Technical University of Denmark

Mikkel Viager

Detection and tracking of people and moving objects

Bachelor's Thesis, January 2011

DTU Electrical Engineering
Department of Electrical Engineering

Mikkel Viager

Detection and tracking of people and moving objects

Bachelor’s Thesis, January 2011

Detection and tracking of people and moving objects

Report written by:
Mikkel Viager

Advisors:
Jens Christian Andersen
Nils Axel Andersen

DTU Electrical Engineering
Technical University of Denmark
2800 Kgs. Lyngby
Denmark

studieadministration@elektro.dtu.dk

Project period:	30/08/2010 - 24/01/ 2011
ECTS:	20 Points
Education:	B. Science
Field:	Electrical Engineering
Class:	Public
Remarks:	This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.
Copyrights:	© Mikkel Viager, 2010/2011

[illegible]

Abstract

The goal of this project has been to develop a software solution capable of detecting and tracking humans in the immediate environment of mobile robots. Limitations are imposed: only a single laser scanner (a.k.a. range finder) may be used and the final product must be compatible with the Mobotware framework. In compliance with these demands, the chosen solution has been created as a plugin running directly in the Mobotware framework, analyzing data on shapes in the environment to separate human legs from static objects. Through multiple experiments with use in both simulated and real environments, capabilities and limitations are revealed to provide a good evaluation of reliability. Detection capability is shown to be high in terms of detecting most humans, but tends to wrongfully include leg-shaped static objects as well. Tracking capability is shown to be highly dependable on both environmental noise conditions and the scan rate of the laser scanner; this leads to a review showing how good results can be expected from the most commonly used scan rates. The project concludes with two examples of implementation in robotics control, utilizing both detection and tracking capabilities and demonstrating the usability of the plugin in real-world applications. With documentation in focus, the developed solution is easily implementable right away and also leaves the option for further expansion and development.

Resumé

Målet med dette projekt har været at udvikle en software-løsning som er i stand til at detektere og spore mennesker i mobile robotters umiddelbare omgivelser. Der er stillet krav om kun at anvende en enkelt laserscanner, samt at det færdige produkt skal være kompatibelt med Mobotware strukturen. Under efterlevelse af disse krav er den valgte løsning udviklet som et plugin der kører direkte i Mobotware, med hvilket der analyseres data som beskriver former i omgivelserne, for at udskille menneskelige ben fra statiske objekter. Gennem adskillige eksperimenter med anvendelse i både simulerede og virkelige miljøer, er evner og begrænsninger fundet for at give et klart billede af pålideligheden. Detektions-evnen vises at være god til at detektere de fleste mennesker, men har en tendens til fejlagtigt at markere ben-formede genstande også. Sporings-evnen vises at være betydeligt afhængig af både støj i omgivelserne, og scan-raten for den anvendte laser scanner, hvilket fører til en gennemgang af hvilke resultater der kan forventes med de mest almindelige scan-rater. Projektet konkluderes med to eksempler på anvendelse i kontrol af mobile robotter, hvorved det under anvendelse af både detektering og sporing demonstreres hvorledes plugin'et kan anvendes i virkelige problemstillinger. Med fokus på dokumentationen er den udviklede løsning klar til implementering med det samme, og inkluderer samtidig en mulighed for videre udvidelse og udvikling.

Table of Contents

1. Introduction

All around the world, robots are being introduced in many new tasks and fields of operation every day. These applications include everything from very simple interactions to highly sophisticated maneuvering. Common to all of them is the need to interact with a somewhat dynamic environment, which often includes the requirement to act properly in direct contact with humans.

Many stationary robots used for industrial purposes are shielded from direct contact with humans and from as many non-predictive influences as possible. This is possible because the robots are only required to complete a single task consecutively, and therefore don't need the ability to counteract changes in the predefined environment. The only behavior needed is simplified down to an emergency stop of the robot as soon as any unexpected event occurs. While this is a viable solution for stationary robots, this approach leads to highly undesirable behavior if used with mobile robots.

Mobile robots need to successfully navigate dynamic environments, in addition to completing certain tasks by interacting with objects in these environments. Prediction of any obstacles and changes in a robot's environment is required to provide a complete set of behavioral rules for the robot to follow. Unfortunately such a set of rules can never be constructed to include actions for every single change in the real world, which is detectable by the many sensors available. To achieve similar but highly simplified behavior, the decisions made are based on only a few key elements in the environment, which are relevant to each action completed. As an example, if a mobile robot is instructed to drive through a door, follow a wall or move from A to B, it searches for predefined shape-features of a doorway in its environment to use as guide marks.

Making correct decisions in controlling mobile robots requires availability of correct and continuously updated environmental data. To improve reliability, it is advantageous to compare and merge data obtained from several types of sensors, to create a more detailed model of the environment.

The prototype "Iromec" shown in Figure 1 is an example of a mobile robot carrying out decision making based directly on human behavior. This robot has been developed to help teach mentally impaired children how to interact with other humans. Providing much simplified and easily readable emotional responses to many kinds of

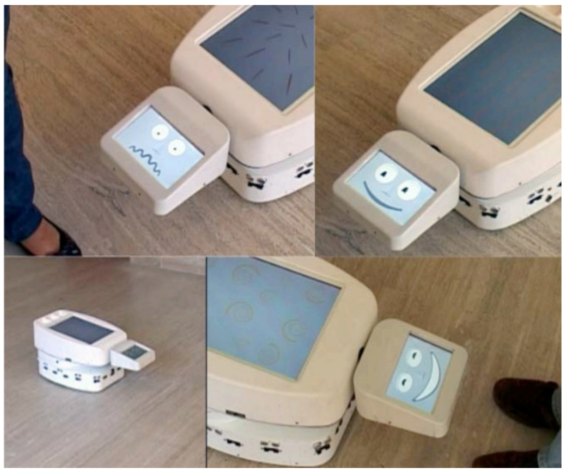


Figure 1: The Iromec Prototype

physical interactions, this robot helps to establish an understanding of how to notice and handle the basic rules of communication [1].

Using both laser scanners and a camera to detect and track its surroundings, the robot can achieve great detail in the information gathered on its surroundings [2].

A common part of a mobile robot development process is the need to create new and platform-specific software for every new robot-type or design. Even though many physical concepts, such as steering and basic structure, are similar among many robots, component-specific drivers may vary, requiring significant restructuring for any pre-made software to be compatible. In an attempt to avoid this problem, an adaptive software package for mobile control called "Mobotware" is being developed at the Institute of Automation and Control, DTU (Technical University of Denmark).

Providing the option to keep high and low level software apart (high level: behavior control. Low level: hardware control), this platform is useable with a great variety of robots. To make a new robot type compatible with Mobotware, only the hardware control has to be re-configured, making all previously developed high level software directly usable. This great advantage allows the continuous addition of new functionalities to the platform, in the form of results from research projects carried out at the institute. With no need to recreate the low level implementation, focus can be kept on development of high level control functionality.

This thesis is the documentation from completion of such a project, with the goal to further develop the capabilities of Mobotware and contribute with useful functionalities to the existing platform.

2. Thesis statement

Development of any new piece of software is always limited and slowed by the necessity to create secondary tools and methods, in order to complete a primary task. Taking up time which could have been used to further develop and improve a main functionality, these side-developments are desirably kept to a minimum.

On the other hand, a large set of secondary tools and methods will allow a main program to achieve more sophisticated behavior, and depend on a broader variety of external factors.

2.1 Problem

For future development of software for the Mobotware platform, it is desired to expand the collection of ready-made and easy-to-use tools in the form of plugins, capable of running directly in the framework. These tools should provide future developers with the ability to use high level sensor-based information, without having to handle the lower level analysis of raw sensor data.

The most desirable kind of information for operating a mobile robot is information on its environment, such as immediate surroundings and localization. With this information it is possible to make better decisions, as they are backed up by analyses of the current situation. Most navigation algorithms for mobile robots don’t distinguish humans from static obstacles when calculating optimal paths from A to B. Instead, the capability to handle disturbances in an otherwise static map can be used to continuously re-evaluate decisions, based on changes in the environment. If data on the location of humans in an environment was directly available in an easy-to use form, such information could be used in several ways to improve both future and existing mapping and localization algorithms. Examples of such uses could be: mapping algorithms completely filtering out humans, localization algorithms ignoring humans when doing positioning, or direct use of human positions to successfully navigate and avoid collisions in highly populated environments.

Such considerations form the ambition behind this project, and the problem at hand can thus be formulated as:

There is no implementation in Mobotware allowing easy-to-use (high-level) localization-information on humans in the immediate environment of mobile robots. Without such information the quality of mapping and localization done in any environments involving humans is far from optimal, due to the disturbances caused by the presence of untreated moving obstacles and inability to act properly on human movement and behavior.

2.2 Limitations

For seamless integration with the existing implementation, a set of limitations has been defined by the researchers and developers in charge of Mobotware:

- Only sensor data from a laser scanner may be used for all detection purposes.
- The solution must be in the form of a "plugin" compatible with the Mobotware platform (portable to other platforms using this framework)
- Predefined standards for internal communication in Mobotware should be used. (callGlobal and XML-like replies as explained in chapter 4.1.5)

In addition, hardware available at the institute in the form of mobile robots and laser scanners are to be used for development and experiments, as these are considered to be the primary clients in future use. This also encourages a laser scanner placement around the average human knee height or below, in order to support the current structure of the robots.

2.3 Solution functionality

To successfully provide a solution to the problem, while staying within the boundaries of the given limitations, the desired primary functionalities are considered to be:

Primary functionality:

- Plugin providing simple (high-level) information on human positions in the environment, based on data from a laser scanner.

Secondary functionality:

- Ability to keep track of, and distinguish between, individual humans.
- Configurability for use under varying environment conditions.
- Easy-to-use: simple to load, configure and run.

Furthermore, examples of use for the final solution are highly desirable as documentation for future users of the plugin, as well as obtaining conclusive results for the project.

2.4 Levels of success

To explain the desired goal of achievement, three acceptable levels of successful outcome are chosen. These are used as guidelines through the development, as well as in the final conclusions in chapter 6.

Basic level of success:

The developed plugin should be capable of determining immediate positions of humans, making this information directly available to other Mobotware processes.

High level of success:

The developed plugin should be capable of determining immediate positions of humans, as well as keep track of each person after initial detection. Configuration for the tracking should be easily adjustable, and all position information should be easily available for direct use in other plugins as well as in XML-like format for use with SMRCL.

Highest level of success:

The developed plugin should be capable of determining immediate positions of humans, as well as keep track of each person after initial detection. Configuration for the tracking should be easily adjustable, and all position information should be easily available for direct use in other plugins as well as in XML-like format for use with SMRCL. As a proof-of-concept and guidelines for future use, examples on how to make use of the provided positional data should be developed, tested and documented.

2.5 Documentation overview

Documentation of both implementation and test results can be found in the following chapters, where concepts and methods are described in detail. Alternative versions of several sub-chapters can be found in Appendix A. These are meant to target C++ programmers and for use with future maintenance or development of the plugin.

Chapter 3 evaluates the given limitations and the impacts hereof.

Chapter 4.1 focuses on the basic concepts in terms of designing the plugin structure.

Chapter 4.2 provides a short overview of the chosen functionalities, while going through the approaches for implementation.

In the final part of the report, test results for use in both simulated and real world environments are presented and evaluated.

Chapter 5.1 briefly introduces the "Small Mobile Robot" platform used during tests.

Chapter 5.2 presents and evaluates results from use of the plugin in a simulated environment.

Chapter 5.3 presents and evaluates results from operation in real-world environments, as well thoughts on experienced noise sources.

Chapter 5.4 provides two implemented cases utilizing the new functionality made available with the plugin, along with the results and final evaluation hereof.

Chapter 5.5 suggests additional options of usability for the plugin in other applications.

Chapter 6 concludes the project by evaluating the implemented functionality, test results, and the overall level of success achieved.

3. Design

Based on the limitations given, considerations are made to determine the options to solve the problem, and whether the limitations cause significant disability to use otherwise advantageous designs.

3.1 Sensors

Mobotware includes implementation allowing the use of both vision (cameras) and 2D laser scanners for obstacle sensing. Since mobile robots are designed in many varieties, the sensor software is made to scale along with the robot and its sensors. Despite project limitations to the use of a laser scanner, a brief comparison to a vision-based alternative is considered.

Whereas a camera provides a good overview of the environment directly in front of the robot, it is limited to a viewing angle of approximately 60^o, depending on the lens. Furthermore, any distance calculations based on a single camera are dependent on knowledge about the size of the object beforehand. A setup with stereo cameras could work for depth determinations, but the viewing angle would still be limited as can be seen from the example in Figure 2.

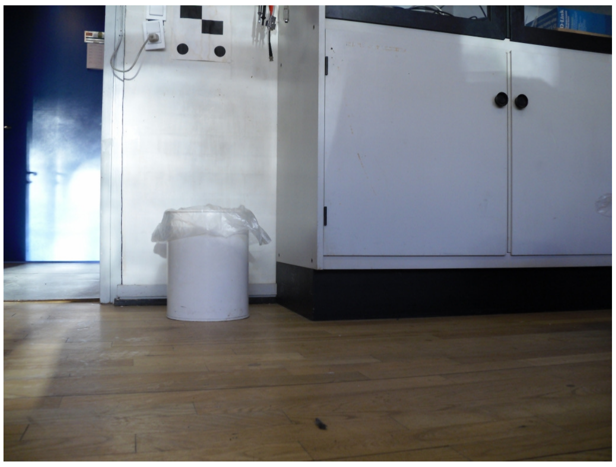


Figure 2: example of a camera view

Many 2D laser scanners are capable of doing measurements in a 240^o angle with a distance capability of 4m or more. This however comes at the price of only knowing the distance to objects and an outline of their shape. The picture in Figure 3 illustrates a viewing angle of 180^o, which is attainable with most laser scanners.

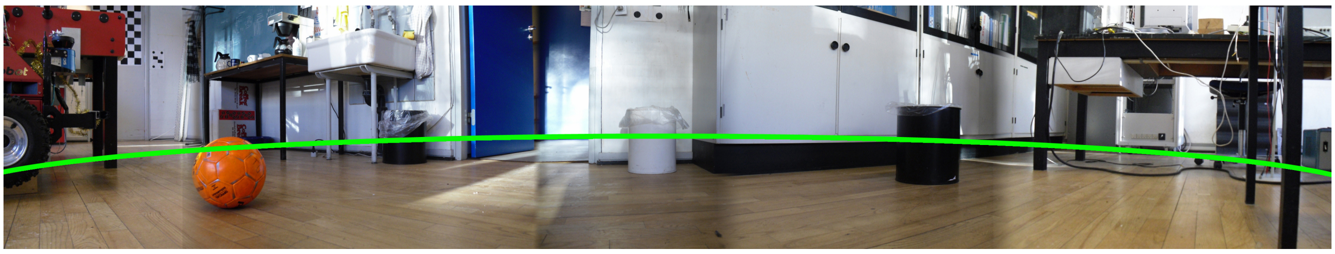


Figure 3: Merge of 5 pictures taken from the same origin. Green overlay is approximate laser-scan.

Even when provided with only the distances to objects from a horizontal laser-scan, it is possible to determine many features in the surroundings, as shown in Figure 4.

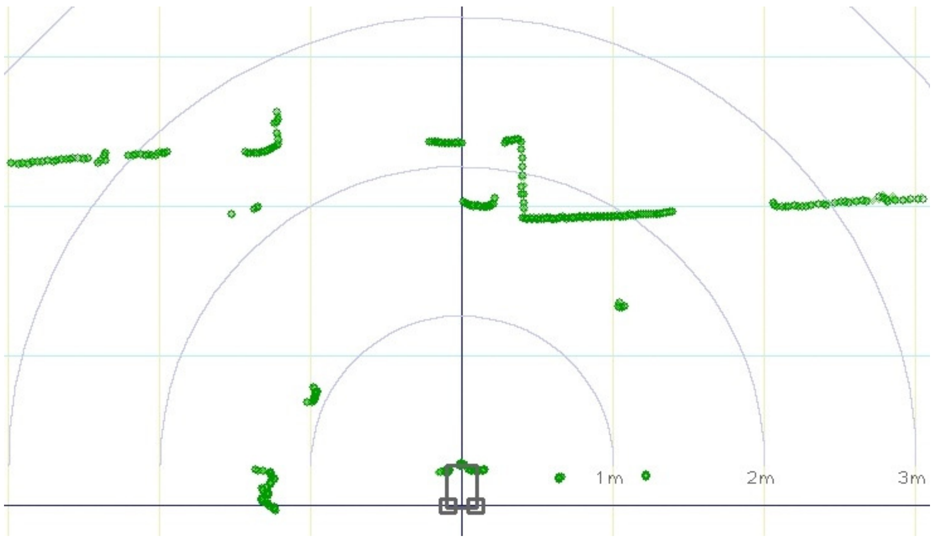


Figure 4: The resulting laserscan (240^{o}) of the environment from Figure 3.

It is seen that detecting items of the color black is problematic for the laser scanner. The impacts hereof are considered in chapter 5.3.2.

The laser scanner creates a set of points in a coordinate system by measuring the distance between the laser scanner and obstacles blocking a transmitted laser beam. Measuring the time interval between sending out and detecting the returning reflection of the laser beam, allows calculation of the distance to an obstacle. Knowing the angle in which the measurement was made, makes it possible to place the point in a coordinate system relative to the sensor. Repeating this in an entire sweep of measurements with approximately 1^{o} resolution or less, a map of the entire environment is created.

With a much wider angle of operation than the camera, as well as easily available and generally reliable distance measurements of high precision, the laser scanner is a viable choice for use in solving the problem.

3.2 Mobotware and plugins

The Mobotware system is focused on development and expansion through plugins, and includes several ways to share information between these. As the control of robots using Mobotware is divided into several interdependent parts (servers), which are each controlling the handling of a sensor, behavior or hardware, plugins can be developed to fit a corresponding server. Camera-based feature extraction is done from the camera server, laser-based feature extraction from the laser server, and behavior-related actions from the behavior server [3].

Since the task at hand is sensor-oriented (based on information procurement, and action passive) and should be solved using only the laser scanner, it is highly beneficial to do the implementation as a plugin for the laser scanner server; "ulmsserver". The entire implementation approach is described in chapter 4.

4. Implementation

Many considerations are undertaken during development of new software. This chapter provides an overview of the choices made during implementation of the plugin. The first choice is to name it "pplfinder".

4.1 Plugin approach

The basic requirement of the plugin is to receive and analyze data from a laserscanner (as previously illustrated in Figure 4), making key information about positions of humans easily available for use wherever needed.

