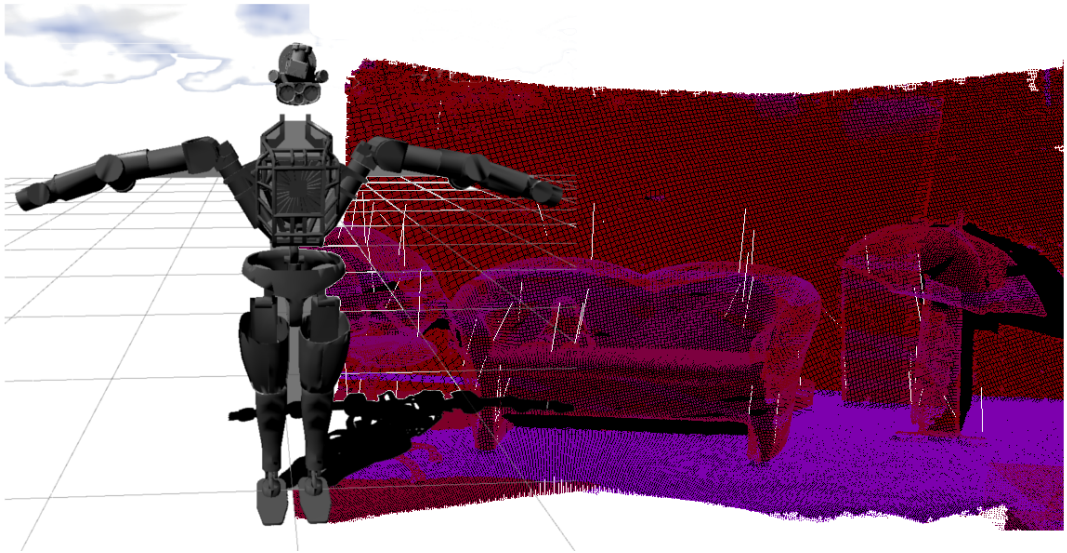
Project AI - The Darpa Robotics Challange
F.O.O.T.L.O.O.S.E.

Authors:

Maarten Inja *(5872464)*,
Norbert Heijne *(10357769)*,
Sander Nugteren *(6042023)*,
Maarten Waard, de *(5894883)*

February 5, 2013



Abstract

This report gives a description of an implementation of a walking engine for a humanoid robot to traverse rough terrain. This approach uses a point cloud sensor information of the surroundings to select places where the foot can be placed, and inverse kinematics to balance the robot while moving its foot.

1 Introduction

On October 24th, the DARPA Robotics Challenge (DRC) kicked off [1]. To quote their website:

> The primary technical goal of the DRC is to develop ground robots capable of executing complex tasks in dangerous, degraded, human-engineered environments. Competitors in the DRC are expected to focus on robots that can use standard tools and equipment commonly available in human environments, ranging from hand tools to vehicles, with an emphasis on adaptability to tools with diverse specifications.

The robot that is used in this challenge is the Atlas, a bipedal human sized robot, shown in figure 3. The Atlas has 28 degrees of freedom (DoF), the skeleton is shown in figure 1. This is a modified version of the PETMAN/Atlas which has different joint settings. It comes equipped with a laser scanner and two cameras located in the head. Compared to the Nao which has either 14, 21 or 25 DoF. The specialized Nao used in the Robocup has 21 DoF. The Atlas is human sized and has roughly the weight of a human, and also the same kind of proportions. Relatively, the Nao also has a much larger contact surface with the ground. This means that the Atlas is much more unstable than the small Nao.

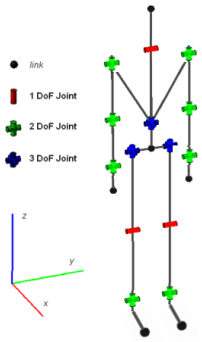


Figure 1: Skeleton of the Atlas showing the degrees of freedom.

Three tasks are part of the virtual DARPA challenge:

- Climb into a utility vehicle, drive along a roadway at no greater than 16 kph (10 mph), and climb out of the utility vehicle.
- Walk across progressively more difficult terrain, for example, progressing from parking lot to short grass to tall grass to tall grass on slope to ditch to rock field. In the earlier terrain, the GFE Platform balancing and walking behaviors will suffice. In the later terrain, DARPA expects perception and footstep planning will be needed.

[1] http://www.darpa.mil/Our_Work/TTO/Programs/DARPA_Robotics_Challenge.aspx

- Connect hose to spigot. This is purely a manipulation task, that is, the robot starts with everything within reach and so does not need to travel to the work site.

For this project we will try to tackle the second task in de virtual DARPA challenge. This virtual challenge entails using the Gazebo[2] simulator together with the Robot Operating System[3] (ROS) to make a model of the robot perform the task.

This task was split into two parts. The first part consists of making a model of the surface in front of the robot and selecting candidate surfaces for foot placement. The second part consists of computing a path to reach this location with the foot in a safe way. The candidate surfaces will serve as input for the walking engine, where it first performs a stable leg lifting motion, and then put the foot down in the desired location while keeping the robot balanced.

