Advanced Robot Programming Labs
C++ Programming

Lab 2: Mobile robot with sensors

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.

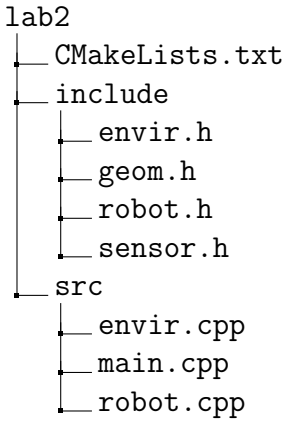

Figure 1: Files used by the simulator

The main file is `main.cpp` and is compiled to an 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 If you do the lab on your own computer

You of course have to install QtCreator and CMake:

```
sudo apt-get install cmake qtcreator
```

You will also have to install two Ubuntu packages that are used for the plotting of the robot trajectory:

```
sudo apt-get install python-matplotlib
sudo apt-get install libpython2.7-dev
```

1.3 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.

1.4 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: `Twist` with attributes `vx, vy, w`

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

- `Pose Pose::transformDirect(Pose _transform)`
  returns the pose expressed in the given transform
- `Pose Pose::transformInverse(Pose _transform)`
  returns the pose expressed in the given transform inverse
- `Twist Twist::transformDirect(Pose _transform)`
  returns the twist expressed in the given transform
- `Twist Twist::transformInverse(Pose _transform)`
  returns the twist expressed in the given transform inverse

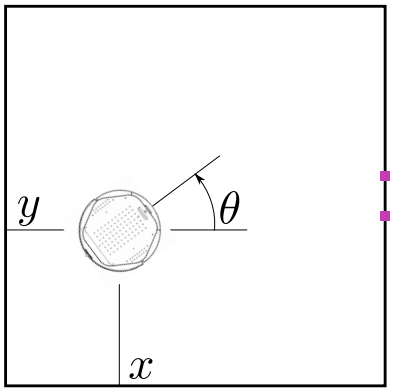

Figure 2: Robot in its environment

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

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.

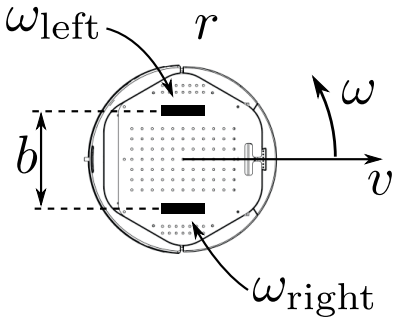

Figure 3: Differential drive model

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 not realistic. 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:

$$\begin{cases} r &= 0.07\ m \\ b &= 0.3\ m \end{cases}$$

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:

$$\begin{cases} \omega_l &= \dfrac{v + b\omega}{r} \\ \omega_r &= \dfrac{v - b\omega}{r} \end{cases} \tag{1}$$

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

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

3 Sensors

In this section we will try to avoid the walls by using some range sensors.

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

- `Sensor(Robot &_robot, double _x, double _y, double _theta)`
  initializes a sensor at a given position on the given robot
- `virtual void update(const Pose &_p)`
  updates the measurement from the sensor current position
- `virtual void checkTwist(Twist &_t)`
  checks if the given Twist is fine with the current sensor measurement
- `void updateFromRobotPose(const Pose &_p)`
  computes the sensor absolute position from the given robot pose, then calls `update`
- `void checkRobotTwist(Twist &_t)`
  computes the twist in the sensor space, then calls `checkTwist`

Some of the methods are pure virtual functions, which makes the `Sensor` class an 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*`.

For example to update all sensors with the current robot pose, we can write the following:

```
for(auto & sensor: sensors_)
    sensor->updateFromRobotPose(pose_);
```

The arrow is because we use a pointer and not an actual value. The corresponding methods have thus to be called with an arrow instead of a dot.

3.1 Range sensor

Q1 Create a `sensor_range.h` file that defines a `RangeSensor` class that is derived from `Sensor`. The `update` and `checkTwist` methods have to be defined so that the code compiles. For now, just make the methods print something to the screen. As the `Sensor` class does not have a default constructor, the `RangeSensor` class must have its constructor call directly the one from `Sensor`:

```
class RangeSensor : public Sensor
{
public:
    RangeSensor(Robot &_robot, double _x, double _y, double _theta) :
        Sensor(_robot, _x,  _y,  _theta) // call the Sensor constructor
        {// the RangeSensor constructor does nothing more}
}
```

Q2 Include this file in `main.cpp` and declare a `RangeSensor` variable. We will use a front range sensor placed at $(0.1, 0, 0)$ in the robot frame.