In terms of structuring, the backbone of the plugin handles all communication with external processes and activates corresponding internal procedures when information is requested. This approach makes it possible to implement an informative user interface (UI), requiring no advanced knowledge about the plugins internal structure from the user.

In order to only use as little processing power as possible, the plugin behavior is chosen to be based on an idle state while waiting for external requests to act. This approach allows future users to decide how processing power can be distributed most efficiently in their application, instead of having to incorporate predefined timing intervals. Thus, the plugin always return to the idle state after completing a given task.

The main task is to run the entire detection procedure: obtain new laser data, extract data on legs, compare with result from previous scan, merge and save these two as a new resulting data set, and finally update the user-accessible information accordingly.

Secondary tasks are to provide specific pieces of information when requested, while making sure to convert this information to be interpretable by the requester. The three kinds of requesters are;

- Human console input (for development)
- Plugin-to-plugin (handling both "callGlobal" and "callGlobalV" method calls)
- SMRCL-scripts (requiring an XML-like reply format).

In the following five sub-chapters, approaches chosen to complete the desired tasks are explained. For a quick overview of the chosen design, a diagram illustrating the flow of actions can be found in Appendix A-1.

4.1.1 Handling raw data

Each time the command ”run” is issued, a complete analysis of the most recent laser scan is carried out.

As every laser scan contains up to several hundred points defining the general shapes of obstacles in an environment, the idea is to analyze these shapes and extract those similar to human legs into separate clusters. The shape of a human leg is usually similar to that of a cylinder, resulting in what appears to be around 1/3 or 1/2 the circumference of a circle on the laser scan, as shown in the top of Figure 5 (These clusters are taken from the auclient, making it hard to see the individual points, because of the history-functionality fading out old points instead of having them disappear instantly).

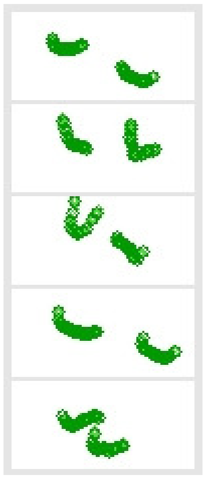


Figure 5: examples of results for occurring leg shapes.

As the four bottom examples in Figure 5 shows, it is not always easy to predict the shapes created by the fabric of clothes. Pants often have a much larger diameter than the leg wearing them, causing them to bend into folds and odd shapes. Because of this, it is not a viable solution to find legs by comparing shapes to a piece of a perfect circular circumference.

Utilizing knowledge about general sizes of human legs along with commonly expected features in terms of shape, an algorithm implemented in a previous master’s thesis by Daniel Muhle-Zimino [4] has proved to provide very reasonable results for dividing and finding leg-shaped clusters.

Unfortunately, this algorithm was only integrated for use in one specific control case, and is not directly usable in any other applications. In order to make use of the algorithm, it has been extracted from the original implementation and modified to match the new requirements by 2 major modifications:

- Originally the algorithm wrote calculation results to a file. This has been changed to transfer all data internally.
- The coordinate system in which the results were previously calculated, was relative to the laser scanner. A translation and coordinate transformation has been introduced to have the results expressed in terms of world-coordinates.

The final implementation receives and handles the scan data by dividing all given points into clusters, and evaluating the number of points in each. Clusters with too few or too many points do not match the size of a leg, and are discarded. Any remaining clusters are then evaluated by their shape. If the middle point in a cluster is the one closest to the scanner, along with the two end points in the same cluster both being located further away than this, the object would appear to have a uniform shape and is considered to be a human leg. When all clusters have been analyzed, the algorithm has completed its task and the results are sent back to the main part of the plugin.

The implementation is done in a way which allows easy substitution of the algorithm, should the current one prove to have undesirable limitations or in case any improved version is developed at a later point.

Further details on the modifications of the algorithm can be found in Appendix A-1.1.

4.1.2 Storing information

Detailed information about each leg is saved for as long as the leg is considered to be present in the scan area (the chosen approach takes advantage of the object oriented C++ language, as described in Appendix A-1.2).

The data received for storage is the coordinate values of several points defining each leg. Instead of storing all this information, it is considered sufficient to store only the mean X and Y values of the points as an acceptable approximation for the location.

Additional information stored for each leg:

• ID-number	(unique generated ID identifying this leg)
• Scan-ID	(ID of the laser scan in which this leg was detected)
• Timestamp	(specific time for detection of this leg, in unix-time)
• Point count	(the number of points detected as defining this leg)
• Certainty	(measure for likeliness of this actually being a leg)
• Position History	(the entire history of positions since discovery of this leg)

The ID-number, scan-ID and timestamp are used for identification purposes, and point count along with position history is only stored for availability in further development of pplfinder. Details on the possible uses for these are explained in chapter 4.2.2.

Most important of the additional information is the certainty, which is used to rate the likeliness of the detected object actually being a leg and allowing “wildcards”. This is explained in the following chapter.

4.1.3 Subsequent scan-data linking

For each sweep of the laser scanner, several legs from a previous scan are very likely to re-occur. This is due to a relatively high scan rate of 0.2 seconds or better (depending on the laser scanner). By always saving information from the most recent scan, comparison and possible matching of legs between scans can be done for every sweep. All positions of new legs are compared to positions of the previous ones, in search for any matches. A match is found when the distance between a new leg and the position of a previous leg is below a certain threshold. Should more than one leg qualify as a match, the closest one is chosen. When a match is made, all data saved in the previous leg is transferred to the new one, including the unique ID.

When a new leg is detected it gets a certainty value of 1. This value then increases by +1 every time the leg is successfully matched and reduced by -1 every time no match is made. Removing all legs with a certainty value of less than 1 makes it possible to handle a moderate amount of errors in readings and detection. Should one sweep fail to detect certain legs due to corrupted data, as a result of interference from noise sources in the real world, legs with high certainty can "survive" for several sweeps until hopefully being rediscovered.

This approach uses the certainty value to allow errors in the readings to some extent. Should the detection of a leg fail because of noise, the history of that leg should not necessarily be deleted right away. By using the certainty rating as a wildcard, it is possible to re-match lost legs that have been undetectable for a short time.

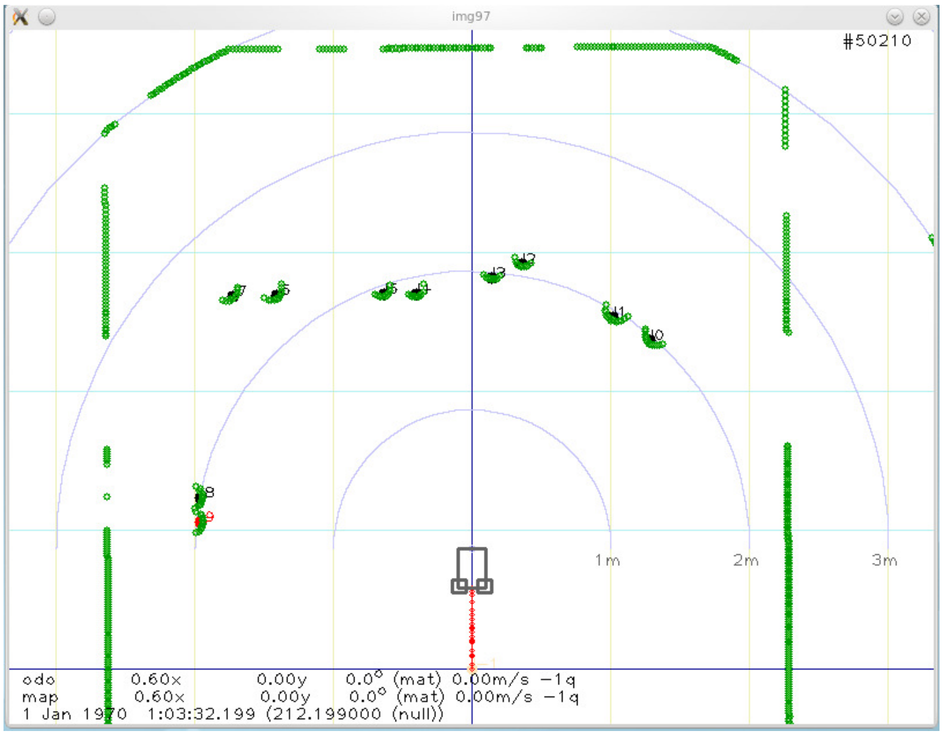
It is however important to set a maximum value for the certainty, to match the speed of the laser scanner in use. Using unreasonably large values of certainty might result in non-existing legs remaining as ghost for much longer time than reasonable, and they may even end up being "picked-up" as other legs passing by at a later time.

Thus, the maximum value of certainty should be set high enough to prevent loss of legs due to noise, but not so high that other legs are able to get within range and be mistaken for the lost one.

Adjustments can also be made by changing the acceptable matching distance, as explained in chapter 5

4.1.4 Visualization of results

With information on any legs considered to be currently visible, a way of visualizing these is desirable for debugging purposes as well as future development and use of pplfinder. Utilizing the pre-made plugin "aupoly", it is possible to visualize polygons on the laser-scan display of the auclient. As all data for the current laser scan is already shown, an overlay of leg positions is added in the form of circular polygons of various colors, as shown in Figure 6.



```
odo        0.60x        0.00y     0.0° (mat) 0.00m/s -1q
map         0.60x        0.00y     0.0° (mat) 0.00m/s -1q
1 Jan 1970  1:03:32.199 (212.199000 (null))
```

Figure 6: Laser scan data visualized in the auclient with leg positions overlaid.

Thin blue lines mark the world-coordinate system, green circles represents the raw laser scan data, and the red marks indicate the path history for robot movement. The detected legs are marked with colored circles and a polygon-number.

Colors are also used to illustrate the certainty level of each leg. Red indicates a leg with certainty 1, which has either just been discovered or is about to be lost. Blue ones are for all certainties between 1 and the chosen maximum value, and finally the color black indicates a certainty of the highest value allowed.

These color codes provides a good overview of the detection performance in real time, making it very intuitive to understand and adjust the configuration parameters for maximum certainty and matching distance.

4.1.5 Interface and Availability of information

Even though the use of data for visualization is advantageous for a human user, the most important is to make the data available for use in other applications and thereby making sure that key information is accessible in a form where it can be used directly. How the output should be formatted depends on the type of client requesting it. For use with Mobotware there are three types of clients: humans, other plugins, and SMRCL-scripts.

The kind of information desired by a human is somewhat satisfied with the visual representation. However, in some cases it is advantageous to have access to the numbers behind this, providing many details in precise numerical values. Even though this approach to data retrieval will never be used in final autonomous solutions, it is an important tool for development and maintenance of the plugin. To satisfy this need, all important information can be requested and shown directly in the console of the ulmsserver, including: current values for the calibration parameters, the amount of legs currently in view, ID for the nearest leg and detailed information on any leg by providing its ID.

For plugin-to-plugin communication a special method is used in all parts of Mobotware, defining a standardized way of implementation to follow. Meeting these requirements, pplfinder has the capability to communicate as defined by the standard (with implementation for both callGlobal and callGlobalV method calls). Basic information can be read directly in "public" read-only variables, and detailed information on specific legs is delivered per request.

As the most basic way of creating autonomous solutions with the Mobotware platform is to create scripts written in SMRCL, this is an important aspect to support. Because of a different syntax for passing information, additional communication options have been created for use with SMRCL (formatted as XML-like replies).

A helpful overview of the interface is accessible with the help-function directly in the console, and can also be found in Appendix A-1.5.

4.2 Plugin overview

Providing a brief summary of the final implementation in its entirety, this sub-chapter evaluates the main aspects in future use of pplfinder.

4.2.1 Functionality

The primary functionality achieved is the ability to process data from a laser-scan sweep; detecting, tracking and storing information on human legs, as well as making this information available for easy use as a tool in future research and development of the Mobotware platform.

The pplfinder plugin also includes the possibility to recognize specific legs through several separate scans, based on adjustable configuration values. Being configurable to match various hardware specifications and environments, the portability of Mobotware along with this plugin is not made any less favorable.

With focus on ease-of-use, the chosen interface includes both an informative help-function, reasonable initial values for all configuration variables, implementation including warnings and errors for troubleshooting, intuitive real-time visualization of results, as well as extensive compatibility with other plugins and usability in SMRCL-scripts. Along with the programming oriented explanations of functionality found in the appendixes of this report, a good basis for the future maintenance and use of the pplfinder plugin has been established.

4.2.2 Expandability

Storing information that is not used in the current functionalities of pplfinder, is done to provide an option for continued development of the plugin. Additionally, the chosen structure of software allows an optional use with alternative algorithms for detection of any shapes, by replacing the current algorithm and using the plugin as a framework.

As an example, the position history could be used to further improve recognition of legs in consecutive scans. Knowing several recent positions of a leg, it would be possible to estimate movement direction and speed, providing valuable information to the control software for prediction of future obstacle positions and collisions.

5. Results

Testing of the plugin has been carried out in two phases. As part of the development process, a simulation environment has been used to eliminate errors and bugs. Using a simulated environment makes it possible to notice even small implementation errors, which could have been overlooked in real world test where noise is present. When functioning as intended in the simulations, the next step was to use a real robot in a real-world environment. In the field of mobile robotics, this is often the most critical part of testing, as the introduction of numerous noise sources often results in a far-from ideal usage of the theory-based solution implemented.

5.1 The SMR-Platform

The platform used in all test is the Small Mobile Robot (SMR). The SMR is the common development platform used for courses and projects at the department of automation and control at the Technical University of Denmark (DTU), as it is small and differential controlled, which is advantageous for indoor navigation. A laser scanner is mounted in front of the robot, along with several other sensors (which won´t be used by pplfinder). The option to use other robots and laser scanners at a later time is available, since the plugin is compatible with any robot using Mobotware.

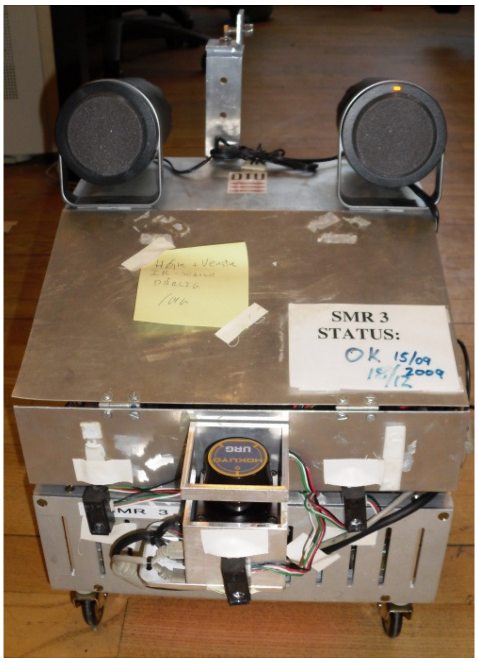


Figure 7: Small Mobile Robot (SMR) with repositioned laser scanner.

The laser scanner is a "Hokuyo URG-04LX" [5], allows scans of up to 240° angle with 4m range and is usually mounted only a few centimeters above the ground on a SMR. However, as the plugin is developed to handle scans of legs (not feet), the laser scanner has been repositioned to 20cm above the ground, as can be seen in Figure 7. This reposition includes a vertical flip resulting in mirroring of the collected scan data. To cancel this undesired error, it is important to make a correction in the ulmsserver initialization file (which should be set to "`scanset mirror=false`" as the scanner is no longer positioned upside-down).

A close-up image of the scanner placement can be found in Appendix B.

5.2 Simulation

Mobotware provides the option to control a virtual robot in a virtual environment. This functionality builds on a library from the Player/Stage project [6] which is distributed under the GNU-license. Only the "Stage" part of the project is used here.

5.2.1 Stage

Building environments for use in "Stage" is very intuitive. A bitmap can be used directly as a template for a 2D environment, scaling the image resolution to a desired environment size.

For simulation purposes in relation to this plugin, a 15m x 5m environment consisting of three 5m x 5m rooms is created. Circles in the bitmap are approximated to resemble the cylindrical shape of human legs, and are placed throughout the three rooms as shown in Figure 8. The general leg size is set to resemble a diameter of 16cm, and the small legs to resemble 10cm.

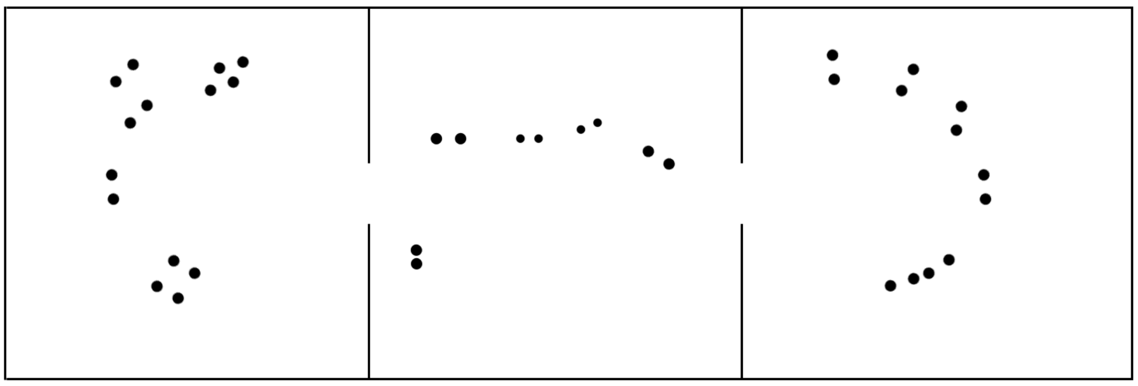


Figure 8: Bitmap file used as a 2D map within the simulator.

Each room is meant to test a different scenario of data collection as a result of human standing positions. These tests are done within a static environment, but with a moving robot. Furthermore, the simulated data collection is noise free, and the leg shapes are perfectly cylindrical.

A video of the entire simulated run-through can be found on the CD, as video 1.

The purposes of the rooms are to test:

Left room: Humans standing close to each other; blocking line-of-sight to another person's leg, as well as making it difficult to decide which leg belongs to which human.

Middle room: Humans of different age (leg sizes), in addition to a pair of legs placed closely together.

Right room: Humans standing very close to each other, and single humans facing in a direction that is not directly towards the robot.

The run function in pplfinder is set to execute sequentially with 0.2 sec interval, making all detected legs shows up on the map of laser data.

5.2.2 Simulation results

In summary, the plugin easily locates most of the legs, which is a direct impact of the detection algorithm expecting legs to be very similar to cylinders. It is however surprising that two legs positioned close together is often detected to be only one leg. In some cases this would be an acceptable error, since the robot at least knows about the human presence. The testing also shows that a correction will take place as soon as the robot has a better angle of detection, as illustrated in Figure 9.

