AE588
Assignment 2: Models and Line Search

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 2 (PDF)` and source code to `Assignment 2 (code)`. Your grade will be based on your report, not solely on the autograder's score. Please make sure to include all figures and key results in your report. The report must be typed; not handwritten.

Code submission

Name your runscripts as `prob2_1.py`, `prob2_2.py`, and `prob2_3.py` for Problems 2.1, 2.2, and 2.3, respectively. Each script should solve all subproblems (if applicable). Upload these Python files to Gradescope without putting them into a folder.

The autograder will execute your scripts using the following commands: `python prob2_1.py`, `python prob2_2.py`, and `python prob2_3.py`. As long as you have these files, feel free to use other Python files. For instance, you might define a function in `func.py` and import it in `prob2_1.py`. All Python files need to be submitted to Gradescope.

Code outputs

For each subproblem, please print key results to `stdout` (by simply using the `print` function) in a human-readable format. For example, print x and f values at the converged solution, number of iterations, etc. There is no specific format, but please be concise and turn off debug prints. If a subproblem only requires plotting, you don't need to print anything to `stdout`.

Please note that passing all tests does not imply you will get full credit. The autograder only tests if your code runs without errors; however, it does not check whether the outputs of your code are correct. The outputs will be manually checked by a (human) grader. Also, the autograder does not show figures. If the tests fail in your final submission, points will be deducted. 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, 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

2.1 (30 pts) Kepler's equation, which we mentioned in Chapter 2, defines the relationship between a planet's polar coordinates and the time elapsed from a given initial point and is

$$E - e\sin(E) = M,$$

where M is the mean anomaly (a parameterization of time), E is the eccentric anomaly (a parameterization of polar angle), and e is the eccentricity of the elliptical orbit.

(a) Implement Newton's method and find E when $e = 0.7$ and $M = \pi/2$. What is the maximum possible precision you can achieve?

(b) Devise a fixed-point iteration to solve the same problem.

(c) Compare the number of iterations and rate of convergence.

(d) Evaluate the robustness of each method by trying different initial guesses for E.

(e) Plot E versus M in the interval $[0, 2\pi]$ for $e = [0, 0.1, 0.5, 0.9]$. Optional: interpret your results physically.

(f) Produce a plot showing the numerical noise by perturbing M in the neighborhood of $M = \pi/2$ with $e = 0.7$ using a solver convergence tolerance of $|r| \leq 0.01$. Note: you might want to randomize the starting points for the solver.

2.2 (10 pts) Consider the function:

$$f(x_1, x_2) = x_1^4 + 3x_1^3 + 3x_2^2 - 6x_1x_2 - 2x_2.$$

Find the critical points analytically and classify them. Where is the global minimum? Plot the function contours to verify your results.

2.3 (60 pts) Implement the two line search algorithms from Sec. 4.3, such that they work in n dimensions (x and p can be vectors of any size).

(a) As a first test for your code, reproduce the results from Examples 4.8 and 4.9 of the book and plot the function and iterations for both algorithms. For the line search that satisfies the strong Wolfe conditions, reduce the value of μ_2 until you get an exact line search. How much accuracy can you achieve?

(b) Test your code on another easy two-dimensional function, such as the bean function from Example 4.11, starting from different points and using different directions (but remember that you must always provide valid descent direction, otherwise the algorithm might not work!). Does it always find a suitable point? *Exploration*: Try different values of μ_2 and ρ to analyze their effect on the number of iterations.

(c) Apply your line search algorithms to the two-dimensional Rosenbrock function and then the n-dimensional variant with $n = 6$. Again, try different points and search directions to see how robust the algorithm is, and try to tune μ_2 and ρ.