Distributed Cooperative Robot Systems

Lecturers and Students

October 31, 2018

Contents

1. Introduction

1.1. Purpose of this Document

Instead of giving the students a lot of Powerpoint slides that compress the complex topics of this course into a bunch of bullet points, we wanted the students to have a dedicated document for preparing their exam. Unfortunately, such a document is a lot more work and it needs several iterations of thorough reviewing. Therefore, we came up with the compromise that the students write this document theirselves and the lecturers only have to review and improve it.

Maybe you ask: "What is wrong with Powerpoint?"

A Powerpoint presentation should aid a presenter in presenting a certain message or information to his audience. The slides alone often miss critical parts (that couldn't easily condensed into bullet points) and are unclear without the corresponding vocal line of the presenter. Nevertheless, students end up alone with the slides trying to prepare their examinations.

Apart from that, try to google topics like "disadvantages of powerpoint" or "why we should ban Powerpoint" and you will find interesting articles that connect Powerpoint and the Accident of the Columbia Space Shuttle [1] [2].

1.2. How to Contribute to this Document?

Basically there are two aspects. At first you need to get the LaTeX source code of this document from our cnc-turtlebots GitHub Repository. It is located in the folder `doc/latex`. Anybody can download it, but for making changes to it you need to have an GitHub-Account which was given the corresponding privileges. That is typically arranged during the first session of this course.

This leaves us with the requirements, that you are able to write LaTeX and know how to use GIT in combination with GitHub. Regarding GIT and GitHub we recommend to read Section 3.1.

Maybe LaTeX is hard to master when it comes to tables, graphics and customising the layout of a document, but you don't have to do that for contributing to this document. You should only be able to write text referencing images and place useful links from time to time. Furthermore, you should be able to structure the content into chapters, sections, and subsections. Everything just mentioned is already done in this document, so you can start by copying commands from the corresponding sections.

For further reference and an in-depth study of LaTeX, we recommend the following sources:

LaTeX – eine Einführung und ein bisschen mehr: A little bit longer introduction to LaTeX basics.

LaTeX WikiBook: Nice for looking up simple stuff.

tex.stackexchange.com: Stack Overflow for LaTeX

TikZ: LaTeXpackage for designing advanced graphics directly in LaTeX.

The final issue is to decide which editor you choose for compiling this LaTeXdocument. We recommend TexMaker, because it is available for Ubuntu and Windows, although, there currently exists an issue with short cuts under Ubuntu 16.04 LTS. Consider this Ask Ubuntu Post for solving it.

With TexMaker you only have to open the `Software_Reference_Book.tex` file and click on `Quick Build`. If you get compile errors, you probably need to install the necessary LaTeXpackages. Under Ubuntu this is done by executing the following console command:

```
sudo apt install texlive-full
```

1.3. How to Setup your Development-PC?

The development environment recommended to develop software for the TurtleBots of the Distributed Systems Department is the result of best practises over years and continuously under development. Nevertheless, we try to keep this section up-to-date. There also exist scripts for doing these steps, however for the purpose of getting hands on experience we recommend to drudge through the drudgery at least once and thereby trying to understand what is actually going.

1. **Install Ubuntu** The currently used operating system is Ubuntu 18.04 LTS (long-term support) with all available updates installed.

2. **Install Ubuntu - Extra Packages** Simply install the package with the following command: `sudo apt install vim git ssh myrepos capnproto libcapnp-dev libqt5x11extras5-dev qtscript5-dev liblua5.1.0-dev bison re2c libcanberra-gtk-module:i386`

3. **Install ZMQ** Execute the following lines in a terminal:

```
sudo su
echo "deb http://download.opensuse.org/repositories/network:/
   messaging:/zeromq:/git-draft/xUbuntu_18.04/ ./" >> /etc
   /apt/sources.list
exit
wget https://download.opensuse.org/repositories/network:/
   messaging:/zeromq:/git-draft/xUbuntu_18.04/Release
   .key -O- | sudo apt-key add
sudo apt install libzmq3-dev
```

4. **Install Clingo** Clone the Clingo Repository and follow their installation instructions.

3. **Install ROS Desktop Full** The currently used ROS version is ROS Melodic Morenia with all available updates installed. Like Ubuntu 18.04 it is also a long-term support version. The instructions for installing ROS under Ubuntu can be found on the corresponding ROS Wikipage.

4. **Install ROS - Extra Packages** Simply install the package with the following command: `sudo apt install ros-melodic-costmap-2d ros-melodic-urg-node ros-melodic-base-local-planner ros-melodic-nav-core ros-melodic-kobuki-msgs ros-melodic-joy ros-melodic-ar-track-alvar`

5. **Create ROS Workspace** We recommend to follow this folder structure, as things can get complicated otherwise.
   - `cd`
   - `mkdir -p rosws/src`

6. **Checkout Github Repositories** You need access to our GIT repositories on GITHub. Therefore, please provide your GITHub Username to us. We will grant you the corresponding permissions. Furthermore, it is recommended to setup ssh access to your GITHub account, in order to avoid typing the passwords over and over again:
   - `ssh-keygen` followed by pressing enter until the command terminates. ;-)
   - Follow the GITHub Tutorial about adding SSH-Keys to your account.

   The next steps are for checking out the repositories under the assumption that you have the corresponding permissions:
   - `cd ~/rosws/src`
   - `git clone git@github.com:carpe-noctem-cassel/alica.git`
   - `git clone git@github.com:carpe-noctem-cassel/clingo.git`
   - `git clone git@github.com:carpe-noctem-cassel/cnc-turtlebots.git`
   - `git clone git@github.com:carpe-noctem-cassel/supplementary.git`
   - `git clone git@github.com:carpe-noctem-cassel/symrock.git`
   - `git clone git@github.com:carpe-noctem-cassel/turtlebot.git`

7. **General Configurations** You need to configure some of the installed tools:
   - **git:** Copy the 'gitconfig' file from the cnc-turtlebots/configuration folder into your home folder and rename it to '.gitconfig'. Open the '.gitconfig' file and replace the 'email' and the 'name' in the 'user' section.
   - **myrepos:** Jump into each repository and execute: `mr register`
   - **.bashrc:** You need to enter the lines from Appendix A.1 into your '.bashrc' file.

- **mr branches:** Add the following lines at the beginning of the '.mrconfig' file in your home folder:
  ```
  [DEFAULT]
  branch = git branch
  ```

8. **Switch Repository Branches** During the lecture, we are working on the following branches of the corresponding GIT repositories:
   - alica - master
   - clingo - master
   - cnc-turtlebots - pkvr
   - supplementary - ttb_dev
   - symrock - zmq
   - turtlebot - pkvr

   The main work will take place within cnc-turtlebots, supplementary, turtlebot, and symrock. That is why the master branch (incl. branch protection) for alica and clingo are ok for us. In order to switch a branch execute `git checkout <branch name>` within the repositories root folder.

9. **Compile Workspace** Within the 'rosws/src' folder execute `catkin build`.

10. **Setup CLion**

1.4. Unix Basics

Differences between Unix and Windows can be found in this nice article. This special article about the different filesystems is also very enlighting to read for Unix newcomers.

In order to operate easily with a Unix-based operating system like Ubuntu, you should understand the following terminal commands including their common parameters

cd Changing the directory, including special folders like '.', '..', and ' '.

mkdir Creating folders, including '-p' for nested folders.

rm Deleting files and '-r' for folders.

ls Showing the content of the current folder.

ll Like ls, just including several more information like permissions etc.

grep Finding strings inside of files, incl. the meaning of each of the three letters '-inr':
`grep "test" .`

find Finding files: `find . -name "regex-expression"`

vim Terminal-based editor for quick editing. Useful commands ':wq', 'ESC', 'i'

ssh Connecting to another machine goes like: `ssh -pPORT USER@IP`,
e. g., `ssh -p222 cn@141.51.122.141`

ssh-keygen Creates a new ssh-key (or overwrites the old one).