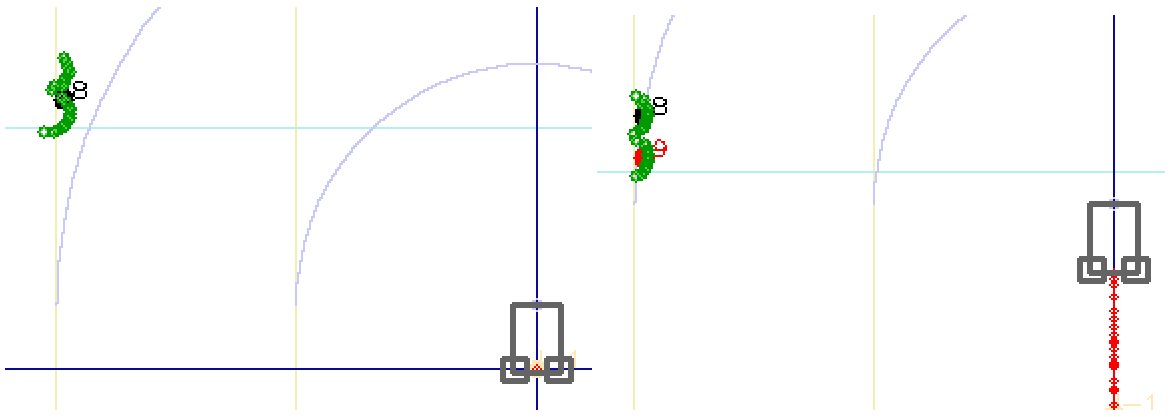


Figure 9: Detection of legs placed close to each other

The results from the left test room show that legs partly covering each other are acceptable to a certain degree. With this resolution and leg size it is required to have around ¼ of the circumference detectable. As expected it is impossible to decide which two legs make a pair, especially when not all legs are visible at the same time.

The right room confirms the theory that it is very plausible for a person to have one leg cover the other, and that it becomes easier to do so if the person is standing close to the laser scanner. With regards to the people standing close to each other; all legs are detected. It does however prove difficult to detect these from some angles, as can be seen from the simulation video (1) on the CD.

Pictures from the tests in all three rooms can be found in Appendix C

5.3 Real world

Testing the theory in a stationary simulated environment does not provide a very good basis for final evaluation when the plugin is designed for use in the real world. Introducing moving persons and a dynamic environment with numerous noise sources, is the only way to test if pplfinder is actually working as intended. To evaluate correctly, an analysis is conducted on the conditions under which the testing is carried out. Moving humans with legs of varying shape caused by pants, as well as unwanted wrongful detections of static objects as human legs, is a major issue to handle.

5.3.1 Humans

The average human walking speed is around 5 km/h with a variation of approximately ±1 km/h for young and old individuals. Using the formula; [km/h] / 3.6 = [m/s], and assuming that only one leg is moving at a time (which is then twice the human speed), these movement speeds are converted as shown in Table 1.

Human [km/h]	1.0	2.0	3.0	**4.0**	**5.0**	**6.0**	7.0
Human [m/s]	0.28	0.56	0.83	**1.11**	**1.39**	**1.67**	1.94
Leg [m/s]	0.56	1.11	1.67	**2.22**	**2.78**	**3.33**	3.89

Table 1: Human walking speed conversion chart

Depending on the laser scanner in use, the time taken to do one full scan varies from approximately 25ms to 120ms.However, due to a compatibility problem in the Linux drivers for the URG-04LX scanner used in these tests; every second scan contains corrupted data. This lowers the scan rate from a hardware capability of 10 scans per second to software limited 5 scans per second, until a driver correction is issued by either the manufacturer or in new distributions of Linux. Comparing common scan rates to the above leg speeds, the distance moved between each scan can be calculated as in Table 2 below.

Leg speed / Scan rate	0.56 m/s	1.11 m/s	1.67 m/s	**2.22** m/s	**2.78** m/s	**3.33** m/s	3.89 m/s
25 ms	0.01	0.03	0.04	**0.06**	**0.07**	**0.08**	0.10
60 ms	0.03 m	0.07 m	0.10 m	**0.13** m	**0.17** m	**0.20** m	0.23 m
80 ms	0.04 m	0.09 m	0.13 m	**0.18** m	**0.22** m	**0.27** m	0.31 m
100 ms	0.06 m	0.11 m	0.17 m	**0.22** m	**0.28** m	**0.33** m	0.39 m
120 ms	0.07 m	0.13 m	0.20 m	**0.27** m	**0.33** m	**0.40** m	0.47 m
200 ms	**0.11** m	**0.22** m	**0.33** m	***0.44** m*	***0.56** m*	***0.67** m*	**0.78** m

Table 2: Distance traveled by a leg between two consecutive laser scans. Color indicates reliability to be expected for use with pplfinder

The diameter for a leg in pants (below the knee) is approximately 0.16m, which is then also the distance between the centers of two legs positioned very close to each other. Considering the scenario that one of these two legs is not detected during a scan, the maximum distance allowed for "merging" legs in consecutive scans should be below 0.16m, in order to avoid any incorrect matching of one leg as the other. Even though this threshold can be higher for normal movement (with much longer distance between the legs), the lowest threshold is the one to use unless a dynamic evaluation process is designed.

Even though it's impossible to generalize the size of all human legs, this example can provide a reasonable scope of what to expect from the tests carried out on the SMR platform. The coloring of Table 2 visualizes this, and makes it obvious that a solution to the previously mentioned driver issue would greatly increase performance.

Furthermore, these numbers are based on a simplified model of the real world, not taking into account that the leg movement between each scan is doubled in the case of just a single failed detection, as well as the additional distance change caused by any movement of the mobile robot between scans.

5.3.2 Environment

The real-world environment used to test the plugin is far more varied than the clean simulation walls. Especially everyday objects shaped as what the detection algorithm would find to be a human leg, are interesting elements of high relevance and included in the environment as shown in Figure 10.

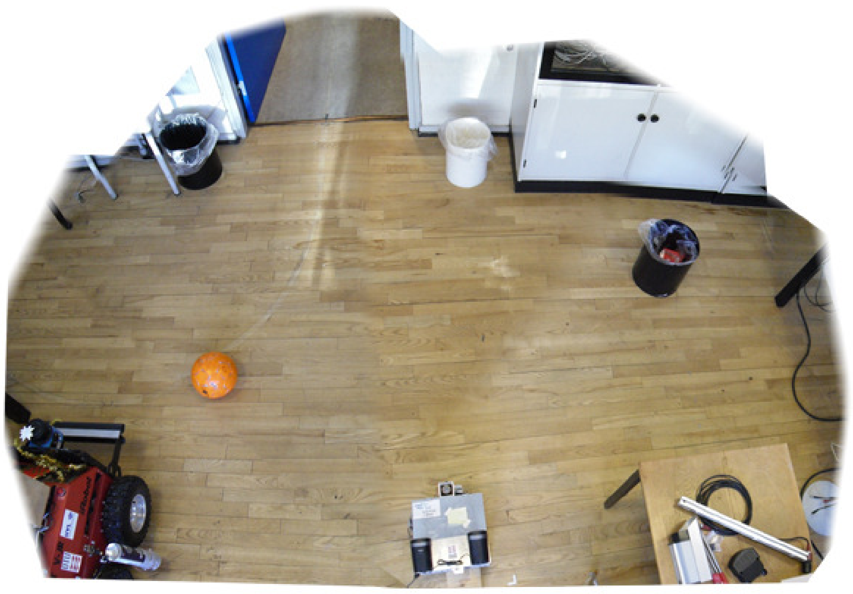


Figure 10: Test environment (combination of 4 pictures)

The laser scan obtained at the same position, along with the resulting overlay of several pplfinder runs, is shown in Figure 11.

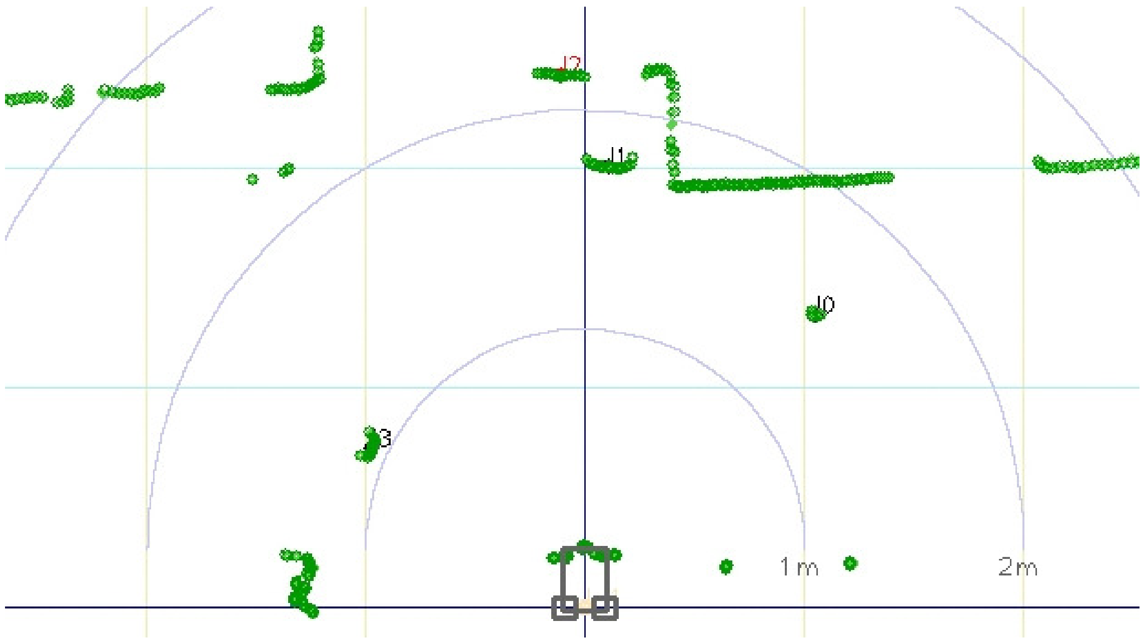


Figure 11: Laser scan for the environment shown in Figure 10

An obvious issue with detection of environments by using a laser scanner, is the problem of getting readings from black surfaces with a shiny finish. On this scan it shows on both of the black cans, as well as a black table leg in the right side of the picture. If a surface is too reflective it works like a mirror and redirects the reflected laser beam almost completely. If the angle of refraction is not very close to zero, the scanner is left without any returning beam to do time measuring, and the point is considered as being out of the 4m scanner range. Since the black cans still cast a "shadow" in the form of missing points behind them, an approach using mapping of the entire environment would still be able to show their presence without any successful readings directly off their surfaces.

In terms of detection, another problem is also very obvious: the algorithm has found four legs in an environment containing none. The image in Figure 11 was taken after several runs of the pplfinder plugin, allowing the certainties to settle. The laser scan of the top of the soccer ball resembles the outline of a human leg very well, justifying the choice to regard it as one. It is quite the opposite with both the white and black can, which are obviously too large or small respectively, to resemble normal human legs. Even though this seems to be a flaw in the detection algorithm, it is partly justified by an issue of portability. Depending on the laser scanner in use, the number of points defining a leg varies with the resolution used. Thus, instead of using a number of points in a cluster to evaluate upon, a measurement in length should be used in order to achieve a good portability. As the used algorithm is not designed to be portable, this is a reasonable flaw.

The flat piece of wall in the back is detected as a leg in only one out of a few scans, as also indicated by the color. Fortunately this kind of detection doesn't reach more than very low certainty ratings, which can be used to filter it from the rest.

Even though it is a problem to have static objects showing up as legs, it is impossible to completely avoid without using other types of sensors as well. And in the case of detection errors it is definitely better to mistake objects for being legs, than mistaking legs for being just objects.

5.3.3 Calibration parameters

Configuration of the calibration parameters is normally done during the initialization, but the option to adjust the values at any time is also available. If it is chosen to use dynamic values, any actions towards adjustments should be based on information from other sources than pplfinder. Since; when the data provided by pplfinder is qualified for making decisions on adjustments, the parameters must then already be perfectly adjusted.

In all tests presented in this report, static configuration values have been used. The values for maximum distance have been chosen in the interval 0.11 – 0.33 m as indicated to be reasonable in Table 2, and maximum certainty values in the range 5 -20 are also considered reasonable, as these resemble 1-4 seconds of subsequent perfect detections.

5.3.4 Real-world results

The detection-capabilities of pplfinder have proven to work very well in a real-world environment. Even with strange shapes of pants, the algorithm detects almost all present legs with each scan, as shown in video 2 on the CD. However, the high detection rate comes at a price of also evaluating several static objects as being legs. This behavior was anticipated to some extent, and as shown in chapter 5.4 the plugin is still very usable for its intended purpose.

Tracking legs with the relatively low scan rate proved to be a major problem at anything but very slow movement speeds. Best results were attained with a maximum distance of 0.15 - 0.20 m along with a maximum certainty of 5-10. With this configuration it was possible to keep track of legs moving well below the average human walking speed, with some problems in the case of one leg blocking another as shown in video 3 on the CD

Even though tracking was only possible at low movement speed, the tests have shown that the implemented theory does work in practice, indicating increased capability with higher scan rates.

5.3.5 Noise Sources

Several noise sources have proven to have significant negative impact on the performance in real-world operation, and are estimated to have influence on the following:

Black color	The color black absorbs a lot of light instead of reflecting it. If the returning laser reflection is too weak, it won't be detected by the laser scanner.
Reflectivity	Shiny surfaces will reflect almost all light in a single direction, resulting in no light being reflected back towards the laser scanner, unless the angle of reflection is very small.
Sunlight	Direct sunlight on surfaces in the environment or directly at the laser scanner makes the laser light severely less distinctive and harder to detect.
Pants	Persons wearing non-tight pants will be detected as legs of varying shapes. Many of these are however still recognized as leg by the currently used algorithm.
Distance to sensor	If a leg is close to the sensor it will include many points of detection compared to a leg at a longer distance. This complicates the approach used in the detection algorithm, which evaluates the number of points defining a cluster.
Movement speed	Moving objects can be a problem when merging consecutive scans, and for slow scan rates the possibility of slight deformation of data should also be considered.
Overlapping legs	One leg blocking the view of another makes such a hidden leg both undetectable and untrackable for the duration of the overlap.
Laser range	When moving out of the maximum laser range, a tracked leg will remain at the point of disappearance until its certainty reaches zero.

5.4 Application and usability

Even though pplfinder is designed to provide information on positions of human legs, the option to use it as a framework for other detection algorithms makes it usable in many other applications.

Using a laser scanner to detect and track humans successfully in a dynamic environment has high requirements to the performance of the laser scanner. With an investment in the right hardware, the plugin would achieve results of good use in robot control. Primarily in controlled environments such as industrial facilities, where the number of leg-shaped static objects can be considered and taken care of.

On the SMR-platform the usability of the plugin is limited by the scan rate and position of the laser scanner. The low scan rate severely limits the ability to track legs, and the initial position of the laser scanner makes it impossible to even detect legs (standard placement is approximately 5cm above the ground, resulting in data on shapes from shoes instead of legs, and causing problems when feet are lifted from the ground while walking). With a repositioned laser scanner the detection functionality becomes useful, but any other control software using the laser scanner needs to be adapted to this new hardware configuration. Even with only the functionality of detection (at least until the laser scanner driver is fixed), the resulting data is still useable as guide marks for other sensors and control processes. As an example, the efficiency of camera based detection of humans would be greatly improved when provided with positions of what appears to be human legs to the laser scanner. Instead of searching the entire field of vision, it would be possible to concentrate on the regions of interest pointed out by the pplfinder plugin. Furthermore, a way of communicating the outcome of the camera analysis back to pplfinder could also help reduce the number of wrongful detection of static objects as human legs.

Primary use of the plugin in its current form and with the available hardware will most likely be with the other mobile robots used as research platforms for Mobotware, of which two are shown in Figure 12. With higher scan rates and higher position of the laser scanner, these robots will be able to benefit from the plugin immediately.

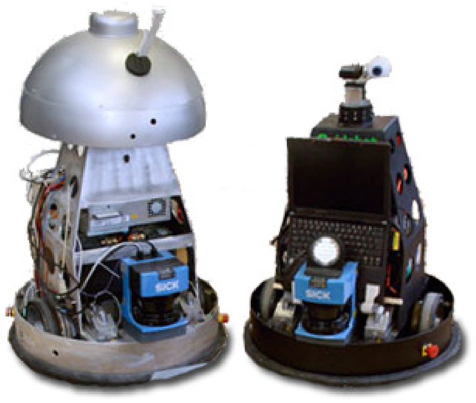


Figure 12: Two of the larger mobile robots from the AUT department of DTU

Additional options for use with the largest robots using Mobotware should also be considered. A major research field is development of robust control methods for mobile robots in agricultural use. These robots are autonomous tractors equipped with a wide range of sensors and computers in order to achieve a high level of control robustness, making it reasonable to allow these giant machines to operate without direct human influence. One way of achieving this is by merging data from sensors of several types to get a better model of the environment. An obvious use for the pplfinder plugin would be for navigating an orchard with lines of trees, where the trunks of the trees would be detected as legs because of their shape. Positional data on the trunks could be used to estimate where the two closest lines of trees are placed relative to the tractor, and send out warnings if the tractor is not driving in the middle of the two.

5.5 Proof of concept

In order to provide ideas for using pplfinder, as well as to show examples of implementation for use as reference in the future, two SMRCL-scripts has been created. A simple script using only positional data to find and move towards the closest visible leg, and a more advanced approach of having the robot follow a specific leg.

These examples are meant to demonstrate the capabilities of pplpfinder as best possible with the limitations of using the SMR-platform for the tests.

5.5.1 Follow nearest

Inspired by a functionality of the Iromec robot mentioned in chapter 1, making the robot follow the nearest leg available is a good way of demonstrating that the plugin is working as intended.

A simple instruction to follow the nearest leg has limited practical use, due to the uncertain identity of the leg being followed. But even with this limitation, a use for this approach still exists. In an exercise of keeping attention, the Iromec robot makes use of this simple rule to follow the person currently granted its attention (the closest one). A child quickly understands the robots rules of behavior, and will usually try to get the attention of the robot by stepping between it and the person being followed.

While the Iromec prototype has multiple sensors and specially designed control software, allowing the ability to avoid obstacles while pursuing a person, it is possible to replicate the basic idea with only one laser scanner and the pplfinder plugin.

