ROB 502 FA24: Homework 3

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 `hw3` in VSCode. You can do this by `cd`-ing to the `hw3` directory and running `code .` . Do not run VSCode from subdirectories of `hw3` .

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.**

Problems:

1. **Simple Calculator** [9 points]: In this problem you will create a simple calculator that does arithmetic operations on sequences of numbers. In `hw3/calc` you will see a file called `input.txt` which contains a series of numbers followed by a single math operator on each line. Your code should read in each line and perform the specified operation on all the numbers. The result of each line should be output to a file `output.txt` (the order of results in `output.txt` should correspond to the order of lines in input.txt).

   The numbers in `input.txt` can be any real number and the possible operators are `+` ,`*` , and `/`. For example, the line `8.2 -7.1 2 +` means that you should sum all the numbers: `8.2-7.1+2`. The output should then be `3.1`. For `/`, the operation is only valid if exactly two numbers are provided.

   If it is impossible to perform the specified operation on the sequence of numbers, your code should write `ERROR` for the corresponding line in `output.txt`, but it should not crash, and it should continue on to the following line in `input.txt`.

2. **Sorting** [9 points]: In this problem you will sort a list of objects by their mass using the `std::sort` algorithm. In `hw3/sort`, you will see a file called `objects.txt`, which contains a list of objects with their corresponding masses. `objects.txt` can contain any number of objects. Object names will never contain spaces or special characters. Your program will read in this list and output a file called `output.txt` which contains the list of objects in the same format (including the masses), except that the list is sorted by the mass of the object (lowest to highest).

   You will need to edit both`sortmass.cpp` and `sortmass.h` to complete the program.

3. **Operators** [10 points]: Isn't it annoying that arithmetic operations and printing methods aren't defined for `std::vector`? In this problem, you will define operators for certain types of vectors, so you can easily manipulate and print them in the future. Specifically, you will define the operators `<<`, `+`, `-`, `*`, `/` and `,` for the types `std::vector<double>`, `std::vector<float>` , and `std::vector<int>`and scalars (`int` , `float` , or `double`). To do this, you will need to overload these operators in a file you create, called `operators.h`, in `hw3/operators`. `operators.cpp` in this directory will call these operators on several combinations of inputs to test your code. The output of every operator (except `<<`) should be of type `std::vector<double>`, regardless of the types of the input.

   A description of each operator's expected behavior is below:

   - `<<`: This should print the input vector to an output stream. E.g.

     ```
     std::vector<double> vec{1.1,2,3};
     cout << vec;
     ```

     should print `[1.1, 2, 3]` . Don't forget `[` and `]`and the commas!

- `+`: This should sum the vectors element-wise.
- `-`: This should subtract the elements of the second vector from the first vector.
- `*`: This can be used in two ways:
    1. Elementwise multiplication of two vectors
    2. Multiplication by a scalar. E.g. `2.1*vec` will multiply every element in `vec` by `2.1`. `vec*2.1` should produce the same result.
- `/`: This should divide a vector element-wise by a scalar. E.g. `vec/2.1` will divided every element in `vec` by `2.1`. Don't worry about dividing a scalar by a vector, that's not allowed. Don't worry about dividing by zero, it won't be a test case.
- `,`: This concatenates two vectors in the given order. E.g.

  ```
  std::vector<double> vec1{1,2,3}
  std::vector<float> vec2{4,5}
  std::vector<double> vec_concat = (vec1,vec2); //vec_concat is [1,
  2, 3, 4, 5]
  ```

  Note that you will need to use parentheses as above. Otherwise, the use of `,` is ambiguous.

We should be able to use inputs of different types; e.g. the following code should all work:

```
std::vector<double> v_double{0.2, 0.5, 1};

std::vector<int> v_int{4, 2, 6};

std::vector<double> result1 = v_int + v_double;

std::vector<double> result2 = v_double + v_int; //should produce same
output as above
```

If an operation cannot be performed because the vectors have different sizes, the operator function should throw an error, which should be done with this line:

```
throw ERROR_MESSAGE;
```

ERROR_MESSAGE has already been defined at the top of `operators.h`.

