376.040
Fachvertiefung - Robotik und Bildverarbeitung
Selected Topics - Robotics and Computer Vision

Michal Staniaszek, Markus Leitner,
Jeremy Itamah, Matthias Hirschmanner

March 24, 2023

Contents

1 Introduction

This document is regularly updated. You can always find the latest version here.

This is the documentation for the robotics and computer vision course. In this course, you will work in a team to write software to make a robot autonomously play a simple hockey-like game. The aim of the course is to introduce you to robotic systems and computer vision in the real world, and give you experience in programming such systems in a team environment. You will need to organise and delegate tasks within the team and work together to produce a working system.

Section 2 introduces the task in more detail. Section 3 gives details on how to install the software that you need to use for the course. Section 4 gives an overview of the hardware that you will use. section 5 goes through some information about recommended software and how to use it. section 6 gives some tips regarding ROS development and some suggestions for popular control structures such as Subsumption, finite state machines and behaviour trees.

If you find any part of this document unclear, or feel that something is missing, please contact the tutors with suggestions.

1.1 Contact

To facilitate communication, we will use Discord, so that everyone can see questions, answers and other discussion. You can join the workspace using this link. If there is something that needs to be discussed that you do not want to be shared with everyone on the course, use email instead. For questions relating

to the lectures or welfare related things, such as illness or problems with the course, contact Markus Vincze. For questions about the task, contact Jeremy Itamah.

Markus Vincze (vincze@acin.tuwien.ac.at): Professor

Matthias Hirschmanner (hirschmanner@acin.tuwien.ac.at): Assistent

Robert Tamas (robert.tamas@tuwien.ac.at): Tutor

Michael Kubiczek (michael.kubiczek@tuwien.ac.at): Tutor

1.2 Lab access

You will have access to a lab in Gußhausstraße 25 (Altes EI), where a hockey field will be set up, and where there will be space to work with the robots. The room is CFEG42, with the label "ACIN V4R". To access the lab, you must ask the tutors to open it for you. Please set up a date on the Discord server so we can make sure, somebody is present to unlock the door for you. To get to the room, entering from the front entrance of GH25, turn left after the entrance hall and walk up a few steps to a glass door. After the glass door, the lab is on the right side.

2 Task

Your task for the course is to write software for a robot to participate in a game of robohockey with another robot. The objective of the game is for each team's robot to move pucks into the other team's goal. You are free to solve the task with whichever methods you choose.

2.1 Milestones

There will be four milestones during the course. The milestones are intended to give you smaller sub-tasks, which solve some part of the overall task. Grading for the course is split between an oral examination (25 %), and the evaluation of each milestone and evaluation of the final system (75 %). The grading for the practical part of the course is as follows:

- Milestone 1: 10 %
- Milestone 2: 20 %
- Milestone 3: 20 %
- Milestone 4: 15 %
- Overall system: 10 %

The grading will evaluate how well your system performs the sub-tasks, how well the different components work together, and the quality of your code.

You can receive a passing grade for each milestone even if you do not manage to complete the task perfectly. You will receive points for attempts at a solution, dependent on how well it works. To receive full points for the milestone you must demonstrate your system solving all of the requirements for the milestone.

> This task is very challenging. Do not be disheartened if you do not have a working system at the end of the course.

2.1.1 Milestone submission items

For each milestone, you are required to do the following:

- Demonstrate your work in a meeting with the tutors. You should be able to show your code running on the robot and explain your implementation. If your system works only in simulation, you will receive fewer points.

- Create a short document (1-2 pages) detailing the work on the milestone, preferably in Markdown format. If you're not familliar with it, here is a quick tutorial. Since we're hosting your code on GitHub Classroom you might want to also have a look at GitHub flavored Markdown, their extension to CommonMark.
  - Describe your approach to the problem. Why did you choose the approach you ended up using? What issues did you encounter? How did you solve them?
  - Each team member should write what they worked on for the milestone
  - You should put the milestone document, and any other documents you use to plan the system, in a `doc` folder in your repository on GitHub
- Create a tag in your git repository with the code you used in the demonstration (e.g. `git tag -a milestone1 -m 'code for milestone 1'`)

2.1.2 Milestone 1: Motion and collision avoidance (2023-03-06 to 2022-03-24)

Before starting with this milestone, you should complete the ROS beginner tutorials, which will introduce you to the basics of ROS. This milestone only needs to be tested in simulation. You should do this milestone by yourself. We will create the student teams after the first milestone.

- The robot should drive around randomly (it's up to you what "randomly" means)
- It must avoid obstacles, either by driving somewhere else, or around the obstacle.
- You can implement something basic at this stage, but remember that in milestone 4 you will need to drive around with an "obstacle" in view all the time!
- (Optional) Consider basing your system on one of the control architectures described in Section 6. Having a structured system early on can save you a lot of integration work at the later stages.

2.1.3 Milestone 2: Field component detection (2023-03-24 to 2023-04-28)

- Implement detection algorithms for all the field components — pucks, poles, goals as well as the enemy robot. Your system should be able to process the availabe information (laser, rgb/depth images, pointclouds) and indicate which of the field components are present, and where in the image or laser data they are.
- It is not enough to print the data to the console, indicate present field components visually in a debug image, for example by drawing bounding boxes arround them or publish appropriate messages (e.g. rviz/DisplayTypes/Marker) so they can be visualized in rviz.

2.1.4 Milestone 3: Field dimension detection and localisation (2023-04-28 to 2023-05-12)

> You are not allowed to use the `gmapping`, `amcl` or similar packages for this milestone.

- Your robot should be able to detect the dimensions of the field, to within approximately 30 cm of the actual value.
- Your robot should be able to detect its position on the field, based on any of the information that the field provides. You can use puck locations, goal locations, and pole locations to do this. Publish a ROS message (e.g. geometry_msgs/Pose) with the robot's pose at regular intervals. Again the values should be accurate to at least 30 cm. The values should be relative to a coordinate system of your choice, centered on one of the corners of the field.
- Your robot should be able to avoid driving outside of the bounds of the field.

2.1.5 Milestone 4: Puck manipulation, scoring, and referee communication (2023-05-12 to 2023-06-02)

- Use your robot's ability to detect pucks and goals, and know its own location, to drive up to a puck and move it into the opposing team's goal.
- You should also be able to communicate with the referee in order to play the game.

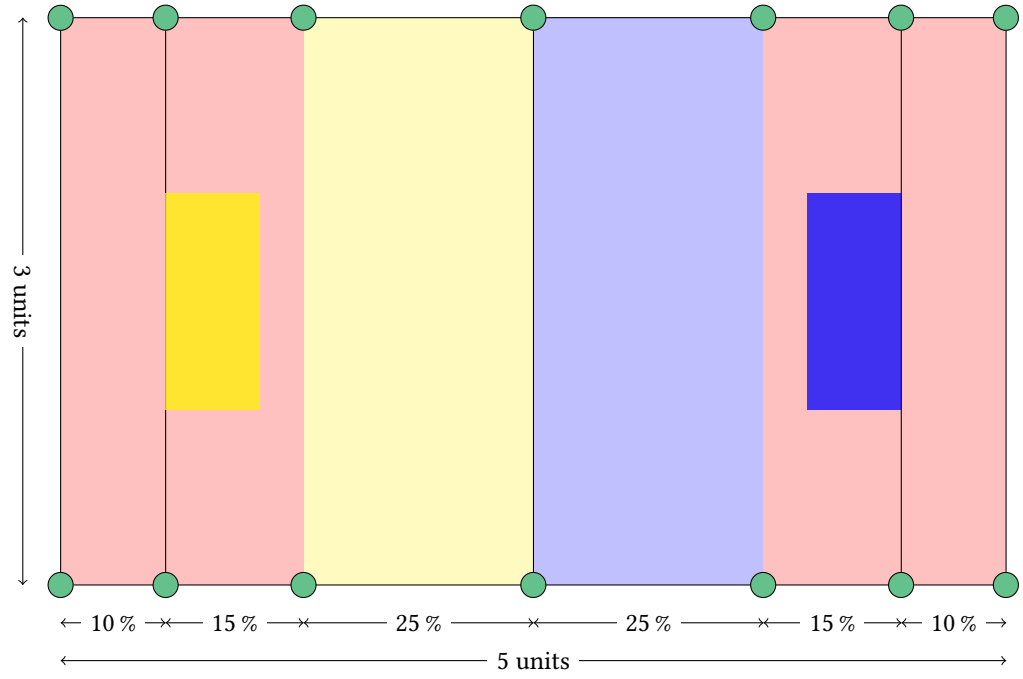

Figure 1: Layout of the field. Black lines show the boundaries, midline, and the line which defines where the goal is placed. Green circles are poles delineating the boundaries. The goals are the yellow and blue rectangles on either side of the field, and are not to scale. The robot for each team starts in the red area, and pucks start in the light yellow or light blue areas corresponding to their colour.

