Dissertation

Footstep Planner and Environment Sensing System for a Cost-Oriented Humanoid Robot

carried out for the purpose of obtaining the degree of Doctor technicae (Dr. techn.), submitted at TU Wien, Faculty of Mechanical and Industrial Engineering, by

Francesco d'APOLITO

Mat.Nr.: 01329734

under the supervision of

Em.O.Univ.Prof. Dipl.-Ing. Dr.techn. Dr.h.c.mult. Peter Kopacek

Institute of Mechanics and Mechatronics, IHRT

reviewed by

Ao.Univ.Prof. Prof.h.c. Dipl.-Ing. Dr.techn. Dr.h.c. Durakbasa Numan M.

TU Wien, Institute of Production Engineering and Photonic Technologies

Karlsplatz 13, 1040 Wien

Prof. Curti Fabio

University of Rome "La Sapienza", School of Aerospace Engineering

Via Salaria 851, 00138 Rome

I confirm, that going to press of this thesis needs the confirmation of the examination committee.

Affidavit

I declare in lieu of oath, that I wrote this thesis and performed the associated research myself, using only literature cited in this volume. If text passages from sources are used literally, they are marked as such.

I confirm that this work is original and has not been submitted elsewhere for any examination, nor is it currently under consideration for a thesis elsewhere.

Vienna, January, 2019

_____

Kurzfassung

Es ist absehbar, dass in der nahen Zukunft humanoide Roboter Teil des menschlichen Alltags im Haushalt sowie am Arbeitsplatz sein werden. Es kann davon ausgegangen werden, dass sie auf dem Markt zu einem vernünftigen Preis erhältlich sein werden. Die Forschung über kostenorientierte humanoide Roboter begann im Jahr 2011. An der technischen Universität Wien (TU Wien) ist derzeit ein kostenorientierter humanoider Roboter, genannt Archie, in Entwicklung. Er setzt sich derzeit aus zwei Beinen, einer Hüfte und einem Rumpf zusammen und verfügt über die grundlegenden Gehfähigkeiten.

Diese Doktorarbeit gibt einen Überblick über die Steigerung der Fähigkeiten des Roboters auf Hardware und Softwareebene. Das Ziel dieser Arbeit ist es, dem Roboter die Wahrnehmungsfähigkeit der Umgebung zu ermöglichen. Eine weitere Zielsetzung ist, dem Roboter die Fähigkeit zur Durchführung einer autonomen Pfadplanung von einer bestimmten Startposition zu einem Zielpunkt, inklusive der Hindernisvermeidung auf dem Weg, zu geben.

Zur Zielerreichung wurde einer Stereo-Kamera auf dem Roboter installiert. Dies ermöglicht es, Daten aus der Umgebung zu sammeln. Des Weiteren wurde ein leistungsfähiger On-Board Computer eingebaut, um die Autonomie des Roboters zu erhöhen.

Im Hinblick auf die Software wurde eine „Detect And Avoid“ Software (Erkennung und Vermeidung) realisiert. Die Zielsetzung ist es, Archie die Möglichkeit der Umgebungserfassung, durch die Verarbeitung der Daten aus den Stereo-Kameras, zu geben. Weitere Ziele sind die Bereicherung der Map mit den erkannten Hindernissen, die Erkennung der aktuellen Position auf der Map und die Planung eines sicheren Pfads zu einer Zielposition.

Abstract

It is foreseeable that in the next future, humanoid robots will be part of everyday life in the household and in the workplace, and that they will be available on the market for a reasonable price. The research on cost-oriented humanoid robot started in 2011. At the Technische Universität Wien (TU Wien) a cost-oriented humanoid robot, named Archie, is in development. It is currently composed of two legs, a hip and a torso and it has basic walking capabilities.

This PhD dissertation describes the efforts made to increase the capabilities of the robot at hardware as well as at software level. The goal of this work is to give the robot the capability of sensing the environment around it and to autonomously plan the path from its current position to a target position, while also avoiding the obstacles on the way.

Such a goal required the addition to the robot of a stereo-camera in order to allow it to collect data from the environment around it. Furthermore, a powerful on-board computer was added in order to increase the autonomy level of the robot.

From the software point of view, this work consisted in the first implementation of a detect and avoid software. It aims to give Archie the capability of mapping the environment, by processing the data coming from the stereo-cameras, enriching the map with the obstacles detected, knowing its position in the map and planning a safe trajectory from its position to a target position.

Acknowledgment

I would like to thank Professor Peter Kopacek for the help and the support given during the PhD. All the things I learned during my PhD studies, I learned from him.

I would also like to thank my beloved wife, for the support and the love she provides me with every day. I would have probably never made it without her.

Furthermore, I would like to thank my parents for the help and support they gave me during my studies. They always inspire me to improve myself.

I would also like to thank my parents-in-laws for accepting me as part of their family right from the beginning and for supporting me during my studies.

List of content

List of Figures

List of tables

1. Introduction and problem formulation

The humanoid robot Archie is under development at the Technische Universität Wien (TU Wien). The development of the robot started in (Byagowi, 2010) under the supervision of Prof. Peter Kopacek. Further developments were then performed in (Daniali, 2013) and (Dezfouli, 2013). Currently, Archie is able to perform basic human like walking motions (Kopacek; 2013).

Archie has been developed in order to be a cost-oriented and research-oriented application. Thus, the hardware must fulfil the cost orientation target and should also allow the implementation and testing of advanced centralized control algorithm. Furthermore, being a research-oriented application, the software must keep as many possibilities as possible for further development and research.

This PhD thesis presents the work performed in order to increase Archie's capabilities. More precisely, the aim of this PhD work is to make Archie able to autonomously plan the necessary footsteps from a start position to a target position while also avoiding the obstacles on the way. In order to achieve such a goal, hardware modifications and the implementation of a new software were necessary. The hardware modifications aimed at providing the robot of an environment sensing system. The software instead, consists in the first implementation of an obstacle detector and of a footstep planner.

Nowadays, the environment sensing systems for a humanoid robot consists of multiple sensors which give an accurate data model of the environment. On the software side, the footstep planners are based on 3D obstacle avoidance algorithms which computes directly the footstep position and the trajectory of the centre of mass (COM) for the balance of the robot.

There are two main challenges in the development of such a system for Archie. The first is represented by the cost orientation which sets a limit to the number of sensors and consequently to the amount and density of data available to the software. The second challenge relies on the fact that Archie is a research-oriented application and it aims to be a test base for advanced gait and joint control algorithm. For this reason, the software should guarantee a safe and stable walking while also having a modular and flexible architecture in order to make easier further development

and research. Another requirement for the software consists in the computational load of the algorithms. It should be kept as low as possible in order to allow embedded data processing.

The thesis is organized as follows. The next chapter will describe the state of the art in the field of humanoid robots with a reference to the environment sensing systems of the most famous humanoid robots. Subsequently, a description of the state of the art of the footstep planners for humanoid robots will be given, with a focus on path planning and collision avoidance algorithms and their implementation.

Chapter 3 will depict a picture of the state of the development of the humanoid robot Archie, followed by an overview of the advanced gait and joint control algorithm currently in evaluation. These will be explained in order to give the requirements for the new electronic architecture and for the new software object of this PhD work.

Next, the new hardware design will be given. Finally, the new footstep planner implemented for Archie will be described in detail along with all the algorithms used.

2. State of the art

The research on humanoid robots involves various fields, from the artificial intelligence to the control theory to the dynamic and kinematics study of the human gait.

The bipedal form is for sure the best for a robot due to the possibility of travel though every kind of terrain while also keeping the ability to manipulate the environment. As a matter of facts, a humanoid robot, unlike wheeled robot for instance, can climb stairs or step over obstacles without problems.

A humanoid robot development though, is particularly challenging from many points of view, from the mechanical configuration to the power management to the sensor and on-board computer and all these problems are interlaced with each other. For example, a humanoid robot application requires a high computational power and a vast number of sensors i.e. tactile force sensor, stereo-cameras and depth camera. These choices have an impact on the amount of power used by the robot and the more power the robot need, the bigger and heavier the batteries will be. A big and heavy battery will have an impact on the balance of the full system.

This chapter will give a description of the state of the art in the field of humanoid robots. First, a description of some existing humanoid robots will be outlined with a special reference to the hardware dedicated to the perception of the environment around the robot. Then, the topics of the path planning and obstacle avoidance will be introduced along with a description of the only two open source footstep planners available.

2.1. History of humanoid robots

The Japanese research team at the University of Waseda was one of the first in the development of humanoid robots. In 1967 the first bipedal robot WL-1 appeared. Since this first prototype, they finished in 2009 the development of WABIAN-2R.

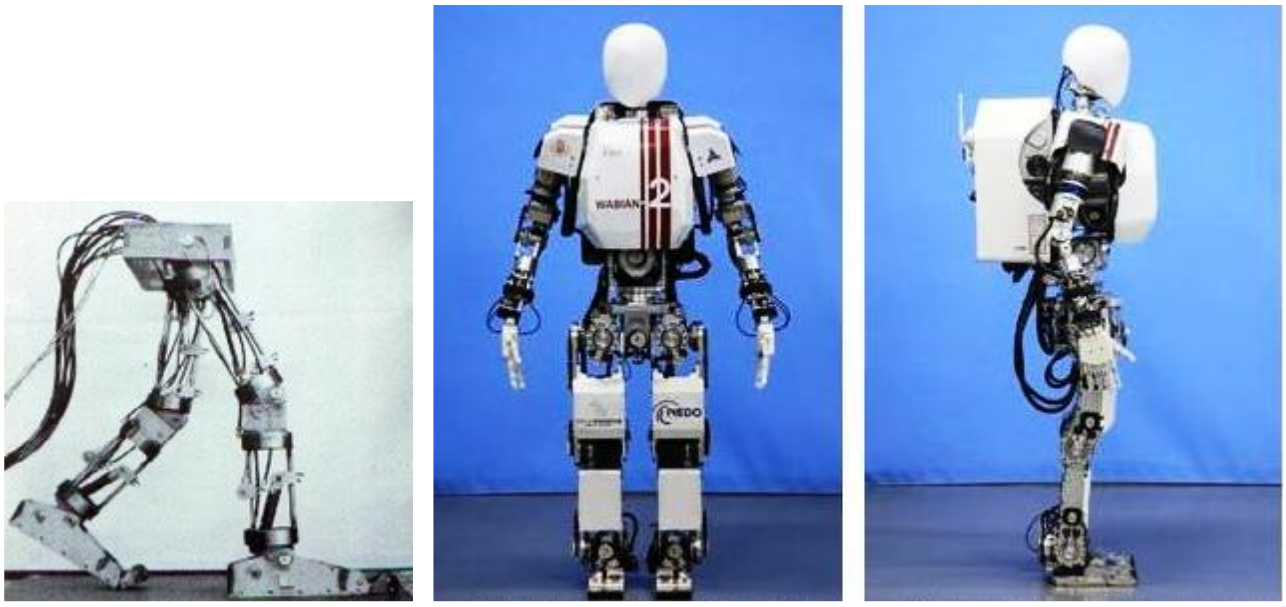

Figure 1: WL-1 and WABIAN 2R (Daniali,2013)

The Honda Motor ASIMO is also another remarkable example of humanoid robot. It was developed in the year 2000. It is 1,30 m tall and it has a total of 34 Degrees of Freedom (DOF). The neck has 3 DOF, the arms 7 DOF for every arm, 2 DOF for every hand, 1 DOF for the hip and 12 DOF for every leg. ASIMO's environment identifying sensor system is composed of visual sensors placed in the head, which allows him also to recognize a person and ground sensors placed in the hip and composed of laser sensor and infrared sensor. The laser sensor is able to detect the ground and any obstacle 2 m from its feet. Furthermore, the ultrasonic sensors are placed in front and in the back of the robot and are able to detect obstacle up to three meters from it. With the ultrasonic sensors, it is also possible to detect glass as an obstacle which is not possible with the visual sensor. This subsystem is depicted in Figure 2.

Since ASIMO, the research in the field of humanoid robotics continued at a remarkable speed. NASA developed a humanoid robot, called Robonaut, to help the astronauts in their daily job on board the International Space Station (ISS). The second version of this humanoid robot, called Robonaut 2 was launched on February 2011. At first just the torso of the robot was developed but then, in 2014 the "mobility platform" i.e. two climbing manipulators as legs were added. The tests and the developments on this robot are still in progress and soon more subsystems and sensors will be added to the current platform. Figure 3 depict Robonaut 2 on board the ISS.

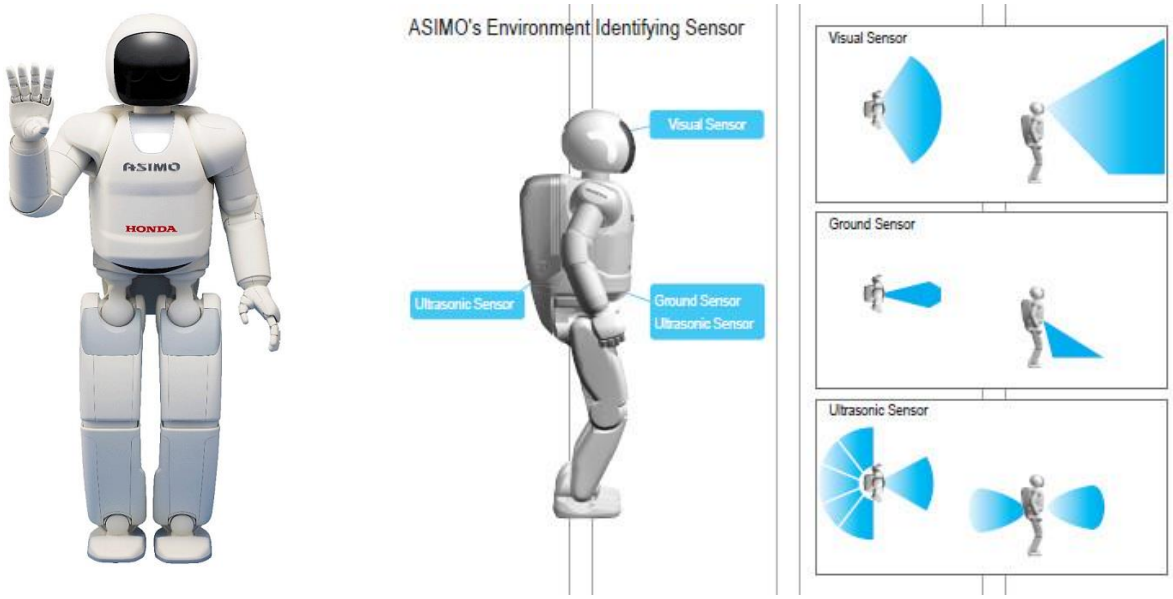

Figure 2: ASIMO (Honda Asimo gallery, last retrieved 2017) and its environment identifying sensors (ASIMO Technical Information, 2007)

Figure 3: Robonaut 2 on board the ISS (NASA Robonaut website, last retrieved 2017)

Two other noteworthy examples of humanoid robots are ATLAS, developed by Boston Dynamics and THORMANG developed by Robotis which are depicted in Figure 4.

ATLAS is a 28 DOF humanoid robot. It is 1.8 m tall and it weighs 150 kg. All its joints are hydraulically actuated and it is equipped with stereo-cameras and laser range finders. It is powered by an external power source by means of a flexible tether. ATLAS is capable of walk through rough outdoor environment and it is strong and coordinated enough to climb using hands and feet (Boston Dynamic website Atlas description, last retrieved 2017).

THORMANG 3 is a 29 DOF humanoid robot. Its joints are actuated by Dynamixel motors. It is 1,37 m tall and it weighs 42 kg. It is equipped with force and torque sensors and with Inertial Measurement Units (IMUs), LIDAR and cameras.

The same company which developed THORMANG 3, developed also small humanoid robots' kits, the Bioloid series, easy to mount and easy to program for educational purposes. The Bioloid robot kit is showed in Figure 5.

Figure 4: ATLAS on the left side (Boston Dynamic website Atlas description, last retrieved 2017). THORMANG 3 on the right side (I, Bioloid blog, last retrieved 2018)

Figure 5: Bioloid robot kit (Robotis website Bioloid description, last retrieved 2017)

Another example of small commercial humanoid robot platform is NAO, developed by Aldebaran Robotics. It is a 25 DOF humanoid robot platform which is 58 cm tall. It is equipped with an inertial unit which allows him to maintain its balance. Numerous sensors in its head and in its hands and feet allows it to perceive the environment around him and the cameras in its head make it able to record images and video in high resolution as well as to recognize shapes and objects. NAO is showed in Figure 6.

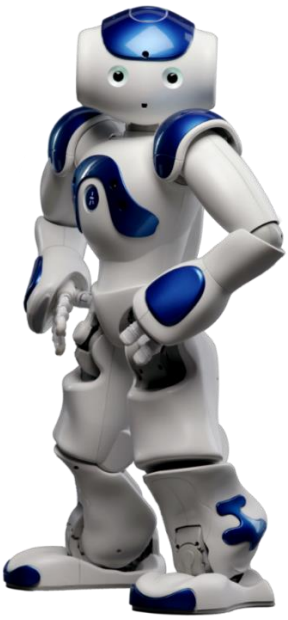

Figure 6: NAO Humanoid robot (NAO online description, last retrieved 2017)

2.2. Footstep planner for humanoid robots

The planning of a collision free trajectory from an initial position to a target position requires a hardware and software architecture that is also capable of analyzing the environment around the robot. In the previous paragraph is listed some example of environment sensing hardware for some of the most remarkable examples of humanoid robots.

The software needs then to be able to analyze the environment around the robot and create a map. There are two possible approaches for creating a map: discrete and continuous approximation. Maps produced with the continuous approximation are also known as topological maps. In this kind of maps, the environment is simplified in order to have every unnecessary information removed.

With the discrete approximation, the environment is divided into cells of different shapes and size. It could be possible to have, for instance, grid maps or hexagonal maps and the different cells can be as big as a room in an apartment or as a small volume. The use of this approach allows a graph representation. In this case, every chunk of the map corresponds to a vertex (also known as “node”), which are connected by edges, if a robot can navigate from one vertex to the other (Correll, 2011).

Currently, there is no preferred way to solve this problem. Every application has its own best way to create a map of the environment. The use of mixed approaches may also be required. One of the most common method used is the occupancy grid map. In this method, the environment is discretized into squares of various resolution on which obstacles are marked. Another version of this method is the probabilistic occupancy grid map in which the cells of the map are marked with the probability of the presence of an obstacle. A drawback of grid methods is the memory requirement and the computational load of the path planning algorithm which must plan a trajectory in such a map.

The computational complexity and the computational load, for our application, needs to be kept under consideration both in the map creation and in the path planning algorithm in order not to slow down the entire system. This last requirement makes the use of the artificial potential field algorithm, for instance, not usable for such an application.

The artificial potential field is an algorithm which associates the target position an attractive potential and a repulsive potential to the obstacle. By setting to zero the anti-gradient of the total

potential, the algorithm provides an optimal path between the start and the target position that also avoid the obstacles. Figure 7, depicts the attractive, repulsive and total potential field.

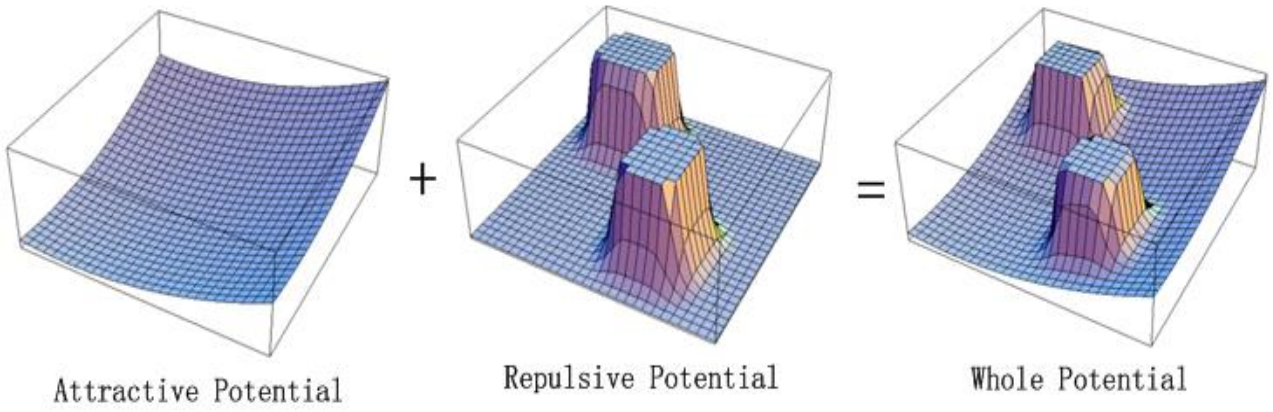

Figure 7: Attractive, repulsive and total potential field (My Point Cloud blog, last retrieved 2017)

Such an approach though has the drawback of the local minima. It is possible, as a matter of fact, that, due to symmetry in the total potential field, some local minima create in which the planning can get stuck. The strategy to avoid these local minima can be computationally expensive and can also have an impact on the optimality of the computed trajectory.

Another approach for the path planning problem is the Dijkstra algorithm. It is a graph-based algorithm, meaning that it is usually used with a discrete map representation of the environment. It starts from the initial vertex marking all the neighbour vertex with the cost to get there. The planning then considers the vertex with the minimum cost associated to it, analysing the neighbour vertex and marking them with the cost to get to them via itself if the cost is lower (Correll, 2011). The algorithm goes then further analysing the neighbour of the vertex with the lower cost. After the search reaches the target position, the robot can then follow the edges with the lower costs. It is obvious that such a planning is computationally expensive because the planning does a lot of computations that are not needed examining the cells of the grid that are not in the direction of the target position. Figure 8 depicts the operation of the Dijkstra algorithm.

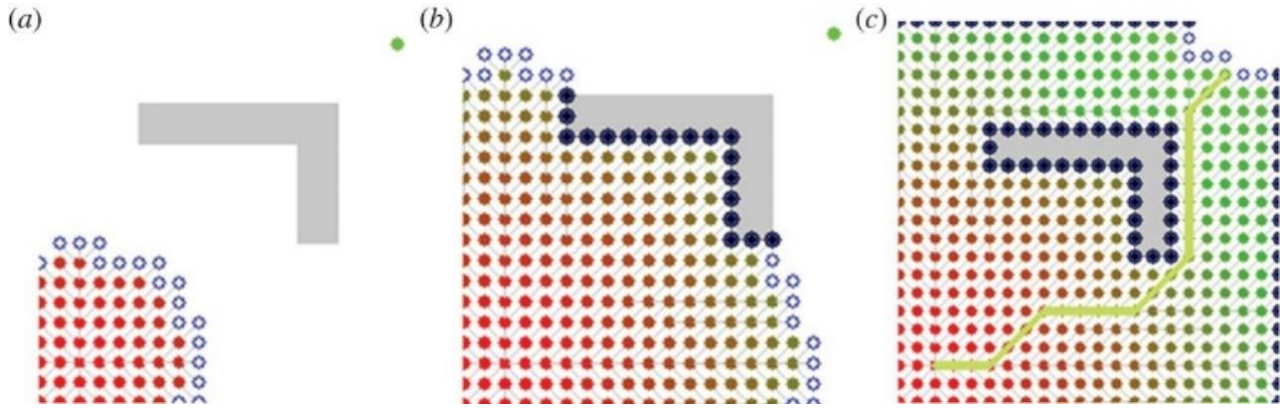

Figure 8: Operation of the Dijkstra algorithm (Milford M. and R. Schulz, 2014)

A solution to this drawback of the Dijkstra algorithm is to add a further condition besides the cost of the vertex described above. This condition is based on the estimated distance to the goal. A vertex will be considered just if the estimated distance is lower with respect to others. This method is known as A* algorithm. As it is possible to see in Figure 9, the A* algorithm is less computationally expensive than the Dijkstra algorithm because it just analyzes cells in the direction of the target but, its computational load can still dramatically increase when a re-planning is necessary due to the presence of a new obstacle. For this reason, another version of this algorithm, called D*, was implemented. Although the computational load of the A* and D* approach is less compared to the Dijkstra algorithm, they become computationally expensive when either the search space is large, e.g., due to a fine-grain resolution required for the task, or the dimensions of the search problem are high (Correll, 2011).

For a humanoid robot, path planning and obstacle avoidance are complex tasks because they don't merely consist in the computation of a collision free path between an initial and a target position but they also rely on the balance of the full system and they have to take into account the increased number of possibilities that the humanoid configuration can offer such as, for instance, the possibility to step over an obstacle.

The planning of a collision free trajectory between two positions for a humanoid robot requires also the computation of the position of the feet and of the COM of the robot for the balance. Currently, the most remarkable examples of path planning and collision avoidance algorithms for humanoid robots extend the 2D planner algorithms, such as the ones presented before in the chapter, making them a 3D planner which computes directly the position of the feet. An example of them is the planner developed in (Garimort et.al., 2011) and in (Hornung et.al., 2012).

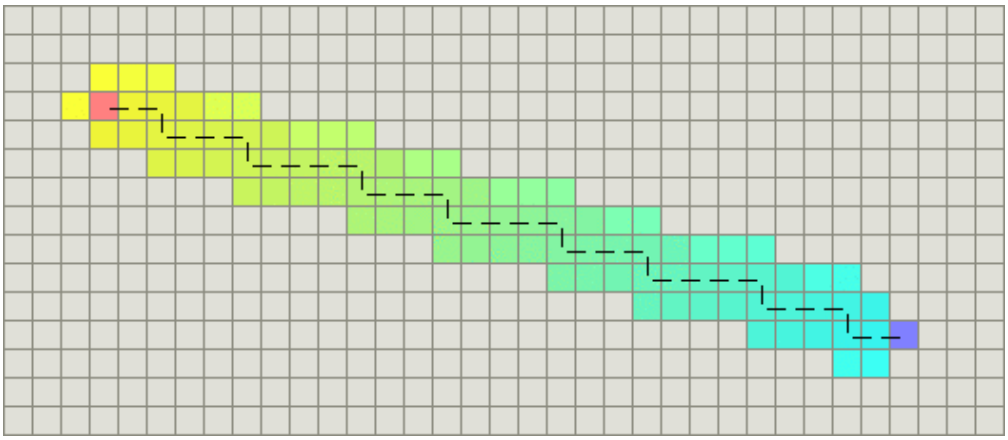

Figure 9: A algorithm (A* algorithm Stanford website description, last retrieved 2017)*

They developed an open source footstep planner for humanoid robots based, in the first version of the software, on the D* algorithm. The planner they implemented used the D* algorithm to compute directly the position of the feet of the robot. This approach showed good results both in simulation and in real world by means of the NAO humanoid robot platform as showed in Figure 10.

