Autonomous Robots – Home Work 1

Adam Tonderski, 930524-1037 29/04/2018

1 Basic Kinematics

In the section we will derive analytic expressions for $x(t)$ and $y(t)$ for the following differentially steered behaviour

$$v_L(t) = v_0(t/t_0) \tag{1a}$$
$$v_R(t) = v_0(t/t_1) \tag{1b}$$

where v_0, t_1 and t_2 are given. From the lecture notes we know the following:

$$\begin{aligned} v &= \frac{v_L + v_R}{2} \\ \omega &= -\frac{v_L - v_R}{2r} \\ v_x &= v\cos(\phi) \\ v_y &= v\sin(\phi) \\ x(t) &= x(0) + \int_0^t v_x(t')dt' \\ y(t) &= y(0) + \int_0^t v_y(t')dt' \\ \phi(t) &= \phi(0) + \int_0^t \omega(t')dt' \end{aligned}$$

Assuming $x(0) = y(0) = \phi(0) = 0$ and putting this together with (1) we get:

$$\phi(t) = \int_0^t \frac{v_L(t') - v_R(t')}{2r}dt' = -\frac{v_0}{2r}(\frac{1}{t_1} - \frac{1}{t_2})\int_0^t t'dt' = -\frac{v_0}{2r}(\frac{1}{t_1} - \frac{1}{t_2})\frac{t^2}{2} = C_1t^2 \tag{2}$$
$$x(t) = \frac{v_0}{2}(\frac{1}{t_1} + \frac{1}{t_2})\int_0^t t'\cos(C_1t'^2)dt' = \frac{v_0}{2}(\frac{1}{t_1} + \frac{1}{t_2})\frac{\sin(C_1t^2)}{2C_1} = \frac{C_2}{2C_1}\sin(C_1t^2) \tag{3}$$
$$y(t) = C_2\int_0^t t'\sin(C_1t'^2)dt' = C_2\left[\frac{-\cos(C_1t'^2)}{2C_1}\right]_0^t = \frac{C_2}{2C_1}(1 - \cos(C_1t^2)) \tag{4}$$
$$\tag{5}$$

where, to clarify, $C_1 = -\frac{v_0}{4r}(\frac{1}{t_1} - \frac{1}{t_2})$ and $C_2 = \frac{v_0}{2}(\frac{1}{t_1} + \frac{1}{t_2})$

Figure 1 shows a plot of these equations for $t \in [0, 10]$, $r = 0.12\,\mathrm{m}$, $v_0 = 0.5\,\mathrm{m/s}$, $t_1 = 10\,\mathrm{s}$ and $t_2 = 5\,\mathrm{s}$.

