Visual Tracking of Humanoid Robots

University of Amsterdam

Honoursproject

Authors:
Nick de Wolf, 10165401
Steve Nowee, 10183914

Supervisor:
Dr. Arnoud Visser

January 22, 2013

Acknowledgements

We would like to thank our supervisor, Arnoud Visser, and also Bas Terwijn and Ninghang Hu for helping us throughout this project. We would like to thank Raquel Fernandez also, for coordinating the honours program and spending her time on it.

Contents

Introduction

The Project

This project is about detecting and tracking humans, as well as Nao's at the same time. The Nao is a humanoid robot, that is also used in the soccer matches of the RoboCup. This detection will be executed with a fisheye camera that hangs above a Nao soccer field. This setting can be seen as a simulation of a RoboCup soccermatch. During such a match, Nao's and referees will be moving on and next to the field. So the main goal of the project is to tell the difference between Nao's and humans, and then to track all of the detected moving and non-moving Nao's and humans.

The Purpose

The Nao's that compete in the RoboCup, have an internal localization module. Although, this module can be inaccurate. The data, resulting from the tracking with a fisheye camera, can be used to cross validate with the internal localization of the Nao's. With such a cross-validation, the Nao can get an affirmation as to where it is, or the Nao may find that it's position is incorrect. By combining the external localization with the fisheye camera and the internal localization of the Nao, improvements to the Nao's own internal localization can be made.

Getting Started

The Search Space and Camera

The search space in this project, will be the soccer field. This field is green with white markings on it and it measures four meters in width and six meters in length. The camera itself is suspended approximately 2.8 meters in height. The camera is not entirely above the center of the field, it is shifted vertically along the field by 15 centimeters. Looking at Figure 1, below, the camera is shifted towards the upper line of the field.

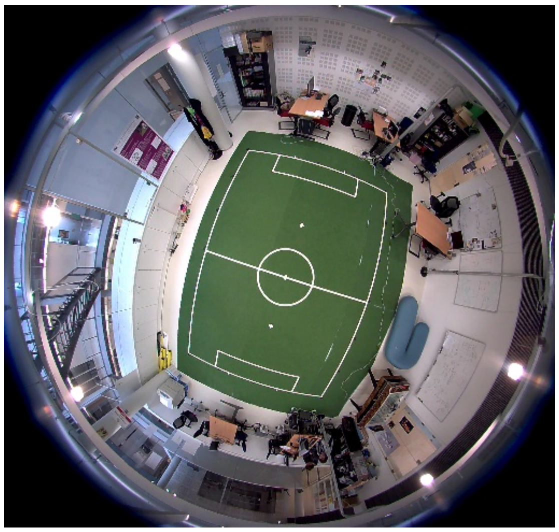

Figure 1: Image of the search space, the field.

The fisheye camera is built, using multiple lenses to distort a larger amount of light rays that fall onto the outermost lens. Because of these multiple distortions a very large area can be recorded, using just one camera. Although, these distortions bring some complications as well. The straight lines of the soccer field, for example, will become curved, because of all the distortions that the multiple lenses cause. The definition of localization is knowing where something is. To accomplish this goal, a coordinate system has to be created for the field. This cannot be easily achieved with curved lines, however. That is why the curved lines should be straightened. This will be discussed in more detail in a further paragraph.

Calibrating the Camera

Calibration is a comparison between measurements, where one of the measurements is known and is considered correct (the standard) and the other measurement is considered wrong and is the to-be-tested measurement. The goal of calibration is to make the differences between the measurements as small as possible. In the case of the fisheye camera, straight lines will be considered as the standard and the curved lines (how the camera perceives the world) will be considered the to-be-tested measurement. The calibration process has mutiple stages, the intrinsic calibration and the extrinsic calibration. To succesfully perform the calibration, one has to start with the intrinsic calibration.

Intrinsic Calibration

The intrinsic calibration is a calibration of the camera itself, that does not have anything to do with the search space. It is a way of undistorting the images

made by the fisheye camera. In other words, straightening the curved lines in the recorded frames. This is done, using a chessboard pattern. The pattern is printed out on an A0 sized sheet of paper. The measurements, e.g. the size of its squares, are known.
Next, this pattern is recorded. This is done in such a way, that when looking at all the recorded frames the pattern has covered almost every part of the perceivable area, at different angles. Because the measurements are known, one can calculate the transformation that is needed, to transform the distorded chessboard pattern into its original form. This is why the pattern has to have covered as much of the perceivable area as possible, and at different angles. Because then the optimal tranformation can be found to undistort the images. Below in Figure 2 is an example of an original image and the undistorted version of that image.

