UNIVERSITY OF MICHIGAN

ROB422/EECS465
Introduction to Algorithm Robotics

HW2 - Convex Optimization

Junhao TU

October 4, 2023

Question 1

A single point can be regarded as convex.

According to the definition of convexity, a set is convex if for any two points x, y in it, the segment $\theta x + (1-\theta)y$ for all θ in [0,1] lies within the set. For a single point p, this segment is just p, which is obviously within the set, making the single point convex.

Question 2

$f(x) = (|2-5x| + 2x + 8e^{-4x}) - 1$ is convex. The reasons are as below:

- $|2-5x|$ is convex since the absolute value function is convex and the inner function is affine.
- $2x$ is affine, hence both convex and concave.
- e^{-4x} is a common convex function, and this can be confirmed by its positive second derivative. Therefore, the term of $8e^{-4x}$ is also a convex function.

The sum of convex functions remains convex. Combining $|2-5x|$, $2x$, and $8e^{-4x}$ yields a convex result. Adding or subtracting a constant doesn't alter convexity. Thus, $f(x) = (|2-5x| + 2x + 8e^{-4x}) - 1$ is convex.

Question 3

The problem can be written in the standard form as:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f_0(x) = 4x_2 - 3x_1 + 3 \\ \text{subject to} \quad & f_1(x) = x_2 + 3 \leq 0 \\ & f_2(x) = 2x_1 - 5x_2 + 2 \leq 0 \\ & f_3(x) = -x_2 + 5 \leq 0 \\ & h_0(x) = x_1 - x_2 - 6.3 = 0 \end{aligned}$$

Due to this being a linear programming problem, a more concise form can be also presented as:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x + d \\ \text{subject to} \quad & Gx \leq h \\ & Ax = b \end{aligned}$$

Where:

$$c = \begin{bmatrix} -3 \\ 4 \end{bmatrix}, \quad d = 3, \quad G = \begin{bmatrix} 0 & 1 \\ 2 & -5 \\ 0 & -1 \end{bmatrix}, \quad h = \begin{bmatrix} -3 \\ -2 \\ -5 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad b = 6.3$$

Question 4

Make $3x^2 - 2$ and $2x - 1$ equal to obtain intersection:

$$3x^2 - 2 = 2x - 1$$

Therefore, there are two intersection points, which are $x_1 = -\frac{1}{3}$ and $x_2 = 1$.

For $x_1 = -\frac{1}{3}$: Derivative of $3x^2 - 2$ is $6x$, so at x_2: $6(-\frac{1}{3}) = -2$. Derivative of $2x - 1$ is 2, which remains constant.

$$\partial f(-\frac{1}{3}) = [-2, 2]$$

For $x_2 = 1$: Derivative of $3x^2 - 2$ is $6x$, so at x_1: $6(1) = 6$. Derivative of $2x - 1$ is 2, which remains constant.

$$\partial f(1) = [2, 6]$$

So, the subdifferential $\partial f(x)$ contains more than one subgradient at $x_1 = -\frac{1}{3}$ and $x_2 = 1$, and the subdifferentials at these points are [-2, 2] and [2, 6], respectively.

Question 5

a. Lagrange Dual Function

Introducing the Lagrange multipliers λ and ν The Lagrangian $L(x, \lambda, \nu)$ is then given by:

$$L(x, \lambda, \nu) = c^T x + \lambda^T (Gx - h) + \nu^T (Ax - b)$$

The Lagrange dual function $g(\lambda, \nu)$ is then:

$$g(\lambda, \nu) = \inf_x L(x, \lambda, \nu)$$

b. Dual Problem

To derive the dual problem, we need to maximize the dual function for the dual variables, subject to the constraint that the multipliers for the inequalities are non-negative:

$$\begin{array}{ll} \underset{\lambda,\nu}{\text{maximize}} & g(\lambda, \nu) \\ \text{subject to} & \lambda \geq 0 \end{array}$$

This can be written more explicitly as:

$$\begin{array}{ll} \underset{\lambda,\nu}{\text{maximize}} & \inf_x (c^T x + \lambda^T (Gx - h) + \nu^T (Ax - b)) \\ \text{subject to} & \lambda \geq 0 \end{array}$$

c. Relationship between d^* and p^*

Under the assumption that the primal problem is feasible and bounded, the strong duality holds for linear programming, which is:

$$d^* = p^*$$

Where: d^* is the optimal value of the dual problem. p^* is the optimal value of the primal problem.

The reason behind this is the fundamental theorem of linear programming which states that if the primal has an optimal solution, then the dual also has an optimal solution, and their objective function values at the optimum are equal.

Implementation 1

a. backtracking.py

b. gradientdescent.py

c. newtonsmethod.py

d. i. plot_descents.py

ii.