Here are some resources for your own studies. Nevertheless, our experience is that, if you are open and willing to learn, attending the lecture in an attentive way, will be enough to understand everything you need.

- Basic Terminal Stuff
- Linux Permission System
- Vim - "Quick" Guide

And finally always remember: You can 'google' everything you need to know very easily. So don't get frustrated, just keep digging.

2. Foundations

In this chapter topics that are tackled during the lecture are introduced in a basic fashion. This includes agents, sensor, actuators, communication, etc. . .

2.1. Agents

Although not every agent is autonomous, our focus in this lecture is to design autonomous agents. Autonomous means, that the agent is not controlled from some external entity. It only perceives its environment and acts in order to change it. Why it tries to change its environment depends on the agent's properties. Agents can be reactive or proactive, they can communicate with other agents, remember the history of the environment, and predict the future of the environment according to some model. As a designer of the agents, you need to decide what kind of agents you need to achieve your goal. The famous book *Artificial Intelligence – A Modern Approach*, written by Russel and Norvic, elaborates properties of agents and environment in chapter two. The book is available in our library. **It is recommended to read chapter two for your exam preparations.**

2.1.1. Robot Control Architectures

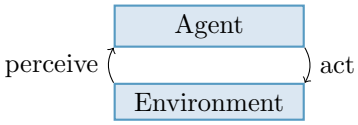


Figure 2.1.: Simple Control Loop

Usually it is believed that the agent is interacting with its environment –whatever that environment is. This relation is often depicted as an agent-environment cycle (see Figure 2.1). As mentioned before, the details of an agent architecture do vary from use case to use case.

In Figure 2.2 an overview of the details of a state-of-the-art robot control system is shown. You should consider a robot as a physical agent, i.d., an agent with a physical representation. The Data Layers on the left hand side represent the utilised algorithms to gain information and knowledge from raw sensor data. During the data processing, the data is continuously filtered, condensed, and abstracted until symbols and relations can be integrated into a symbolic knowledge representation. This knowledge is required by the algorithms on the right hand side of Figure 2.2. The

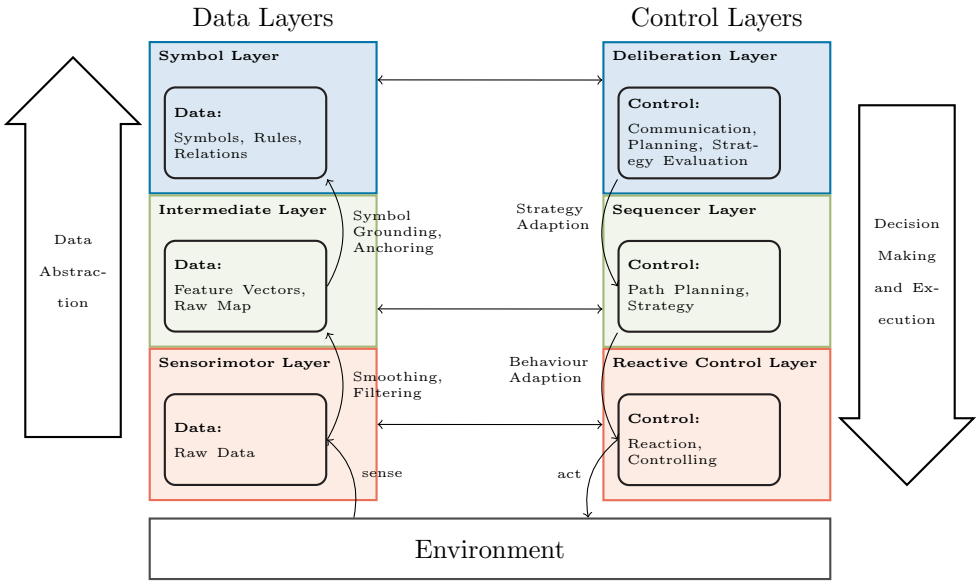


Figure 2.2.: Control Loop

Deliberation Layer, including algorithms for planning, communication, and strategy evaluation, computes its decisions based on the knowledge of the symbolic knowledge representation. The results are processed by the Sequencer Layer and transformed into commands for controlling actuators of the robot.

The data abstraction and decision making cycle is not perfectly followed in all applications. Often the abstraction of data up to the level of symbols and relations is not necessary. Moreover, it isn't clever to make everything a decision on the top most available layers. Think of a robot control architecture as of a human brain. A lot of things in a human body are handled without having the brain actively thinking about it, e.g., breathing, heart beat, reflexes on your knee, sneezing, blinking, etc. For a robot the corresponding things could be obstacle avoidance, motor controlling, avoiding rough terrain such as stairs, etc. Things like that should also in a robot control architecture be handled on lower levels and not bothering algorithms for planning or communication.

However, don't take this easy. It can be quit difficult or even impossible to assign a certain task to a certain level. Obstacle avoidance, for example, is sometimes part of all three layers: While driving along a path, the robot in a office environment avoids humans or other robots by directly reacting to its 2D distance scans from its laser scanner (lower layers). The path itself is planned in order to avoid already known walls and finding the shortest path (intermediate layers). Planning this path is influenced by some knowledge about the environment, e.g., the robot knows that there is currently a meeting going on in some room and as the robots motion is relative noisy, it knows that it should avoid meetings like a virtual obstacle, in order to not

disturb the meeting (top layers).

If you are in the position to decide on which layer something should be implemented, it sometimes helps to think about the following. The effort for abstracting data rises with each necessary processing step, therefore creating symbolic knowledge is often very costly. On the one hand, this argues for putting everything as on the lowest levels. On the other hand, the data on the lowest levels is often very large in terms of memory. You often have to ponder, whether it is more efficient to abstract the data and thereby reduce it to the smallest possible size or whether it is more efficient to directly operate on millions of raw camera pixel values without explicitly representing objects in the environment. Sometimes technical limitations also force you to a certain direction. If your robots, for example, need to communicate about objects it is probably impossible to send whole images of the objects to each other due to bandwidth limitations. Instead, the objects have to be abstracted to symbols or coordinates in order to match the bandwidth restrictions.

2.2. Sensors

Sensors are the only option for an agent to perceive its environment. This also means, that everything that helps an agent to perceive its environment can be considered as a sensor. Nevertheless, sensors are also used to measure internal values, not belonging to the environment. Before we are going to deep, let us start the sensor topic a little bit more general.

2.2.1. Sensor Categories

There are two sensor properties that help to categorise the plethora of sensors available nowadays: Active vs. passive sensors and external vs. internal sensors. Whether a sensor is active or passive determines the way it measures some value.

active: An active sensor emits something, e.g., energy, for measuring what ever it should measure. Typical examples are laser scanners or radars that emit electromagnetic waves of different wave length.

passive: Passive sensors directly measure energy from the environment. A camera for example perceives sunlight that is reflected from a surface, furthermore a compass measures the terrestrial magnetic field.

The difference between an internal or external sensor are the type of values the sensor measures, not how it measures it.

internal: Internal sensors measure values that are part of the system itself. Therefore it depends on the borders between the system and its environment. In case of a robot an internal sensor value could be its battery levels.

external: External sensors measure values from the outside of the system, i.d., values from its environment. Typical values are distances, light, and sound.

2.2.2. Sensor Properties

Subsection 2.2.1 is for categorising different types of sensors. If you need to choose a sensor for your application these categories are only a first orientation. In this subsection you will get a short overview of general things you have to consider in your decision for taking the right sensor for your application.

Size Does the sensor fit into your robot?

Weight Can the sensor be lifted or carried by your robot (especially relevant for drones)?

Energy Consumption Does the energy consumption of the sensor significantly reduce the runtime of your robot?

Sensor Range Sensors often have minimum and maximum values they can measure, e.g., distance starting from 0.4m up to 100m. Does the range fit your application?

Field of View In which direction can the sensor measure? Is it big enough? Can it be integrated into your robot without limiting the field of view?

Operating Conditions Some sensors are sensitive to temperature, vibration, light and so on. Does your application match their operating conditions?

Resolution What is the granularity of the measured values (directly influences properties from Subsection 2.2.3)?