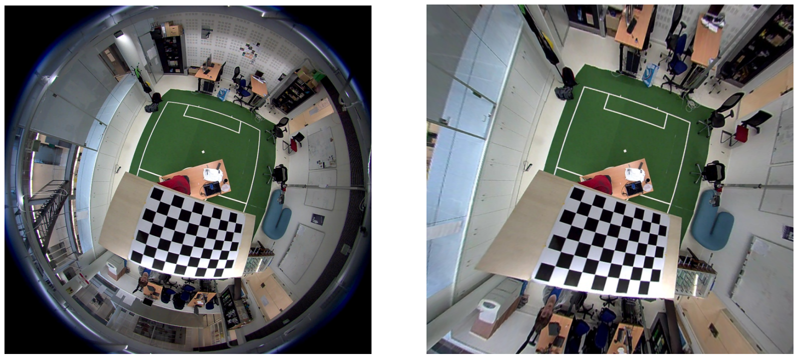
Figure 2: Distorted image (left) and the undistorted version of that image (right).

As can be seen in the above images, all the curved lines in the distorted image do indeed get straightened by the intrinsic calibration. A side effect of straightening the lines, is that the image looks to be somewhat stretched out. But the field itself does get more right-angled, which is the goal of the process.

Extrinsic Calibration

This part of the calibration of the camera is very important for the localization process, and works together with the intrinsic calibration. Note, that after calibrating the camera extrinsically, the camera must remain in the exact same position. Briefly, the extrinsic calibration makes a coordinate system in the images. Every frame the camera makes can be described using two dimensions, which are the horizontal and vertical pixels in the image. To make the localization understandable, these two dimensional positions should be transformed into a three dimensional coordinate system of our world.
To establish such a transformation, one must pair some of the two dimensional coordinate points, with the corresponding three dimensional coordinate points.

The space in which the localization takes place, in this case the soccer field, has to be marked. The best way to mark the field, is at a one meter interval. Starting in one corner of the field, multiple tape markers are placed every meter, around the edges of the field. These tape markers will also be placed over the middle line of the soccer field, so there will be a sufficient amount of paired points. And last, some height points have to be marked. We have used some pieces of furniture around the lab for these points, like the elbow rest of a chair and a peg. All of these marked points have to be noted, three dimensionally, in a text file. Then a program can be used to click every marked point in the field, which creates another text file but now with the two dimensional coordinates of the points clicked. It is important that the marked points are clicked in the same sequence as they where noted three dimensionally. Otherwise, the points do not correspond to each other. In combination with the intrinsic calibration, this creates a straight three dimensional coordinate system that can be used for localization. Because this coordinate system is created for the specific position the camera is in, it is important that the camera should not shift after completing this section.

Below, in Figure 3, are some images that illustrate the extrinsic calibration process. The first images are the result of a first try.

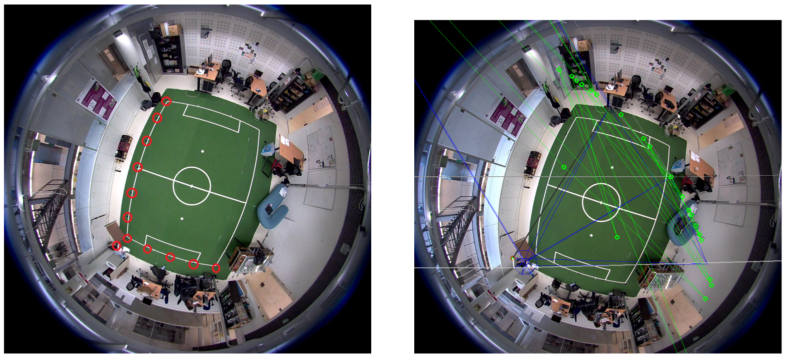

Figure 3: Image of the field with markers at the edges (left), the result of using the markers in the left image for the extrinsic calibration (right).

The red circles in the left image clarify the locations of the markers, of which eleven are on the ground and one is above the ground. The right image shows the result of the calibration process, using the extrinsic calibration that follows from using these particular markers. As can be concluded from the right image, these markers do not result in a correct calibration.
Below, in Figure 4, are two images of a second try, but this time with more markers.

