Advanced Robot Programming Labs
C++ Programming

Lab 2: 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.

```
lab2
├── CMakeLists.txt
├── include
│   ├── envir.h
│   ├── geom.h
│   ├── robot.h
│   └── sensor.h
└── src
    ├── envir.cpp
    ├── main.cpp
    └── robot.cpp
```

Figure 1: Files used by the simulator

The main file is `main.cpp` and is compiled to an executable.
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 files `envir.h` and `envir.cpp` defines the environment in which the robot is moving. These files are not to be modified.

The file `sensor.h` defines a virtual class `Sensor` that will be used to create two sensor types: range and bearing sensors.

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.

1.3 Geometry structures

In the `geom.h` file are defined two simple geometry structures:

- One for a 2D pose: `Pose` with attributes `x, y, theta`
- One for a 2D motion: `Pose` with attributes `vx, vy, w`

Those structures have classical constructors, and also methods to express change of frames.

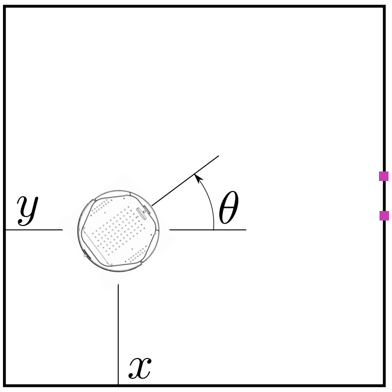


Figure 2: Robot in its environment

2 Building the `Robot` class

The default behavior is the robot following a moving target in the environment. The environment consists of 4 walls defining a 20×20 m square. The target is leaving the square at some point, and for now the robot is also leaving the square as nothing tells it that there are some walls there.

2.1 Defined methods

Some methods of the class are already defined:

- `Robot(std::string _name, double _x, double _y, double _theta)`
  constructor, initialize the robot with a given name at a given (x, y, θ) position
- `Pose pose()`
  returns the current pose of the robot
- `void moveXYT(double _vx, double _vy, double _omega)`
  sends a (v_x, v_y, ω) velocity to the robot and updates its position
- `void goTo(const Pose &_p)`
  tries to have the robot reach the given Pose
- `void :moveWithSensor(Twist _twist)`
  tries to follow a given velocity while ensuring the sensor constraints

2.2 Incomplete methods

Q1 Compile and execute the program. According to the `main.cpp` file, the robot is trying to go to the position of the target. In which files is the target motion defined?

Q2 Explain the signature of `Robot::Robot`, especially the way to pass arguments. From the `main()` function, can the passed arguments be modified while defining a new robot?

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.

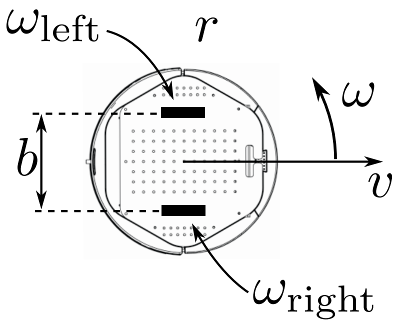


Figure 3: Differential drive model

The corresponding model is quite simple:

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

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 The `moveWithSensor` method was using the XYT motion, which is actually not possible. Modify it so that it calls the `moveVW` method. A simple way to change a desired $(\dot{x}, \dot{y}, \dot{\theta})$ motion to a (v, ω) motion is:

$$\begin{cases} v & = \dot{x} \\ \omega & = \alpha\dot{y} + \dot{\theta} \end{cases}$$

We will use $\alpha = 20$ here.

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:

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

We will need to define new attributes in the `Robot` class in order to initialize the radius and base distance. Create also a new method in the `Robot` class, called `initWheel`, that does so. In the `main.cpp`, use the following values:

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

Centrale Nantes

Q6 Now that the robot has some wheel radius and inter-distance, implement the `Robot::rotateWheels` method, so that it can be possible to control the robot by sending wheel velocities. The method should call `Robot::moveXYT` after having computed the (x, y, θ) velocities from (ω_l, ω_r).

Q7 By using a `bool wheels_init_` attribute, make sure that it is impossible to do anything in `Robot::rotateWheels` if the radius and base have not been initialized.

2.3 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::motateWheels`

These two methods call `Robot::moveXYT`[1] and the robot can reach any velocity. In practice, the wheels have a limited velocity at ±10 rad/s.

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.

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 according to the following algorithm:

Data: desired wheel velocities ω_l, ω_r, velocity limit $\omega_{\max}$
Result: actual velocities ω_l, ω_r
$a \leftarrow \max(|\omega_l|/\omega_{\max}, |\omega_r|/\omega_{\max})$;
if $a < 1$ **then**
  $a \leftarrow 1$;
end
return ω_l/a, ω_r/a;

Algorithm 1: Scale wheel velocities with maximum value

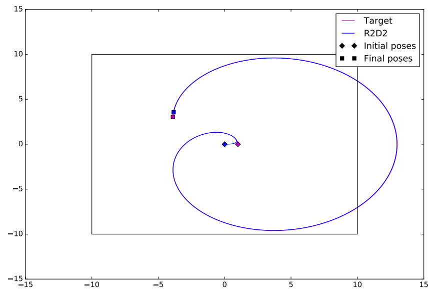
Q3 Although the robot actually moves by having its wheels rotate, it is more natural to send linear and angular velocity setpoints. Modify the `Robot::moveVW` method so that a (v, ω) setpoint is changed to a (ω_l, ω_r) setpoint that will then be called through `Robot::rotateWheels`. The inverse of model (2.2) yields:

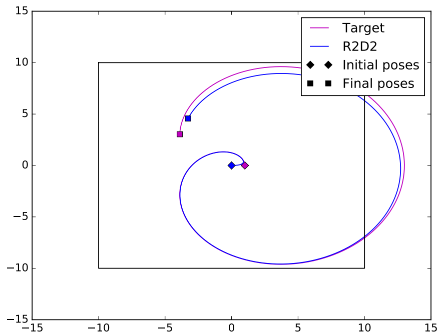
$$\left\{\begin{array}{ll} \omega_l & = \dfrac{v + b\omega}{r} \\ \omega_r & = \dfrac{v - b\omega}{r} \end{array}\right. \qquad (1)$$

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

Centrale Nantes

The corresponding behavior should be that the robot cannot follow perfectly the target anymore, because it is not fast enough.





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:

$$\begin{cases} x &= x_r + x_s \cos\theta_r - y_s \sin\theta_r \\ y &= y_r + x_s \sin\theta_r + y_s \cos\theta_r \\ \theta &= \theta_r + \theta_s \end{cases} \tag{2}$$

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:

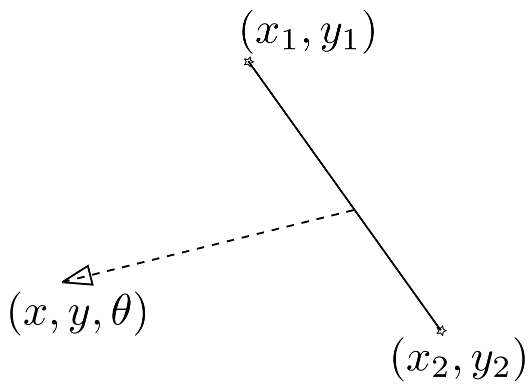


Figure 4: Distance to a segment defined by two points.

$$d = \frac{x_1y_2 - x_1y - x_2y_1 + x_2y + xy_1 - xy_2}{x_1 \sin\theta - x_2 \sin\theta - y_1 \cos\theta + y_2 \cos\theta} \tag{3}$$

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.