ORIE 6365 — Continuous Optimization: Algorithms and Complexity, Spring 2026

Practical Assignment

Released: April 13, 2026 *Due: May 5, 2026 (extended)*

In this assignment, you are invited to implement and test first-order algorithms for solving the following unconstrained minimization problem:

$$\min_{x\in\mathbb{R}^n}\Big[\; f(x) \;=\; \tfrac{1}{m}\sum_{i=1}^{m}\ell(\langle a_i, x\rangle - b_i) + \psi(x)\;\Big], \tag{1}$$

which often appears in applications with machine learning and statistics. In problem (1),

$$\mathcal{D} \;=\; \Big\{a_1,\ldots,a_m \;\in\; \mathbb{R}^n,\; b_1,\ldots,b_m \;\in\; \mathbb{R}\Big\} \tag{2}$$

is the data that might come from different sources,

$$\ell : \mathbb{R} \to \mathbb{R}$$

is a convex *loss function*, which is part of the model, and ψ is a possible regularizer.

The full solution to this assignment consists of two components:

- A source code in Python[1] of *all parts* of your implementation (data generator, optimization algorithms, and the code that you used for running all the experiments and producing the plots), *well organized* and *readable*.
- A report in PDF, clearly explaining the experimental setup, generated plots, and *discussion of the results*. You should check if your experimental results match the theory, or provide a reasonable explanation if it is not the case (e.g., which assumptions were not satisfied, so an algorithm does not achieve a predictable behavior).

Please organize all your files and report into a single ZIP file and submit via the gradescope.

The assignment consists of the following key parts:

1 Data Generation

Implement a function that generates a random instance of data (2), with the following signature[2]:

```
def generate_data(n, m, sigma, seed=6365)
    ...
    return (A, b)
```

where $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ is a generated instance (`numpy arrays`), and $\sigma > 0$ is a parameter of *ill-conditioning* of the data. Namely, we want to have a control over the following quantity:

$$\sigma \;=\; \lambda_{\max}(B)/\lambda_{\min}(B), \tag{3}$$

where $B := A^\top A$. So, the way you generate A and b should satisfy (3) up to a reasonable precision, and you should return the same random data for different runs of the function with a fixed `seed`.

[1] Please contact me if you strongly prefer to substitute Python with another programming language in your implementation.

[2] If you want, you can use additional an parameter with a default values set in all functions.

2 Algorithms for Smooth and Strongly Convex Problem

Implement the following methods for solving problem (1) with a smooth loss ℓ and strongly convex regularizer, for some $\mu \geq 0$:

$$\min_{x \in \mathbb{R}^n} \Big[f(x) \quad = \quad \frac{1}{m} \sum_{i=1}^{m} \ell(\langle a_i, x \rangle - b_i) + \frac{\mu}{2} \|x\|_2^2 \Big]. \tag{4}$$

2.1 The gradient method

Please use the following signature:

```
def gradient_method(A, b, loss, mu, x_0, n_iters, L_0, adaptive=False):
        ...
        return x_sol, history, status
```

where

- `A`, `b` are given data for the problem instance;
- `loss` is a string with the following possible values: `loss = 'quadratic'` which corresponds to linear regression loss: $\ell(t) = \frac{1}{2}t^2$ and `loss = 'logistic'` which corresponds to the logistic regression loss: $\ell(t) = \log(1 + e^t)$;
- `mu` is the value of the regularization parameter $\mu \geq 0$;
- `x_0` is the initial point, `n_iters` is the number of iterations to run the method;
- `L_0` is an estimate of the Lipschitz constant of the gradient;
- `adaptive=True or False` corresponds to whether using an adaptive search for estimating the Lipschitz constant at each iteration of the method, or using the constant value `L_0` for all iterations (constant step-size).

The function should return:

- `x_sol` which is a point with the best function value among observed iterates;
- `history` should contain the following information:
  - `history['func']` is the list of function values among all main iterates $\{x_k\}_{k \geq 0}$
  - `history['grad']` is the list of gradient norms among all main iterates $\{x_k\}_{k \geq 0}$
  - `history['time']` is the list of time snapshots taken at the start of every iteration;
  - `history['mat_vec']` is the list of total numbers of matrix-vector products Ah and $A^\top h$ used up to each iteration (cumulative statistics);

A good implementation should effectively use matrix and vector operations from `numpy` and `linalg`, avoiding auxiliary loops for computing the function value and the gradient as much as possible. Please also note that all parameters sent to the function *should remain unchanged* (in particular, x_0 should remain unchanged after running the method);

2.2 The fast gradient method

Please use the following signature:

```
def fast_gradient_method(A, b, loss, mu, x_0, n_iters, L_0, adaptive=False):
            ...
            return x_sol, history, status
```

with the same meaning of the parameters as for the basic gradient method.

