TDDE25 Project: Humanoids
Version 5.0

Jon Dybeck (jondy276)
Carl Ehrenstråhle (careh585)
Erik Hansson (erikha172)
Mattias Tiger (mattias.tiger@liu.se)
Fredrik Heintz (fredrik.heintz@liu.se)

October 30, 2017

Contents

Project overview

Introduction

The goal of this project is to get hands on experience with programming humanoid robots. You will do this by programming the Nao robot to play soccer.

To make the Nao robot play soccer you have to implement both relatively simple behaviours, such as finding and kicking the ball, and more complex behaviours, such as scoring a goal. However, programming a humanoid robot to play soccer from scratch is a huge task. Therefore, you will work with a python interface for NaoQi which provides high level of functionality for the Nao robot. Moreover, you will get detailed instructions on how to get started with the python interface and the assignments will lead you trough the basic behaviours for playing soccer. However, you still have to put in a lot of time and effort into the project for it to be successful.

The examination of the project is to complete all the assignments described in this document on time. This includes writing a short report describing your final assignment, where you decide what you want to do.

When developing robots, it is quite common to have a simulator as a tool in the development process easier. However, this is mostly efficient when the behaviour is becoming more complex than what is expected of you in this course. Therefore, we will not use any simulator in this course. Nevertheless, if you want to know more about simulators there is a section explaining what a simulator is in the extra reading section.

Background

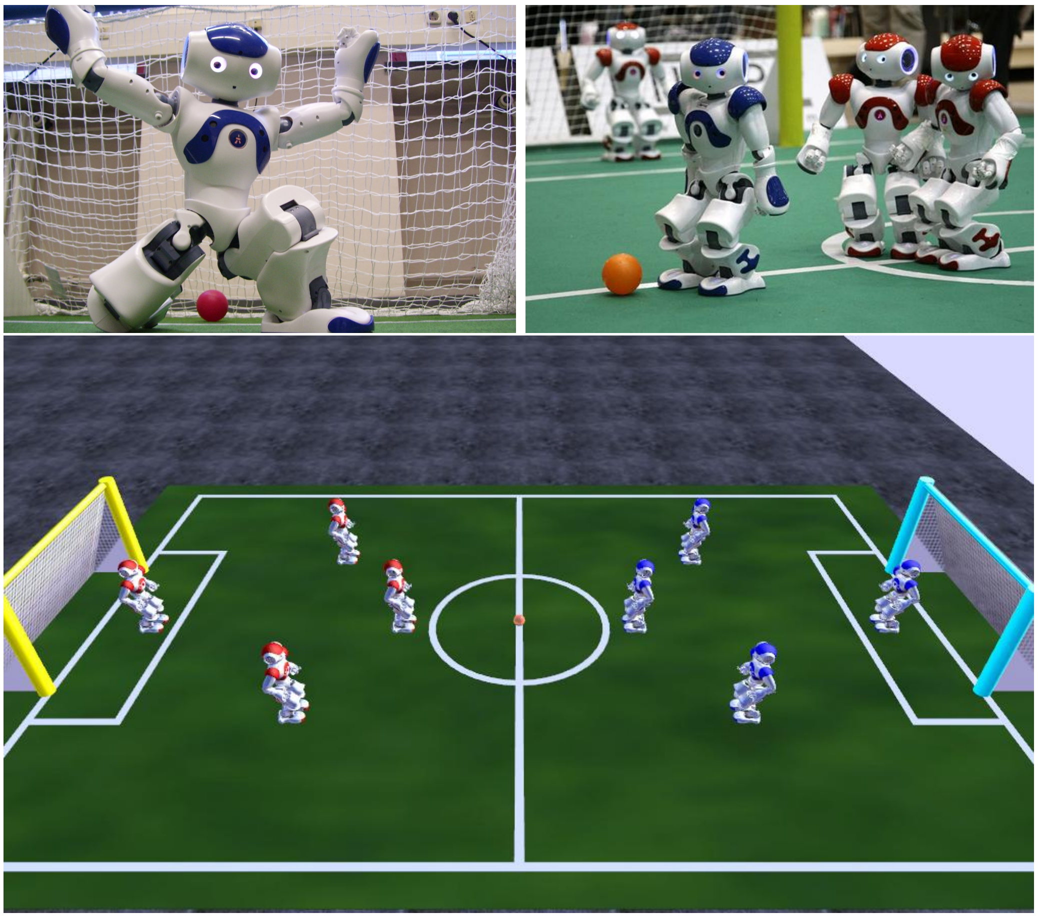

Figure 1.1: The goalkeeper of team Bembelbots (top left), picture from RoboCup official webpage (top right) and screenshot from the simulation league of RoboCup (bottom).

The history of robots playing soccer dates back to 1992. At that time it was mainly a subject of some papers, workshops on challenges in artificial intelligence and discussions. However, this led to the start of RoboCup, which is an international robot soccer competition held every year starting in 1997 in Japan. Sony CSL has more information about the history of RoboCup and some of the original papers on their website if you are interested in reading more (`http://www.sonycsl.co.jp/person/kitano/RoboCup/RoboCup-old.html`).

RoboCup has many leagues, such as the simulation league, the small size league, the standard platform league and the kid size humanoid league. In the simulation league teams of 11 simulated robots and 1 simulated coach compete against each other, while the other leagues consists of teams of real robots.

The Nao humanoid robot, that are used in this course, is currently the robot used in the standard platform league (SPL). A standard platform ensures that all teams compete in the league on the same terms when it comes to hardware. Hence, it is the software developed by the different teams that decides the winner.

A game in the standard platform league takes 20 minutes, 2×10 minutes. The game is played on a soccer field that is $9m \times 6m$, with two white goals and a black and white ball. For more information see the RoboCup Soccer home page (`http://www.tzi.de/`

`spl/bin/view/Website/WebHome`) and the SPL rules (`http://www.tzi.de/spl/pub/Website/Downloads/Rules2016.pdf`).

As the interest for RoboCup and robots in general has increased, more and more types of competitions have been introduced. For example, there is now competitions in rescue and dance.

The main goal of RoboCup is to beat the FIFA (human) world champions in soccer by 2050. Of course, this has to be done on human terms, which means that the robots are limited to humanoids that may not harm people and with human-like sensors. To achieve this, many breakthroughs are required. For example, the robots need significantly better hardware that makes them faster and more robust while still being safe to play against. They also need better software to control the actuators and sensors on the robots, implement behaviours such as dribbling like Maradonna or Messi and making the right decisions at the right time.

More information about RoboCup can be found at their website: `http://www.robocup.org/`.