Getting the X and Y coordinates of the closest leg is done with a single command, but calculating the angle to turn in order to have the robot face the right direction has to be done in the SMRCL-script. Initially a desired distance to the closes leg is chosen, making the robot move towards it whenever the actual distance is greater than the desired. Additionally it was seen through test results that continuous adjustments on the direction of the robot was a great improvement compared to only doing directional adjustments when driving (using the SMRCL command “driveon”, and driving only a fraction of the total distance before re-evaluating).

Since this implementation uses only immediate positions of legs, the configuration values for both maximum certainty and distance difference should be set relatively low, as there is no need remember the identity of each leg. This change of values is done directly in the SMRCL-script.

The performance of the final implementation is very decent, taking into account that there is no attempt of obstacle avoidance. Within the range of human movement speed allowing the robot to keep up, the task of following is successfully completed. Further improvements could be made with a limitation in the detection angle, making the robot less likely to suddenly turn towards passing humans or table legs to the sides, when they are detected as being legs of closest distance. A demonstration is shown in video 4 on the CD.

5.5.2 Follow best

Another option to solve the problem with passing humans has been tested. The idea was that with a higher allowance in the value of maximum certainty, the leg being followed would have a higher certainty value than a briefly passing one. With the only change being to follow the best leg (of highest certainty) instead of the closest, results was not very good. The robot would simply follow any stationary person within range of the laser scanner, since moving is the main cause of failed detections and lowering of certainty. Where the “follow closest” solution is usable, the “follow best” is not.

5.5.3 Follow VIP

As a combination of the advantages with each of the two previous examples, a solution to follow a specific person has been created. This could be a very useful ability in many cases where a mobile robot has to navigate in an environment shared with humans.

An example of use could be in fields where rolling tables are used to carry objects, such as hospitals and restaurants, requiring the full attention of an employee to navigate the table. By making the table intelligent and able to follow a specific person, the employee has both hands free for other purposes while transporting the table.

Another advantage could be the ability to navigate in an environment filled with humans, without getting in their way. While the fastest way for a robot to move from A to B would be in a straight line, any humans in its way would have to move out of the way. Whereas humans walking towards each other usually manage to avoid collision almost by instinct, robots need many sensors to achieve the same ability of avoiding collision with a human. Taking advantage of the human abilities, following a human would require less important decision making from the robot and could maybe even allow us to learn more about instinctive human navigation rules.

The developed script is designed to demonstrate the ability of using the tracking functionality of pplfinder to follow a specific leg, while ignoring all other occurrences of legs in the environment. This approach also deals with the problem of having static objects detected as legs, since these will also be ignored while the robot is following the VIP. When no VIP is chosen, the robot remains stationary and search for legs within its "target zone". To be chosen as the VIP, one has to stand in a 0.6m by 0.8m square in front of the robot and achieve a certainty of the maximum value allowed. When this happens, the robot changes state to "following" after informing its surroundings by speaking the word "found". Once the VIP has been chosen, the robot will follow that person until he disappears (his certainty goes below 1). When this happens, the robot pronounce the word "lost", stops at its current position and goes back to initial stage to search for a new VIP. Using the approach to stop when in doubt is a good example of practical use because of its high safety, and is also a good way of demonstrating the problem of having a slow scan rate.

With a maximum certainty of 20 and maximum merging distance of 0.16m, the SMR should be able to keep up with human movement of approximately 1.5 km/h = 0.42m/s (a leg speed of 3.0 km/h = 0.83m/s) under ideal conditions. Results from tests have however shown that the conditions are not always ideal, and that even slower movement is required to gain a good reliability. Alternatively, using a laser scanner with a faster scan rate would be another way to overcome this problem.

5.6 Further work

Even though the plugin is ready for use in its current state, it is also a possibility to continue its development.

Utilizing the saved position history could increase reliability of the tracking capability, by expanding the amount of information available for each leg to also include movement direction and speed.

After this, the obvious step would be to experiment with combination of detected legs to indicate which two legs are considered to be a pair from the same human. This could prove useful to overcome the problem of having one of a person's legs overlapping the other.

Finally the plugin could be used as a framework for testing and using new laser based detection algorithms, for use with any robot capable of running with Mobotware.

6. Conclusion

In expanding the tools available for use with the Mobotware framework, the option to detect and track humans has been successfully added through completion of this project. Providing information on positions of human legs in the immediate environment of mobile robots has been made possible by creation of a plugin utilizing data from a single laser scanner. For use with laser scanners of high scan rates, the option to track individual legs over several scans is also a viable information resource.

Through evaluation of the given limitations it became obvious that these would not compromise the overall functionality of the solution to be developed. The use of a laser scanner provides a wider area of detection than a camera, and implementation of the solution as a plugin allows direct use in existing solutions as well as good portability alongside Mobotware.

The plugin was developed in C++ with a software structure allowing future maintenance and further development. The algorithm finding clusters of leg-shaped points from a laser scan is easily replaceable, providing the option to extract any other shapes for use in applications other than detection of humans.

Detection of human legs has been proven to work reliably, through extensive testing conducted in both simulated and real-world environments. With high scan rates it is possible to compensate for occurrences of failed detections by comparing and matching subsequent scan results, providing the basic level of noise tolerance required when used in real applications.

Successful tracking of human legs requires a scan rate reflecting the movement speed to be tracked. This has been made clear in an example using the tracking ability of the plugin to follow a specific leg until lost, which proved to provide good results for visualization of the problem. By using the saved history of each leg it would be possible to further improve tracking capabilities, with prediction of movement calculated from a history of previous positions to estimate speed and direction of each leg. To achieve truly reliable tracking data for standard human walking speeds, it is necessary to use a laser scanner with a scan rate of approximately 60ms per scan or less.

By using adjustable values in the process of matching legs from consecutive scans, the plugin has been made configurable to match many different noise conditions and environments, which is an important factor in terms of adaptability.

Even without the increased reliability achieved with fusion of data from more than one sensor type, a basic behavior similar to that of a professional mobile robot prototype has been achieved. Providing data on human positions for fusion with data from other sensor types or as a tool in development of new control algorithms, significant advantages can be achieved with use of the developed plugin, in the increasing number of cases where mobile robots have to navigate and share an environment with humans.

When reviewing the entire project in terms of results achieved, the final solution is compared to the initial desired success levels presented in chapter 2.4. With a reliable functionality to provide other plugins and even SMRCL-scripts with information on human positions, the basic level of success is considered to have been achieved. Including the option to also track these people when moving, along with the several simulated and real-world test-results and examples of use presented in this report, increases the general usability of the product in many future applications. Even though the ability to track humans moving at average walking speed could not be proven to work with the laser scanner on the SMR, the theory behind it has been documented and is thereby estimated to work using laser scanners with higher scan rates. With this detail in mind, the results achieved through completion of this project are considered to nearly fulfill the requirements of the highest success level desired.

Appendix A

This appendix contains more details on the plugin implementation explained in chapter 4. With focus on the use of C++ in the programming process, this documentation is mainly targeting future programmers maintaining or further developing pplfinder.

A-1 Flow Diagram

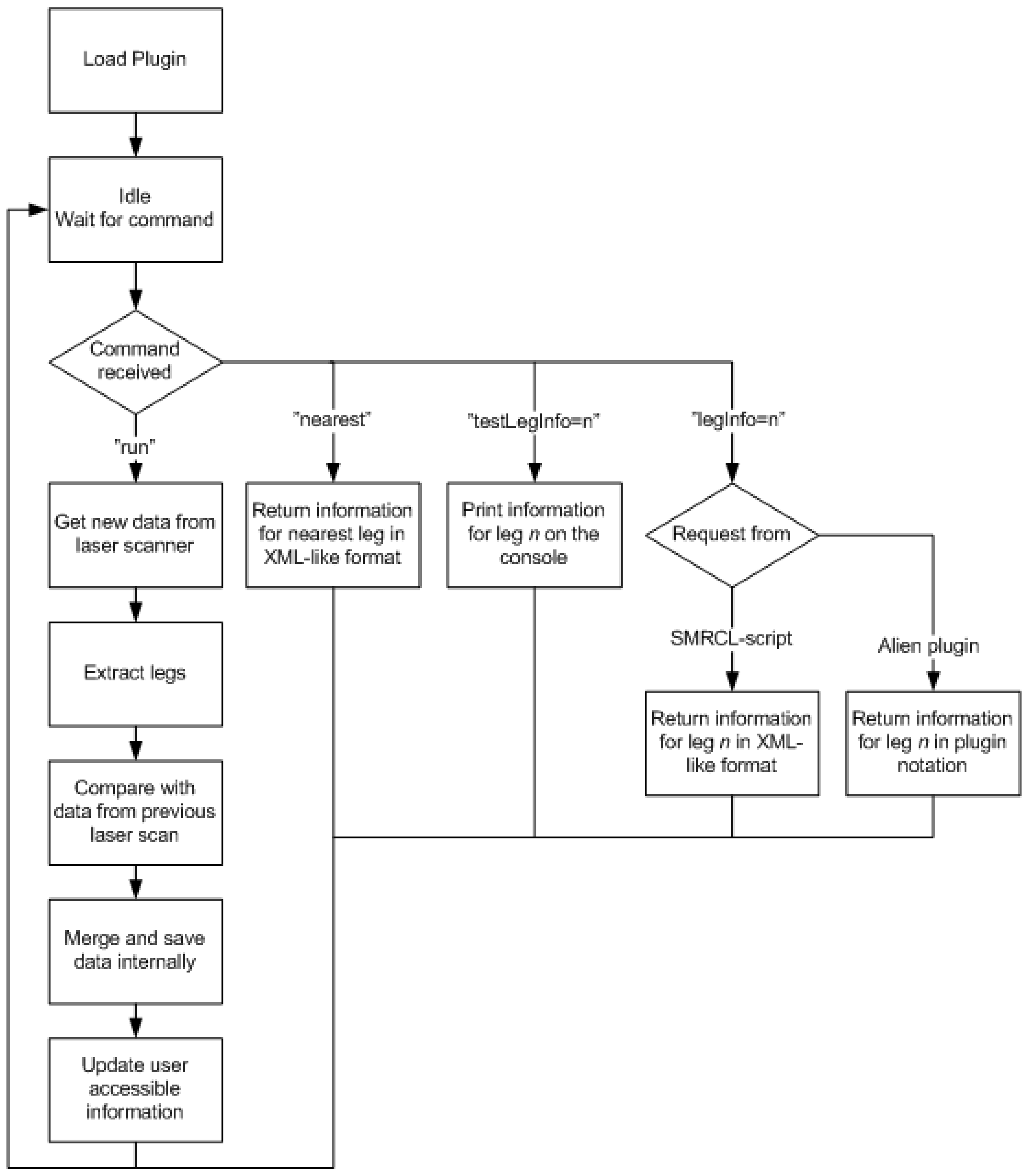

Load Plugin
Idle Wait for command
Command received
"run"
"nearest"
"testLegInfo=n"
"legInfo=n"
Get new data from laser scanner
Extract legs
Compare with data from previous laser scan
Merge and save data internally
Update user accessible information
Return information for nearest leg in XML-like format
Print information for leg n on the console
Request from
SMRCL-script
Alien plugin
Return information for leg n in XML-like format
Return information for leg n in plugin notation

Figure 13: Overview of the flow of tasks in the plugin.

A-1.1 Sorting of raw data

The original implementation of the detection algorithm had all results saved to a file for further analysis. Since opening and writing in files is much more time consuming than internal information exchange in a program, it was chosen to do change this. For future needs to debug the algorithm by also having the cluster data available in files, a boolean variable "DEBUG" can be set to true.

Instead of modifying the algorithm to provide the results in world-coordinates, and not relative to the laser scanner, the coordinate translation and transformation is carried out in the main part of the plugin (ufuncpplfinder.cpp). When a point is given in laser scanner coordinates to be at (x, y) and the origin and orientation of the robot is known to be (x_0, y_0, θ_0) with an offset of $(\Delta y_{scan}, \Delta x_{scan})$ from the laser scanner to robot origin, the same point can be expressed in world coordinates as:

$$\begin{bmatrix} x_w \\ y_w \end{bmatrix} = \begin{bmatrix} cos\theta_0 & -sin\theta_0 \\ sin\theta_0 & cos\theta_0 \end{bmatrix} \begin{bmatrix} x + \Delta x_{scan} \\ y + \Delta y_{scan} \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

Which has been implemented in C++ code as the method `robotToOdoCoo-Transf(…)`.

The detection algorithm is kept in a seperate .cpp file (zimino_Pdetection.cpp), and it is clearly marked which part has been added (also, the original code is commented in Danish).

A-1.2 The "Leg" class

Taking advantage of option to create objects in C++, Leg-objects are used to keep track of the gathered data. The new class is defined in the files Leg.cpp and Leg.h, and has been build from scratch. As a Leg object is meant for storing information, it contains several fields with accessors and mutators with only very basic calculation to be done within the object itself. The basic field values are set already when the constructor is called, and only some are editable afterwards.

Upon creation of a new object, the following information is provided and stored;

ID	- An number unique to this leg (integer).
scanID	- The ID of the laser scan in which this leg was first detected (long).
time	- Timestamp for time of detection, in unix-time (double).
p	- Positions of all scan points belonging to this leg (vector<UPosition>).
DEBUG	- Boolean value allowing detailed console output if set to TRUE (boolean).

Instead of storing information of all the points defining a leg, the constructor calculates mean values for X and Y and stores these in fields, as well as in the position history list. The position history is saved for each leg as a log of position and behavior. A direct use of this information could be to always have an updated trajectory and speed for each leg, in order to predict basic movement. Whereas this functionality haven´t been implemented, it leaves the possibility to add additional functionality if desired.

A-1.3 Subsequent scan-data linking

For each scan with the laser, several legs from the previous scan are very likely to remain. This is due to a relatively high scan rate of 0.2 seconds or better, depending on the laser scanner. Putting this theory to use, the plugin saves a vector of the legs detected in the previous scan before a new scan is processed. All positions of new legs are compared to positions of the previous ones in search for any matches. A match is found when the distance between positions of a new leg to the position of a previous leg is below a certain threshold. Should more than one leg qualify as a match, the closest one is chosen. When a match is made, all data from the previous leg is transferred to the new one, including the unique ID, and the certainty rating of this leg is increased by 1 (if possible).

Upon completion of the matching process, all non-matched legs from the previous scan get a reduction of -1 in their certainty value. Finally, any legs with positive certainty are accepted as being present in the current scan. This meaning that legs with high certainty can "survive" for several scans without any leg being detected at that position.

This approach uses the certainty value to allow errors in the readings to some extent. Should the detection of a leg fail because of noise, the history of that leg should not necessarily be deleted right away. Using a certainty rating as a wildcard it is possible to re-match lost legs that have been undetectable for short amounts of time.

It is however important to calibrate the value of certainty_max (int), to match the laser scanner in use. Having unreasonably large values of certainty might result in non-existing legs remaining as ghost for much longer time than anticipated, and maybe even get "picked-up" as other legs passing by at a later time.

Thus the certainty value should be set high enough to prevent loss of legs due to noise, but not so high that other legs are able to get within range to be mistaken for the lost one.

Adjustments can also be made with the variable max_dist (int), defining the threshold for the matching distance between two legs.

A-1.4 Visualization of results

To have the results visualized on the display of the auclient, it is required to load `aupoly.so.0` in its configuration file.

All of the leg-points are deleted and re-drawn width each run of pplfinder, and are shown with numbers starting at zero and counting up in the scan direction. A single point at the initial starting position of the robot has been added as a workaround for a problem with placement of the text for the first point. The issue was investigated far enough to conclude that the problem did not exist in pplfinder, and a workaround was thus used.

Whereas an option to disable the visualized results is not implemented, this can easily be done by making execution of a single line of code conditional

A-1.5 Interface and availability of information

Following is the ulmsserver-console reply received when executing the command "`pplfinder help`" after successfully loading `aupplfinder.so.0`.

```
--- available pplfinder options ---
help            This message
run             Run pplfinder; analyze current laserscan, and update all
                 variables and history.
                (Must be run every time you want to update. This is the
                 command you want to \"push\").
nearest         Return information for the nearest leg in XML-like format
                 for SMRCL, with the variables;
                l0 = <ID>             (the unique ID-number for this leg)
                l1 = <scanID>         (the laser scan ID-number)
                l2 = <timestamp>      (time since plugin load)
                l3 = <Xmean>          (mean X value for points defining this leg)
                l4 = <Ymean>          (mean Y value for points defining this leg)
                l5 = <certainty>      (rises +1 every time the leg is re-discovered)
                                      (falls -1 every time the leg is not re-discovered)
                                      (when this reaches 0, the leg is deleted)
                l6 = <Certainty_max>  (maximum value the certainty can reach)
legInfo=n       Return information on the leg with ID = n (for SMRCL, the
                 same format as for "nearest")
testLegInfo=n   Write information on the leg with ID = n to the console
                 for testing purposes.
see also: SCANGET and SCANSET

--- available variables ---
The command "var pplfinder" can be used to see help for the variables
You can get/set the variables with the command "var pplfinder.<variableName>"

--- plugin-to-plugin interface ---
Both callGlobal and callGlobalV responses are available with the commands;
"var call=pplfinder.legInfo(n)"      (where n is the ID for the desired leg)
"var call=pplfinder.testLegInfo(n)"  (where n is the ID for the desired leg)
```

Appendix B

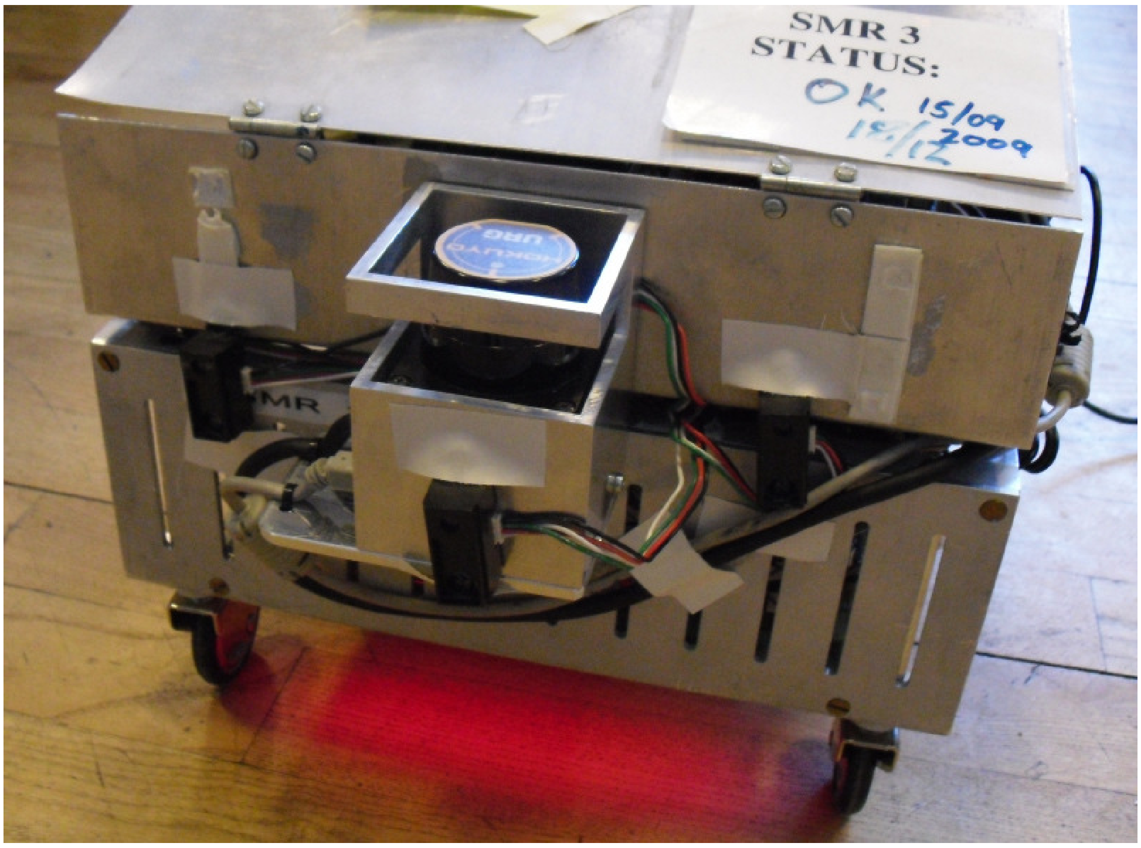


Figure 14: Close-up on the alternative placement of the laser scanner.