This report is organized as follows: in section 2 the related work that has been done on both subjects will be discussed; in sections 3 and 4 footstep planning and balancing will be described, followed by results, conclusion and future work.

[2]http://gazebosim.org/
[3]http://www.ros.org/wiki/

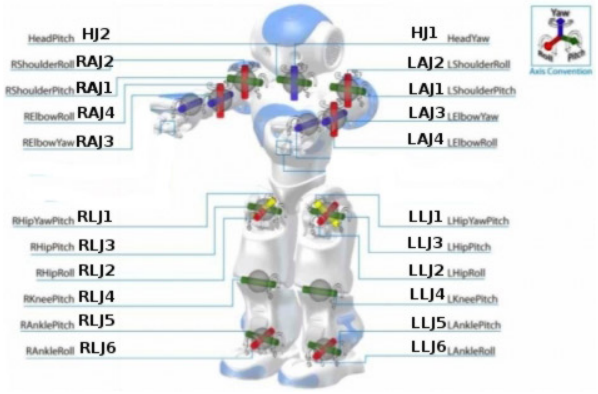


Figure 2: Skeleton of the Nao showing the degrees of freedom.

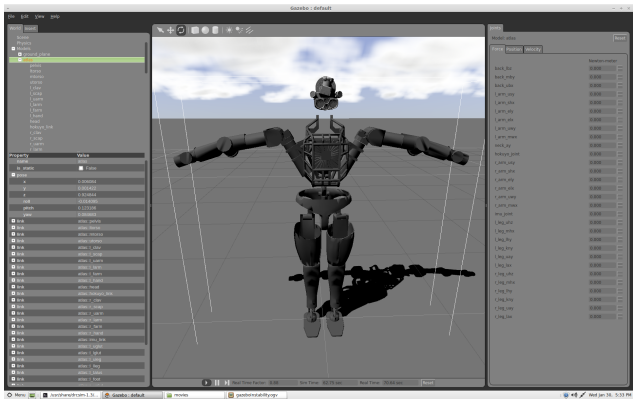


Figure 3: Screenshot of the GUI of Gazebo showing the rendering of the Atlas URDF model.

2 Related Work

2.1 Footstep Planning

There are several approaches to the planning of footsteps.

To prefer a straight surface over a surface with slopes one can use A* and include the angle of the ground as a heuristic [3]. This works well but the terrain is only categorized in four models, none of which deals with additional obstacles. Furthermore, the path planning is divided into two parts, first the path is planned from the current position to the robot, then the actual footsteps are planned to follow the first path.

By modelling the robots valid (foot) configurations, estimating the bounding box for the shape of the robot, and modelling the environment (obstacles) one can represent the robot and the world in a mathematical context. This representation can be used to determine illegal and legal positions and configurations, or states, for the robot. The states can be seen as nodes in a graph, in which the edges are the transitions from one configuration, or pose, to another. This results in a searchable graph that can be searched with a modified A* algorithm without having to be built entirely (the states do not all have to be calculated) [4].

A single rock can be seen as an obstacle over which the robot could step entirely, but if the terrain consists of many rocks it becomes 'rough' terrain. The difference with the previous methods is that there is no flat surface at all, the robot should attempt to find the best spot for its foot that would minimize falling or slipping risk.

Dealing with rough terrain can successfully be learned through reinforcement learning [5]. The observed terrain is modelled and matched to models, called templates, which are enriched with the positions that experts deem the best position for foot placement.

Ideally, low ceilings should also be dealt with by the robot by either avoiding such an area or by crouching. However, this is considered not part of our objective.

2.2 Balancing

There are multiple ways of balancing a bipedal robot. For example, some work was done on balancing a Nao in the context of Robocup Soccer [1]. In the paper the Nao is balanced by keeping the center of mass within the support polygon. This is done using force torque sensors on the feet of the Nao to find the current center of mass and adjust accordingly. In the case of the DARPA Robotics Challenge the Atlas is not as easy to balance as the Nao. The foot width to robot length ratio is much larger than the Atlas. Furthermore, the force torque sensors that are installed on the Nao cannot be installed on the Atlas in the current version of the simulator.

Another way to balance is described by [6], in this paper the motions are planned using inverse kinematics equations. In their research the robot performs balancing while avoiding a certain space and picking up a ball from the floor. The motions are planned offline, but they do mention that their method is usable in an online scenario for planning small amounts of steps at a time.

Therefore, using inverse kinematics to balance the robot's center of mass in the support polygon while performing the planned movement seems like a good way to handle balancing.

3 Footstep Planning

3.1 Theory

Section 2.1 describes all the necessities we have to implement for complete footstep planning. We focussed on the first step by investigating point cloud data and see if we could extract surfaces and

make the distinction between those that will support the robots foot and those who do not.

A point cloud is a collection of points, in which a point is defined as an x, y, z value, which might be collected by a sensor, such as a kinect or laser range scanner.

The Point Cloud library [4] which already is integrated into DRCSim, and thus into ROS offers two methods to do plane segmentation: region growing and plane segmentation using RANSAC.

