ROB 502 FA24: Homework 4

See Canvas for due dates.

Rules:
1. All HW must be done individually, but you are encouraged to post questions on Piazza and discuss during OHs (recall: do not email assignment questions to teaching staff!)
2. Late work adheres to the late grading policy (see syllabus, or use 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:
The code is available on the ROB 502 FA24 Git repo in the appropriate HW folder.

We prefer that you use the `'fetch'/'pull'` Git methods for the HWs, but we know that Git can be hard/confusing; if necessary, you can always just **download** the code from the repo link. However, learning to use Git will be to your benefit for multiple reasons:
1. You can get some easy extra credit if you properly pull and version-control (and comment) your work for the HW assignments!
2. ROB 550 will require you to use Git in a much more complex capacity, and without much learning time (large code databases, lots of teamwork with multiple branches, etc.).
3. If you ever need to setup a new VM or use your code on another device (new computer, VM crashes, etc.) you can easily restore your code progress with Git.
4. Many robotics-related industries and research labs rely on Git for robust version-controlling of their work… if mastered, you can put this on your resumé!

Open the directory `hw4` in VSCode. You can do this by `cd`-ing to the `hw4` directory and running `code .` . Do not run VSCode from subdirectories of `hw4` .

Instructions:

See previous homework assignments for notes about using Autograder.io, but you should be an expert now!

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!**

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.**

In this assignment you will get practice using inheritance and polymorphism. Additionally, you will practice using the Eigen C++ library.

1. **Project Euler #4** [2 points]: Code a solution to the following problem: **Largest palindrome product**

   Template file: `euler4.cpp`

2. **Fitting a Plane using RANSAC** [14 points]: Your job is to take in point cloud data from a file and find the plane of the form $ax + by + cz + d = 0$ that best fits the data using RANSAC. The pointclouds are given as txt files (e.g. `pointcloud1.txt`), where the first line is the number of points, and the following lines are the (x,y,z) positions of the points. The point cloud data is noisy and has outlier points, meaning the points are not exactly on the same plane, and some points are very far.

   Recall that RANSAC iterative calls a problem-specific model-fitting algorithm on a random subset of the data. You will implement two different model-fitting algorithms for finding a plane given a set of points. To do this, edit the file `ransac.h` to create two subclasses of `BaseFitter`, one called `AnalyticFitter` and one called `LeastSquaresFitter`, each class should override the `fit` function and include an appropriate constructor. You should also implement RANSAC in the `ransac` function. You are given the file `ransac.cpp` which runs RANSAC with both `AnalyticFitter` and `LeastSquaresFitter`, and saves the outputs to `planes.txt`. You do not need to modify `ransac.cpp`, and the autograder will use our version, not yours. The fit methods should work as follows:

   - for `AnalyticFitter`: select three random points to fit the model of the plane using the equations of a plane derived from three points.
   - for `LeastSquaresFitter`: select 10 random points to fit the model of the plane, using least squares.

   For this problem, an inlier is a point whose distance to the plane is below some threshold (which you should pick). To compute distance from a point $[x, y, z]$ to a plane $ax + by + cz + d = 0$ you can use the following formula (HINT: think about how to compute this distance for all points simultaneously, without a for-loop):

$$\frac{|ax+by+cz+d|}{\sqrt{a^2+b^2+c^2}}$$

   Make sure you tune the number of iterations and inlier threshold parameters to get good fits on the example data. The autograder will check whether the angle between your output `[a, b, c, d]` and the true coefficients is within 10 degrees. The true coefficients for the two given examples are:

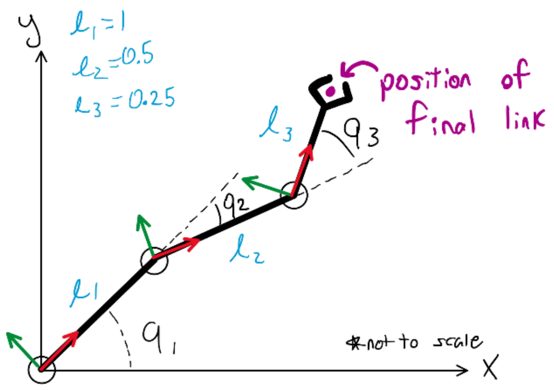
pointcloud1.txt	[1, -1, -0.4, -1.0]
pointcloud2.txt	[-0.4, 1.0, -0.4, 1.0]

You can use the script `viz_plane.py` to visualize the data. If you run it with no arguments, it will visualize the point cloud in `pointcloud.txt` and a plane with (incorrect) default parameters. If you give it four numbers (a,b,c,d) as arguments, it will visualize that plane. See `python viz_plane.py -h` for help info.

Template file: `ransac.h`

3. **Forward Kinematics** [9 points]: Forward kinematics refers to the use of the kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters (Wikipedia, Ch. 3 of "Modern Robotics"). In this problem you will write down the kinematic equations of a 3-link robot arm in C++.