Value Transformation Is the value received from the sensor in the unit you need? If not, do you know the transformation and how long does the transformation take?

Linearity Does a linear change in the value to be measured induce a linear change in the value received from the sensor? If not, what is the relation (see Value Transformation)?

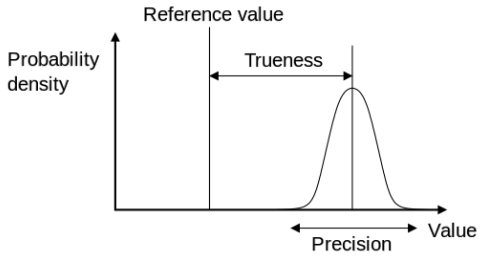
2.2.3. Sensor Value Properties

The sensor value properties of possible sensors are very important for making the right choice, too. In this subsection we collect these properties, explain their meaning, and explain why they are important.

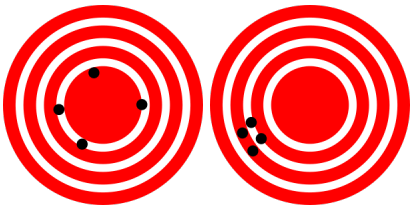
The sketched definitions given below follow the ISO Standard 5725.

Precision Precision is about the reproducability or repeatability of measurements. The sensor has a high precision, if it always measures the same values under same conditions. Note that this measured values can be completely wrong!

Accuracy Accuracy is the average difference between the reference value (correct value) and the measured values. Please read this carefully, as it only slightly differs from trueness.



(a) Trueness vs. Precision



(b) Low accuracy, poor precision, good trueness

(c) Low accuracy, good precision, poor trueness

Figure 2.3.: Examples for Precision, Accuracy and Trueness (Source: Wikipedia)

Trueness Trueness is the difference between the reference value (correct value) and the average of the measured values.

In Figure 2.3 several helpful pictures for understanding the above definitions are given. Especially the targets make the difference between accuracy and trueness comprehensible. In order to get a feeling for the relevance of precision, accuracy and trueness in practice let us exercise the utilisation of sensors with different properties. Imaging your task is to get to know what the correct value for a certain distance is.

The easiest case is when you have a highly accurate sensor, because this means that each measurement is relatively close to the correct value. It also means that the precision of the sensor must be relatively high, too. Here is why: If all measurements are close to a single static value (being accurate), the measurements cannot vary very much at the same time (low precision).

Now let us consider to use a sensor with low accuracy, but with high precision (like in Subfigure 2.3c). The measured values are relatively wrong, but the are reproduceable. Therefore, the sensor will measure under the same conditions the same wrong values. That gives you the chance to correct this relative constant offset or error by calibrating your sensor. For calibration you need a series of measurements for that you know the correct value. As you have a precise sensor, the difference between the measured values and the correct value should be more or less the same. In a naive calibration you could simply take this difference and correct the sensor's measurements by this difference at runtime. The problem is that precision means reproducability under the **same conditions**. These conditions often include the following parameters: the correct value, the state of the sensor (temperature, power supply, operation modes), and properties relevant for the measurement process (temperature, surfaces, light conditions). Therefore, calibration means to find the correct offset for all possible parameter combinations that influence the operating conditions of the sensor. Maybe this sounds like an almost unmanageable task, but here are some arguments against it:

- The sensor already can calibrate itself with a special operation mode that you only have to trigger from time to time.

- The most parameters are irrelevant for the measuring.
- Have a look at your application scenario, hopefully a lot of parameters are constant in you application (artificial lights, indoor temperatures).

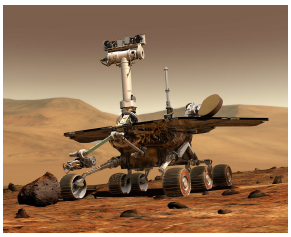
As a last relevant case, let us consider a sensor with low accuracy, but with high trueness (like in Subfigure 2.3b). Here the average of the measured values is relatively close the correct value. Therefore, you only need to take the average of some measurements, in order to know the correct value. The only question is: "How much measurements are necessary?" Simply make some experiments where you observe the change of the average relative to the increasing number of measurements. When the change of the average is low enough for your application you know the minimal number of measurements necessary to calculate a good average for determining the correct value. Think about your application then: Is it possible to make this number of measurements before the correct value changes, e.g., distance to moving obstacles. Obviously, you need more measurements if the standard deviation of the measurements is high, and less if its low. Note that a sensor with high trueness and a small standard deviation is actually an accurate sensor.

To summarise: Accurate sensors are fine, precise sensors need to be calibrated, and sensors with high trueness demand for calculating the average of some measurements.

2.3. Actuators

Actuators are the counterpart to sensors. While an agent perceives its environment with the help of sensors, it changes its environment with the help of actuators. Let's consider some examples and try to derive general categories from it (see Figure 2.4.

(a) Robot Arm (Source: Wikipedia)



(b) Robot Locomotion (Source: Wikipedia)

(c) Hard Disk Drive (Source: Wikipedia)

Figure 2.4.: Examples for Actuators of Agents

Subfigure 2.4a shows a robto arm in a laboratory environment. This arm can manipulate the environment, e.g., by executing pick-and-place tasks, but it also represents the whole robot. In the context of this lecture we barely consider this arm as a physical agent for the following reasons: It is not autonomous, it has no external sensors for perceiving its environment, and it probably does the same movements over and over again. However, imagine that this arm is mounted on a mobile platform and

controlled by an autonomous control architecture (see Subsection 2.1.1) that is also able to perceive its environment through dedicated sensors. Let us collect a list of rather blunt and naiv statements about this actuator.

- The arm is made for manipulating the environment, so it is not just an actuator, it is a manipulator.
- It has a certain flexibility which is often denoted as degrees-of-freedom (DOF).
- Reaching out from its mobile platform, its range is limited.
- Depending on its motors and energy sources, it can execute its movement with a limited force / speed.
- Depending on the mechanism at the end of the arm, it can only manipulate objects that suit this mechanism (denoted as Endeffector).

2.4. Communication

3. External Software and Developing Tools

This chapter is intended to give a short introduction to external software packages or frameworks, we utilise for the Carpe Noctem Cassel software framework.

3.1. GIT and GitHub

GIT is one of the most advanced version control systems currently available. Nevertheless, during our daily work we only use 20% of its functionality. So for starters try to learn the stuff you need and ignore its advanced features like *rebase* and *cherry pick*.

The best reference and documentation about GIT can be found at `http://www.git-scm.com/docs`

Most of our software is published open source under the MIT License at our GitHub Repositories. Therefore, it is also interesting to read about the features of GitHub, like SSH-Key based authorization, groups, organisations, and MarkDown.

The README.md files in our repositories are written in MarkDown, because GitHub parses these MarkDown files and auto-generates an HTML documentation from it.

A list of GIT commands, that should be enough for the start, can be found on git-tower.com.

3.2. Robot Operating System (ROS)

ROS, as we use it, is a simple inter process communication middleware. Before you ask, yes it is not intended to be used for inter machine/robot communication. Therefore, we have developed a simple ROSUdpProxy, for our purposes.

Tutorials for ROS can be found here: `http://wiki.ros.org/ROS/Tutorials`

After following this tutorial you should be able to explain a bunch of things:

Topics: How do they work?

Nodes: What is a ROS node?

roscore: What is its job?

package.xml: build_depend, run_depend, licences, ...

CMakeLists.txt: What are the critical ROS specific macros and how do they work? (not very well explained in the tutorial)

Console Commands: rosrun, rospack, roscd, rosls, roslaunch, rostopic, rosnode

catkin_make: How to compile a workspace/a package? DON'T USE 'catkin_make'! USE 'catkin build' INSTEAD!

ROS-Workspace: What is its structure and why is it structured that way?

ROS-Services: How are they defined, compiled/generated, and how do they work?

ROS-Messages: How are they defined and compiled/generated?

roscpp API: How to create a publisher and a subscriber?