3.1.1 Region Growing

Region growing segments points based on their curvature and surface normals, which are both local features based on the nearest neighbors of the points. [10]

A region is subset of a point cloud which are classified as belonging to the region, in our case we mean with a 'region' a plane, or surface. So any points that make up a plane should be considered a region. For different purposes one could for example want to find the regions that make up spheres. Segmentation is the process of dividing, or segmenting, the point cloud data in to different subsets, or regions.

The nearest neighbors could picked using several methods, using KD-trees or octrees, but also simply by taking the points in a radius.

It is easiest to consider the point and its nearest neighbors as a surface of which the normal is the vector perpendicular to the surface, and the curvature a scalar value indicating the curvature of the surface.

Region growing starts at the point with the lowest curvature value, this point is the start of the region, it is added to a new set called seeds. The algorithm is as followed:

- For each point in the seeds set, for each neighboring point:
  - Add neighboring point to the region if the angle between this point and the seed point is below the angle threshold θ_{th}
  - Add the neighboring point to seed set if its curvature value is below the curvature threshold c_{th}
  - Remove the seed point from the seed set.
- If the seeds set is empty, then a region has been found.

3.1.2 Plane Segmentation Using RANSAC

The second method to plane segmentation in the pointcloud library is to match the model of a plane in the pointcloud using RANSAC (RANdom SAmpling Consensus) [7].

The RANSAC algorithm informally goes as followed:

- Randomly select a subset of the point cloud and estimate the free model parameters
- Other data is considered, if a point fits the model a point is added (considered an inlier)
- The model is re-estimated considering all the inliers
- The model is evaluated by estimating the error relative to the model

A model is sufficient if a sufficient amounts of points are considered inliers.

[4] http://www.pointclouds.org/

3.1.3 Plane Evaluation

The planes, or surfaces, that are found using the region growing or plane modelling should be evaluated to a scalar that indicates how much the robot would want to place a foot on that plane.

First the average surface normal vector for a plane is calculated, normalized. Then the euclidean distance to the example unit vector $[0, 0, 1]$ (which is a vector pointing straight up) is calculated. This scalar should be sufficient for a path planner similar to [3] to prefer flat and straight surfaces over slopes.

Additionally we would like the size of the plane to be considered; planes smaller than the robots feet should be discarded.

3.2 Implementation

Some choices were made, to enable easy implementation of footstep planning. At first, the point cloud sensor was chosen as the most fitted sensor to find the environmental data that was needed for accurate footstep planning. This was because point cloud sensors are capable of collecting a lot of information about the environment in a small time frame. Using the point cloud sensor also enabled us to use the C++ Point Cloud Library (PCL), which enables a user to easily use many state-of-the-art point cloud processing algorithms [8]. Because of dependencies between the DRC simulator, ROS and PCL, in this project, the 1.5 version of PCL was used.

The current implementation finds planes in the environment of the robot and gives those planes a measure as to how badly the robot should want to step on them. This works in the following three steps. These steps require a robot to have a working point cloud sensor. Alternatively, a rotary laser range scanner can be used to simulate one.

1. Find planes in the point cloud
2. For each plane, find its mean surface normal
3. Evaluate each plane and its surface normal

The following subsections will explain the methods in detail.

3.2.1 Finding planes in a point cloud

As mentioned in section 3.1, planes can be located in various manners. In version 1.5 of the point cloud library, Region Growing has not been implemented yet. That is why our software solution uses plane segmentation based on Ransac. This is implemented using the PCL provided `SACSegmentation`. The exact parameters for the segmentation differ per goal and sensor.

The planes that are found using this segmentation algorithm are extracted from the cloud using `ExtractIndices`, also provided by PCL. Then the program loops through these planes, and enters the next step.

3.2.2 Finding mean surface normals

Normals of the points in a point cloud can be found using the pcl class `NormalEstimationOMP`, which uses a K-nearest neighbour search to find other close points, and estimates a normal vector using those points and, if specified, a camera position for the correct direction.

The mean surface normal is then calculated by simply adding them all and dividing by the number of points.

3.2.3 Evaluate each plane and its surface normal

The surface normal of the ideal standing surface is pointing up. That, by definition, means that a surface is placed horizontally, which is good to stand on. For that reason, the surface normal of each found surface is compared with one pointing up (vector $\begin{pmatrix} x=0 & y=0 & z=1 \end{pmatrix}^T$). Comparison is calculated with the following equation:

$$2 - \frac{\sqrt{(x_{in} - x_{compare})^2 + (y_{in} - y_{compare})^2 + (z_{in} - z_{compare})^2}}{2}$$

which is always used on unit vectors, assuring the result is a scalar between 0 and 1, 0 meaning completely different, and 1 meaning completely alike.

3.2.4 Viewer

The last thing we implemented is a 'viewer', which listens to the ROS topics published by the feature calculation program, and visualizes everything. This viewer represents the found plane segments in colors between red and purple. This color is created by a rgb-value based on the outcome of the comparison. A value of 125 for the red color is always given, so that every point can be seen. The value for blue differs from 0 to 255. For this the measure of equality is used as follows: $equality^5 * 255 = color$. We take the fifth power of the equality, in order to exaggerate the measure in which the ground should be horizontal: a 45° slope is fairly hard to stand on, especially for a robot.