   Your program will read in a text file where each line is the joint positions `θ₁ θ₂ θ₃` (separated by spaces), and may have multiple lines. You should then compute and print to `std::cout` the corresponding position for the (x, y) position of the final link. For printing, use `std::setprecision(3)` and the `operator<<` defined for `Eigen::Vector2d`. The diagram below describes the kinematics of the robot. The counter-clockwise rotations are positive (right-handed system):



   The forward kinematics are computed using a series of matrix multiplications, where each T matrix below is a homogeneous transformation matrix. You should use the "current axis" semantics from the lecture slides (i.e., post-multiply, on the right). Since we're working in 2D, these transformation matrices are 3x3, and have the following format, where `q` is a joint angle in radians (see diagram):

$$T_{3x3} = \left[Rot(q)_{2x2}\ t_{2x1}\ 0_{1x2}\ 1 \right] = \left[\cos cos\,(q) \quad -\ \sin sin\,(q)\ \ t_x\ \ \sin sin\,(q)\ \ \cos cos\,(q)\ \ t_y\ 0\ 0\ 1 \right]$$

   First, implement a function `transform_mat`, which takes in the joint angle `q` and the length of the link `l` and returns the transform `T`. Use the diagram above to figure out how to use `l` to set `tx` and `ty`. The values for `l1,l2,l3` are given in the diagram above. (HINT: draw the arm in the configuration where all the q's are 0!)

   Next, use `transform_mat` inside `main` to compute the xy position of the final link, reading each line in the input file `joint_angles.txt` and printing the 2D position of the end-effector. Use a newline between each output. For example, if the input is `0 1.5707 0`, then the output should be:

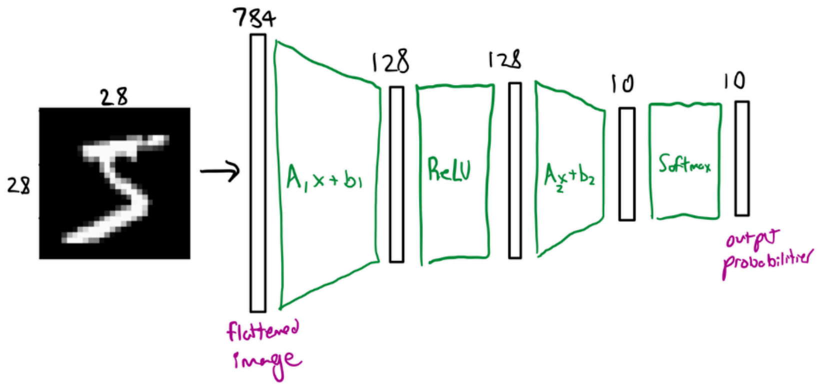
```
   1
0.75
```

You can test your code with different inputs by editing the input file, or by running it in the terminal and supplying the name of another input file as an argument, for example: `./fk joint_angles1.txt`

Template file: `fk.cpp`

4. **Neural Networks** [8 points]: Many common neural networks can be represented as a series of simple matrix operations, such as matrix multiplication or element-wise maximum. Each of these individual operations is called a *layer*. In this problem you will implement a simple neural network for image classification.

   The input data is an image of handwritten digits (example below), flattened to a vector of size 784, and the output is 10 numbers which are the class probabilities. For example, if `probabilities[3]=0.2`, that would mean 20% confidence the input is the number `3`.



   The architecture consists of four layers (green shapes in the diagram):

   - Linear layer, which computes `y=Ax+b`
   - ReLU, which computes the element-wise max with 0: `y=max(x,0)`
   - Another Linear layer
   - Softmax, which converts a vector **x** to a vector of class probabilities **y**, using the formula:

$$y_i = \frac{e^{x_i}}{\sum_i e^{x_i}}$$ (This ensures the class probabilities sum to 1)

   The sizes are shown in the table below, and all matrices are already loaded for you in the template. You job is to finish implementing the class `Linear` and create and implement the classes for the `ReLU` and `Softmax` layers. Then in `main`, you then need to construct the layers and call them in the right order to produce the output `y` given the input `x`. The code should write the class probabilities to `output.csv` (already in the template for you). You can then run `python network/viz_outputs.py` to visualize the 4 test images and their associated class labels. The predicted and actual class labels should match, and the confidences should be >0.99.

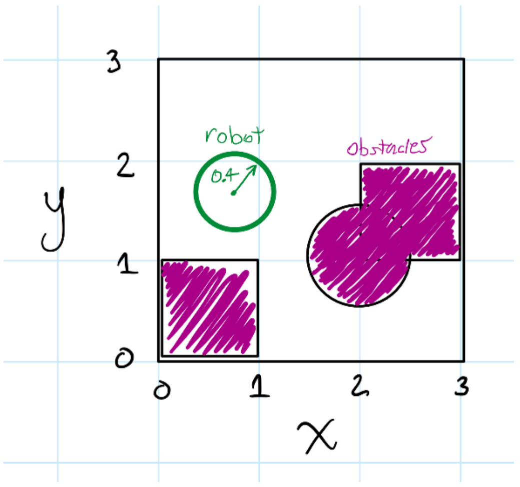
x	(784, 1)
A1	(128, 784)
A2	(10, 128)
b1	(128, 1)
b2	(10, 1)
y (your output!)	**(10, 1)**