Appendix C

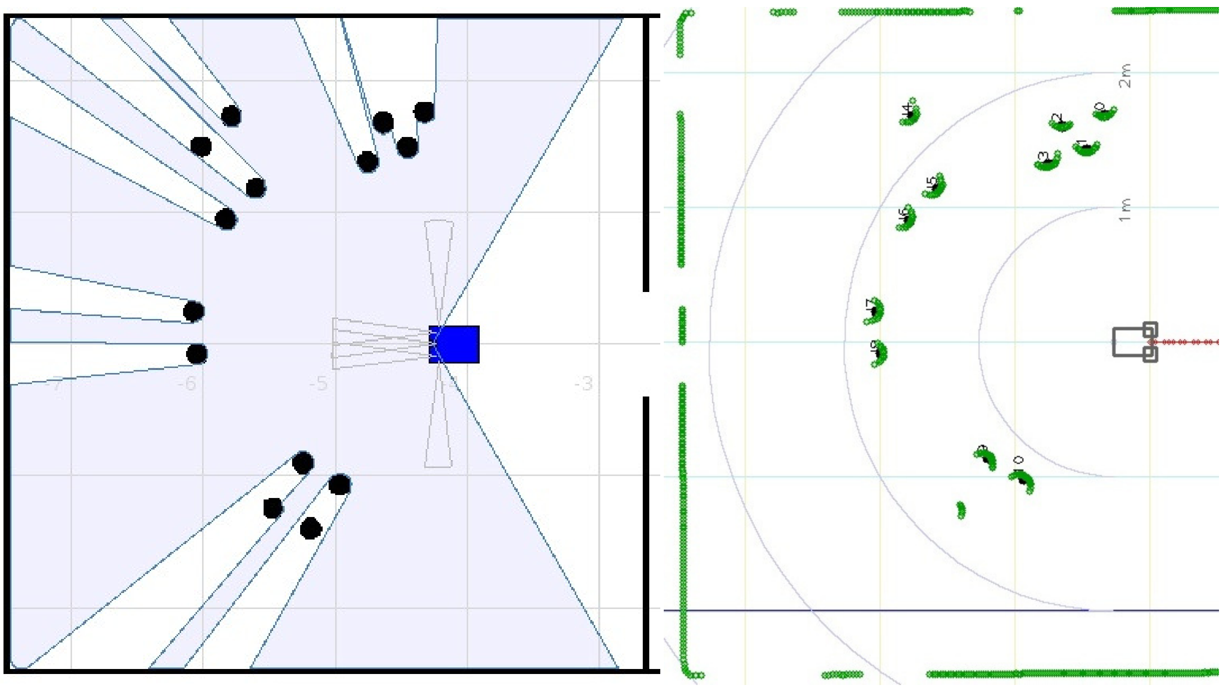


Figure 15: Left room test images

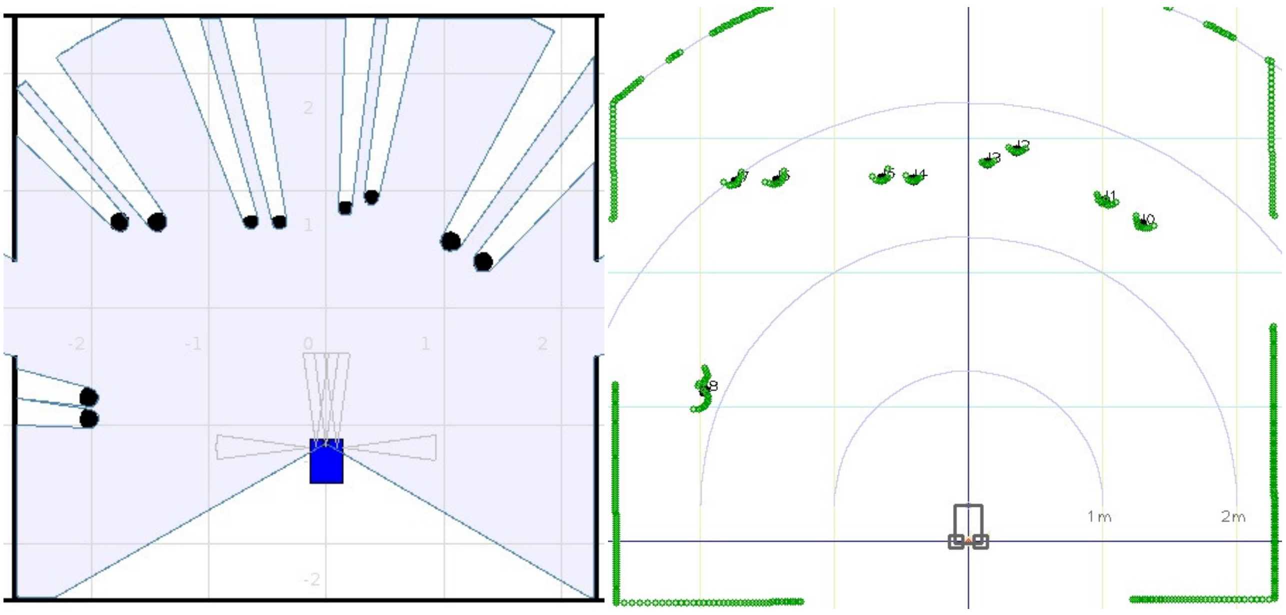


Figure 16: Middle room test images

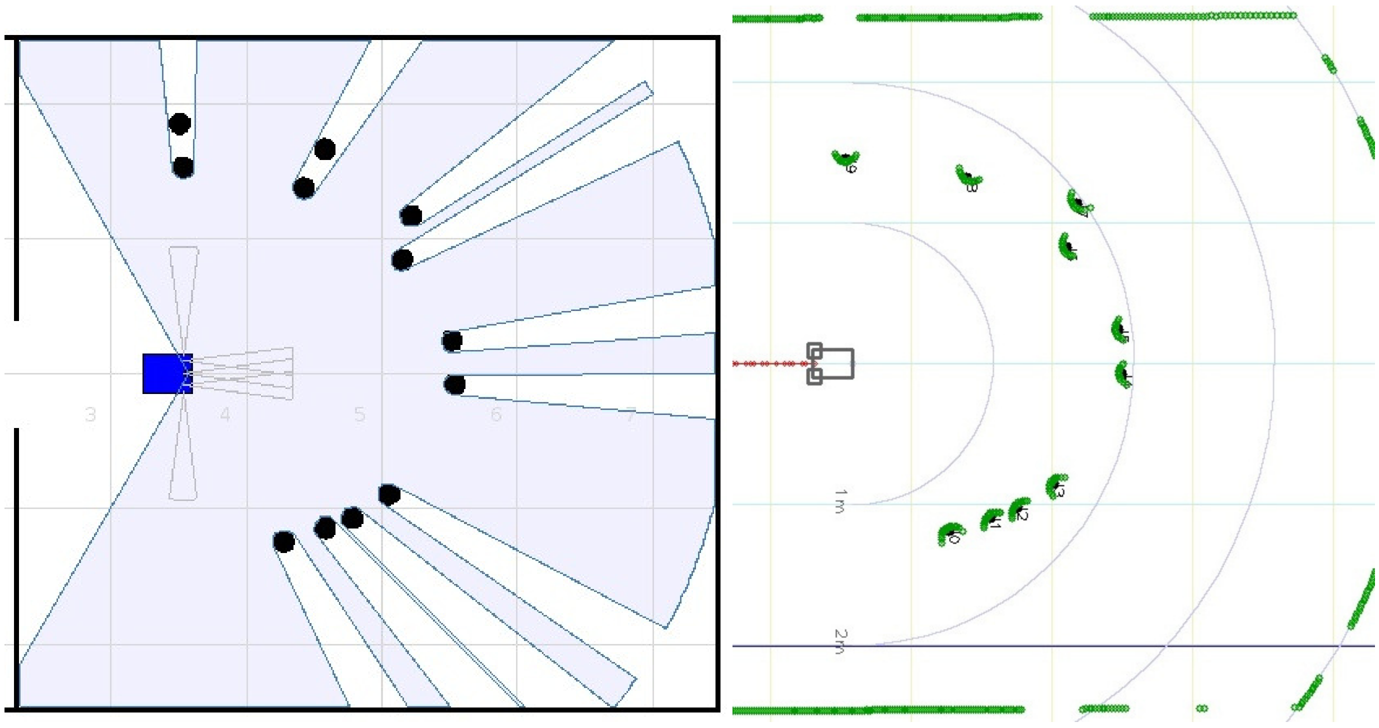


Figure 17: Right room test images

References

[1] marti_giusti_ICRA10.pdf (available from the CD)

[2] Patrizia Marti, Assistant Professor, University of Siena (Information obtained attending the Playware conference of 9th September 2010, Mariott Hotel, Copenhagen)

[3] (04/01-2011) http://timmy.elektro.dtu.dk/rse/wiki/index.php/AU_Robot_Servers#Servers

[4] Daniel Muhle-Zimino – "Software for long-term operational service robot", Ørsted – DTU 2007.

[5] (17/01-2011) http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html

[6] (05/01-2011) http://playerstage.sourceforge.net/

CD Contents

As the files provided here are configured to work with a specific folder structure. Several path changes might need to be made before they can be run.

To do a completely new installation, get Mobotware from http://timmy.elektro.dtu.dk and copy the entire folder "aupplfinder" from this CD for direct use. After "making" the plugin, `aupplfinder.so.0` is created and can be loaded in the ini file of the ulmsserver.

Folders on the CD:

Live
Contains all files used for running the live tests on smr3

Mobotware
Contains the trunk version Mobotware, including the folder "aupplfinder" in which all code files developed and used for this project can be found, including a make-file.

pdf
Contains the pdf version of this report, as well as the pdf mentioned in the references.

Simulator
Contains all files used for running the simulation tests (configured to run on smr3)

Videos
Contains videos demonstrating the functionalities of the plugin:

Video1 – Screen capture of simulation demonstration.

Video2 –Demonstration testing detection of pants shapes.

Video3 – Demonstration testing tracking at low movement speed.

Video4 – Demonstration of the SMRCL-script to follow the nearest leg.

Developed C++ code

As per request, the developed code is included here in addition to the digital copy on the CD.

follow_nearest (SMRCL-code)

```
PI = 3.14159265
laser "push t=0.2 cmd='pplfinder run' n=10000"
laser "push t=0.2 cmd='pplfinder nearest' n=10000"
laser "var pplfinder.certaintyMAX=4"
laser "var pplfinder.maxDist=0.2"
log "$l0" "$l1" "$l2" "$l3" "$l4" "$l5" "$l6"
l1First = $l1

label "newUpdate"
l1Last = $l1
id = $l1
xw = $l3
yw = $l4
x = $l3-$odox
y = $l4-$odoy
dist = sqrt(abs(($l3-$odox))*abs(($l3-$odox))+abs(($l4-$odoy))*abs(($l4-$odoy)))

if($l5 < $l6 | (x == 0 & y == 0) | sqrt(x*x+y*y) > 1.5) "checkScanID"
angle = atan(y/x)

%negative x
if ( x > 0 ) "skipNegX"

%4th quadrant
if ( y > 0 ) "skip4th"
angle = PI + angle
label "skip4th"

%3rd quadrant
if ( y < 0 ) "skip3rd"
angle = -PI + angle
label "skip3rd"

label "skipNegX"

stringcat "angle=" angle
stringcat "odoth=" $odoth

thdiff = angle - $odoth

thdiffdeg = thdiff*(180/PI)

stringcat "world coord: x=" xw "y=" yw
stringcat "robot coord: x=" x "y=" y "th=" angle
stringcat "th diff: thdiff=" thdiff
stringcat "th diff degrees: th=" thdiffdeg

if (abs(thdiffdeg) < 5 & sqrt(abs(($l3-$odox))*abs(($l3-$odox))+abs(($l4-$odoy))*abs(($l4-$odoy))) < 0.6)
"newUpdate"
turn thdiffdeg
drive @v0.5 :(sqrt(abs(($l3-$odox))*abs(($l3-$odox))+abs(($l4-$odoy))*abs(($l4-$odoy))) <
0.6)|($drivendist > (dist/2))
stop
resetmotors
label "skipstop"

goto "newUpdate"

label "checkScanID"
wait 0.19
if (l1Last != $l1) "newUpdate"
goto "checkScanID"

label "stop"
stop
```

follow_VIP (SMRCL-code)

```
%---initialization---
PI = 3.14159265
laser "push t=0.2 cmd='pplfinder run' n=10000"

laser "var pplfinder.certaintyMAX=15"
laser "var pplfinder.maxDist=0.26"

log "$l0" "$l1" "$l2" "$l3" "$l4" "$l5" "$l6"

GOALDIST = 0.60

HALFWIDTH = 0.3
DETECTDIST = 1

label "lost"
stop
speak "lost"
wait 0.1
%---wait and search---
label "state0"
scanID = $l1
wait 0.1
laser "push cmd='pplfinder nearest' n=1"

%stringcat "prevLaser= " scanID
%stringcat "currLaser= " $l1
if(scanID == $l1) "state0" %is this a new laserscan?

if($l5 < $l6) "state0" %if not maximized certainty

xw = $l3
yw = $l4
stringcat " xw: " xw
stringcat " yw: " yw

anglew = atan(yw/xw)

if(xw >= 0) "skipNegXw0"

if(yw >=0) "skipNegYw0"15
anglew = -(PI/2) + anglew
label"skipNegYw0"

if(yw < 0) "skipPosYw0"
anglew = (PI/2) + anglew
label"skipPosYw0"

label "skipNegXw0"

thwdeg = anglew * (180/PI)

diffX = $odox * -1
diffY = $odoy * -1
diffth =  $odoth * -1

trans diffX diffY diffth xw yw thwdeg

stringcat " xr0: " $res0
stringcat " yr0: " $res1

xr = $res0
yr = $res1

if( yr > HALFWIDTH | yr < HALFWIDTH*-1 | xr > DETECTDIST | xr < 0.40) "state0"

IDtoFollow = $l0
scanNo = 0

speak "found"
wait 0.1

%---follow leg until lost---
```

```
label "state1stop"
stop
%wait 2
%resetmotors
label "state1"
scanNo = $l1
wait 0.05
stringcat "push cmd='pplfinder legInfo=" IDtoFollow "' n=1"
laser "$string"

if ( $l0 < 0 ) "lost" %leg lost, restart

if(scanNo == $l1) "state1" %is this a new laserscan?

xw = $l3
yw = $l4
stringcat " xw: " xw
stringcat " yw: " yw

anglew = atan(yw/xw)

if(xw >= 0) "skipNegXw"

if(yw >=0) "skipNegYw"
anglew = -(PI/2) + anglew
label"skipNegYw"

if(yw < 0) "skipPosYw"
anglew = (PI/2) + anglew
label"skipPosYw"

label "skipNegXw"

thwdeg = anglew * (180/PI)

diffX = $odox * -1
diffY = $odoy * -1
diffth =  $odoth * -1

trans diffX diffY diffth xw yw thwdeg

stringcat " xr: " $res0
stringcat " yr: " $res1

xr1 = $res0
yr1 = $res1
thr1 = atan(yr1/xr1)*180/PI

stringcat "thr: " thr1
stringcat "thw: " thwdeg

dist = sqrt( xr1*xr1 + yr1*yr1 )
stringcat "dist to target:" dist

if ( abs(thr1) < 10  | dist > GOALDIST) "skipturn"
toturn = thr1
turn toturn @v0.2
stop
label "skipturn"

if(dist < GOALDIST | abs(thr1) < 1) "state1stop"
driveon xw yw thr1 @v0.25 :($drivendist > dist/5 | sqrt(abs(($l3-$odox))*abs(($l3-$odox))+abs(($l4-
$odoy))*abs(($l4-$odoy))) < GOALDIST)
%stop
goto "state1"
```

