# EECS 465/ROB 422: Introduction to Algorithmic Robotics
# Fall 2023
# Homework Assignment #2
# Due October 4th at 11:59pm

Rules:

1. **All homework must be done individually, but you are encouraged to post questions on Piazza.**
2. No late homework will be accepted.
3. The goal of this homework is to develop your understanding of optimization methods. You should use python to implement solutions. You may not use any other language, only python will be accepted.
4. Submit your python files in a `zip` along with a `pdf` of your answers to Gradescope. Do not paste your code into your `pdf`.
5. Remember that copying-and-pasting code from other sources is not allowed.

## Questions

1. (5 points) Is a single point convex? Use the definition of convexity to justify your answer.
2. (10 points) Is the function $f(x) = (|2 - 5x| + 2x + 8e^{(-4x)}) - 1$ convex? Use the principles of composition and the common convex functions shown in the lecture to justify your answer.
3. (10 points) Rewrite the following optimization problem in standard form ($x = [x_1, x_2]^T$):

   Standard form or Linear programming

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & -4x_2 + 3x_1 - 3 \\ \text{subject to} \quad & x_2 \leq -3, \\ & -x_1 - 2 \geq x_1 - 5x_2, \\ & x_2 + 6.3 = x_1, \\ & -x_2 + 5 + 4x_1 \leq 4x_1. \end{aligned}$$

4. (15 points) Consider $f(x) = \max\{3x^2 - 2, 2x - 1\}$. At what value(s) of $x$ will the subdifferential $\delta f(x)$ contain more than one subgradient? What is $\delta f(x)$ at each such $x$ value?
5. A linear program is defined as:

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

a. (10 points) Write down the Lagrange dual function for this problem and define the variables.

b. (5 points) Write down the dual problem for this LP.

c. (5 points) Suppose you solved the dual problem and obtained an optimal value $d^*$. Assuming the primal is feasible and bounded, how does $d^*$ relate to the solution of the primal problem $p^*$? Explain why.

## Software

1. Install `Matplotlib`, which is a plotting package for python. Open a terminal and run the following command:

   ```
   sudo apt-get install python-matplotlib
   ```

   Documentation is here.

2. Download `HW2files.zip`. Included in `HW2files.zip` is a file called `plotter.py`. This shows a simple example of how to plot a function using `matplotlib`. Run this script and verify that it produces a plot. You can use the code in this script to help you debug your code in the implementation section.

3. Install cvxpy. Open a terminal and run the following commands:

   ```
   sudo apt-get install python-pip
   pip install cvxpy
   ```

   Documentation is here.

## Implementation

1. Descent Methods: Here you will implement two descent methods and compare them.

   a. (5 points) Implement backtracking line search for functions of the form $f(x) : \mathbb{R} \to \mathbb{R}$. Set $\alpha = 0.1$ and $\beta = 0.6$. Submit your code as `backtracking.py` in your `zip` file.

   b. (10 points) Implement the Gradient Descent algorithm. Use your backtracking line search implementation (with the same $\alpha$ and $\beta$ as above) to compute the step length. Use $\epsilon = 0.0001$. Submit your code as `gradientdescent.py` in your `zip` file.

   c. (15 points) Implement the Newton's Method algorithm. Use your backtracking line search implementation (with the same $\alpha$ and $\beta$ as above) to compute the step length. Use $\epsilon = 0.0001$. Submit your code as `newtonsmethod.py` in your `zip` file.

   d. (20 points) Run Gradient Descent and Newton's method on the following problem, starting at $x^{(0)} = 5$:

   $$\underset{x}{\text{minimize}}\ f(x) = e^{(0.5x+1)} + e^{(-0.5x-0.5)} + 5x$$

   Generate the following plots (this can be done using the `matplotlib` library). Include these plots in your `pdf` and the code to generate them as `plot_descents.py` in your `zip`. Remember to label the axes.

i. A plot showing the objective function over the interval $[-10, 10]$ (black) and the sequence of points generated by Gradient Descent (red) and Newton's Method (magenta).

ii. A plot showing the $f(x^{(i)})$ vs. $i$ for Gradient Descent (red) and Newton's Method (magenta).

Explain which algorithm performed better in this example in terms of number of iterations and why.

2. Stochastic Gradient Descent: Open the `SGDtest.py` file included in `HW2files.zip`. Here you'll see that a function `fsum(x)` has been defined as a sum of functions `fi(x,i)` for $i = 1, ..., \texttt{nFunctions}$. The first and second derivatives of `fsum` and `fi` are also given.

a. (10 points) Implement the Stochastic Gradient Descent (SGD) algorithm as a separate file `sgd.py` (submit this in your `zip` file). You can call your algorithm from `SGDtest.py` to test it. Set the parameters in the following way:

- Set the step size $t = 1$. It's OK to use this large step size for this problem because $\nabla f_i$ is very small.
- Run the algorithm for 1000 iterations (this is the termination condition).
- Choose one random $\nabla f_i$ per iteration.
- Start at $x^{(0)} = -5$.

