AE588
Assignment 3: Unconstrained Optimization

Submission Instructions

Please submit both your report (in PDF) and source code to Gradescope. You must make two separate submissions: submit your report to `Assignment 3 (PDF)` and source code to `Assignment 3 (code)`. Your grade will be based on your report, not solely on the autograder's score. Please make sure to include all key steps, results, tables, and figures in your report. The report must be typed; handwriting, screenshots of the code outputs, or "see code" are not accepted.

Code submission
Please submit the following Python files without zipping or putting them in a folder.

- `uncon_optimizer.py`: implement your optimizer in the provided template file, and submit it without changing the file name. This script will be called by the autograder. Before the final submission, please comment out all print functions in this file.
- `prob3_3.py, prob3_4.py, prob3_5.py`, and `prob3_6.py`: runscripts for each subproblem. Autograder will not run these scripts, but you still need to submit all files.
- All the other Python files you call from `uncon_optimizer.py` and runscripts.

Please do not submit the provided test file `tests_uncon_optimizer.py`. You can resubmit your scripts as many times as you wish before the deadline.

Autograder Environment
Python 3.10.6, Numpy 1.25.2, Scipy 1.11.2, Sympy 1.12, Jax and Jaxlib 0.4.16, and Matplotlib 3.7.2.

Other packages are not installed by default but can be added upon request. Please email the GSI (shugok@umich.edu) and provide the package name, (its version), and a brief explanation of why you need it. If your script fails with a `ModuleNotFoundError`, it means you're using a package not installed on the autograder.

Problems

Objective: Develop an unconstrained optimization algorithm and apply it to test problems. Interface with an existing optimization algorithm. Learn how to present convergence plots and optimize larger dimensional problems.

3.1 (30 pts) Implement an unconstrained gradient-based optimization algorithm using the provided template file `uncon_optimizer.py`. We have discussed various line search methods and methods for determining the search direction. You can use any combination of method(s) you want, but you must develop your own code. The use of Scipy's optimizer and line search are not allowed. Your optimizer should work for any problem dimensions (i.e., the number of design variables), not just for two-dimensional problems.

The autograder will test your optimizer on two easy 2D problems, which are provided in the testing script `tests_uncon_optimizer.py`. You can run the tests on your machine to make sure that your optimizer works. Put `uncon_optimizer.py` and `tests_uncon_optimizer.py` in the same folder, and run `python tests_uncon_optimizer.py`.

3.2 (10 pts) Describe your algorithm and why/how you selected that methodology. Assume that you do not have analytic Hessians and therefore Newton's method is out of the question. It would be advisable to try more than one line search approach and/or more than one search direction method to help justify your approach.

3.3 (10 pts) Solve two 2D test problems (slanted quadratic function and 2D Rosenbrock function) using your optimization algorithms and off-the-shelf optimization software (e.g., `scipy.minimize`). The test problems are detailed at the end of this assignment. Please note that these test problems you'll be reporting for are different from the ones in `tests_uncon_optimizer.py`.

Create a table reporting the number of function calls required for convergence on both test problems. The table should include the results of all algorithms you tried for Problem 3.2 as well as the off-the-shelf optimizer. Discuss your findings.

3.4 (10 pts) Create a convergence plot for both 2D test problems. For each plot, you only need to show the result of your best algorithm. A convergence plot should show iterations on the x-axis and a convergence metric with a log scale on the y-axis. Some suggested convergence metrics include $|f - f^*|$ or $||\nabla f||_\infty$ (since f^* is usually not known).

3.5 (10 pts) Create an optimization path plot for both 2D test problems. For each plot, you only need to show the result of your best algorithm. The path should be overlaid onto function contours.

3.6 (20 pts) Solve the multidimensional Rosenbrock problem (see Appendix D of the textbook) and study the effect of increased problem dimensionality. Start with 2 dimensions and double the dimension each time up to the highest you can reasonably manage (i.e., $n = 2, 4, 8, 16, 32, 64, \ldots$). Plot the computational cost vs. the problem size. The x-axis of this plot should be the problem dimensions, and y-axis should be a computational cost metric (for example, iterations, function calls, wall time, etc.). Compare the performance of your best algorithm and off-the-shelf optimization software. Discuss your findings.