This viewer enables the user to easily see what works and what works not and is a useful combination between listening to ROS topics and the PCL Visualizer class.

4 Balancing

4.1 Theory

The knowledge of a few concepts are required to understand the following sections on balancing. The center of mass (COM), COM-projection, support polygon, zero moment point (ZMP) and inverted pendulum. In our case the COM is defined as the weighted average location of the segments' mass, which together compose the Atlas. The COM-projection or vertical projection is a vertical line at the location of the robot's COM on the x and y grid. For the robot to achieve stability it must be located over the base of support. The support polygon is an area located on the floor that is a representation of the base of support and is defined as the convex hull of the set of points of the robot in contact with the walking surface [1].

Within the scope of our paper the ZMP is a concept related with the motion of legged robots. The ZMP is defined as the point where the inertia and gravitational forces equal zero, in other words the point in which the supporting foot has no momentum in any horizontal direction. This concept is critical when planning motion of legged robots, since bipedal robots only have two contact surfaces with the ground, the motion has to be planned concerning the dynamical stability of the robot's whole body [11].

Balancing using the ZMP is often compared to the problem of balancing an inverted pendulum. An inverted pendulum is a pendulum where the center of mass lies above its pivot point. Figure 4 shows the principle of the following quote.

> A metaphor of inverted pendulum is often used to explain such a system, since it is unstable in nature and controls its motion indirectly using the external forces generated

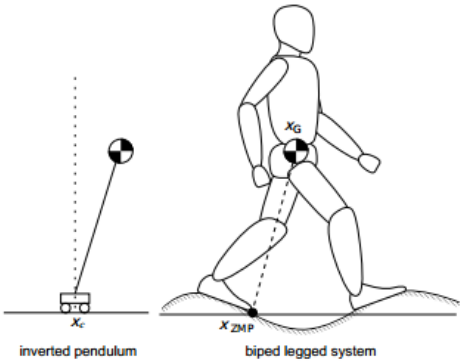


Figure 4: Inverted pendulum and legged system [9].

as reaction forces. Legged robots generally interact with the environment at multiple contact points. They can be equivalently represented by a single point on a virtual floor and its associated force/moments. The point is the ZMP. [9]

4.2 Implementation Dependencies

The requirements for the inverse kinematics part of this project are presented in this section. The ROS packages that our implementation depends on is hrl_kinematics[5], the kdl_parser[6], KDL[7] and the packages these three depend on.

hrl_kinematics is the package required for the center of mass calculations, these calculations are dependent on the data structures that are provided by the KDL package. In particular it uses the KDL tree data structure, the kdl_parser can convert the internal XML representation of the segments and connecting joint of the Atlas (URDF model) to a KDL tree. The tree contains representations of each segment and joint, and can be used to recursively calculate the Cartesian positions and rotation of a particular segment starting from the base segment.

The dependencies that were not part of the standard installation of DRCSIM for the packages hrl_kinematics and kdl_parser could be installed by using apt-get install on Linux Mint Maya or Ubuntu Precise with the stack to which the package belongs to.

To use the hrl_kinematics package to compute the center of mass, the KDL tree must be extracted with kdl_parser. KDL requires a URDF model as input. the standard URDF model of the Atlas was not sufficient since the KDL parser assumes there are no inertia parameters in the base link of the URDF model. Therefore a modified version must be made with a dummy link and a fixed joint attached to the original base link.

Secondly the hrl_kinematics source must be adjusted to be used with the Atlas, the adjustments made are explained in the implementation section, details can be found in the pseudo code section.

[5] http://www.ros.org/wiki/hrl_kinematics
[6] http://www.ros.org/wiki/kdl_parser
[7] http://www.ros.org/wiki/kdl

4.3 Implementation

The center of mass calculations are done via a modified version of the hrl_kinematics package which is part of the humanoid_navigation stack. The package has a method `computeCOM`, that is also described in pseudo code block 1, that takes the joint state representation as an external source, and then calculates the center of mass with the pelvis of the robot as reference point. This has caused some problems, since the hip joints are not fixed and there is no guarantee that it is possible to keep the pelvis levelled. Our modified method `computeCOMfoot` takes a foot segment as base using `getFrame`, shown in pseudo code block 3, and applies the inverse origin and rotation to the pelvis, and further calculates the center of mass the same way as `computeCOM`. The modified method is shown in pseudo code block 2.

The order in which our controller is balancing is by first displacing the center of mass with the largest part of its body which does not conflict with the walking animation. We were left with the upper body and the arms. Our controller applies our balancing code to the x and y joints of the lower torso, and afterwards tries to make the last minor adjustments to the center of mass using the shoulder's x and y joints.

The main controller tries to accomplish the basic stable stepping animation by calculating the corrections that need to be made for the lower body movement, and apply both the movements and corrections simultaneously. Using geometric calculations the controller tries to put the robot in a stable position in which velocity has a small effect. For example, the hip and knee movement are used to compute the ankle movement to allow the robot to stand straight. No stable stepping animation has been achieved for this controller as of yet.