ufuncpplfinder.h

```
/***************************************************************************
 *   Copyright (C) 2011 by Mikkel Viager and DTU                           *
 *   s072103@student.dtu.dk                                                *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 ***************************************************************************/
#ifndef UFUNC_NEARGET_H
#define UFUNC_NEARGET_H

#include <cstdlib>

#include <ulms4/ufunclaserbase.h>
#include <urob4/uresposehist.h>
#include "Leg.h"
#include "../ugen4/upolygon.h"

////////////////////////////////////////////////////////

/**
 * Laserscanner function to demonstrate
 * simple laser scanner data handling and analysis
 * @author Christian Andersen
*/

class UFuncPpl : public UFuncLaserBase
{
public:
  /**
  Constructor */
  UFuncPpl()
  {
      //create option vars
      varCertaintyMax = addVar("certaintyMax", 5.0, "d", "maximum certainty status the legs can attain");
      varMaxDist = addVar("maxDist", 0.09, "d", "maximum distance difference on two scans allowed for leg
to be verified as \"same\"");

      //create vars
      varLegsInView = addVar("legsInView", 0.0, "d", "the number of legs currently in view");
      varPplInView = addVar("pplInView", 0.0, "d", "the number of people currently in view");
      varClosestLegDist = addVar("closestLegDist", 0.0, "d", "distance in meters to the closest leg");
      varClosestLegID = addVar("closestLegID", 0.0, "d", "the ID number of the closest leg");
      varBestLegID = addVar("bestLegID", 0.0, "d", "the ID number of the leg with best certainty");

      //create methods
      addMethod("legInfo", "d", "return array of info for given leg (ID). On the form [ID, timestamp,
Xmean, Ymean, Certainty, Certainty_max]");
      addMethod("testLegInfo", "d", "print array of info for given leg (ID). On the form [ID, timestamp,
Xmean, Ymean, Certainty, Certainty_max]");

      // set the command (or commands) handled by this plugin
        setCommand( "pplfinder", "pplFinder", "Detects and provides info on people, based on laser scan
data");

        UDataDouble d;
        d.setVal(0);
        for (int i = 0; i != 6; i++)
            legInfo.push_back(d);

  }
  /**
  Handle incomming command
  (intended for command separation)
  Must return true if the function is handled -
  otherwise the client will get a failed - reply */
  virtual bool handleCommand(UServerInMsg * msg, void * extra);
  bool compareResultLegs(std::vector<Leg> * prev, std::vector<Leg> * curr);
  bool drawPolygons(int polygons, int option, std::vector<Leg> * legs);
  bool createPolygon(Leg * leg, UPolygon * poly);
  bool robotToOdoCooTransf(std::vector<std::vector<UPosition> > *  points, UResPoseHist * odoPose, UPosi-
tion * offset);
  bool calculateVars();
  bool updateVars();
  bool methodCall(const char * name, const char * paramOrder,
                      UVariable ** params,
                      UDataBase ** returnStruct,
                      int * returnStructCnt);
  bool methodCall(const char * name, const char * paramOrder,
            char ** strings, const double * doubles,
            double * value,
            UDataBase ** returnStruct,
            int * returnStructCnt);

private:
    std::vector<Leg> prevLegsInView;
    std::vector<Leg> currLegsInView;
    std::vector<UDataDouble> legInfo;

    UResPoseHist * odoPose;
    int ID;                         //next available ID

    // Parameters and parameter pointers (set in .ini file)

    UVariable * varCertaintyMax;
    int certainty_max;              //max value of certaincy

    UVariable * varMaxDist;
    double max_dist;

    // vars and varPointers

    UVariable * varLegsInView;
    double legsInView;

    UVariable * varPplInView;
    double pplInView;

    UVariable * varClosestLegDist;
    double closestLegDist;

    UVariable * varClosestLegID;
    double closestLegID;

    UVariable * varClosestLegPos;
    double closestLegPos[2];

    UVariable * varBestLegID;
    double bestLegID;

};

#endif
```

ufuncpplfinder.cpp

```
/***************************************************************************
 *   Copyright (C) 2011 by Mikkel Viager and DTU                          *
 *   s072103@student.dtu.dk                                                *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 ***************************************************************************/

#include "ufuncpplfinder.h"
#include "zimino_pDetection.h"
#include "urespplfinder.h"

#ifdef LIBRARY_OPEN_NEEDED

/**
 * This function is needed by the server to create a version of this plugin */
UFunctionBase * createFunc()
{ // create an object of this type
  /** replace 'UFuncNear' with your classname */
  return new UFuncPpl();
}
#endif

///////////////////////////////////////////////////

bool UFuncPpl::handleCommand(UServerInMsg * msg, void * extra)
{  // handle a plugin command
  const bool DEBUG = false;
  certainty_max = varCertaintyMax->getInt(0);
  max_dist = varMaxDist->getDouble(0); // max acceptable distance a leg can move bewteen to consecutive
scans

  const int MIN_LASER_HEIGHT = 0.10; // in meters
  const int MRL = 500;
  char reply[MRL];
  bool ask4help;
  const int MVL = 30;
  char value[MVL];
  double * pdvalue;
  double dvalue;
  pdvalue = &dvalue;
  ULaserData * data;
  //UResPoseHist * odoPose;
  UPoseTime pose;
  ULaserPool * lasPool;
  UPosition lasPlacementOffset;
  //
  //int i;
  //double r;
  //double minRange; // min range in meter
  //double minAngle = 0.0; // degrees
  double d1 = 0.25, d2, h;

  bool gotHeading = false;

  // vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
  std::vector<std::vector<UPosition> > * zLegsAsPoints = new std::vector<std::vector<UPosition> >;
//vectors containing vectors of Upositions
  std::vector<Leg> * pPrevLegsInView = &prevLegsInView;
  std::vector<Leg> * pCurrLegsInView = &currLegsInView;

  // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  // check for parameters - one parameter is tested for - 'help'
  ask4help = msg->tag.getAttValue("help", value, MVL);
  gotHeading = msg->tag.getAttDouble("heading", &d2, 5.0);
  if (ask4help)
  { // create the reply in XML-like (html - like) format
    sendHelpStart(msg, "pplfinder");
    sendText(msg, "--- available pplfinder options ---\n");
    sendText(msg, "help              This message\n");
    sendText(msg, "run               Run pplfinder; analyze current laserscan, and update all\n");
    sendText(msg, "                   variables and history.                                  \n");
    sendText(msg, "                  (Must be run every time you want to update. This is the \n");
    sendText(msg, "                   command you want to \"push\").                          \n");
    sendText(msg, "nearest           Return information for the nearest leg in XML-like format\n");
    sendText(msg, "                   for SMRCL, with the variables;                          \n");
    sendText(msg, "                  l0 = <ID>              (the unique ID-number for this leg)\n");
    sendText(msg, "                  l1 = <scanID>          (the laser scan ID-number)\n");
    sendText(msg, "                  l2 = <timestamp>       (time since plugin load)\n");
    sendText(msg, "                  l3 = <Xmean>           (mean X value for points defining this leg)\n");
    sendText(msg, "                  l4 = <Ymean>           (mean Y value for points defining this leg)\n");
    sendText(msg, "                  l5 = <certainty>       (rises +1 every time the leg is re-
discovered)\n");
    sendText(msg, "                                         (falls -1 every time the leg is not re-
discovered)\n");
    sendText(msg, "                                         (when this reaches 0, the leg is deleted)\n");
    sendText(msg, "                  l6 = <Certainty_max>   (maximum value the certainty can reach)\n");
    sendText(msg, "legInfo=n         Return information on the leg with ID = n (for SMRCL, the\n");
    sendText(msg, "                   same format as for \"nearest\")                            \n");
    sendText(msg, "testLegInfo=n     Write information on the leg with ID = n to the console \n");
    sendText(msg, "                   for testing purposes.                                   \n");
    sendText(msg, "see also: SCANGET and SCANSET\n");
    sendText(msg, "\n");
    sendText(msg, "--- available variables ---\n");
    sendText(msg, "The command \"var pplfinder\" can be used to see help for the variables\n");
    sendText(msg, "You can get/set the variables with the command \"var pplfinder.<variableName>\"\n");
    sendText(msg, "\n");
    sendText(msg, "--- plugin-to-plugin interface ---\n");
    sendText(msg, "Both callGlobal and callGlobalV responses are available with the commands;\n");
    sendText(msg, "\"var call=pplfinder.legInfo(n)\"       (where n is the ID for the desired leg)\n");
    sendText(msg, "\"var call=pplfinder.testLegInfo(n)\"   (where n is the ID for the desired leg)\n");
    sendText(msg, "\n");

    sendHelpDone(msg);
  }
  else
  { // do some action and send a reply

      //to get info on placement of laser-scanner
      lasPool = (ULaserPool *)getStaticResource("lasPool", false);
      if(lasPool != NULL)
      {
          lasPlacementOffset = lasPool->getDefDevice()->getDevicePos();
          if(lasPlacementOffset.z < MIN_LASER_HEIGHT)
              printf("WARNING: Your Laser scanner is placed at %fm height, which is too low!\n"
                      "You should place the laser scanner at least %dcm above the ground!\n"
                      ,lasPlacementOffset.z, MIN_LASER_HEIGHT);
      }

    data = getScan(msg, (ULaserData*)extra);

      odoPose = (UResPoseHist *) getStaticResource("odoPose", true);
      if (odoPose != NULL)
      {
        pose = odoPose->getNewest();
        h = odoPose->getHistHeading(d1, d2, &pose, NULL);
        //snprintf(reply, MRL, "<pplfinder histHeading=\"%g\"/>\n", h);
        //sendMsg(reply);
      }
      else
        sendWarning("no odometry pose history");
    //
    if (data->isValid())
    {
      // vvvvvvvvvvvvvvvvvvvvvvvvvvvvv

      if (msg->tag.getAttValue("run", value, MVL))
      {
          //Update prevLegsInView, and clear (make ready) currLegsInView;
          prevLegsInView = currLegsInView;
          currLegsInView.clear();

          //printf("RUNNING ZIMINO's PEOPLE_DETECTION\n");

          zLegsAsPoints->clear();
          handlePeopleDetection (data, zLegsAsPoints);
          //transfer from robot to odometry coordinates
          robotToOdoCooTransf(zLegsAsPoints, odoPose, &lasPlacementOffset);

          std::vector<std::vector<UPosition> >::size_type i;
          double time = (double) data->getScanTime().GetDecSec();
          long scanID = data->getSerial();

          for ( i = 0; i != zLegsAsPoints->size(); i++)
          {
              Leg* tempLeg = new Leg( ID+i, scanID, time, (*zLegsAsPoints)[i], DEBUG);
              //Create a new leg with the given ID, time, data and decide if we want debug output
              pCurrLegsInView->push_back((*tempLeg));
          }

          if(ID < 2147483000) //take care of overflow (very unlikely)
              ID += i;
          else
              ID = 0;

          //compare previous and current legs viewed, and store result in currLegsInView
          compareResultLegs(pPrevLegsInView, pCurrLegsInView);

          if(drawPolygons(pCurrLegsInView->size(), 1, pCurrLegsInView)&& DEBUG)
              printf("createPolygons have been successfully run for %d Leg(s)...!\n",pCurrLegsInView-
>size());

      }
      else if(msg->tag.getAttDouble("legInfo", pdvalue, 1.0))
      {
          UDataBase *pd[6] = {NULL, NULL, NULL, NULL, NULL, NULL};
          UVariable *par;

          UVariable uvar;
          uvar.setValued(*pdvalue); // pass on the variable
          par = &uvar;

          int n = 6;

          bool isOK = callGlobalV("pplfinder.legInfo", "d", &par, pd, &n);

          if(n > 5 && isOK)
          {
               snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\"
l6 =\"%g\" />\n",
                       ((UDataDouble*) pd[0])->getVal(),  // Leg ID
                       ((UDataDouble*) pd[1])->getVal(),  // Scan ID
                       ((UDataDouble*) pd[2])->getVal(),  // Timestamp
                       ((UDataDouble*) pd[3])->getVal(),  // Xmean
                       ((UDataDouble*) pd[4])->getVal(),  // Ymean
                       ((UDataDouble*) pd[5])->getVal(),  // Certainty
                       (double) certainty_max);           // Certainty_max

          }
          else
          {
              snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\" l6
=\"%g\" />\n",
                       -1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0);
          }
          // send this string as the reply to the client
          sendMsg(reply);

      }else if(msg->tag.getAttDouble("nearest", pdvalue, 1.0))
      {
          UDataBase *pd[6] = {NULL, NULL, NULL, NULL, NULL, NULL};
          UVariable *par;
          UVariable uvar;
          uvar.setValued(closestLegID); // pass on the variable
          par = &uvar;

          int n = 6;

          bool isOK = callGlobalV("pplfinder.legInfo", "d", &par, pd, &n);

          if(n > 5 && isOK)
          {
               snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\"
l6 =\"%g\" />\n",
                       ((UDataDouble*) pd[0])->getVal(),  // Leg ID
                       ((UDataDouble*) pd[1])->getVal(),  // Scan ID
                       ((UDataDouble*) pd[2])->getVal(),  // Timestamp
                       ((UDataDouble*) pd[3])->getVal(),  // Xmean
                       ((UDataDouble*) pd[4])->getVal(),  // Ymean
                       ((UDataDouble*) pd[5])->getVal(),  // Certainty
                       (double) certainty_max);           // Certainty_max

          }
          else
          {
              snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\" l6
=\"%g\" />\n",
                       -1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0);
          }
          // send this string as the reply to the client
          sendMsg(reply);
      }else if(msg->tag.getAttDouble("best", pdvalue, 1.0))
      {
          UDataBase *pd[6] = {NULL, NULL, NULL, NULL, NULL, NULL};
          UVariable *par;

          UVariable uvar;
          uvar.setValued(bestLegID); // pass on the variable
          par = &uvar;

          int n = 6;

          bool isOK = callGlobalV("pplfinder.legInfo", "d", &par, pd, &n);

          if(n > 5 && isOK)
          {
               snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\"
l6 =\"%g\" />\n",
                       ((UDataDouble*) pd[0])->getVal(),  // Leg ID
                       ((UDataDouble*) pd[1])->getVal(),  // Scan ID
                       ((UDataDouble*) pd[2])->getVal(),  // Timestamp
                       ((UDataDouble*) pd[3])->getVal(),  // Xmean
                       ((UDataDouble*) pd[4])->getVal(),  // Ymean
                       ((UDataDouble*) pd[5])->getVal(),  // Certainty
                       (double) certainty_max);           // Certainty_max

          }
          else
          {
              snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\" l6
=\"%g\" />\n",
                       -1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0);
          }
          // send this string as the reply to the client
          sendMsg(reply);
      }else if(msg->tag.getAttDouble("testLegInfo", pdvalue, 1.0))
      {
          UDataBase *pd[6] = {NULL, NULL, NULL, NULL, NULL, NULL};
          UVariable *par;

          UVariable uvar;
          uvar.setValued(*pdvalue); // pass on the variable
          par = &uvar;

          int n = 6;

          bool isOK = callGlobalV("pplfinder.legInfo", "d", &par, pd, &n);
          if(n > 5 && isOK)
          {
               snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\"
l6 =\"%g\" />\n",
                       ((UDataDouble*) pd[0])->getVal(),  // Leg ID
                       ((UDataDouble*) pd[1])->getVal(),  // Scan ID
                       ((UDataDouble*) pd[2])->getVal(),  // Timestamp
                       ((UDataDouble*) pd[3])->getVal(),  // Xmean
                       ((UDataDouble*) pd[4])->getVal(),  // Ymean
                       ((UDataDouble*) pd[5])->getVal(),  // Certainty
                       (double) certainty_max);           // Certainty_max
               callGlobalV("pplfinder.testLegInfo", "d", &par, pd, &n);
          }
          else
          {
              snprintf(reply, MRL, "<laser l0=\"%g\" l1=\"%g\" l2=\"%g\" l3=\"%g\" l4=\"%g\" l5=\"%g\" l6
=\"%g\" />\n",
                       -1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0);
          }
          // send this string as the reply to the client
          sendMsg(reply);

      }

      else
        sendDebug ("Command not handled (by me)");
      //return result;
      // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    }
    else
      sendWarning(msg, "No scandata available");
  }

  // return true if the function is handled with a positive result
  return true;
}

//compare previous and current legs viewed, and store result in currLegsInView
bool UFuncPpl::compareResultLegs(std::vector<Leg> * prev, std::vector<Leg> * curr)
{
    //printf("running compare %d times with %d prev. legs\n", curr->size(), prev->size());
    std::vector<Leg>::size_type i,j;
    for (i = 0 ; i!=curr->size(); i++)
    {
        //printf("\nTest on current leg no. %d\n",i);
        int bestMatchAt = -1;              //invalid vector position

        double bestMatchDist = 100.0;      //initial not acceptable distance
        std::vector<Leg>::iterator iter = prev->begin();
        std::vector<Leg>::iterator iterBestMatchAt;

        //go through all legs in
        for (j = 0 ; j != prev->size(); j++ )
        {
            //find distance between this and leg no. i
            double distDiff;

            anyCooPhytagoras((*prev)[j].getXmean(), (*curr)[i].getXmean(),
                    (*prev)[j].getYmean(), (*curr)[i].getYmean(), &distDiff);
            //see if this has a closer distance
            //printf("distdiff = %f\n",distDiff);

            if(distDiff < bestMatchDist)
            {
                bestMatchAt = j;
                iterBestMatchAt = iter;
                bestMatchDist = distDiff;

                iter++;
            }

        }
        //printf("best match diff: %f\n", bestMatchDist);
        if(bestMatchDist < max_dist && bestMatchAt != -1)
        {
            //printf("found an acceptable match for leg no. %d \n",i);

                    //do something with trajectory?
            (*curr)[i].setID((*prev)[bestMatchAt].getID());
            (*curr)[i].setPosHist((*prev)[bestMatchAt].getPosHist());     //save history
            (*curr)[i].addCertainty((*prev)[bestMatchAt].getCertainty(), certainty_max); //add saved cer-
tainty
            (*curr)[i].setColor((*prev)[bestMatchAt].getColor());    //save color
            if((*curr)[i].getCertainty() == certainty_max)
                (*curr)[i].setColorChar('k');
            else
                (*curr)[i].setColorChar('b');
            //printf("Detected leg now has certainty %d\n",(*curr)[i].getCertainty());

            //remove to indicate that this has been used already
                prev->erase(iterBestMatchAt);
        }else
        {
            //new leg found
        }

    }

    //go through all legs from previously, that was not matched
    for (i = 0 ; i!=prev->size(); i++)
    {
        //printf("Size of prev: %d\n", prev->size());
        (*prev)[i].addCertainty(-1, certainty_max);        //subtract one from certainty

        if((*prev)[i].getCertainty() > 0)   //if leg has a build-up certaincy
        {
            //This leg has been successfully matched before
            //the inability to detect the leg must have been an error.
            //include in current legs, as if this was detected
            curr->push_back((*prev)[i]);
            //printf("Saved undetected Leg (ID %d)! certainty now at %d\n",
(*prev)[i].getID(),(*prev)[i].getCertainty());
            //printf("Position was: Xmean = %f, Ymean = %f\n",(*prev)[i].getXmean(),
(*prev)[i].getYmean());
        }
    }


    calculateVars();
    updateVars();
    return true;
}


//adapted from copyCellPolys in uresavoid.cpp
bool UFuncPpl::drawPolygons(int polygons, int option, std::vector<Leg> * legs)
{
UResBase * pres;
  int i, n;
  const int MSL = 30;
  char s[MSL];
  UVariable * par[3];
  UVariable vs;
  UVariable vr;
  UVariable vCoo;
  UDataBase * db, *dbr;
  bool isOK;
  UPolygon40 poly;
  //
  pres = getStaticResource("poly", false, false);
  if (pres != NULL && polygons > 0)
  { // delete old footprint polygons in polygon resource
    snprintf(s, MSL, "legPoly.*");
    vs.setValues(s, 0, true);
    par[0] = &vs;
    dbr = &vr;
    isOK = callGlobalV("poly.del", "s", par, &dbr, &n);
    //
    // set polygon coordinate system to odometry (0=odo, 1=map, 3=utm)
    vCoo.setDouble(0.0);
    par[2] = &vCoo;
    if (option == 1)
    { // all cells

      for (i = -1; i < polygons; i++)
      {
          if(i == -1){
                //create starting point polygon
                //workaround for first polygon offset. Makes count start at 1
                poly.clear();
                poly.add(0, 0, 0.0);
                poly.color[0] = 'w';
          }else
                isOK = createPolygon(&(*legs)[i] ,&poly); //avcg->getCellPoly(i, &poly);
        if (isOK)
        {
          snprintf(s, MSL, "legPoly.%03d", i);
          vs.setValues(s, 0, true);
          db = &poly;
          par[1] = (UVariable *) db;
          isOK = callGlobalV("poly.setPolygon", "scd", par, &dbr, &n);
          if ((not isOK and i == 0) or not vr.getBool())
            printf("UResAvoid::copyCellPolys: failed to 'poly.setPoly(%s, %d)'\n", s, i);
        } else
            printf("createPolygon failed for polygon no. %d!\n",i);
      }
    }

  } else
      printf("failed on getStaticResource, or polygon number invalid!\n");

    return true;
}

//take info in Leg object, and set options in given poly
bool UFuncPpl::createPolygon(Leg * leg, UPolygon * poly)
{
    poly->clear();
    poly->add(leg->getXmean(), leg->getYmean(), 0.0);
    poly->color[0] = (*leg->getColorChar(0));
    poly->color[1] = (*leg->getColorChar(1));
    poly->color[2] = (*leg->getColorChar(2));
    poly->color[3] = (*leg->getColorChar(3));

    return true;

}

//correct given points in robot coordinates to odometry coordinates
bool UFuncPpl::robotToOdoCooTransf(std::vector<std::vector<UPosition> > *  points, UResPoseHist * odo-
Pose, UPosition * offset)
{

    UPose p = odoPose->getNewest();
    double totX,totY;
    double theta = p.h;
    double odoX = p.x;
    double odoY = p.y;
    double offX = offset->x;
    double offY = offset->y;


    std::vector<std::vector<UPosition> >::iterator iter1= points->begin();
    std::vector<UPosition>::iterator iter2;

    for(iter1 = points->begin(); iter1 != points->end(); iter1++)
    {
        for(iter2 = iter1->begin(); iter2 != iter1->end(); iter2++)
        {
            totX = iter2->x + offX;
            totY = iter2->y + offY;
            //translation and rotation
            iter2->x = (totX*cos(theta) - totY*sin(theta))+odoX;
            iter2->y = (totX*sin(theta) + totY*cos(theta))+odoY;
        }
    }
    //printf("successfuly translated and rotated\n");
    return true;
}

bool UFuncPpl::calculateVars()
{
    // legsInView
    legsInView = currLegsInView.size();

    // pplInView


    // closestLegDist closestLegID bestLegID
    double closest_dist = 4.0;
    int closest_ID = -1;
    int best_certainty = 0;
    int best_ID = -1;

    UPose p = odoPose->getNewest();
    double odoX = p.x;
    double odoY = p.y;

    std::vector<Leg>::iterator iter;

    for (iter = currLegsInView.begin(); iter!=currLegsInView.end(); iter++)
    {
        double legX = iter->getXmean();
        double legY = iter->getYmean();
        double distDiff;

        anyCooPhytagoras(odoX, legX, odoY, legY, &distDiff);

        if(distDiff < closest_dist)
        {
            closest_dist = distDiff;
            closest_ID = iter->getID();
        }

        if(iter->getCertainty() > best_certainty)
        {
            best_certainty = iter->getCertainty();
            best_ID = iter->getID();
        }


    }

    if(closest_ID != -1)
    {
        closestLegDist = closest_dist;
        closestLegID = closest_ID;
    }else
        printf("Error finding closest leg. Data not updated");

    if(best_ID != -1)
    {
        bestLegID = best_ID;
    }else
        printf("Error finding best leg. Data not updated");

    return true;
}

bool UFuncPpl::updateVars()
{
    varLegsInView->setValued(legsInView);
    varPplInView->setValued(pplInView);
    varClosestLegDist->setValued(closestLegDist);
    varClosestLegID->setValued(closestLegID);
    varBestLegID->setValued(bestLegID);
    return true;
```

```
}

//For use with callGlobalV
bool UFuncPpl::methodCall(const char * name, const char * paramOrder,
                       UVariable ** params,
                       UDataBase ** returnStruct,
                       int * returnStructCnt)
{
  bool result = true;

  // evaluate standard functions
  if ((strcasecmp(name, "legInfo") == 0) and (strcmp(paramOrder, "d") == 0))
  {

      legInfo[0].setVal(0);
      legInfo[1].setVal(0);
      legInfo[2].setVal(0);
      legInfo[3].setVal(0);
      legInfo[4].setVal(0);
      legInfo[5].setVal(0);

      std::vector<Leg>::iterator iter;

        for(iter = currLegsInView.begin(); iter != currLegsInView.end(); iter++)
        {
            if (iter->getID() == (int) (*params)->getValued())
            {
                legInfo[0].setVal(iter->getID());           //ID
                legInfo[1].setVal( (double) iter->getScanID()); //scanID
                legInfo[2].setVal(iter->getTimeStamp());    //Timestamp
                legInfo[3].setVal(iter->getXmean());        //Xmean
                legInfo[4].setVal(iter->getYmean());        //Ymean
                legInfo[5].setVal(iter->getCertainty());    //Certainty
            }
        }

      if((int) legInfo[2].getVal() == 0)
      {
          printf("ERROR: a leg with the given ID is not available.\n");
          legInfo[0].setVal(-1);
          legInfo[1].setVal(-1);
          legInfo[2].setVal(-1);
          legInfo[3].setVal(-1);
          legInfo[4].setVal(-1);
          legInfo[5].setVal(-1);
      }

      if(*returnStructCnt > 5)
      {
          returnStruct[0] = &legInfo[0];
          returnStruct[1] = &legInfo[1];
          returnStruct[2] = &legInfo[2];
          returnStruct[3] = &legInfo[3];
          returnStruct[4] = &legInfo[4];
          returnStruct[5] = &legInfo[5];
      }

      //*returnStructCnt = legInfo.size();
  }
  else if((strcasecmp(name, "testLegInfo") == 0) and (strcmp(paramOrder, "d") == 0))
  {
      UDataBase *pd[6] = {NULL, NULL, NULL, NULL, NULL, NULL};
      UVariable *par;

      UVariable uvar;
      uvar.setValued((*params)->getValued()); // pass on the variable(s)
      par = &uvar;

      int n = 6;

      bool isOK = callGlobalV("pplfinder.legInfo", "d", &par, pd, &n);

      if(n > 5 && isOK)
      {
           printf("reply[0] ID = %f\n",((UDataDouble*) pd[0])->getVal());
```

```
            printf("reply[1] scanID = %f\n",((UDataDouble*) pd[1])->getVal());
            printf("reply[2] Timestamp = %f\n",((UDataDouble*) pd[2])->getVal());
            printf("reply[3] Xmean = %f\n",((UDataDouble*) pd[3])->getVal());
            printf("reply[4] Ymean = %f\n",((UDataDouble*) pd[4])->getVal());
            printf("reply[5] Certainty = %f\n",((UDataDouble*) pd[5])->getVal());
      }

      *returnStructCnt = 6;
  }
  else
    result = false;
  return result;
}

//For use with callGlobal
bool UFuncPpl::methodCall(const char * name, const char * paramOrder,
            char ** strings, const double * doubles,
            double * value,
            UDataBase ** returnStruct,
            int * returnStructCnt)
{
    bool result = true;

  // evaluate standard functions
  if ((strcasecmp(name, "legInfo") == 0) and (strcmp(paramOrder, "d") == 0))
  {

      legInfo[0].setVal(0);
      legInfo[1].setVal(0);
      legInfo[2].setVal(0);
      legInfo[3].setVal(0);
      legInfo[4].setVal(0);
      legInfo[5].setVal(0);

      std::vector<Leg>::iterator iter;

        for(iter = currLegsInView.begin(); iter != currLegsInView.end(); iter++)
        {
            if (iter->getID() == *doubles)
            {
                legInfo[0].setVal(iter->getID());           //ID
                legInfo[1].setVal( (double) iter->getScanID()); //scanID
                legInfo[2].setVal(iter->getTimeStamp());    //Timestamp
                legInfo[3].setVal(iter->getXmean());        //Xmean
                legInfo[4].setVal(iter->getYmean());        //Ymean
                legInfo[5].setVal(iter->getCertainty());    //Certainty
            }
        }

      if((int) legInfo[2].getVal() == 0)
      {
          printf("ERROR: a leg with the given ID is not available. Returning [-1,-1,-1,-1,-1]\n");
          legInfo[0].setVal(-1);
          legInfo[1].setVal(-1);
          legInfo[2].setVal(-1);
          legInfo[3].setVal(-1);
          legInfo[4].setVal(-1);
          legInfo[5].setVal(-1);
      }

      if(*returnStructCnt > 5)
      {
          returnStruct[0] = &legInfo[0];
          returnStruct[1] = &legInfo[1];
          returnStruct[2] = &legInfo[2];
          returnStruct[3] = &legInfo[3];
          returnStruct[4] = &legInfo[4];
          returnStruct[5] = &legInfo[5];
      }
      *value = 1;
      *returnStructCnt = legInfo.size();
  }
  else if((strcasecmp(name, "testLegInfo") == 0) and (strcmp(paramOrder, "d") == 0))
  {
      UDataBase *pd[6] = {NULL, NULL, NULL, NULL, NULL, NULL};
      UVariable *params;

      UVariable uvar;
      uvar.setValued(*doubles); // pass on the variable(s)
      params = &uvar;

      int n = 6;


      bool isOK = callGlobalV("pplfinder.legInfo", "d", &params, pd, &n);

      if(n > 5 && isOK)
      {
            printf("reply[0] ID = %f\n",((UDataDouble*) pd[0])->getVal());
            printf("reply[1] scanID = %f\n",((UDataDouble*) pd[1])->getVal());
            printf("reply[2] Timestamp = %f\n",((UDataDouble*) pd[2])->getVal());
            printf("reply[3] Xmean = %f\n",((UDataDouble*) pd[3])->getVal());
            printf("reply[4] Ymean = %f\n",((UDataDouble*) pd[4])->getVal());
            printf("reply[5] Certainty = %f\n",((UDataDouble*) pd[5])->getVal());
      }



      *returnStructCnt = 6;
      *value = 2;
  }
  else
    result = false;
  return result;
}
```