The Nao robot

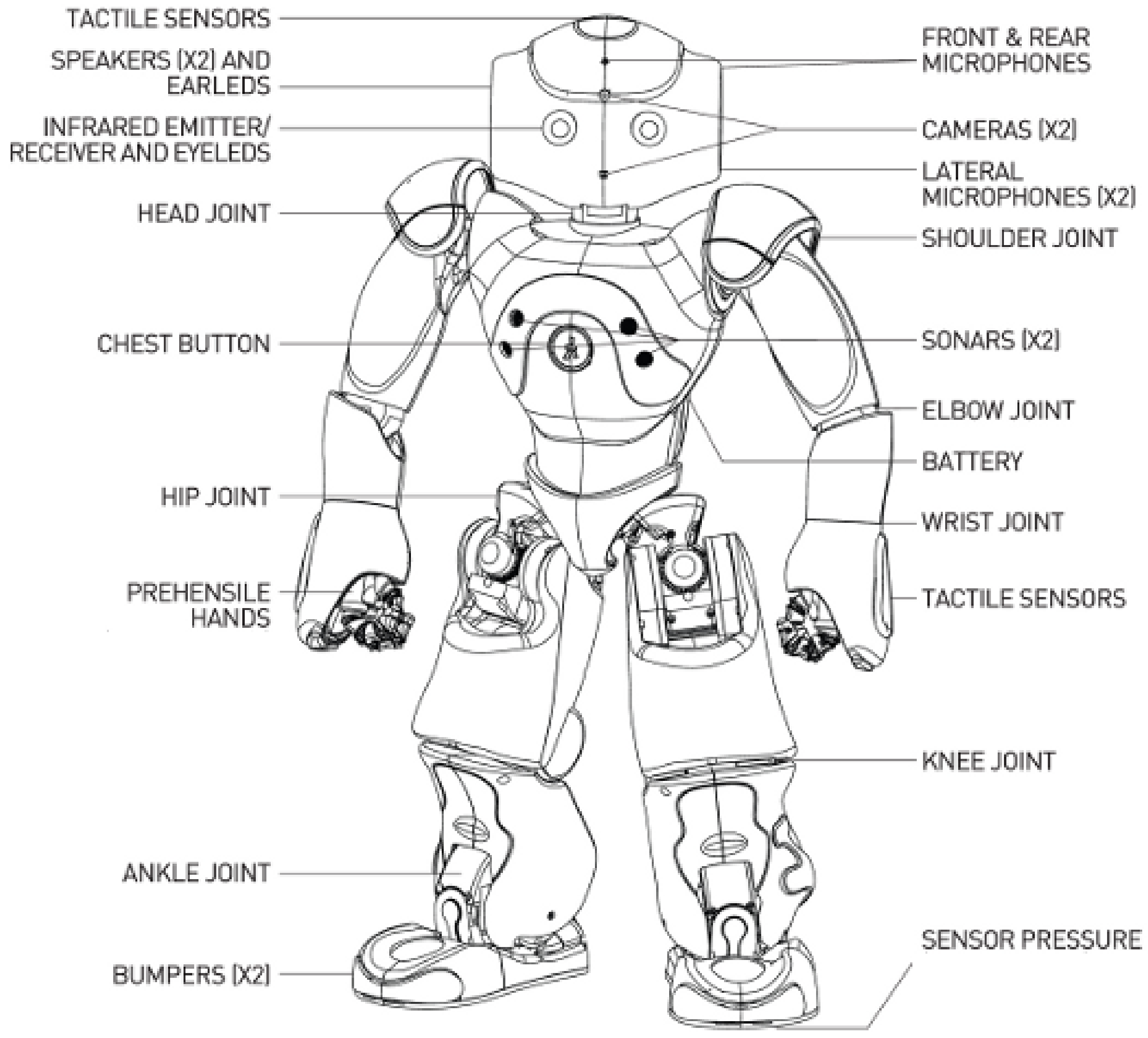

Figure 1.2: The Nao humanoid robot with its sensors and actuators.

In this project you work with a robot called Nao, developed by the French company Aldebaran Robotics (`http://www.aldebaran-robotics.com/en/`). It is a humanoid robot with 25 degrees of freedom (DOF). DOF is a measurement of independent motions, for example a robot with one joint that can turn in one direction has one DOF. For the Nao, the 25 DOF are in the form of 25 motors and actuators. In addition, it has several sensors: Two cameras, four microphones, a sonar range finder, two infra-red (IR) emitters and receivers, one inertial measurement unit (IMU) and nine tactile sensors. Furthermore, it is fitted with eight high-fidelity speakers and several LEDs.

The Nao is equipped with an Intel ATOM 1.6 GHz CPU that runs a Linux kernel and controls the mechanical hardware. It also has a middleware called NAOqi developed by Aldebaran Robotics. NAOQi provides a lot of predefined functionality and the possibility for users to add their own modules to NAOqi. However, one drawback is that the predefined functions are not always optimized. Therefore, the teams which competes with Nao have implemented their own versions of, for example, walk engines, predefined motions and image processing. However, this functionality still uses the NAOqi to control each joint and to fetch data from the sensors.

The technical documentation for Nao can be downloaded from Aldebarans website (`http://www.aldebaran-robotics.com/Downloads/Download-document/192-Datasheet-NAO-Huma` `html`).

Handling Nao and guidelines for the lab

There are, to avoid any unnecessary damages to the robot, rules for how to handle the Nao robot in the lab.

- You should under no circumstances run any code on the robot unless someone is on the field and is prepared to catch the robot in case something happens.
- When you are lifting the robot you should have a firm grip around its waist with both your hands (see the picture below).
- Use a new robot if the one you are using runs out of battery. If there is no other robot, you will have to change battery by following the instructions at the table. Finally, connect the empty battery, or robot, to a charger.
- Do not leave the robots on the floor unsupervised.
- Contact an assistant or supervisor immediately and mark the robot if something has broken on the robot.
- Connect the robot to a charger when you are done using it.

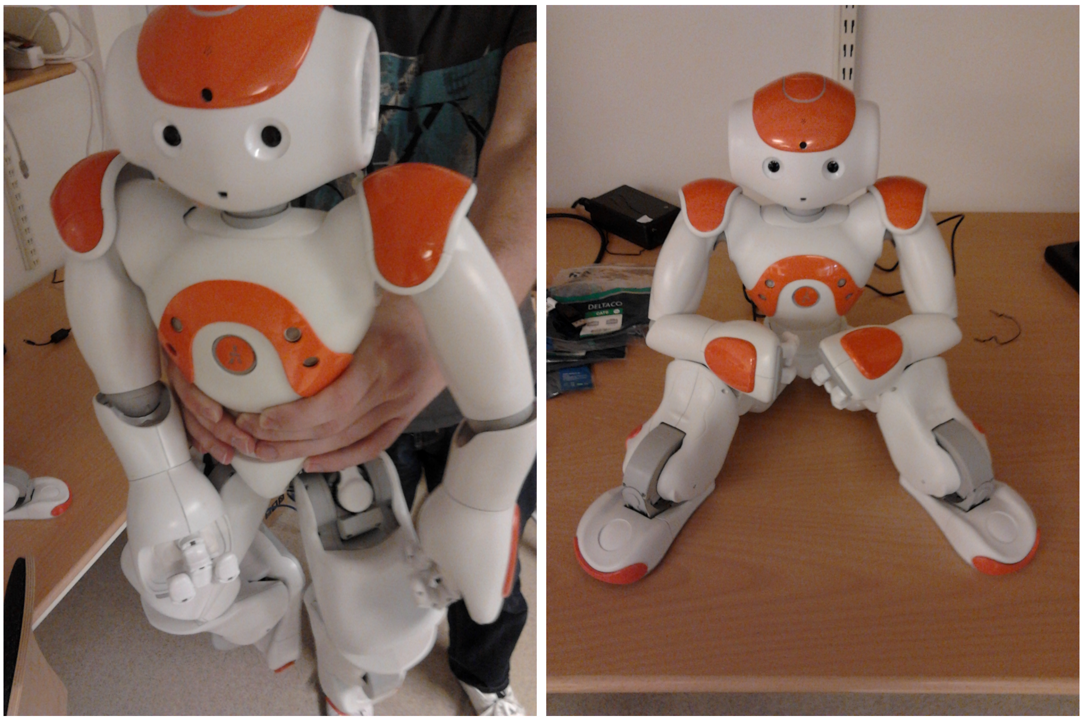

Figure 1.3: How to hold the Nao robot (left), Nao in sitting position (right)

In addition to the rules for how to handle the Nao humanoid robot there are a few guidelines for the lab:

- You should never wear shoes in the lab. We want to keep it as clean as possible.