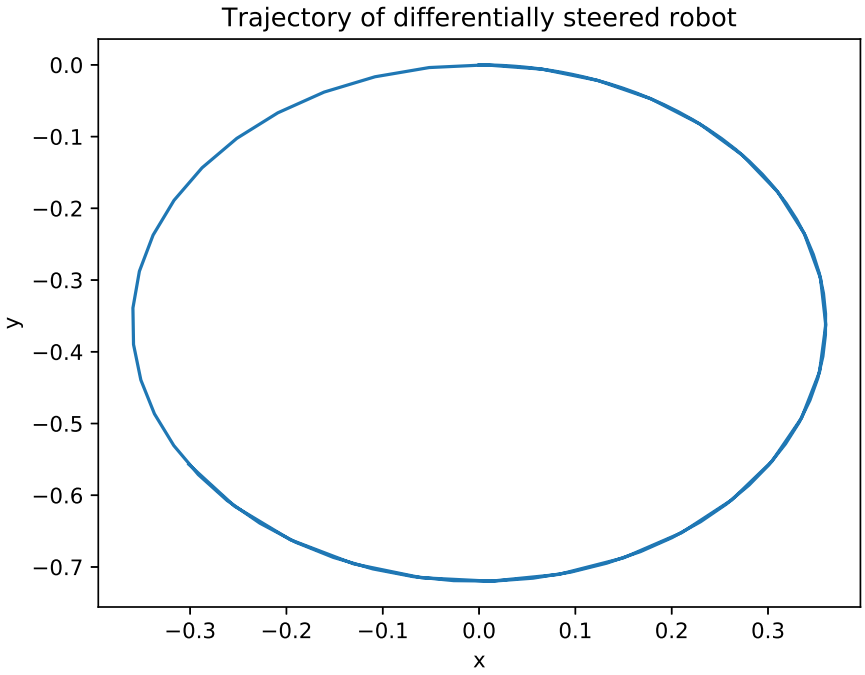

Figure 1: Python simulation of the kinematic equations.

2 Software development

a) dockerfile

A dockerfile was created that contains the helloworld example from the lecture slides. In order to verify this assignment you need to navigate to the directory that contains the code solutions (see the attachments) and run the following commands.

```
cd prob2
docker build -t myrepository/mydockerimage .
docker run --rm -ti --net=host myrepository/mydockerimage
```

b) docker compose

Here a docker-compose file was created that prints ”Hello World!” using a couple of different base images. To run it do the following:

```
cd prob2
docker-compose up
```

Docker compose simplifies running multiple containers with their respective commands. This example is roughly equivalent to running the following commands

```
docker run archlinux/base echo Hello World!
docker run ubuntu:16.04 echo Hello World!
docker run alpine:3.7 echo Hello World!
```

In this case ”network_mode:host” is completely unnecessary since all the containers run in isolation.

c) cmake

Cmake is a build tool that simplifies the process of building, testing and packaging a C++ program. In principle all of these things can be done without cmake but this would quickly get complicated as the size of a project grows. Additionally cmake tries to make the process as platform independent as possible, which avoids complications when deploying/building in different environments.

Cmake is not used instead of the compiler, but rather as a layer on top of it. From the instructions in the CMakeLists.txt it figures what commands to send to the compiler and in which order (among other things).

d) multicast

UDP is an transport layer protocol where packets are simply sent out without getting any confirmation that they have been received. This is opposed to for example TCP, where the receiver confirms that packets have been received (which allows for retrying etc). This generally means that UDP is really fast, but unreliable. This speed and simplicity makes UDP a great candidate for multicast.

Multicast is not a protocol, but rather a form of group communication where a message is replicated to many parties on the network. It can be both one-to-many (as in our libcluon microservices) and many-to-many. This is very useful in distributed systems since the sender doesn't need to be aware of each individual recipient. Instead, the message (for example a sensor reading) is sent to the entire group, knowing that it will reach anyone that is interested in that particular data.

Note that while UDP itself is unreliable, it possible to make multicast on UDP reliable by adding a higher level layer that is specialized for this particular purpose.

3 Numerical implementation of the kinematic equations

In this assignment two different libcluon microservices were written to be able to simulate a differentially steered robot.

First the "kinematics" service, which listens for opendlv::proxy::WheelSpeedRequest for the left and right wheels and convert them to a opendlv::proxy::KinematicState. The kinematic state is calculated using the kinematic equations covered in the first assignment. In order to reduce the amount of work, the code for the kinematics microservice is highly based on the opendlv-sim-motor-kiwi.

The second service, "control", sends the opendlv::proxy::WheelSpeedRequest which are processed by the kinematics service. It also listens to messages that contain sensory data, even though these are not used for this assignment. Instead, the behaviour is fixed and defined by the following equations:

$$(v_L(t), v_R(t)) = \begin{cases} (0, v_0 \frac{t}{t_1}) & \text{if } 0 \leq t \leq t_1 \\ (v_0 \frac{t-t_1}{t_2}, v_0) & \text{if } t_1 \leq t \leq t_2 \end{cases} \tag{6}$$

with $v_0 = 0.5\,\mathrm{m/s}$, $t_1 = 3\,\mathrm{s}$ and $t_2 = 10\,\mathrm{s}$.