b. (10 points) Create a plot showing $\texttt{fsum}(x^{(i)})$ vs. $i$ and insert it in the `pdf` and include the code to generate the plot as `plot_sgd.py` in your `zip`. Note that you should **not** evaluate `fsum` within the SGD loop because this will be very slow. Instead, store the $x^{(i)}$s you produce and evaluate `fsum` after you're done running the algorithm. Is $\texttt{fsum}(x^{(i)})$ always decreasing? Explain why or why not.

c. (10 points) Run SGD 30 times and compute the mean and variance of the resulting $\texttt{fsum}(x^*)$. Run SGD with 750 iterations 30 times and compare the resulting mean and variance to what you got with 1000 iterations. Explain the results.

d. Now we will compare SGD with 1000 iterations to Gradient Descent and Newton's Method in terms of computation time. Use $x^{(0)} = -5$ and $\epsilon = 0.0001$ for both Gradient Descent and Newton's Method. To time how long an algorithm takes, see the timing code in `SGDtest.py`.

i. (10 points) Put the runtimes you get for the three algorithms in your `pdf`. Explain your results, i.e. why are you getting the order of times you get here.

ii. (5 points) Compare the three algorithms in terms of $\texttt{fsum}(x^*)$. Which is the best and which is the worst? Explain why and explain if the difference is significant.

3. Implementing the Barrier Method: Before you start writing code, you will need to figure out some of the equations you will need. The solutions to the following should be included in the `pdf` you submit:

a. (5 points) Write down the objective function used in the centering problem of the log barrier method (in the general case). Define the variables used in this function.

b. (5 points) Write down the function for (a) in the case of a linear program. Define the variables used in this function.

c. (5 points) Write down the derivative of the function for (b).

d. (5 points) Write down the second derivative of the function for (b).

e. (5 points) Write down the duality gap for the barrier method with log barrier functions. It should be in terms of `numplanes`, the number of hyperplanes, and `t`, the optimization "force" increment.

The code: Included in `HW2files.zip` is a file called `barrier.py`. This file sets up, draws, and solves a 2-dimensional LP. The barrier method is heavily commented but some parts are missing. You will need to fill in those parts and run the code. You will need to go through the code carefully to understand what is going on in order to fill in the parts correctly. Please make sure not to change anything in the script except what is marked as `###YOUR CODE HERE###`. When you are finished and you run your code, it should produce an output that looks like Figure 1.

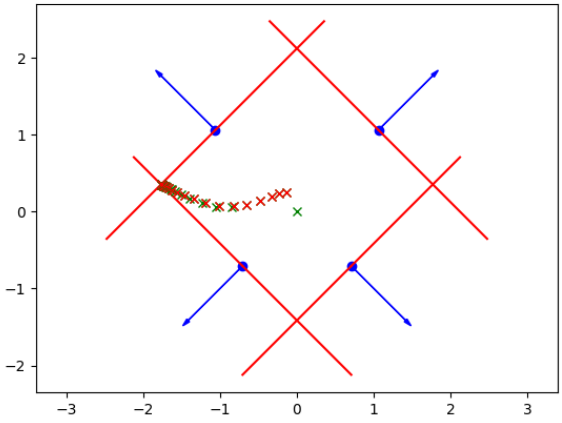


Figure 1: The red lines are the hyperplanes defining the constraints, blue arrows are the normals of these hyperplanes. Green xs are the $x$ values computed during the inner loop iterations (i.e. the Newton iterations). Red xs are the values at every outer-loop iteration.

The script should also print out this message at the end:

The optimal point: (-1.767550, 0.353440)
Total number of outer loop iterations: 28

If you're not getting this solution (with accuracy up to the 2nd decimal place), something is wrong with your implementation.

f. (35 points) Include your `barrier.py` in your `zip` file. We will test your code on a different problem to verify that it is correct (i.e. by changing the A,b, and c matrices). Do not assume the number of hyperplanes is constant. The problem will still be 2-dimensional. If your code does not run or crashes, you will receive 0 points for this part of the problem.

g. (10 points) Now it's time to verify your solution using a state-of-the art solver. Write a script using `cvxpy` to solve the same problem. You only need to give it the proper A,b, and c matrices and it will do the rest. Include your code as `lptest.py` in your `zip`. When we run this code it should only print out the solution, which should be the same as what you found in `barrier.py` (with accuracy up to the 2nd decimal place). See here for an example LP in `cvxpy`.