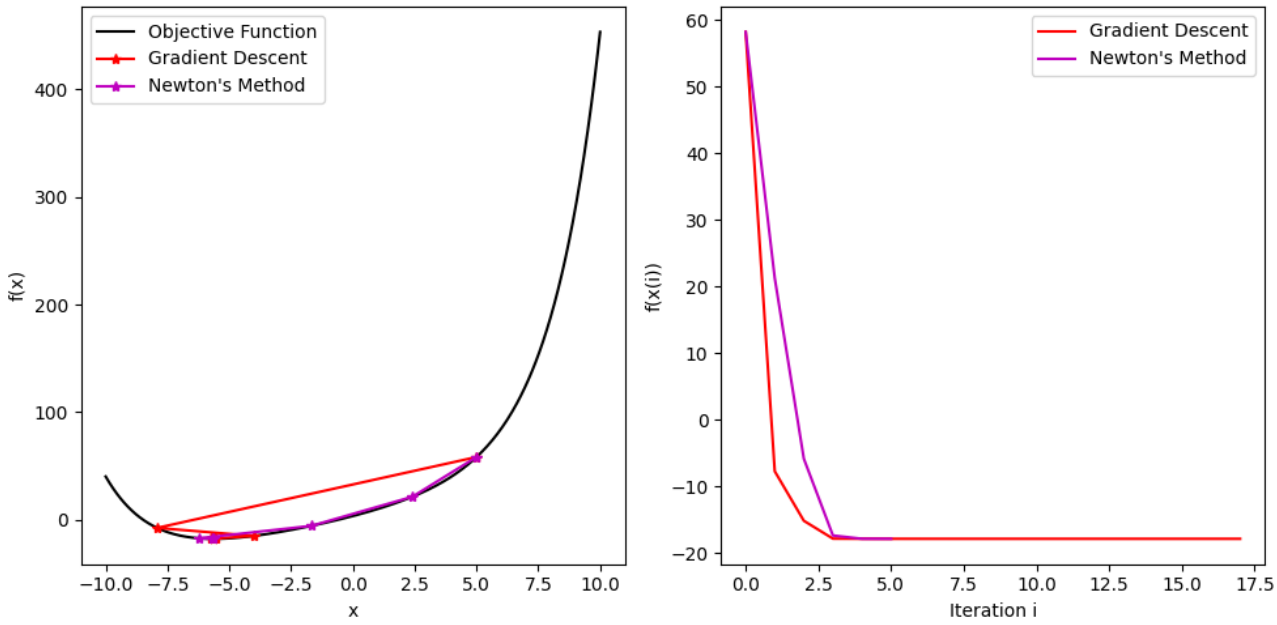

Figure 1: Left: Object function over the interval and the sequence of points generated by Gradient Descent and Newton's Method. Right: $fsum(x^i)$ vs. i for Gradient Descent and Newton's Method

The number of iterations for Gradient Descent is slightly smaller than that for Newton's Method in this specific example. The potential reasons for this could be:

- The initial guess might favor Gradient Descent's steepest descent direction.
- Backtracking could lead to smaller steps for Newton's Method.

In summary, for this function and starting point, Gradient Descent's approach was more effective than Newton's. However, fewer iterations don't always mean faster overall computation.