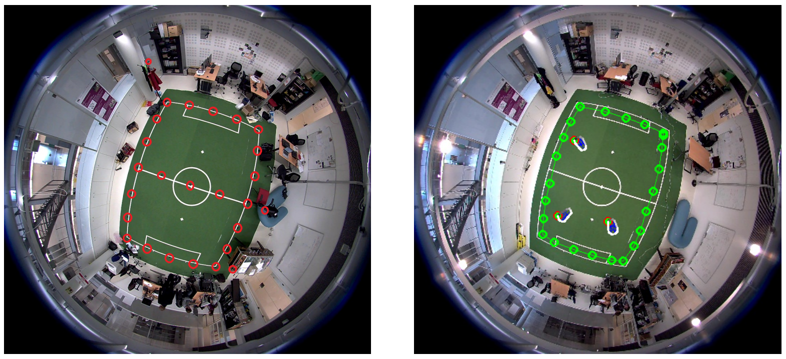

Figure 4: Image of the field with a larger amount of markers at the edges (left), the result of using the markers in the left image for the extrinsic calibration (right).

In this second try, more markers were put on the ground, as can be seen. But also, there are more markers up in the air. This larger amount of markers is needed to create an (accurate) coordinate system. The result, following from using these markers, is visibly better than in the first try. There are three examples of a Nao template in the image. The origin of the coordinate system depends on the order in which the markers are noted in the text file. In this project, the lower left corner is the origin of the coordinate system.

Background Model

To make the localization and recognition work correctly, a background model has to be made. This is done by taking a set of snapshots with the camera. These snapshots will then be used to produce the background model. This background model will be made, by taking the average of all the pixels from each picture. The background model will be used as a reference for the detection and localization of the program. As it records what color each pixel should have. So every value that diverges from the value it should have, will be seen as an object that is on the field. The background model is an important part of the localization, because it is used to calculate the possibility of something being on the field. So it is important that the field is completely empty and preferably the space around the field as well.

Prior

Before the camera can be used for localization, a prior has to be made. A prior is, in short, the area where in this case the Nao's can walk, and thus where the localization has to take place. This prior is created by clicking around the edges of the area in which you want the localization to take place. The selection that follows from this process is converted into a text file, which is used later on. In our case, the prior is the soccer field itself. Although, we have seen some noise around the edges when the prior is exactly on the lines of the soccer field. The definition of noise, in this case, is when something is localized but there is nothing at that location. By making the prior a little bit smaller than the soccer field itself, this noise is removed. The Nao's that walk outside of the smaller prior will still be found, if the prior is not too small. The Nao's have a certain height and if it stands just outside the prior but inside the field, the camera will still detect it as if it was standing on the edge of the prior.

Making the Distinguishment

Used Software

The software that has been used in the project, has originated from the Accompany project [1]. In this project, the software is more focused on the localization of humans. To track the Nao's instead of humans, some changes had to be made. To track a human, a human's measurements have to be known. So to track a Nao, its measurements have to measured first. The software tracks entities that have approximately the same size as the measurements that are given as input. These measurements are stored in an XML-file, *params.xml*. A part of this file can be seen below.

```
<personHeight>600</personHeight>
<wg>230</wg>
<wm>280</wm>
<wt>140</wt>
<midRatio>.83</midRatio>
<persDist>70</persDist>
```

Some of the XML tags above can be logically derived, like *personHeight* which is the average height of the to be detected entity. The tags *wg*,*wm* and *wt* are respectively, the width on the ground, the width at the middle and the width of the head. The last two tags, *midRatio* and *persDist*, correspond respectively to the ratio between shoulderheight and total height and the minimal distance between two detected entities. These parameters can be adjusted to the measurements of a Nao, after which the Nao's are tracked. The measurements you can see above, are already of a Nao. The problem with this, is that a human will be detected as multiple Nao's. If, during a RoboCup soccer match, a referee walks onto the field, you do not want all those non-existing Nao's to be tracked and give a lot of false detections. Possible solutions to solve this problem, will be discussed later on. Below are three images of how people get recognized as multiple Nao's.