- The computers in the lab is under the same rules as the rest of the computers at the university. This means that you are, for example, not allowed to play games on them.
- Share the robots between each other fairly since there is a limited number of robots available and some might get damaged during the course.

Configuration and installation

Configuring the tools

The workstations in the lab uses the same system as the computers in the SU-rooms. Therefore, you will be able to access the files from any computer outside the lab and can have your personal configuration for the programs you use. However, we strongly recommend you to use the same configuration for all the project members since it ensures that the code is formatted in the same way for everyone and that the code can run on every project members account. Moreover, we strongly recommend that your configuration helps you to keep tab on the length of your lines in your code since you should follow the standard for Python, pep-8 (a good idea is to read or re-read the chapter 0 "programmeringspraxis" in the python programming course, TDDD73). Furthermore, we recommend that you configure the indentation, according to pep-8, to four spaces. This saves you the trouble of having different indentation when writing code with others and on different computers. Moreover, it will make sure that you will not get any problems in the case that one of the assistants decides to test your code on their own accounts.

Lastly, in the project you are required to use version control to store your code. There are two options for version control. The version control system you will use in this course is Git, which you have already used in your python corese (TDDD73). To use Git log in to the university's gitlab server, create a repository, add all your project members and all the assistants as members (or admins) in the repository.

Downloading your repository

Your repository is the place where you will store your code during the project. Moreover, it contains the python wrapper that you will use to build your code during the project. To fetch the start of your project you have to log into `https://gitlab.ida.liu.se` and open the project (a project is usually called repository) that has a name similar to **humanoids-#** where # is the number of your group. The website will now display a link which starts with "git@gitlab.ida.liu.se" which we will call *<your-url>*. Fetching your repository can now be done with the following command:

```
andan000$ git clone --recursive <your-url>
```

This creates a folder for your project that also contains python wrapper.

Generating API

The API for the Nao robot are some websites that are generated from the code. To generate the websites one has to move to the root folder for the python wrapper in the terminal and enter the following:

```
andan000$ make doc
```

This will automatically generate or update the documentation. Opening the API is done by simply open the **nao_api.html** file in a web browser (or by writing *gnome-open* *nao_ api.html* in the folder where it is located).

Version numbering

All source files in the wrapper are marked with a version number in the header of the file. This is so that you can make sure that you have the latest version. In the file tab on the git repositories web-site (`https://gitlab.ida.liu.se/frelo223/humanoid-lab/tree/master/nao_api`) you will see a list of all the files and which is the latest version.

When you have got the latest version you can compare the version numbers in the files with those in the README.md (or the on the repository website) to make sure that everything is up to date.

Assignments

Introduction

The project consist of two phases, the introductory phase and the main phase. The introductory phase consists of two assignments and should be finished, at the latest, by the end of the first period. The main phase consists of three assignments and should be finished by the end of the course.

You should separate your code for each assignment in different folders in your Git repository (e.g. <Git root>/ass1, <Git root>/ass2). This should be done so that the assistants and the supervisor easily can see which files that is used for each assignment. However, this does not mean that you should rewrite the code but that you should copy your old code that you will use again.

We recommended that you do the assignments in the order they are presented since they, in most cases, build upon previous assignments. This allows you to reuse the code that you developed in previous assignments. Furthermore, we strongly suggest that you follow the time schedule below so that you will be finished on time.

Assignment	**Expected week for completion**
Intro phase: Assignment 1	week 44
Intro phase: Assignment 2	week 46
Main phase: Assignment 3	week 47
Main phase: Assignment 4	week 48
Main phase: Assignment 5	week 49
Main phase: Assignment 6	week 50
Main phase: Demonstration	week 51

Table 3.1: Project time schedule

The instructions to all the assignments are presented in the same way as this introduction, i.e. an introduction and then the following subsections.

Purpose A short description of what you are supposed to learn from the assignment and a very brief description on how it will be learned.

Task A description on what you are supposed to do in the assignment.

Examination A detailed description on what you have to do to pass the assignment.

Milestone(s) A detailed description for every milestone of an assignment.

Intro Phase: Assignment One – Hello world!

In this assignment you will learn how you use the python wrapper to program Nao. Since this is the first assignment, you will get highly detailed descriptions on how to complete each milestone. However, keep in mind that you will not get equally detailed descriptions in the following assignments.

Purpose

The purpose of this assignment is that you should learn the basics of programming the Nao humanoid robot. Through the assignment you will be guided on how to use the Nao, the python wrapper and other related software.

Tasks

Your task is to complete the three milestones of the assignment. This includes using the basic motions of the Nao robot and reacting to information from the world through the sensors of the Nao robot.

Examination

To pass this assignment you have to:

1. Complete all the milestones.
2. Demonstrate your programs for each of the milestones of the assignment to an assistant.
3. Demonstrate an understanding of the code and the system by individually answering questions regarding your code.

Milestone One – Stand and sit

The first milestone in the project is to get a simple program to run on the robot. In this case the program should make the robot first stand up and thereafter, sit down. When building this program there are three major steps that is good to follow:

1. Make a general idea for how the behaviour is to be implemented (which actions does the robot have to do to stand and then sit?). In this simple case this might be done by simple discussing how to do it or simply by implementing it. However, this will not be as easy to do when the behaviour gets more complex. Therefore, we will introduce the idea of a concept diagram. A concept diagram can take its form in many shapes. For example, a state diagram or a flow chart. However, the most important part is that it describes how the behaviour is going to be implemented on a conceptual level. An example is the diagram below which describes the behaviour in this assignment.
2. Implement the behaviour according to the concept diagram that you have created (which algorithm implements the idea you came up with in step 1?).
3. Test the code by running it on the robot (how well did the algorithm work?). During this step it is most likely that you will encounter some problems with your concept for how the behaviour is to be implemented. However, this is not something to be afraid of. It is rather one of the most fundamental parts of science: find a hypothesis, test it and thereafter, reformulate the hypothesis if needed. However, it is important to keep concept diagrams through the iterations of these three steps

since it can be used to show progress and, more important, they document what already have been tested. Furthermore, by adding notes to the concepts diagram of what went wrong you have a good documentation that can be used to review the project and from which it is possible to write a good report or make improvements.

As mentioned above we start the work on this assignment by coming up with a concept of how to implement the wanted behaviour. In this case it is simple. We first want the robot to stand and when it is standing it should sit down. However, there is two more part to it that is not mentioned explicit. Firstly, the actuators (the electrical motors in the robot) has to be turned on. Secondly, when the robot has sat down the actuators needs to be turned off to save energy and to prevent overheating. Hence, our concept can take its form in the following concept diagram (here in the form of a UML state diagram):

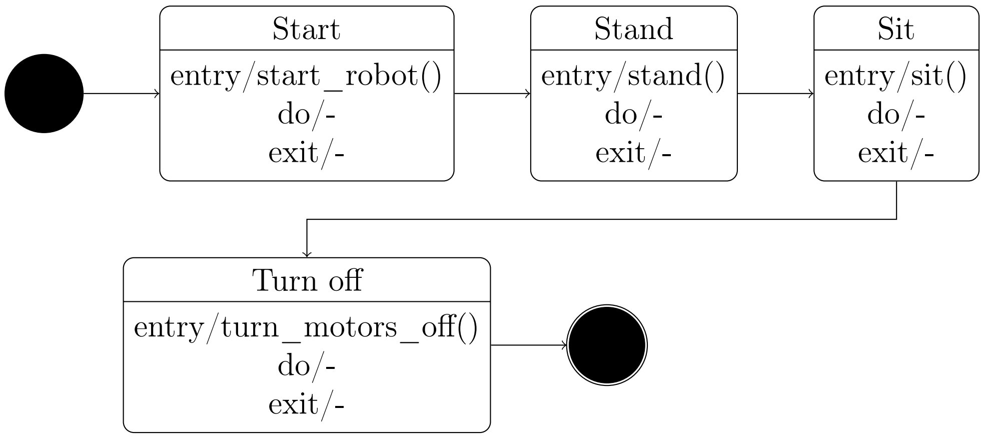

Figure 3.1: State diagram describing the stand and sit program.