Implementation 2

a. sgd.py

b. plot_ sgd.py

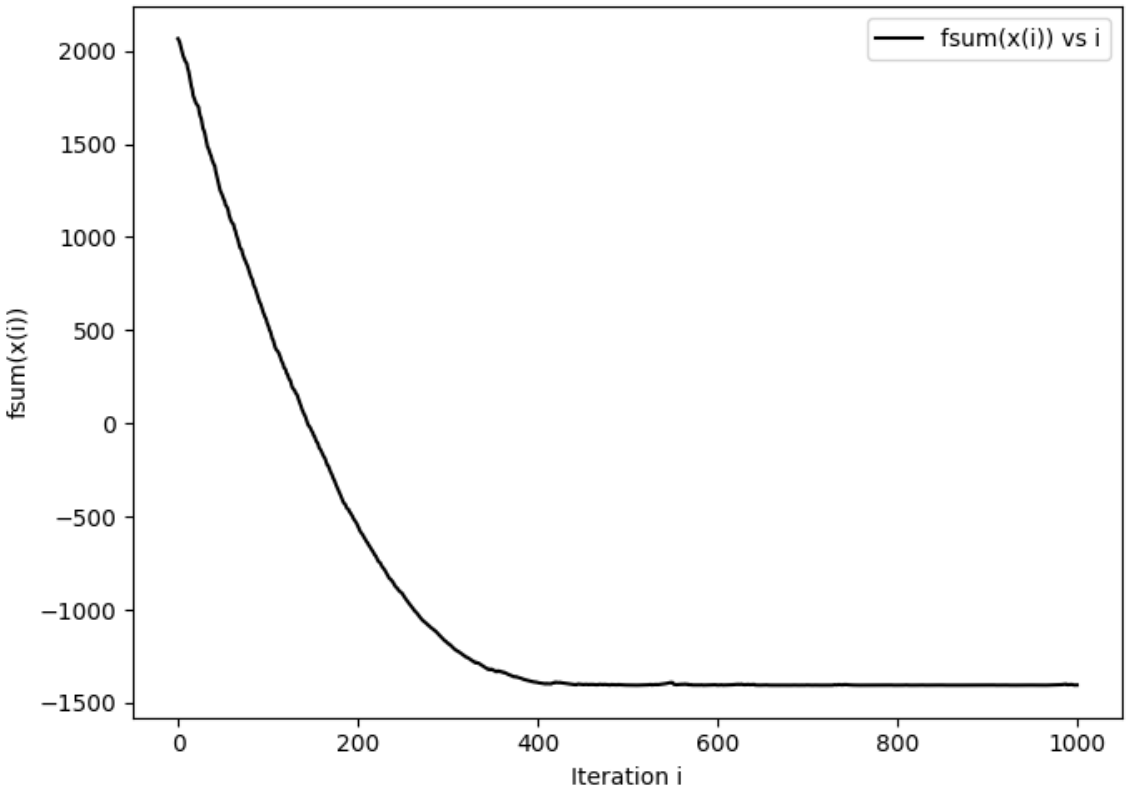

Figure 2: $fsum(x^i)$ vs. i for Stochastic Gradient Descent

$fsum(x^i)$ is not always decreasing. While the expected direction of movement is downhill (toward the minimum), stochastic nature means we're only moving in the direction of the negative gradient of one randomly selected f_i, which might not align perfectly with the gradient of $fsum(x)$. Therefore, the overall image is on a downward trend, but it fluctuates locally.

c.

Table 1: Mean and Variance in Different Iterations

Iterations	**Mean**	**Variance**
1000	-1403.545	4.687
750	-1403.733	4.093

The mean value for 1000 iterations is slightly lower than for 750 iterations, suggesting that more iterations lead to better convergence on average. The variance for 1000 iterations is slightly higher than for 750 iterations. This indicates that while 1000 iterations might get closer to the optimal value on average, the results are also slightly more spread out.

Overall, more iterations seem to give a marginally better average result, but with slightly more variability in the outcomes. Running the algorithm for too many iterations without any significant improvement in the results can be computationally wasteful. In real-world applications, it's essential to find a balance and use techniques like learning rate annealing or early stopping to optimize SGD's performance.

d.

Table 2: Comparison of Methods

Metric	**SGD**	**Gradient Descent**	**Newton's Method**
Runtimes [s]	1.968×10^{-3}	6.554	0.356
$fsum(x^*)$	-1405.343	-1405.267	-1405.266

i.

- **SGD**: It's the fastest. This is because, during each iteration, SGD evaluates the gradient of only one randomly chosen term from the sum, making it computationally lightweight per iteration.
- **Gradient Descent**: It's the slowest. This is expected since, at every iteration, it computes the gradient using all terms of the sum, making it more computationally intensive.
- **Newton's Method**: Faster than Gradient Descent but slower than SGD. It needs to compute both the first and second derivatives but converges quicker due to its second-order nature, balancing out the computation time.

ii.

- Gradient Descent is the best, and SGD is the worst.
- All methods provide values close to each other. Given its stochastic nature, slight variations from the true minimum are expected for SGD.
- Both Gradient Descent and Newton's Method converge to nearly the same value, which makes sense since they both deterministically approach the function's minimum, albeit with different update mechanisms.

Implementation 3

a. For a general optimization problem:

$$\min_x \quad tf_0(x) + \phi(x)$$

Therefore, the objective function with the log barrier in the centering problem is:

$$\min_x \quad tf_0(x) - \sum_{i=1}^{m} \log(-f_i(x))$$

Where: $f_0(x)$ is the original objective function. $f_i(x)$ are the inequality constraints. t is a positive parameter that determines the strength of the barrier. As t increases, the barrier's influence decreases.

b. The objective function with the log barrier for a linear program is:

$$\min_x \quad tc^T x - \sum_{i=1}^{m} \log(-a_i^T x + b_i)$$

Where: c is the cost vector. a is the i-th row of the constraint matrix A. b is the right-hand side of the constraints.

c. The gradient of the objective function with respect to x is:

$$\nabla f_t(x) = tc - \sum_{i=1}^{m} \frac{1}{b_i - a_i^T x} a_i$$

d. The Hessian matrix of the objective function with respect to x is:

$$\nabla^2 f_t(x) = \sum_{i=1}^{m} \frac{a_i a_i^T}{(b_i - a_i^T x)^2}$$

e. The duality gap for the barrier method with log barrier functions is given by:

$$\text{Duality Gap} = \frac{m}{t}$$

Where: m is the number of inequality constraints, which in the context of the question is referred to as 'numplanes'. t is the optimization "force" increment. So, the duality gap in terms of 'numplanes' and t is:

$$\text{Duality Gap} = \frac{\text{numplanes}}{t}$$

f. barrier.py

g. lptest.py

Implementation 4

a. inearize dynamics numerically function in carqp.py

b. optimize single action functionin carqp.py