2.1.6 Competition (2023-06-02 to 2023-06-30)

Refine the components and make sure that they work together properly, so that you can play the game against another team.

2.2 Field setup

The field is made up of three components: poles, pucks, and goals. The poles define the bounds of the field, and can be used to determine the field size. The pucks are used to score goals. The goals are regions which the pucks must be moved into to score goals.

The field has a length:width ratio of 5:3; a field 5 metres long is 3 metres wide. Poles on the sides of the field indicate its length and width. There are two goals, one in each half of the field. There are six pucks on the field, three for each team. You can see the layout in Figure 1.

2.2.1 Poles

The field bounds are delineated by 14 poles, 7 on each long side. The poles are spaced along each side based on percentages of the length of the field. There is a pole at each end of a side, and the 5 poles in between those two corner poles are distributed such that the second pole is 10 % of the length of the field from the corner, and the third 25 %. The fourth pole is at 50 % of the length of the field. The fifth is at 75 %, and sixth is at 90 %. The layout is symmetrical from the midpoint.

The poles used to delineate the field have the following properties:

- height: 50 cm
- diameter: 8.5 cm
- colour: green

2.2.2 Pucks

Each team has 3 pucks which are used to score. They start in the middle 50 % of the field, with each team's pucks on their side of the field.

The pucks are stacked cylinders with the following properties:

- lower part height: 18 cm
- lower part diameter: 10 cm
- upper part height: 17 cm
- upper part diameter: 6.5 cm
- total height: 35 cm
- colour: yellow or blue

2.2.3 Goals

The goals do not scale with the field size; they will always have a fixed size of 1 m × 0.5 m. They are placed with the back of the goal on the line between the second pole from the end of the corresponding team's side of the field. They have the same colour as their team's pucks.

3 Getting Started

In this course we will use Ubuntu 20.04. You can get the desktop image here. When you have the ISO, you can create a bootable USB drive by following the tutorial for the OS you currently use (Windows, MacOS, Ubuntu). To install, reboot your PC with the USB stick inserted, and in the boot screen press the key that brings you into the boot medium selection. This is usually one of the F-keys. Here is the official tutorial for the installation process. If you don't have much experience with Unix, there is a basic tutorial here. MIT's missing-semester lectures are also very useful.

3.1 Install packages

For ROS noetic, follow the instructions to install the `desktop-full` version here. Some additional ROS packages are required. Install them by entering the following commands in a terminal:

```
sudo apt install ros-noetic-joy
sudo apt install ros-noetic-teleop-twist-joy ros-noetic-teleop-twist-keyboard
sudo apt install ros-noetic-ros-control ros-noetic-ros-controllers
```

We also install `git`, which will be used for version control, and `catkin_tools`, which helps with compilation of ROS packages.

```
sudo apt install git python3-catkin-tools
```

3.2 Create and initialize a new ROS workspace

Create a new workspace directory somewhere in your home directory, e.g. at `~/robohockey_ws`, and intialize a new ROS catkin workspace. You need to source `/opt/ros/noetic/setup.bash` first, to have access to the catkin commands.

```
source /opt/ros/noetic/setup.bash
mkdir -p ~/robohockey_ws/src
cd ~/robohockey_ws
catkin init
```

3.3 Download code

Before you can download the code, you will need to get access to the Github Classroom. After accepting the invite, we will need to unlock you before you have access to all repositories in the organization. However, you can already do the ROS beginner tutorials before continuing with the next steps if you do not have full access to the Github organization yet.

You will need to set up the permissions so you can clone the required repositories from Github. The easiest way to set up the required permissions is to create an SSH key for your account and add it to your Github account as described here.

To download the simulation environment and the referee, enter the following in your `src` directory:

```
# If using Access Tokens:
git clone https://github.com/v4r-robohockey/game.git

# Or if using an SSH key:
git@github.com:v4r-robohockey/game.git
```

If your personal GitHub Classroom repository has already been created, clone it into your src directory. If not you can clone (but not push to) the default player repository until then:

```
git clone https://github.com/v4r-robohockey/player.git
```

Once your team's repository has been set make sure to clone it into your src directory instead of the default one.

Now, your workspace should contain the directories game and player. `game` contains the provided environment, you should not change its contents (you also cannot commit any changes to this code to the git repository). `player` contains your empty code construct, which you can work on as a team and commit changes to your team git repository.

3.4 Compile your workspace

```
cd ..
catkin build
```

All packages should compile without any errors and your workspace should now have two additional directories `build` and `devel`. If you do get an error try building again, sometimes catkin needs two passes to generate all dependent files. Warnings for the referee package can be ignored.

Make sure your code compiles on your computer before compiling on the robot!

3.5 Set your ROS environment variables

```
source ~/robohockey_ws/devel/setup.bash
```

To have ROS variables and commands available automatically each time you open a new shell, add the setup command to `/.bashrc`, which you can do like this:

```
echo "source ~/robohockey_ws/devel/setup.bash" >> ~/.bashrc
```

3.6 Test your installation

You should now be able to start the simulation environment:

```
roslaunch hockeysimulator start_hockey_world.launch
```

If the launch is successful, two new windows should appear, the simulation (gazebo) and rviz. To place the pucks and the robot randomly on the field, open another terminal window and run

```
roslaunch hockeysimulator init_hockey_world.launch
```

The initial positions of pucks and robot can be defined in the configuration file

```
game/hockeysimulator/cfg/init_hockey_game.yaml
```

To drive the robot around using your keyboard, start a teleoperation node (works as long as the terminal window from which the node is called has the input focus):

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py /cmd_vel:=/robot1/cmd_vel
```

Once you've checked that your system runs properly, you should complete the ROS beginner tutorials.

3.6.1 Player node

The player node is a simple skeleton you can use to familiarise yourself with some of the ROS internals and interact with the simulation. You can start the player node with the following command:

```
roslaunch player player.launch
```

By default, this launch uses the C++ version of the player node. You can launch the python version by adding `use_python:=true` to the command.
You can also use `rosrun` to run each node, but you will need to manually remap the topics so that they match the topics which are published by the simulation.

Python version:

```
rosrun player player_node.py cmd_vel:=/robot1/cmd_vel kinect/rgb/image_raw:=/robot1/
    kinect/rgb/image_raw front_laser/scan:=/robot1/front_laser/scan
```

C++ version:

```
rosrun player player_node cmd_vel:=/robot1/cmd_vel kinect/rgb/image_raw:=/robot1/kinect/
    rgb/image_raw front_laser/scan:=/robot1/front_laser/scan
```

If you cannot get the robot to move, or don't receive messages you expect, make sure that your topics are all correctly lined up. Use `rostopic list` to check that there aren't more topics than you expect, and use `rostopic info` on a specific topic to see which nodes are publishing and subscribing to it.

3.6.2 Compiling while running the simulation is slow

You will probably find that you use almost all your CPU for the simulation, which will probably result in very long compile times. To mitigate this, you can temporarily stop the simulation processes. These are usually called `gzserver` and `gzclient`. You can see currently running processes with the `top` command, or using the `ps` command.