For this problem you will only be uploading `operators.h` to autograder. You can use `operators.cpp` to test out your program but it won't be uploaded (autograder will test your program with its own `operators.cpp`).

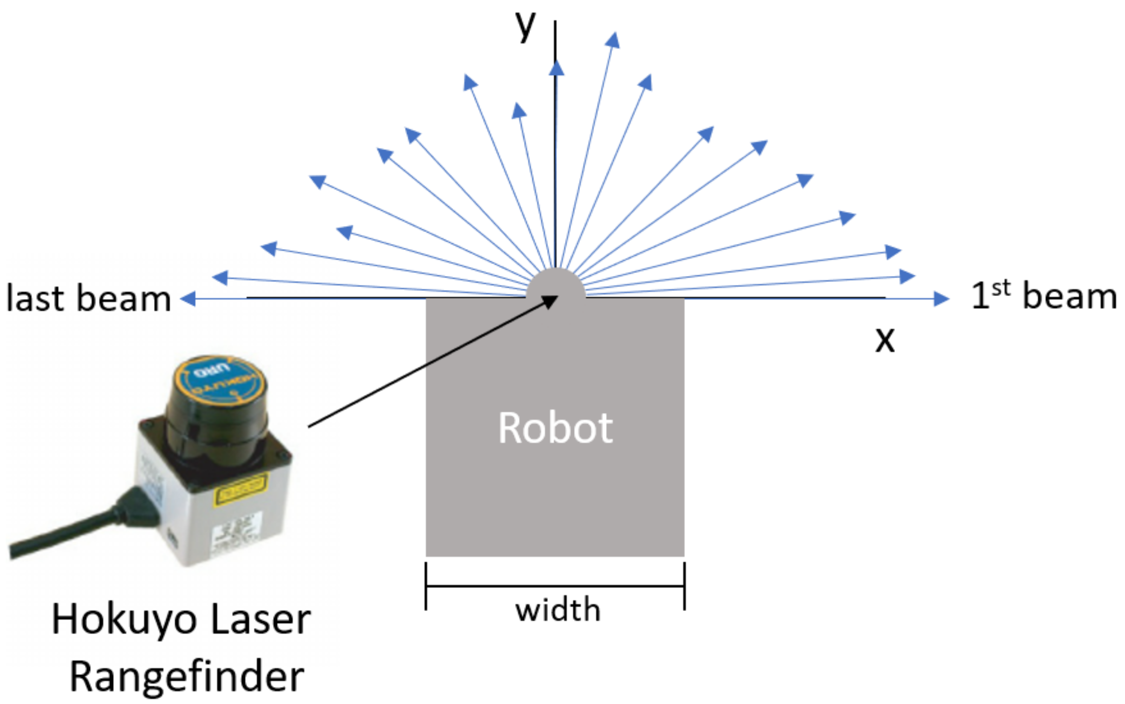
- **Hint 1**: You should use templates so you don't have to define operators for every combination of input types.
- **Hint 2**: Implement the `<<` operator first to help you print things out to debug the other operators.

4. **Braking System** [8 points]: Many mobile robots have a planar laser rangefinder (e.g. made by Hokuyo) for mapping and localization purposes. These sensors can also be used to stop the robot if a collision is imminent. In this problem you will write a program that takes in some robot parameters and laser beam returns and determines whether the robot should stop.