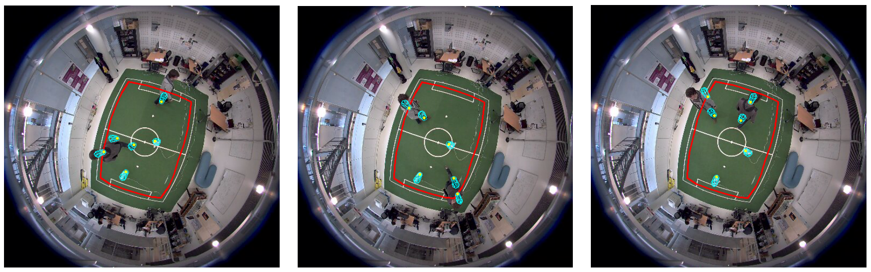

Figure 5: Three examples of how people get recognized as multiple Nao's.

As can be seen in the above images, parts of a human body get recognized as a Nao, like legs and arms. But not all of the limbs get detected as Nao's. This is probably because of the minimal distance between two detected entities.

The localization procedure itself is based on the paper: *Fast Bayesian People Detection* [2]. In the following paragraph, this procedure will be explained. The prior, that is created before running the localization, is the area where it is possible to detect something. This can be seen as a set of possible locations for detecting entities. For each of these locations a probability will be calculated. This probability denotes the chance that an entity, of the right parameters, is at a location. This probability is based mostly on the differences with the background model. The highest of these probabilities is chosen and at that location should be an entity that is to be detected. That location is then added to a list of locations where something exists, called *existing*. Depending on the number of entities in the prior, which can be either given manually, or calculated by chance, the program will search on. The locations that are listed in *existing* will not be used in the calculation of the new probabilities. So the next best probable location, where something can be detected, will be found. This process runs untill it reaches the maximum number of entities to be found.
This process is repeated for every frame that is taken by the camera. This way, the movement of the to be found entities can be tracked, as long as a large enough number of frames is taken each second.

Theoretical Approach

Returning to the main goal of the project, the simultaneous tracking of people and Nao's. The code, as it is, will not achieve this goal, as was stated before. If only one template of parameters is used, only one type of entities can be detected. Now that the environment and code have been explained to some extent, we can elaborate on solving this problem. In this section, a theoretical solution will be discussed.

In the previous section was explained that in each recorded frame all entities of one sort would be detected, were it people or robots. But instead of finding that one type of entities, an extra step should be built in. This extra step involves a second template of parameters. Following is an explanation for the simultaneous detection in one single frame. This should be considered for every frame that is recorded by the camera. Firstly, the people will have to be detected in the frame. As mentioned before, all the locations where someone is detected will be set apart in a list (*existing*). So these locations will not interfere with the potential other locations. If all the people are found though, instead of moving on to the next recorded frame, the detection of the Nao's should start. The idea is that the Nao's will be tracked with the same technique as the people. Although, another parameter template will be used and the list *existing* should be passed on for the detection of the Nao's. This way, a mask will cover all the areas where humans were detected and thus will not be detected as multiple Nao's. Using this method, the locations of only the Nao's can be extracted. These locations can be used as necessary, e.g. for cross-validation with the internal localization of the Nao's.

Another option that was thought about, is making the distinguishment by color. Since the field is always green, any change of color on the field should be easily distinguishable. Nao's are mostly white. So by searching in the field for a shape, about the size of a Nao or human, a distinguishment can be made by getting the average color of that shape. If the average is close enough to white, it will be identified as a Nao. The official referees for the RoboCup wear a black shirt. So a referee can be distinguished if the average of the upper part of the shape is close enough to black. These thresholds will have to be determined for each individual pairing of field and camera. This is due to the fact that there will be different lighting, different fields and the environment will not always be the same.

Practical Implementation

To be able to localize different sized objects, different parameter files will have to be loaded after each other. So the first thing that should be changed in the code, is the part where the parameter file is loaded. After looking through the source code, it was concluded that the part where such a file is loaded, is located in multiple places. It is basically embedded into the code at more than one location, making it hard to load extra parameter files or to make any changes to it. This made the first idea, of having two parameter files, very hard to accomplish.
The part, where we thought the parameter file was loaded, can be seen below.

```
params_file = path + "/" + "params.xml";
intrinsic_file = path + "/" + "camera_intrinsic.xml";
extrinsic_file = path + "/" + "camera_extrinsic.xml";

loadCalibrations(params_file.c_str(), intrinsic_file.c_str(),
  extrinsic_file.c_str());
```

This piece of code is embedded in the main function of the localization program. We tried adding a *params_file* with the measurements of a Nao. But even after changing and removing the function call *loadCalibrations* the program still loaded a standard parameter file. This was the reason for our thoughts of another loading location, which we tried to look for but never found.