In the initial stages of the run we planned to use a pre-programmed walk. The reason for this is that the footstep planner and the point cloud it utilizes does not see anything on the ground directly in front of it. This means that we require a hard-coded way of determining our steps so that it can pass the point from which it can get all its planned steps from the history based foot step planner.

5 Results

5.1 Footstep Planning

In this section we will show the results of our footstep oriented environment segmentation on various types of point cloud.

5.1.1 Gazebo's Points2 Sensor

The most logical sensor to try, in this case, is Gazebo's built in Points2 sensor. This sensor is mounted on the MultiSense-SL head. This sensor was used in a world with some objects, some of which one should want to stand on, others of which one should not. This world can be seen in figure 5.

The resulting image can be seen in figure 6. As can be seen the side of the golf cart is considered bad to stand on, just like the slope. The table and floor are good. Unfortunately planes parallel to the slope are also found in the stairs. For that reason the stairs are also calculated to a red color value. This problem is mainly caused by two reasons:

1. The PCL plane segmentation function has no way of setting a threshold of cloud density. In other words: cloud segments that exist of only five parallel lines of points can be fitted by a plane model, no matter how far these lines are apart from each other. This way, in stead

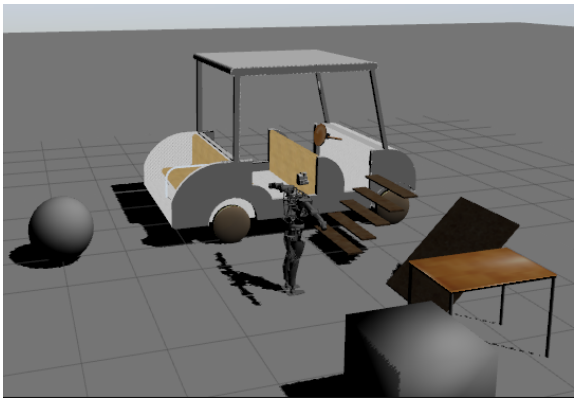


Figure 5: The Gazebo environment that was used to test the algorithm with the points and laser scanners.

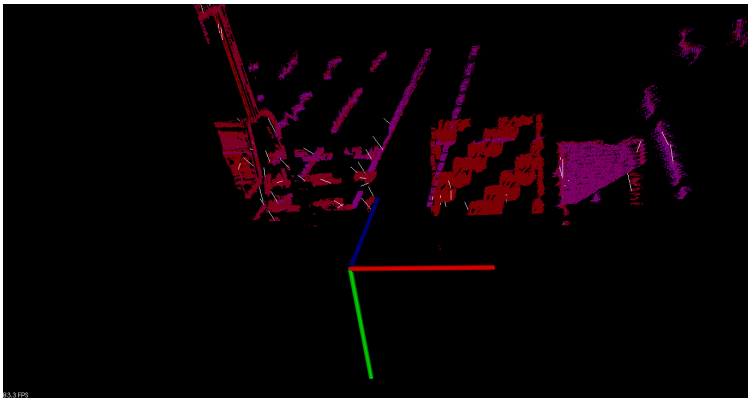


Figure 6: The output image of our algorithm when used with the Points2 sensor in the world of figure 5. A purple tint indicates that that part of the point cloud consists of a surface that is good to stand on. A red tint indicates the opposite. The white arrows are the calculated mean surface normals. From left to right part of the golf cart, stairs, slope and table can be seen. Table is seen as a good solution.

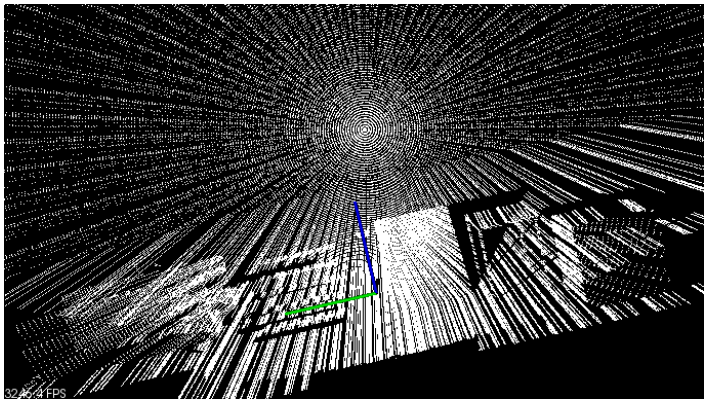


Figure 7: An image of the point cloud retrieved when collecting points from the rotary laser range scanner for 10 seconds, using the Gazebo world of figure 5.

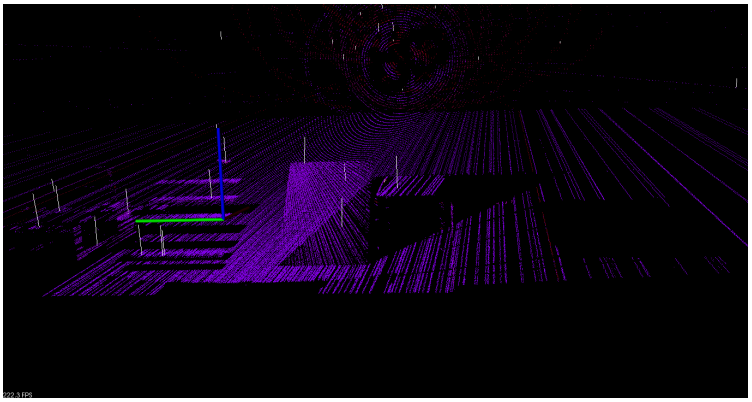


Figure 8: The output of the algorithm when using a rotary laser range scanner in the Gazebo world from figure 5. The same color codes as before count.