3.3. Build Chain

We utilise *catkin* from the ROS Universe as our build chain. Catkin is basically a workspace-oriented extension of CMake. Therefore, it heavily relies on CMake and in order to understand catkin it is recommended to understand CMake first.

CMake is open source and developed by KitWare. Basically CMake autogenerates Makefiles out of CMakeLists.txt files located in each software module. Therefore, our build chain can really be considered as a chain ☺:

Catkin $\xrightarrow{manages}$ CMake $\xrightarrow{auto-generates}$ Makefiles $\xrightarrow{commands}$ GCC $\xrightarrow{to\ compile}$ executables and libraries.

4. Process Manager and Remote Control GUI

The process manager is an executable for managing the processes running on a PC. Compared with the former C# framework, it replaces the process manager Care. The process manager can run on the robot for testing with real a robot, or on your local PC for testing with a simulator (add -sim as parameter). With the help of the *ROBOT* environment variable, it is possible to manage processes for multiple robots on a single PC, which is useful for multi-robot testing on a single PC. The name of the ROS-Package of the process manager is *process_manager* and can be found by using `roscd process_manager`.

The remote control GUI for the process manager is an RQT plugin (see `http://wiki.ros.org/rqt` for details). In the former C# framework this GUI was highly integrated into the LebtClient and mixed with robocup msl specific GUI elements. The idea of the new GUI is, to make it useable in other domains, too. The name of the ROS-Package of the remote control GUI is *pm_control* and can be found by using `roscd pm_control`.

4.1. Quickstart Guide

In order to bring up a single process manager and its remote control GUI, on the same PC, execute the following commands in the given order:

- `rosrun process_manager process_manager`
- `rosrun pm_control pm_control`

The first command starts the process manager, which will automatically start a roscore, if none is running. Please add the `-sim` parameter to the process manager, if you want to use it locally managing multiple robots (e.g. for simulation). The second command start the remote control GUI, which should display the received information of the process manager (see Figure 4.1).

If the process manager and the remote control GUI should run on different PCs, which are in the same network, you need to make sure, that on both machines a roscore and a UDP proxy is running. Therefore, you need to execute the following commands on the machine, where the process manager should run:

- `rosrun msl_udp_proxy msl_udp_proxy`
- `rosrun process_manager process_manager`

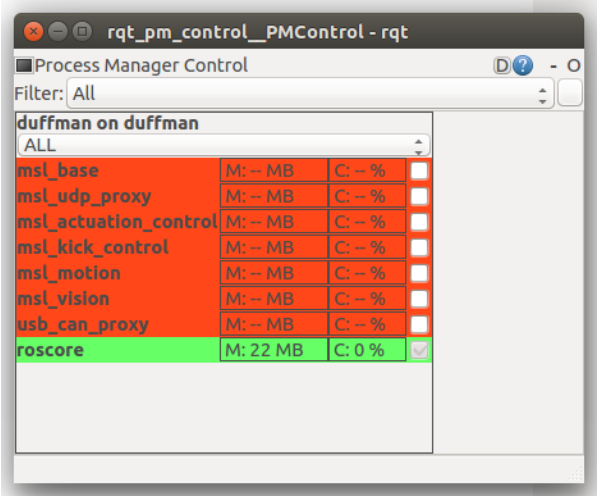


Figure 4.1.: The Remote Controle GUI

The msl-udp-proxy is an autogenerated proxy, which broadcasts MSL specific ROS messages from the local machine to a multicast address on the network. You can also use another UDP proxy (e.g. ttb-udp-proxy, or bbb-udp-proxy), as along as it forwards the *ProcessCommand* and *ProcessStats* ROS messages of the process manager.

On the machine, where the remote control GUI should run, execute:

- `roscore`
- `rosrun msl_udp_proxy msl_udp_proxy`
- `rosrun pm_control pm_control`

You need to start the roscore manually, as the remote control GUI does not start a roscore automatically.

4.2. General Information on the Process Manager

PROC Filesystem

The information about the managed processes are collected from the */proc* filesystem. This is a specific path, which is available on each linux-kernel-based operating system. Inside the /proc folder, each running process has a seperate subfolder, named after its process id (PID). Please note, that this PID is determined by the kernel and has nothing to do with the process ids of the ProcessManaging.conf file. The process manager uses the information available in the processes *cmdline* and *stat* file (e.g. /proc/2341/cmdline). These files are continuesly updated by the kernel. For

further information about the proc filesystem, consider chapter 7 of the book *Advanced Linux Programming* (available at: http://www.advancedlinuxprogramming.com/alp-folder/).

The interesting thing about the proc filessystem approach is, that it enables the process manager to attach to processes it did not start in the first place! If you had some processes already up and running, just start the process manager and its remote control GUI to see the statistics about those processes.

Another fact about the process manager is, that it does not do anything to the processes, when it is closed. The launched processes are independent of the process manager and continue their execution without it. You accidentally stopped the process manager? No problem just restart the process manager and it will continue to monitor the started processes.

Communication

The process manager defines two ROS messages: *ProcessCommand* and *ProcessStats*. The ProcessCommand message is used to let him start and stop processes. The ProcessStats message is used by the process manager to report the statistics about its managed processes. The process manager is configured to scan the proc filesystem every two seconds, so that the ProcessStats is send roughly every two seconds.

The process manager subscribes on the ROS topic */process_manager/ProcessCommand* and publishes its commands to */process_manager/ProcessStats*.

Allowed Number of Processes

The process manager is written in a way, that it only allows to run a certain process only one time per robot. So if you want to run a process twice on the same machine, you need to modify the *ROBOT* environment variable for at least one of the two process instances. E.g.: `ROBOT=nase rosrun msl_base msl_base -m WM16` and `ROBOT=myo rosrun msl_base msl_base -m WM16`.

Another feature of the limitation on the number of allowed processes is, that if you did start too many processes, e.g. two or more image processing processes on one robot, you can simply start the process manager and it will clean up the mess. It will kill all but one process of each kind and start reporting statistics about the left processes.

4.3. General Information on the Remote Control GUI

Process GUI

The remote control GUI has a single process GUI element for each process (see Figure 4.2). From left to right it shows: the process name, its memory usage in MB, its cpu usage in percent (100% means one core), its check box for start and stop. The background color of the process GUI is either red (not running), green (running), gray (unknown). Note that the check box does not determine, whether a process is

running or not, it is just for starting and stopping a process. Start-Commands for running processes are ignored by the process manager. The check box is disabled for the roscore, because stopping it would cut the communication to the process manager. Check boxes of other processes are disabled too, if they are running with parameters that differ from the currently selected bundle (see 4.3).

Figure 4.2.: GUI Elements for one Process

If you have a running process (background is green), you can hover over the process GUI element, in order to show its ToolTip. The ToolTip shows the command, which was used to start the process. This way, you don't have to check the ProcessManaging.conf file, in order to know what parameters are used in a certain bundle.

Communication

The GUI elements are created at runtime, when a corresponding message from a process manager arrives. So if you don't receive any ProcessStats messages, your remote control GUI won't show anything. Furthermore, the GUI elements are deleted, if you don't receive messages for certain amount of time (roughly 3 seconds). The remote control GUI subscribes on the ROS topic */process_manager/ProcessStats* and publishes its commands to */process_manager/ProcessCommand*.

Bundle Selection

Selecting a bundle in the drop down box of the GUI, means that the listed processes will be started with the corresponding parameter set, as specified in the ProcessManaging.conf file (see Section 4.4). It also means, that you cannot stop a process which runs with a different parameter set, then specified in the selected bundle. In such cases, the check box of the process is disabled (grayed out). Nevertheless, there are two default bundles: ALL and RUNNING. If you select one of these two bundles, you can interact with all processes.

ALL lists all processes configured in the ProcessManaging.conf file. This is useful, if you want to start a set of processes, which is not specified as an explicit bundle. RUNNING lists all processes of the ProcessManaging.conf file, which are currently up and running. This is useful, for determining, whether there are unwanted processes running, which are not part of the bundle that you would like to use.