```
ps ax | grep gz
26429 pts/0    Sl+    2:58 gzserver
26370 pts/0    Rl+   40:35 gzclient

top
top - 14:51:09 up  6:41, 12 users,  load average: 6.47, 4.30, 2.58
Tasks: 312 total,   8 running, 303 sleeping,   0 stopped,   1 zombie
%Cpu(s): 72.5 us, 13.7 sy,  0.0 ni, 12.8 id,  0.0 wa,  0.0 hi,  1.0 si,  0.0 st
KiB Mem : 16150880 total,  2157768 free,  8985180 used,  5007932 buff/cache
KiB Swap: 39062524 total, 39062524 free,        0 used.  6428288 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26429 michal    20   0 3788768 503688 105324 R 121.3  3.1   4:27.45 gzserver
26370 michal    20   0 2676512 479512 141208 R  98.7  3.0   3:38.60 gzclient
```

Here, the server and client have the PIDs 22749 and 22750. To stop these processes temporarily, use `kill -STOP PID` to stop the process, and `kill -CONT PID` to restart.

3.7 Errors

3.7.1 `Ogre::RenderSystem::setDepthBufferFor`

If upon launching Gazebo you encounter the following error, it may be the case that you are running in a virtual machine (which is not recommended).

```
Ogre::RenderSystem::setDepthBufferFor(Ogre::RenderTarget*): Assertion `bAttached && "A
    new DepthBuffer for a RenderTarget was created, but after creation" "it says it's
    incompatible with that RT"' failed.
```

A solution may be to switch to a native installation. Alternatively, see this issue, which proposes some fixes, the easiest of which is to install a version of Gazebo above 7.4. See this page on how to install a newer version. You should install `gazebo7` rather than `gazebo9`, and use the "step-by-step" installation rather than the one-liner installation.

3.7.2 `ASSERT: "false" in file qasciikey.cpp`

There isn't yet a proper fix for this, just a workaround. If you remove the lines

```
cv::imshow(IMAGE_WINDOW, image);
cv::waitKey(10);
```

from `playnode.cpp` and run `catkin build player`, you should be able to run the player node. This will mean that you won't be able to see the image the robot is seeing. You can instead use rviz to view these images by publishing them on a topic.

3.8 On the robot

The robots should both be set up with the required ROS environments and programs. You should set up a workspace for your team on the robot, much like you have done on your own computer. Do not add a line to `.bashrc` to source the setup script in your workspace, as this will interfere with the work of other teams. You should instead create a tmuxinator configuration which sources your workspace in each pane, see Section 5.4.2. You can also add an alias which will source your workspace, see Section 5.6.1.

3.8.1 Workspaces

The `pioneer_ws` currently contains the `robot` package, which has a few launch files that you can use to launch the whole system. The `robohockey_setup` directory contains various configuration files, and this documentation, should you need them for any reason.

You should create a workspace for your team in the home directory and keep the files you create inside that directory, so that there is less likelihood of clutter.

> Do not copy a workspace which you have previously set up. This will likely cause misconfiguration and things won't work.

3.8.2 Connecting the robot driver

The interface to the robot hardware is controlled by `RosAria`. You should be able to find the robot USB-serial connection at `/dev/pioneer`. Make sure that the USB connector, usually a silvery one, is connected, and that the robot base is on. Note that you should only need to run the robot driver if you want to move the wheels. After starting `roscore`, you can run the robot driver independently of other code with

```
rosrun rosaria RosAria _port:=/dev/pioneer
```

This will give you several topics, and you can move the robot around using the `cmd_vel` topic. For the most part you will not run the driver in this way.

3.8.3 Starting the system

Usually, you will want to start the hardware using

```
roslaunch robot robot.launch
```

This will start the connection to the robot itself, as well as creating topics for the kinect, camera, and laser. There are various options available for launching, and you should inspect the file for more details. Brief details are:

- `use_kinect:=true` use the kinect. This is `true` by default
- `use_aria:=true` run the robot driver. This is `true` by default. Run with this set to `false` if you do not need to move the robot
- `use_rplidar:=true` use the laser. This is `true` by default

3.8.4 ROS tools

Once you have run things, you should make sure the topics you want to use are in the namespaces you expect them to be, and that there is something being published there. Use `rostopic list` often to make sure topics exist. Check individual topics with `rostopic info` to see what nodes are subscribing and publishing to them. You can look what nodes exist with `rosnode list`, and see which topics a node is subscribing or publishing with `rosnode info node_name`. Check if messages are appearing with `rostopic hz` or `rostopic echo`. You can also view all of the nodes and topic connections present in the system using `rqt_graph`.

3.8.5 Network setup

It is convenient to configure the network so that you can access the `roscore` running on the robot remotely. First, you must configure your network so that you can ping the robot and the robot can ping you. See the ROS network setup page for more specific details. To summarise, you should add the robot to your `/etc/hosts` file, and your machine to the robot's `/etc/hosts` file. If you had the hostname `richter`, with the IP 192.168.50.128, and the robot is `rupaul` at 192.168.50.100 you would add

```
192.168.50.100 rupaul
```

to your `/etc/hosts`, and

```
192.168.50.128 richter
```

to `/etc/hosts` on `rupaul`.
You should then be able to ping `rupaul` from your machine, and `richter` from `rupaul`.
Once you have done this, you need to point ROS to where the `roscore` is running:

```
export ROS_MASTER_URI=http://rupaul:11311
```

Doing this will mean that `rostopic` or other ROS commands will show you what is happening on the robot rather than on your machine. You will need to do this in each terminal where you want to view topics on the robot.

3.8.6 Visualisation

You can use `rviz` on your PC to visualise the data that the robot is receiving. Start an rviz instance, and then add topics that you are interested in.

If you like, you can display a model of the robot using

```
roslaunch robot view_model.launch run_state_pub:=true run_join_pub:=true
```

and then adding the `TF` and `Robot Model` visualisation from the "add by display type" dialogue. You should run this on your own PC. If you run it on the robot, there will be issues with the paths to the robot model components and you will not see much.

3.8.7 Recording data

You may wish to record data for convenience, so that you can test things when you aren't in the lab. ROS provides a very useful tool for this, called `rosbag`. With `rosbag record topic_1 topic_2 ... topic_n` you can record specific topics (any number you like). If you want to record everything (not recommended because the file will be extremely large) you can use `rosbag record -a`.

Once you have recorded a bag file, you can replay it with all the correct timings for message publishing and so on, with `rosbag play filename`. You can also remap topics as usual, for example

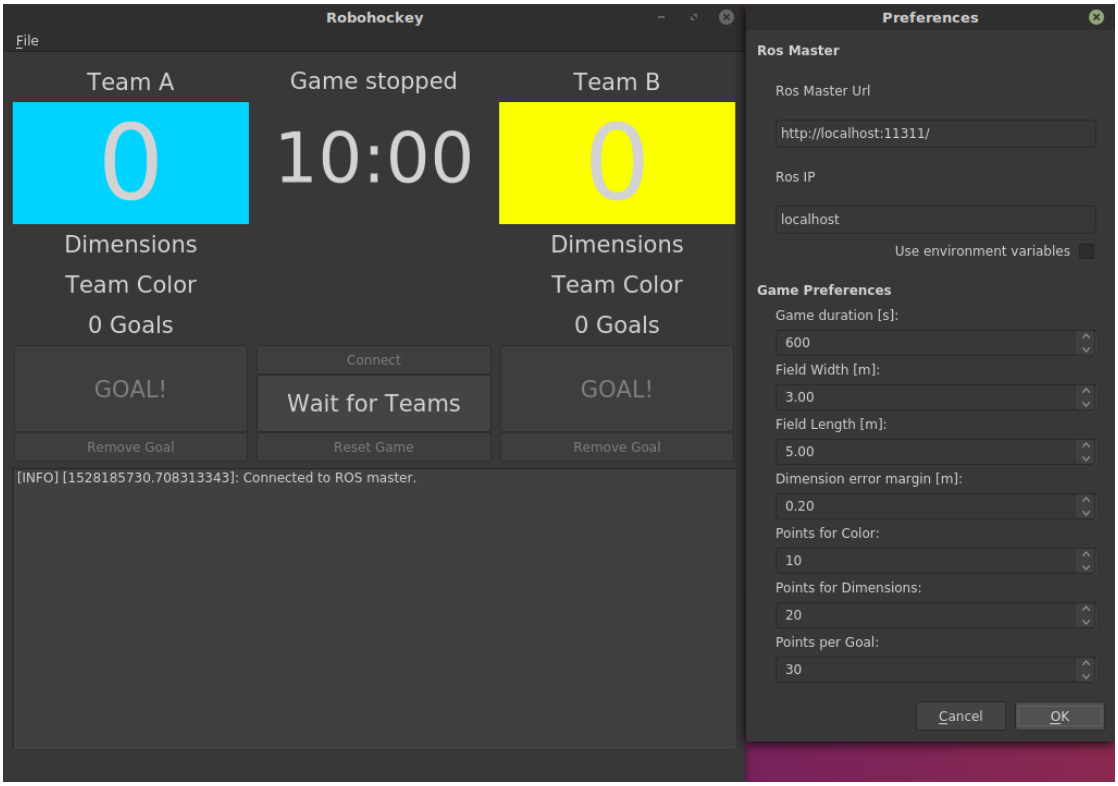

Figure 2: Referee GUI.