3.7 (10 pts) Discuss the main challenges you faced, lessons learned, and areas of improvement for your optimization algorithm.

3.8 (extra) The performance of your optimizer will be evaluated on undisclosed test problems ranging from 2D to 512D. Top-performing algorithms will be awarded extra points. See below for details.

Notes

- You should differentiate the test functions and provide the analytic derivatives to the optimizer. For the multidimensional Rosenbrock function, you can get the components of the gradient by taking the partial derivative of the function with respect to x_i.
- You should not get the impression from this assignment that writing your own algorithms is superior to using existing implementations. It is not. There are many excellent optimization packages out there that are robust and have great performance. Development of these tools takes many years and lots of expertise. The reason for this exercise is that there is a big difference between users who only know how to connect their code to an optimization algorithm and users who understand what's going on under the hood and what the means for their problem setup. We can discuss theory all day, but it is hard to really understand what is going on without wrestling with the details at some point.

Test Problems for 3.3, 3.4, and 3.5

1. Slanted quadratic function: Eq. (D.1) from Appendix D in the textbook, with $\beta = 1.5$.
2. Two-dimensional Rosenbrock function: Eq. (D.2) from Appendix D.

Test Procedure for 3.8

Your optimizer will be tested on 5 categories of undisclosed test functions. For each category, we will test your algorithm from a set of randomly generated starting points or for various problem dimensions. The same random points and the problem dimensions will be used for all participants. Both the function values and gradient of the function will be provided. The algorithm should be accurate. The accuracy will be measured by

$$||\nabla f(x)||_\infty < 10^{-6} \ . \tag{1}$$

A trial is one starting point for one problem. The performance of the optimizer is measured based on the number of function calls required. Each trial will receive a score of $1/n_{\text{function calls}}$ if it satisfies the accuracy check, and 0 if it does not. For each category, the trial scores will be averaged across all starting points or all problem dimensions, and then normalized by the score of the Scipy's optimizer. Your final score will be the average of your normalized scores across 5 categories.

The score represents the relative performance of your optimizer compared to the Scipy's optimizer. The goal is to achieve the largest score, which implies the smallest number of function calls across various test problems. Extra credit will be awarded to the top 10 algorithms: 10 points for number 1, 9 points for number 2, and so on.

Provided scripts

- `uncon_optimizer.py`: template file for your optimizer. Please follow the instructions in the file.
- `tests_uncon_optimizer.py`: testing script. The autograder will run very similar tests to this file for Problem 3.1. You do not need to understand or edit this script, but it shows how your optimizer will be called.

Debugging Tips:

Before testing a problem on your optimizer, I recommend you test it on an off-the-shelf optimizer. If it does not work with those, then you know there is a problem with your function or gradient. Below are general debugging tips for running any optimization problem.

- Anytime you provide gradients you should always check them. First, run the optimization without providing gradients (the optimizer will internally finite difference). Then, compare the provided gradients against finite differencing. This is easy to do yourself, but many optimizers will also check for you.
- Check your function behavior. Create parameter sweeps of every output with respect to every input and plot them before attempting optimization. Make sure things behave like you would expect and that the behavior is smooth. Does your function behavior make physical sense?
- If an optimization breaks down, investigate why. Take the point where it failed and put it in your analysis. What is the function doing? Does it blow up somewhere? Look at nearby parameter sweeps. Look at the gradients.
- I highly recommend learning how to use a debugger, which can be found in many Python IDEs. With a debugger you can step through a function line by line and see what is happening in your function. You can also accomplish this by printing things to screen, but a debugger is generally more powerful.

Tips for testing the development of your algorithm:

- Test your line search function in isolation. Give it various cases and make sure the results are as you would expect.
- Check your search direction algorithm. Try something simple to implement like steepest descent or conjugate gradient and see how it compares. If multiple search direction methods do not work, it probably means that the error is somewhere else like the line search or convergence criteria. If the simple methods work, then it is likely the problem is in your approximate Hessian update.
- Again, use a debugger. Look at how your function value changes, look at the gradients, look at α, look at where the optimizer is moving to.