In the diagram above the different parts are the following:

- States: the states are the rectangles with the rounded corner. They consist of a name followed by a horizontal line and thereafter some actions. These actions are in general an entry action which is executed when the state is entered, a do action which is executed while in the state (this can be more than once) and an exit action which is executed when leaving the state.
- Transitions: the arrows going between the states that directs which the next state will be.
- Start: the start of a state diagram is marked with a single filled black circle with a transition from it.
- End: the end of a state diagram, in the cases where one exists, is marked with a filled black circle that is circled by another non-filled black circle.

Writing the program

The next step after the concept diagram is created is to implement it in code. The first part of our diagram is to start the robot. However, at the moment we do not have any functions for starting the robot with. Hence, we look in the module that we have available and find that the Control class has a method start_robot so we start of with importing the Control class:

```
from nao_api import Control
```

Unfortunately, this is not enough for the program to import the Control class. We also need to tell the Python interpreter where the nao_api module is located. The easiest way to do this is to add the path to the *PYTHONPATH* variable in the shell. This can be done by navigating to the folder containing the folder nao_api and enter the *pwd* command. Copy the result from the command and add the following line to your **.bashrc** file:

```
PYTHONPATH=$PYTHONPATH:<result-from-pwd>
```

The next step is to create a controller for the robot that you are using. This is done by calling the constructor with the name of the robot (the name is "nao#" where # is the number the robot has). In our example the robot is "nao1":

```
control = Control("nao1")
```

Now we have a controller for our robot which we can use for making the robot do things. In our case we will simply tell it to do what we drew in the concept diagram. To start up followed by stand up, sit down and finally, to turn off all the motors. To do this we write the following code:

```
control.start_robot()
control.stand_up()
control.sit_down()
control.turn_motors_off()
```

If you have done as it said you should now successfully have written a program that will make the robot stand up and thereafter sit down. The next step is to run the program which is described in the next section.

Running the program

To run the code on the robots you have to do the following steps:

1. Make sure one of your group members are beside the robot on the soccer field.
2. Enter the following code:

   ```
   andan000$ python my_program.py
   ```

   Note that this runs the program with Python2 instead of Python3. This is meant to be since NAOqi, which the wrapper uses, is written in Python2 and therefore, require us to run the program in Python2.

Milestone Two – "Hello world!", says Nao

In this milestone you will extend your program from milestone one to make the robot say "Hello world" after it has stood up. To implement this behaviour we will add to the program in the previous milestone since they are extremely similar. Furthermore, you will not need to save the code for the previous milestone since this is within the same assignment.

The first step is to modify our concept to include the new parts of the behaviour. The resulting concept diagram is the following:

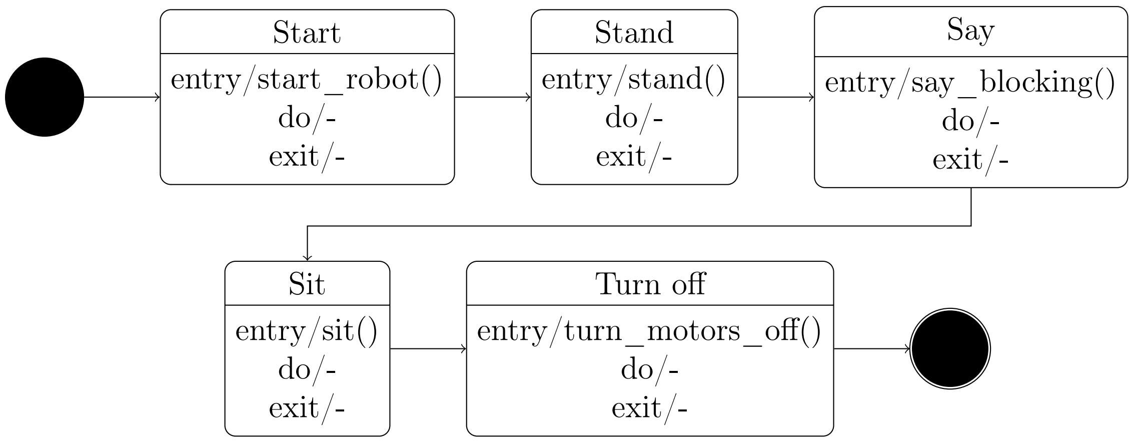

Figure 3.2: State diagram describing the hello world program. Using blocking say function.

As some of you might have noticed, the action in the Say state in the diagram is not only named say. That is because there are two say methods: say_blocking and say_non_blocking. The only difference between these two are, as can be deduced from the name, that one is blocking and the other is not. Of course, that raises the question of what blocking and non blocking means. These properties are something that you will encounter, under a lot of different name, in many systems as an engineer. The general idea is that a blocking function will hinder the program to continue until it has been finished and a non-blocking function will run in parallel while the program continues with its remaining parts. The use of the non-blocking say method would require updating our concept to avoid having the robot start talking and then sit down before it has finished. Hence, our concept diagram could look like the following:

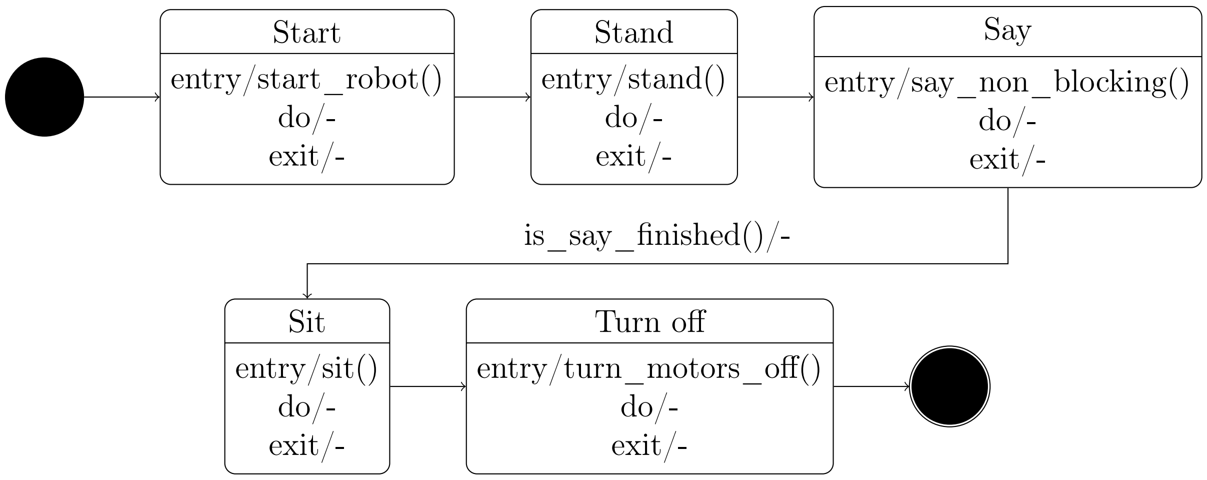

Figure 3.3: State diagram describing the hello world program. Using a non-blocking say function.

In this state diagram we introduce a new thing. Namely, the text on one of the transitions. The text on a transition is on the format "condition/action". The condition is a boolean statement that must evaluate to True before the transition can happen and the action is an action that will be executed when the transition happens.