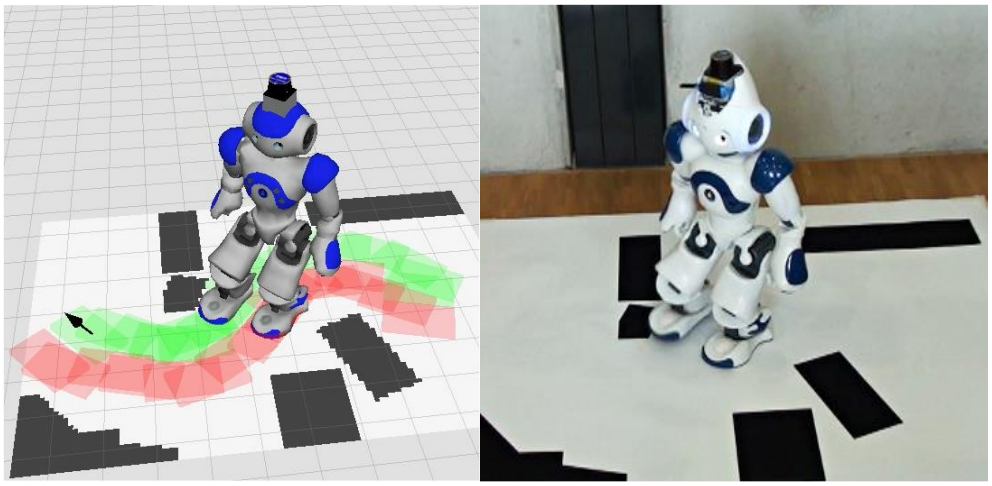

Figure 10: Simulation and experiments of the footstep planner developed in (Garimort et.al., 2011)

The navigation of a humanoid robot in a cluttered environment though, is still a challenging problem (Garimort et.al., 2011) for this planner. Their work was thus improved with the use of the Anytime Repairing A* (ARA*) and with the use of the randomized A* (R*) algorithm. Both algorithms are variations of the A* algorithm which sacrifice the optimality of the solution in order to have a quicker

execution time. The experiments performed showed the efficiency of the planner even in cluttered environment as showed in Figure 11.

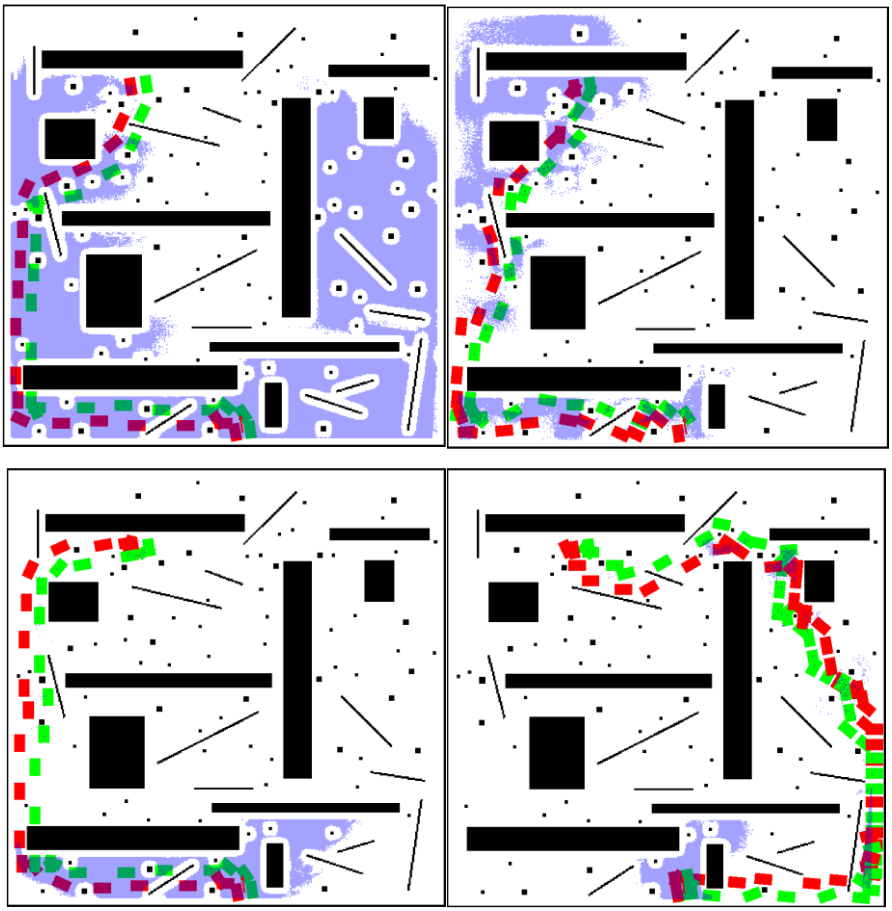

Figure 11: Performance of the footstep planner developed in (Hornung et.al., 2012)

This planner, as mentioned before, is available open source as a ROS (Robot Operating System) package. ROS is a framework described in (Quigley et al., 2009) which provides tools and libraries to help developers to implement robot applications and it has emerged as a de facto standard in robotics research in recent years (Kohlbrecher et.al., 2016). To our knowledge, besides this planner the only other open source solution currently available is the planner developed in (Stumpf et.al., 2016). In their opinion, the planner described above is very easy to use but provides limited adaptability. In order to consider robot specific constraints, the base code has to be changed directly (Stumpf et.al., 2016). They proposed thus, a more versatile solution which could be used with different kind of robotic platforms. Their footstep planner is designed in order to be a framework which integrates perception, world modelling, full 3D planning, step execution tracking, and

coactive planning, while being modular and extensible (Stumpf et.al., 2016). The open source ROS package they developed, as a matter of fact, is divided into plugins which allow an easy customization or the extension of the software. For the world perception, they used a lightweight approach based on the grid-based elevation map and an octree-based data structure. The first data structure is a grid-based map in which every cell stores the information about the elevation. The second is a tree structure in which every leaf has no more than 8 children which stores the information about the surface orientation. During the planning and the execution of the walk, the user has the possibility of following the robot and adjust the computed footstep if the planning was wrong. The visualization of the planning and of the terrain model is shown in Figure 12.

This footstep planner was tested on 4 different humanoid robots and the planner fully proved its portability.

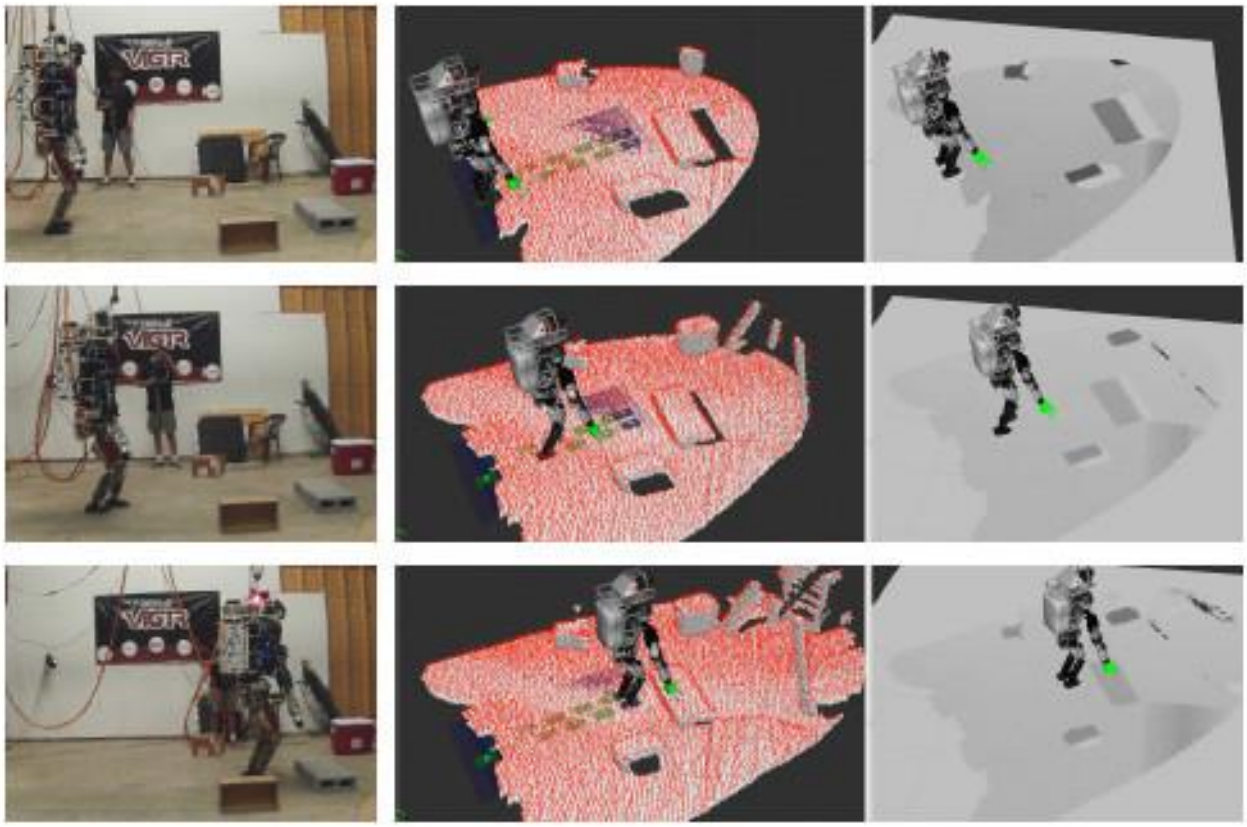

Figure 12: Visualization of world model, and step planned of the footstep planner developed in (Stumpf et.al. 2016)

2.3. Summary

In this chapter, an introduction of the state of the art in the humanoid robot research has been given. Different humanoid robot platforms were described with a note on the sensing hardware interface. All the humanoid robots have many sensors dedicated to this purpose. The most common are of course the stereo-cameras but also other kind of sensors are needed due to the limits of the stereo-cameras. For this reason, ASIMO sensing system for instance, is also provided with ultrasonic sensor distributed in various parts of its body.

From the software point of view, different kind of world modelling and planning algorithm were introduced at first. The path planning algorithms described are though not enough to solve the problem of the computation of a suitable collision free trajectory for a humanoid robot which configuration allows to step over obstacles and to traverse terrains which other robot configurations could not.

For this reason, the two footstep planners described at last in the chapter, are 3D planning algorithm which plan directly the position of the feet of the robot.

3. The humanoid robot Archie

This chapter describes the mechanical, electrical and control architecture of the humanoid robot Archie and the lacks of the current design. Its first development started in (Byagowi, 2010) and was first developed as a 31 DOF robot. The development then continued in (Daniali, 2013) and (Dezfouli, 2013). Currently, just the lower part of the body has been developed (torso, hip and leg) and the whole robot has 12 DOF which is the minimum DOF for the stable biped walking (Daniali, 2013).

As showed in Figure 13, the robot has two legs connected by a hip and a torso composed by a spinal column: on every side of the hip, three motors are mounted, one brushed and two brushless. On every leg instead, three brushless motors are mounted. Every side of the robot has thus 6 DOF and the torso position and attitude can be estimated by the Denavit-Hartenberg (DH) parameters of the joints of the legs. The link connecting all the joints of the robot are made in aluminum alloy which provide a good strength with a low weight. Every leg is 68.6 cm long while the robot is 110 cm tall and weighs 20 Kg.

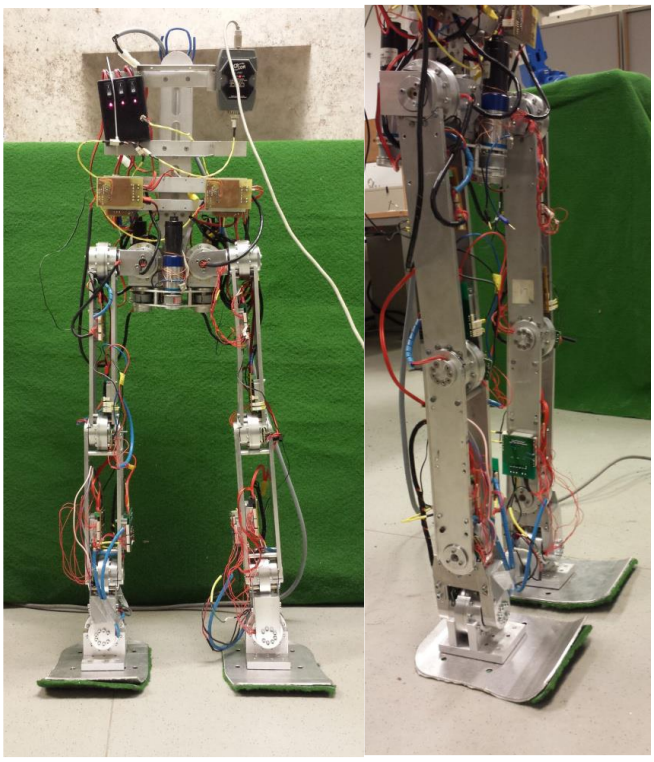

Figure 13: The humanoid robot Archie (Daniali, 2013)

In the next chapter, the motor, the controllers and the electrical configuration will be showed

3.1. Brushless DC motors

The brushless DC motors used on the robot are the Maxxon Motor EC 45 flat. This model is a powerful and compact motor which is perfectly suitable for the joints of a humanoid robot. Figure 14 depicts the components of such a motor and Table 1 lists the technical specification of the motor.

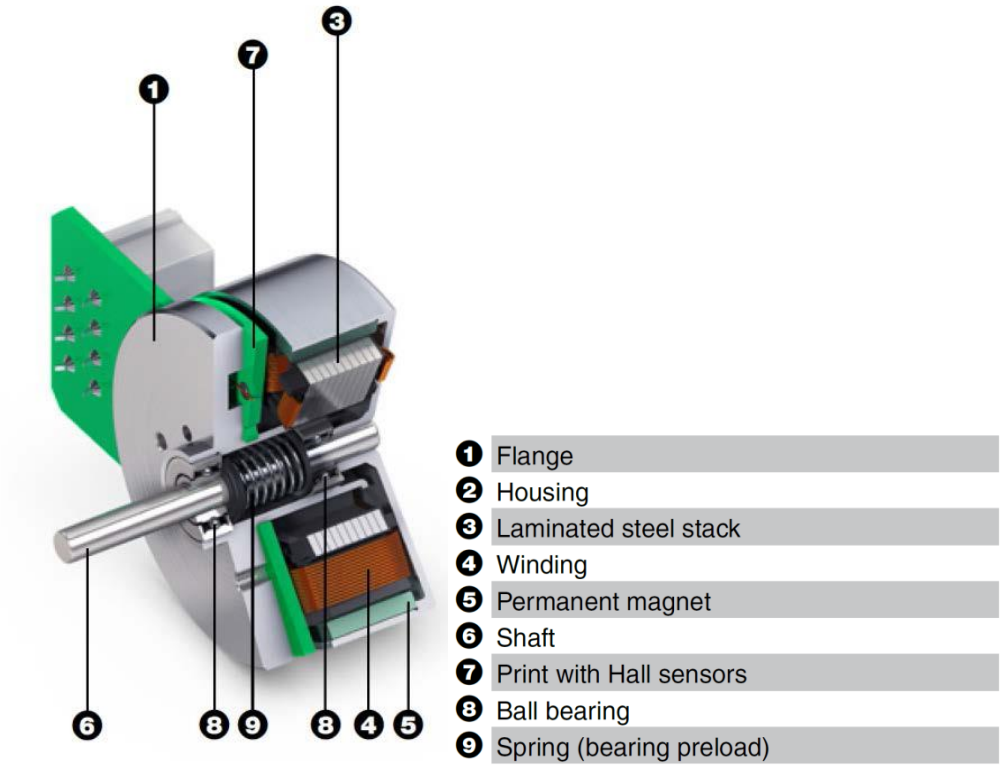

Figure 14: The brushless DC motor used for the actuation of Archie's joints (Daniali, 2013).

Every brushless DC motor used for Archie is attached to a harmonic drive to transmit the movement of the motor to all the joints. This drive has a ratio of 160 and it can tolerate a maximum torque of 76 Nm.

Tree hall sensors are integrated in this motor to report the position of the rotor to drive (Daniali, 2013).

Nominal voltage	24 V
No load speed	6700 rpm
No load current	201 mA
Nominal speed	5260 rpm
Max. continuous torque	84.3 mNm
Max. continuous current	2.36 A
Stall torque	822 mNm
Starting current	24.5 A

Table 1: The technical specification of the brushless DC motor (Daniali, 2013)

3.2. Brushed DC Motors

For the hip, two brushed DC motors are mounted. This choice was done due to the small movement required for these joints. Figure 15 depicts the scheme of the brushed DC motors and Table 2 their technical specification.

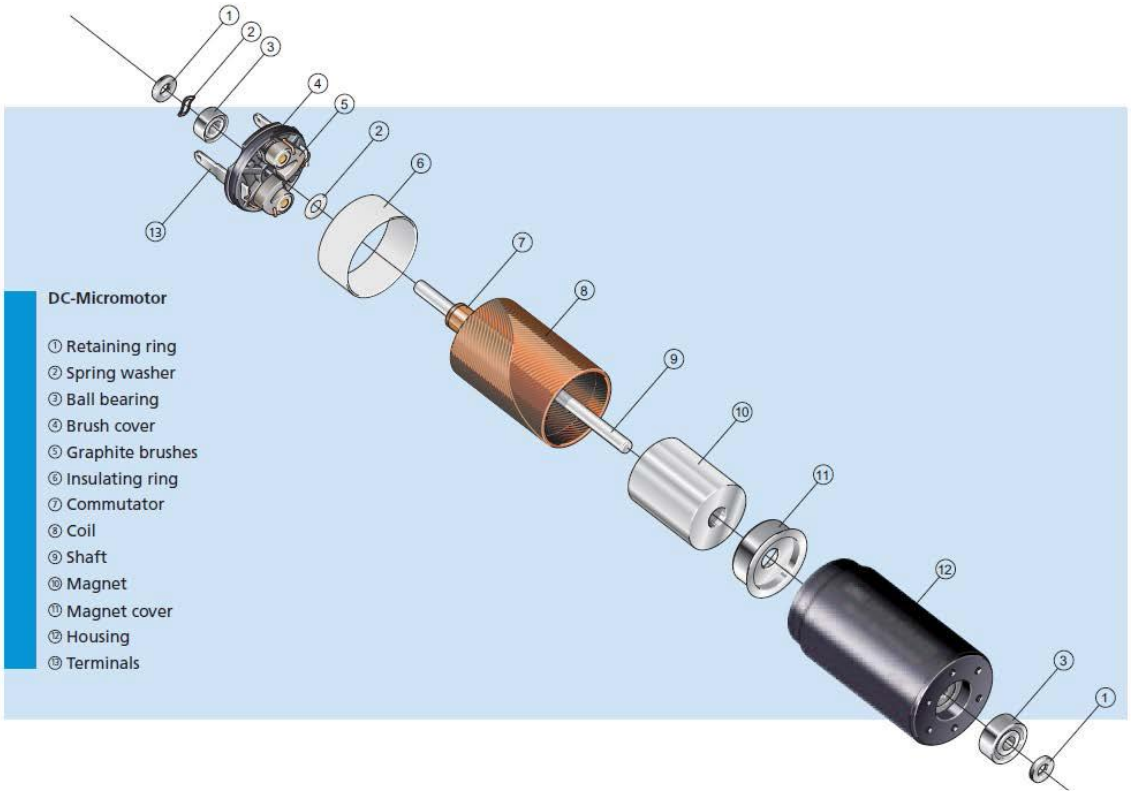

Figure 15: Scheme of the DC motors (Daniali, 2013)

Nominal voltage	24 V
No load speed	5900 rpm
No load current	129 mA
Nominal speed	5000 rpm
Max. continuous torque	70 mNm
Max. continuous current	1.86 A
Stall torque	539 mNm
Starting current	14.3 A

Table 2: Technical specification of the brushed DC motor used in Archie (Daniali, 2013)

Unlike the brushless DC motors, these motors are mounted on a planetary gear in order to transmit the torque to the joints. The ratio of the planetary gear is 415 and it can transmit a maximum torque of 15 Nm. Figure 16 depicts the components of the planetary gear.

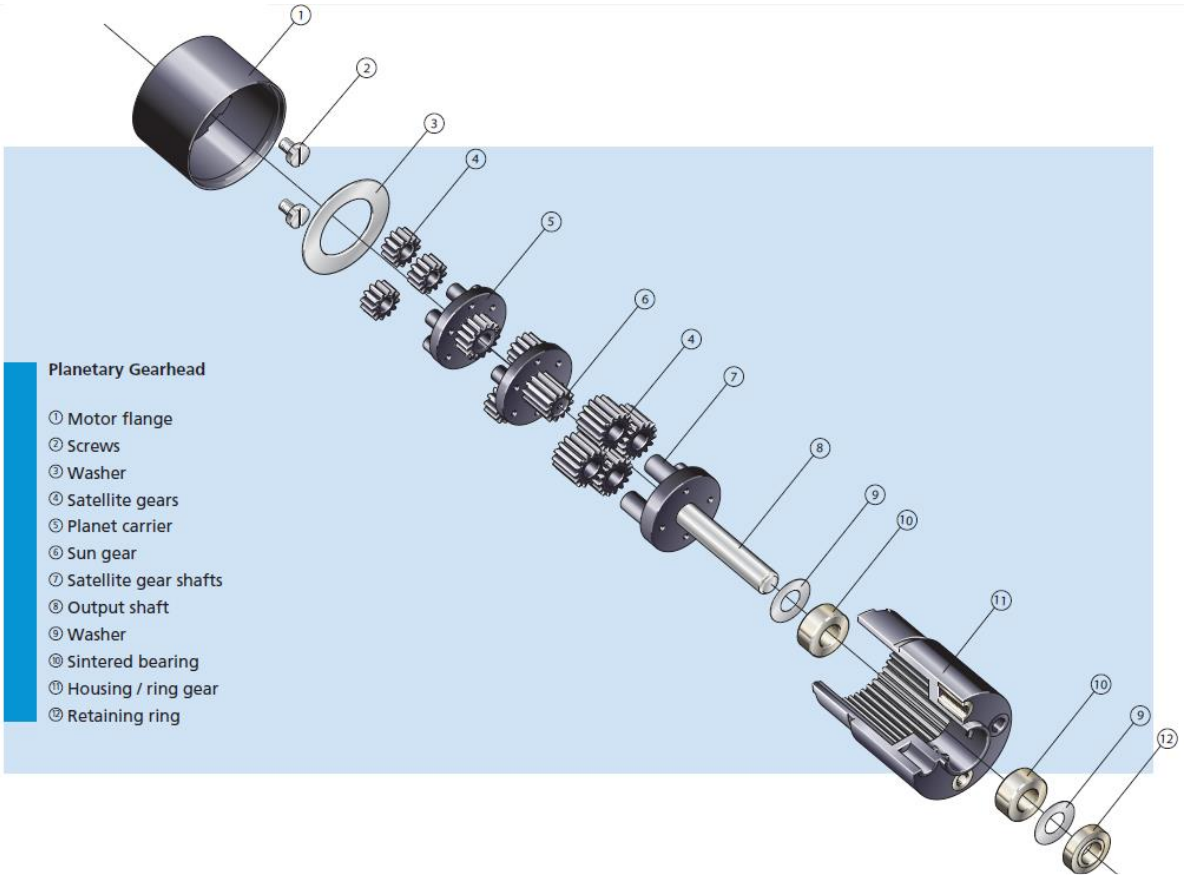

Figure 16: Components of the planetary gear (Daniali,2013)

3.3. Joint controllers

In order to control the joints, an industrial controller named Elmo motion controller is used. It is composed by a PI cascaded controller and a power amplifier. It allows the communication by means of the CAN or of the RS232 protocol. Currently the CAN communication protocol is used to transfer the data between the computer and the controllers. The Elmo motion controller is showed in Figure 17.

Figure 17: Elmo motion controller (Daniali, 2013)

The control algorithm implemented inside the Elmo motion controller is composed of 3 cascaded PI controllers, one for the position, one for the velocity and one for the torque. Every one of these 3 PI controllers work at its own loop rate. The control algorithm scheme is depicted in Figure 18.

As it is possible to see in Figure 18, every control loop uses a gain scheduling function to choose the proper gain for the control algorithm. This is necessary because the joints have some nonlinearities that need to be taken into account.

A system identification on the robot joints has been performed by (Schoerghuber, 2014). This study showed that the joint plan has a non-linear behaviour. There are two main nonlinearities:

- A dead zone at low speed caused by the high reduction gear
- A non-linear transmission behaviour depending on the speed and likely due to friction in the reduction gear and to the non-linear dynamic of the DC motor.

This system identification study produced optimal values for the gain of the PI controllers depending on the speed of the motor and used by the Elmo whistle in the gain scheduling function.

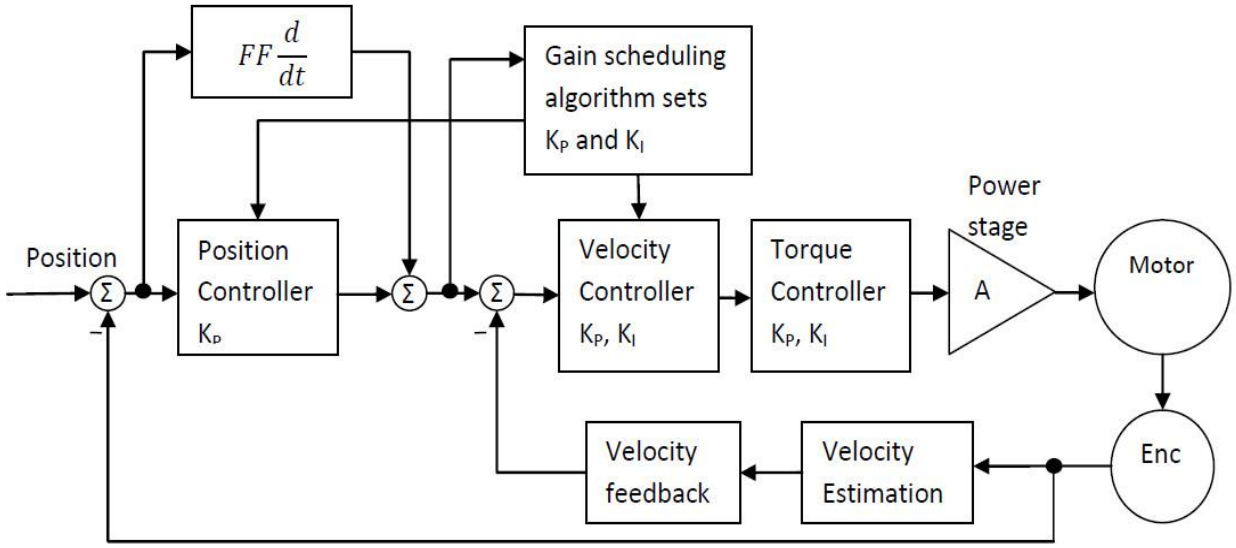

Figure 18: Elmo Whistle motion controller (Byagowi, 2010)

3.4. Hardware integration

The scheme of Archie hardware is instead depicted in Figure 19. In this figure, the dashed blocks are the missing part of the robot. All the controllers are connected in parallel to the power source and to the USB-to-CAN converter. This connection is made possible by means of a PCB specifically designed for the robot. The PCB was also designed to allow a more suitable pin position and allow a more efficient distribution of the power. Figure 20 depicts a brushless DC motor in the harmonic drive connected to the Elmo motion controller by means of the developed PCB.