Since the code for this and the next assignments are so similar, I have combined them. The flag that makes the behaviour follow the above behaviour is –demo. The program can be built using the following commands:

```
cd prob34/control
docker build -t atonderski/control -f Dockerfile.amd64 .
cd ../kinematics
docker build -t atonderski/kinematics -f Dockerfile.amd64 .
```

The following commands starts a simulation with the –demo behaviour:

```
cd prob34
docker-compose -f docker-compose-demo.yml up
```

Figure 2 shows a python plot that is generated using the x and y coordinates as printed by the simulation microservice (in verbose mode). Figure 3 show a screenshot that is taken from the simulation UI. In both cases the path is an expanding spiral.

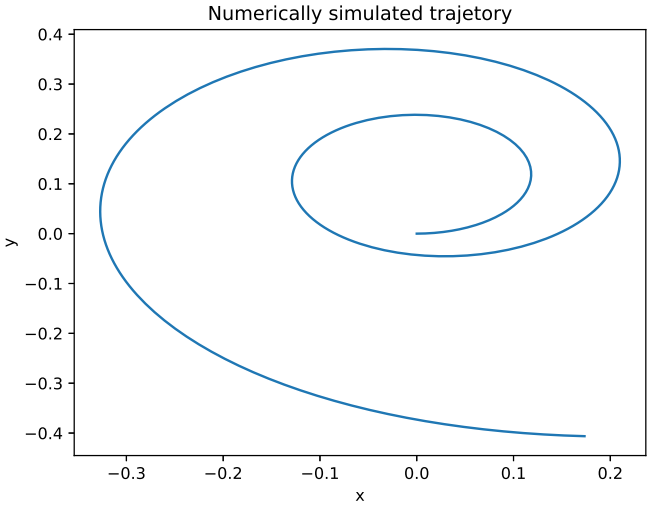

Figure 2: Python plot of the car positions. We can see that the given equations describes spiral movement.

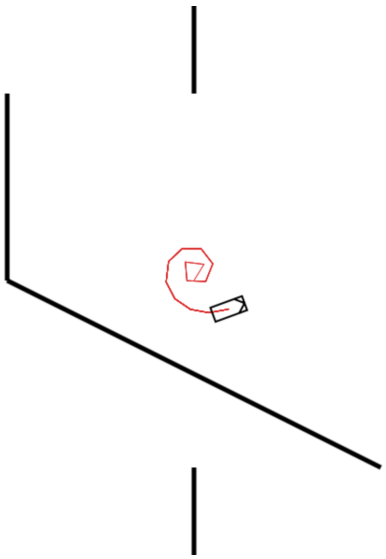

Figure 3: Screenshot of the path as captured in the simulation.

Note that the docker-compose files don't contain the entries that correspond to net=host. Instead the UI container has the necessary ports exposed. This is due to the fact that net=host is unsupported in docker

for mac, which is the environment I use. This alternative setup works perfectly fine since all containers can reach each other by default and the only outside interaction comes through the UI.

4 Simple robot behaviour

The code from the previous assignment was extended to create an exploratory random walk behaviour. The logic is pretty simple and described using a turn rate and forward velocity (instead of left and right wheel speeds).

There is a default speed that is adjusted based on the front and rear distance. Additionally there is a preferred turning angle, which has a random chance of changing the sign every time step. In order to avoid driving into obstacles, the turning direction is changed based on the proximity to the left and right.

This was enough to drive around randomly and avoid driving into walls but it didn't give enough exploratory behaviour. To rectify this, wall-following tendencies were added. For example, when approaching a wall, the preferred direction is set to left if the right distance is closer and vice versa. Also, a state variable was added that tracks whether a wall is being followed. When that variable changes to false, the preferred direction also has a high probability of flipping. Combined, these factors gave a wall-following behaviour with random tendencies to deviate.