The drawback with using a non-blocking say function in this case is that we will have to implement the function is_say_finished ourselves since it is not implemented in the python wrapper. In general, it is recommended to use existing functionality over rewriting it as your own.

Note that if there are no text on a transition it is assumed that it is under a condition that always evaluates to True. Hence, it will always happen.

Writing the program

As you might have noticed this program has a lot of similarities with the program in the previous assignment. Therefore, you can just modify the code that you wrote in the previous assignment. However, we must first decide what we are going to modify. If you compare the two concept diagrams, there is only one difference. That is that the robot should say "Hello World!" after it has stood up and before it sits down. Therefore, it seems logical that we add something that makes the robot say something in between the statements that makes the robot stand and the statement that makes the robot sit:

```python
from nao_api import Control

control = Control("nao1")

control.start_robot()
control.stand_up()
control.say_blocking("Hello World!")
control.sit_down()
control.turn_motors_off()
```

Running the program

Just like in the previous assignment you have to follow the special steps for running code on the robots in the lab. Therefore, make sure that at least one of your group members are with the robot on the robot field before you run the code from the computer. Moreover, do not forget to use Python 2 instead of Python 3.

Milestone Three – Wait for sensors

In this milestone you will extend your program to make the robot wait until one of its bumpers (see the Nao overview) is pressed before it stands. Furthermore, the robot should wait with sitting down until one of its bumpers has been pressed again. Moreover, it should say "goodbye" before it sits down.

As before, we will start by updating the concept to describe the extended behaviour. The new concept can be described in the following concept diagram (now using a flow chart instead of a state diagram):

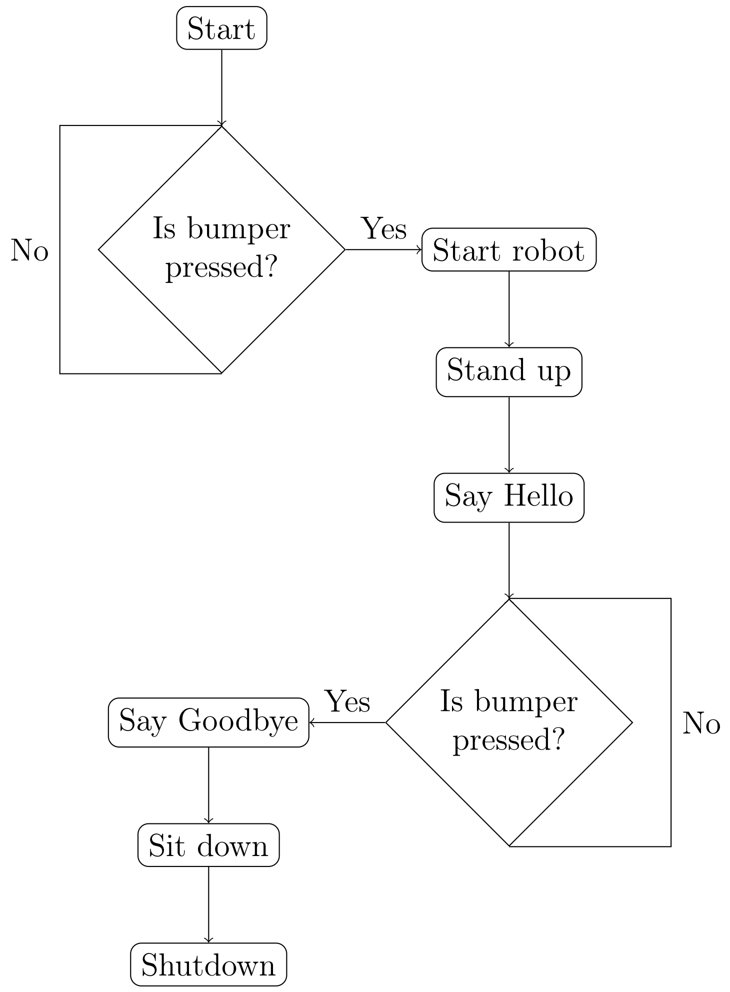

Figure 3.4: Flowchart describing program with waiting for bumper pressed.

A flowchart is different from a state diagram in multiple ways. Mostly, it does not describe the actions or transitions that the system needs. Hence, it, usually, describes the system on a higher abstraction level. As a result, it leaves more room for interpretations of the diagram than the state diagram. Therefore, we recommend that you use the state diagram when making a concept diagram for your programs.

In the diagram you can see that we have arrived to a decision node which questions whether a bumper is pressed. Hence, our implementation will now have to react to the value from our sensors. In the case of the bumpers it is quite easy because the sensors are binary, that means they are either pressed or they are not. To get the information from the sensors we will use the is_left_bumper_pressed and the is_right_bumper_pressed methods.

Writing the code

Just as the previous assignment, this is similar to the previous program so it is a good idea to modify the code that was written before. However, there are some problems that you have to face when writing this part of the program. Namely, how do you make the robot wait for someone to press one of its bumpers? A good idea might be to take a top-down approach and assume that you will write a function that waits until someone press one of the bumpers given that it gets something to access the sensors of the robot. Then your code for the program might look something like this:

```python
wait_until_bumper_is_pressed(perception)
control.start_robot()
control.stand_up()
control.say_blocking("Hello World!")
wait_until_bumper_is_pressed(perception)
control.say_blocking("Goodbye")
control.sit_down()
control.turn_motors_off()
```

Now there are two major things left for to do. Firstly, what is the *perception* variable that the wait_until_bumper_is_pressed function uses? Secondly, how do we implement the function?

The answer to the first question can be found by looking at the classes that are available. There are three available, the Control class, the Communication class and the Perception class. Out of these three the Perception sounds the most promising. So we can define the control and perception variables as following:

```python
control = Control("nao1")
perception = Perception("nao1")
```

Now to the implementation of the wait_until_bumper_is_pressed function. One of the most simple way of implementing this is to use a while loop that does not do anything until the head button is pressed. However, this is quite a bad idea because it will constantly evaluate the expression in the while loop. Therefore, we will use unnecessary amounts of the processor's capacity. With a bit of thinking, and maybe testing, we can come to the conclusion that it is not realistic that the head button is pressed for less than a tenth of a second. Therefore, there could be something in the loop that waits for a tenth of a second before it checks if the head button is pressed again. Moreover, if this is placed inside the while loop then the result will actually become that we ask every tenth of a second if the head button is pressed and will not leave the loop before it is pressed. Sleeping for a set amount of time can be done with a standard python function called sleep that is in the time module. Hence, our wait function could look like this:

```python
INPUT_SLEEP_TIME = 0.1


def wait_until_bumper_is_pressed(perception):
    """
    Waits until one of the bumpers are pressed
    :param perception: the perception object that is used to sense
    """
    while not perception.is_left_bumper_pressed() and \
          not perception.is_right_bumper_pressed():
        sleep(INPUT_SLEEP_TIME)
```

As you can see we defined a variable called *INPUT_SLEEP_TIME* that we set to 0.1 instead of passing 0.1 as an argument to the sleep function. This is a so called a constant, which can easily be changed. Furthermore, by using a constant it is guaranteed that if that it is changed, then the value will be changed in all places that the constant is used (compared to having to change the same value to a new in a lot of places).

With these additions the final program could look like this:

```
from nao_api import Control, Perception
from time import sleep

INPUT_SLEEP_TIME = 0.1


def wait_until_bumper_is_pressed(perception):
    """
    Waits until one of the bumpers are pressed
    :param perception: the perception object that is used to sense
    """
    while not perception.is_left_bumper_pressed() and \
          not perception.is_right_bumper_pressed():
        sleep(INPUT_SLEEP_TIME)


control = Control("nao1")
perception = Perception("nao1")

wait_until_bumper_is_pressed(perception)
control.start_robot()
control.stand_up()
control.say_blocking("Hello World!")
wait_until_bumper_is_pressed(perception)
control.say_blocking("Goodbye")
control.sit_down()
control.turn_motors_off()
```

Running the code

Running the code is done in the same way as for the two previous assignment: At least one of the group members should be at the soccer field with the robot before you run your code using Python 2.

