ROB 502 Fall 2022: Assignment 5

Due 11/9 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 (unless you use your late-day tokens)
3. Submit your code on autograder.io
4. Remember that copying-and-pasting code from other sources is not allowed

Code Download:

Download the template code zip here. Unzip the the contents of the zip file to your Dropbox directory. Make sure you have set up VScode before starting the assignment. For instructions on VSCode setup, see this guide.

Open the directory `hw5` in VScode. You can do this by `cd`-ing to the `hw5` directory and running `code .` . Do not run VScode from subdirectories of `hw5` .

Instructions:

Each problem will give you a file with some template code, and you need to fill in the rest. Make sure to only put your code in the areas that start with `// --- Your code here` and end with `// ---`. Do not edit code outside these blocks!

We use the autograder, so if you're ever wondering "will I get full points for this?" just upload your code in the autograder to check. There is no limit to how many times you can upload to autograder. The autograder is set to ignore whitespace and blank lines, so there is some forgiveness on formatting. In most of these questions, the input/output and printing is handled for you in the template code, but in some you are asked to write it yourself. The autograder may test your problem with multiple different inputs to make sure it is correct. We will give you examples of some of these inputs, but the autograder will also try your code with some other inputs. The autograder will only show you if you got it right/wrong, so if you don't get full points, try to test some other inputs and make sure your program works.

Sample input and output are given as `input.txt` and `output.txt` respectively. Getting your program to successfully compile and correctly reproduce the sample output will get you some points, while the remaining points will be from hidden inputs that test some edge cases. Think about valid inputs that could break your logic. Note that the templates given will read from `input.txt` and output to `output.txt`, so it might be a good idea to make a copy of the sample output to avoid overriding it.

Problem 1: Binary Search [10 points]

A very important task is to be able to search for items of interest. By sorting a sequence, we can dramatically speed up search. This is the famous *binary search* algorithm. It is termed binary because at each iteration it divides the sequence into two - one half that might contain the element and the other that definitely does not have the element. Your task is to implement the iterative and recursive versions of binary search in `hw5/binary_search/binary_search.cpp`.

See https://en.wikipedia.org/wiki/Binary_search_algorithm for pseudocode.

The content of `input.txt` will be a sequence of two lines, the first one being a **sorted** sequence of integers (in ascending order), and the second line being the element to lookup. The elements are guaranteed to be unique. For example,

```
-6 -3 0 1 2 5 8 9 12
2
```

Looks for the value 2 in the sorted sequence sequence. For each of these pairs of lines, you should output a line to `output.txt` in the format

```
<index> <numIterativeCalled> <numRecursiveCalled>
```

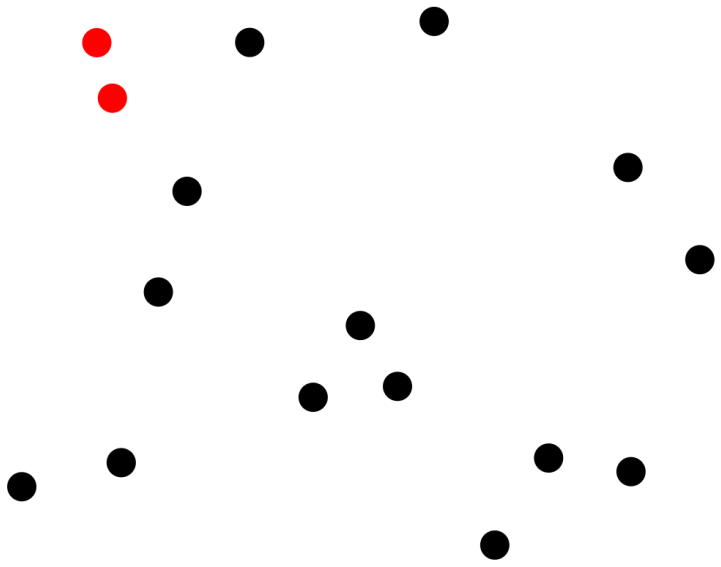
In the above example, the index for 2 would be 4. Note that **binary search only works on sorted sequences**. If the element cannot be found in the sequence, you should output `ERROR` for that line.