Figure 19: Scheme of Archie hardware (Dezfouli, 2013)

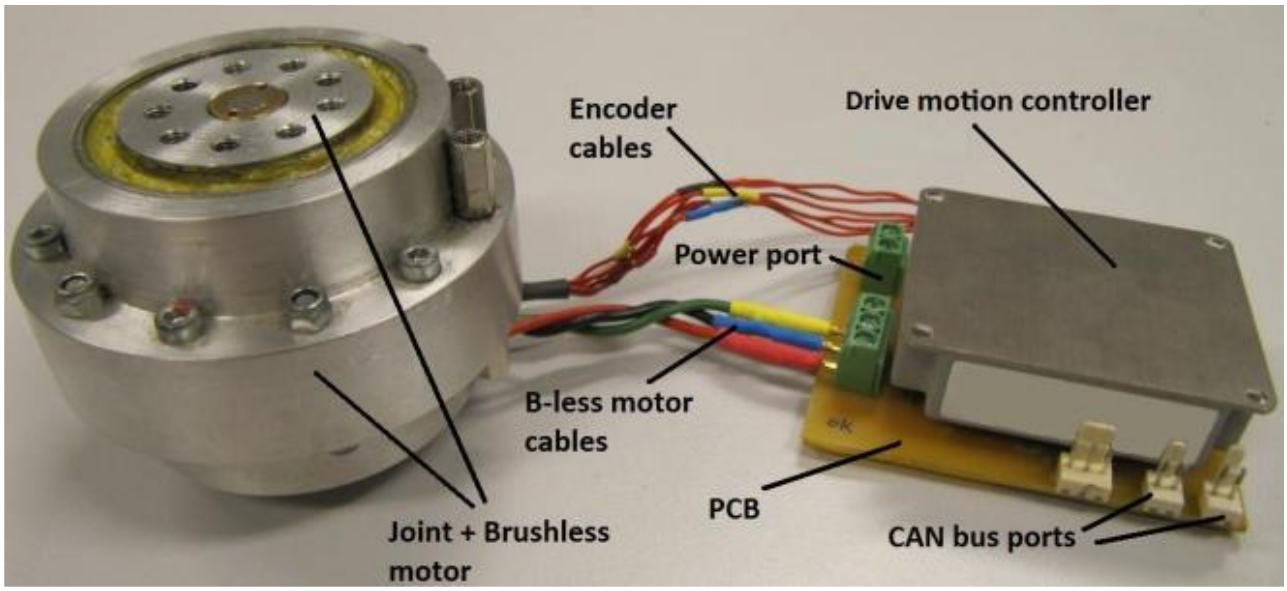

Figure 20: Brushless DC motor in harmonic drive connected to the Elmo motion controller by means of the developed PCB (Dezfouli, 2013)

3.5. Software

The software implemented for the realization of a stable walking for the robot is a C++ program. Using the inverse kinematics, it generates the trajectories of the joints of the robot during multiple gait. These trajectories are the sent via the CAN bus interface in the form of PT (Position Time) or PVT (Position Velocity Time) tables to the controllers. After reading these input data, the controller makes then the corresponding joints move accordingly.

Figure 21 depicts the flow diagram of the current software.

The robot uses a decentralized control approach, i.e. the control of the movement of the joint is entirely handled by the PI controllers and not by the software which only plans the movement before the execution.

The GUI (Graphical User Interface) of the software is depicted in Figure 22. As it is possible to see in the figure, it gives the user the possibility to send commands to the joint controllers, shows a feedback of the CAN messages produced and of the command executed by the motor controllers.

The software runs on an external Linux computer connected with the USB to CAN converter.

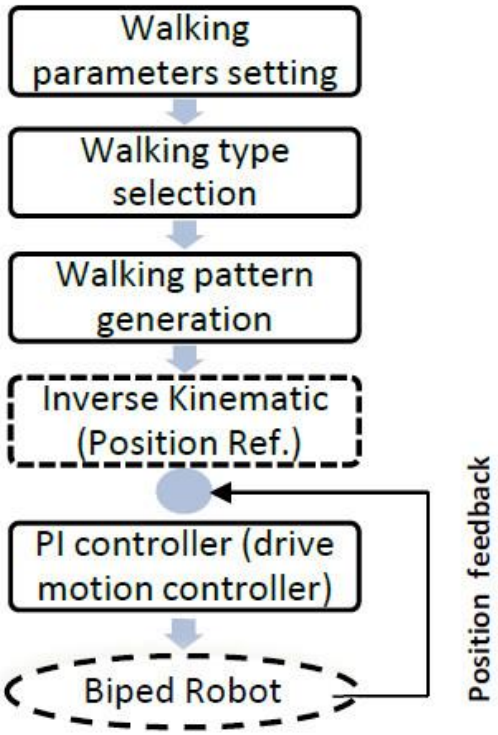

Figure 21: Motion controller flow diagram (Dezfouli, 2013)

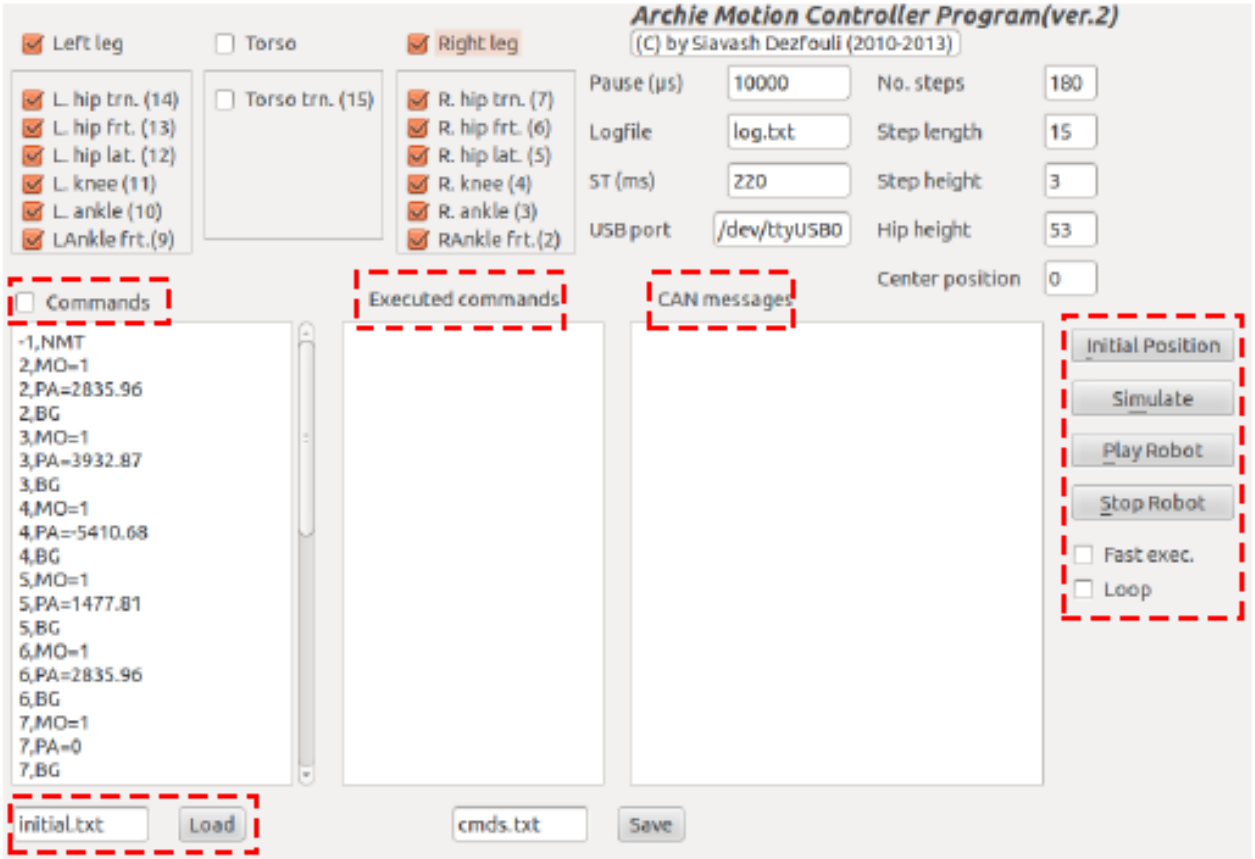

Figure 22: GUI of the current motion controller software (Dezfouli, 2013)

3.6. Summary

To sum up, the mechanical, electronical and software architecture of the humanoid robot Archie was presented in this section. From the hardware point of view, Archie is a 12 DOF humanoid robot composed of two legs, a hip and spinal column. The joints of the legs are actuated by brushless DC motors while on the hip, brushed motors are utilized for the actuation.

All the motors are controlled making use of an industrial controller which communicates with an external computer by means of a CAN interface. The control software runs on an external computer and it only plans the movement of the robot and sends the position planned to the controllers

Compared to the state of the art, it must be pointed out that the robot misses a sensing hardware architecture and an on-board computer. Furthermore, the software provides Archie with a basic walking functionality. There is no possibility of turning or avoiding obstacles. The GUI of the software gives the user the possibility of sending basic commands that will be executed by the joints of the robot. Furthermore, due to the absence of proper sensors and feedback from the controllers, it does not give the user a proper feedback of the status of the robot.

4. Advanced gait planner and control algorithm

As already mentioned, Archie is meant to become a test platform for advanced control algorithm, i.e. a control approach as centralized as possible.

As it is possible to see in Figure 23, the usual approach consists in the use of a high-level controller which outputs the proper command (position, velocity etc.) to the joints' motors receiving in input the feedback from them. The commands are then received from the embedded controllers (usually a PID) which then actuate the motors.

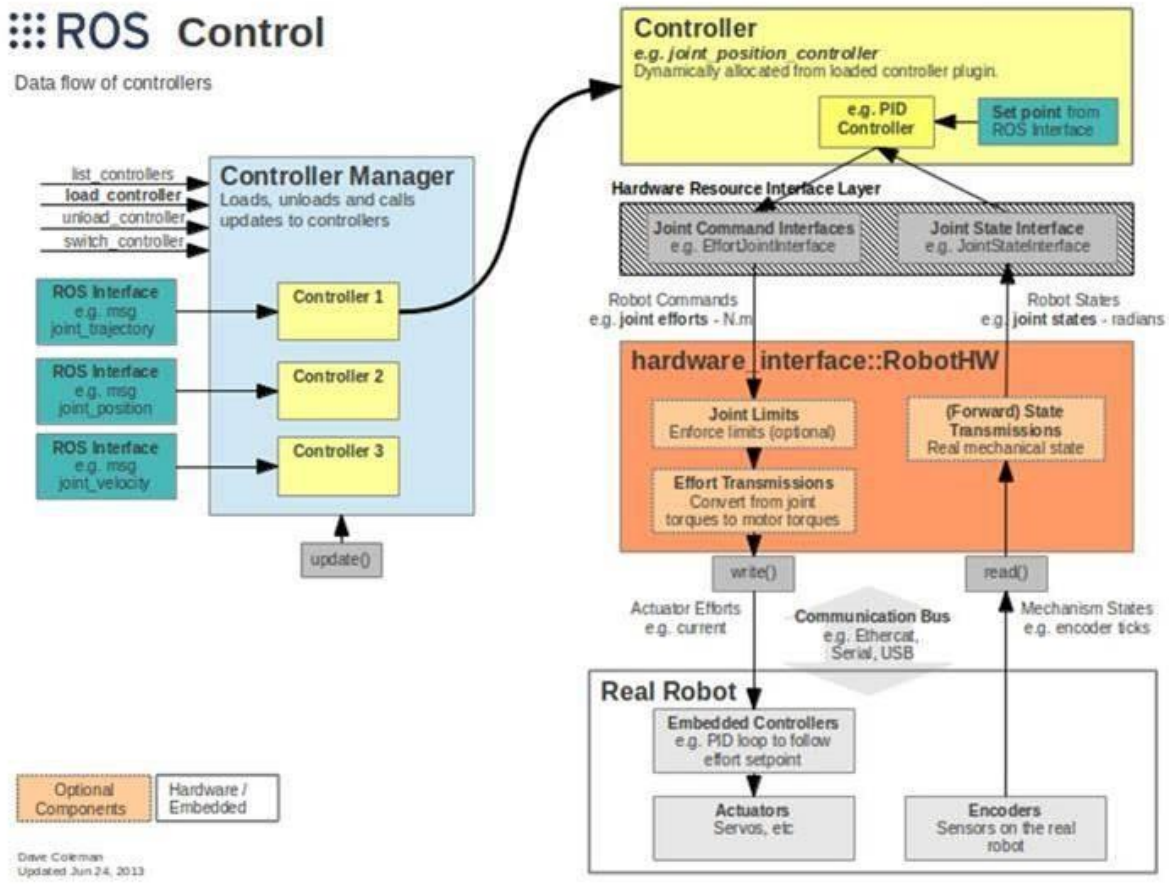

Figure 23: Ros control architecture (Chitta et.al., 2017)

Ideally, this high-level controller should enable Archie to plan the movements of the joints of the legs taking into account the joint constraints of every leg, the feedback coming from the joints, and

the influence that the movement of every joint has on the balance and stability of the full system. The complexity of such a task requires a model (at least kinematic) of the robot.

The planning of the movement can be performed with the walking pattern generation method which is used to compute the movement of the COM of the robot during the gait. The model of the robot can be provided by the forward and inverse kinematics and the stabilization can be performed with several different methods based on the walking pattern generation.

This chapter introduces one of the gait planning and control algorithm currently in evaluation for Archie. First, the walking pattern generation method will be presented, followed by the forward and inverse kinematics of the robot. Subsequently, an overview of the possible approaches for the stabilization of the system will be given. Finally, this chapter will provide a description of the approach in evaluation. In particular, it will be described the planning of the joint movements during the gait, hat has been developed in the course of this PhD work for the control strategy in evaluation. This planning was, at first, thought to be included in the footstep planner subject of this dissertation. Finally, it will be presented a discussion on the extendibility of the approach in evaluation when the upper body will be built.

4.1. COM Planning algorithm

Let us approximate the robot as a 3D linear inverted pendulum. Let us assume that all the mass of the robot is concentrated in its centre of mass (COM) and that it is connected to the ground (a huge 3D plane) by means of a massless leg, as depicted in Figure 24.

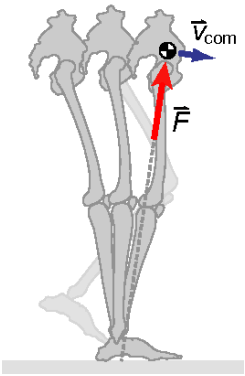

Figure 24: leg-hip system approximated as an inverted pendulum (Kuo, 2007)

It is possible to decompose the force f, named kick force in the three components parallel to the three axes:

$$\begin{aligned} f_x &= \left(\frac{x}{r}\right) f \\ f_y &= \left(\frac{y}{r}\right) f \\ f_z &= \left(\frac{z}{r}\right) f \end{aligned} \tag{1}$$

It can be noticed that only gravity and the kick force act on the inverted pendulum. Its dynamics equations are then:

$$\begin{aligned} M\ddot{x} &= \left(\frac{x}{r}\right) f \\ M\ddot{y} &= \left(\frac{y}{r}\right) f \\ M\ddot{z} &= \left(\frac{z}{r}\right) f - Mg \end{aligned} \tag{2}$$

Let's define the constraint plane on which the inverted pendulum will move:

$$z = k_x x + k_y y + z_c \tag{3}$$

k_x and k_y define the slope of the constraint plane while z_c define the height of the plane. The inverted pendulum can move on this plane only if its acceleration is perpendicular to it. Thus,

$$\begin{bmatrix} f\left(\frac{x}{r}\right) & f\left(\frac{y}{r}\right) & f\left(\frac{z}{r}\right) \end{bmatrix} \begin{bmatrix} -k_x \\ -k_y \\ 1 \end{bmatrix} = 0 \tag{4}$$

If one solves equation (4) for f and substitute it in the equation of the constraint plane, one get (5)

$$f = \frac{Mgr}{z_c} \tag{5}$$

Substituting equation (5) in the dynamic's equation leads to the acceleration component of the inverted pendulum in the cartesian plane one gets:

$$\begin{aligned} \ddot{x} &= \frac{g}{z_c} x \\ \ddot{y} &= \frac{g}{z_c} y \end{aligned} \tag{6}$$

These equations have only z_c as parameter. The slopes k_x and k_y do not influence the motion of the linear inverted pendulum and consequently of the robot's COM. Figure 25 shows a linear inverted pendulum moving along the constraint plane. The analytical solution of equations (6) is as follows:

$$\begin{aligned} x(t) &= x(0)\cosh\left(\frac{t}{T_C}\right) + T_C\dot{x}(0)\sinh\left(\frac{t}{T_c}\right) \\ x(t) &= \frac{\dot{x}(0)}{T_C}\text{sınh}\left(\frac{t}{T_c}\right) + \dot{x}(0)\cosh\left(\frac{t}{T_C}\right) \end{aligned} \tag{7}$$

For the x. For the y, similarly, one gets:

$$\begin{aligned} y(t) &= y(0)\cosh\left(\frac{t}{T_C}\right) + T_C\dot{y}(0)\sinh\left(\frac{t}{T_c}\right) \\ y(t) &= \frac{\dot{y}(0)}{T_C}\text{sınh}\left(\frac{t}{T_c}\right) + \dot{y}(0)\cosh\left(\frac{t}{T_C}\right) \end{aligned} \tag{8}$$

With

$$T_c = \sqrt{\frac{z_C}{g}} \tag{9}$$

Figure 26 depicts instead a walking pattern generated with the inverted pendulum model dynamics. Every piece of the trajectory coloured with a different colour is the motion of an inverted pendulum having its own support base. The z_c parameter is considered fixed for every inverted pendulum model.

Figure 27 depicts one of the pieces of the generated walking pattern with the diagrams of the components of the velocity in the cartesian plane.

The piece of the computed trajectory depicted in Figure 27, is computed in the time period $[0, T_{sup}]$ where T_{sup} is called support period and it represents the time period in which one of the two feet of the robot is the support foot. This piece of trajectory is called walking primitive and, as it is possible to see in the figure, has a hyperbolic behaviour with a symmetry along the y axis.

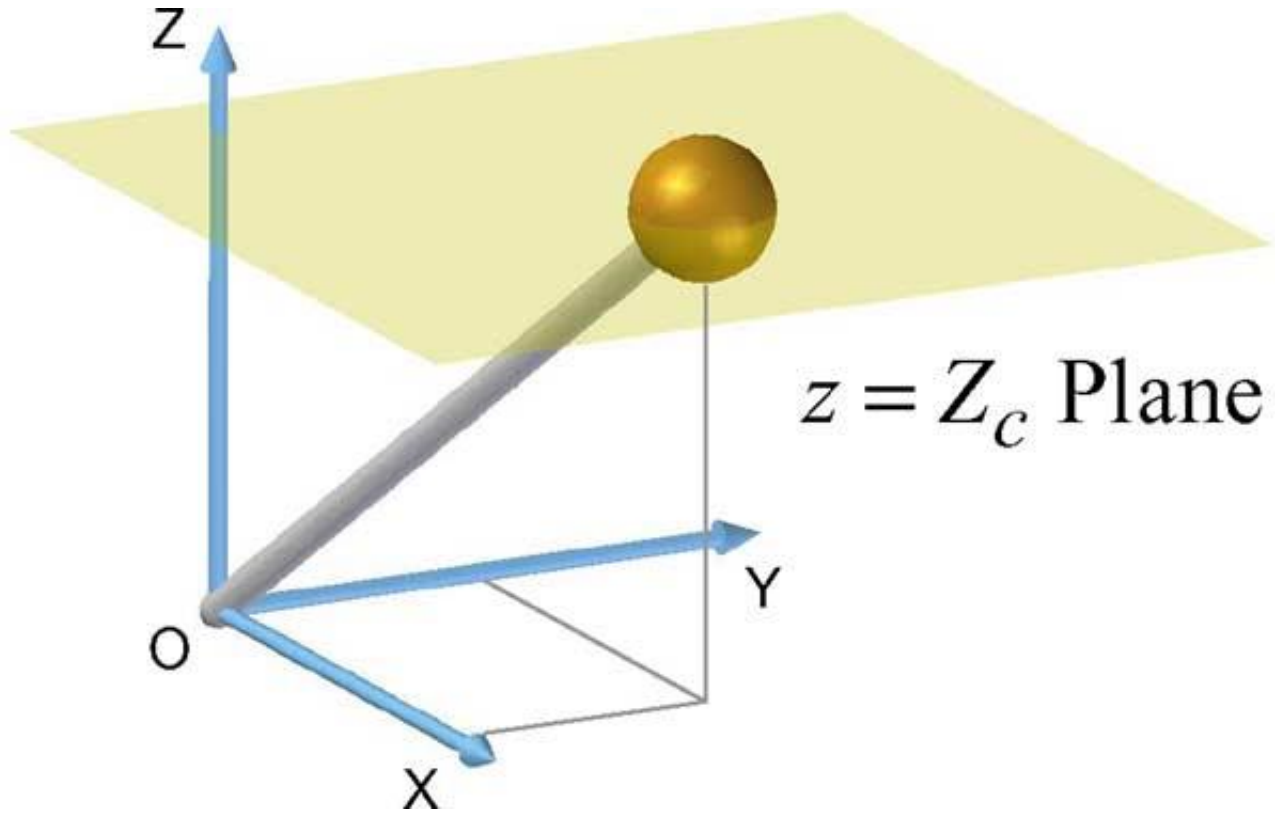

Figure 25: Inverted pendulum moving on its constraint plane (Lee et.al., 2008)

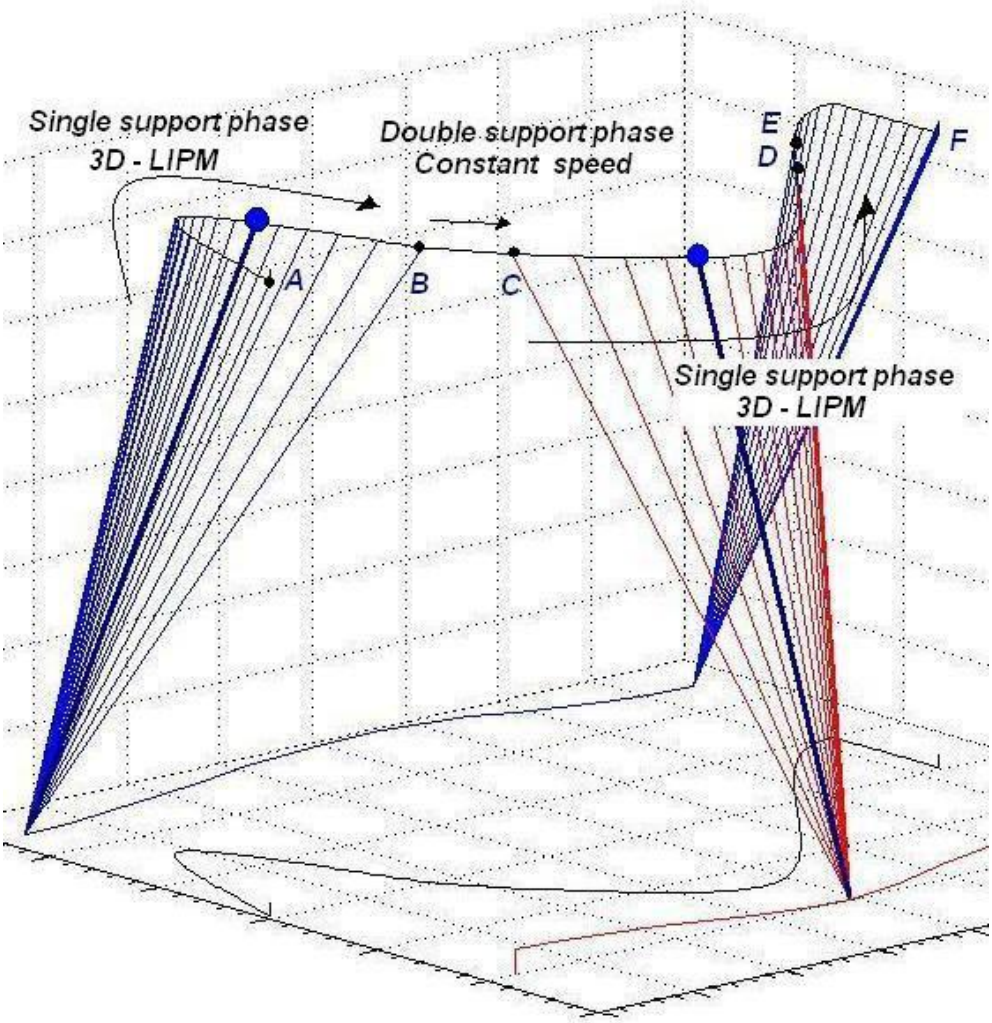

Figure 26: Walking patter generated with the inverted pendulum model dynamics (Arbulú et.al., 2010)

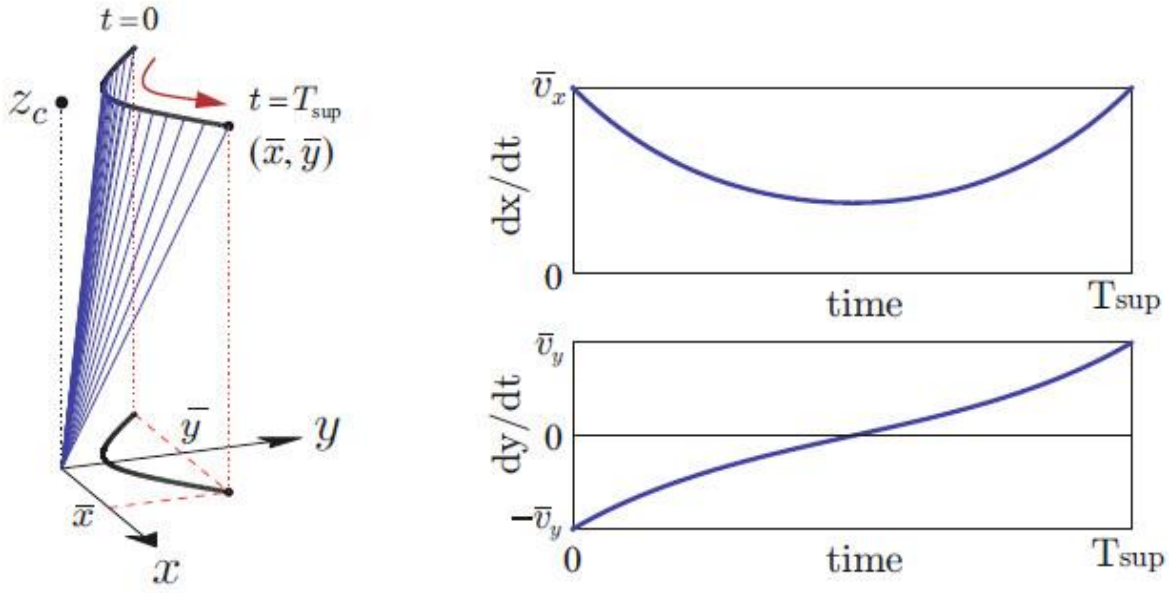

Figure 27: inverted pendulum motion in the generated walking pattern and diagrams of the velocity on x and y with time (Kajita et.al., 2014)

Due to its symmetry, the walking primitive is defined only by its terminal position $(\bar{x} \quad \bar{y})$. It is possible to compute the terminal speed $(\overline{v_x} \quad \overline{v_y})$ analytically. As a matter of facts, considering the walking primitive depicted in Figure 27, the initial conditions of the x are $(-\bar{x} \quad \overline{v_x})$ and the initial condition of the y are $(\bar{y} \quad -\overline{v_y})$. By substituting the initial condition in the analytical solution of the inverted pendulum dynamic equation one get:

$$\begin{aligned} \bar{x} &= -\bar{x}C + T_C\overline{v_x}S \\ \bar{y} &= \bar{y}C - T_C\overline{v_y}S \end{aligned} \tag{10}$$

Solving equation (10) for the terminal velocity leads to:

$$\begin{aligned} \overline{v_x} &= \frac{\bar{x}(C+1)}{T_C S} \\ \overline{v_y} &= \frac{\bar{y}(C-1)}{T_C S} \end{aligned} \tag{11}$$

With:

$$C = \cosh\left(\frac{T_{sup}}{T_C}\right) \quad S = \sinh\left(\frac{T_{sup}}{T_C}\right) \tag{12}$$

Using the walking primitives allows to easily compute a walking trajectory as follows. For this purpose, it is frequently required to directly specify the foot placement (Kajita et.al., 2014).