Intro Phase: Assignment Two – Basic skills

Purpose

The purpose of this assignment is to get a deeper understanding of how to program a robot, how to use actuators and sensors. This will be done by programming the basic skills needed for playing football with the Nao.

Task

The task of this assignment is to make the robot find the ball, even if it is not in its line of sight, and then follow it.

Examination

To pass Assignment Two you have to:

- Complete all the milestones.
- Present a concept diagram over your robot.
- Divide the code in a logical manner between different files.
- Demonstrate a working ball tracker.
- Individually explain how the code works.

Milestone One – Walking

The first milestone of the assignment is to make the robot walk. First, it should walk forward for a while; then it should turn roughly ninety degrees on the spot; finally, it should walk in an arc so that it is close to the same position as it started in but facing the opposite direction (basically doing an u-turn).

As with the previous assignment you should start by creating a concept for the wanted behaviour and make a diagram over it. In which form you make the diagram is up to you. However, we recommend using UML state diagrams and at least start by sketch them out with pen and paper since it usually goes faster than using a computer program for creating diagram.

Milestone Two – Turning the head

In the second milestone you will write a new program that turns the head of the robot. Try to come up with a motion that could be useful when searching for a ball.

To finish this assignment it is recommended that you follow the same procedure as described in the previous milestone.

Milestone Three – Using the camera

The focus in the third milestone is to make use of the robot's camera. This should be done by detecting the ball as well as reacting to finding the ball and loosing it. However, you must be aware that there is a risk that the robot will not recognise the ball in front of it every frame of the camera. Furthermore, you must take this into account since it is not acceptable if the robot believes it has lost the ball only because it has not been recognised in a couple of camera frames.

Hint: The `say_non_blocking`, `set_face_leds` and `set_ear_leads` functions is really good for troubleshooting while running a program on the robot.

Hint: All observations, and messages, has a time stamp. These time stamps can be used to determine how old an observation, or a message, is. Moreover, the time stamps are set with the `time` function in the standard `time` module.

Note: Nao has two cameras, one facing forwards and the other facing downwards. Both of them can not be active at the same time, instead you have to swap between them. Furthermore, the top camera can not see the feet of the robot. Therefore, you have to use both cameras to keep the ball in sight. Switching camera can be done with the `use_upper_camera` and the `use_lower_camera` functions. However, this switch do not happen instantaneously. Therefore, you will have to be aware of the fact that it might still be the old camera that the data comes from for a while, (typically a few seconds.)

Milestone Four – Finding and tracking the ball

To achieve this milestone, the robot should be able to find the ball when it is within a two meter radius of the robot (it does not have to find the ball if it is located in an impossible spot, such as between its feet). This milestone combines the two previous milestones.

For completing this milestone we strongly recommend you to structure your code in multiple logical sections. For example, one file could contain the code for finding the ball while another file could contain the code for tracking the ball.

Milestone Five – Following and kick the ball

In the final milestone of this assignment you should make the robot find the ball and then follow it as it is pushed around the field (within reasonable limits). Moreover, if the robot ever comes close to the ball, it should kick the ball as well. As in milestone four it can be very useful to use multiple files to implement this behaviour.

Main Phase: Assignment Three – One player

Purpose

The purpose of this assignment is to expand your code so that the robot is able to play soccer and not just mastering basic skills. To achieve this, you need to use more complex programs than in the previous assignments. You will most likely also have to improve the basic skills that you have started to develop in the previous assignment.

Task

The task is divided into four milestones that incrementally creates a program that can find the ball and kick it into the goal. As usual, we strongly recommend you to re-use the code you have written in previous assignments.

Examination

To pass Assignment Three you have to:

- Complete all the milestones.
- Present concept diagram(s) for how the behaviour is implemented
- Demonstrate how the robot can find the ball, position itself and then score a goal.
- Be prepared to individually explain how the code works.

Milestone One – Using localisation

The first part of this assignment is to start using the localisation parts of the `nao_api` module. The localisation consists of two methods in the `Perception` class, `get_goalposts` and `get_robots`. Use these two methods to make the robot look towards one goalpost of your own choice.

Hint: Use a constant that keeps track of which goalposts you are to look at (and which goal your are scoring at later in the assignment). This will make it easier to change to the other goal.

Milestone Two – Aligning the robot

We are now ready to start with slightly more complex tasks related to making a robot play soccer. First, aligning it in relation to the goal and the ball in order to kick the ball in the right direction.

Hint: Can your code keep control of the ball while aligning the robot?
Hint: Can the robot realise that it has lost the ball?

Milestone Three – Scoring a goal

As the final step in this assignment your robot should actually be able to score a goal. There are many ways to achieve this task and you are free to explore whichever of these that you want. However, your program should be able to score a goal from different positions around the field. This will most likely require that the robot kicks the ball more than one time. Hence, your code should be able to handle that in an acceptable manner.

Main Phase: Assignment Four – Two players

Purpose

In a soccer game with multiple players, communication and co-operation is essential to winning. Until now, your robot has been incapable of communicating with other robots. The purpose of this assignment is to teach you the basics of co-operation.

Task

In this assignment you will be introduced to the communication part of the python wrapper. Furthermore, you will create a program that allows two or more robots to co-operate in order to solve a couple of tasks. The first task consists of making one robot react to sensor inputs to another. Following this is a task to make one of the robot direct the other robot through giving permission for shooting. The final task is to make one robot pass the ball to the other.

Examination

To pass Assignment Four you have to:

- Complete all the milestones.
- Present a concept diagram of the program.
- Demonstrate two robots passing the ball between themselves.
- Be prepared to individually explain how the code works.

Milestone One – Send messages

For communication between robots there is a final part to python wrapper. Namely, the Communication class. This class contains five functions that can be used to communicate between robots. The details for the functions are as usually located in the API. The Communication class can be used to send messages between the robots. An example is the following program.

```python
from nao_api import Communication
from time import sleep

nao_com = Communication("nao1")

# No feedback on delivery
nao_com.broadcast("I'm alive!")
# Feedback on delivery
nao_com.send_message("nao2", "Are you there?")

while True:
    if nao_com.has_unread_messages():
        recv_msg = nao_com.get_unread()
        if recv_msg["msg"] == "Nope":
            nao_com.send_message("nao2", "...")
        elif recv_msg["msg"] == "Yes":
            nao_com.send_message("nao2", "Great, let's play football!")
        break
    else:
        sleep(1)
```

As you can see in the example above the message that was fetched with get_unread contained a dictionary and just not the message. That is because a message contains two parts. The first is the message that was sent and has the key "msg". The second is the time when the message was sent and has the key "ts".

Your task is now to modify the code from the last milestone of the first assignment so that touching the bumper of one robot makes another robot react.

Milestone Two – Attacking with pass

In the previous milestone you learned the basics of communication. In this final milestone, you are to use this knowledge to make two robots co-operate. Your task is to make two robots work together by having both the robots locate the ball (preferably, they will work together to locate the ball). Thereafter, they should work together to score a goal. One restriction is that the robot is not allowed to walk with the ball or make a forward pass to itself. In essence, a robot passes the ball to the other robot until one of them can score a goal.