```
rosbag play my_bag.bag /camera/image_raw:=/robot1/camera/image_raw
```

will remap the camera topic. You can use the `-l` switch to loop the bag. Once you reach the end of the recording, the bag will start playing again from the beginning.

3.9 Using the referee

Your system will need to be able to interact with the referee by the end of milestone 4. The referee is a Qt GUI with ROS connectivity. To run the system in the same way it will be run during the competitive games, you should run the system with roscore on a laptop, and have the robots connect to that. See Section 3.8.5 for more about that. To be specific, on the robot, you should run `ROS_MASTER_URI=http://[your ip here]:11311` in the terminal, *before* you start the tmux. Otherwise you will have to run it in every tmux pane. On your PC, you should start the roscore. You can check to make sure things are working by using `rostopic pub -r1 /test std_msgs/Empty "{}"` on the robot and making sure that you can see the topic and messages on your PC. You can then run `rosrun referee referee_node` on your PC, and you should see the GUI pop up.

In the GUI, you can adjust parameters using the file menu and going into preferences. You should make sure to set the network information correctly. By default it is set to use localhost, but when interacting with the robot you may wish to be more specific and use the IP your laptop has on the network. If it is set up correctly, clicking the connect button should produce a success message in the text box, as shown in Figure 2. After that, you need to click the wait for teams button to start waiting for the team name messages. One the messages have been received from both teams, the start game button will start the timer. To simulate the system, you can send requests to the various services with `rosservice call` and see the results in the GUI. Consult the flowchart in Figure 3 for specific information about the system.

3.9.1 Published topics

Topic name	Message type	Information
`/gameControl`	`std_msgs/Bool`	**true** game started **false** game ended
`/waitForTeams`	`std_msgs/Empty`	Published message indicates that referee is ready to register teams

3.9.2 Provided services

`/TeamReady` is called by the robot to register with the referee.

Request: `string team` — calling team name

Response: `bool ok` — True if registration successful, false if other team used the same name

`/SendColor` is called by the robot to send the detected team colour

Request:

- `string team` — calling team name
- `string color` — "blue" or "yellow"

Response:

- `bool ok` — true if received colour is correct
- `string correctColor` — correct team colour, "blue" or "yellow"

`/SendDimensions` is called by the robot to send detected field dimensions

Request:

- `string team` — calling team name
- `geometry_msgs/Point dimensions` — detected dimensions in metres (`.x` length, `.y` width)

Response:

- `bool ok` — true if dimensions are within error margin
- `geometry_msgs/Point correctDimensions` — correct dimensions in metres (`.x` length, `.y` width)

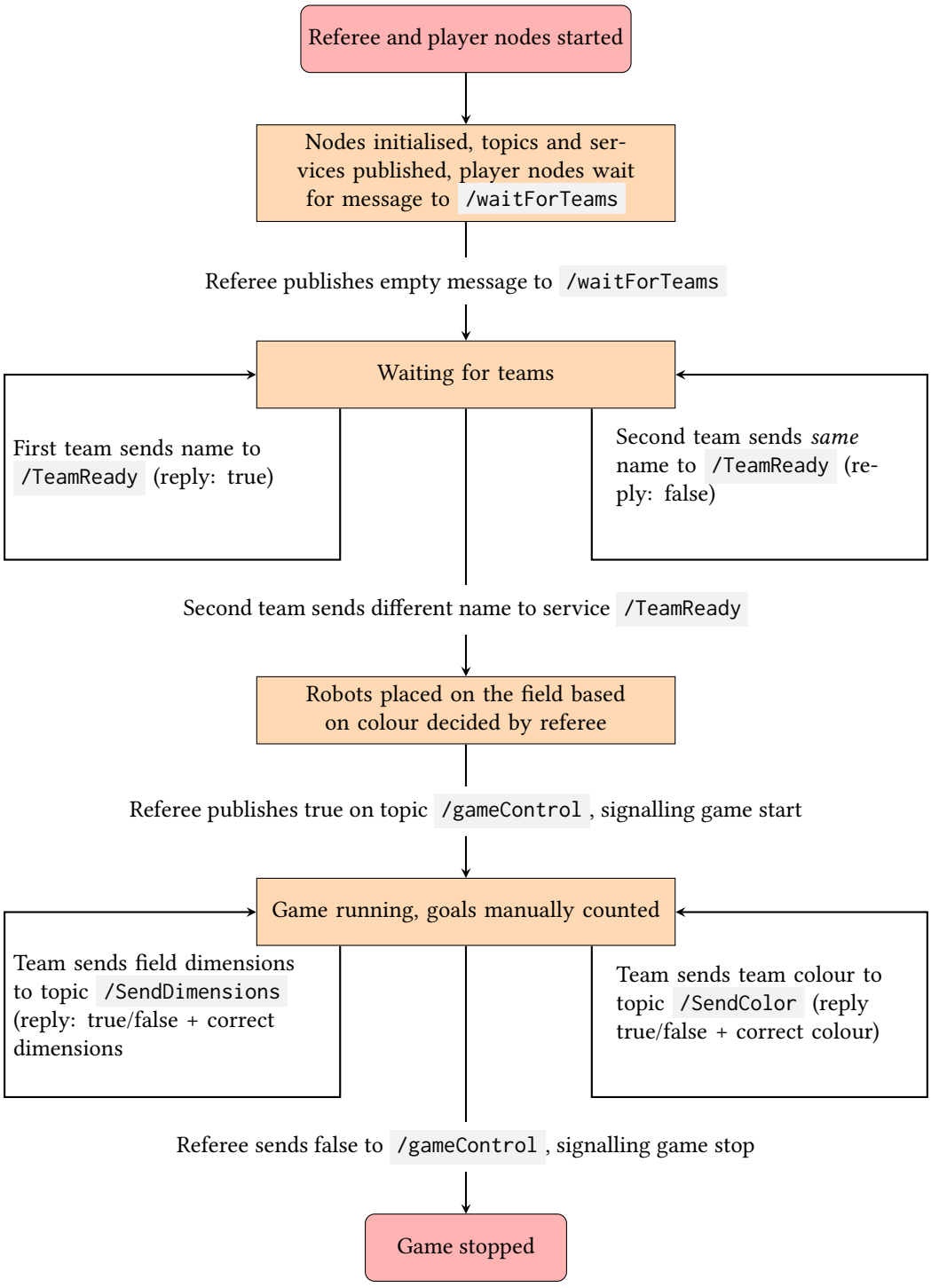

Figure 3: Flowchart of the referee system

4 Hardware and Networking

4.1 Pioneer P3-DX

In this course, you will use Pioneer P3-DX robots. You can find a manual here, which contains some information about the system.

4.1.1 Batteries

The pioneer uses 12 V, 7 A sealed lead-acid batteries. It can have up to 3 batteries in it at a time, and the batteries can be hot-swapped to allow for longer operation. You should make sure that batteries are balanced in terms of weight distribution. If using only one battery, place it in the central slot, and if using two, place them in the side slots.

In order to not unnecessarily drain the batteries, you should only power the PC and kinect using the Pioneer's batteries if you really need to. If you do not need to move the robot, connect them to mains power instead.

Charging Charging current for the batteries should be set at a maximum of 4 A, and you should ensure that the charger is in the mode for charging 12 V batteries, if it has that option.

There are several chargers available which can be used to charge batteries. You can find manuals for all of the chargers near the router.

- Charge batteries before they drop below 10.5 V output. You can check the current voltage with the `battery_voltage` topic.
- Do not attempt to charge the batteries while they are inside the robot! You *must* take them out before charging.
- You should only charge a single battery per charger
- Do not short circuit battery contacts. Disconnect the batteries from the charger or remove them from the robot if you want to measure the voltage.

Voltcraft charger The voltcraft charger has several charging options. The battery voltage should be automatically detected as 12 V, 6 cells. You will then need to enter the amperage of the battery 7 A, and select which type of lead-acid battery it is (option 2, lead-gel). Finally, select the "CHA-laden" option.

Ansmann charger When you connect this charger to a battery, the lights on the right side will flash green, and then the 12 V light should remain illuminated. With this charger, you can select the charging current by pressing the "Mode" button after the battery is detected. When the battery is charged, the "100 %" light will turn on.

IVT charger This is a small charger with a switch on the left side. Ensure that the switch is set to 12 V before charging the batteries. When the battery is fully charged, the yellow "Akku voll" light will turn on.

4.2 XMG Laptop

The onboard computer is a XMG Core 15 laptop running Ubuntu 20.04, with 16 GB RAM, an Intel Core i7-9750H processor and Nvidia GeForce GTX 1660 Ti graphics card.

The laptop will automatically connect to the robohockey network. The password for the `ros` user is `Pion44R!`.

The battery life of the laptops is fairly limited so you should always plug in one of the chargers labelled 'xmg-laptop' when the robot is stationary.

4.3 Network

The lab network is labelled `robohockey` with password robohockey. There is no internet access through this router so if you have to look something up you'll have to connect to the tunet on your own laptop. If you need to simultaneously have internet access and communicate with the robots you

can use an ethernet cable to connect to the robohockey network and access the tunet via the wi-fi. Don't forget to configure the wired connection to use a static IP as explained in section 4.3.1.

4.3.1 Connecting your laptop

User	IP
xmg-01	192.168.1.101
xmg-02	192.168.1.102
A. N. Other	192.168.1.128

You should set up your network connection so that you have a fixed IP address on the network. You can do this in the connection manager:

1. Connect to the network as usual
2. Open the "network connections" program either using the super (windows) key, or clicking on the network icon in the tray
3. Select the robohockey connection and click "edit"
4. Go to the "IPv4 settings" tab
5. In the "method" dropdown, select "Manual"
6. Add a new address, and enter your desired IP, within the range 192.168.1.xxx, where xxx >102, which is not already in Table 4.3.1 or in use by anyone else (see the IP list), a netmask of 24, gateway 192.168.1.1, and DNS 8.8.8.8
7. Disconnect and reconnect to the network

You should now be able to see other devices on the network. Check if things are working by pinging either of the two laptops, using the `ping` command with one of the addresses in section 4.3.2. You should also ask someone to try and ping your machine, to make sure other devices can see you.

4.3.2 Robots

The two robots have fixed IP addresses on the network:

```
xmg-01: 192.168.1.101
xmg-02: 192.168.1.102
```

To log in to the machine, use

```
ssh ros@192.168.1.101
```

and enter the password for the 'ros' user.

You may also wish to modify your `/etc/hosts` file so that you can use the hostname instead of the IP to connect to the robot. Just add `192.168.1.101 xmg-01` to the file, and you should be able to `ping xmg-01`. You can also make it easier to log in to the robot by sending your ssh public key. See this question for instructions. Basically, you need to do the following:

```
ssh-keygen
ssh-copy-id ros@xmg-01
```

You should then be able to log in using `ssh ros@xmg-01` without having to type the password.

4.3.3 Transfering files to the robot

Since there is generally no internet access on the robot laptops it is necessary to transfer your workspace files over the network. You should use rsync or scp to do this. Here is a basic tutorial for rsync.

4.4 Kinect

The Kinect is an RGBD sensor produced by Microsoft. Unlike more recent RGBD sensors, the Kinect requires 12 V power, and as such cannot be run purely from USB power. When you are doing static testing, you should not use the robot's batteries to power the sensor. If you want to move around, then you can use the yellow connector on the robot. The sensor gives you several different outputs:

- RGB
- Depth
- IR
- Point Cloud
- Coloured Point Cloud

These are all easily accessible via ROS topics. If you want to use the sensor connected to your own laptop, libfreenect is required (Kinect is not compatible with OpenNI). This requires you to build libfreenect and the freenect stack from source.

```
mkdir ~/external_repos

# Install libfreenect
cd ~/external_repos
git clone https://github.com/OpenKinect/libfreenect.git
cd libfreenect
mkdir build
cd build
cmake ..
make
sudo make install

# Install freenect ros stack
cd ~/<your catkin workspace>/src
git clone https://github.com/ros-drivers/freenect_stack.git
cd ..
catkin build
```

You can then launch the Kinect with

```
roslaunch freenect_launch freenect.launch
```

The topics of most interest to you are probably the following:

`depth/image_raw` The raw grayscale depth image, with each pixel corresponding to the depth of that pixel

`depth/points` Point cloud with XYZ data only

`depth_registered/points` Point cloud with XYZRGB data

`rgb/image_raw` Raw RGB image. Note that you may need to use `rgb/image_color` instead

For pointcloud processing we recommend that you use C++, as this is the language in which PCL is written. There are python bindings, but this covers only a subset of the entire library and as such may be missing features that you need. The ROS wiki has a page on PCL which gives examples of how to use it. PCL itself also has extensive tutorials.

5 Software

The Pioneer robots run Ubuntu 20.04 with ROS Noetic, and you should have the same installed on whichever PC you want to use to work on this course, to ensure you have access to the same programs.

5.1 ROS

The Robot Operating System is a middleware which can be used to build robotic systems, and is commonly used both in industry and research environments. In this course we will be using ROS noetic.

5.1.1 Terminal emulators

With ROS, there are often many terminal windows open at the same time, and it can be annoying to switch between windows. One way of mitigating this problem is to use a terminal emulator which can handle multiple terminals in the same window. Some examples are `terminator`, `yakuake`, and `guake`.

5.2 git

`git` is a version control system. It is used to keep track of changes to files, and is an invaluable tool for programming projects of any size. You can find an interactive tutorial at GitHub, one of the main hosting sites for git repositories, and a very short non-interactive one which covers the most useful commands and concepts here. Atlassian has another set of tutorials. Make use of them or stackoverflow, or ask the tutors if you don't understand something!

If you want to make use of pull requests, it may be a good idea to also fork the repository (or repositories) that you will be developing into your own user space. Pull requests are useful because they provide a space for discussion of the code that is to be added to the main repository, and can help filter out errors.

It is not required that you use only a single repository for all development. You should feel free to create whatever repositories you need, if you would like to split your system into different packages for different purposes. This may make it easier to split up the functionality of the system, rather than keeping it in a single package. You may wish to convert the player package to a ROS metapackage, which will sit in a single repository but have distinct packages for each part of the functionality. You should discuss this with your team and decide which approach you wish to take.

- Favour committing small chunks of code often, rather than large blocks rarely
- Write useful commit messages. Something like "changed file" is not helpful to anyone, see this blog post for some tips on do's and don'ts when writing commit messages.
- Try to ensure that `master` is always compilable. Consider working on new features in separate branches and merging them in to master later.

5.2.1 Merge conflicts

When working in teams, you will likely be editing the same files, which means that there may be changes which conflict. In the ideal case this wouldn't happen because people would always work on different code sections, but in messy reality this is very rarely the case.

In our small example, we have a test repository on a server somewhere which functions as the remote `origin`. This is the repository which everyone has access to.

We have two people working on the repository, Alice and Bob, who each have a repository on their machines. Currently there is only a single empty file in the repository, `edit-me.txt`.

Bob adds some text to the file, commits it, and pushes it to the repository.