urespplfinder.h

```
/***************************************************************************
 *   Copyright (C) 2011 by Mikkel Viager and DTU                           *
 *   s072103@student.dtu.dk                                                *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 ***************************************************************************/

#ifndef URESPPLFINDER_H
#define URESPPLFINDER_H

void anyCooPhytagoras(double x1, double x2, double y1, double y2, double * c);
void copyChar(char FromCh, char ToCh, int number);

#endif  /* URESPPLFINDER_H */
```

urespplfinder.cpp

```
/***************************************************************************
 *   Copyright (C) 2011 by Mikkel Viager and DTU                          *
 *   s072103@student.dtu.dk                                                *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 ***************************************************************************/

#include <math.h>

void anyCooPhytagoras(double x1, double x2, double y1, double y2, double * c){

    double a, b;

    //find a
    if (x1 < 0)
    {
        if(x2 < 0)
        {
            a = fabs(fabs(x1)-fabs(x2));
        }else
        {
            a = fabs(x1-x2);
        }
    }else
        a = fabs(x2-x1);


    //find b
    if (y1 < 0)
    {
        if(y2 < 0)
        {
            b = fabs(fabs(y1)-fabs(y2));
        }else
        {
            b = fabs(y1-y2);
        }
    }else
        b = fabs(y2-y1);

    (*c) = sqrt(a*a + b*b);
}
```

Leg.h

```
/***************************************************************************
 *   Copyright (C) 2011 by Mikkel Viager and DTU                          *
 *   s072103@student.dtu.dk                                                *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 ***************************************************************************/

//Header file for Leg.cpp
//Defines the Leg-object

#ifndef PPL_LEG_H
#define PPL_LEG_H

#include <vector>
#include <list>
#include "../ugen4/u3d.h"

class Leg {

    public:
        Leg();
        Leg(int ID, long scanID, double time, std::vector<UPosition> p, bool DEBUG);
        void setColor();
        void setColor(char * color);
        void setColorChar(char c);
        char * getColor();
        char * getColorChar(int i);
        void setTimeStamp(double time);
        double getTimeStamp();
        double getXmean();
        double getYmean();
        int getCertainty();
        void addCertainty(int x, int max);
        void setPosHist(std::list<UPosition> * l);
        std::list<UPosition> * getPosHist();
        int getID();
        void setID(int ID);
        long getScanID();

        char color[8];

    private:
        int ID;
        long scanID;
        double timestamp;
        double Xmean;
        double Ymean;
        std::list<UPosition> posHist;

        int points;
        int certainty;

};

#endif
```

Leg.cpp

```
/***************************************************************************
 *   Copyright (C) 2011 by Mikkel Viager and DTU                           *
 *   s072103@student.dtu.dk                                                *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 ***************************************************************************/

// Defines the Leg-Object

#include <list>

#include "Leg.h"
#include "string.h"


Leg::Leg(){}

//Given a vector of Upositions with points defining this leg, create with the
// given values.
Leg::Leg(int ID, long scanID, double time, std::vector<UPosition> p, bool DEBUG)
{
    double tempX = 0.0;
    double tempY = 0.0;
    double count = 0.0;

    //take mean of x and y values
    for (std::vector<UPosition>::size_type i = 0; i !=p.size(); i++)
    {
        tempX += p[i].x;
        tempY += p[i].y;
        count++;
    }

    Xmean = tempX/count;
    Ymean = tempY/count;
    UPosition pos(Xmean, Ymean, 0);
    posHist.push_back(pos);
    timestamp = time;
    points = (int) count;
    setColor();
    certainty = 1;
    this->ID = ID;
    this->scanID = scanID;


    //DEBUG - Print points and mean values to console
    if(DEBUG)
    {
        printf("\nP Leg no. %d\n",ID);
        for (std::vector<UPosition>::size_type j = 0; j != p.size(); j++){
            p[j].print("Point:");
        }
        printf("Mean    x: %f y:  %f\n", Xmean, Ymean);
        printf("Points in leg: %d\n", (int) count);
        printf("Color of leg: %s\n",color);
    }
    //END DEBUG

}
void Leg::setColor()
{
    color[0] = 'r';
    color[1] = '9';
    color[2] = 'o';
    color[3] = 'd';
    color[4] = '"';
    color[5] = '"';
    color[6] = '"';
    color[7] = '\0';
}

void Leg::setColor(char * color)
{
    if(strlen(color) >= 4)
    {
        this->color[0] = color[0];
        this->color[1] = color[1];
        this->color[2] = color[2];
        this->color[3] = color[3];
    }
    else
        printf("ERROR: setColor in Leg had less than 4 characters\n");
}

void Leg::setColorChar(char c)
{
    color[0] = c;
}

char * Leg::getColor()
{

    return this->color;
}

char * Leg::getColorChar(int i)
{
    if (i >= 0 && i < 8)
        return &color[i];
    else
        return 0;
}

double Leg::getTimeStamp()
{
    return timestamp;
}

void Leg::setTimeStamp(double time)
{
    timestamp = time;
}

double Leg::getXmean()
{
    return Xmean;
}
double Leg::getYmean()
{
    return Ymean;
}

int Leg::getCertainty()
{
    return certainty;
}

void Leg::addCertainty(int x, int max)
{
    if(certainty + x <= max)
    {
        certainty += x;
    }else
        certainty = max;

}

//update current position, and store previous in history vector
void Leg::setPosHist(std::list<UPosition> * l)
```

```
{
    std::list<UPosition>::iterator iter = l->begin();
    while(iter != l->end())
    {
        UPosition pos = *iter;
        posHist.push_back(pos);
        iter++;
    }
}

std::list<UPosition> * Leg::getPosHist()
{
    return &posHist;
}

int Leg::getID()
{
    return ID;
}

void Leg::setID(int ID)
{
    this->ID = ID;
}

long Leg::getScanID()
{
    return scanID;
}
```

zimino_PDetection.h

```
/***************************************************************************
 *   DMZ - Headerfile til plugin til menneske-genkendelse                  *
 *   s011359@student.dtu.dk                                                *
 *                                                                         *
 ***************************************************************************/


#ifndef UFUNC_COG_H
#define UFUNC_COG_H

#include <cstdlib>
#include <urob4/ufunctionbase.h>
#include "../ulmsserver/ulaserpool.h"
#include <vector>
#include "Leg.h"

  /**
  Function to get closest distance */
  bool handlePeopleDetection(ULaserData * pushData, std::vector<std::vector<UPosition> > * zLegsAs-
Points);

#endif
```