The first footstep $\left(p_x^{(0)} \quad p_y^{(0)}\right)$ is the initial position of the first support foot. From the footstep positions, it is possible to determine the walk primitive as:

$$\begin{bmatrix} \bar{x}^{(n)} \\ \bar{y}^{(n)} \end{bmatrix} = \begin{bmatrix} \frac{s_x^{(n+1)}}{2} \\ (-1)^n \frac{s_y^{(n+1)}}{2} \end{bmatrix} \tag{13}$$

Where s_x and $s_y$are respectively step length and width.

As mentioned before, the walk primitive can be determined just by its terminal state. This can be noted in the equation where the nth walk primitive depends from the n+1th walk parameters.

The terminal velocity can then be calculated by (11) as follows:

$$\begin{bmatrix} \bar{v}_x^{(n)} \\ \bar{v}_y^{(n)} \end{bmatrix} = \begin{bmatrix} \frac{(C+1)}{T_c S} \bar{x}^{(n)} \\ \frac{(C-1)}{T_c S} \bar{y}^{(n)} \end{bmatrix} \tag{14}$$

The method creates a series of walk primitives which compose a walking pattern. The walk primitive computed in this way though are discontinuous and the walking pattern thus is not realizable. To solve this problem; it is possible to adjust its foot placement in order to control its speed. The intuitive explanation of this is depicted in Figure 28. On the left side of the figure it is depicted the process of speeding up by taking a shorter step and to slow down taking a longer step.

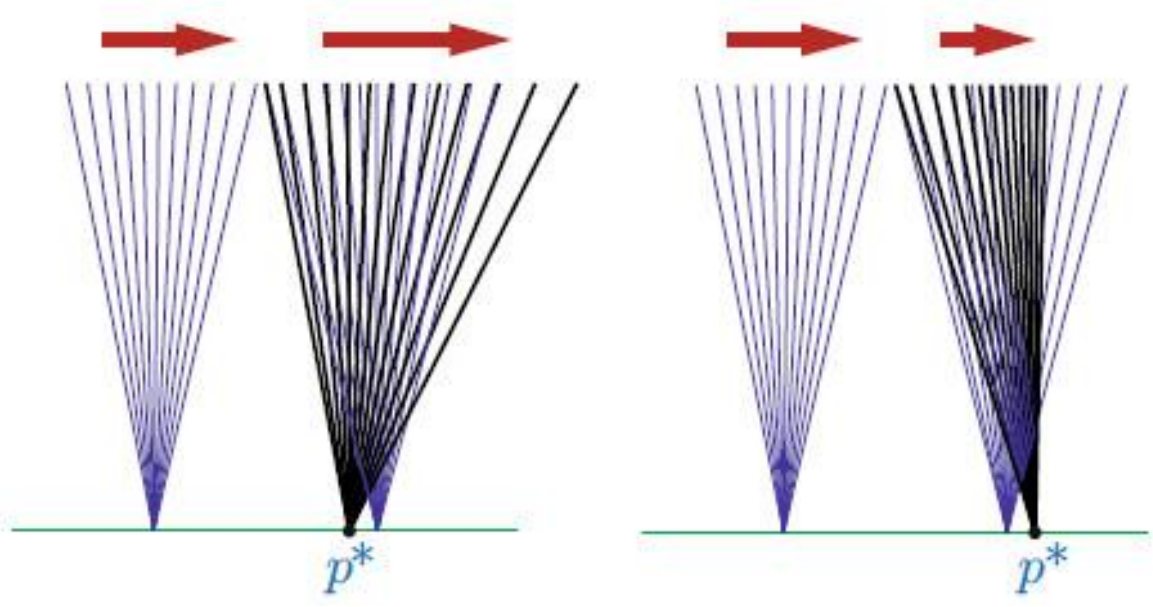

Figure 28: Intuitive explanation of the modified foot placement method (Kajita et.al., 2014)

Considering the modified foot placement in the dynamic equation of the inverted pendulum leads to (15).

$$\ddot{x} = \frac{g}{z_C}(x - p_x^*)$$
$$\ddot{y} = \frac{g}{z_C}(y - p_y^*) \tag{15}$$

Which analytical solution is:

$$x(t) = \left(x_i^{(n)} - p_x^*\right)\cosh\left(\frac{t}{T_C}\right) + T_C\dot{x}_i^{(n)}\sinh\left(\frac{t}{T_C}\right) + p_x^*$$
$$\dot{x(t)} = \frac{x_i^{(n)} - p_x^*}{T_C}\sinh\left(\frac{t}{T_C}\right) + \dot{x}_i^{(n)}\cosh\left(\frac{t}{T_C}\right) \tag{16}$$

On the x. Similarly, for on the y axis one has:

$$y(t) = \left(y_i^{(n)} - p_y^*\right)\cosh\left(\frac{t}{T_C}\right) + T_C\dot{y}_i^{(n)}\sinh\left(\frac{t}{T_C}\right) + p_y^*$$
$$\dot{y(t)} = \frac{y_i^{(n)} - p_y^*}{T_C}\sinh\left(\frac{t}{T_C}\right) + \dot{y}_i^{(n)}\cosh\left(\frac{t}{T_C}\right) \tag{17}$$

From this set of equation, one can find the relation between the modified foot placement and the final state of the nth step:

$$\begin{bmatrix} x_f^{(n)} \\ \dot{x}_f^{(n)} \end{bmatrix} = \begin{bmatrix} C & T_C S \\ \frac{S}{T_C} & C \end{bmatrix}\begin{bmatrix} x_i^{(n)} \\ \dot{x}_i^{(n)} \end{bmatrix} + \begin{bmatrix} 1 - C \\ -\frac{S}{T_C} \end{bmatrix} p_x^* \tag{18}$$

$$\begin{bmatrix} y_f^{(n)} \\ \dot{y}_f^{(n)} \end{bmatrix} = \begin{bmatrix} C & T_C S \\ \frac{S}{T_C} & C \end{bmatrix}\begin{bmatrix} y_i^{(n)} \\ \dot{y}_i^{(n)} \end{bmatrix} + \begin{bmatrix} 1 - C \\ -\frac{S}{T_C} \end{bmatrix} p_y^* \tag{19}$$

Consider as the target position, the final state of the walk primitive presented in the ground frame described by the (18) and (19) as shown in (20) and (21)

$$\begin{bmatrix} x^d \\ \dot{x}^d \end{bmatrix} = \begin{bmatrix} p_x^{(n)} + \bar{x}^{(n)} \\ \bar{v}_x^{(n)} \end{bmatrix} \tag{20}$$

$$\begin{bmatrix} y^d \\ \dot{y}^d \end{bmatrix} = \begin{bmatrix} p_y^{(n)} + \bar{y}^{(n)} \\ \bar{v}_y^{(n)} \end{bmatrix} \tag{21}$$

In order to compute the foot placement which ends with the closest final state to the target one can define the evaluation function:

$$N_x = a\left(x^d - x_f^{(n)}\right)^2 + b\left(\dot{x}^d - \dot{x}_f^{(n)}\right)^2 \tag{22}$$

$$N_y = a\left(y^d - y_f^{(n)}\right)^2 + b\left(\dot{y}^d - \dot{y}_f^{(n)}\right)^2 \tag{23}$$

With a and b positive weights. By setting to zero the derivatives of N_x and N_y with respect of p_x^* and p_y^* one can compute the modified foot placement which will minimize N_x and N_y:

$$\begin{aligned} p_x^* &= -\frac{a(C-1)}{D}\left(x^d - Cx_i^{(n)} - T_C S\dot{x}_i^{(n)}\right) - \frac{bS}{T_C D}\left(\dot{x}^d - \frac{S}{T_C}x_i^{(n)} - C\dot{x}_i^{(n)}\right) \\ p_y^* &= -\frac{a(C-1)}{D}\left(y^d - Cy_i^{(n)} - T_C S\dot{y}_i^{(n)}\right) - \frac{bS}{T_C D}\left(\dot{y}^d - \frac{S}{T_C}y_i^{(n)} - C\dot{y}_i^{(n)}\right) \\ D &= a(C-1)^2 + b\left(\frac{S}{T_C}\right)^2 \end{aligned} \tag{24}$$

An example of walking pattern computed with the foot placement modification explained above is showed in Figure 29. In this figure the walking pattern generated is along a straight line. For changing the direction of motion, it is just necessary to make a few changings in the method described above.

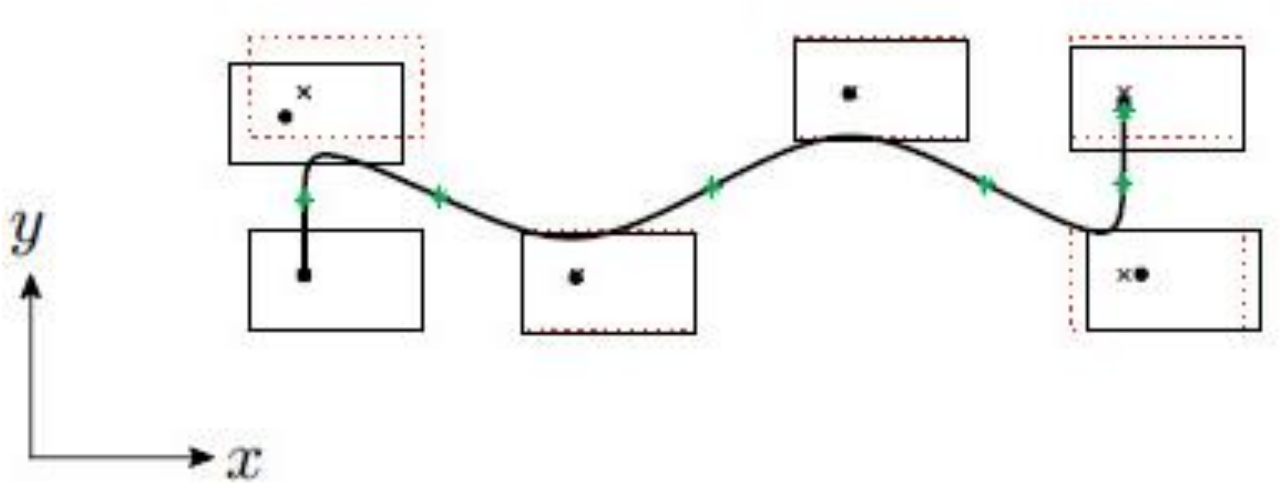

Figure 29: Walking pattern generated with the modified foot placement method (Kajita et.al., 2014)

For changing the direction of motion, one just need to consider also the parameter s_θ indicating the heading of the feet.

The walk primitive of the n[th] steps is then:

$$\begin{bmatrix} \bar{x}^{(n)} \\ \bar{y}^{(n)} \end{bmatrix} = \begin{bmatrix} \cos s_\theta^{(n+1)} & -\sin s_\theta^{(n+1)} \\ \sin s_\theta^{(n+1)} & \cos s_\theta^{(n+1)} \end{bmatrix} \begin{bmatrix} \frac{s_x^{(n+1)}}{2} \\ (-1)^n \frac{s_y^{(n+1)}}{2} \end{bmatrix} \tag{25}$$

And finally, the speed of the walk primitive becomes:

$$\begin{bmatrix} \bar{v}_x^{(n)} \\ \bar{v}_y^{(n)} \end{bmatrix} = \begin{bmatrix} \cos s_\theta^{(n+1)} & -\sin s_\theta^{(n+1)} \\ \sin s_\theta^{(n+1)} & \cos s_\theta^{(n+1)} \end{bmatrix} \begin{bmatrix} \frac{1+C}{T_C S} \bar{x}^{(n)} \\ \frac{C-1}{T_C S} \bar{y}^{(n)} \end{bmatrix} \tag{26}$$

Substituting these last equations with the correspondent ones in the method explained before will lead to a walking pattern with a direction changing. It is possible to note that, if $s_\theta^{(n)}$ is zero, the equations will result in a straight trajectory as the one in Figure 29.

Figure 30 shows the walking pattern generated with the inclusion of the last walking parameter.

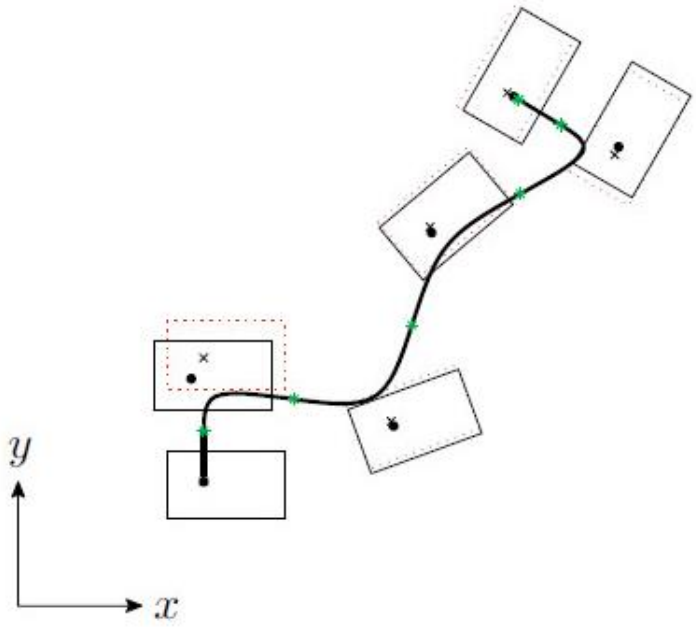

Figure 30: walking pattern with changing direction (Kajita et.al., 2014)

4.2. Forward kinematics of the current Archie configuration

Forward and inverse kinematics are two methods widely used in robotics. The first one allows the calculation of the position of the end effector based on the values of the joint angles. The inverse kinematics instead provide the joint angles based on the position of the end effector.

The forward kinematics can be solved easily using the transformation matrices. These matrices are matrices composed by a rotational part and a translational part. For the construction of these transformation matrices, different parameters called Denavit-Hartemberg parameters are used.

Considering the two connected links depicted in Figure 31, the Denavit-Hartenberg parameters are defined as follows:

- a_i, defined as distance between O_i and O_i,
- d_i, defined as the coordinate of O_i on the axis z_{i-1}
- α_i, defined as the angle between z_{i-1} and z_i around the axis x_i
- θ_i, defined as the angle between x_{i-1} and x_i around the axis z_i

All the angles are defined positive in direction counter-clockwise. The parameters a_i and α_i are always constant and their values depend on the mechanical construction of the links. Of the parameters d_i and θ_i just one of them is constant. Depending on the kind of joint, one of them is constant and the other is not. If the joint is prismatic just the parameter d_i is the variable while θ_i is constant. If the joint is rotational instead, the variable is θ_i while d_i is constant.

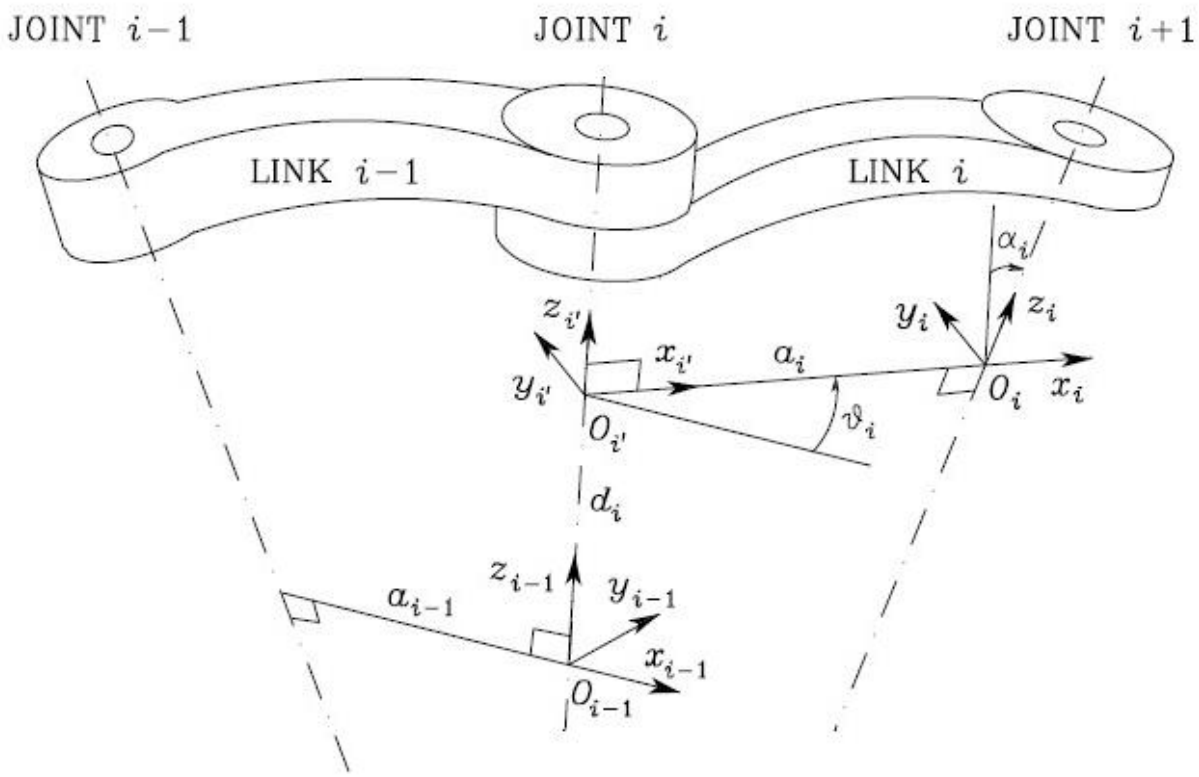

Figure 31: Denavit-Hartenberg parameters for a kinematic chain (Siciliano et.al., 2009)

These parameters allow us to compute the position and orientation of the link i from the known frame $i-1$ with the following steps:

- From the frame $i-1$, translate the frame of d_i along the axis z_{i-1} and rotate it of θ_i around z_{i-1} obtaining the frame i'. The transformation matrix is then:

$$A_{i'}^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{27}$$

- Perform a further translation along the axis x_i, and rotate the frame i' of α_i, around the axis x_i, obtaining the frame i. The transformation matrix for this last step is:

$$A_i^{i'} = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{28}$$

The total transformation matrix can be computed by means of the post-multiplication of the two transformation matrices:

$$A_i^{i-1} = A_{i'}^{i-1} \cdot A_i^{i'} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

For an open kinematic chain like the one showed in Figure 32, it is possible to compute the pose of the last ith frame post-multiplying the transformation matrices of the frames 0 to i-1th.

$$T_n^0 = A_1^0 \cdot A_2^1 \cdot A_3^2 \cdot \dots A_n^{n-1} \tag{30}$$

Considering a kinematic chain like a manipulator or like the legs of a humanoid robot, it is possible to compute the position and attitude of the end effector computing the transformation matrix from the frame of the joint 0 (the base of the manipulator) to the frame n (the end effector).

Figure 33 and Figure 34 depicts the kinematic chain of, respectively, the left and the right leg of Archie. For the kinematic chain of Archie's left leg, it is possible to define the Denavit-Hartenberg parameters showed in Table 3 and taken from (Daniali, 2013).

Table 4 taken from (Daniali, 2013) instead, shows the Denavit-Hartenberg parameters of Archie's right leg.

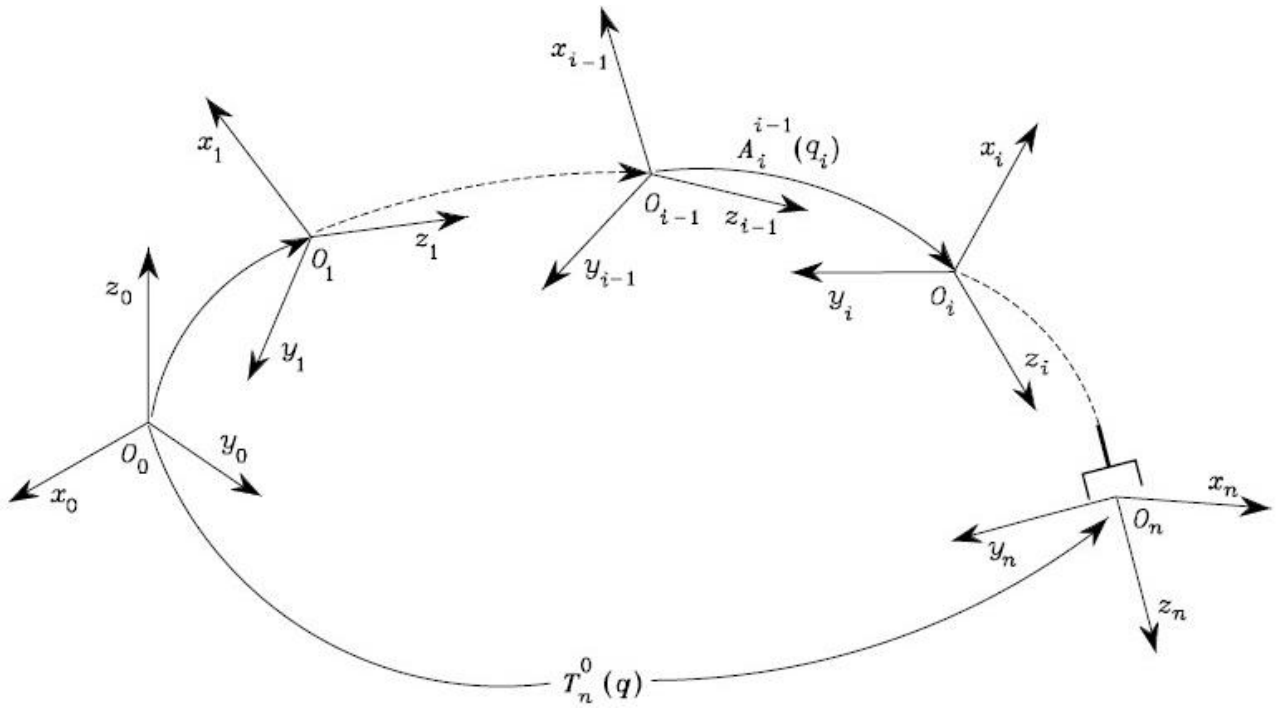

Figure 32: Kinematic chain (Siciliano et.al., 2009)

Parameter	a_i	α_i	d_i	θ_i	θ_0
Link 1	a_1	$\pi/2$	0	θ_1	0
Link 2	a_2	0	0	θ_2	0
Link 3	a_3	0	$-d_3$	θ_3	0
Link 4	0	$-\pi/2$	0	θ_4	0
Link 5	0	$-\pi/2$	0	θ_5	0
Link 6	a_6	0	d_6	θ_6	0

Table 3: Denavit Hartenberg parameters of Archie's left leg (Daniali, 2013)

Parameter	a_i	α_i	d_i	θ_i	θ_0
Link 1	a_1	$-\pi/2$	0	θ_1	0
Link 2	a_2	0	0	θ_2	0
Link 3	a_3	0	$-d_3$	θ_3	0
Link 4	0	$\pi/2$	0	θ_4	0
Link 5	0	$-\pi/2$	0	θ_5	0
Link 6	$-a_6$	0	d_6	θ_6	0

Table 4: Denavit Hartenberg parameters of Archie's right leg (Daniali, 2013)

Considering the Denavit-Hartenberg parameters of the right leg, the transformation matrices of the joints have been computed in the course of this PhD work and are showed in (31)-(36). For clarity, it has to be noted that further in this chapter the joints will be named with the numeric convention used in Figure 33 and Figure 34 followed by the letter "*l*" in case of left leg and "*r*" in case of right leg. Joint number 6 instead is the COM.

$$A_{1r}^{0r} = \begin{bmatrix} \cos\theta_0 & 0 & \sin\theta_0 & a_1\cos\theta_0 \\ \sin\theta_0 & 0 & -\cos\theta_0 & a_1\sin\theta_0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{31}$$

$$A_{2r}^{1r} = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_2\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_2\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{32}$$

$$A_{3r}^{2r} = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_3\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_3\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{33}$$

$$A_{4r}^{3r} = \begin{bmatrix} \cos\theta_3 & 0 & \sin\theta_3 & 0 \\ \sin\theta_3 & 0 & -\cos\theta_3 & 0 \\ 0 & 1 & 0 & -d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{34}$$

$$A_{5r}^{4r} = \begin{bmatrix} \cos\theta_4 & 0 & \sin\theta_4 & 0 \\ \sin\theta_4 & 0 & -\cos\theta_4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{35}$$

$$A_{6}^{5r} = \begin{bmatrix} \cos\theta_5 & -\sin\theta_5 & 0 & a_6\cos\theta_5 \\ \sin\theta_5 & \cos\theta_5 & 0 & a_6\sin\theta_5 \\ 0 & 0 & 1 & -d_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{36}$$

Similar transformation matrices can be obtained for the right leg. The transformation matrices from the joint 0 to the COM of the robot can be computed by multiplying the transformation matrices (30) to (35) as showed in (37)

$$A_6^{ol} = A_{1l}^{ol} A_{2l}^{1l} A_{3l}^{2l} A_{4l}^{3l} A_{5l}^{4l} A_6^{5l} \tag{37}$$