3 Algorithm for Non-Smooth Problems

Implement the projected subgradient method for solving problem (1) with smooth and non-smooth losses and a ball constraint, for some $R > 0$:

$$\min_{x \in \mathbb{R}^n \,:\, \|x\|_2 \leq R} \Big[\; f(x) \;\; = \;\; \tfrac{1}{m} \sum_{i=1}^{m} \ell(\langle a_i, x \rangle - b_i) \; \Big] \tag{5}$$

Please use the following signature:

```
def subgradient_method(A, b, loss, R, x_0, n_iters, gamma, normalized=False):
            ...
            return x_sol, history, status
```

where the differences with the smooth case are the following:

- `loss` is a string with possible values: `loss = 'quadratic'`, `loss = 'logistic'`, or `loss= 'l1'` which corresponds to $\ell(t) = |t|$;
- `R` is a radius of the ball;
- `gamma` is a step-size parameter in the subgradient method;
- `normalized=True or False` corresponds to whether normalize the subgradient direction by its Euclidean norm, or not;

The function should return the same result as in the smooth case, but use for `history['grad']` the norms of the subgradients used over the iterations.

4 Experiments

Consider several instances of the randomly generated data for fixed moderate values n and m (around 100 and around 1000) and different values of the condition number $\sigma = 10, 10^3, 10^5$. For each instance of the generated data, perform the following set of experiments:

1. Run both the gradient method and the fast gradient method on quadratic and on logistic loss, for different values of $\mu \geq 0$, starting from $\mu = 0$, with a constant value of the regularization parameter L (chosen according to the theory).

   Plot the graphs of the functional residual $f(x_k) - f^\star$ and the gradient norm $\|\nabla f(x_k)\|$ (in logarithmic scale), versus the iteration number k, the computational time `history['time']` and the total number of matrix-vector products `history['mat_vec']` (in standard linear

scale). For estimating the minimal function value $f^\star$ you might use either the best function value found after the launch of all the algorithms, or use the built-in function `scipy.optimize` in Python to estimate it.

Try answering the following questions:

What is the rate of convergence for each of the method?

How is the convergence rate affected by the strong convexity parameter μ and the data condition number σ?

2. Repeat the same set of experiments but enabling the adaptive search: `adaptive = True`.

   Does adaptive search help for a faster convergence?

3. Run the subgradient method on quadratic, logistic, and `l1` losses, using normalized and non-normalized step-sizes, with γ chosen to be a constant, according to theory, for few different values of the radius R and the number of iterations. Try answering the following questions:

   What is the rate of convergence of the subgradient method? Compare the obtained plots with the previous ones for the methods on smooth instances.

   Does normalization of subgradients help for a faster convergence?

5 Bonus

This is a bonus part for extra credit. The core part of the assignment is out of 100 points. The following part weights out of 15 additional points.

5.1 Newton's Method

Implement the Newton method with the gradient regularization:

$$x_{k+1} \quad = \quad x_k - \left(\nabla^2 f(x_k) + M_k \|\nabla f(x_k)\| I\right)^{-1} \nabla f(x_k), \qquad k \geq 0, \tag{6}$$

for the instance of (4) on logistic loss. Try the following choices of the parameter $M_k \geq 0$:

- Pure Newton's method: $M_k \equiv 0$;
- The constant choice: $M_k \equiv M_f$, where $M_f \geq 0$ is the parameter of quasi-self-concordance of the objective;
- The adaptive search: which finds the value of M_k, similarly to the adaptive search in gradient methods, such that it ensures the following sufficient progress of each iteration $k \geq 0$ with the condition:

$$f(x_k) - f(x_{k+1}) \quad \geq \quad \frac{\|\nabla f(x_{k+1})\|^2}{2M_k \|\nabla f(x_k)\|}.$$

1. Compare each of these strategies for choosing M_k on several different instances of the problem.
2. Compare the best instance of method (6) with that ones for the gradient and fast gradient methods, for few values of σ, μ and the problem dimension n, in terms of the oracle complexity and in terms of the total computational time.