   In `hw3/laserbrake`, you will see the file `input.txt`, which contains three lines:

   ```
   width: 0.5 threshold: 0.1 0.6 1.0 0.3 1.2 3 2 0.1 0.15 0.5 0.4
   ```

   The first line specifies the width of the robot (in meters). The second is the distance threshold (in meters) for stopping the robot. The third line contains at least two positive numbers. These numbers are the laser returns for the scanner. Assume the robot coordinate system is defined as below:



   Here the beams (in blue) are emitted from (0,0), which is at the front of the robot. The number of beams is determined by how many laser returns are specified in the file (there will always be at least two). The first beam is always emitted at 0 degrees and the last is always emitted at 180 degrees. All other beams are emitted at fixed intervals between the first and last beams. For example, if there are 5 beams they will be emitted at 0, 45, 90, 135, and 180 degrees. The laser return value for each beam is the distance along the beam between (0,0) and an object in the world.

   Write the x and y coordinate of every point detected by the laser (i.e. the tips of the vectors in the diagram above for all the blue vectors) to `output.txt` (one x y pair per line, see `output.txt` for examples). If any point is 1) In front of the robot (within the width of the robot on the x axis); **and** 2) is less than or equal to the given distance threshold, then your robot should stop. The program should write `Stop! n` on the last line of `output.txt`, where you replace n with the index of the first point that meets the criteria for stopping. Otherwise, it should write `OK` on the last line of `output.txt`.

5. **Policy Rollout** [11 points]: A control policy $u = \pi(x)$ (often just called "a policy") for a robot is function which tells the robot what action u to execute a given state x. Usually in robotics, these policies are continuous, but in this problem we will work with a discrete policy and discrete actions. In hw3/policy the file policy.txt contains a list of states and corresponding actions in following format: the first two integers specify the x and y coordinates of the state and the next two integers specify the action to be taken. For example, the line `1 1 0 1` means for the state $x = (1,1)$, take action $u = (0,1)$.

   To take an action, we need to know the dynamics of the system, which is a function $x_{t+1} = f(x_t, u_t)$. In this problem the dynamics are very simple: $f(x_t, u_t) = x_t + u_t$.

   When creating a policy, it is very useful to test it by doing rollouts. A *rollout*, is simply the execution of a policy for some number of steps. The file `rollout.txt` consists of three integers, where the first two are the start state and the last is the number of steps. Your code should read in the policy and rollout files and output a file called `output.txt` which contains the sequence of states visited in the rollout (including the start state at the beginning). You should use an `std::map` to store the policy.

   If the rollout takes you to a state that is not in the policy, you should print out the following to `std::cerr`:

   ```
   State x y is not in policy, terminating.
   ```

   where you replace *x* and *y* with the x and y coordinates of the state you went to that was not in the policy. After printing this, your main function should return `1`.

6. **Robot Employment Office** [10 points]: In this assignment you'll practice using `std::queue`, `std::map`, and `std::sort`. In the `hw3/queue` directory, you will find `bots.txt`, which contains a list of robot names and their corresponding types (e.g. the robot named `CameraBot` has the type `Inspector`). Note that the "types" of robots here are not C++ types they are just strings. The file `jobs.txt` contains a list of job identification numbers (integers) and job names (strings with no spaces). For example, job id number `1` is an `inspect` job. There is also a `robots.h` file which defines a class and a data structure you will need to use in your code. **Read the code in `robots.h` carefully before starting the assignment.**

   Imagine that the robots are standing in a line at the office of a construction company (in the order of `bots.txt`). Jobs arrive at the office in the order shown in the `jobs.txt` file.

- For each job, your code should check if the first robot in line can do the job (each type of robot has one or more jobs that it can do).
- If the robot can do the job, you should assign it to the robot.
- If the robot can’t do the job, you should not assign the job to the robot.
- The robot then goes to the back of the line regardless of whether or not it was assigned the job.
- If the job was not assigned, go on to the next robot and repeat the same process.
- Once the job is assigned, move on to the next job in `jobs.txt`, repeating the process above. You can assume that there is at least one robot that can do any job in `jobs.txt`.

When you have assigned all the jobs, print out the robots along with their job assignments **in alphabetical order** to the file `output.txt` .