Using relation (37), and taking into account the symmetry of the system, the coordinates of the joints of both legs, in the fixed reference system based on the right foot, depicted in Figure 34, have been defined, in the curse of this PhD work, with the relations depicted in (38)-(40).

$$x: \begin{cases} x_{fl} = 0 \\ x_{ol} = x_{fl} \\ x_{1l} = x_{ol} \\ x_{2l} = a_{2l} \sin\theta_{1l} \\ x_{3l} = x_{2l} + a_{3l} \sin\theta_{12l} \\ x_{4l} = x_{3l} \\ x_{5l} = x_{4l} - d_6 \sin\theta_{123l} \\ x_{5r} = x_{5l} + a_6 \cos\theta_{40l} \cos\theta_{123l} \sin\theta_{5l} \\ x_{4r} = x_{5r} + d_6 \sin\theta_{123l} \\ x_{3r} = x_{4r} + d_3 \cos\theta_{40l4r} \cos\theta_{123l} \sin\theta_{5l5r} \\ x_{2r} = x_{3r} + a_{3r} \sin\theta_{123l3r} \\ x_{1r} = x_{2r} + a_{2r} \sin\theta_{123l32r} \\ x_{0r} = x_{1r} + a_{1r} \sin\theta_{123l321r} \\ x_{fr} = x_{0r} + a_{0r} \sin\theta_{123l321r} \end{cases} \tag{38}$$

$$y: \begin{cases} y_{fl} = 0 \\ y_{0l} = y_{fl} \\ y_{1l} = y_{0l} + a_{1l} \sin\theta_{0l} \\ y_{2l} = y_{1l} + a_{2l} \sin\theta_{0l} \cos\theta_{1l} \\ y_{3l} = y_{2l} + a_{3l} \sin\theta_{0l} \cos\theta_{12l} \\ y_{4l} = y_{3l} + d_3 \cos\theta_{0l} \\ y_{5l} = y_{4l} + d_6 \sin\theta_{04l} \\ y_{5r} = y_{5l} + a_6 \cos\theta_{04l} \cos\theta_{123l} \cos\theta_{5l} \\ y_{4r} = y_{5r} - d_6 \sin\theta_{04l} \\ y_{3r} = y_{4r} + d_3 \cos\theta_{04l4r} \cos\theta_{123l} \cos\theta_{5l5r} \\ y_{2r} = y_{3r} + a_{3r} \sin\theta_{04l4r} \cos\theta_{123l3r} \\ y_{1r} = y_{2r} + a_{2r} \sin\theta_{04l4r} \cos\theta_{123l32r} \\ y_{0r} = y_{1r} + a_{1r} \sin\theta_{04l4r} \cos\theta_{123l321r} \\ y_{fr} = y_{0r} + a_{0r} \sin\theta_{04l40r} \cos\theta_{123l321r} \end{cases} \tag{39}$$

$$z: \begin{cases} z_{fl} = 0 \\ z_{0l} = d_0 \\ z_{1l} = z_{0l} + a_{1l} \cos\theta_{0l} \\ z_{2l} = z_{1l} + a_{2l} \cos\theta_{0l} \cos\theta_{1l} \\ z_{3l} = z_{2l} + a_{3l} \cos\theta_{0l} \cos\theta_{12l} \\ z_{4l} = z_{3l} + d_3 \sin\theta_{0l} \\ z_{5l} = z_{4l} - d_6 \cos\theta_{04l} \cos\theta_{123l} \\ z_{5r} = z_{5l} + a_6 \sin\theta_{04l} \cos\theta_{123l} \\ z_{4r} = z_{5r} + d_6 \cos\theta_{04l} \cos\theta_{123l} \\ z_{3r} = z_{4r} + d_3 \sin\theta_{04l4r} \cos\theta_{123l} \\ z_{2r} = z_{3r} - a_{3r} \cos\theta_{04l4r} \cos\theta_{123l3r} \\ z_{1r} = z_{2r} - a_{2r} \cos\theta_{04l4r} \cos\theta_{123l32r} \\ z_{0r} = z_{1r} - a_{1r} \cos\theta_{04l4r} \cos\theta_{123l321r} \\ z_{fr} = z_{0r} - a_{0r} \cos\theta_{04l40r} \cos\theta_{123l321r} \end{cases} \quad (40)$$

The same equations can be derived for the reference system based on the left foot depicted in Figure 33.

4.3. Gait stabilization

Concerning the balance and the stabilization of the gait of a humanoid robot, it is possible to introduce some useful definitions. First, it is possible to define the support polygon as the region formed by enclosing all the contact points between the robot and the ground by using an elastic cord braid (Kajita et.al., 2014). The support polygon is showed in Figure 35.

Another important concept, regarding the balance and the stability of the human gait, is the Zero Moment Point (ZMP). As showed in Figure 36, the ZMP is the point on the ground plane in which the sums of all the ground reaction forces are zero. The ZMP always exists inside the support polygon, whereas the vertical projection of the COM on the ground floor can exist outside of the support polygon.

As showed in Figure 37, for a standing human, the COM projection on the ground floor is inside the support polygon and it is coincident with the ZMP point. In a more dynamic situation, instead, the COM may fall outside while the ZMP is always inside the support polygon.

There are two approaches for the realization of a stable human-like walk, static walking and dynamic walking. The static walking assumes that, if the movements of the joints are slow enough, they will

not influence the stability of the system. The movements are therefore planned considering the low speed constraint. Furthermore, the balance of the robot is guaranteed by the movement of the hip. The ankle joint is rotated in order to have the projection of the COM of the robot on the ground plane inside the support polygon.

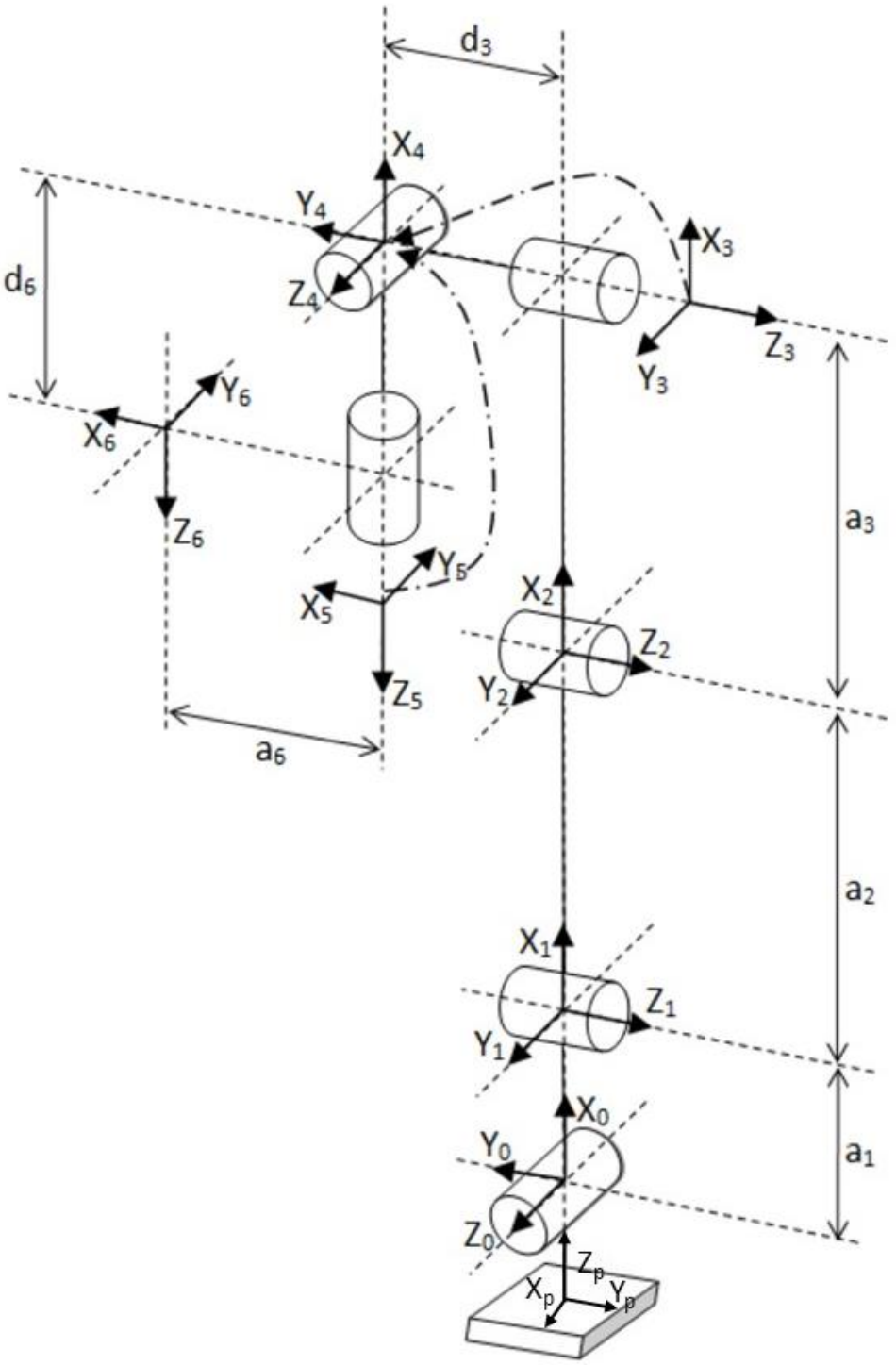

Figure 33: kinematic chain for the left leg of Archie. Adapted from (Daniali, 2013)

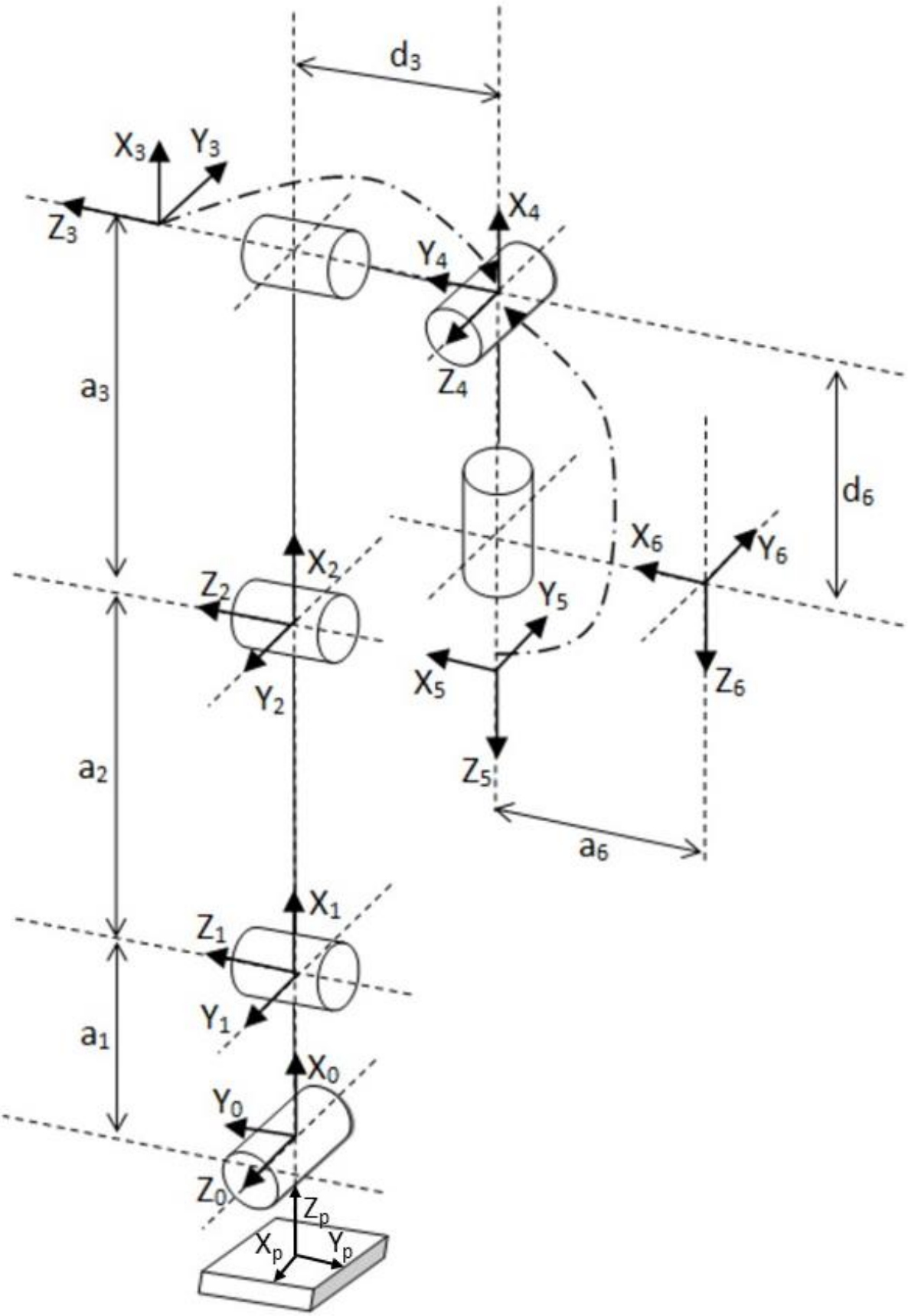

Figure 34: Kinematic chain of the right leg. Adapted from (Daniali et.al., 2013)

In the dynamic walking instead, the movement of the swing leg is not planned but computed online during the gait considering the data input from various kind of sensors in order to keep the balance and the stability of the system. This last approach guarantees the balance and the stability even in presence of external disturbances like for instance floor unevenness.

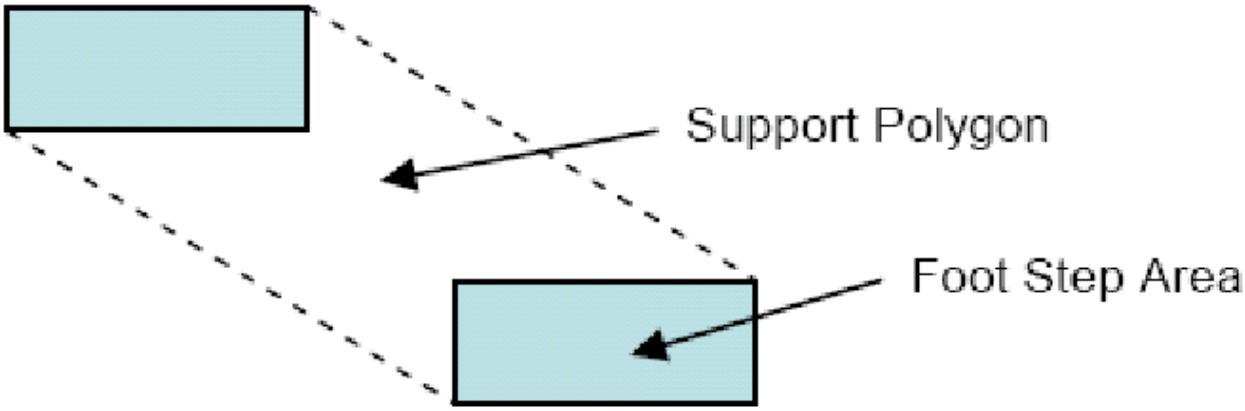

Figure 35: support polygon for a humanoid robot (Vadakkepat, P. and D. Goswami, 2008)

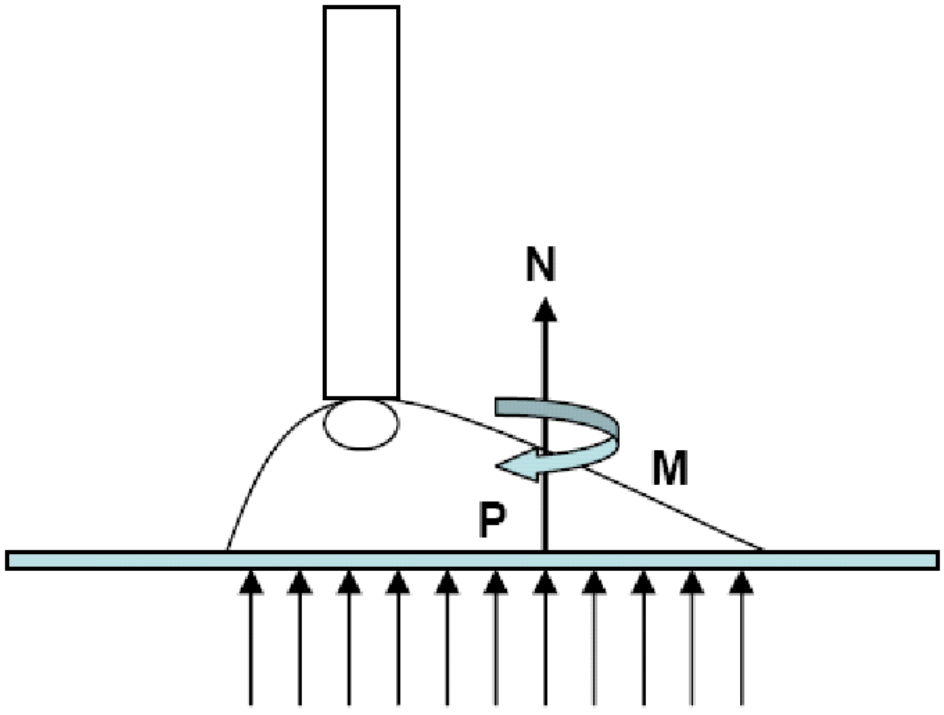

Figure 36:ZMP and ground reaction forces (Vadakkepat, P. and D. Goswami, 2008)

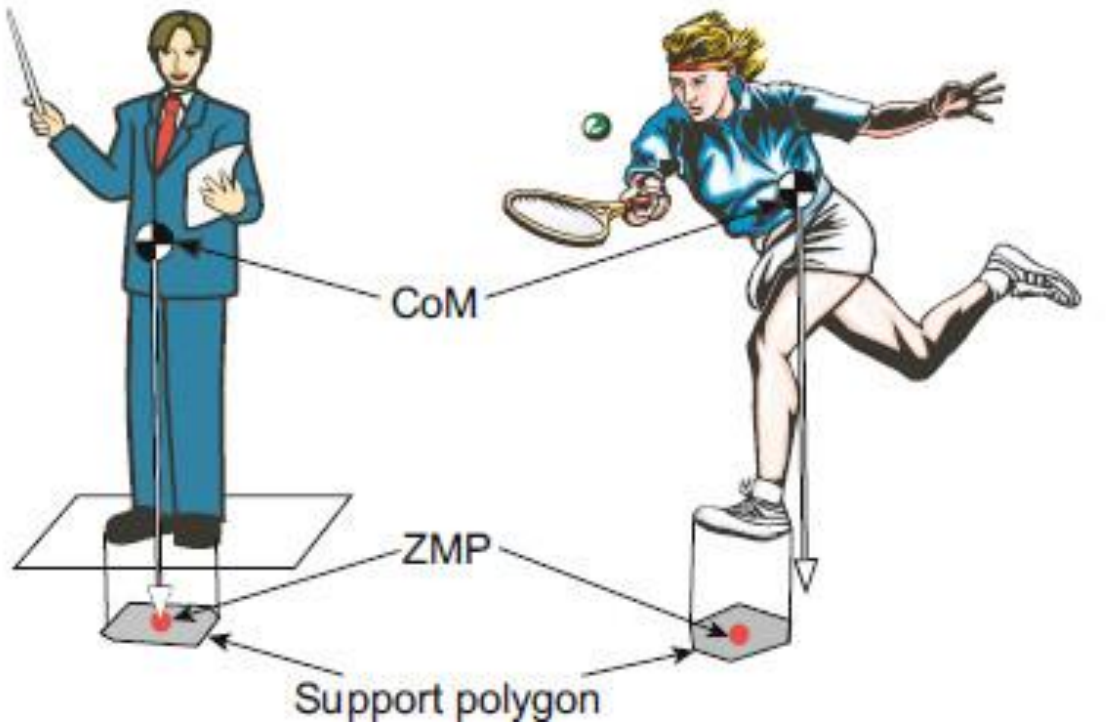

Figure 37: ZMP and support polygon (Kajita et.al., 2014)

The balance of a humanoid robot in the dynamic walking is controlled by means of a stabilizer. The basic functionality of the stabilization is the correction of small errors around the previously computed walking pattern.

One of the most successful stabilization methods is the control of the Zero Moment Point (ZMP). The position of the ZMP can be computed by the readings of various sensors, such as force sensors placed on the feet or by the acceleration of the COM of the robot. The control of the ZMP is based on the cart table model depicted in Figure 38.

The cart table model consists in a massless table on which a cart is placed. All the mass of the robot and, consequently, the COM is concentrated on the cart. The movement of the cart causes the centre of pressure of the table on the floor, that is, the ZMP, to translate along the base of the table.

It is also possible to stabilize the walk by controlling the ankle torque. Nevertheless, this solution is the most difficult to implement since the torque control on mechanism equipped with high reduction gear is an extremely difficult problem (Kajita et.al., 2014).

Another method consists in keeping the body of the robot in an upright position by controlling the hip joint during the walk. The attitude of the hip and of the body can be controlled with the use of attitude sensor such as an IMU, for instance.

This last approach is the one used by (Hashimoto et.al., 2014). The balance control they developed is based on the readings of an IMU attached to the robot trunk. The foot placement of the robot is then modified according to an attitude angle measured by the IMU as showed in Figure 39.

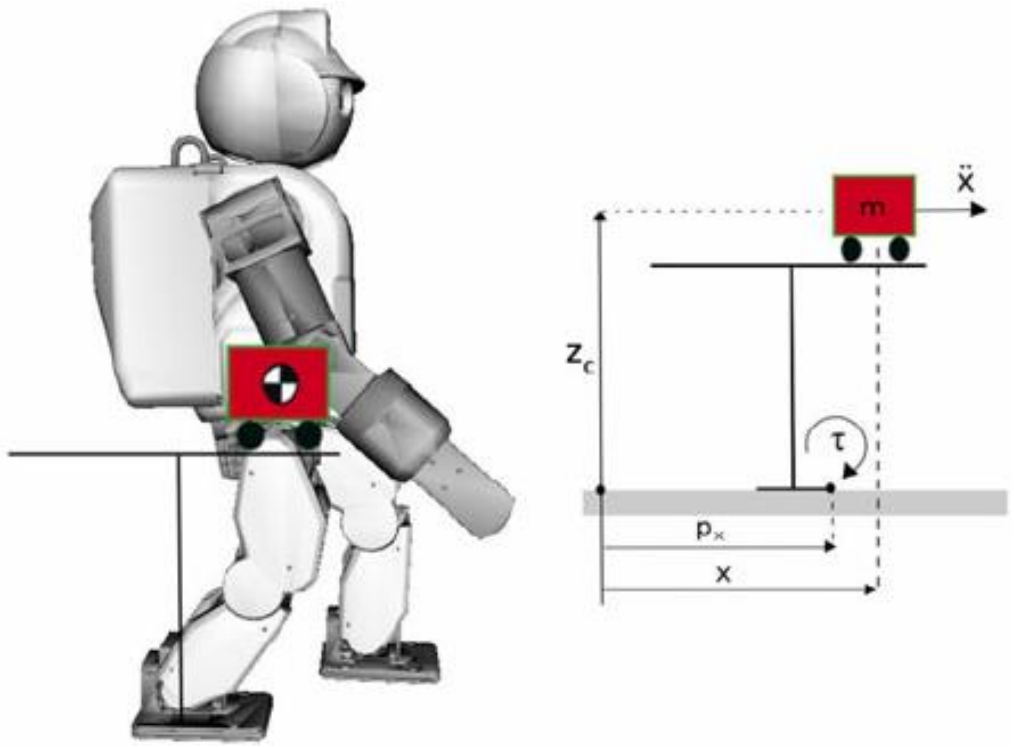

Figure 38: cart table model (González-Fierro et.al., 2015)

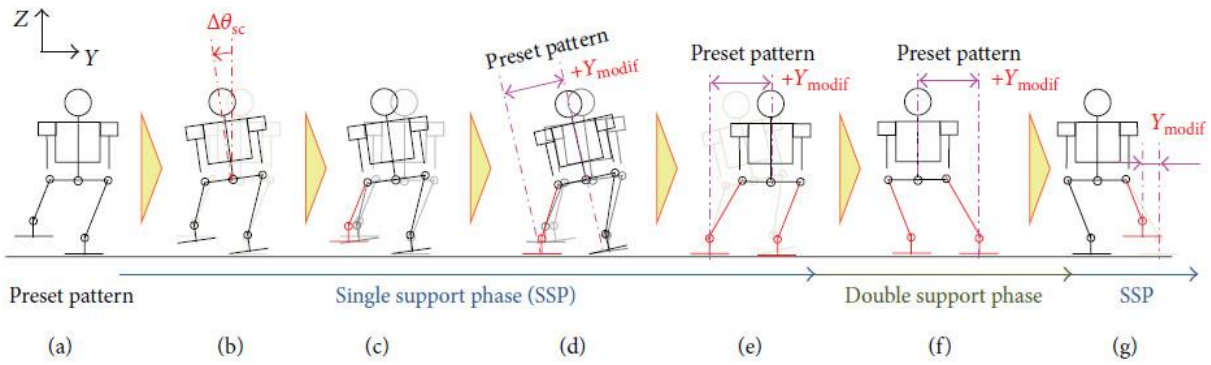

Figure 39: stabilization of a humanoid robot gait by measurements of an attitude angle (Hashimoto et.al., 2014)

To be more specific, the attitude of the swing foot is modified on the roll angle to make it touch the ground horizontally. Furthermore, its landing point is translated laterally to prevent the robot to fall on the outside. This last step is necessary because this control has been implemented for the robot WABIAN-2R which has narrow feet. The trajectory of the legs is then returned to the pre-set pattern and the waist is translated laterally in order to begin the next swing phase. The angle and landing point modifications are done using simple geometrical consideration and the new trajectories are then computed using interpolating polynomials.

Another simple geometrical algorithm for the balance of the robot based on the measurements of an angle is the one developed in (Park et.al., 2014). Their balance control takes in input the data from an inclinometer placed on the back of the right foot. The balance control they developed was able to make the humanoid robot GHR on an inclining surf board which was moving simulating the wave of the ocean. GHR is a simple humanoid robot platform composed by two legs and a hip developed at the Griffith University and showed in Figure 40.

In order to stay in balance on a surf board, one leg needs to be stretched out and the other needs to be bent. Basing on this simple assumption, (Park et.al., 2014) developed a balance control using simple geometrical relations between the two legs. The experiments of this method showed in Figure 41 were successful.