Note that the number of calls scales roughly with the logarithm (base 2) of the length of the sequence, so we say the runtime has complexity `O(log n)` .

`numIterativeCalled` should count the number of iterations the main loop in the iterative version is run while `numRecursiveCalled` should count the number of times the recursive version is called.

Problem 2: Closest Pair of Points [14 points]

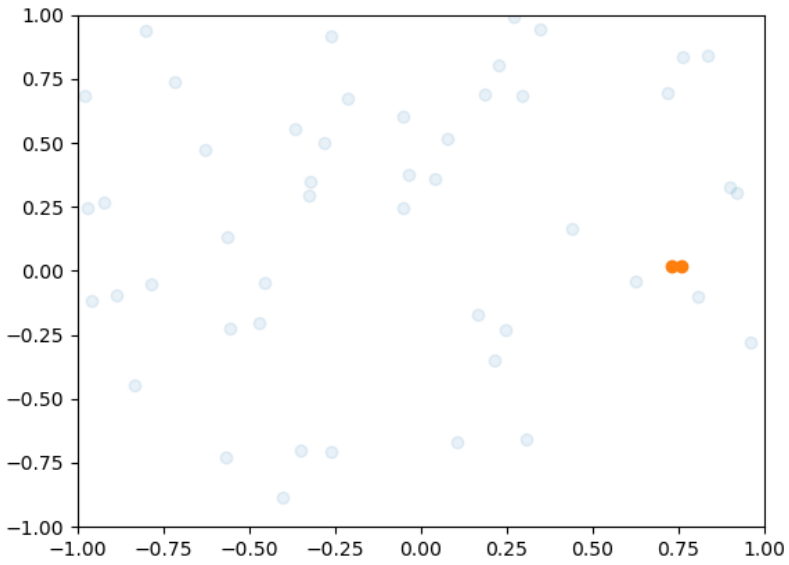
Simulators have to do collision checking of objects. One potential sub-task of that to speed up the process is to first find out what objects could be potentially in collision with each other. A way to do this filtering is to find the pairwise distance between the object centers. This problem is a variant of this process where we instead find the closest pair of points and the distance between them. You will be given `N` points in 2D (problem extends to 3D, but for simplicity we will only do 2D), and you need to find the closest pair of points. See diagram below:



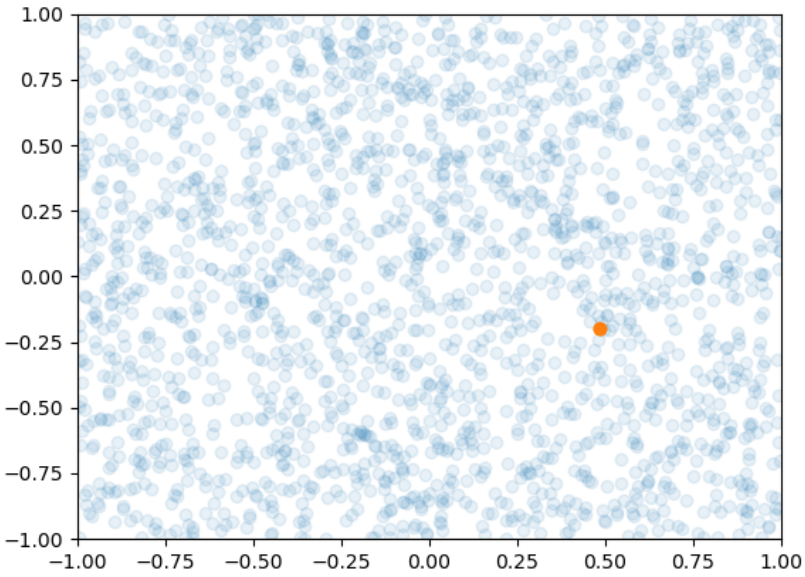
Your task is to implement `closest_pair` in `hw5/closest/closest_pair.h` . The element with the lower ID should be the first element in the returned pair. Each line of `input.txt` is `num seed`, where `num` is the number of points, and `seed` is a seed for a

random number generator. Generating and outputting the points will be taken care of for you in `hw5/closest/closest_pair.cpp` **which you should not modify**.

