Advanced Robot Programming Labs
C++ Programming

Labs 3-4: Using classes

1 Content of this lab

The goal of this lab is to read, use and build C++ classes in order to develop an elementary simulator for a ground robot.

1.1 The simulator

The simulator is defined by several headers and source files, as defined in Fig. 1.

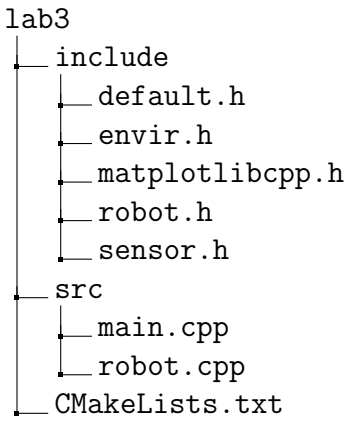

Figure 1: Files used by the simulator

The main file is `main.cpp` and is compiled to an eecutable.
The files `robot.h` and `robot.cpp` define a `Robot` class and will be modified in the first lab to implement new methods that allow using the simulator.

The file `envir.h` defines the `Environment` structure that describes the environment in which the robot is moving. It uses another structure, a point with attributes `x` and `y`. An environment is then defined by a `std::vector` of points, defining a polygon which is considered as walls.

The `defaults.h` file defines a default environment consisting of a $20 \times 20m$ square, as shown in Fig. 2.

1.2 Expected work

During the lab the files will be modified and others will be created. At the end of the lab, please send by email a zip file allowing to compile and test the program.

You may answer the questions by inserting comments in the code at the corresponding lines.

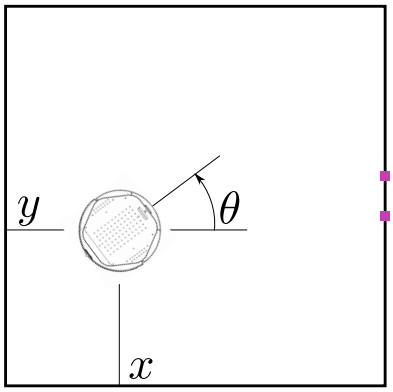

Figure 2: Robot in its environment

2 Building the `Robot` class

2.1 Incomplete methods

Q1 Compile and execute the program. According to the `main.cpp` file, the robot is following some velocity setpoints defined in the (x, y, θ) space. In which files are the following elements defined:

1. `DefaultEnvironment`
2. `DefaultCartesianSetpoint`
3. `Environment`

Q2 Explain the signature of `DefaultCartesianSetpoint`, especially the way to pass arguments. From the `main()` function, can `i` be modified in `DefaultCartesianSetpoint`? What about `vx`?

Q3 In practice it is often impossible to control a ground robot by sending (x, y, θ) velocities. A classical way to control such a robot is to send a setpoint with a linear velocity v and an angular velocity ω, expressed in the robot frame as shown in Fig. 3.

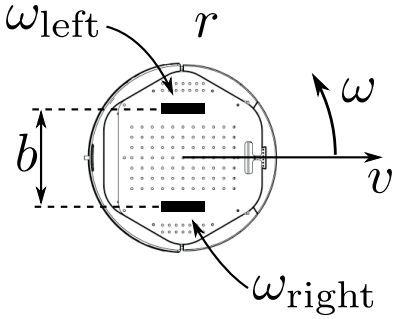

Figure 3: Differential drive model

The corresponding model is quite simple:

$$\begin{cases} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \omega \end{cases} \tag{1}$$

Implement such a function in `Robot::MoveVW`. This method should compute the (x, y, θ) velocities from (v, ω) and then call the `Robot::MoveXYT` method.

Q4 Now that a realistic way to control the robot is possible, should the `Robot::MoveXYT` method stay available for external use? What can we do in the `robot.h` file to make it impossible to use it from outside the `Robot` class?

Q5 When a robot is equipped with two actuated wheels, a simple model is the differential drive model, as shown in Fig. 3. Assuming the two wheels have a radius r and are separated with a distance b, then the kinematic model yields:

$$\begin{cases} v &= r\dfrac{\omega_l + \omega_r}{2} \\ \omega &= r\dfrac{\omega_l - \omega_r}{2b} \end{cases} \tag{2}$$

As in question 3, implement such a function in the `Robot::RotateWheels` method, so that it can be possible to control the robot by sending wheel velocities. You may want to define new attributes in the `Robot` class in order to initialize the radius and base distance through the `Robot::InitWheel` method. We use the following values:

$$\begin{cases} r &= 0.05m \\ b &= 0.3m \end{cases} \tag{3}$$

The method should rely on `Robot::MoveXYT` after having computed the (x, y, θ) velocities from (ω_l, ω_r).

Q6 By using the `wheels_init_` attribute, make sure that it is impossible to do anything in `Robot::RotateWheels` if the radius and base have not been initialized.