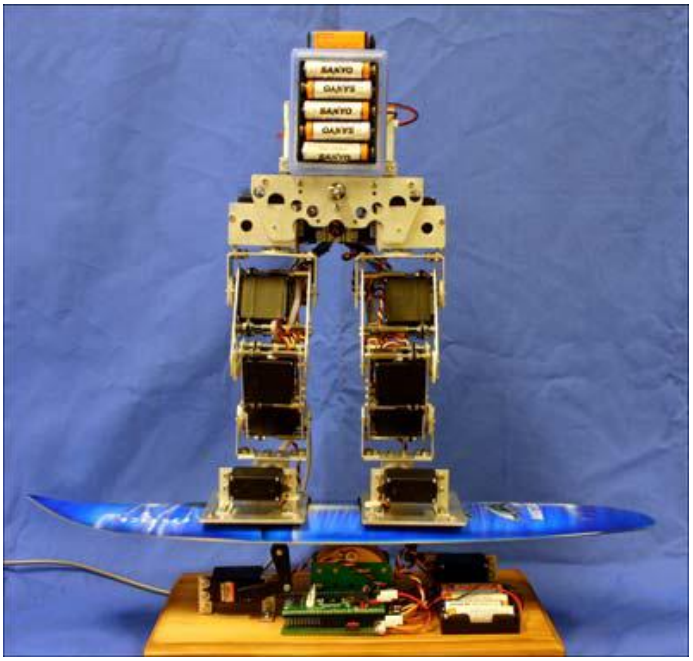

Figure 40: The humanoid robot GHR on the surf board used for the experiments (Park et.al., 2014)

Figure 41: The humanoid robot GHR during the experiments (Park et.al., 2014)

4.4. Advanced gait planning and control algorithm

The study from (Hashimoto et.al., 2014) and (Park et.al., 2014) presented in the last chapter showed that it is possible to control and balance the robot based only on the attitude of the COM. The approach currently in evaluation for Archie is similar to the one proposed by (Hashimoto et.al., 2014) as it uses attitude variations of the hip from a pre-set pattern in order to stabilize the robot. This pre-set pattern for the gait of the humanoid robot is computed with a method defined during the course of this PhD work which, similarly as in the work of (Park et.al., 2014), links between each other the movement of the joints during the gait by means of geometrical relationships.

Let us consider the support foot to be the right one and the swing foot to be the left one. Consider the reference system to be on the right foot as showed in Figure 34, and that the COM of the robot moves of a Δy in order to move towards the support foot and of a Δx in order to advance.

On the frontal plane, as showed in Figure 42, in order to keep the trunk in an upright position, the angle of the joint {4l} is set to be always equal to {0l} with a changed sign. On the sagittal plane instead, as showed in Figure 43, the joint {3r} angle is set to be always equal to the joint {1r} with the sign changed in order to keep the trunk in an upright position. Supposing also that the angle of the joint {2r} is 0 during the gait one can relate directly the movement of the COM to the movement of the joint {0r} and {1r}. As a matter of facts, substituting the joint angle in the (38) one obtains the equation system:

$$\begin{cases} y_{COM} = (a_{1r} + (a_{2r} + a_{3r})\cos\theta_{1r})\sin\theta_{0r} + d_3\cos\theta_{or} + \frac{a_6}{2} \\ x_{COM} = (a_{2r} + a_{3r})\sin\theta_{1r} \end{cases} \quad (41)$$

Whose solution leads to:

$$\begin{cases} \theta_{0r} = \tan^{-1}\left(2\frac{2k_1 \pm \sqrt{4k_1^2 - 4k_2k_3}}{2k_2}\right) \\ \theta_{1r} = \sin^{-1}\left(\frac{x_{COM}}{a_{2r}+a_{3r}}\right) \end{cases} \tag{42}$$

With

$$\begin{aligned} k_1 &= a_{1r} + (a_{2r} + a_{3r})\cos\theta_{1r} \\ k_2 &= \frac{a_6}{2} - d_3 - y_{COM} \\ k_3 &= d_3 + \frac{a_6}{2} - y_{COM} \end{aligned} \tag{43}$$

Consider now the joint of the swing leg. The angle of the knee joint of the swing leg {2l} and {3l} are set in order to modify the length of the projection on the yz plane of the swing leg of a $\tan 2\theta_{0r}$ factor and to guarantee its advancement.

Consider for clarity Figure 44. The relation for the computation of the length of projection of the swing leg on the yz plane in the reference system based on the right foot is the (44).

$$l_s = z_{3r} - \Delta h \tag{44}$$

Where

$$\Delta h = y_{3l}\tan 2\theta_{0l} \tag{45}$$

The joint angle θ_{3l}, θ_{2l} and θ_{1l} can be computed using the cosine law as showed in (46), (47), (48) and (49):

$$h_s = l_s - a_{1l} - a_{0l} \tag{46}$$

$$r = \sqrt{\Delta x^2 + h_s^2} \tag{47}$$

$$\theta_{2l} = \pi - \cos^{-1}\left(\frac{a_{3l}^2 + a_{2l}^2 - r^2}{2a_{3l}a_{2l}}\right) \tag{48}$$

$$\theta_{3l} = \cos^{-1}\left(\frac{a_{3l}^2 + r^2 - a_{2l}^2}{2a_{3l}r}\right) + \beta \tag{49}$$

With

$$\beta = \tan^{-1}\left(\frac{\Delta x}{h_s}\right) \tag{50}$$

If one assumes that the foot is always parallel to the ground, then:

$$\theta_{3l} + \theta_{2l} + \theta_{1l} = 0 \tag{51}$$

From which:

$$\theta_{1l} = -(\theta_{3l} + \theta_{2l}) \tag{52}$$

All the other joint angles are set to be 0 during the full gait. To resume the geometrical relations that links all the joints of the leg with each other are for left (53) and right (54) leg:

$$\begin{cases} \theta_{1r} = \sin^{-1}\left(\frac{x_{COM}}{a_{2r}+a_{3r}}\right) \\ \theta_{0r} = \tan^{-1}\left(2\frac{2k_1 \pm \sqrt{4k_1^2-4k_2k_3}}{2k_2}\right) \\ \theta_{2r} = 0 \\ \theta_{3r} = -\theta_{1r} \\ \theta_{4r} = -\theta_{0r} \\ \theta_{5r} = 0 \\ \theta_{5l} = 0 \\ \theta_{4l} = 0 \\ \theta_{3l} = \cos^{-1}\left(\frac{a_{3l}^2+r^2-a_{2l}^2}{2a_{3l}r}\right) + \beta \\ \theta_{2l} = \pi - \cos^{-1}\left(\frac{a_{3l}^2+a_{2l}^2-r^2}{2a_{3l}a_{2l}}\right) \\ \theta_{1l} = -(\theta_{3l}+\theta_{2l}) \\ \theta_{0l} = 0 \end{cases} \quad (53)$$

$$\begin{cases} \theta_{1l} = \sin^{-1}\left(\frac{x_{COM}}{a_{2r}+a_{3r}}\right) \\ \theta_{0l} = \tan^{-1}\left(2\frac{2k_1 \pm \sqrt{4k_1^2-4k_2k_3}}{2k_2}\right) \\ \theta_{2l} = 0 \\ \theta_{3l} = -\theta_{1r} \\ \theta_{4l} = -\theta_{0r} \\ \theta_{5l} = 0 \\ \theta_{5r} = 0 \\ \theta_{4r} = 0 \\ \theta_{3r} = \cos^{-1}\left(\frac{a_{3l}^2+r^2-a_{2l}^2}{2a_{3l}r}\right) + \beta \\ \theta_{2r} = \pi - \cos^{-1}\left(\frac{a_{3l}^2+a_{2l}^2-r^2}{2a_{3l}a_{2l}}\right) \\ \theta_{1r} = -(\theta_{3l}+\theta_{2l}) \\ \theta_{0r} = 0 \end{cases} \quad (54)$$

If the support leg is the right one. With the geometrical relation showed in (53) and (54) between the joints of the legs it is possible to relate the movements of the COM and of all the joints of the legs during the gait just to the movements of the joint {0} and {1} of the support foot. Figure 45 and Figure 46 shows the movements of the legs during the gait computed with the method described above.

As mentioned before, the planning of the joint movements during the gait was developed at first as a part of the footstep planner. Besides the already described approach, another approach was also evaluated.

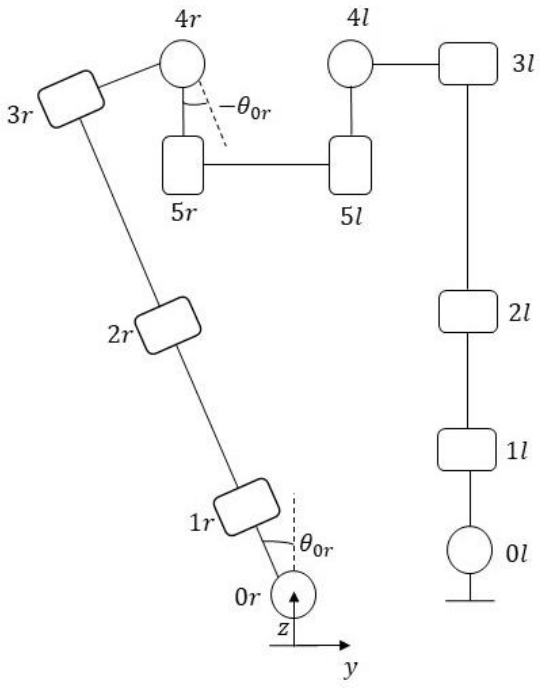

Figure 42:Symmetries in Archie's movement on the frontal plane

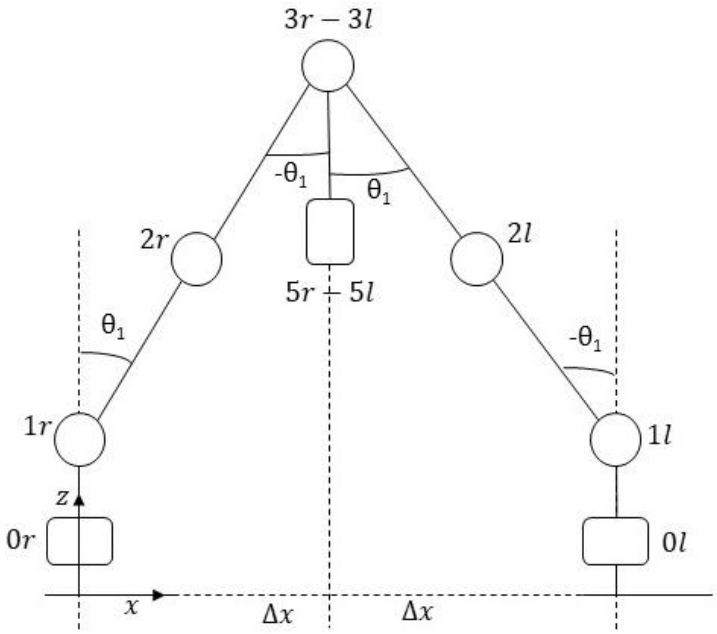

Figure 43: Symmetries in Archie's movements on the sagittal plane

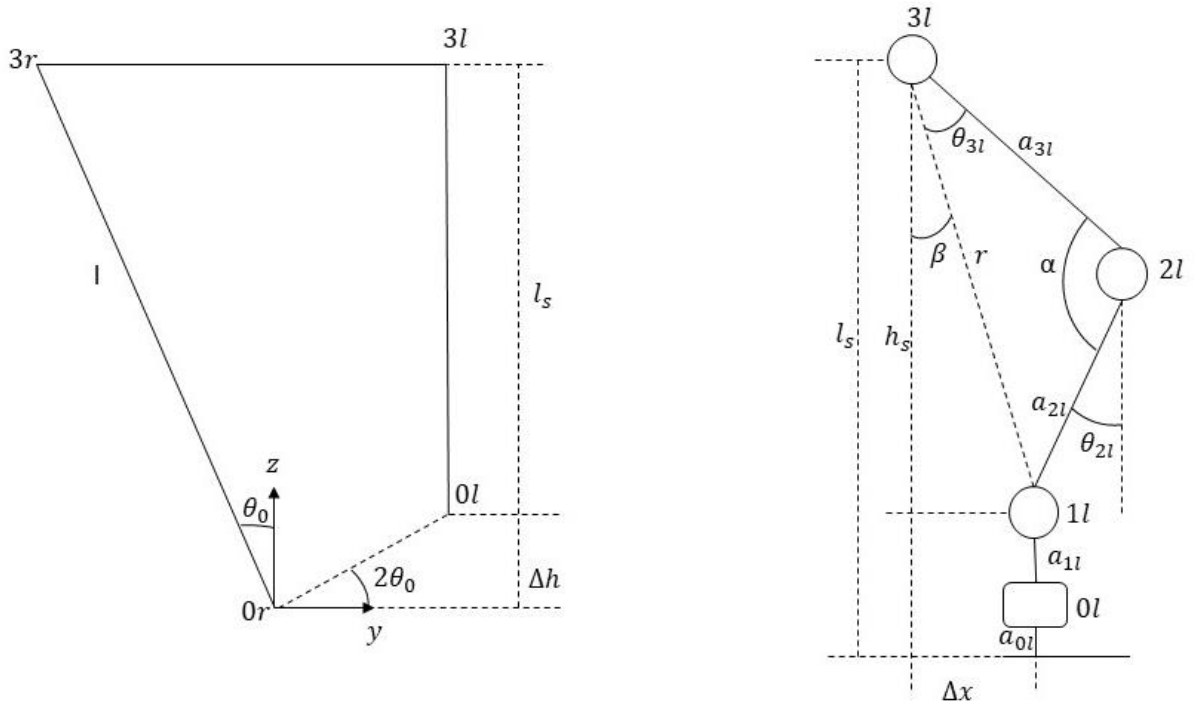

Figure 44: Movement of the swing leg during the gait

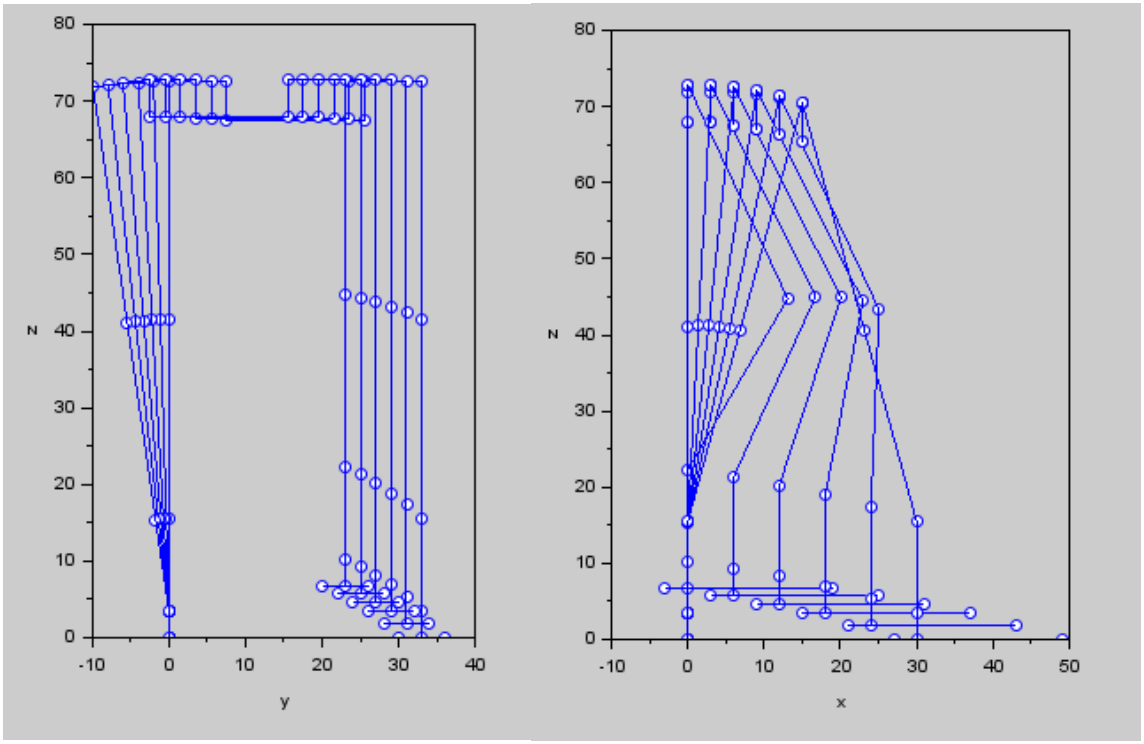

Figure 45: Movements planned on the frontal and sagittal plane

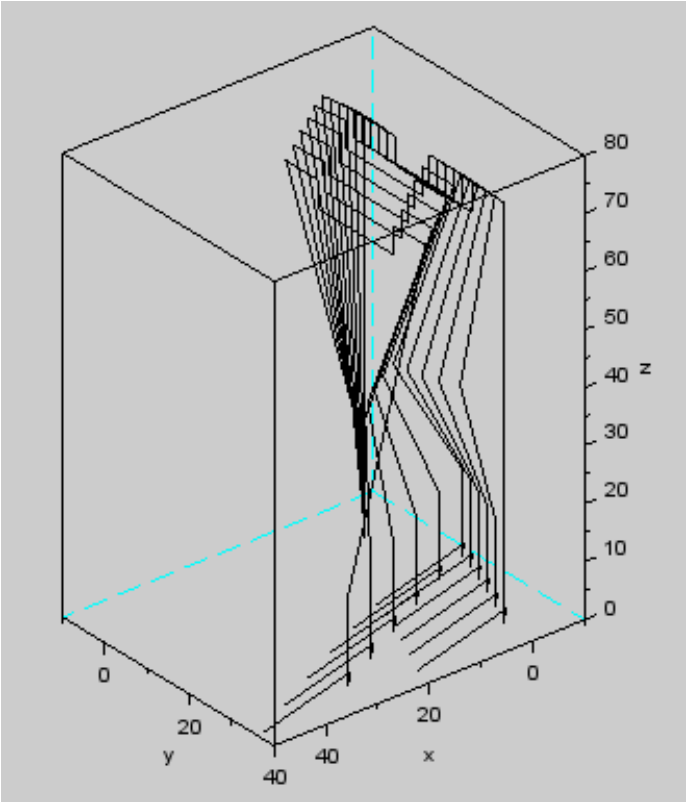

Figure 46: 3D view of the planned movements.

The other tested approach consisted on the artificial potential field approach applied to a virtual manipulator which could approximate Archie's leg and hip. As mentioned in chapter 2, the artificial potential field algorithm consists in the creation of two artificial potential fields, one for the obstacle and one for the goal. Then, by computing the anti-gradient of the total artificial potential field, it is possible to compute a free collision path for the robot.

Denoting as U_a the attractive potential (associated to the goal), U_r the repulsive potential (associated to the obstacles) and with U_t the total potential field, one can compute the gradients of the repulsive potential field and the gradient of the attractive potential field as:

$$\frac{\partial U_r}{\partial q} = \left(\frac{\partial U_r}{\partial x}, \frac{\partial U_r}{\partial y}, \frac{\partial U_r}{\partial z}\right) \tag{55}$$

$$\frac{\partial U_t}{\partial q} = \left(\frac{\partial U_t}{\partial x}, \frac{\partial U_t}{\partial y}, \frac{\partial U_t}{\partial z}\right) \tag{56}$$

In this formulation, only the end effector is subject to the potential of the goal. As a matter of fact, the end effector is the only part of the leg that has to reach the destination while the links just have to stay away from the obstacles.

The virtual manipulator approximating Archie's leg has been defined as follows. The supporting foot can be considered as the base of the manipulator whose position is fixed in time. The foot of the swing leg instead, can be regarded as the end effector. By assuming the hip always perpendicular to the legs, it is possible to considered it dot-like and as a rotational joint. Furthermore, by using a numerical notation for the joints, one will have that the point {0} is the point on the base, point {1} will be on the first link (the one that is closer to the base) and so forth until the point P that is the end effector. This virtual manipulator is presented in Figure 47.

The algorithm is basically a loop which, on every iteration, computes a new point of the path that must be followed by the end effector and the joints. The velocity of the end effector and the joint is computed by means of the:

$$\dot{q} = -\sum_{i=1}^{P-1} J_i^T(q)\nabla U_r(p_i) - J_P^T(q)\nabla U_t(p_P) \tag{57}$$

Where $\mathrm{J_i^T}$ is the Jacobian of the direct kinematics relative to the point i.

Using this velocity, it is possible to compute with the Euler method, the next point of the planned trajectory:

$$q_{k+1} = q_k + T\dot{q_k} \tag{58}$$

Where T is the integration step. This algorithm has been tested with various simulations and the results showed that the algorithm had to be improved.

Figure 48 shows the results of the first simulation. The red rectangle is the obstacle Archie must step over and the lines of different colours are the links composing Archie's legs. Blue and purple are the links from the foot to the ankle of, respectively, the supporting and the swing leg. Red and green are the links from the ankle to the knee of, respectively, the supporting and swing leg. Black and

Yellow lines are the link between the knee and the hip of, respectively, the supporting and swing leg.

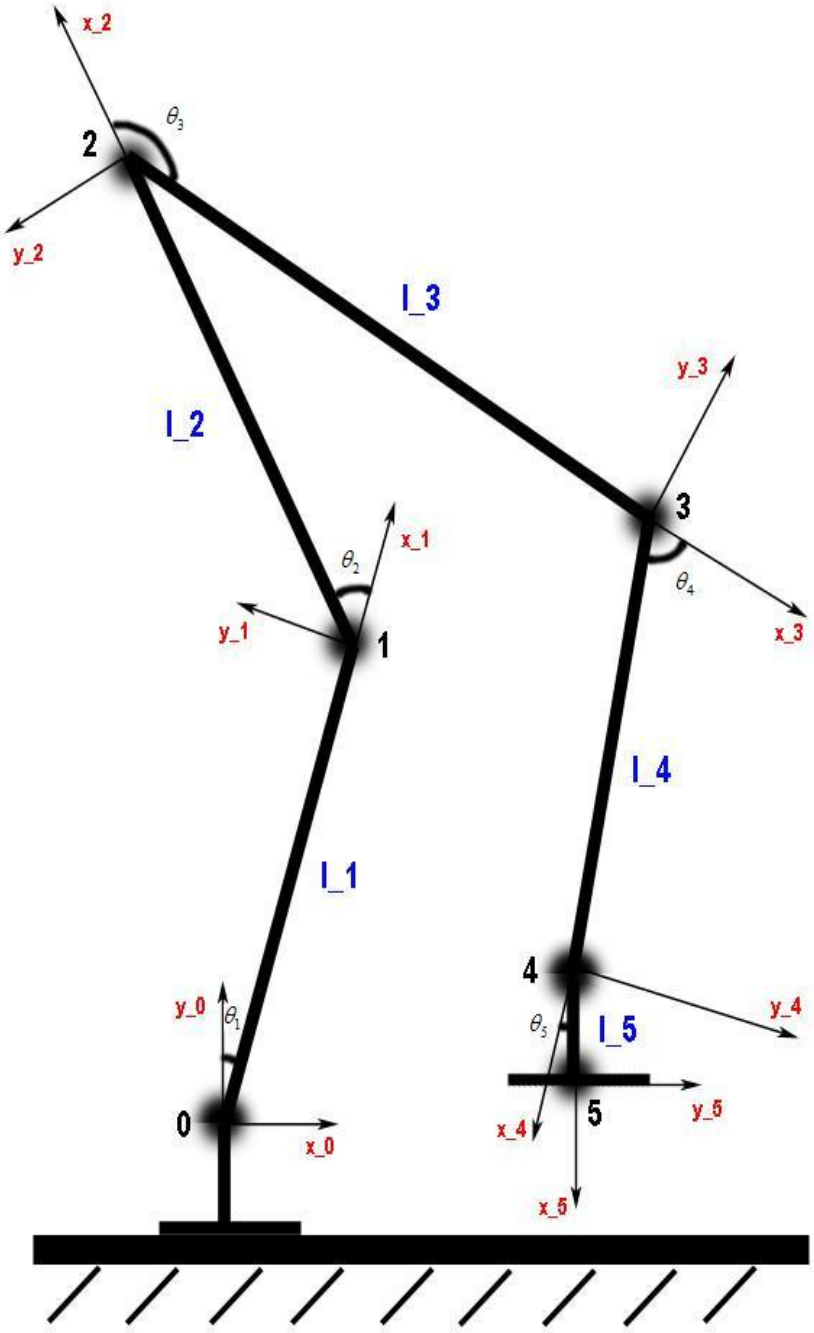

Figure 47: First model of Archie's leg for the simulations

As it is possible to see in Figure 48, the movement of Archie's legs are unnatural and, since it bends backwards to cross the obstacle, it could fall.

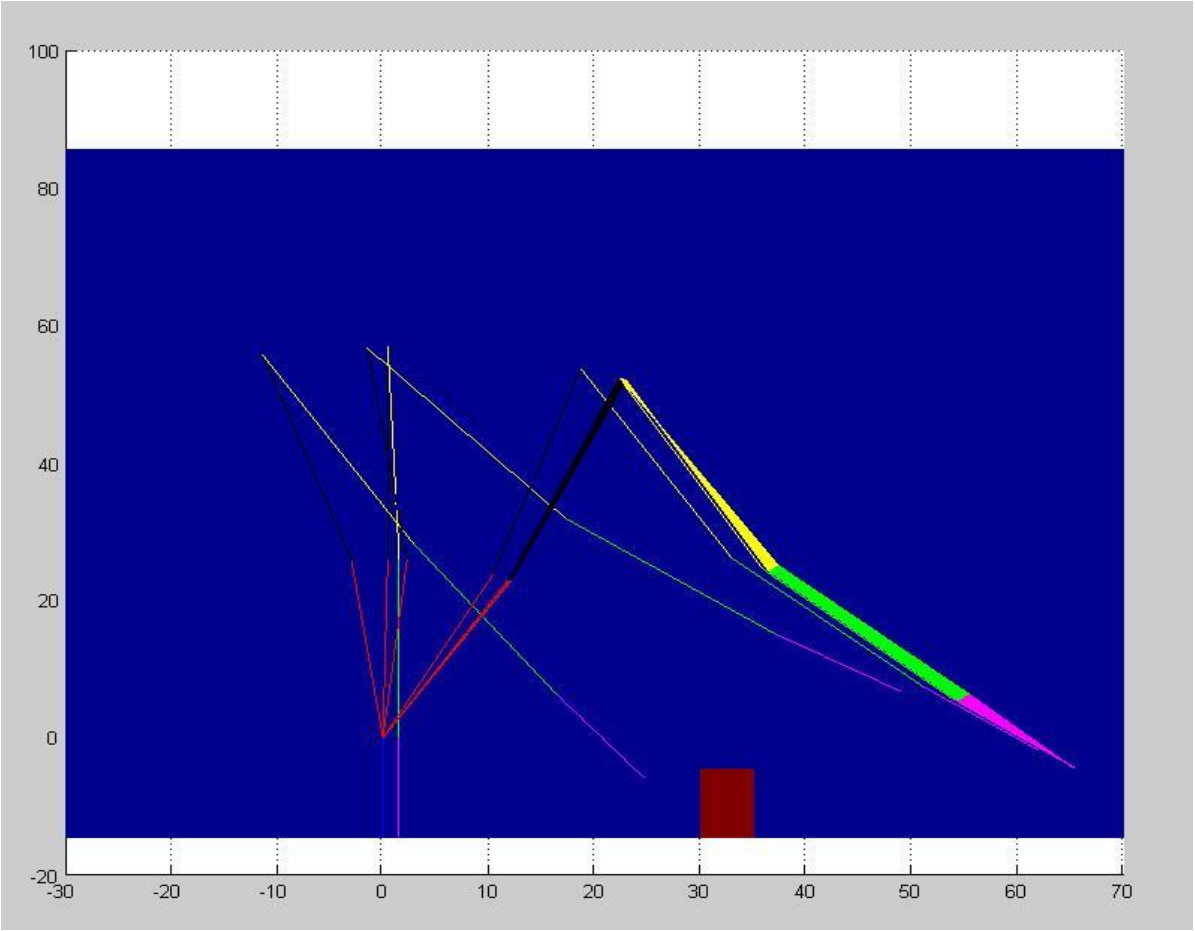

Figure 48: Movement of archie's leg during the tests of the algorithm

The algorithm needed some limitations to the possible movement of the leg. This can be realized by defining a different kind of artificial potential field or by defining a different virtual manipulator. For this reason, a second representation of Archie's leg for a second test of the algorithm has been defined. In this representation the fixed reference system is on the hip which is approximated as a prismatic joint. Only the swing leg is considered. The new model of Archie's leg is depicted in Figure 49.