Main Phase: Assignment Five – Avoidance

Purpose

When playing soccer there are, usually, some opponents. These will, and sometimes even the players on your own team, get in the way of you. Understandably, it is not allowed to simply run them over. The very same rule applies to robots playing football. Hence, the purpose of this assignment is for you to get a taste of simple obstacle avoidance.

Task

Your task is to implement a simple obstacle avoidance for the robots and integrate this with your existing program that plays soccer.

Examination

- Complete all the milestones.
- Present a concept diagram of the program.
- Demonstrate a robot avoiding another robot as described in the milestones.
- Be prepared to individually explain how the code works.

Milestone One – Naive avoidance

Obstacle avoidance is a quite complex matter if done properly. Therefore, we will only use a simple method to avoid obstacles, walk sideways before continuing forward. Moreover, we will assume that there are only three objects on the field: Robots, goals and balls. Conveniently, these are exactly the same as you can perceive using the `nao_api` module.

Your task in this milestone is to write a program that makes the robot walk forward. In case that your robot encounter another in front of it, yours should avoid it by walking sideways, go pass it and thereafter continue.

Milestone two – Return to course

This is a small increment to the naive avoidance. Firstly, the robot should now walk towards one of the goal posts (your choice). Secondly, your robot should continue towards the goal post after it has avoided another robot.

Milestone three – Avoid with ball

Now it becomes a bit more tricky. In this scenario, your robot is in control of the ball when another robot is walking towards the ball. Your goal is now to make your robot kick the ball around the approaching robot, pass the robot and thereafter take control of the ball again.

Milestone four – Integration

The final part is one of the most error prone when it comes to software development, integrating the new functionality with the existing. In this case it means that you should include the obstacle avoidance behaviour in your existing code that makes two robot co-operate to score goals.

Main Phase: Assignment Six – Elective task

Purpose

Robots are used for a huge variety of tasks, not just playing soccer. There are always new ways of completing a task and new ideas on what could be solved by robots. This assignment gives you the opportunity to try your own ideas for what the Nao robot can be used for.

Task

You are relatively free to define the task for this assignment. However, it has to be approved by one of the assistants or the supervisor.

These are some examples of what you can do:

- Improve the obstacle avoidance to make use of the other robots movements and using a smarter obstacle avoidance method.
- Program a goalkeeper.
- Make a throw-in.*
- Make two robots hand the ball to each other.*
- Make the robot go trough an obstacle course.

* These tasks require functionality that is not part of the python wrapper. However, they can be implemented quite easily by only using Python and you are very welcome to ask an assistant on how to do so.

Examination

The examination of this assignment is to define a task and to make a serious effort to complete the task.

Extra reading

Monitor

Monitor is a software that allows you to stream video from the Nao robot. Naturally, this can be a powerful tool when debugging your program since it allows you to determine whether the problem lies in the fact that the robot never sees the ball or if the problem is something else. However, it comes at a cost namely network traffic since the video feed is streamed over the network it adds to the load on the network. Unfortunately, there are some network problems at the moment which becomes more likely to occur the heavier the load on the network is. Therefore, it is essential that you only use the Monitor software when needed and turn it of otherwise, especially when there are more groups working in the lab than your.

To start the monitor program you execute the following command in a terminal:

```
andan000$ /sw/aldebaran/choregraphe-suite-2.1.3.3-2-linux64/monitor
```

Football field

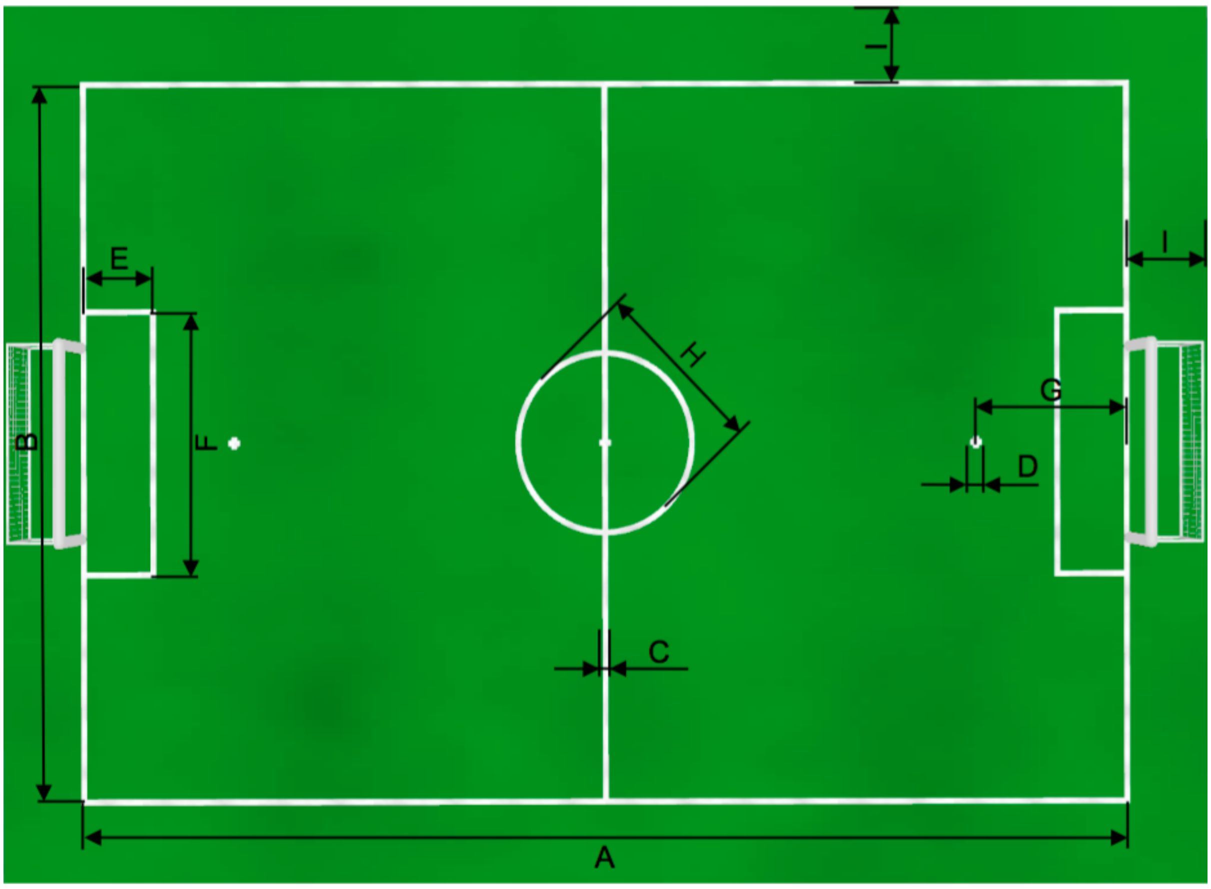

Figure 4.1: The coordinates of the football field. (0, 0) is in the corner closest to the door towards the computer area. The picture is taken from the Official rules for RoboCup SPL `http://www.tzi.de/spl/pub/Website/Downloads/Rules2016.pdf`.

Line	length(m)	Line	length(m)	Line	length(m)
A	6.15	B	4.35	C	0.05
D	0.1	E	0.6	F	2.15
G	1.3	H	1.5	I	0.7

Coordinate system

When working with the Nao robot (or any robot that is to move around in the world) it is important to have a well defined coordinate system that is used. In this wrapper the same coordinate system that is used in NAOqi is used with some addition (included below). For your convenience we provide a short summary of the coordinate system and terminology below. If you are interested in the full documentation we refer to the `http://doc.aldebaran.com/2-1/index_dev_guide.html`NAOqi documentation.