The Figures 4 and 5 shows the paths for 2 different versions of the car. The logic is similar in both cases but in the first case the robot is unable to pass through the narrow opening in the bottom. In the second case the behaviour was improved with a sharper turn rate, which in turn forced the addition of a random turn rate decay to improve exploration in open spaces.

The containers that were built in the previous section already contain all the necessary logic so all that is needed to test the random walk behaviour is to use the default docker compose file:

```
cd prob34
docker-compose up
```

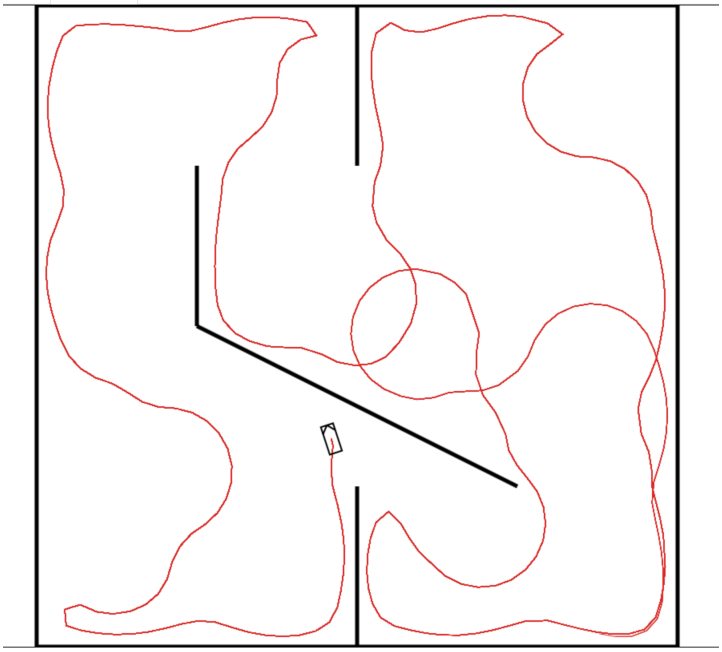

Figure 4: An earlier version of the behaviour, we can see that it drives around randomly while still managing to explore almost the entire map. However, it could not pass through the narrow passage at the bottom.

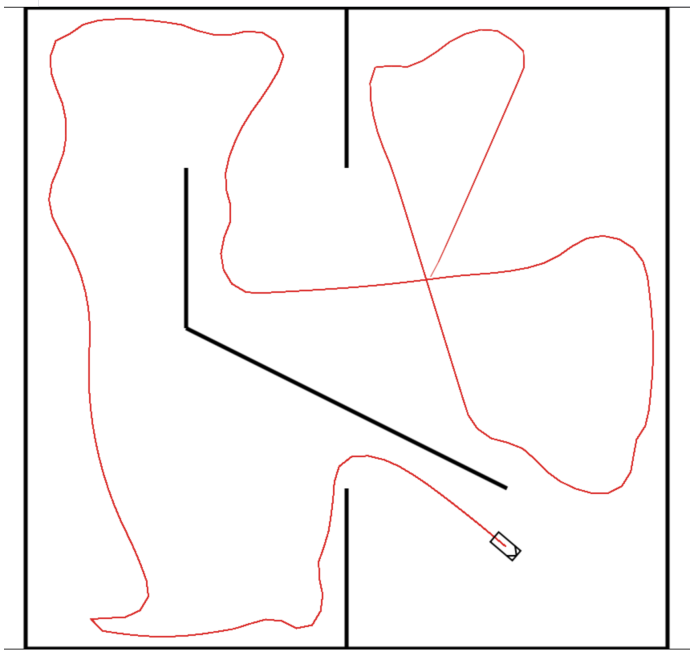

Figure 5: Path using the final behaviour. There is a little bit more structure to the path (fewer loops). This was accomplished by adding a random decay to the turn rate when the car was in open spaces, which let us increase the default turn rate. Here, the robot explores the entire map, including the narrow passage.