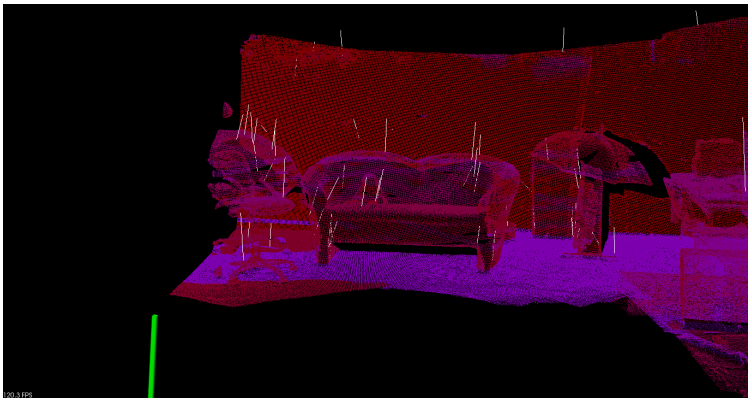


Figure 9: The output of the algorithm when using a point cloud that was retrieved from a real environment.

of only finding horizontal planes in the stairs, it is also possible to find planes that are fitted through several stairs, and are diagonal on the ground plane. However unwanted this result is, no solution to this problem could be found.

2. Another problem is that the point cloud finds very distorted points. As can be seen, only points in a grid form are found on the ground and on the slope. This is most probably a result of the manner in which the point cloud sensor was implemented in Gazebo.

   As a result of this noise in the point cloud, the processing of especially the stairs could have been worsened. Also finding good segments using any other algorithm than plane segmentation would be harder in the slope, because of the vertical stripes with point clouds, in stead a desired dense cloud, like can be seen on the table top.

5.1.2 Gazebo's Laser Range Scanner

Because of the gaps in the point cloud, the laser range scanner was put to use. This scanner returns points over one axis, and is capable of being rotated. This results in the capability of collecting points data of the surroundings. When collecting data for 10 seconds, the point cloud of figure 7 is retrieved. The first thing that can be seen is that this point cloud is more accurate than the point cloud retrieved by the Points2 sensor; all planes are straight and the points are divided more evenly. Another thing that can be noticed when looking at this data, is that the laser range scanner always returns the maximum value, when no points are found. This results in a dome of invalid points around the scanned environment. This will later be referred to as the 'noise dome'.

Using the laser scanner, and an 'assembler' to send the points of the last 10 seconds in one point cloud, the algorithm provides the image as seen in figure 8. As can be seen, the table top, each of the stairs and part of the golf cart are correctly detected and calculated to be viable stepping planes. A problem that occurs, however, is that the slope is also detected as a plane with a mean normal that is pointing straight up.

This has the following explanation: The plane that is found, consists of not only the slope, but also numerous points in the 'noise dome' that surrounds the environment. This results in a surface normal that is calculated based on not only the slope normals, but also on the 'dome normal', which is calculated by only one small ring of points and thus faulty.

5.1.3 A Point Cloud from the Real World

To prove that the algorithm works, and that is should also work in real-life, in stead of a simulator, we also downloaded a point cloud from the internet, that was created from a real life sensor. The point cloud that was used was the one from a Point Cloud Library tutorial [8]. This resulted in the image that can be seen in figure 9. In this image it can be seen that planes for the ground and walls are found and that the right value is computed for them. The couch forms a more difficult challenge for the algorithm, because the seat and backrest are more curved. Still most of the planes are evaluated correctly.

In the backrest of the couch, some purple planes are found. These are considered 'good to stand on', because they are returned together with the desk, that can be seen on the right side of the image. This is due to the same problem as was treated in the first point in section 5.1.1: The points in the couch are considered as being part of the desktop, whereas they really should have been part of a plane in the backrest of the couch.

5.2 Balancing

The results of the balancing of the revision "human like stumbling" is shown in figure 10.

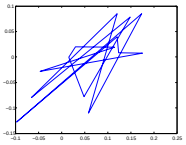


Figure 10: COM shift when attempting a walking movement of multiple steps left and right, coordinates are in pelvis coordinates.

What can be seen is that the shift is not very steady and that the COM is shifting towards the legs, but this resulted in the robot falling over and over again. Therefore we will discuss what must be improved before proper results can be shown in the future work section for balancing.

6 Conclusion

6.1 Footstep Planning

From the results shown in section 5.1 it can be concluded that using planar segmentation and normal estimation to find planes in the environment that can be stepped on by the robot should work. This should work in the Gazebo Simulator, if a noiseless point cloud is given and the points that do not have any value are set to *NaN* or omitted.

[8] http://www.pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php

Furthermore it is safe to conclude that some improvements can be done in the segmentation method, but that this robust version works in theory.