Term	description
X	Forward and backward direction, positive values are forward and negative values are backward.
Y	Sideways direction, positive values indicate left and positive values indicate left.
Z	Up and down direction, positive values are up and negative are down.
Roll	Rotation around the X-axis, positive values are counter-clockwise and negative values are clockwise.
Pitch	Rotation around the Y-axis, positive values are counter-clockwise and negative values are clockwise.
Yaw	Rotation around the Z-axis, positive values are counter-clockwise and negative values are clockwise.
α	The horizontal angle in radians from origo to a point in a plane when observed from a point outside the plane (see the picture below), positive values are to the left and negative values are to the right in the plane.
β	The vertical angle in radians from origo to a point in a plane when observed from a point outside the plane (see the picture below), positive values are down and negative values are up in the plane.

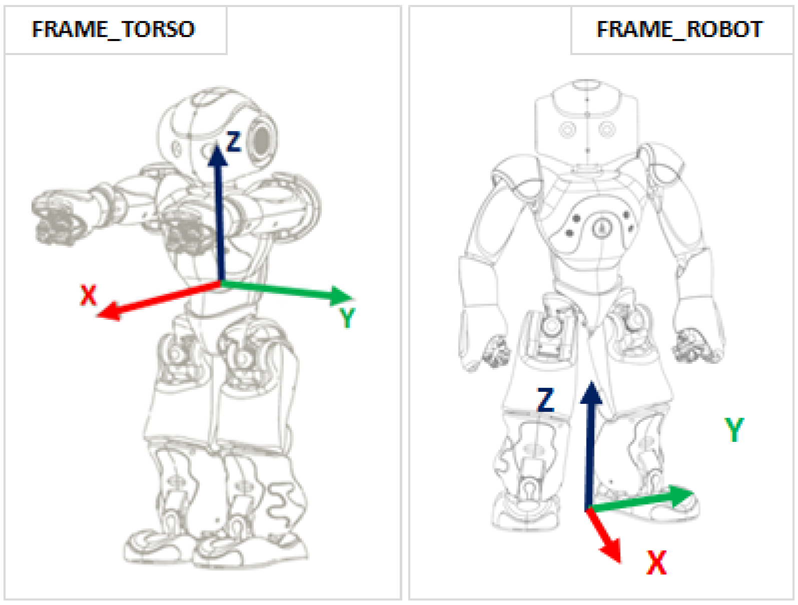

Figure 4.2: Coordinate axis.

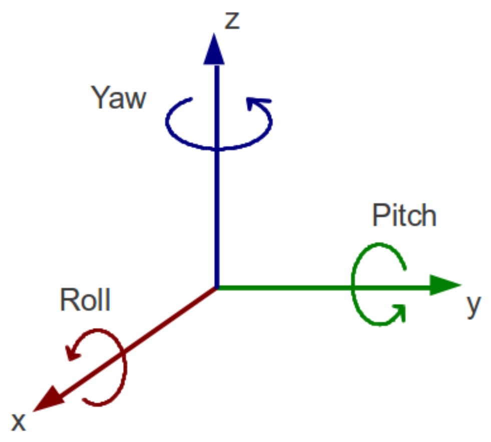

Figure 4.3: Roll, Pitch and Yaw angle visualization.

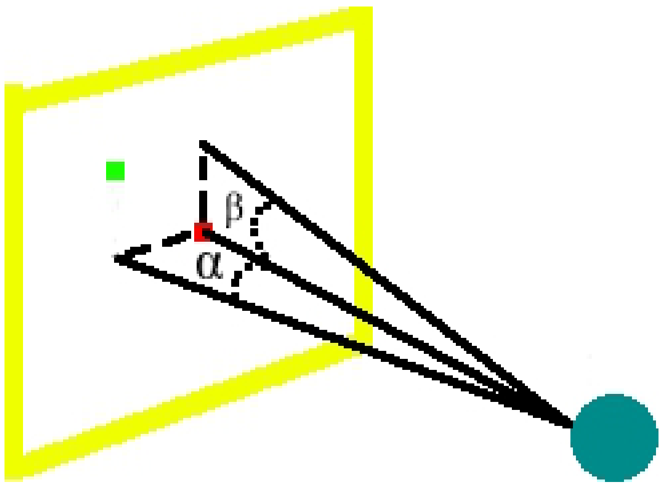

Figure 4.4: α and β visualization.

Simulator

Why use a simulator?

As you might have noticed when developing on the Nao robots (if you are reading this before you'll have to take our word for it), it can take quite some effort to always test code on the physical robots. For example, there has to be a person present to catch the Nao robot in case that it falls over and there is no way to speed up the program to get to the part you actually want to test (since that would require to speed up the physical world). And this is just for a quite small humanoid robot. Imagine that you are writing code for Boston Dynamic's BigDog (`http://www.bostondynamics.com/robot_bigdog.html`) or one of Witas UAV Platform (a unmanned helicopter `http://www.ida.liu.se/divisions/aiics/aiicssite/uastech/photos.en.shtml`). For the BigDog you might have to first build a setting for the robot and for the UAV you would have to take it where there are no risk of crashing into people. Obviously, that takes some effort. Moreover, it could prove quite costly to do those tests as well. Therefore, it is quite common to run preliminary tests in a simulator before testing it on the real robots to rid the programs of bugs and to test concepts. Hence, using a simulator can **lower the effort and cost of development**.

What is a simulator?

Now that we have covered why it is a good idea to use a simulator, we naturally have to ask what it is. Essentially, **a simulator is something that simulates an environment and/or agents**. When used in robotics this usually means that the simulator emulates the real world environment the robot is supposed to work in as well as the robot itself. For example, the Nao robot could have a simulator which environment is a football field and the agent is a Nao robot.

A simulator can also have different levels of details that are simulated. In our example with the Nao robot it could vary from a simple two dimensional simulation where the robot, for example, can not fall to a complex three dimensional simulation which tries to simulate the physics of the real world. Naturally, the three dimensional simulation can simulate more than the two dimensional (potentially, it could simulate if the robot will fall when some of its limbs are moved). However, the drawback of the more complex simulation is that it requires a lot more computation power than the simple one. In fact it can be too heavy for a modern computer if the goal is to simulate a whole team of the robots. Therefore, the two dimensional simulator might be a better choice if the aim is to simulate how the decision making works for the whole team even if it does not take into consideration that the robot can fall.

Why not only a simulator?

Imagine having a three dimensional simulator that simulates the physics, the environment that the robot is going to be in and the robot as well. Furthermore, the simulator is quite exact so you are fairly sure that the code that controls the robot in the simulator will be able to control the real robot as well. Why would we not do all the testing in the simulator since it is cheaper and just ignore the real robot until it is deployed?

In theory it is interesting idea. However, for a robot that acts in the real world the simulator would have to be able to simulate all aspects of the world if we are going to be sure that it will behave the same in the real world as in the simulator. Unfortunately, this is so complex that we would not be able to simulate it in a reasonable time. Hence, simulations are usually simplified so that they can be computed within a reasonable time. Therefore, **a simulator only captures parts of the real world**. This leads to the fact that any code tested in a simulator is not guaranteed to work the same way in the real world. Hence, the code should always be tested on the physical robot before its considered tested.

Simulators for Nao

There are a few simulators that is widely used for simulating Nao in a three dimensional world. For some time there was only one available, Webots. However, Aldebaran released the meshes for Nao a couple of years ago and since then a couple of more simulator has been released. For example, VRep is a simulator that now supports the Nao robot to some degree.

In addition to all the three dimensional simulators there are also a couple of two dimensional simulator available. However, since a two dimensional simulator is relatively easy to implement there are quite a few that writes their own. For example, in our own lab we have our own two dimensional simulator for simulating a soccer game.