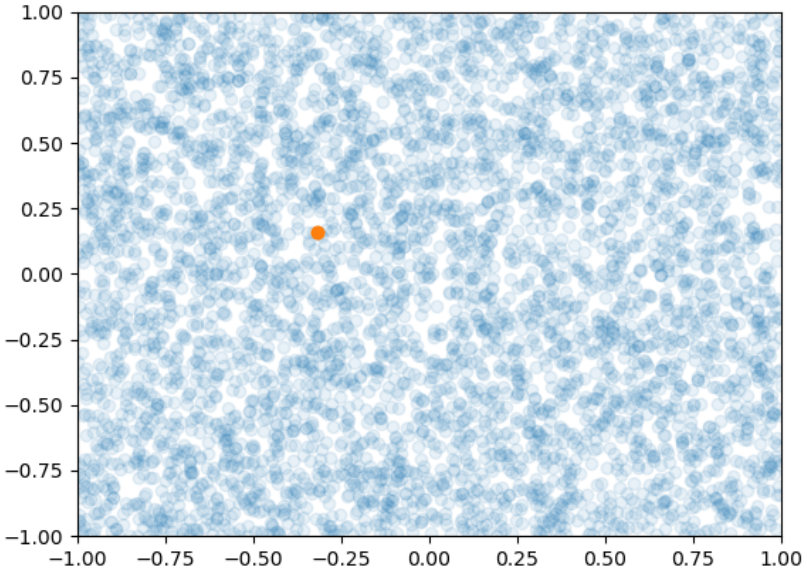
You can visualize the points using the provided `viz_points.py` script, by calling it for example: `python3 viz_points.py points0.txt --output output.txt --index 0` to visualize the points, along with the solution points given by the first line (index 0) in the `output.txt` file. You can omit `--output` and `--index` to just show the points. To generate the points, compile with `export_points = true` in `hw5/closest/closest_pair.cpp` - you don't have to worry about remembering to turn this back off when submitting since you are only submitting the header.



Correct solution for the first set of given points and the first given input.



Correct solution for the second set of given points and the second given input.



Correct solution for the third set of given points and the third given input.

Note that with the naïve $O(N^2)$ implementation of simply computing pairwise distances between all points and taking the minimum will allow you to pass the sample test cases, but this method will fail the hidden test cases due to timing out. You need to implement at least a $O(N(logN)^2)$ time algorithm to pass.

Hint: You should use recursion. Sorting the list of points can be done in $O(NlogN)$ time and this kind of structure may be necessary for speedups. Consider each x,y dimension separately and whether you can break the problem up into simpler, smaller sub-problems. If so, how do the solutions to the sub-problem relate to the bigger problem? Note that the the naïve $O(N^2)$ method be good enough for *small-enough sub-problems*.

Problem 3: Optimization Descent [12 points]

Optimization is useful in robotics in many ways, especially in controllers. This problem concerns setting up an optimization problem and solving it using various methods. This problem uses gradient descent while a later question requires setting it up as a quadratic program (QP).