The *roscore* process is specially handled. If the roscore is stopped, the process manager cannot receive commands anymore. Therefore, it is not allowed to stop the roscore process within the remote control GUI.

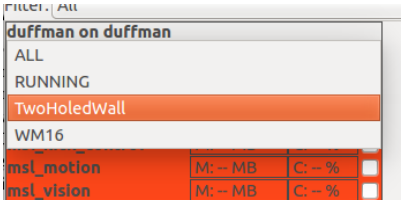


Figure 4.3.: Bundle Selection by Drop Down Box

4.4. Configuration

In order to configure, which processes should be managed by the process manager, you need to edit the *ProcessManaging.conf* file. The file is usually located in the *etc* folder, determined by the *DOMAIN_CONFIG_FOLDER* environment variable. Inside the ProcessManaging.conf file several comments explain the config values itself. Nevertheless, lets explain the config for the msl_base process in detail:

```
[Base]
  id = 7
  execName = msl_base
  rosPackage = msl_base
  mode = none
  [paramSets]
    1 = -m, TwoHoledWallMaster
    2 = -m, ActuatorTestMaster
    3 = -m, WM16
    4 = -m, TestApproachBallMaster
    5 = -m, TestCheckGoalKicki
    6 = -m, WM16, -sim
  [!paramSets]
[!Base]
```

The **name** (Base) of the outmost section denotes the GUI-String representing the executable. The executable denoted by **execName** (msl_base) need to be located in the *PATH* environment variable, or should be found by executing
`catkin_find --libexec msl_base`
in order to work with the process manager. Here `msl_base` is the **rosPackage** name.

The **id** of the process must be unique in the ProcessManaging.conf file and is used to refer to this process in the bundles section (explained later) and in the messages send to and received from the process manager.

The **mode** decides how the process manager handles crashes of the process and some other things. At the moment, there are 3 different modes.

none Basically does nothing. It does not autostart the process, when the process manager is started with the *-autostart* parameter. It does not restart the process, when it did crash.

keepAlive Processes, configured with this mode, will be restarted by the process manager, when they crashed.

autostart This mode makes the process manager start this process, when the process manager is started with the *-autostart* parameter and restarts it after chrashes.

In the **paramSets** sections, it is possible to specify different sets of parameters which can be used to start the process. Each parameter set follow a simple key-value-pair convention, where the key must be a (for this process unique) positive integer, greater than 0. The value is a comma (,) seperated list of parameters. Please note, that the parameter `-m TwoHoledWallMaster` is actually two parameters: `-m` and `TwoHoledWallMaster`. The parameter set with the lowest key is considered to be the default parameter set, which means that this parameter set is used, if not specified otherwise (e.g. by choosing a bundle).

The **bundles** section of the ProcessManaging.conf file allows to specify a set of processes with a specific parameter set for each of it. Here is a small example:

```
[Bundles]
  [WM16]
    processList        = 0,1,2,3,4,5,6,7
    processParamsList  = 0,0,1,0,0,0,0,3
  [!WM16]

  [TwoHoledWall]
    processList        = 0,1,2,3,4,5,6,7
    processParamsList  = 0,0,1,0,0,0,0,1
  [!TwoHoledWall]
[!Bundles]
```

In this example, two bundles are specified: WM16 and TwoHoledWall. Each bundle consists of two key-value-pairs: processList and processParamsList. The processList specifies the list of processes, by listing the process ids. The processParamsList specifies the parameter sets for each process. It is important that the order of the processParamsList has to be the same as in the processList. For example: In the WM16 bundle the process with the id 7 (Base) is started with the parameter set id 3 (`-m, WM16`). In the TwoHoledWall bundle, it is started with the parameter set id 1 (`-m, TwoHoledWallMaster`). The parameter set id 0 is a special value, with the meaning, that there is no parameter set specified for this process, because it has no parameters. Another special value, which is only used in the messages send to and received from the process manager, is -1. It says that the parameters of a process are unkown, e.g., the msl_base is running with an unknown master plan (`-m FancyNewTestPlanMaster`).

4.5. Future Work

- Make the retry timeout for starting processes a parameter in the ProcessManaging.conf file.
- Make it possible to add the GUI for another virtual robot, in order to command a process manager to start processes for another robot. This is necessary for local testing with multiple robots.
- Document the implemented feature of starting launch scripts here. (For details see the ProcessManaging.conf file).

5. ALICA Client

The ALICA Client is a simple GUI for visualising AlicaEngineInfo messages. Currently it can be started for visualising the messages of a single engine by calling `rosrun alica_client alica_client`. If you have several robots running, the GUI will flicker between their incoming messages. For visualising the messages of multiple robots you should use the Robot Control GUI (see 6). It integrates the same GUI component multiple times.

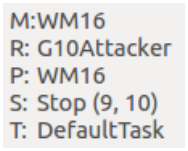


Figure 5.1.: The ALICA Client GUI

In Figure 5.1 you can see the ALICA Client GUI. **M:** denotes the currently executed master plan (WM16). **R:** denotes the current role of the robot (G10Attacker). **P:** is the deepest plan in the alica plan hierarchy, the robot is currently executing. As it is WM16, it means, that it is inside the master plan, but not deeper. Inside this plan, the robot is currently one of two robots inside the Stop state (**S:**). In this case, it is either the robot with id 9 or 10. The name of the task associated with the active state is the DefaultTask (**T:**).

This tool is definitely usefull for debugging ALICA plans, but please note, that the AlicaEngineInfo messages are only send roughly every 100ms, but the plan state of an ALICA Engine typically changes much faster, than that. Therefore, you want recognize agents racing through the plans' state machines. Making this visible needs a litte bit more sophisticated GUI, which should be able to read logs of the ALICA engine itself. Please see the project description for some details, about this possible bachelor project.

6. Robot Control

TODO

7. TurtleBot Hardware

8. TurtleBot Software

The TurtleBot Software consists of several launch files which start the processes and load the needed xml descriptions.

8.1. Quick Start Guide

In order to use a laptop to control a TurtleBot several environment variables have to set in the .bashrc, which are show in the following List 8.1 for Robot Leonardo.

- `export ROBOT=leonardo`
- `export TURTLEBOT_3D_SENSOR=kinect`
- `export TURTLEBOT_MAP_FILE="$DOMAIN_CONFIG_FOLDER/map_final.yaml"`

The export `$DOMAIN_CONFIG_FOLDER` should have been set already by the setup script (ttb-setup.sh) found in cnc-turtlebots/configuration/setupscripts. Otherwise it has to be set to cnc-turtlebots/etc. Once these preparations are done the laptop can be used to control it should be placed with closed lid on the layer below the laser scanner. After this is done the USB-hub can be connected with the laptop and the TurtleBot is ready. The setup of the Turtlebot is shown in Figure 8.1. To prepare the laptop/pc

Figure 8.1.: TurtleBot setup.

to communicate with the TurtleBot these processes are started in three terminals are need on the local laptop/pc in which the following commands have to be executed:

- `roscore`

- `rosrun ttb_udp_proxy ttb_udp_proxy`
- `roslaunch turtlebot_bringup rviz_robot.launch`

`rviz_robot.launch` is used to start a GUI that is used to locate the TurtleBot and to create NavGoals which tell the TurtleBot where to drive to. The GUI with buttons for leonardo is depicted in Figure 8.2. Buttons for additional robots and functions can be added with the plus symbol. the checkbox next to leonardo on the left side of the screenshot has to be clicked to show the model of the robot. These steps have

