Numerical Optimization with Python

Programming Assignment 01

In this exercise we will:

- Implement Line Search minimization with several methods.
- Test it on some examples and visualize its performance
- Organize our project and use one of Python's testing frameworks

1. **Instructions for project organization:**
    a. Your numerical optimization project should have two directories: `src` and `tests`.
    b. Your `src` directory should have two modules: `unconstrained_min.py` (your algorithms) and `utils.py` (common functions such as plotting, printouts to console, etc.)
    c. Your `tests` directory should have two modules: `test_unconstrained_min.py` and `examples.py`

2. **Requirements for implementing your line search minimization:**
    a. Your implementation can be either a class or a function, that should support, according to user's selection:
        i. Gradient descent as well as Newton search directions
        ii. Fixed (user specified) step length as well as Wolfe condition with backtracking (no need to implement the second Wolfe condition shown in class).
    b. The minimization function (or class method) should be implemented in `unconstrained_min.py`. It should take the following parameters: `f, x0, step_len, obj_tol, param_tol, max_iter.`
    c. `f` is the function minimized, `x0` is the starting point, `step_len` is either the float constant step length or the string specifying usage of the Wolfe condition with backtracking. `max_iter` is the maximum allowed number of iterations.
    d. `obj_tol` is the numeric tolerance for successful termination in terms of small enough change in objective function values, between two consecutive iterations ($f(x_{i+1})$ and $f(x_i)$), or in the Newton Decrement based approximation of the objective decrease.

e. `param_tol` is the numeric tolerance for successful termination in terms of small enough distance between two consecutive iterations iteration locations (x_{i+1} and x_i).

f. At each iteration, the algorithm reports (prints to console) the iteration number i, the current location x_i, and the current objective value $f(x_i)$.

g. The algorithm returns the final location, final objective value and a success/failure Boolean flag.

h. Your algorithm should enable access to the entire path of iterations and objective values when done (either return them or store them in your class) for later usage in visualization.

3. **Requirements for implementing `examples.py`:**

a. The examples are the objective functions we minimize. In this exercise they are implemented as functions taking a vector `x` and a `bool` flag, specifying whether or not Hessian evaluation is needed.

b. NOTE: do not evaluate Hessians if not needed!

c. There are three return values `f, g, h`: the scalar function value, the gradient vector and the Hessian matrix (if needed only), evaluated at `x`, respectively.

d. Implement three quadratic examples: $f(x) = x^T Q x$ for the following Q's:

i. $Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ (contour lines are circles)

ii. $Q = \begin{bmatrix} 1 & 0 \\ 0 & 100 \end{bmatrix}$ (contour lines are axis aligned ellipses)

iii. $Q = \begin{bmatrix} \frac{\sqrt{3}}{2} & -0.5 \\ 0.5 & \frac{\sqrt{3}}{2} \end{bmatrix}^T \begin{bmatrix} 100 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} & -0.5 \\ 0.5 & \frac{\sqrt{3}}{2} \end{bmatrix}$ (contour lines are rotated ellipses)

e. Implement the Rosenbrock function: $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$. Contour lines are banana shaped ellipses. This is a famous optimization benchmark which is challenging to test your implementation on.

f. Implement a linear function $f(x) = a^T x$ for some nonzero vector a you choose. Contour lines are straight lines.

g. Implement the function $f(x_1, x_2) = e^{x_1 + 3x_2 - 0.1} + e^{x_1 - 3x_2 - 0.1} + e^{-x_1 - 0.1}$. Contour lines look like smoothed corner triangles (example is adopted from Boyd's book, p. 470, example 9.20).

4. **Requirements for implementing `utils.py`:**
   a. A utility to create a plot, that given an objective function and limits for the 2D axes, plots the contour lines of the function.
   See https://matplotlib.org/3.5.1/api/_as_gen/matplotlib.pyplot.contour.html for a possible implementation choice. Make sure you chose proper levels and limits so the picture is clearly showing the interesting area of the problem. Make a clear title that describes the plotted function.
   b. If also provided algorithm paths, the above plotting utility should plot the paths and their names in the legend.
   c. A utility that plots function values at each iteration, for given methods (on the same, single plots) to enable comparison of the decrease in function values of methods.

5. **Requirements for implementing `test_unconstrained_min.py`:**
   a. See the very first, basic example in https://docs.python.org/3/library/unittest.html for test module structure using Python's `unittest` framework.
   b. For each of the functions in your examples file, your testing module should trigger minimization with both methods, and with backtracking Wolfe conditions for step length.
   c. The test run should create two plots for each example:
      i. The contour lines of the objective with iteration paths of both methods
      ii. The function values vs. the iteration number for both methods

6. **Submission instructions:**
   a. Submit a single file, your report in PDF format, and send a GitHub link to your code by e-mail.
   b. Your report should include the plots created by each of your tests (contours with iteration paths, and function value vs. iteration plots)
   c. For each test – your report should include the last iteration report printed to console (the details of your final iterate and success/failure algorithm output flag).

7. **Important tips and other helpful info:**
   a. Choose Initial points for all your examples to be: $x_0 = [1,1]^T$, except for the Rosenbrock example, for which $x_0 = [-1,2]^T$

b. Choose numeric tolerances for your termination to be 10^{-8} for step tolerance and 10^{-12} for objective function change tolerance.
c. Allow max iterations 100 for all your examples, except for Gradient Descent with Rosenbrock example, for which you should allow 10,000.
d. Use the Wolfe condition constant 0.01 with backtracking constant of 0.5.
e. Regarding all the above constants: play with several values to get a feel of their effect on the behavior, before you submit your final run!

Good luck!