The problem is to control a modified Dubin's car which has discrete-time dynamics

$$x_{t+1} = f(x_t, u_t)$$

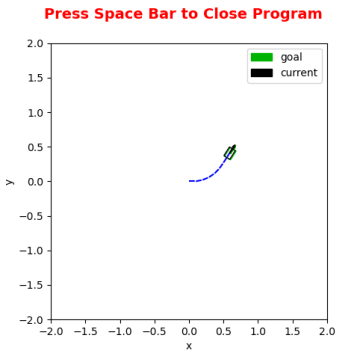
where x_t is the state at time t and u_t is the control command applied at time t. The state is $[x, y, \theta]^T$ and the control is $[speed, turn]^T$. You can evaluate the true dynamics at any x, u, by calling `true_dynamics(x, u)` but the analytical function for f is not given to you. Your task is to implement the `estimate_control_gradient(goal, x, u)` function in `hw5/control_descent/control_descent.h`. You need to formulate a cost function that measures the predicted state's Euclidean distance to the goal state (similar to the least squares problem in Lab 16), and estimate its gradient with respect to the control.

$$c(x, u, x_g) = ||f(x, u) - x_g||_2$$
$$\nabla_u c(x, u, x_g) = \frac{\partial c}{\partial u}$$

Note that this gradient is a function of the goal state, state, and control, hence why `goal, x, u` are passed into the function. We then perform gradient descent using the estimated gradient to find the optimal single-step action in `optimize_single_action` satisfying control bounds. This is repeated 40 times to produce a trajectory, outputted to `output.txt`. Outputting the trajectory is done for you in `hw5/control_descent/control_descent.cpp` , which you should not modify.

`input.txt` has a single line with 6 numbers; the first 3 correspond to the starting state and the later 3 corresponds to the goal state. Reading the input is done for you.

You can visualize the system and the trajectory by calling `python3 trajectory_plotter.py input.txt output.txt`



Problem 4: Optimization QP [14 points]

Continuing with the system from the problem before, we now consider solving it via a different form of optimization that considers constraints, a quadratic program (QP). A QP has the form:

$$\begin{aligned} \text{(QP)} \quad \text{minimize } & \mathbf{x}^T D\mathbf{x} + \mathbf{c}^T\mathbf{x} + c_0 \\ \text{subject to } & A\mathbf{x} \gtrless \mathbf{b}, \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

We will be using the CGAL library to solve QPs. You can read about the syntax for interfacing with this library, and what each parameter means here: https://doc.cgal.org/latest/QP_solver/index.html

You should have it installed in the VM, but in case you don't, you can install it with: `sudo apt-get install libcgal-dev`

The car dynamics are nonlinear, but you can approximate the true dynamics with a linearization. We assume we do not know the dynamics function in closed form, so we resort to using numerical differentiation to linearize the dynamics, into the form

$$x_{t+1} = S(x_r, u_r)x_t + B(x_r, u_r)u_t$$

where $S \in \Re^{3x3}$ and $B \in \Re^{3x2}$ are not constant matrices, but computed for some reference state x_r and control u_r. You can concatenate the matrices to make the linearization code easier to write:

$$S(x_r, u_r)x_t + B(x_r, u_r)u_t = [S(x_r, u_r)\ B(x_r, u_r)][x_t\ u_t]^T$$

$[S(x_r, u_r)\ B(x_r, u_r)]$ is the Jacobian of f evaluated at x_r, u_r. You can estimate it numerically very similar to how you estimated the gradient with respect to the cost function in the previous problem. The Jacobian is just a multi-dimensional output generalization of the gradient.

Your task in this problem is to implement `linearize_dynamics_numerically` and `optimize_single_action`, both of which are in `hw5/control_qp/control_qp.h`.

To implement `optimize_single_action`, you need to set up the QP, where the objective function is the predicted state distance to the goal state (**hint**: optimizing the squared distance is the same as optimizing the distance). You are optimizing over the 2 dimensional control, with given min and max bounds for each dimension in `hw5/control_qp/control_qp.cpp`. **hint**: Specifically, you are minimizing the objective function