Q3 In the `Robot::moveWithSensor` method we currently do not take into account the sensor measurement. Before calling `moveVW` we should loop through all the sensors of the robot and call their `updateFromRobotPose` method.

Q4 In this question we will build the `update` function of the range sensor. This sensor should compute the distance to the nearest wall in the direction of the x-axis.

In the `envir_` attribute of the sensor class, 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

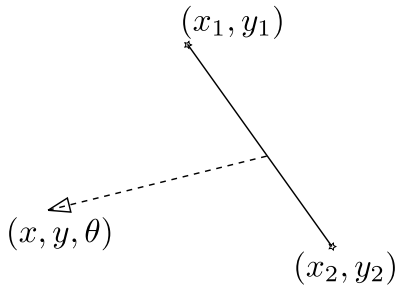

Figure 4: Distance to a segment defined by two points.

is:

$$d = \frac{x_1 y_2 - x_1 y - x_2 y_1 + x_2 y + x y_1 - x y_2}{x_1 \sin\theta - x_2 \sin\theta - y_1 \cos\theta + y_2 \cos\theta} \tag{2}$$

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.

A simple code to loop trough all the walls is:

```
Pose p1, p2;
for(int i=0;i<envir_->walls.size();++i)
{
    p1 = envir_->walls[i];
    p2 = envir_->walls[(i+1)%envir_->walls.size()];

    // do whatever you want to do with points p1 and p2
}
```

Define the `update` function so that it updates the attribute `s_` of the sensor with the distance to the nearest wall. Make it also print the distance to see if it is fine.

Q5 We now want the sensor to be able to modify the robot twist to prevent any collisions, which is the goal of the `checkTwist` method. We assume the `Twist` that is passed to this method is already expressed in the sensor frame. Hence we are only interested in the `vx` attribute and do not want it to be too large (fast motion in the sensor axis) if we are near to a wall.
A simple way to do so is to define a minimum security range and to impose an upper bound to the velocity:

Q6 Modify the `Robot::moveWithSensor` method so that we also check the current robot twist by calling the `checkRobotTwist` method for all sensors. The robot should then stay inside the walls. We will use the following parameters:

$$\begin{cases} g & = .1 \\ s_m & = 0.1\ m \end{cases}$$

Data: Current twist T, current range s, gain g, minimum range s_m
Result: Modified twist T
if $T.vx > g * (s - s_m)$ **then**
  $T.vx \leftarrow g * (s - s_m)$;
end
return T

Algorithm 2: Modify the given twist to avoid collisions

Q7 Add some other range sensors to the robot to check if everything is going fine.

3.2 Bearing sensor

In this section we will create a new kind of sensor, that returns the bearing angle to the other robot in the simulation. Define first a new robot with smaller wheels (0.05 m).
In the main loop, we just want this robot to go forward at 0.4 m/s:
`robot2.moveWithSensor(Twist(0.4,0,0));`
Of course it will go out of the wall for now.

Q1 Create a `sensor_bearing.h` file that defines a `SensorBearing` class that is derived from `Sensor`. The `update` and `checkTwist` methods have to be defined so that the code compiles. For now, just make the methods print something to the screen.

Q2 Include this file in `main.cpp` and declare a `SensorBearing` variable. We will use a front bearing sensor placed at $(0.1, 0, 0)$ in the second robot frame.

Q3 Here we build the `update` method. The sensor measurement should be the angle under which the sensor is detecting another robot. We have to loop through all the robots of the environment, which can be done with this loop:

```
// look for first other robot
    for(auto other: envir_->robots_)
        if(other != robot_)
        {
            // compute angle between sensor and detected robot

            break;
        }
    // set angle back to [-pi,pi]
```

The angle between the sensor and the robot can be computed with:

$$\alpha = \arctan2(y_r - y_s, x_r - x_s) - \theta_s$$

where (x_s, y_s, θ_s) is the sensor pose and (x_r, y_r) is the robot position.

Q4 We now build the `checkTwist` method. The bearing sensor should try to align with its target. To do so, we just want to change the rotation velocity:

Data: Current twist T, current bearing s, gain g
Result: Modified twist T
$T.w \leftarrow T.w - g \times s$;
return T

Algorithm 3: Modify the given twist to avoid collisions

We use this method with the following parameters: $g = 0.5$.

Q5 Check that the second robot now follows the first robot, just by using its bearing sensor.