Template file: `network.cpp`

*****EXTRA CREDIT*****

5. **Collision Checking** [3.5 points]: *This problem will be scored as **extra credit**, try it out if time allows! It may take a little longer to implement correctly though, so be sure to make sure you have the rest of the assignment ready before spending too much time here. There are 7 test cases scored, but you will only receive +0.5 pts per correct output in Autograder, for a max of 3.5 extra credit points.*

   You will write a collision checker for a disc-shaped robot in a world made up of axis-aligned rectangles and disc-shaped obstacles (see image below for an example). Your job is to implement the `check_collision` and `contains` methods which override the methods in the base class, the `check_intersection_with_edge` function, and the `check_collisions` function. The provided `main` function in `collision.cpp` will then construct a world using these shapes and check whether a disc-shaped robot is in collision at various positions and with various radii. The position and radius are read from the input text file (e.g. `robot_positions.txt`), where each line is the `x y radius` separated by spaces.



While it is possible to define a more general collision check between arbitrary shapes, it is much faster to define more specific collision checks for certain simple shapes. This is a common pattern used in physics engines and collision checkers. The shape-specific `check_collision` routines should work as follows:

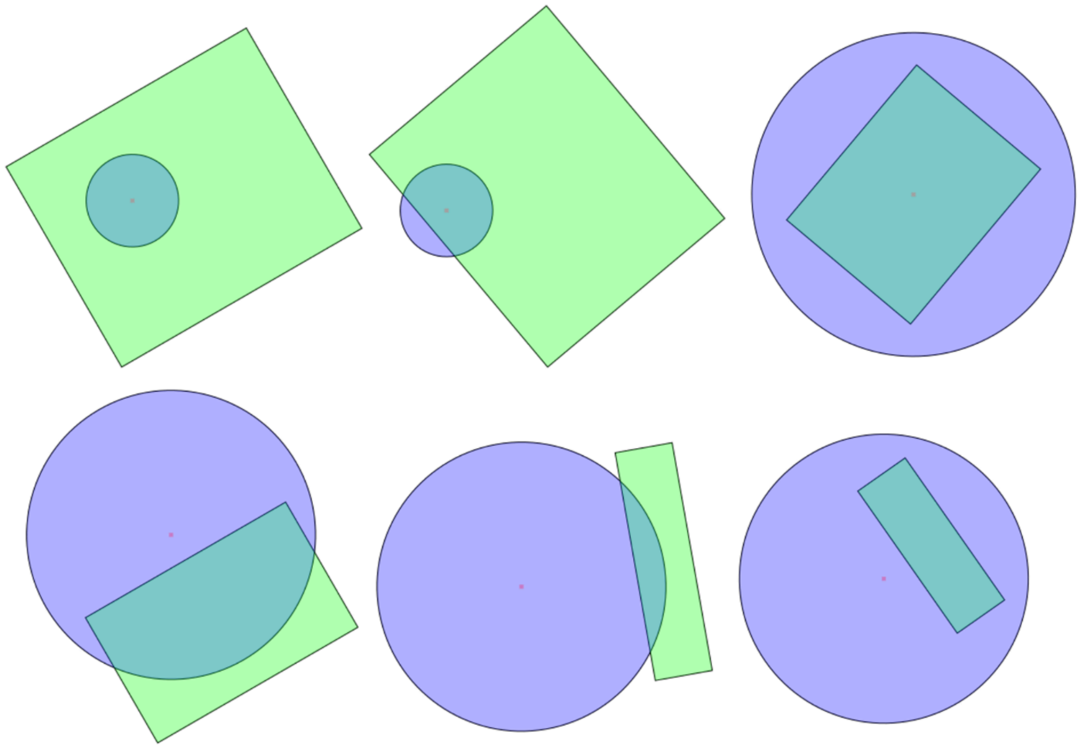
For `Disc` versus `Disc`:

- Compute the distance between the centers of the discs. If that distance is less than or equal to the sum of their radii, then return collision = true

For `Rectangle` versus `Disc`:

- Check whether the center of the disc is contained inside the rectangle. If so, return collision = true
- Otherwise, check whether any of the corner points of the rectangle are contained inside the disc, if so return collision = true
- Finally, check whether any of the rectangles edges intersect the disc by checking each edge for intersection with the disc. To check whether an edge (line segment) intersects a disc (circle), we attempt to find a point on the segment which also lies on the circle. If we can find such a point, then the circle and segment intersect. Please refer to the Lab 13 document for information on how to do this.

See the diagram below for some examples of how circles and rectangles may or may not intersect:



Credit: https://stackoverflow.com/a/402019

You can test your code with different inputs by running it in the terminal and supply the name of the input file as an argument, for example: `./collision robot_positions2.txt`. You can add more test cases by adding more lines to the text file, specifically positions where the robot is just barely touching or not touching the obstacles. This will help you pass the hidden test cases in the autograder. You can use the `viz_world.py` script to help you visualize the different test cases.

Input filename	**Expected outputs**
robot_positions.txt	1
robot_positions2.txt	1
robot_positions3.txt	0

Template File: `collision.h`