```
echo "bob's text" > edit-me.txt
git add -u
git commit -m 'add bob text'
git push
```

Alice does the same thing on her PC

```
echo "alice's text" > edit-me.txt
git add -u
git commit -m 'add alice text'
git push
```

However, when she tries to execute the push command, she gets this:

```
To https://git.server.com/example/test_repo
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://git.server.com/example/test_repo'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

When things don't work, it's often a good idea to check `git status`, which might indicate what the problem is. Here's the output of that command:

```
git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
nothing to commit, working directory clean
```

Because Bob pushed his changes to the repository, and the changes were on the same file, there is now a divergence between the code on the remote server, and the code on Alice's PC, which must be resolved. To start this process, we use `git pull`

```
git pull
pAuto-merging edit-me.txt
CONFLICT (content): Merge conflict in edit-me.txt
Automatic merge failed; fix conflicts and then commit the result.
git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   edit-me.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The status indicates which files were successfully merged, and which we have to look at manually. Sometimes, if the lines which were changed do not overlap, then the automatic merge will work without an issue, in which case you can carry on with what you were doing. Otherwise, as in this case, Alice must now resolve the merge conflict. The file now looks like this:

```
cat edit-me.txt
<<<<<<< HEAD
alice's text
=======
bob's text
>>>>>>> e68cfb4e32103cd5f8f7e39b91dcdf156adb8cb7
```

`HEAD` refers to the changes that Alice has in her branch. The SHA-1 hash shows the reference to the commit which is conflicting, which we can find if we look at `git log` in Bob's version of the repository:

```
git log
```

```
commit e68cfb4e32103cd5f8f7e39b91dcdf156adb8cb7
Author: Bob Bobson <bobson@company.com>
Date:   Mon Mar 5 16:30:14 2018 +0100
    add bob text
```

Alice would like to keep both bits of text, so she edits the file to look like this:

```
  alice's text
  bob's text
```

Now, Alice and Bob might want to argue about the ordering of that text, in which case they need to talk to each other (as should you, if necessary, when doing merge commits). In this case we assume that Alice is the more senior programmer and what she says goes.

Now, Alice can add the file and commit as usual, which creates a merge commit, indicating that two divergent branches have been merged:

```
  git add -u
  git commit
```

The `git commit` command will bring up a text editor in which you can modify the commit message. In the case of Alice's merge commit, it looks like this:

```
  Merge branch 'master' of https://git.server.com/example/test_repo

# Conflicts:
#       edit-me.txt
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit each, respectively.
#   (use "git pull" to merge the remote branch into yours)
#
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   file1.txt
#
```

Now if we look at the status of the repository, we can see what's happened:

```
git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
git log
commit 7125ab902b0a293e7cef9c80bb51fc79ade8c937
Merge: 464080f e68cfb4
Author: Alice Alisson <alisson@company.com>
Date:   Mon Mar 5 17:04:03 2018 +0100

    Merge branch 'master' of https://git.server.com/example/test_repo

commit 464080f16b9fd78c6cf573c94c90db91dbaebf52
Author: Alice Alisson <alisson@company.com>
Date:   Mon Mar 5 17:03:32 2018 +0100
    add alice text
```

Now that the merge commit is complete, Alice can use `git push` to update the remote repository. This is one of the ways to deal with merge conflicts, and there are various arguments between programmers about what the best approach is. Feel free to read about alternatives and decide on which method you would like to use (e.g. here).

5.3 tmux

`tmux` is a terminal multiplexer program which works similarly to `screen`. It allows for the use of terminals which persist over sessions. You can connect to the robot via ssh, start up tmux, and it will be available to you later even if you log out. This allows you to keep commands running, and is generally very useful. Multiple people can also view the same tmux session, but cannot use it at the same time as keypresses from different sources will interfere. You can find a (very) detailed introduction here, but you should be fine with just the basics.

5.3.1 Terminology

pane a single terminal.

window like a tab, which can contain multiple panes, and have a name.

session the main unit of `tmux` instances, which can contain multiple windows. Different sessions can be accessed in parallel. You can see currently running sessions from the commandline with `tmux ls`.

5.3.2 Commands

You can find a tmux cheatsheet here.

From the commandline, you can start a new tmux session and attach to it with

```
tmux new -s session_name
```

You can attach to a session with

```
tmux a -t session_name
```

You can kill an existing session with

```
tmux kill-ses -t session_name
```

These are the commands that you will most likely need to use. If you want to see what other commands exist, use `man tmux` or look at the many cheatsheets or tutorials that exist.

To use a command when inside a tmux session, you must first enter the tmux prefix key combination, which is `C-b` (ctrl-b). Some basic commands:

`C-b "` split the window vertically, creating one pane above another

`C-b %` split the window horizontally, creating one pane next to another

`C-b c` create a new window

`C-b x` kill the current pane

`C-b` ↑ / ↓ / ← / → move to pane in given direction

`C-b q [0-9]` move to pane with the given number

`C-b n` move to the next window

`C-b p` move to the previous window

`C-b [0-9]` move to window with given number between 0 and 9

`C-b '` move to window with the given number after entering the number

`C-b w` show interactive list of windows to select from

`C-b [` enter scrolling mode for current pane, scrolling as for usual terminal

`C-b d` detach from the current session

5.3.3 On the robot

When the robot PC starts, a tmux session named `core` should already exist. This session is intended to stay active for the whole time the system is running, and starts `roscore` and `rosaria`, which means that you should not have to run a roscore yourself. If the robot is not turned on when the PC starts up, you will have to rerun the `rosaria` command if you want to connect to the robot. You can of course restart the session if necessary using `tmuxinator start core`.

Be careful not to interfere with the tmux sessions of other teams!

5.4 tmuxinator

We will use `tmuxinator` to make it easier to preserve your tmux sessions over shutdowns of the robots. tmuxinator allows you to specify how a session should be created, and which commands should be run when it is created. You can see an example of a basic configuration file here.

You may wish to create a configuration file for your team, so that you can define what sort of layout you want to have, and customise it to whichever ROS nodes you end up using.

You can create a new tmuxinator configuration with

```
tmuxinator new config-name
```

This will open an editor and allow you to edit the file (created in `/.tmuxinator/team_config.yml` as you wish. Alternatively, you can just copy an existing configuration and modify it:

```
cp ~/.tmuxinator/example.yml ~/.tmuxinator/team_config.yml
```

If you copy the core config, make sure that you comment out the `attach:` line, as this non-default behaviour that is used to make sure that the core session can run on login.

To use your configuration, open a terminal and run

```
tmuxinator start team_config
```

To edit your configuration, run

```
tmuxinator open team_config
```

If your editor isn't set, you can set it to 'vim' for example with this command:

```
export EDITOR=vim
```

5.4.1 Wait before sending commands

When tmuxinator starts tmux, all commands you specify for each pane will be run. Sometimes you will not want this, as nodes must be started in a specific order, or maybe you only want to run a subset of your nodes. You can get around this with the `read` command, which will wait until the terminal receives a return keypress before moving on to the next command. The following is a short example configuration which demonstrates how this works:

```
# ~/.tmuxinator/read-test.yml

name: read-test
root: ~/

windows:
  - editor:
      layout: tiled
      panes:
        - echo "I will be executed straight away"
        - read && echo "I will be executed when you press the return key"
```

This is hacky, but limitations on the program mean that it's not possible to do this in a nicer way.

5.4.2 Sourcing your workspace

To source your workspace in each pane, add the following to your yaml file:

```
pre_window: source ~/team1_ws/devel/setup.bash
```

5.5 Coding tools

You can develop your code in any program you like. Below are a few suggestions you might wish to consider. In general it is better to compile and run programs from the commandline using `catkin build` inside your workspace rather than from inside an IDE. The ROS wiki has a page on IDEs.

5.5.1 VSCode

VSCode has become a very popular general purpose editor with a rich extension ecosystem. Be sure to check out the (language specific) documentation here.

Live Share If you have a GitHub or Microsoft account, you can use the live share extension which enables several people to remotely edit the same file simultaneously.

5.5.2 PyCharm

PyCharm is a Python IDE you can install with

```
sudo snap install pycharm-community --classic
```

and run with `pycharm-community`. It works well with the ROS python libraries, providing function previews and so on without any requirement for additional configuration. Students also get the pro version for free which you can apply for here.

Debugging If you want to debug a node which does not take any parameters, add breakpoints where you want them by clicking to the right of the line numbers, and then press `Shift+F9`, and you will be dropped into the debugger when the breakpoint is hit.

If you want to debug a node with arguments, open the file you want to debug and then select Run > Edit Configurations. Here, enter the parameters to send to the node.

Here is a basic script which you can use to try things out:

```
#!/usr/bin/env python
import rospy

if __name__ == '__main__':
    rospy.init_node("test_node")
    test_param = rospy.get_param("~test_param", None)

    if test_param:
        rospy.loginfo('Got test param "{}"'.format(test_param))

    loop_rate = rospy.Rate(0.5)
    loop_count = 0

    while not rospy.is_shutdown():
        rospy.loginfo("This is a test")
        loop_count += 1
        loop_rate.sleep()
```

To populate `~test_param`, which is a parameter in the local namespace of the node (i.e. if you used `rosparam list`, you would see it as `/test_node/test_param`), you should add a parameter `_test_param:=test`. The underscore before the parameter name indicates to ROS that you want to set the local parameter with that name.

If you use `roslaunch`, or run the node from outside and you want to debug it, you will need to use Run > Attach to process. For this to work, you first need to enable `ptrace`, which allows processes to control other processes (i.e. it's pretty dangerous). To allow ptrace temporarily, run

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

PyCharm has a short tutorial on other options and attaching to processes.
For the above test node, put some breakpoints into the file as usual, and then run

```
rosrun test_package test.py _test_param:=test
```

Then, in PyCharm go to the attach to process dialog, and select the process from the list that appears.
One problem with this approach is that you will have trouble debugging the node initialisation. A simple solution for this is to add a sleep before the initialisation so that you can attach to the process before it runs past the parts you want to debug.

5.5.3 gdb

`gdb` is a text-based C++ debugging tool which can be run from the commandline. Its usage is more complex than debugging tools found in most IDEs, but it is very powerful. You can install it with

```
sudo apt install gdb
```

To use it with a ROS node written in C++, you should run the node with

```
rosrun --prefix "gdb --args" my_package my_node
```

After you have set up breakpoints you can run the node with the `run` command. You can also (since version 7.2) save breakpoints that have been set up using

```
save breakpoints <filename>
```

5.6 Aliases

Aliases are used in the terminal to allow you to run commands with a shortcut rather than having to type out the whole thing. For example, we can create an alias `xmg-01` which will run the command `ssh ros@xmg-01` by adding the following line to `~/.bash_aliases`:

```
alias xmg-01="ssh ros@xmg-01"
```

The following aliases have been added to the pioneers:

- `tac`: Attaches to the `core` tmux session

If you find that you need to use a certain command a lot, let us know so we can add it to the list for future years.

5.6.1 Workspace alias

To add an alias to move you quickly to your workspace, sourcing its setup file, add something like the following to `.bash_aliases`:

```
alias t1w="source ~/team1_ws/devel/setup.bash && cd ~/team1_ws/src"
```

To use this alias in existing terminals you will need to re-source the bash configuration. To do this enter `bash` into the terminal.

6 Implementation

6.1 General coding recommendations

Since this is a team project, you should try to make things understandable for other team members, not just for yourself. We will also be looking at the code for grading purposes. You should try and do the following:

- Comment your code. Explain any difficult logic or things that might not be obvious to someone reading the code. You should not comment every line.
- Use good variable and function names. Prefer longer, more descriptive names over short ones. Try to avoid one-letter variable names unless you're working with coordinates or something similar.
- Be consistent with your naming conventions. There are conventions recommended by ROS for C++ and python (PEP 8).
- Don't Repeat Yourself. If you find yourself writing or copying the same code block multiple times, consider writing a function you can call instead.

6.2 ROS

You should try to make use of the tools that ROS provides for you:

- Prefer having more nodes over making one node do many things
- Use services for when you need to do request-response type actions
- Use launch files to consolidate nodes needed for certain tasks into an easily manageable group
- Use the parameter server to define configuration parameters for nodes and services. You can load all the parameters in a launch file and test their effects without having to recompile code.
- Use dynamic_reconfigure to change the values of parameters at runtime, this is incredibly useful for testing.
- Use ROS printing (e.g. ROS_INFO, rospy.loginfo, etc.)

6.3 Control architectures

While working on your system, you should consider using an existing control architecture. If you choose to go in another direction, you may find it difficult to debug problems with the command flow. It is also likely to be difficult to add new elements to the system, making integration of components more difficult.

You can find a repository which contains examples of the architectures here. You can launch an example of each architecture with

```
roslaunch control_architectures {type}_test.launch
```

where `type` can be `subsumption`, `smach` or `behaviour_tree`.

To demonstrate the architectures, we will use a simple example robot which is supposed to randomly drive around, avoid obstacles, and take pictures of objects. These are the three states or behaviours of the system, and are implemented using an actionserver.

Drive randomly In this state, the robot will drive randomly until it encounters an object or an obstacle. It is implemented in `drive_randomly_server.py`

Take picture In this state, the robot will take a picture of the object it has discovered, which may come out blurry. It is implemented in `take_picture_server.py`

Avoid obstacle In this state, the robot will move and attempt to avoid the obstacle, but has a small chance of crashing. It is implemented in `avoid_obstacle_server.py`

Note that all of the following examples are not necessarily the most optimal or correct way of setting up the structure of the system. There are often many ways of achieving an equivalent end result, and it depends on your preference as to how you want to set things up.

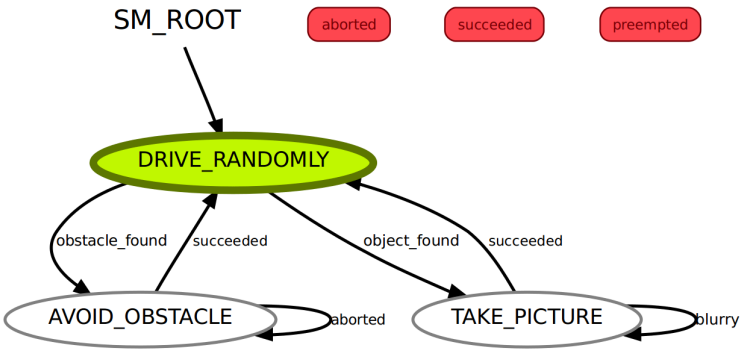

Figure 4: Representation of the simple robot system in SMACH.

6.3.1 Finite state machines

Finite state machines are the most commonly used control architecture in robotics. The state machine has states, which are behaviours, and transitions between states, which move the system from one state to another depending on the output of the currently running state.

In ROS, state machines are implemented in SMACH, which comes with a good set of tutorials. Figure 4 shows the `smach_viewer` visualisation of the state machine. The system starts in the `DRIVE_RANDOMLY` state, and based on the output of that state will transition to either of the two other states. If `AVOID_OBSTACLE` fails to avoid the obstacle, then it will try again, and if `TAKE_PICTURE` gets a blurry result, it will also try again. You can find the code for this example in `smach_test.py`.

```
smach.StateMachine.add("DRIVE_RANDOMLY",
                       SimpleActionState("drive_randomly",
                                         RandomDriveAction,
                                         result_cb=random_drive_cb),
                       transitions={"object_found": "TAKE_PICTURE",
                                    "obstacle_found": "AVOID_OBSTACLE"})
```

Here we add the drive randomly state. Instead of a function in the second argument to handle the execution of the state, we interface with the actionlib servers using the `SimpleActionState` class, which is a built-in SMACH class that does all of the necessary communication and checking with minimal code in its creation. This requires a topic on which the actionserver exists, the action that the server is expecting, and we also add a callback for when the result is received so that we can receive the result object returned by the actionserver. The last parameter is the transitions, which define what state the FSM will transition to based on the outcome of the actionserver (or the function, if you use that).

Here we are using two custom outcome names, to make the transitions more understandable. By default, the `SimpleActionState` has the outcomes `preempted`, `aborted`, and `succeeded`, all of which are automatically generated by the object depending on the outcome of the actionserver. Our outcomes depend on the result of the action. The `drive_randomly` actionserver returns success if either an object or obstacle is detected. As such, we use the `result_cb` parameter to define a callback which is entered when the actionserver returns.