Conclusion

The conclusion of this project would be, that we had too little experience and knowledge. Many times, we were severely slowed down, or halted in our progress. In the first weeks we had a lot of obstacles following from the calibration. This slowed the progess down and thus we got behind on the schedule a couple of weeks into the project (e.g. manual not working correctly). In the end, this time could not be made up later on in the project. So in other words, a lot of time was wasted on the calibration, due to the lack of knowledge and the lack of documentation. We had to mail and meet with Ninghang Hu (one of the writers of the program that was used), to get problems fixed or to get an idea on where to look for a solution. Though, this took quite some time. Still, we learned a lot about the calibration process this way, as we were busy with the code for quite some time. Although we did understand quite a bit about the code, and which file was used when, we still stranded at some point.

Collaboration

The collaboration between the two of us was quite good. We had the advantage that we have the same schedule. So when we wanted to, we could both work on the project after our shared lectures. We have not had any real cooperation issues. The thing we probably should have done more consistently, was keeping contact with the supervisor of our project, Arnoud Visser. He might have been able to help us sometimes, if we had contacted him. Another problem we found after talking with Arnoud, is that our labjournal was not detailed enough in the beginning. Also, we did not always document all the errors we encountered, as we thought it would not add anything of value to the journal. But he made us see that writing down your mistakes is a good thing, as you can use it to see your previous thought processes later on. We tried to apply the advises given to the labjournal.
This journal can be found at: `https://sites.google.com/site/2012robottracking/`. Though, this journal was made in Dutch.

Project Course

The project could be divided into several stages. It was partly installing, preparing and setting up the system. After this preparatory stage, we could start looking at the real goal of the project and how we were going to achieve it. Both of the stages had some setbacks. The main problem we had is probably, that we did not have a lot of experience with some of the used packages and programming languages like ROS and C++. We could work through it to some degree, but it still left a certain black box feeling in certain parts. Some parts of the code worked, but we did not fully understand why, even after some explanation. Though, some of the code was clear. In the preparatory stage, we had a lot of trouble with the calibration of the camera. In theory, this was just following steps from a manual. But since the software we were working with was still experimental, some of these steps did not work the way they were supposed to. The commands that were in these steps produced errors that we could not explain, which might originate from the fact that we did not fully understood some of the code. Eventually, these problems were solved with some help. But this still resulted in some backlog. We finished this stage later than we would have liked.
The second stage, let us call it the experimenting stage, started in the second week of the second period. We realized that we had a lot less time than we would have liked for this stage. This was because the curricular courses that we had in the second period, were a lot more demanding. So we lost time in the preparatory stage with the calibration and we had some catching up to do, with less time than planned. We tried to come together as much as possible, through the other deadlines. Mostly, we were looking through the code and just trying to grasp the idea of the implementations. We also had some meetings with one of the programmers from the Accompany Project, Ninhang Hu, who had also helped us with parts of the calibration before. He tried to walk us through the code, explaining it. We also had contact with Bas Terwijn and Arnoud, talking about possibilities of achieving the goal of detecting multiple different shapes. One of the solutions offered, was the idea of multiple templates. We started looking more purposefully at the code, for a way to implement that idea of the multiple templates. This is the point where we stranded. We started by trying to load two templates after each other, to see if it would be possible. Though the piece of code where we thought the template was loaded, actually did nothing. As said before, we changed that specific line, but the same standard template was loaded. And after that, we deleted the whole line and still a standard template was loaded. This loading of the parameter file was happening somewhere else, and we did not know where. We tried looking for it, but did not succeed. At this point, about three weeks before the deadline of the report, we had to start with the report, or we might not be able to finish it. So the priorities shifted a bit. We still looked, but we never found the location where the file was loaded.

References

[1] Accompany Project Website, http://www.accompanyproject.eu/.

[2] G. Englebienne, B.J.A. Kröse, *Fast Bayesian People Detection.* University of Amsterdam, 2010.

Appendix A: User's Guide

General Options

This section elaborates on the multiple actions, that can be executed after the complete installation of the system and calibration of the camera.

The only thing that can be done after the installation of the system, but before the calibration of the camera, is watching the live feed of the camera. Also, this live feed can be saved to an .AVI file or published directly to a ROS thread. If the video feed is directly published to a ROS thread, the localization can be performed real-time. The corresponding commands, to these actions, can be found at the end of the section 'Dependencies'.