2.2 Velocity limits

With the current simulation, we can control the robot:

- by sending linear and angular velocity setpoint with `Robot::MoveVW`
- or by sending wheel velocities with `Robot::RotateWheels`

These two methods call `Robot::MoveXYT`[1] and the robot can reach any velocity. In practice, the wheels have a limited velocity at ±80 round per minute (rpm).

[1] which should not be calable anymore from outside the `Robot` class

Q1 Modify the `Robot::InitWheels` method in order to pass a new argument that defines the wheel angular velocity limit. You may need to define a new attribute of the `Robot` class to store this limit. Update the `main.cpp` file with the value in radian per second corresponding to 80 rpm.

Q2 Modify the `Robot::RotateWheels` method in order to ensure that the applied velocities (ω_l, ω_r) are within the bounds. The method should also print a message if the velocity setpoint is too high. Note that if you just saturate the velocities, the robot motion will be different. A scaling is a better strategy, in this case we keep the same ratio between ω_l and ω_r.

Q3 Although the robot actually moves by having its wheels rotate, it would be easier to be able to send linear and angular velocity setpoints. Modify the `Robot::MoveVW` method so that a (v, ω) setpoint is changed to a (ω_l, ω_r) setpoint that will then be called through `Robot::RotateWheels`. Apply the default (v, ω) trajectory in the simulator. The inverse of model (2) yields:

$$\begin{cases} \omega_l &= \dfrac{v + b\omega}{r} \\ \omega_r &= \dfrac{v - b\omega}{r} \end{cases} \tag{4}$$

3 Sensors

The `sensor.h` files defines a `Sensor` class that has four methods:

- `void Init`: initializes the relative pose between the sensor and the robot
- `virtual void Update`: updates the measurement from the robot current position
- `void Print`: prints the current measurement
- `void Plot`: plots the measurement history

The `Update` method is defined as a pure virtual function, which makes the `Sensor` class a abstract class. It is thus impossible to declare a variable to be of `Sensor` type, as this class is only designed to build daughter-classes depending on the sensor type.

The `Robot` class already has a attribute called `sensors_` which is a vector of `Sensor*`. As the `Sensor` class is abtract it is forbidden to use it by itself, but pointers are still possible.

3.1 Range sensors

Q1 Create a `sensor_range.h` file that defines a `SensorRange` class that is derived from `Sensor`. The `Update` method has to be defined so that the code compiles. For now, just make the method print something to the screen.

Q2 Include this file in `main.cpp` and declare a `SensorRange` variable. We will use a front range sensor placed at $(0.1, 0, 0)$ in the robot frame. Call the `Update` method at the beginning of the `for` loop. Run the program and ensure that the sensor is updated.

Q3 In this question we will build the `Update` function. This sensor should return the distance to the first wall in its x-axis. The sensor can thus be simulated in two steps:

1. Compute the absolute position and orientation of the sensor. As the robot is passed to the `Update` method, we can use its own (x_r, y_r, θ_r) position and the relative position (x_s, y_s, θ_s) of the sensor to get the absolute sensor position:

$$\begin{cases} x &= x_r + x_s \cos\theta_r - y_s \sin\theta_r \\ y &= y_r + x_s \sin\theta_r + y_s \cos\theta_r \\ \theta &= \theta_r + \theta_s \end{cases} \tag{5}$$

2. Compute the distance to the nearest wall. In the environment variable, the walls are defined by a list of points available in `envir.walls`. Fig. 4 shows a configuration where the sensor is at (x, y, θ) and is facing a wall defined by (x_1, y_1) and (x_2, y_2). In this case, the distance to the wall is:

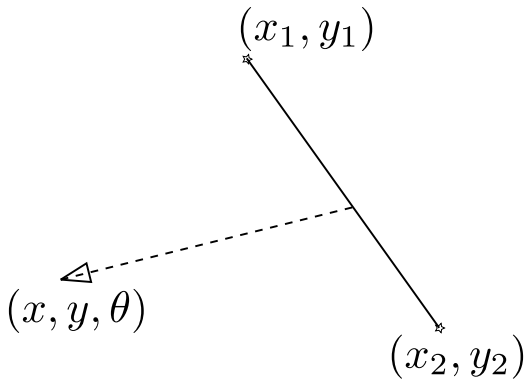

Figure 4: Distance to a segment defined by two points.

$$d = \frac{x_1 y_2 - x_1 y - x_2 y_1 + x_2 y + x y_1 - x y_2}{x_1 \sin\theta - x_2 \sin\theta - y_1 \cos\theta + y_2 \cos\theta} \tag{6}$$

The computed distance is positive if the wall is in front of the sensor, and negative if it is behind (in this case this wall is actually not measured). Also, the denominator may be null if the wall is parallel to the sensor orientation.

Define the `Update` function so that it updates the attribute `s` of the sensor with the distance to the nearest wall.

Q4 At the end of the `Update` method, add the command to append the measurement history: `s_history_.push_back(s_);`
At the end of the program call the `Plot` method of the range sensor in order to display the measurements.