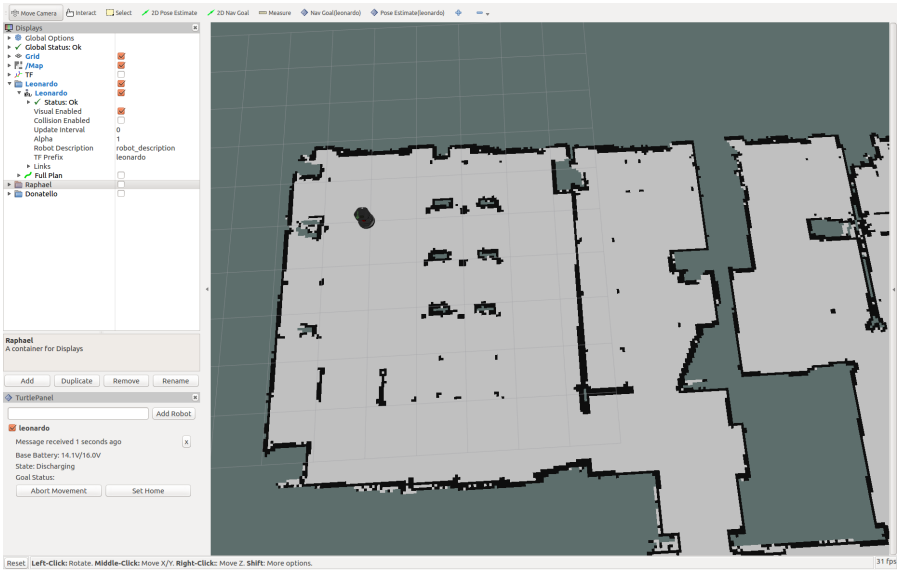


Figure 8.2.: Rviz screen.

to be done before starting to processes on the control laptop. To start the TurtleBot four terminals with a ssh connection to the laptop on the TurtleBot are needed, which can be done either with `ssh tb@robotname.local` or `ssh tb@robotIP`. Once the ssh connections have been established the following commands have to be executed on the terminals:

- `roscore`
- `rosrun ttb_udp_proxy ttb_udp_proxy`
- `roslaunch turtlebot_bringup ninjaturtles.launch`
- `roslaunch turtlebot_bringup amcl.launch`

`ninjaturtles.launch` starts all parts of the TurtleBot, `amcl.launch` enables the motion and `ttb_udp_proxy` is used to communicate with other laptops/pcs. As soon as all processes have been started the TurtleBot's location has to to be set on rviz

regarding the position in the real world and its orientation. This is done by clicking on the pose estimation button with the specific robot name. Afterwards the position and orientation of the TurtleBot can be determined by clicking and holding the left mouse button on the position the TurtleBot should be placed and dragging the cursor in the direction it should face. Once the left mouse button is released position and orientation are determined. To send the TurtleBot to a position the Nav Goal button is used. After clicking this button a position on the map can be marked as a navigation goal in the same way the position of the TurtleBot was determined.

8.2. Drive Topic and Message

To steer the TurtleBot without using amcl you need to publish a message under a specific topic. A message is published by using `rostopic pub TOPIC MESSAGE`. The topic used to steer the TurtleBot is named `/cmd_vel_mux/input/teleop` the message is named `geometry_msgs/Twist`. After the command is entered the message contents can be show by pressing tab twice and can be edited before sendin the message with enter. Linear x is used to move the robot forward with a max value of 1.5. Angular z is used to rotate the robot. The max value is 14.

9. TurtleBot Gazebo Simulation

9.1. Quickstart Simulation

With this section, you should be able to start one of our TurtleBot Gazebo simulation environments.

9.1.1. Setting up your .bashrc file

You should have the lines from Figure A.1 in Appendix A.1 inserted and adapted in your `.bashrc` file. The `.bashrc` file is located in your home folder.

Don't forget to close the terminal after you edited the `.bashrc` file. It will only be reloaded after you reopen the terminal or you type `. .bashrc` into each of the currently open consoles.

9.1.2. Starting Gazebo Simulation

The following command starts Gazebo with the Turtlebots in the map of our department and provides a ROS map_server node.

```
roslaunch turtlebot_bringup simulation.launch
```

Another possible environment is `disaster.launch`.

Next you can start the navigation and localization stack for your turtlebots. The following command starts the Adaptive Monte Carlo Localization (AMCL) and move_base including its local and global path planners, costmaps, and recovery behaviours.

```
roslaunch turtlebot_bringup navigation_gazebo.launch
```

For debugging purposes ROS provides us with the Robot Vizualization program (rviz). We have an already configured launch file for rviz:

```
roslaunch rviz_sim.launch
```

9.2. Simulation Environment

The different simulation environments of Gazebo are denoted as worlds. The worlds we developed are located in the *world* folder of the *turtlebot_bringup* ROS package. Currently there exist three worlds: *distributed_system_simple.world*, *distributed_systems_complex.world*, *distributed_systems_disaster.world*. The simple world only includes the walls of the floor of the Distributed Systems department. In the complex version of it, furniture and several items are added for a more realistic simulation environment. The disaster world is an evaluation environment for the NICER project and not of further interest for this lecture.

The world files are formatted in a custom XML format, denoted as SPF. The format is relatively well documented on its project homepage. Please note the different available versions of the format. In the world file the world's properties are specified. These properties include physic properties, lighting, atmosphere, wind, etc. Furthermore different models can be inserted into the world. Therefore, other files can be referenced in the world file. The following lines, e.g., place a model instance of a small table at the given position. The instance can later we referenced by its given name.

```
<model name='r1411_small_table'>
  <pose frame=''>0.77 4.46 0 0 0 0</pose>
  <include>
    <uri>model://small_table</uri>
  </include>
</model>
```

The line `<uri>model://small_table</uri>` is a where the magic happens. The protocol of this URI (`model://`) states that the model needs to be found in a collection of locations given by the `GAZEBO_MODEL_PATH` environment variable (see Section 9.1.1). Taking a look at the *models* folder in the *turtlebot_bringup* ROS package, you will find one folder per model. Each folder includes a *.config* file and maybe several version of *.spf* files for the model. The *.config* file includes meta data about the model and references the different existing *.spf* version files. In the *.spf* files the model itself is specified, according to the SPF Model Specification.

There exist several neat features in the SPF format. Especially, it is possible to include Ruby code, in order to automatically generate parts of models or worlds. Nevertheless a robot itself is seldom specified directly in the SPF format. Instead it is spawned at run-time into the simulation environment by an extra spawning process denoted as *spawn_model* that is part of the *gazebo_ros* package. The arguments for the spawning process is a description of the robot model given in the Unified Robot Description format (URDF). Documentation for this format are given by the corresponding URDF package pages. URDF is actually deprecated an should be replaced by SDF, but for legacy reasons it still exists (see ROS Answer Post). If it should happen to you, that you are in the need of changing something at a robot model that is specified in URDF, the Gazebo Tutorial about URDF could become handy, too.

9.3. TurtleBot Topics

At this point it is expected that you have profound understanding of ROS topics and the corresponding commandline tools. The purpose of this section is to help investigating or debugging typical topic errors related with our TurtleBot simulation environment. Furthermore, the most interesting topics are listed and explained.

9.3.1. Investigating Topics

The first approach is always to see which topics are available in your running system. Therefore, type `rostopic list` into your terminal. The output looks like this:

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/leonardo/amcl/parameter_descriptions
/leonardo/amcl/parameter_updates
/leonardo/amcl_pose
/leonardo/camera/depth/camera_info
/leonardo/camera/depth/image_raw
/leonardo/camera/depth/points
/leonardo/camera/parameter_descriptions
/leonardo/camera/parameter_updates
/leonardo/camera/rgb/camera_info
/leonardo/camera/rgb/image_raw
...
```

Much more information can be gained with the verbosity option of the same command: `rostopic list -v`. Now the output is distinguished in published topics and subscribed topics, including the corresponding numbers of publishers and subscribers currently registered at the rosmaster process. As you should already know, the rosmaster process is the central ROS communication registry. It is spawned by the roscore process which in turn is spawned by any roslaunch process in case roscore is not running already. Here is an example for the verbatim output:

```
Published topics:
* /map_metadata [nav_msgs/MapMetaData] 1 publisher
* /leonardo/move_base/local_costmap/obstacle_layer/parameter_updates [dynamic_reconfigure/Config] 1 publisher
* /leonardo/move_base/local_costmap/footprint [geometry_msgs/PolygonStamped] 1 publisher
* /leonardo/odom [nav_msgs/Odometry] 1 publisher
...