A further difference between the first and the second formulation relies in the resolution method. The algorithm, in this second formulation, computed the movement just of the end effector and then, computed the movement of the other joints using inverse kinematics.

The reason for this change lies in the fact that the artificial potential field algorithm assures to find always a safe path. However, it does not assure that the found path is the optimal and, above all, that the safe path computed is suitable for a humanoid robot legs, which movements are restricted by balance limitation.

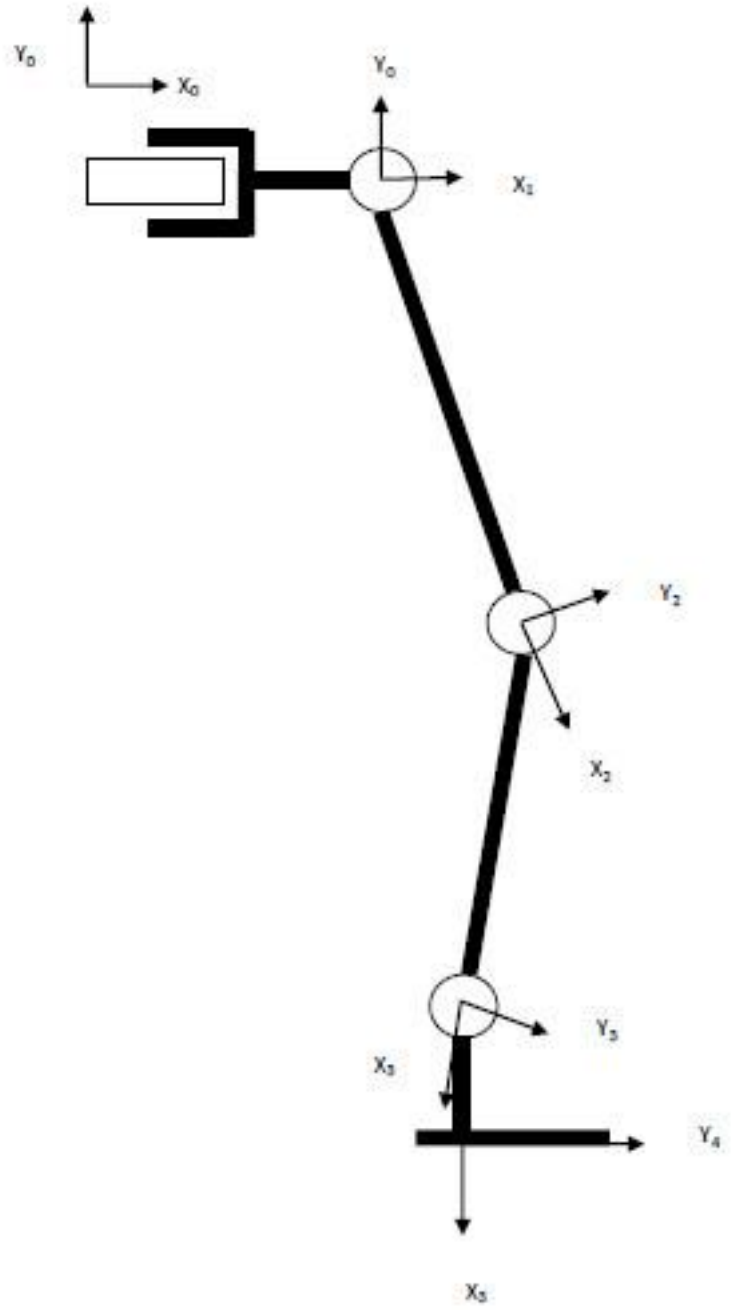

Figure 49: Second considered model of Archie's leg for the simulations

Computing the movement of the end effector with the artificial potential field algorithm assures finding a path and the inverse kinematics allows us to be in accordance with the limitation of the movement of Archie's leg.

To summarize, the algorithm computes the trajectory of the end effector with the (59).

$$q_{k+1} = q_k - T\nabla U_t \tag{59}$$

Where T is the integration step, ∇U_t is the gradient of the total potential field and $q_i = \begin{bmatrix} x_4 \\ y_4 \end{bmatrix}$, the vector containing the positions of the end effector.

The potential functions are the same as in the first formulation:

$$U_a = K_a \left\| q - q_g \right\|^2 \tag{60}$$

$$U_r = \begin{cases} \frac{K_r}{\gamma}\left(\frac{1}{\mu_i(q)} - \frac{1}{\rho}\right)^{\gamma} & if\ \ \mu_i(q) \leq\ \rho \\ 0 & if\ \ \mu_i(q)\ \geq\ \rho \end{cases} \tag{61}$$

Where K_r and K_a are empirical constants. $\mu_i(q)$ is the minimum distance between the end effector and the boundaries of the obstacle and ρ is the distance at which one wants that the end effector *"feel"* the potential of the obstacle. Finally, γ defines the decrease in the influence of the obstacle potential with the distance.

After the computation of the new position of the end effector, one can compute the new position of the other joints with inverse kinematics.

The equations for the joints of Archie's legs are:

$$x: \begin{cases} x_0 = 0 \\ x_1 = x_0 + a_1 = a_1 \\ x_2 = a_1 + l_2 \cos\theta_2 \\ x_3 = a_1 + l_2 \cos\theta_2 + l_3 \cos\theta_{23} \\ x_4 = a_1 + l_2 \cos\theta_2 + l_3 \cos\theta_{23} + l_4 \cos\theta_{234} \end{cases} \tag{62}$$

$$y: \begin{cases} y_0 = 0 \\ y_1 = 0 \\ y_2 = l_2 \sin\theta_2 \\ y_3 = l_2 \sin\theta_2 + l_3 \sin\theta_{23} \\ y_4 = l_2 \sin\theta_2 + l_3 \sin\theta_{23} + l_4 \sin\theta_{234} \end{cases} \tag{63}$$

The limitation on Archie's legs are:

$$\begin{array}{l} \theta_{234} = -90° \quad \rightarrow \quad \sin\theta_{234} = -1, \;\; \cos\theta_{234} = 0 \\ \theta_4 = 0° \quad \rightarrow \quad \theta_{234} = \theta_{23} \quad \rightarrow \quad \sin\theta_{23} = -1, \;\; \cos\theta_{23} = 0 \end{array} \tag{64}$$

In order to keep Archie's foot always parallel to the ground. Substituting this relation in the equations above one have:

$$y_4 = x_1 + l_2 \sin\theta_2 + l_3 \sin\theta_{23} - l_4 = y_3 - l_4 \quad \rightarrow \quad y_3 = y_4 + l_4 \tag{65}$$

$$y_3 = l_2 \sin\theta_2 - l_3 \quad \rightarrow \quad \theta_2 = \sin^{-1}\frac{y_3 + l_3}{l_2} \tag{66}$$

$$y_2 = l_2 \sin\theta_2 \tag{67}$$

$$x_4 = a_1 + l_2 \cos\theta_2 + l_3 \cos\theta_{23} = x_3 \quad \rightarrow \quad x_3 = x_4 \tag{68}$$

$$x_3 = a_1 + l_2 \cos\theta_2 = x_2 \quad \rightarrow \quad x_2 = x_3 \tag{69}$$

$$x_2 = a_1 + l_2 \cos\theta_2 \quad \rightarrow \quad a_1 = x_2 - l_2 \cos\theta_2 = x_1 \tag{70}$$

Simulations were performed in order to determine the performances of this approach. The results of this simulations are showed in Figure 50. As in Figure 48, blue is the environment in which Archie is moving and the red rectangle is the obstacle it must step over. Furthermore, as in Figure 48, every coloured line represents a segment of Archie's leg. Green is the foot, black is the ankle, pink is the part of the leg between the ankle and the knee and yellow is the segment between the knee and the hip. Finally, red represents the hip. In the second model, for the hip, a prismatic joint has been chosen. Therefore, the backward movement of the hip represents a backward rotation of Archie's hip to ensure the rise of the leg without touching the obstacle. The foot never touches the obstacle and the leg is able to cross the obstacle. Nevertheless, also this formulation of the artificial potential field algorithm had its limits and needed some optimizations.

The prismatic joint extension representing the rotation of Archie's hip had to be limited because, a full 180° rotation of the hip (which is however not possible), had to represent the maximum elongation of the prismatic joint.

From the computational point of view this second formulation was lighter and faster than the first one but there were still problems in cases of local minima whose resolution could be very computational expensive.

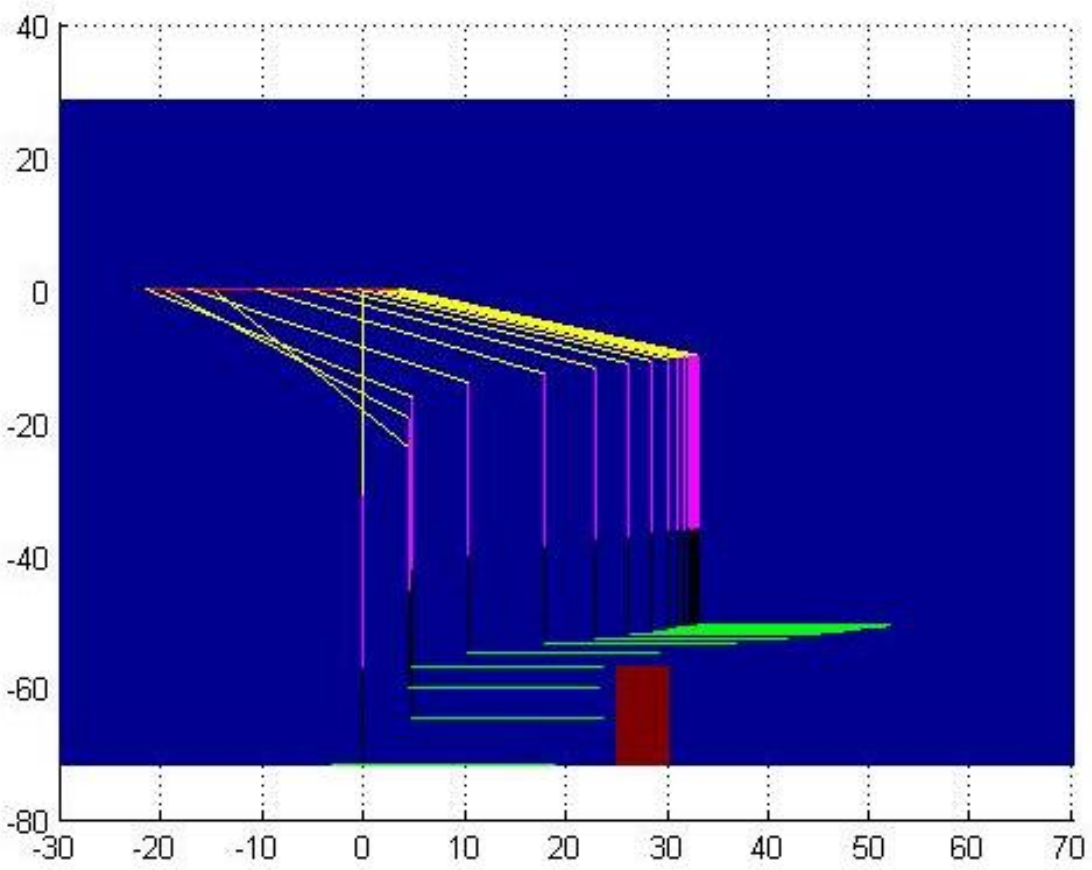

Figure 50: Simulation of the second considered model of Archie's leg

Due to all these implementation difficulties from the computational load point of view and from the robot stabilization point of view, it was then decided not to further improve the artificial potential field approach and to develop the geometric algorithm explained above.

As a matter of fact, from the computational point of view the geometric method has a very low computational load and it is much faster than the artificial potential field approaches. Furthermore,

the resulting movements of the legs' joints computed in this way should also guarantee a stable gait since the hip is always kept in an upright position. Furthermore, forward and backward movement of the hip corresponds to forward or backward movements of the swing leg which would produce a forward or backward movement of the swing leg in case the robot would fall forward or backward.

Due to the stability of the geometric method, it is possible to develop a stabilization strategy that would make every movement of the robot (described by the measured speed and acceleration of the COM and by the feedback from the motor controller) as close as possible to this pre-set movements pattern.

Figure 51 shows a scheme for the control approach proposed. The planning of the movement of the COM is first performed with the walking primitive generation. From the COM positions, the movement of the knee joint of the support leg are computed and consequently, using the (53) and (54), also the DH parameter of all the other joints of the support and swing leg.

Once the desired angles are computed, it is possible to compute the velocity input for the motor controller. The computation of these inputs can be performed using the relations of the position control loop showed in Figure 18. The gains can be chosen from the lookup table that resulted from the system identification performed in (Schoerghuber, 2014) introduced in the previous chapter.

The velocity commands are then sent to the embedded motor controllers which return the feedback velocity of the motors. The high-level control algorithm will then compute the current DH parameters of the legs and send them as input to the kinematical model of the robot together with the data from the IMU placed on the hip. The model will estimate the pose of the robot using the forward kinematics. The computed pose will be utilized by the stabilizer to compute the new desired angles considering the pre-set pattern previously calculated.

4.5. Extendibility

The stabilization approach explained in the previous chapter is good for the current configuration of the robot. As a matter of fact, one can assume the COM of the current configuration to be always on the hip.

In the future though, the upper body will be built and the movements of the other parts of Archie's body, especially the arms, will have an effect on the position of the COM of the robot. The COM will be thus not always on the hip, but its position can be different depending on the movement of all the links of the robot.

The stabilization approach proposed could also work in this case by integrating a more complex kinematical model of the robot. Such a model should be able to provide the transformation between the hip centre (formerly the COM of the lower body) and the actual COM of the full body. This transformation will be considered first in the walking primitive method by setting the correct z_c parameter in the (6) which is the height of the COM for the planning and then also in the stabilization.

First, the position of the COM with respect to the pelvis needs to be computed using the feedback of the motors and the IMU data. Then, it will then be given as input to the walking primitive method. Afterwards, the transformation between the COM and the hip will be used to determine the movement of the hip from the COM movement. Finally, the movement pattern can be set from the movement of the pelvis computed from the COM movements using the transformation between the COM and the pelvis. The motion and the stabilization will then follow this pre-set pelvis movement pattern. This approach is depicted in Figure 52

In addition to this more complex model of the robot it could also be possible to add force sensors under the robot's feet in order to measure directly the position of the robot's ZMP. From the planning of the position of the COM with the walking primitive method, it is also possible to compute a desired position of the ZMP and use this in order to control its position during the gait.

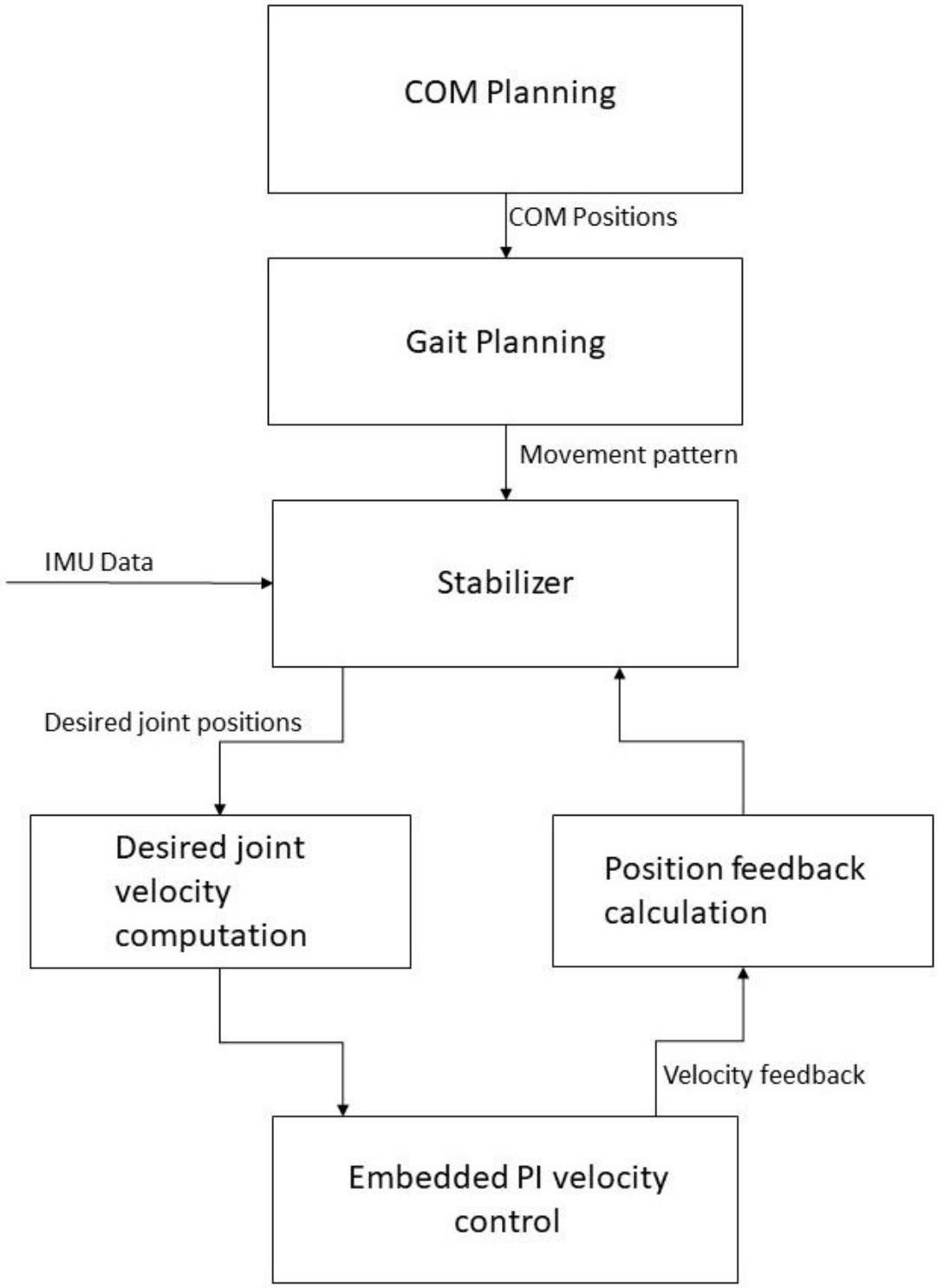

Figure 51: Advanced control algorithm lower body

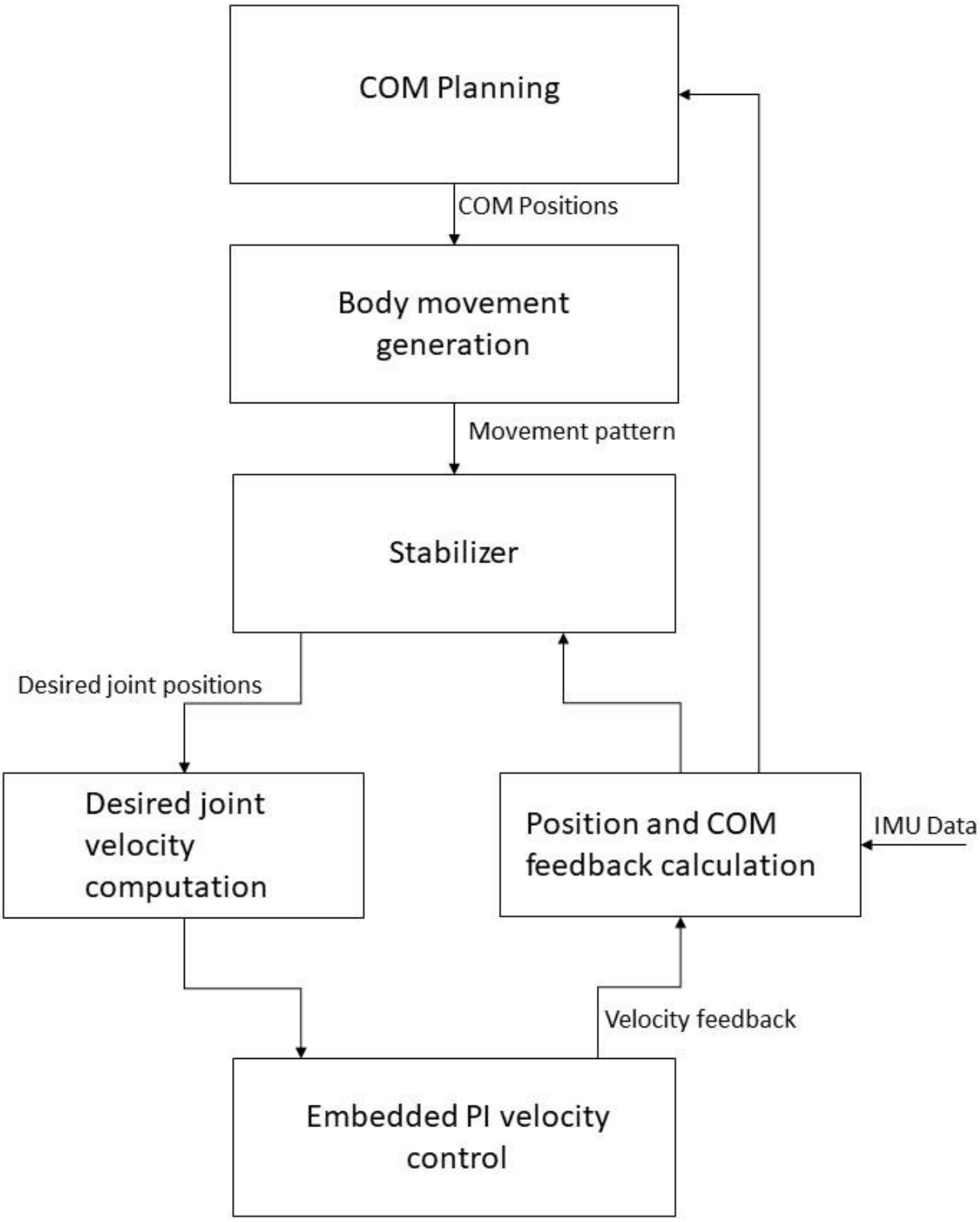

Figure 52: advanced control algorithm upper body

5. New Archie electronic configuration

The main purpose of this PhD work is to increase Archie's capabilities by giving it the possibility to sense the environment around it and to plan a collision free trajectory from a start position to a goal position.

Such a task requires a proper sensing hardware which the current design of the robot lacks. For this reason, this PhD work proposes the addition of stereo-cameras which will be used by the software for the modelling of the environment around it.

Furthermore, in order to increase the autonomy of the robot, an on-board computer was added. The on-board computer can be connected to an external computer through a wireless network dedicated to the robot. The wireless connection is provided by means of a router to which the board connects with the use of a USB Wi-Fi dongle. This design choice is requested from the high computational power needed by the software. Although the computational power of the on-board computer is high, it could be possible to move some of the computation or pieces of the software on other machines connected wireless with the on-board computer, in order to speed up the processing. The visualization for instance, can be executed on a remote laptop.

The control architecture of the robot is currently in replacement and a new design is also under evaluation. This new control architecture aims to allow the implementation of the new centralized advanced control algorithm. The new control hardware architecture is composed of:

- A motor driver, for the control of the motor.
- Arduino Due, for the execution of the embedded control algorithm
- Arduino Micro, for the connection between the motor driver and the Arduino Due
- IMU for measuring the attitude of the hip.

In addition, the on-board computer is thought to be part of the new control hardware architecture as the advanced gait and control algorithm will run on it.

This chapter will depict the new hardware chosen for Archie. First, the new on-board computer will be provided and then the stereo-cameras chosen will be described. Next, the USB Wi-Fi dongle and the router will be shown. The new control hardware architecture in development will then be

described in order to show how the onboard computer and the stereo-cameras are connected to the rest of the hardware of the robot.

5.1. Spinal board

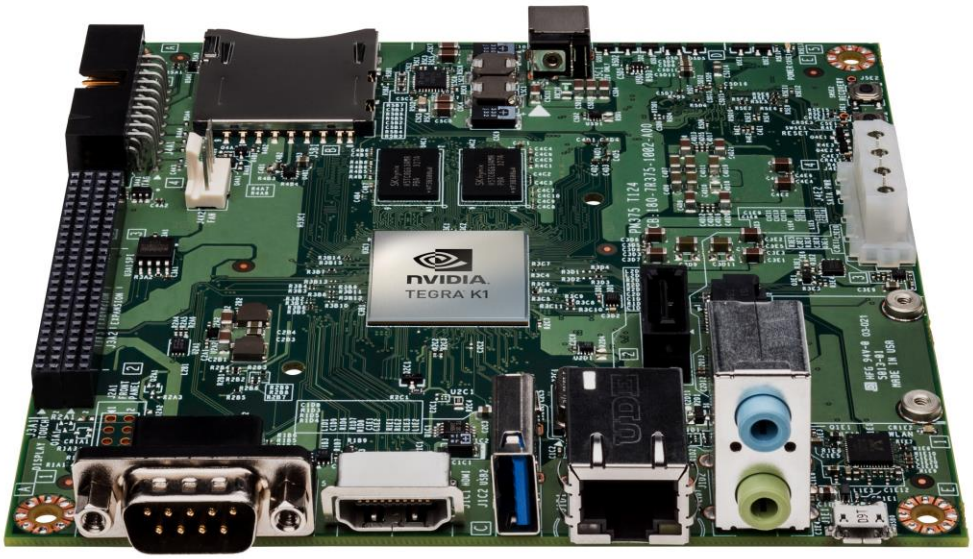

Figure 53: Spinal board. (NVidia, last retrieved 2015)

The spinal board has been chosen to be Archie's on-board computer. An important requirement for a humanoid robot's on-board computer is the high computational power. Many boards were evaluated focussing on their computational power with respect to their costs.