```
@smach.cb_interface(outcomes=["obstacle_found", "object_found"])
def random_drive_cb(userdata, status, result):
    rospy.loginfo("random drive result: {}".format(result))
    if result.obstacle_found:
        return "obstacle_found"
    elif result.object_found:
        return "object_found"
```

Using the `@smach.cb_interface` decorator, we define these custom outcomes. The callback takes the result of the actionserver and simply returns a string for the outcome depending on the contents of the result.

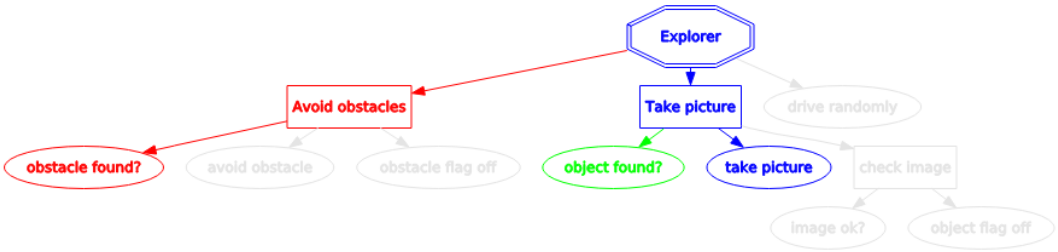

Figure 5: Representation of the simple robot system in a behaviour tree. Red highlight is failure, green is success, blue is running, and grey is inactive.

This is the basic mechanism for constructing SMACH state machines. You can see the rest of the code required in the example files in the repository. There are detailed SMACH tutorials on the ROS wiki.

6.3.2 Behaviour trees

Behaviour trees are often used to control AI agents in computer games, and there has been some interest in the possibility of their use on robots.

Because they are still relatively new in robotics, there is not yet a canonical package to use in ROS. py_trees_ros is a python implementation with documentation here.

A behaviour tree is a tree structure which consists of *behaviours* and *composites*. A behaviour is a leaf on the tree, and performs some kind of action. When it is initialised, a behaviour is in the inactive state. This then transitions to the running state. The running state can transition into either success or failure.

A composite is also a behaviour, but its purpose is to define control flow through the tree. There are four built-in composites: *selector*, *sequence*, *chooser*, and *parallel*.

Sequence Runs all its children in sequence. If all succeed, its state is success. If any fail, its state is failure, and any behaviours after the one that failed are not executed. While behaviours are running it is in the running state.

Selector Selects which behaviour to run based on their priority. If a behaviour fails, it will try the next one. If any behaviour succeeds, its state is success, and if all fail its status is failure. All behaviours are checked every tick, and a running behaviour can be interrupted by a higher priority one.

Chooser The same as the selector, but once a behaviour starts running, it cannot be interrupted.

Parallel Ticks each of its children every time it runs, giving a basic form of parallelism.

A *tick*, mentioned above, is how the tree is executed. What this means in practice is that each behaviour, starting from the root, is checked, and the ticking function gets the result of the behaviour. Depending on that result, the tick will move through the tree, which is the decision part of the tree. The tick rate of the tree defines how often ticks happen. Importantly, it is expected that ticking a behaviour should be very fast. As such, they should not themselves perform any significant computation, simply check variables or input, and do some basic operations. In practice this means that often a behaviour will be attached to an actionserver somewhere which performs the actions, and the behaviour checks the state and output of this actionserver. Alternatively, the behaviour subscribes to a topic (which runs callbacks in a separate thread), and the input is partially processed in the callback, setting some flags that the tick method can look at and decide the state of the behaviour.

You can see the behaviour tree for our simple system in Figure 5. The resulting system is essentially equivalent to the one in the FSM example. The tree should be read from left to right, as that is the order of prioritisation of children. Sequences are rectangles, selectors are octagons, choosers are double octagons, and parallels are notched rectangles (see documentation). You can find the code for this example in `behaviour_tree_test.py`.

To build the structure of the tree, we create each component and then add them as children to other components:

```
obs_avoid = py_trees.composites.Sequence("Avoid obstacles")

obstacle_found = py_trees.blackboard.CheckBlackboardVariable("obstacle found?",
                            variable_name="obstacle_found",
                            expected_value=True,
                            clearing_policy=py_trees.common.ClearingPolicy.NEVER)

# Create other behaviours...
obs_avoid.add_child(obstacle_found)
obs_avoid.add_child(avoid_obstacle)
obs_avoid.add_child(obstacle_avoided)
```

With behaviour trees, the main way of passing information between behaviours is through the use of a *blackboard*, which is accessible from any behaviour and contains all the information that is required to make decisions. There are built-in behaviours which can be used to check and modify these variables.

```
class TakePicture(py_trees_ros.actions.ActionClient):
  def __init__(self, name="take picture", action_namespace="take_picture"):
    super(TakePicture, self).__init__(name, TakePictureAction, TakePictureGoal,
     action_namespace)
    self.blackboard = py_trees.blackboard.Blackboard()
    self.blackboard.blurry = False

  def update(self):
    status = super(TakePicture, self).update()

    # Check the result and put the boolean value about whether or not the
    # image is blurred into the blackboard so it can be checked later
    if status == py_trees.common.Status.SUCCESS:
      result = self.action_client.get_result()
      self.blackboard.blurry = result.blurry

    return status
```

Above is an example of a behaviour implementation which interacts with the actionserver. We take advantage of the methods in the superclass to do most of the work for us. The `update` function is the one which is called when a behaviour is ticked. We update the value of the `blurry` variable in the blackboard so that we can use it in the behaviour which checks if the image was OK. Of course, we could instead check the value in this behaviour, and simply return failure if it is blurry. This illustrates that there are multiple ways of creating the same results. While we could get the same result with fewer behaviours, this would probably make it more difficult to understand what the robot is doing. By having more behaviour we are able to be more explicit about what is going on, and which checks failed or succeeded, which can be useful when working out problems with the system.

6.3.3 Subsumption

Subsumption is a very basic control architecture. Robot behaviours are arranged in a hierarchy, which defines their priority. Each behaviour receives inputs from the system and decides whether it should be activated. The highest priority behaviour which wants to be activated runs.

There does not appear to be a ROS implementation of this architecture at the moment. If you would like to use it, we have created a basic implementation that you can use—please contact us if you find bugs or have suggestions. You can import it by cloning the `control_architectures` repository into your ROS workspace, and using

```
import control_architectures.subsumption as subsumption
```

You can find the example at `subsumption_test.py`.

There are two basic behaviours implemented, the `SubsumptionBehaviour`, which is abstract and requires you to override its methods, and `ActionClientBehaviour`, which can interaction with an actionserver, and only requires that you override the `wants_to_run` method.

```
class RandomDrive(subsumption.ActionClientBehaviour):

  def __init__(self, name="random_drive",
               action_type=RandomDriveAction,
               goal=RandomDriveGoal(),
               topic="drive_randomly"):
        super(RandomDrive, self).__init__(name, action_type, goal, topic)

    def wants_to_run(self, value_dict):
        # Lowest level behaviour always wants to run
        return True

    def is_finished(self, value_dict):
        result = self.client.get_result()
        if not result:
            return False

        # dictionaries are mutable
        value_dict["object_found"] = result.object_found
        value_dict["obstacle_found"] = result.obstacle_found
        return True
```

Here, we create the behaviour which interacts with the random drive actionserver. The superclass sends goals to the server, and also stops the server when preempted. Since this behaviour always wants to run, we just return true from the `wants_to_run` method. `is_finished` is run every time the controller executes its loop. In the current implementation we are using a dictionary as an analogue of the blackboard in behaviour trees, so that behaviours can set data for other behaviours to check. The `value_dict` is passed to `wants_to_run` (usually to check if something is set) and `is_finished` (to set something).

```
  avoid_obstacle = AvoidObstacle()
  take_picture = TakePicture()
  drive_randomly = RandomDrive()
  behaviours = [avoid_obstacle, take_picture, drive_randomly]
  sc = subsumption.SubsumptionController(behaviours, loop_rate=2)
  sc.start()
```

We create three different behaviours for each of the states, and then arrange them in a list, in order of priority. The 0th item is the highest priority. Then, we create a controller, which will loop over the behaviours, doing the various checks for priority and so on, deciding which behaviour should execute.