After the calibration of the camera, the localization itself can be executed. This can be done in two ways. Either, the live video feed is directly published to a ROS thread and the localization is performed on this live feed. Or a video is recorded with the camera and the localization is performed on that file. Both of these methods for localization, can be found in the section 'Localization'.

Preparing the system

This section will describe how to build and use the UvA modules of the Accompany Project. This guide has been made for installation on Ubuntu 11.10 (Oneiric). In this guide, the preparations of the camera are explained step by step, specific for the desktop computer and the fisheye camera that are in the Robolab, at the UvA.

Dependencies

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. But in this case it will be used for the UvA module. ROS can be downloaded and installed by following the instructions on their wiki, in this case 'ros-electric-desktop-full' will be needed and its instructions are found at: `http://www.ros.org/wiki/electric/Installation`.

VXL is a multi-platform collection of C++ software libraries for Computer Vision and Image Understanding. The version that was used in this project is *v1.17.0*. It can be acquired at: `http://sourceforge.net/projects/vxl/files/vxl/1.17/vxl-1.17.0.zip/download`.

To install VXL, execute the following instructions (Be aware, this might take some time):

1. Open a terminal, in the folder you installed VXL, and type:
2. unzip vxl-1.17.0.zip
3. cd vxl-1.17.0/
4. mkdir build
5. cd build
6. cmake .. -DBUILD_BRL=OFF
7. make -j 4
8. sudo make install

For those new to Ubuntu or terminals, the above commands translate to:
First you unzip te file you have downloaded. You then enter the map that was made by unzipping the file. You then make a new folder 'build' using 'mkdir build' and enter it using 'cd build'. Using cmake will create an executable file from source code files. You then execute this file, and perform the installation.
The software package **ubuntu-restricted-extras** allows the user to install essential software that is not already included due to legal or copyright reasons. It can be aquired by doing the following:

1. Open a terminal and type:
2. sudo apt-get -y install ubuntu-restricted-extras

The script 'installUvA.sh' has to be executed to get the remaining dependencies. This script will get serveral of the dependencies at once, and put them in their own server. It can be invoked by doing the following:

1. Open a terminal and type:
2. ./ installUvA.sh

After this, all dependencies should have been installed. To test if the camera is working as well, one of the following actions can be executed.

1. Open a terminal and type:
2. gst-launch rtspsrc location=rtsp://admin:admin@192.168.1.222:8554/CH001.sdp ! decodebin ! videoscale ! videorate ! video/x-raw-yuv, width=512, height=486, framerate=15/1 ! xvimagesink sync=false

This will bring up a window that shows a live feed of the camera. The part 'rtsp-src location' can also be referred to as the camera's IP, in this case 192.168.1.222:8554, is the IP of the camera in the Robolab. If this guide were to be used for another camera, with another IP-adress, this part should be changed to the right IP-adress. The width and height variables will determine what the resolution of these frames will be. The framerate 15/1 means, that it will show 15 frames per second.

1. Open a terminal and type:
2. gst-launch rtspsrc location=rtsp://admin:admin@192.168.1.222:8554/CH001.sdp ! decodebin ! videoscale ! videorate ! video/x-raw-yuv, width=512, height=486, framerate=15/1 ! jpegenc ! avimux ! filesink location=video.avi

This will record the live feed of the camera to a file. The part 'rtspsrc location' can also be referred to as the camera's IP, in this case 192.168.1.222:8554, is the IP of the camera in the Robolab. If this guide were to be used for another camera, with another IP-adress, this part should be changed to the right IP-adress. The width and height variables will determine what the resolution of these frames will be. The framerate 15/1 means, that it will record 15 frames per second. The part 'filesink location' is the location and name of where the file will be saved. Change this name before running the code, because if the name already exists, the existing file will be overwritten.

1. Open a terminal and type:
2. export GSCAM_CONFIG="rtspsrc location=rtsp://admin:admin@192.168.1.222:8554/CH001.sdp ! decodebin ! videoscale ! videorate ! video/x-raw-yuv, width=512, height=486, framerate=15/1 ! ffmpegcolorspace" rosrun gscam gscam – sync false

This will publish the live feed of the camera to a ros thread. The part 'rtspsrc location' can also be referred to as the camera's IP, in this case 192.168.1.222:8554, is the IP of the camera in the Robolab. If this guide were to be used for another camera, with another IP-adress, this part should be changed to the right IP-adress. The width and height variables will determine what the resolution of these frames will be. The framerate 15/1 means, that it will pubilsh 15 frames per second. The command 'rosrun gscam gscam – sync false' will publish the feed to a ros thread.

