Numerical Optimization with Python

Assignment 02 - programming part: Line search methods and constrained optimization

In this exercise we will:

- Extend our first assignment from Gradient Descent to Newton and Quasi-Newton methods
- Add step size selection at each iteration according to the backtracking Wolfe conditions
- Extend our plotting and monitoring of iterative algorithms, so we can monitor the decrease in function values and the rate of convergence
- Implement interior point method for small constrained optimization problems

Instructions for unconstrained minimization:

1. Extend your gradient descent function from HW01, that minimizes unconstrained functions, to a more general `line_search()` function, which:
    a. Takes a new parameter: `dir_selection_method` that is one of three strings: `'gd'`, `'nt'`, `'bfgs'`, to specify the method for selecting the direction for line search (gradient descent, Newton or BFGS, respectively).
    b. Takes new parameters for the backtracking and Wolfe condition to accept step lengths: `init_step_len`, `slope_ratio` and `back_track_factor`.
2. Implement the new search direction computation methods: `bfgs_dir()` and `newton_dir()`, to be called by your line search method. Notes:
    a. Define the inputs to these functions as required by the method. Do not evaluate Hessian matrices if not needed.
    b. For the BFGS method you will require the previous as well as current location/gradient. Manage your loop carefully to support that.
    c. Do not invert matrices! This is unneeded and expensive. Solve the equivalent linear system using an existing linear equations solver, such as `numpy.linalg.solve()`
3. Implement the first Wolfe condition for sufficient decrease appearing in slides 16-19 of lecture 3. Do not implement the second condition (curvature condition from slide 20-23). Use $c_1 = 1e^{-4}$ (`slope_ratio`), use backtracking to check the condition, from `init_step_len=1.0` and updating it until the condition is satisfied with `back_track_factor=0.2`.

4. Change the name of your `test_gradient_descent.py` from HW01 to the more general `test_unconstrained_min.py`, and to each of the tests appearing there, solve the same problem but now with Newton method and with BFGS.
5. Plot the same plots you did for HW01, that present the path of the algorithm over the objective contour lines. You may skip the linear example from HW01.
6. Add a new plot utility to your `utils.py`, which shows the number of iterations on the horizontal axis and the objective function value in the vertical axis. Add such a plot to each of your runs in your submitted report.
7. To support the above, extend each of your examples in the `examples.py` file, so that it can return the Hessian as well, not only the value and gradient value. However, make sure you do not compute a Hessian matrix if it is not needed. Have a flag to control that in the evaluation of the function call of each example.

Instructions for constrained minimization:

1. To your `src` directory, add a new module, `constrained_min.py`
2. Implement `interior_pt(func, ineq_constraints, eq_constraints_mat, eq_constraints_rhs, x0)` which minimizes the function `func` subject to the list of inequality constraints specified by the Python list of functions `ineq_constraints`, and to the affine equality constraints $Ax = b$ that are specified by the matrix `eq_constraints_mat`, and the right hand side vector `eq_constraints_rhs`. The outer iterations start at `x0`.
3. Use the log-barrier method studied in class, with the initial parameter $t = 1$ and increase it by a factor of $\mu = 10$ each outer iteration.
4. To your `tests` directory, add a module `test_constrained_min.py` and define, using the `unittest` framework as in HW01, the function `test_qp()`,`test_lp()` that will demonstrate solutions for a quadratic programming example and a linear programming example.

5. To your `examples.py` file, add the functions and the definition of the matrix and vector, to enable `test_qp()` use them for solving the following problem:

$$\min x^2 + y^2 + (z+1)^2$$
$$\text{Subject to: } x + y + z = 1$$
$$x \geq 0$$
$$y \geq 0$$
$$z \geq 0$$

Note: the problem finds the closest probability vector to the point $(0,0,-1)$. Choose an initial interior point $(0.1, 0.2, 0.7)$, and do not implement a phase I method for finding a strictly feasible point in this exercise.

6. To your `examples.py` file, add the functions to enable `test_lp()` use them for solving the following problem:

$$\max x + y$$
$$\text{Subject to: } y \geq -x + 1$$
$$y \leq 1$$
$$x \leq 2$$
$$y \geq 0$$

Note: the problem finds the upper right vertex of a planar polygon. You only have inequality constraints here, hence at each outer iteration you will solve an unconstrained problem. Choose an initial interior point $(0.5,0.75)$, and do not implement a phase I method for finding a strictly feasible point in this exercise.

7. For both examples above, plot the final candidate, the objective and constraint values, and plot the feasible region and the path taken by the algorithm. Note: in both cases the feasible region is a polygon, but in the first example it is a triangle to be plotted in 3D space, and the path is in 3D space, there are several options to do that, here:
https://matplotlib.org/2.0.2/mpl_toolkits/mplot3d/tutorial.html
Submit the required plots and final iterates in a PDF file to the course site, and your code should be sent over email. If you learn to work with Git and send a link to a GitHub repo to share your code, you will receive a 5% bonus on this project grade (bonus idea credit: goes to Alon Mannor)