6.2 Balancing

Certain problems were made clear in the process of creating the controller. Building a hard-coded step animation is not a good solution due to the nature of the robot and the simulator. The simulator used by DRCSim is Gazebo, in its current version the robot seems to be unstable from the moment of initialization, the physics engine seems to already implement a form of noise generation, since multiple runs of the same code results in a slightly different result each time. The robot itself is quite tall and heavy, the base of the feet are also quite small compared to other robots that have been used with the Gazebo simulator, which makes it more difficult to balance. Therefore without a velocity model and a zero moment point (ZMP) equation, creating a basic stepping animation that is stable enough to handle the displacement of the robot is not feasible.

7 Future Work

7.1 Footstep Planning

Improvements can still be made on the footstep planning. These improvements can be put in two categories.

7.1.1 Improvements on current footstep planning

At first the current planning algorithm will be treated. The first thing that could be improved is the planar segmentation. One would want to find planes that are uninterrupted. Also planes with a small curve in them should be allowed. Initially this could be solved by (re)implementing region growing, as explained in section 3.1.1. Ideally other factors for stability of foot placements could be learned: For example when dealing with a rocky area, when hill climbing or walking through a ditch, a stable foot placement might not always be the flattest area, but a V-shaped hole in the ground.

Another improvement in finding good foot placement locations is retrieving the size of the plane and comparing that with the size of the robot's foot. This would for example make the robot take bigger stairs, when available, to minimize the chance of slipping when traversing upwards.

The last improvement that could be made on the current software is ignoring the points that are in the 'noise dome' of the laser range scanner. This can be done by calculating the distance of a point from the robot, and discarding it if it is above a certain threshold. This should improve the precision of the algorithm, because the computed surface normals of the planes that are currently found will then be more accurate.

7.1.2 Improvements on the overall footstep planning

To actually be able to use the algorithm, footstep locations should be found in the planes that are currently found suitable for walking on. These locations should be based on where the robot's current position is, where it wants to go and the measure of traversal difficulty for the terrain between these positions. As covered in section 2.1, Path planning algorithms like A* are suitable for this goal.

7.2 Balancing

The use of an inverse kinematics solver in our case is to convert a Cartesian position to joint positions, this can be done for balancing and for foot placement. In its current form, the controller does not use an inverse kinematics solver to determine joint positions to balance the robot. And since a stable leg lifting animation is not yet found, the solver has yet to be tested in our controller. The general idea is to perform a stable leg lift animation and use the step planning output to calculate the relative Cartesian positions. These are then used as input in our solver, and should output the corresponding joint positions for the desired pose of the to be placed foot.

Balancing is currently done via the movement of the torso and arms. In the future joints could be selected by using an inverse kinematics solver given the desired Cartesian positions of the joints. Our current geometrical calculations for balancing were meant to be temporary, because the solver would make them obsolete. The way balance is calculated now is shown in pseudo code block 4.

Our controller was also supposed to have a model of the actual joint states. The implementation of the listener implemented in DRCSim fell short, but there is a definite need for a joint state listener. On top of this model we require a velocity model to keep the robot balanced. The bare bones setup of the robot only has a laser scanner and two cameras as sensors, we suggest adding a gyroscopic sensor, or other types of force torque sensors which could measure changes in the center of mass.

The use of inverse kinematics to handle balancing and motion is what we aimed for when we started this project. With ample time for the robot to plan ahead, this is not be a problem. However, there is always the possibility that the robot is not allowed to stand still for seconds, if not minutes, to calculate the optimal path and joint positions required. Short distance planning of movement might be an option [6], but optimally we would require a model that could be adjusted in realtime.

As it it now, the controller could benefit greatly from ZMP equations to generate a stable walking animation due to the noise present within Gazebo and the unstable nature of a human sized robot. Because of this, precomputed trajectories and ZMP equations would most likely not suffice. The controller would need to implement a dynamic trajectory model to deal with the unstable nature of gazebo and the robot. Our current approach of using inverse kinematics to determine the foot placement for the normal walking animation might be unnecessary if we use the online trajectory generation as described in [2]. Although the robot in the paper is of smaller stature and has a larger contact surface with the ground, the trajectory could be adjusted for use with the Atlas. The only thing that this method is missing is a way to adjust for specific placement of the feet. Therefore a mix between the online trajectory generation and inverse kinematics equations might give the desired result.

7.3 Integration

For integration of both of the parts of the program, the footstep planning should output Cartesian foot placement coördinates. Since all the software is contained in ROS packages, this can easily done by using ROS messages. These should then be read by the inverse kinematics part, and put into action: Placing the foot on the Cartesian coördinate. Apart from the problems already covered in this section, some other problems will then arise:

- The laser range scanner will occasionally scan the robots arms and legs, because they will now be moving.
- The footstep planning has not been tested with a moving robot. In theory the ROS framework should fix everything related to translating point clouds to their right relevant position, but in practise things might turn out differently.

References

[1] I. Becht, M. deJonge, and R. Pronk. A dynamic kick for the nao robot, 2012.

[2] S. Behnke. Online trajectory generation for omnidirectional biped walking. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 1597 –1603, may 2006.