Subscribed topics:
* /leonardo/cmd_vel_mux/input/teleop [geometry_msgs/Twist] 1 subscriber
* /gazebo/set_link_state [gazebo_msgs/LinkState] 1 subscriber
* /tf [tf2_msgs/TFMessage] 2 subscribers
* /leonardo/move_base/global_costmap/footprint [geometry_msgs/PolygonStamped] 1 subscriber
* /leonardo/scan_hokuyo [sensor_msgs/LaserScan] 2 subscribers
...
```

As you may have noticed, the message type of each topic is given in square braces. Gaining this information that way, is sometimes much faster than surfing to the corresponding ROS documentation website.

One typical error, during programming your own ROS node, is that the topics of your node are not prefixed with the name of your robot at hand. When your node is working and is integrated into one of our launch files above the topics will be prefixed automatically, but if you test your node "by hand" you have to take care by yourself, that the topics include the robots name (if necessary). The prefixes are necessary, for example, in order to have one path planning topic for each spawned robot. Otherwise all robots would receive the same path planning goals and drive to them.

9.3.2. Interesting Topics

Interacting with the move_base node is a little bit complex, as it uses an ROS Action Server. For simply posting a path planning goal to move_base you can use the topic

`/leonardo/move_base_simple/goal`. The move_base node will drive to that goal without giving you feedback about it.

AMCL is not able to localize itself globally on the map. Therefore, the AMCL node on each robot needs to receive its initial position from extern. The topic for this is `/leonardo/initialpose`. Note that it is easy to send this information via RVIZ and its *initial pose estimate* command.

9.4. Turtlebot Sensor Plugin

To simulate various scenarios using the TurtleBot there was a need for a more specific and configurable type of sensor. We developed a sensor plugin for gazebo to accomplish that. It allows the user to configure detection angles, range and the type of object in a configuration file. You may think of it, as a laser scanner sensor, that can reach through walls and perfectly detects what ever it was specified for.

9.4.1. Configuration

The configuration format uses a similar syntax as the configuration for the Process-Manager mentioned in Section 4.

The file itself is located at the configuration path defined by the `DOMAIN_CONFIG_FOLDER` environement variable (see Section 9.1.1) and has the filename *LogicalCamera.conf*.

Let's take a look at the following example configuration:

```
[LogicalCamera]
        [Victim]
                # meter
                range = 3
                type = victim
                # Hz
                publishingRateHz = 1

                # angles have to be set from 180 to -180 degree
                [DetectAngles]
                        [Front]
                                startAngle = -135
                                endAngle = 135
                        [!Front]
                [!DetectAngles]
        [!Victim]
        [Fire]
                # meter
                range = 5
                type = fire
                # Hz
```

```
                publishingRateHz = 10

                # angles have to be set from 180 to -180 degree
                [DetectAngles]
                        [Front]
                                startAngle = -180
                                endAngle = 180
                        [!Front]
                [!DetectAngles]
        [!Fire]
[!LogicalCamera]
```

The first Block `[LogicalCamera]` is the uppermost block and has to be followed by `[!LogicalCamera]` at the end.

In this example we want to detect two types of objects. The first one is the presence of victims and the second one is fire. Each distinct type has to be defined as a block, similar to the uppermost LogicalCamera Block. Hence we have `[Victim] ... [!Victim]` and `[Fire] ... [!Fire]` here.

In these type blocks, we now have to define the properties of the sensor for that given type of object.

Here, a victim can be detected in a **range** of 3 meters, as given to the range parameter.

The **type** parameter is used by the plugin to determine which object belongs to which type. The plugin filters these types depending on the objects name in gazebo. So an object called *victim_box* is going to be detected as a victim type object.

Using the parameter **publishingRateHz** as the name suggests you can set the rate in which an object may be detected. A victim will be only detected once a second, while a fire can be detected up to ten times a second.

Next, the block **DetectAngles** allows you to specify multiple sections in which the robot can detect the object.

Angles have to be in a range from -180 to 180.

Figure 9.1 shows how the angles have to be speciefied.

9.4.2. Sensor message

Every time an object has been detected according to the configuration, a ROS message gets sent.

The name of the ros topic is `logical\_camera` and is namespaced with the robot name, as configured in the *robot.simulation* file. So for a robot named leonardo, the message will get published on the topic /leonardo/logical_camera.

The message itself has the following fields:

`string modelName`

Name of the detected model/object in gazebo

`geometry\_msgs/Pose2D pose`

Posoition of the object on the x,y plane(map seen from top)

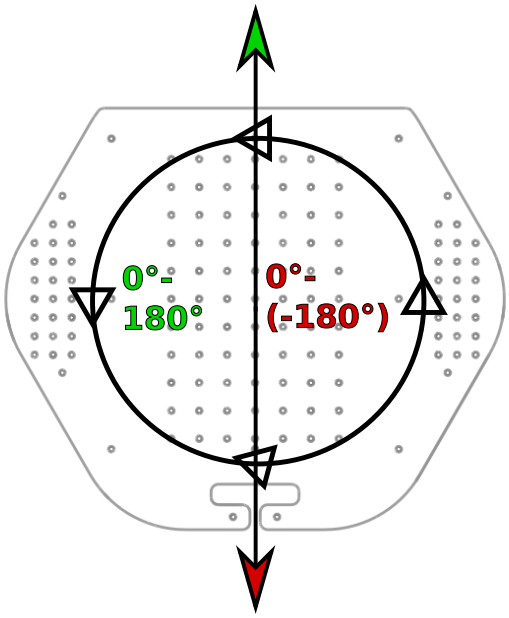


Figure 9.1.: How to specify the sensor angles

`{ObjectSize size`
Approximate size of an object. Contains fields xsize, ysize and zsize.
`time timeStamp`
Time, when the object was detected.
`string type`
Object type. See Configuration Section for details.

9.4.3. Robot coordinate system

Looking from the top at the turtlebot, the coordinate system looks like shown in Figure 9.2.

So the a positive X coordinate means, an object is ahead of the robot, while a negative one means, it is behind the robot. In the same way, a positive Y means that an object is detected left of the robot, while a positive Y coordinate means that it was detected on the right side.

As a side note, the Z coordinate tells you how high is an object in relation to the sensor. A positive Z means, that the object is above the sensor, while a negative Z means, that it is negative.

9.4.4. Using the sensor plugin in gazebo

First, make sure you configured the plugin as described in the configuration section.

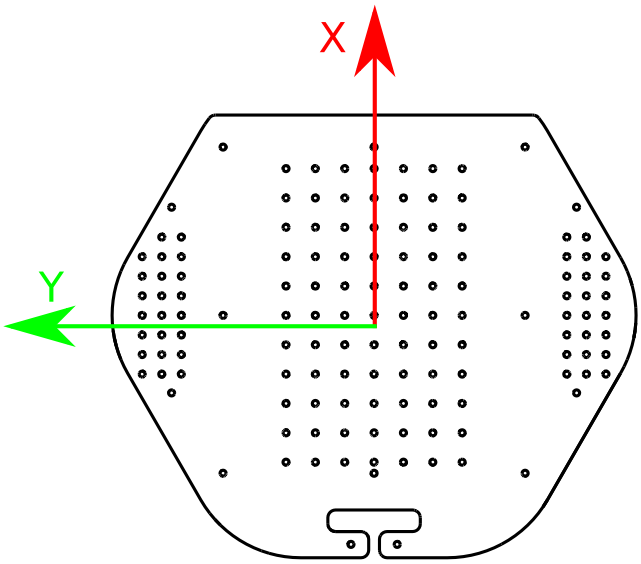


Figure 9.2.: The TurtleBot Coordinate System

Then simply launch gazebo using the simulation launch file:

```
roslaunch turtlebot_bringup simulation.launch
```

In order for the plugin to correctly detect and filter the objects, you have to give each object a name, that contains the type string from the configuration. A *victim_box* may for example be detected as a victim or box type of object.

Now, to add objects in Gazebo you can for example add a cube using the cube button in toolbar on top of the 3D view. For more information about adding objects in gazebo, take a look at the corresponding gazebo tutorial.

To quickly test the sensor you can add a cube or sphere and rename it as follows. First, add an object using the cube, sphere or cylindar button in the top bar. Then select the object by left clicking on it and then right click it to bring up the context menu. In the menu select *Edit Model*. You are now in the Model Editor. In the left Pane select the Model Tab and change the Model Name to a name that contains the type string. After you are done, exit the Model Editor by select *Exit Model Editor* in the *File Menu*.

Now you may have to continue the simulation by pressing the play button on the bar below the 3D view. After that you should be able to receive messages from the plugin on the *logical_camera* topic.

A. Appendix

A.1. .bashrc

```
# ROS specific
source /opt/ros/kinetic/setup.bash

# TurtleBots
ttbws_path=<insert your path to the ttbws folder here>

source ${ttbws_path}/devel/setup.bash

# ALICA/ttb_base specific stuff
export DOMAIN_FOLDER=${ttbws_path}/src/cnc-turtlebots/
export DOMAIN_CONFIG_FOLDER=${DOMAIN_FOLDER}/etc

# for loading the right models (environment and robot)
export GAZEBO_MODEL_PATH=${GAZEBO_MODEL_PATH}:${ttbws_path}/src/turtlebot/turtlebot_bringup/models
export TURTLEBOT_STACKS=ninja-hexagons
export TURTLEBOT_3D_SENSOR=ninja-kinect

#fancy prompt that also shows the current git branch
export PS1='\[\033[01;32m\]\u@\h\[\033[01;34m\] \w \[\033[01;31m\]$(__git_ps1 "[%s]")\[\033[01;34m\]\$\[\033[00m\] '
```

Figure A.1.: Template for your .bashrc

B. Exam

The question in all courses is: "What is relevant for the exam?"

Each semester I will try to give an answer to that question again and again ... and again. **Here is my answer (not fixed yet) for the winter term 2016/17**:

You should be able to give the right answer to the following questions and be able to fullfil the given tasks:

B.1. Questions: Foundations

- What is an agent?
- What is an autonomous agent?
- Sketch a state-of-the-art robot control architecture and explain its components.
- What is the difference between trueness and precision?
- Name two different sensors and explain their categories, properties, and value properties.
- What is an actuator?
- What is odometry?

B.2. Questions: External Software and Developing Tools

- What is a version control system and what is it good for?
- What does git pull, push, commit, etc. do?
- What is ROS and what is it good for?
- Describe the publish-subscribe model for communication.
- Describe the notion of nodes, topics, messages, publishers, subscribers.
- What is a Build Chain?
- What is CMake and for what do we use it?

B.3. Questions: C++

- What is written in the header file and what in the source file of a C++ class?
- How does a shared_ptr work and what is it good for?
- Explain roughly what the gcc compiler does.

B.4. Questions: Real Robots

- Which sensors does the TurtleBot have and what are they good for?
- Explain the details of the laser scanners sensor properties.
- Why do we use the extra laser scanner on the TurtleBot?

B.5. Questions: Gazebo Simulation

- Why do we use the logical camera sensor in simulation?

B.6. Questions: Navigation and Mapping

- What is a CostMap?
- Explain the following diagram (see Figure B.1).
- Sketch the CostMap on the following drawing.
- Sketch an algorithm for driving to the given destination (see Figure **??**) with the help of the laser scanner data.
- Denote the input and the output of AMCL and sketch how AMCL is calculating the robtos position based on the input.

B.7. Questions: Application Scenario

- For what is the ARTracker module?
- How is it possible that two TurtleBots communicate about positions, without having a common map?
- For what did we use the laser scanner?

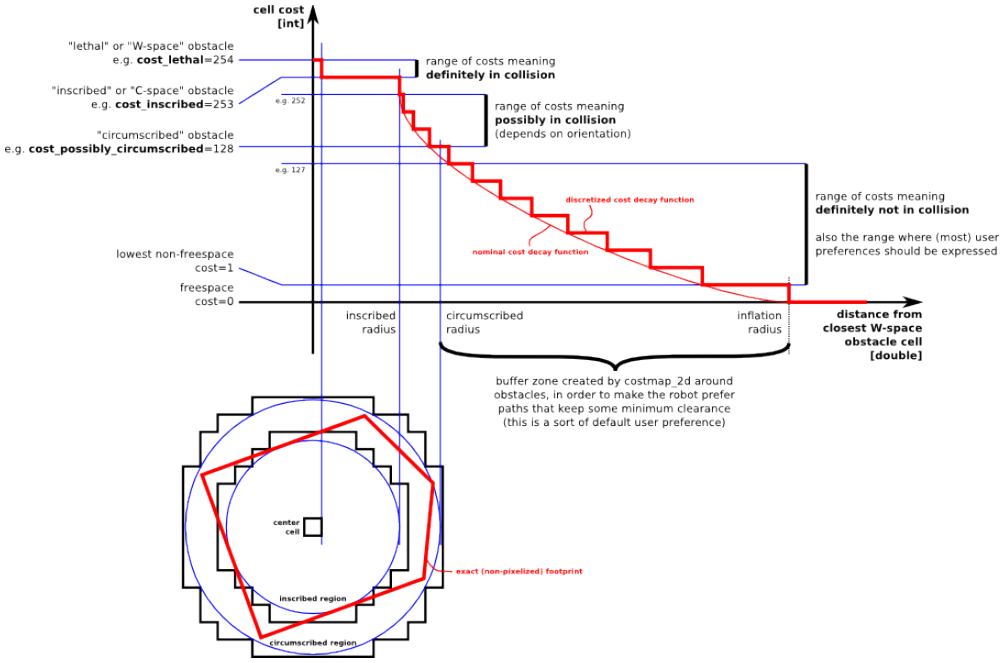


Figure B.1.: CostMap Valuation Diagram (Source: http://wiki.ros.org/costmap_2d)

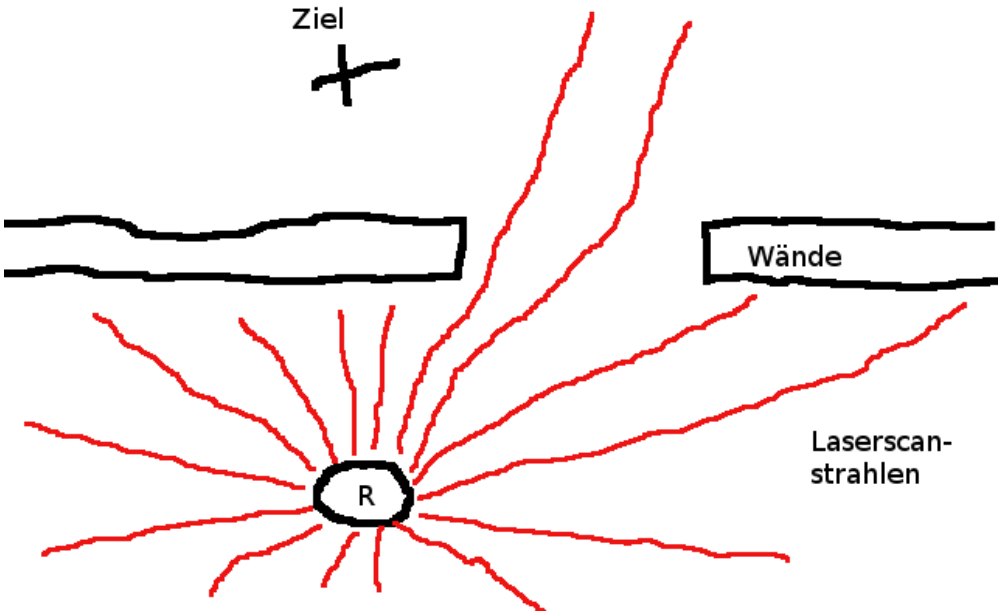


Figure B.2.: A Simple Navigation Task