$$||Sx + Bu - x_g||_2^2 = ||Bu + (Sx - x_g)||_2^2$$

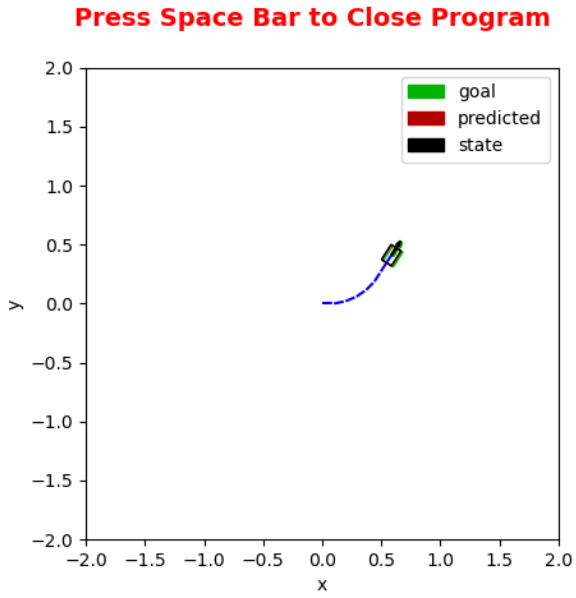
For each time step's x against the goal state x_g to get the control for that time step u. **hint**: consult lab 16 for expanding this expression to see how to extract the D and c^T parts from it.

Consulting the QP form above, for CGAL, you need to specify 2D instead of just D, and you need to specify only the upper triangular parts of D since it is symmetric. Concretely, if you have a matrix

$$D = \begin{bmatrix} D_{00} & D_{01} \\ D_{10} & D_{11} \end{bmatrix}$$

CGAL expects you to tell it $2D_{00}, 2D_{01}, 2D_{11}$

`input.txt` has a single line with 6 numbers; the first 3 corresponds to the starting state and the later 3 corresponds to the goal state. Reading the input is done for you. The program will output to `output.txt` 40 times, each time outputting the control on one line, followed by the linearized `A` and `B` matrices. You can visualize the system and the trajectory by calling `python trajectory_plotter.py input.txt output.txt` . Different from the previous problem, the trajectory using linearized dynamics will be plotted along with the true dynamics. The blue/black trajectory is the actual carried out trajectory using true dynamics while the red trajectory is the predicted trajectory. Note that the predicted trajectory does not get propagated from the previous predicted state, but from the previous actual state.



Problem 5: Optimization LP [8 points]

A linear program (LP) is a more restrictive form of optimization problem than a QP. In fact, a QP with D = 0 is an LP. For this problem, there are robots working at a construction site performing 5 tasks. There are 4 robots, each with prices per hour and capabilities for each task in units per hour. Assume each robot can work on all tasks simultaneously (and that you do not need more than 1 of each robot).

Name	**Price per hour**	`heavy lifting`	`materials transport`	`earth moving`	`concrete pouring`	`brick laying`
SpiderBot P8	$75	1.6	3.5	0.1	2.3	6.1
Gigantimus Maximus	$128	7.2	2.1	7.1	3.2	0.1
VersaDroid X17	$70	3.7	3.2	2.9	3.4	4.9
HedonismBot	$34	0.1	0.15	0.1	0.15	0.1

Each line of `input.txt` consists of 5 numbers, the units of work necessary for `heavy lifting` , `materials transport` , `earth moving` , `concrete pouring` , and `brick laying` , respectively. The program will output to `output.txt` the same number of lines, each line with 5 numbers corresponding to the total cost (rounded to nearest dollar) followed by the hours renting the robots (in top-down order of the table above). Up to 2 decimal places are printed. Specifically, you need to implement the `setup_lp` function in `hw5/lp/lp.h` while `hw5/lp/lp.cpp` should not be changed.

The goal is to minimize the total cost while performing the amount of work necessary for each task (**hint**: that's a ≥ constraint). The hours for each robot is unlimited, but it cannot be lower than 0.

Optional additional references:

- https://doc.cgal.org/latest/QP_solver/classCGAL_1_1Quadratic__program.html#a77408452fb3d9f250a03833c9b456ebb
- https://doc.cgal.org/latest/QP_solver/QP_solver_2first_lp_8cpp-example.html