare the results from different algorithms with the default parameters within 1000000 iterations. Make a table with the statistics, including the number of iterations, run time, objective function value, and identify the fastest algorithm. When testing the L-BFGS algorithm, choose a m value that gives a good performance (see the range of m on the slides). The optimized spring graph can be visualized on the website.

A line search algorithm for the Wolfe conditions (Bonus (6 pts))

On page 60 of the book "Numerical Optimization", there is a description of a line search algorithm that satisfies the Wolfe conditions. Implement the algorithm in the function `wolfe_line_search(...)` in the `LineSearch.hh` file. Run the L-BFGS algorithm with this line search algorithm and compare with the back tracking line search.