zimino_PDetection.cpp

```
/****************************************************************************
 *   This file originates from the pDetection ulmsserver-plugin         *
 *   by DMZ - Daniel Muhle-Zimino,s011359@student.dtu.dk                    *
 *   The below is part of the original plugin, modified to create c++       *
 *   objects, instead of writing data to files as it did originally.        *
 *   Modifications by: MV - Mikkel Viager, s072103@student.dtu.dk           *
 ****************************************************************************/

#include "zimino_pDetection.h"
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*  The function is given a set of laserscan-data,
    and creates objects for each leg detected.      */

bool handlePeopleDetection (ULaserData * pushData, std::vector<std::vector<UPosition> > * zLegsAsPoints)
{
  //saves two files in the given directories if true
  bool DEBUG = false;
  if(DEBUG) printf("zimino:pDetection is in DEBUG-mode (a lot of output on console)");

  // DEBUG
  char saveDir[100] = "/home/mikkel/Dropbox/Bachelor/zimino_scanfiles/";
  char saveFile[100] = "/home/mikkel/Dropbox/Bachelor/zimino_scanfiles/";
  // DEBUG END

  ULaserData *data = pushData;

  int r, i, j, k;
  int minR, minR1;            // range in range units
  int counter1 = 0;
  int cou1 = 0;
  int cou2 = 0;
  const int Laserprecision = 181;
  int clausterCount, tjek;
  int testCount = 0;
  int count = 0;
  int testNum;
  int tester = 0;
  int sorted = 0;
  int finalCount = 0;

  // int finalKlyngeAnt[Laserprecision];
  int longestDistNum = 0;
  int klyngeLength[Laserprecision];
  int finalKlyngeSeperator[Laserprecision];
  double smallestMiddleVal;
  double shortest1 = 0;
  double shortest2 = 0;
  double maxMeasureDist = 0;
  double maxDistClausters = 0;
  int finalKlynge[Laserprecision];
  int counterAll = 0;
  double pPIFR[999];          // max registreringer i personindex kan

  // volde probs
  double pPIFAngle[999];      // max registreringer i

  // personindex kan volde probs
  double pPIFRAll[999];
  double pPIFAngleAll[999];
  double personRange = 0.0;
  double personTestDist = 0.0;
  double minDistCal = 0;
  double maxDistCal = 0;
  double xCoord, yCoord, xPrevCoord, yPrevCoord;
  double aTempCal = 0.0;
  double bTempCal = 0.0;
  double klyngeX[Laserprecision][Laserprecision];
  double klyngeY[Laserprecision][Laserprecision];
  double klyngeRange[Laserprecision][Laserprecision];
  double aCal2[181];
  double bCal2[181];
  double shortest = 4.0;
  double sumXY, sumX, sumY, sumXX;
  double longestDist = 0.0;

  double tempDist;
  unsigned long serial;
  float closestMeasurement = 1000;

  // saveFile = "All";
  // SIKRE AT KLYNGE ER Tï¿½MT
  // Ensure the variable for cluster-mesurements are empty
  for (i = 0; i < Laserprecision; i++)
  {
    for (j = 0; j < Laserprecision; j++)
    {
      klyngeX[i][j] = 0;
      klyngeX[i][j] = 0;
      klyngeRange[i][j] = 0;
    }
  }

  //DEBUG
  // Files to write to
  FILE *FileThirdSort = NULL;
  FILE *FileUnsorted = NULL;
  if(DEBUG){
      // The stringconcatination ensures that the files are
      // created/overwritten in the correct path
      FileThirdSort = fopen (strcat (saveDir, "3sort.txt"), "w");
      strcpy (saveDir, saveFile);
      FileUnsorted = fopen (strcat (saveDir, "unsorted.txt"), "w");
      strcat (saveFile, "default");
      strcat (saveFile, ".txt");
  }
  //DEBUG END

    // Ensures that scandata is valid
    if (data->isValid ())
    {

      // make analysis for closest measurement
      minR = 100000;
      minR1 = 100000;
      serial = data->getSerial ();

      // gets data scanned
      for (i = 0; i < data->getRangeCnt (); i++)
      {
       // range are stored as an integer in current units
       r = *data->getRange (i);
       // test if the object is the closer than the given max
       // distance to object set
       if (r < minR)
       {
         count = count + 1;
       }
       // Person scan tests up to 3 meters from robot
       personTestDist = 3.00;

       // Converts data to meter-format
       switch (data->getUnit ())
       {
         case 0:
           personRange = double (r) * 0.01;

           break;             // cm
         case 1:
           personRange = double (r) * 0.001;

           break;             // mm
         case 2:
           personRange = double (r) * 0.1;

           break;             // 10cm
         default:
           personRange = 20.0;
       }
       // Finding the closest object to robot
       if (personRange < closestMeasurement)
       {
         closestMeasurement = personRange;
       }
       // ////////////
       // Data about pPIF (possible Person In Front)
       // PossiblepersonInFront Range
       // Saves all data in this variable
       pPIFRAll[counterAll] = personRange;

       // PossiblePersonInFront Angle
       pPIFAngleAll[counterAll] = data->getAngleDeg (i);

       // Updating counter
       counterAll++;

       // Saves data that lies between reasonable range from
       // laserscanner in this variable
       if ((personRange < personTestDist) and (r >= 20))
       {
         // PossiblepersonInFront Range
         pPIFR[counter1] = personRange;
         // PossiblepersonInFront Angle
         pPIFAngle[counter1] = data->getAngleDeg (i);
         // counter update
         counter1++;
       }
     }

     for (i = 0; i < counter1; i++)
     {
       if (i > 0)
       {
         // Transform data inside the reasonable range to x and
         //
         // y-coordinates
         xCoord = cos (pPIFAngle[i] * M_PI / 180.0) * pPIFR[i];
         yCoord = sin (pPIFAngle[i] * M_PI / 180.0) * pPIFR[i];
         xPrevCoord = cos (pPIFAngle[i - 1] * M_PI / 180.0) * pPIFR[i - 1];
         yPrevCoord = sin (pPIFAngle[i - 1] * M_PI / 180.0) * pPIFR[i - 1];
         aTempCal = 0.0;
         bTempCal = 0.0;

         // The resonable distance between the legs is decided
         // from the the distance to the mesurements (0.0175
         // meters in each side pr. meter to the robot plus a
         // bufferdistance of 2centimeters, see calculations)
         maxMeasureDist = ((0.0175 + 0.0175) * pPIFR[i] + 0.02);

         // test two related mesurements
         if (sqrt
             (pow ((xCoord - xPrevCoord), 2) +
              pow ((yCoord - yPrevCoord), 2)) < maxMeasureDist)
         {

           // Inserts first element in cluster
           if (cou2 == 0)
           {
             // Converted to real x and y-coordinates
             klyngeX[cou1][cou2] = -yPrevCoord;
             klyngeY[cou1][cou2] = xPrevCoord;
             klyngeRange[cou1][cou2] = pPIFR[i - 1];
             cou2++;
           }
           // Inserts current element in cluster
           klyngeX[cou1][cou2] = -yCoord;
           klyngeY[cou1][cou2] = xCoord;
           klyngeRange[cou1][cou2] = pPIFR[i];
```

```
          cou2++;
        }
        // Does not belong to this cluster, and create a new
        // one.
        else if (cou2 > 0)
        {
          // The length of cluster
          klyngeLength[cou1] = cou2;
          cou1++;
          cou2 = 0;
        }
      }
    }

    // Close last cluster
    if (cou2 > 0)
    {
      // The length of cluster
      klyngeLength[cou1] = cou2;
      cou1++;
      cou2 = 0;
    }
    // Logistics calculation of linear regression
    clausterCount = 0;
    testCount = 0;
    sorted = 0;


    // //// TJEK OP Pï¿½ DENNE //////
    // // Skal nok ikke bruges sï¿½ rettelse udelades
    while (klyngeX[clausterCount][0] != 0 or klyngeY[clausterCount][0] != 0)
    {
      if (klyngeX[clausterCount][2] != 0 or klyngeY[clausterCount][2] != 0)      // nr
      {
        sumXY = 0;
        sumX = 0;
        sumY = 0;
        sumXX = 0;

        while (klyngeX[clausterCount][testCount] !=
               0 and klyngeY[clausterCount][testCount] != 0)
        {
          sumXY +=
            klyngeX[clausterCount][testCount] * klyngeY[clausterCount][testCount];
          sumX += klyngeX[clausterCount][testCount];
          sumY += klyngeY[clausterCount][testCount];
          sumXX += pow (klyngeX[clausterCount][testCount], 2);

          testCount++;
        }

        aCal2[clausterCount] =
          ((testCount * sumXY) - (sumX * sumY)) / ((testCount * sumXX) - pow (sumX, 2));
        bCal2[clausterCount] = (sumY - (aCal2[clausterCount] * sumX)) / testCount;
        testCount = 0;
        clausterCount++;
      }
      else
      {
        sorted++;
        clausterCount++;
      }
    }

    for (i = 0; i < cou1; i++)
    {
      if (klyngeLength[i] > 2)
      {
        // Minumum distance of leg-width mesured. Once again
        // the variable of 0.0175meters is used to to
        // calculate the buffer-distance and a standard
        // leg-with of 8centimeters
        minDistCal = 0.08 - klyngeRange[i][0] * 0.0175 * 2;
        maxDistCal = 0.35;
        tjek = 0;
```

```
          // Absolut shortest leg-width possible
          shortest = 4.0;

          // testing parameters for the comming tests
          bool test1 = false;
          bool test2 = false;
          bool test3 = false;
          bool test4 = false;

          // This switch makes for special-cases for specified
          // cluster-length. For example if clusters with 3 or 4
          //
          // mesurements should be tested different than others
          // because of the few mesurements..
          switch (klyngeLength[i])
          {
            default:
              double buf = 0.005 * klyngeRange[i][(int) floor (klyngeLength[i] / 2)];

              // minimum width of cluster in meters
              if (sqrt
                  (pow
                   ((klyngeX[i][klyngeLength[i] - 1] -
                     klyngeX[i][0]),
                    2) + pow ((klyngeY[i][klyngeLength[i] - 1] -
                               klyngeY[i][0]), 2)) > minDistCal)
              {
                test2 = true;
              }
              // maximum width of cluster
              if (sqrt
                  (pow
                   ((klyngeX[i][klyngeLength[i] - 1] -
                     klyngeX[i][0]),
                    2) + pow ((klyngeY[i][klyngeLength[i] - 1] -
                               klyngeY[i][0]), 2)) < maxDistCal)
              {
                test3 = true;
              }
              // cluster-width in mesurements is an equal number
              if (klyngeLength[i] % 2 == 0)
              {
                if (((klyngeRange[i]
                      [(int) klyngeLength[i] / 2 - 1]) <
                     klyngeRange[i][0]
                     and (klyngeRange[i]
                          [(int) klyngeLength[i] / 2 - 1] <
                          klyngeRange[i][klyngeLength[i] - 1]))
                    or ((klyngeRange[i]
                         [(int) klyngeLength[i] / 2]) <
                        klyngeRange[i][0]
                        and (klyngeRange[i]
                             [(int) klyngeLength[i] / 2] <
                             klyngeRange[i][klyngeLength[i] - 1])))
                {
                  test1 = true;
                }
                else
                {}
              }
              else
              {
                // hvis kun 3 ekstra tjek pï¿½ afstanden mellem
                // punkterne

                // If cluster consist of 3 mesurements, a
                // further check on the distance between
                // mesurements is done
                if (klyngeLength[i] == 3)
                {

                  if ((klyngeRange[i]
                       [(int) floor (klyngeLength[i] / 2)] -
                       buf < klyngeRange[i][0])
                      and (klyngeRange[i]
                         [(int) floor (klyngeLength[i] / 2)]
                         - buf < klyngeRange[i][klyngeLength[i] - 1]))
                  {
                    test1 = true;
                  }
                  else
                  {}
                }
                // if width is not equal then it defines which
                //
                // mesurements in the middle is closest
                // related in meters (decides the middle of a
                // possible leg)
                else
                {
                  if (sqrt
                      (pow
                       (klyngeX[i][klyngeLength[i] - 1] -
                       klyngeX[i][0],
                       2) + pow (klyngeY[i][klyngeLength[i] -
                                           1] - klyngeY[i][0], 2)) < 0.22)
                  {
                    for (j = -1; j <= 1; j++)
                    {
                      if (klyngeRange[i]
                          [(int) floor (klyngeLength[i] / 2) + j] < shortest)
                        shortest = klyngeRange[i][(int) floor (klyngeLength[i] / 2)];
                    }
                    if ((shortest <
                         klyngeRange[i][0]) and (shortest <
                                              klyngeRange[i][klyngeLength[i] - 1]))
                    {
                      test1 = true;
                    }
                    else
                    {}
                  }
                  else
                  {}
                }

              }

              // If the previous test sucseeded and cluster
              // consist of more than 6 mesurements it could be
              // a possible pair of legs
              if (test1 == false and klyngeLength[i] > 6)
              {
                for (j = 1; j < klyngeLength[i] - 1 - 1; j++)
                {

                  tempDist =
                    sqrt (pow
                          ((klyngeX[i][j] -
                            klyngeX[i][j + 1]),
                           2) + pow ((klyngeY[i][j] - klyngeY[i][j + 1]), 2));


                  // divides a possible leg-pair where the
                  // distance between the measurements i
                  // largest.
                  if (tempDist > longestDist)
                  {
                    // divide between j and j+1
                    longestDist = tempDist;
                    longestDistNum = j;
                  }
                }
                // Resets distance-variables to a nonpossible
                // large distance
                shortest1 = 4.0;
                shortest2 = 4.0;
                tester = 0;

                // test if the one of the divided clusters is
```

```
// smaller than 3 mesurements or
if (klyngeLength[i] - longestDistNum < 3)
  tester++;;
if (longestDistNum < 3)
  tester++;
// if one side, of the cluster dived, length
// equal 3 mesurements
if (longestDistNum == 3)
{
  // Test if middlepoints of cluster is
  // closer to robot. The described curved
  // line earlier.
  if (klyngeRange[i][1] <
      klyngeRange[i][0] and klyngeRange[i][1] < klyngeRange[i][2])
    tester++;
}
// if the other side, of the cluster dived,
// length equal 3 mesurements
if (klyngeLength[i] - longestDistNum == 3)
{
  // Test if middlepoints of cluster is
  // closer to robot. The described curved
  // line earlier.
  if (klyngeRange[i][klyngeLength[i] - 2] <
      klyngeRange[i][longestDistNum +
                     1] and
      klyngeRange[i][klyngeLength[i] - 2] <
      klyngeRange[i][klyngeLength[i] - 1])
    tester++;
}
// test if the first divided cluster have a
// width larger than three
if (longestDistNum > 3)
{
  // forloop that test every value i cluster
  //
  // except the first and last regarding
  // curved regression
  smallestMiddleVal = 4.0;
  for (k = 1; k < longestDistNum - 1; k++)
  {
    if (klyngeRange[i][k] < smallestMiddleVal)
      smallestMiddleVal = klyngeRange[i][k];
  }
  // test up on the closest middleval in the
  //
  // cluster
  if (smallestMiddleVal <
      klyngeRange[i][0] and smallestMiddleVal <
      klyngeRange[i][longestDistNum - 1])
    tester++;
}
// test if the second divided cluster have a
// width larger than three
if (klyngeLength[i] - longestDistNum > 3)
{
  smallestMiddleVal = 4.0;
  // forloop that test every value i cluster
  //
  // except the first and last regarding
  // curved regression

  for (k = longestDistNum + 2; k < klyngeLength[i] - 1; k++)
  {
    if (klyngeRange[i][k] < smallestMiddleVal)
      smallestMiddleVal = klyngeRange[i][k];
  }
  // test up on the closest middleval in the
  //
  // cluster
  if (smallestMiddleVal <
      klyngeRange[i][longestDistNum +
                     1] and smallestMiddleVal <
      klyngeRange[i][klyngeLength[i] - 1])
    tester++;
```

```
          }

          if (tester == 2)
          {
            test4 = true;
          }
          else
          {}


          }
        // nï¿½dtest tjekker op pï¿½ de to punkter ved siden
        // af


        break;
    }

    //
    if (test3 == true and test2 == true)       // and
    {
      // saves internal the number and amount of
      // measurements to the final decision about
      // person-detection
      // if the first test is passed directly insertion
      if (test1 == true)
      {
        finalKlynge[finalCount] = i;
        finalCount++;
      }
      // if the first test is passed directly insertion
      else if (test4 == true)   // test4==true
      {

        // I KNOW:::::::: EASY FIX WITH FOLLOWUP evt
        // ren indsï¿½ttelse i fil pï¿½ dette tidspunkt
        // eller endda hurtigere
        finalKlynge[finalCount] = i;
        finalKlyngeSeperator[finalCount] = 1;  // longestDistNum;
        finalCount++;
      }

      testNum = 1;
      // skud pï¿½ maks benbredde og min benbredde 10 og
      // 40cm (et henholdsvis 2 ben
      // desuden antages det at et ben er rundt dvs
      // robotten registrerer en person som buet
    }
  }
}

// HUSK AT ï¿½NDRE 180 TILBAGE TIL LASERPRECISION
int z;

//DEBUG
if(DEBUG){
    // Save data for every degree mesured
    for (z = 0; z < 180; z++)
    {
      // Saves data fullfilling all tests
      fprintf (FileUnsorted, "%g       %g \n",
               -sin (pPIFAngleAll[z] * M_PI / 180.0) *
               pPIFRAll[z], cos (pPIFAngleAll[z] * M_PI / 180.0) * pPIFRAll[z]);
    }
}
//DEBUG END

maxDistClausters = 0.30;        // fixed size

for (i = 0; i < finalCount; i++)
{
  // test the clusters op against each other from left
  // against right
  if (finalKlyngeSeperator[i] == 0 and i < finalCount - 1)
  {
          for (k = i + 1; k < finalCount; k++)
          {
          // Test of the distance between the to clusters
               if (sqrt (pow ((klyngeX[finalKlynge[i]]
                          [klyngeLength[finalKlynge[i]] - 1] -
                          klyngeX[finalKlynge[k]][0]),
                         2) + pow ((klyngeY[finalKlynge[i]]
                                   [klyngeLength
                                    [finalKlynge[i]] - 1] -
                                   klyngeY[finalKlynge[k]][0]), 2)) < maxDistClausters)
            {
            }
          }
        }
      }

      //Added by Mikkel Viager, s072103@student.dtu.dk

      //convert saved data to Upositions, and save to the pointvector in the legvector

      std::vector<UPosition> tempVec;
      UPosition pos;

      for (i = 0; i < finalCount; i++)
      {
            (*zLegsAsPoints).push_back(tempVec);
            if(DEBUG)printf("\n Leg no. %d\n",i);

            for (j = 0; j < klyngeLength[finalKlynge[i]]; j++)
            {
                //transfer to odometry coordinates
                double odoX = klyngeY[finalKlynge[i]][j];
                double odoY = -klyngeX[finalKlynge[i]][j];

              pos.clear();
              pos.add(odoX,odoY,0);
              (*zLegsAsPoints)[i].push_back(pos); //push to the i'th leg
              if(DEBUG)(*zLegsAsPoints)[i][j].print("point:");
            }

      }

      // end added


      //DEBUG
      if(DEBUG){

          for (i = 0; i < finalCount; i++)
          {
            for (j = 0; j < klyngeLength[finalKlynge[i]]; j++)
            {
                //transfer to odometry coordinates
                double odoX = klyngeY[finalKlynge[i]][j];
                double odoY = -klyngeX[finalKlynge[i]][j];

                // Printes all final elements
              fprintf (FileThirdSort, "%g    %g      %i \n", odoX, odoY, i);
            }

          }
          // Closing files
          fclose (FileThirdSort);
          fclose (FileUnsorted);
      }//DEBUG END
    }
    else{
        printf("ERROR: SCAN DATA IN ZIMINO-FUNC NOT VALID!");
    }
  return true;
}
```