4. Optimal control using a QP: The script `carqp.py` simulates driving a robot with simplified car-like dynamics. We will formulate the controller for this car as a quadratic program (QP).

   a. (15 points) The car has discrete-time dynamics $x_{t+1} = f(x_t, u_t)$, where $x_t$ is the state at time $t$ and $u_t$ is the control command applied at time $t$. The state is the pose of the car $[x, y, \theta]^T$ and the control is $[speed, turn]^T$. The dynamics function is not convex. So, the first step to controlling the car with a QP is to approximate the true dynamics with a linearization. We will NOT assume that the dynamics function is known in closed form, so we must use numerical differentiation to linearize the dynamics. The linearized dynamics are of the form

   $$x_{t+1} = f(x_r, u_r) + A(x_r, u_r)[x_t - x_r] + B(x_r, u_r)[u_t - u_r]$$

   where $A \in \mathbb{R}^{3x3}$ and $B \in \mathbb{R}^{3x2}$. $A$ and $B$ are not constant matrices, rather they are computed for some reference state $x_r$ and control $u_r$ (they are a local approximation to the dynamics). Since the dynamics takes as input two vectors ($x_t$ and $u_t$), it is convenient to concatenate them to make the linearization code easier to write:

   $$A(x_r, u_r)[x_t - x_r] + B(x_r, u_r)[u_t - u_r] = [A(x_r, u_r)\ \ B(x_r, u_r)] \begin{bmatrix} x_t - x_r \\ u_t - u_r \end{bmatrix}$$

   In fact, $[A(x_r, u_r)\ \ B(x_r, u_r)]$ is the Jacobian of $f$ evaluated at $x_r, u_r$. In `carqp.py`, you will see the function `linearize_dynamics_numerically`, which takes as input $x_r, u_r, h$, and a function pointer to the true dynamics. Implement Newton's difference quotient to compute the Jacobian. Do not compute the derivative analytically, this will receive no credit. Your code should go in the section of `linearize_dynamics_numerically` denoted by `###YOUR CODE HERE###`.

   To test your implementation run `python3 carqp.py test_linearization`, which will print out the $A$ and $B$ matrices and show the prediction error on a test example. A correct implementation will produce an error $< 10^{-10}$ for this example. We will confirm this when we run your code.

   b. (20 points) Now that you have a method to linearize dynamics, we can use a QP to control the robot. The function `optimize_single_action` in `carqp.py` should produce the optimal control command. This function uses the `cvxpy` optimizer to solve the QP. Your goal is to formulate the constraints and objective function for this QP. Define these in the `###YOUR CODE HERE###` block.

   The objective is to find a control command $u^*$ that produces an $x_{t+1}$ which is as close as possible to the goal state, while obeying the constraints defined in the variables `speed_limit` and `turn_limit`. To simplify the problem, we will linearize about the current state ($x_t = x_r$) and a control of $u_r = [0, 0]$. We will assume that $x_t$ is an equilibrium point, i.e. $x_t = f(x_t, [0, 0])$. The dynamics then become:

$$x_{t+1} = x_t + A(x_t, u_r)[x_t - x_t] + B(x_t, u_r)[u_t - u_r] = x_t + B(x_t, u_r)[u_t - u_r]$$

You will need to use these dynamics as part of your objective function.

`cvxpy` is very flexible, so your constraints and objective function do not have to be in standard form. Note that this function uses `linearize_dynamics_numerically`, so make sure to complete part (a) before starting this part.

To test your method, run `python3 carqp.py run_test 0`. This will show an animation where the car drives toward a pre-specified goal state. At each step of the animation, it runs the `optimize_single_action` function and generates a command. That command is then executed using the true dynamics. The animation stops when the magnitude of the command is very small ($< 10^{-10}$) or a maximum number of steps (40) has been executed. If your function is implemented correctly the final frame of the animation will show the car being very close to, but not exactly at, the goal state (see Figure 2 below). The distance to goal should be less than 0.02. We will test your code by running it on a different example.

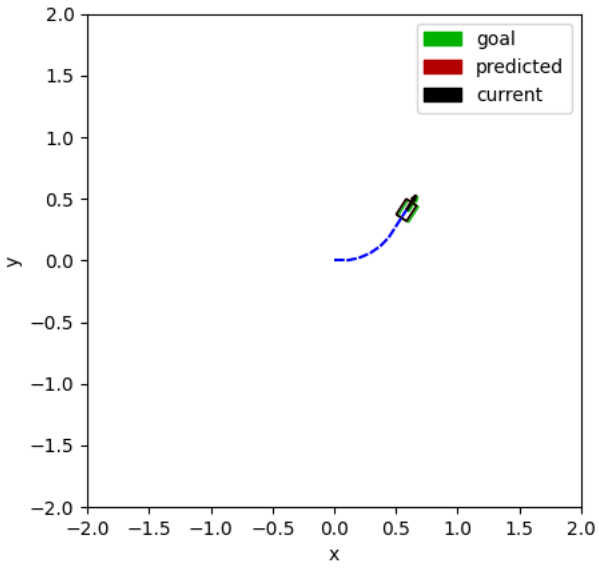


Figure 2: Expected output for test 0.

Make sure to include your `carqp.py` in your `zip` file.

Note: There are two other tests you can run to test out your method: `run_test 1` and `run_test 2`. Test 1 should also bring the car very close to the goal, while there will be a large error for test 2. Remember that the QP approach is a local method that only thinks one step ahead, and so there are many problems it can't solve.