[3] J.M. Bourgeot, N. Cislo, and B. Espiau. Path-planning and tracking in a 3d complex environment for an anthropomorphic biped robot. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2509–2514. IEEE, 2002.

[4] R. Cupec, I. Aleksi, and G. Schmidt. Step sequence planning for a biped robot by means of a cylindrical shape model and a high-resolution 2.5 d map. *Robotics and Autonomous Systems*, 59(2):84–100, 2011.

[5] M. Kalakrishnan, J. Buchli, P. Pastor, and S. Schaal. Learning locomotion over rough terrain using terrain templates. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 167–172. IEEE, 2009.

[6] O. Kanoun, J.P. Laumond, and E. Yoshida. Planning foot placements for a humanoid robot: A problem of inverse kinematics. *The International Journal of Robotics Research*, 30(4):476–485, 2011.

[7] Radu B. Rusu. Plane model segmentation. http://www.pointclouds.org/documentation/tutorials/planar_segmentation.php. Online; retrieved 03-Feb-2013.

[8] R.B. Rusu and S. Cousins. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011.

[9] T. Sugihara, Y. Nakamura, and H. Inoue. Real-time humanoid motion generation through zmp manipulation based on inverted pendulum control. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 2, pages 1404–1409. IEEE, 2002.

[10] Sergey Ushakov. Region growing segmentation. http://www.pointclouds.org/documentation/tutorials/region_growing_segmentation.php. Online; retrieved 03-Feb-2013.

[11] M. Vukobratović and B. Borovac. Zero-moment pointâĂŤthirty five years of its life. *International Journal of Humanoid Robotics*, 1(01):157–173, 2004.

8 Appendix

Algorithm 1 computeCOM

Require: Kinematics tree of the robot `tree`, current joint positions J, pointer to a tree element T
Ensure: Frames contains the rotation and origin of a specified segment

```
T ← root element of tree
COM ← 3d vector initialized to zero
F ← Frame initialized to Identity
MASS ← 0
recursion(T, J, F)
COM ← COM / MASS
return COM
procedure recursion(T, J, F_0)
    s ← segment of tree element T
    j ← joint located at the origin of segment s
    M ← mass of segment s
    F_1 ← current pose of segment s, given the joint position in J of joint j
    COG_s ← 3d vector of center of gravity of segment s
    COM ← COM + M · (F_1 · COG_s)
    MASS ← MASS + M
    for all Child nodes of T do
        t ← next child node of T
        recursion(t, J, F_1)
    end for
end procedure
```

Algorithm 2 computeCOMfoot

Require: Kinematics tree of the robot `tree`, current joint positions J, pointer to a tree element T, name of the foot segment $foot$

Ensure: Frames contains the rotation and origin of a specified segment

$T \leftarrow$ root element of `tree`
$COM \leftarrow$ 3d vector initialized to zero
$F \leftarrow$ getFrame($foot$)
$F \leftarrow$ Inverse Frame of F
$MASS \leftarrow 0$
recursion(T, J, F)
$COM \leftarrow \frac{COM}{MASS}$
return COM
procedure recursion(T, J, F_0)
    $s \leftarrow$ segment of tree element T
    $j \leftarrow$ joint located at the origin of segment s
    $M \leftarrow$ mass of segment s
    $F_1 \leftarrow$ current pose of segment s, given the joint position in J of joint j
    $COG_s \leftarrow$ 3d vector of center of gravity of segment s
    $COM \leftarrow COM + M \cdot (F_1 \cdot COG_s)$
    $MASS \leftarrow MASS + M$
    for all Child nodes of T **do**
        $t \leftarrow$ next child node of T
        recursion(t, J, F_1)
    end for
end procedure

Algorithm 3 getFrame

Require: Kinematics tree of the robot `tree`, current joint positions J, pointer to a tree element T, name of the foot segment $foot$

```
procedure getFrame(foot, J)
    T ← root element of tree
    F_d ← Empty frame
    F ← Frame initialized to Identity
    getFrameRecursion(T, J, F, F_d)
    return F_d
end procedure
procedure getFrameRecursion(T, J, F_0, F_d, foot)
    s ← segment of tree element T
    j ← joint located at the origin of segment s
    F_1 ← current pose of segment s, given the joint position in J of joint j
    if Name of s equals name of foot then
        F_d ← F_1
        exit function
    end if
    for all Child nodes of T do
        t ← next child node of T
        getFrameRecursion(t, J, F_1, F_d)
    end for
end procedure
```

Algorithm 4 calculateBalance

Require: current joint positions J, supporting leg leg, the distance $length$ between the desired moving joint j and the center of mass of every body part it affects, the fraction of the mass of the affected body parts $weightedMass$, a given offset in the joint direction to compensate $offset$

Ensure: The updated J_j moves the affected body parts in the desired direction and is within the joint limit

$foot \leftarrow$ end affector of leg

$com \leftarrow$ computeCOMfoot($J, foot$)

$distance \leftarrow -($ relevant direction of com for j $)$

$J_j \leftarrow sin\left(\frac{distance+offset}{weightedMass \cdot length}\right) \cdot \pi + J_j)$