Intrinsic Calibration

For the intrinsic calibration, a checkerboard pattern is needed. As is known, the lines of a checkerboard are straight. So the program can calculate a transformation from the distorted pattern, to the original pattern it knows. Such a transformation can be found for all points in the view of the camera, as long as it has enough images of the checkerboard placed throughout the camera's view. The checkerboard has to have black and white blocks, and preferably a large white border around the pattern itself. A good checkerboard pattern can be downloaded at: `http://www.ros.org/wiki/camera_calibration/`

`Tutorials/MonocularCalibration?action=AttachFile&do=view&target=check-108.pdf`. The best thing to do is to print this in an A0 format and then secure it to a board. This board should not be able to bend easily, to ensure the lines stay straight. So a thick board is recommended.

Intrinsic calibration can be started with the following actions:

1. Open a terminal and type:
2. export GSCAM_CONFIG="rtspsrc location=rtsp://admin:admin@192.168.1.222:8554/CH001.sdp ! decodebin ! videoscale ! videorate ! video/x-raw-yuv, width=512, height=486, framerate=15/1 ! ffmpegcolorspace"
3. rosrun gscam gscam -s 0
4. roscd accompany_static_camera_localisation/res/calib_frames
5. rosrun image_view image_view image:=/gscam/image_raw
6. rosrun accompany_static_camera_localisation image_saver -n 5000 -p ./ -t /gscam/image_raw

This will start recording images. While the program is recording, the checkerboard should be slowly moved around in the view of the camera. The board with the checkboard should also be tilted at different locations, and it should reach all parts of the camera's view. If done correctly, there should be a lot of similiar images at about the same location in each picture. To remove these abundant pictures, perform the following actions.

1. Open a terminal and type:
2. ./modPics.sh
3. mkdir OLD
4. mv *jpg_OLD OLD/
5. eog *.jpg

This makes a new folder, then eog *.jpg is used to check the pictures.
After that step an image list has to be created. This is done by the following actions:

1. Open a terminal and type:
2. roscd accompany_static_camera_localisation/res/calib_frames
3. rosrun accompany_static_camera_localisation create_calibration_list calib_list.xml *.jpg
4. cat calib_list.xml

5. rosrun accompany_static_camera_localisation calibration_intrinsic -w 6 -h 8 -m ../mask_large.png -k 5 -a 1 -rm -p -zt -o ../camera_intrinsic.xml calib_list.xml

After these steps, it should be possible to take a distorted image, made by the camera, and turn it into a corresponding straightened image. To show if the intrisic calibration was successful and to see if nothing went wrong, the following actions can be performed:

1. roscd accompany_static_camera_localisation/res/calib_frames
2. rosrun accompany_static_camera_localisation undistortion_test -s [image] -i [camera_intrinsic] -f

In this last command, the *[image]* part should be substituted for the name of a distorted image. The *[camera_intrinsic]* part should be substituted for the name of the XML file, that was created with the intrinsic calibration above. Normally, this file is just called 'camera_intrinsic.xml'.

Extrinsic Calibration

A preparation that has to be made to perform extrinsic calibration, is to put markers on the floor and possibly even on objects and walls. These markers should be placed 1 meter apart. Choosing a smaller distance is also possible, but since all of these points have to be clicked on, it could be time consuming if the chosen distance is too small. A file called *points3D.txt*, should be made. The markers represent the coordinate system of our world. Every marker that is put down, should be rewritten into the text file in an *(X, Y, Z)*-system of coordinates. The coordinates of these markers should be noted according to these axes in millimeters, an example is:

```
0,0,0
1000,0,0
2000,0,0
0,1000,0
```

The order in which these points are put into the text file for 3D points, will also be the order in which the 2D-points will have to be selected in the image. Executing the following code will start the program to select these markers in the 2D-image:

1. Open a terminal and type:
2. roscd accompany_static_camera_localisation/scripts
3. ./fisheye_marker_images.sh

By right-clicking a frame, it can be saved.

1. roscd accompany_static_camera_localisation/res/marker/
2. rosrun accompany_static_camera_localisation create_background_list marker_list.txt *.jpg

The above will create an image list of the markers.

1. roscd accompany_static_camera_localisation/res
2. rosrun accompany_static_camera_localisation annotate_image_points marker/marker_list.txt points2D.txt

This is the actual part where the locations of the markers will get selected. Again, please make sure that the order of the points selected, matches with the order in the file *points3D.txt*. Points can be selected by right clicking them in the image viewer. To make the clicking more precise, you can press 'SHIFT' and '+' to zoom in, up to four times. If you try to zoom in more than four times, the image viewer will crash and you have to start over.
Once this is done, press 'ENTER' to save the results.
Now that these points are annotated, a file called points2D.txt has been made. For the algorithm to work correctly, the file points3D.txt is also needed in de /res folder. This file can be moved, by typing the following command in a terminal:

1. cp [location]/points3D.txt .

The final step in the extrinsic calibration is made by typing the following command in a terminal:

1. rosrun accompany_static_camera_localisation calibration_extrinsic -i camera_intrinsic.xml -o camera_extrinsic.xml -p points2D.txt -q points3D.txt

After this params.xml will be created, and now this file can also be modified. Interesting things to change here are the parameters for the size of humans or Nao's, and the scale in which you would want everything to be recorded from that point onwards.

Building the background model

To build a background model, some scripts have to be executed. These scripts can be accessed performing the following actions:

1. Open a terminal and type:
2. roscd accompany_static_camera_localisation/scripts/
3. export GSCAM_CONFIG="rtspsrc location=rtsp://admin:admin@192.168.1.222:8554/CH001.sdp ! decodebin ! videoscale ! videorate ! video/x-raw-yuv, width=512, height=486, framerate=15/1 ! ffmpegcolorspace"
4. ./fisheye_capture_background_images.sh

The viewer of the live-feed will be opened. To save an image as a background frame, right click it. About 25 frames will be enough.
To then build the background model, type the following code into a terminal, but note that this should be done in the same folder. ./create_background_model.sh

Creating the prior

The prior is the area in which the localization will be performed. It can be considered the area where things have to be tracked. The following actions have to be executed to create the prior:

1. Open a terminal and type:
2. roscd accompany_static_camera_localisation/res
3. rosrun accompany_static_camera_localisation create_prior -l background_images/background_list.txt -p params.xml -o prior.txt -i camera_intrinsic.xml -e camera_extrinsic.xml

Then right click on points of the image. A line will be made, drawn between the points and thus marking your prior. Press ENTER to finish the prior once you think it is good enough.
Now that everything is done, you will probably want to see if it worked correctly or not. The following actions will show you a distorted image, and in the image you can see the prior as well. If anything does not look right, look for the step that seems to cause the problem. When you find this wrong step, try to redo it and check the calibration again afterwards. The actions to check the calibration are as follows:

1. Open a terminal and type:
2. roscd accompany_static_camera_localisation/res
3. rosrun accompany_static_camera_localisation annotate_pos -l background_images/background_list.txt -p params.xml -r prior.txt -i camera_intrinsic.xml -e camera_extrinsic.xml -a temp.txt

If everything looks fine, the calibration of the camera is now completed. The only thing that has to be changed, is the background model. The smartest thing is to do this everytime something in the background has been changed, or even every time you run the localization. This way, you reduce the amount of noise detected.

Localization

The last step is to use localization, this is done by the following actions:

1. Open a terminal and type:
2. export GSCAM_CONFIG="rtspsrc location=rtsp://admin:admin@192.168.1.222:8554/CH001.sdp ! decodebin ! videoscale ! videorate ! video/x-raw-yuv, width=512, height=486, framerate=15/1 ! ffmpegcolorspace"

3. roslaunch accompany_static_camera_localisation fisheye_localization.launch

Localization can also be performed on a video file. This can be done by opening the folder in which the video is located and opening one or serveral terminals. When using one terminal add a '&' after each line of code, without the quotes. This will ensure the parts will run in the background and you can continue using the terminal. The lines of code are as followed:

1. rosrun accompany_static_camera_localisation video_publisher -s 0.3 -i examplevideo.avi
2. rosrun accompany_static_camera_localisation camera_localization -v -p.
3. rostopic echo /humanLocations

If roscore is not actived the localization will not work. This can be solved by starting roscore manually before executing the code. This can be done by typing 'roscore' in a terminal.