Our first evaluations were concerning low cost boards usually used by hobbyist such as the Raspberry Pi or the Olinuxino. The reason for these evaluations relies on the fact that such boards are easy to program, and they would have offered various possibilities for the implementation and testing of different algorithms.

Their low costs could have been perfect for a cost-oriented humanoid robot, but their computational power is limited. Both boards could have been interfaced with the stereo-cameras and all sensors. However, the computational power was still not enough for all algorithms that must run on the board. Furthermore, the Olinuxino does not support ROS which it was one of the requirements.

The choice made for the robot was the NVidia Jetson TK1 showed in Figure 53. It is a powerful board equipped with the Tegra GPU. The GPU is the Graphic Processing Unit and it is composed by several cores which can run operations in parallel resulting in a computational speed-up (d'Apolito et.al., 2016). This high computational power is used with the stereo cameras and the data handling algorithms. The CUDA libraries were created by the same producer of the board and with them, it is possible to parallelize every kind of computation and not also the image processing-based computations.

This board has the Ubuntu 14.04 installed on it and it is fully ROS compatible. Its computational power and the ROS compatibility made it the perfect candidate for our application.

Despite all these research applications performed with this board, to our knowledge, this is the first time that this board is used for a humanoid robot application.

The technical specifications of the on-board computer are shown in Table 5. More information on the board can be found at (NVidia, last retrieved 2015).

GPU	NVIDIA Kepler with 192 CUDA cores
CPU	Quad Core ARM® Cortex™-A15 NVIDIA 4-Plus-1™
DRAM	2 GB DDR3L 933 MHz EMC x16 using 64 bit data width
Storage	16GB eMMC 4.51
mini PCIe	1
USB 2.0	1
USB 3.0	1
HDMI	1
Serial Port RS232	1
Codec audio	1 Realtek ALC5639
LAN	GigE Realtek RTL8111GS
SATA	1

Table 5: NVIDIA Jetson TK1 technical specification (Nvidia, last retrieved 2015)

5.1.1. USB hub

Only one USB port is present on the board. Thus, a USB hub is attached to the board to provide all the USB port necessary. It provides a USB 3.0 for the connection to the stereo-cameras and 3 USB 2.0 port one of which is used for the USB Wi-Fi dongle. The USB hub is showed in Figure 54.

Figure 54: The USB hub

5.1.2. Wi-Fi Dongle

The Wi-Fi Dongle is used to provide the board with the possibility of connecting to a wireless network. Such an adapter is the EDIMAX EW-7811UN. It was chosen for its low price and known good performances with the Jetson TK1. It is able to provide a 150 Mbit/s stable connection and it supports WEP, WPA, WPA2 encryption and it is also WPS compatible.

The driver of the dongle for the Jetson TK1 works out-of-the-box and it is possible to download it at (ELinux Online Description Nvidia Jetson TK1, 2017). Figure 55 depicts the USB Wi-Fi dongle.

Figure 55: The Wi-Fi dongle (EDIMAX EW-7811UN Product Description, last retrieved 2018)

5.2. The router

The router chosen for our application is the TP-Link TL-WR841N. It is able to provide 300 Mbps and has a low price. More information can be found at (TP-Link TL-WR841N Online specification, last retrieved 2018). The router is depicted in Figure 56

Figure 56: The router (TP-Link TL-WR841N Online specification, last retrieved 2018)

5.3. Stereo-cameras

Different stereo-cameras were evaluated for the robot but in the end, due to their price and dimensions, the LI-USB30-V024STEREO from Leopard Imaging was chosen. The stereo-cameras are showed in Figure 57. Its 5 mm baseline does not guarantee a great accuracy in the reconstruction of the environment around the robot, but its dimensions are perfect for a teen-sized humanoid robot such as Archie.

The technical specifications of the cameras are listed in Table 6.

Figure 57: Stereo-cameras used for the robot realization. (Leopard Imaging, last retrieved 2015)

Video Resolution	640x480 60 fps
Sensitivity	4.8 V/lux-sec
Dynamic Range	8 bit
IR cut filter	Yes
Sensor Specification	Aptina MT9V024 Global Shutter WVGA Sensor
Shutter	Global
Format	1/3''
Resolution	H:752, V:480
Pixel size	H:6.0 um, V: 6.0 um
Color	Monochrome
Lens mount	M8 lens mount
M8 Lens	2.35mm, 170° DFOV
Interface	USB 3.0
Supply voltage	USB 3.0 +5 VDC power source
Current consumption	Approx. 270 mA at 5 VDC
Dimensions (with lens)	L: 80 mm, W: 15 mm, H: 17 mm
Mass	12 g

Table 6: Stereo-cameras technical specification (Leopard Imaging, last retrieved 2015)

5.4. Inertial Measurement Unit

The inertial measurement unit (IMU) is a combination of sensor composed from 3 accelerometers and 3 gyroscopes: The data coming from these sensors are then combined to obtain the attitude of the IMU alongside the acceleration and the angular velocities on the three axis. The IMU chosen for our application is the MPU 6050 showed in Figure 58.

It is possible to retrieve the sensor data using the I2C communication protocol. More information about the sensor can be found at (MPU 6050, last retrieved 2018). The values of the linear acceleration on the three cartesian axes and the three angular velocities measured respectively by the accelerometer and the gyroscopes are given as input to a complementary filter in order to compute the attitude of the sensor. More information can be found in (Kokaj, 2018).

Figure 58: The MPU 6050 IMU (MPU 6050, last retrieved 2018)

5.5. New joint controllers

The ELMO controllers will be replaced using a combination of 3 different electronic components: a motor driver, an Arduino Micro and an Arduino Due. The main reasons for this replacement are the following:

- The high price of the controller makes them unfit for a cost-oriented application such as Archie
- The high weight of the controllers makes the achievement of a human-like walking more difficult
- The impossibility of developing and testing of advanced control algorithm.

The next chapters will give an overview of the chosen electronic components followed by a description of the new hardware control architecture.

5.5.1. Motor driver

The motor driver chosen for the control of this motor is the Kimbrough BLD-70. It is a brushless DC motor driver of up to 70 W power. Figure 59 shows the motor driver chosen. The technical specifications of the component are listed in Table 7.

Driver Parameters	Minimum value	Rated value	Maximum value	unit
Output current	-	3	6	A
Input voltage	12	24	30	VDC
Hall drive current	-	20	-	mA
Hall signal voltage	4.5	6.25	6.5	V
Applicable motor speed	0	-	20000	RPM

Table 7: Motor driver specification (Motor Driver Online Shop page, 2018)

5.5.2. Arduino Due

The Arduino Due will be used for the low-level control algorithm. There will be one of them for every leg and will translate the command coming from the high-level control into a command understandable from the motor driver. It was chosen for being a cost-oriented and easy programmable solution. Figure 60 shows an image of the Arduino Due and Table 8 lists its technical specification. More information can be found at (Arduino Due, 2018).

Figure 59: The motor driver (Motor drive online shop page, last retrieved 2018)

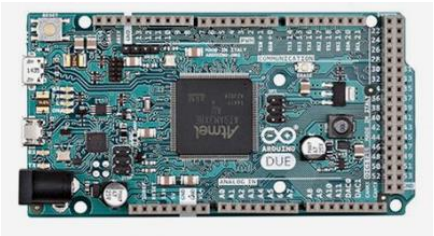

Figure 60: The Arduino Due (Arduino Due, last retrieved 2018)

Microcontroller	AT91SAM3X8E
Operating Voltage	3.3V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-16V
Digital I/O Pins	54 (of which 12 provide PWM output)
Analog Input Pins	12
Analog Output Pins	2 (DAC)
Total DC Output Current on all I/O lines	130 mA
DC Current for 3.3V Pin	800 mA
DC Current for 5V Pin	800 mA
Flash Memory	512 KB all available for the user applications
SRAM	96 KB (two banks: 64KB and 32KB)
Clock Speed	84 MHz
Length	101.52 mm
Width	53.3 mm
Weight	36 g

Table 8: Arduino Due specification (Arduino Due, last retrieved 2018)

5.5.3. Arduino Micro

The Arduino Micro is showed in Figure 61 and it will be used to connect the Arduino Due with the Arduino Micro.

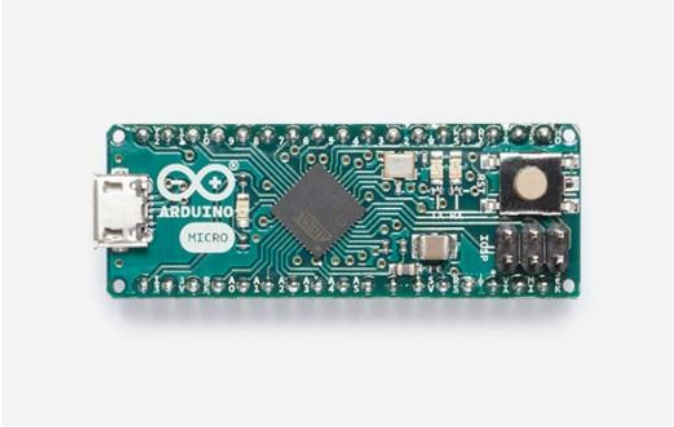

Figure 61: The Arduino Micro (Arduino Micro, last retrieved 2018)

The Arduino Micro is based on the ATmega32U4 processor and it is equipped with 20 input/output pin. 7 of these pins can be used as PWM outputs and 12 as analog inputs (Arduino Micro, 2018). More information about this micro controller can be found at (Arduino Micro, 2018).

5.6. Connections

As showed in Figure 62 the main board will be connected through the USB hub to the stereo-cameras and to the Wi-Fi dongle. With this last component the Jetson TK1 will be able to connect to a network through which it will communicate to an external laptop.

The Jetson TK1 will also be connected through I2C communication protocol to the MPU-6050 which will be attached to the hip of the robot.

There will be two Arduino Due, one for every leg. They will be connected to the Jetson TK1 via serial communication. They will receive the commands from the main board, make the low-level control algorithm run and send the command for the motor drivers to the Arduino Micro by using the serial communication protocol.

The Arduino Micros are connected to the motor drivers through the PWM. Two further digital pins are used for the direction of motion of the motor and for the brake. A PCB integrating the motor driver and the Arduino Micro will be developed. Finally, the motor drivers will then be connected with the motors through the 3 connections for the motor winding and the 3 connection for the hall sensor.

5.7. Housing

Figure 63 shows a 3D drawing of the robot structure with the described electronics mounted on it. The green prism on the torso represents the Nvidia Jetson TK1 and the black one is the USB Hub. The blue prism represents the two Arduino Due while the green prism on the legs are the PCB integrating motor driver and Arduino Micro. As it is possible to notice, the new electronics architecture can be perfectly integrated into the robot structure, leaving also plenty of space for future electronics or sensor additions.

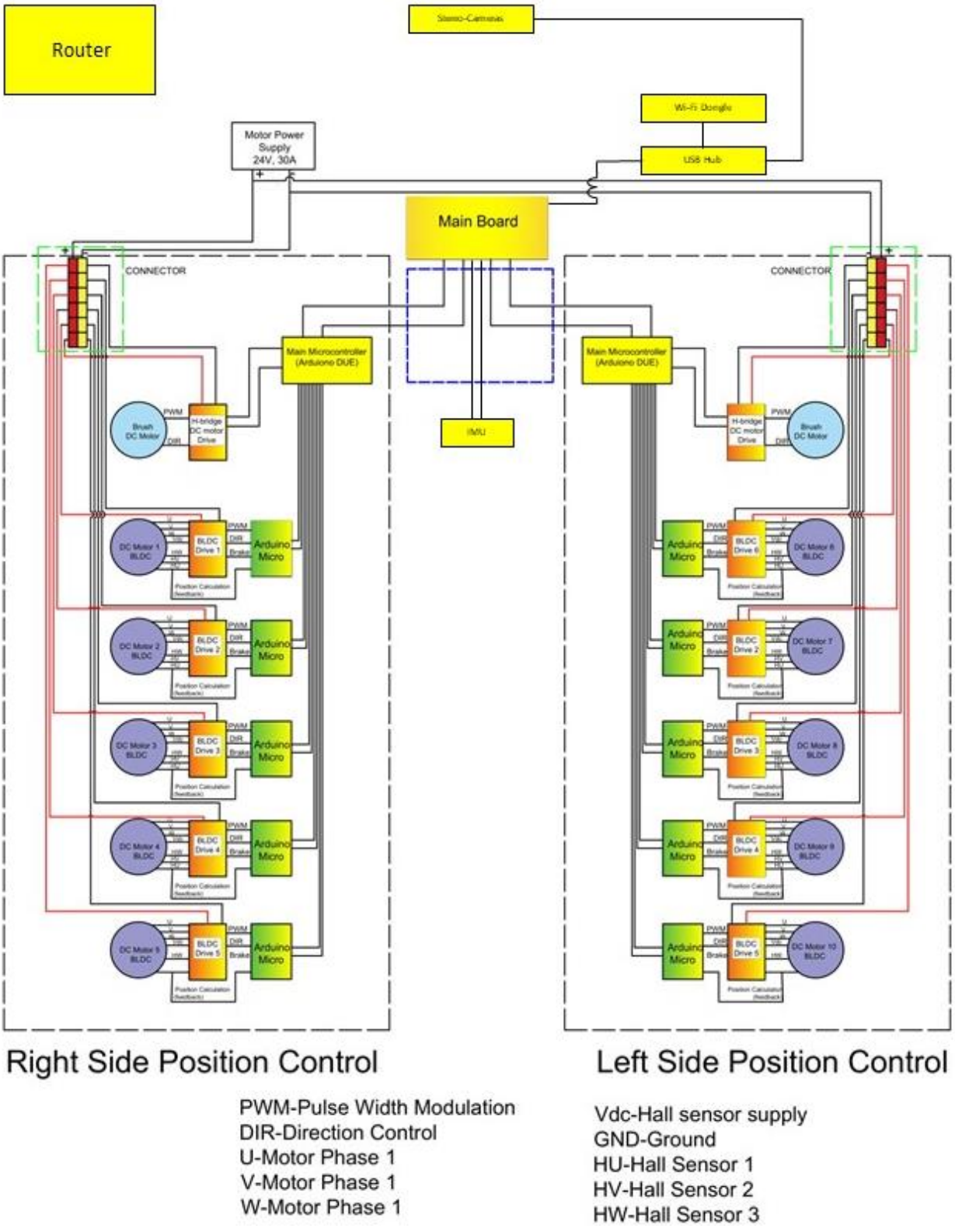

Figure 62: Archie new hardware

Just one small modification to the robot structure should be performed. Considering Figure 64, it is possible to use the notation {L1}, {L2}, {L3}, {L4}, {L5} and {L6} for the horizontal aluminium bars placed on the robot's torso. Starting from the top of the torso, the first two horizontal aluminium bars are named {L1} and {L2}. These bars are 260 mm long and have a width of 26 mm. {L3} and {L4} are the bars under the first two while {L5} and {L6} are the bars in front of them. {L3} and {L5} have also a length of 260 mm and a width of 26 mm. {L4} and {L6}, instead, have a length of 260 mm and a width of 20 mm.

The bars {L3} and {L4} are mounted 14 mm from each other, as well as the bars {L5} and {L6}. The bars {L3} and {L4} are mounted 39 mm from each other while the bars {L1} and {L2} are mounted 93 mm from each other.

The two Arduino Due have a length of 101,6 mm and a width of 53.3 mm. The board is 15 mm heigh and thus, it fits in the space between the bars {L3} and {L4} and {L5} and {L6}. The Arduino Due dimensions are showed in Figure 65. As it is possible to see in the figure, the holes for the screws on the Arduino Due board are placed at 50.8 mm along the maximum dimension of the board. Thus, in order to place the two Arduino Due vertically, it is advised to move the bars {L3} and {L5} 10 mm upward. Considering future possible developments, it would be possible with this configuration to mount up to 4 more Arduino Due vertically or other sensors if it will be required.

The Nvidia Jetson TK1 has a length of 127 mm and a width of 127 mm. It can be mounted in the centre of the torso between the bars {L1} and {L2} leaving a space of 66.5 mm to tis left and to its right. The USB hub can be mounted at its right as showed in Figure 63.

The PCB integrating Arduino Micro and motor driver has a length of 60 mm, a width of 55 mm and a height of 15 mm. It is possible to mount them horizontally along their length. The link connecting the ankle and the knee is 260 mm long and it can store the PCB connecting to the two ankle's joints. The link connecting the knee and the hip, instead, is 310 mm long and it can store 3 PCB respectively for the knee and for the two hip joints. Since the width of these PCBs is 55 mm and the link's width is 54 mm, a small part of the PCB will end outside of the structure. In developing the PCBs, it has to be taken into account that the holes for the screws will have to be placed in the middle of the PCB at 40 mm from each other in order to be mounted on the legs.

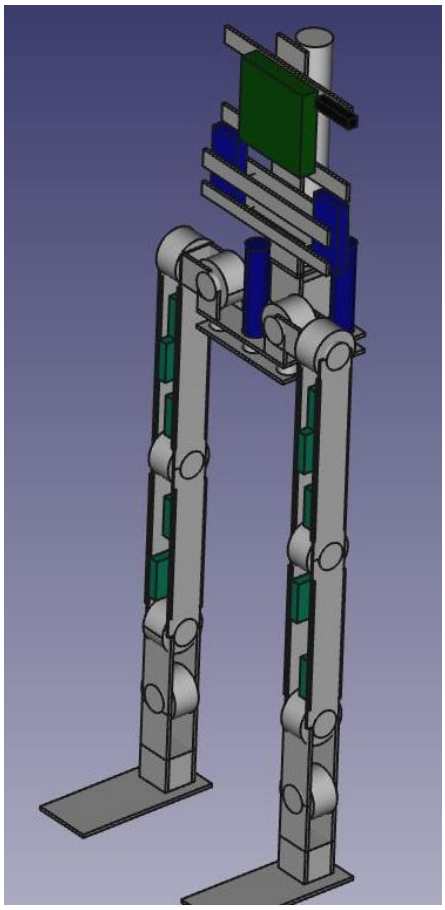

Figure 63: Archie 3D drawing with electronics

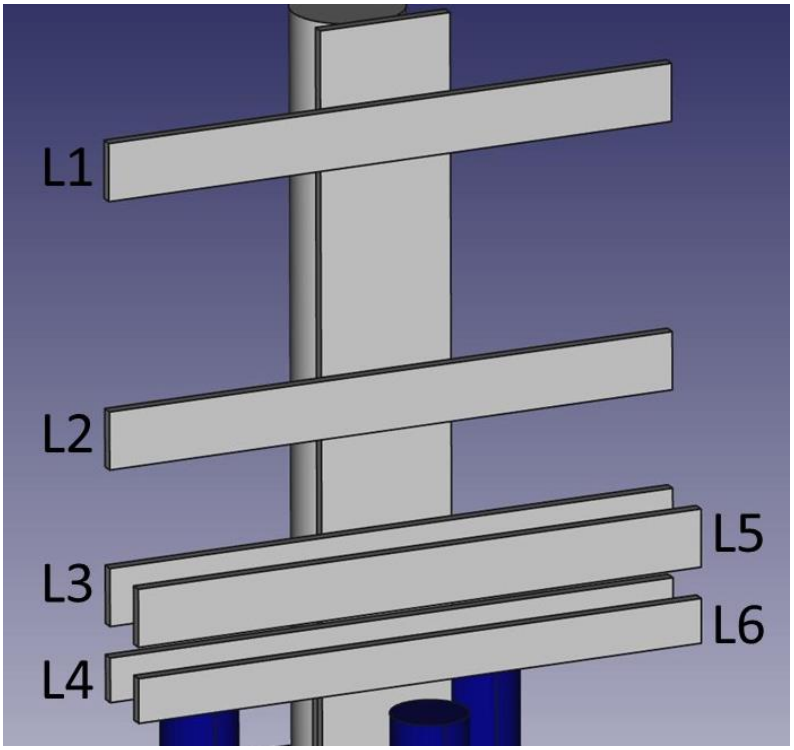

Figure 64: Archie Torso without electronics

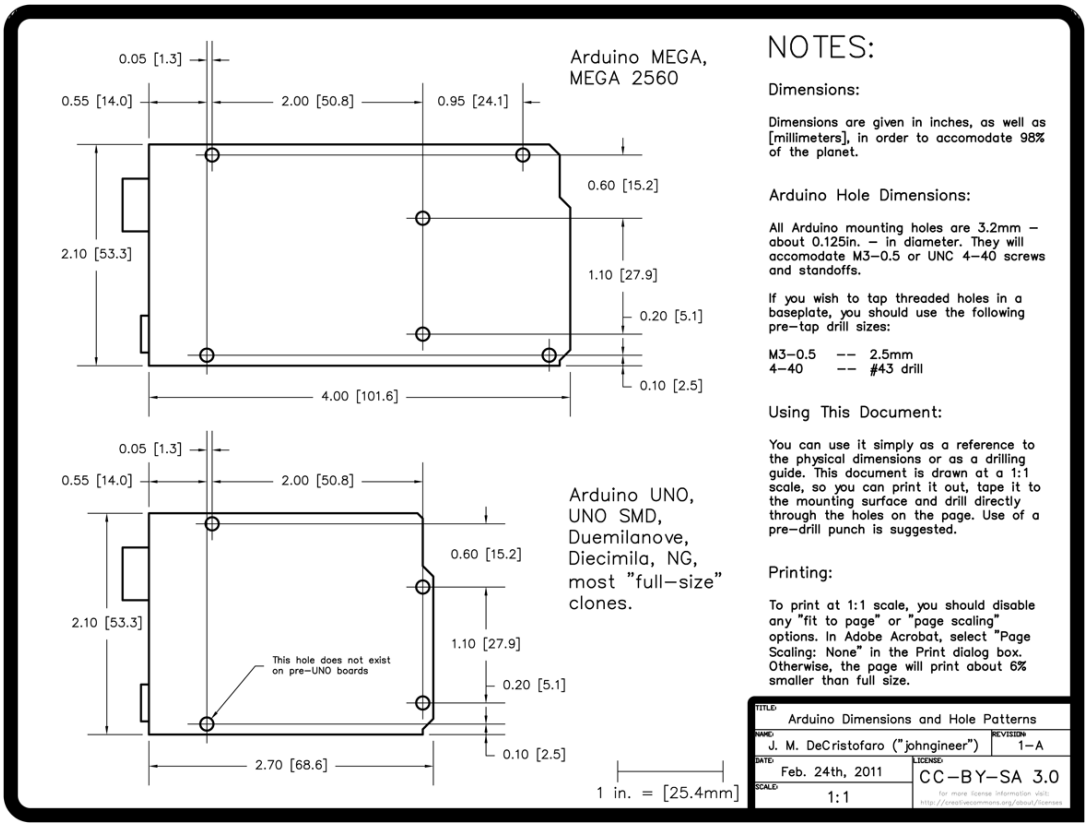

Figure 65: Arduino drawing with dimensions (Arduino dimensions Adafruit, last retrieved, 2018).

6. New Archie software

The new software developed for Archie is supposed to provide driver functionality for the stereo-cameras, the construction of a map of the environment and the planning of the footsteps necessary to reach the target position while also avoiding the obstacles on the way. The software will communicate with the control node in which the advanced control algorithm will run. The advanced control algorithm explained in chapter 4 is still under evaluation and changes are still possible. For this reason, the software had to be implemented in a way that it would make easy the substitution or the removal of one or more part of the software.

Ideally, the aim of the addition of the on-board computer is the increase of the autonomy level of the full system. Thus, one of the key tasks in the realization of the software was to keep the computational power required as low as possible.

The Ubuntu 14.04 Operative system is installed on the Jetson TK1. For the realization of the software, ROS Indigo has been used. Any software, library or tool for robotic application in ROS is called package and it can consist of different nodes or nodelets, each with a specific functionality (d'Apolito et.al., 2016). Nodes and nodelets are both softwares that use the ROS framework. The difference between the two is that the nodelet implements usually just one algorithm allowing developers to be able to use different algorithms without copy transport between them. A ROS node, instead, implements more than one algorithm. The nodes or nodelets of a ROS package exchange data between each other with the use of the topics. These are ROS specific communication data channels which allow the exchange between nodes of ROS specific and custom-made data structure. Another important ROS tool is the ROSService which allows to send requests to the software, like for instance to start or stop the image acquisition from the stereo-cameras.

From the ROS framework different already implemented packages for data manipulation and sensor handling were used. Specifically, in the development of the software were used cv_bridge, image_proc, geometry, rviz, rviz_visual_tools, image_view and pcl_ros.

The cv_bridge package is the interface between ROS and OpenCV and provides a conversion between the ROS image data type and the OpenCV data type. More information can be found at (cv_bridge online documentation, last retrieved 2018).

The image_proc node is also based on OpenCV and provides the removal of the distortions in the images coming from the sensors. The image_view node belongs to the same package of the image_proc node, i.e. the image_pipeline package, and it is used in the developed software for the visualization of the images coming from the stereo-cameras. More information can be found at (image_pipeline online documentation, last retrieved 2018).

The new software analyses the environment around the robot using the point cloud library. As interface between ROS and the point cloud library, the package pcl_ros was used. More information about pcl_ros can be found at (pcl_ros online documentation, last retrieved 2018)

The geometry package, instead, is a mathematical package and it is used for the geometrical computation of the planning algorithm and for handling the pose coming from the SLAM algorithm. More information about the geometry package can be found at (geometry online documentation, last retrieved 2018).

Rviz is a useful tool of ROS which can visualize the point cloud output from the SLAM algorithm, the obstacle detected, the path planned and the computed footstep positions. In order to interact with RViz, the rviz_visual_tools package is used. More information about these two packages can be found at (rviz online documentation, last retrieved 2018) and (rviz_visual_tool, last retrieved 2018).

Figure 66 shows the feedback of the software. On the top, it is possible to see the two image feeds from the image_view node while at the bottom the RViz GUI is visible. In the RViz GUI a grid is automatically drawn in order to give to the user a sense of distance. As a matter of fact, the grid is composed of of 1 m^2 size squares. The point cloud is represented by the white dots over the grid. Moreover, in this representation, every white dot is a point belonging to the point cloud. The obstacles detected are shown in red, the computed avoidance trajectory in blue and the computed footsteps in green. It is possible to see two reference systems in the figure. The one on the grid is the map reference system, while the one over the grid represents the position and attitude of the robot computed by the SLAM algorithm. For these two reference systems, the RGB convention is used (Red = x, Green = Y and Blue = z).

Outside of ROS, also the library libUVC is used for grabbing the images from the stereo-cameras.

The software implemented for Archie, consist in a ROS package composed by 2 nodes:

- A camera manager node, providing the driver functionality for the stereo cameras
- A vision and planning node, providing the obstacle detection and the footstep planning functionality.

As depicted in Figure 67, the user can start and stop the acquisition of the stereo-cameras using two ROSService developed for our application: "*archie_camera/start_capture*" will make the camera manager node start the capture of the picture coming from the cameras and "*archie_camera/stop_capture*" will, instead, stop the capture.

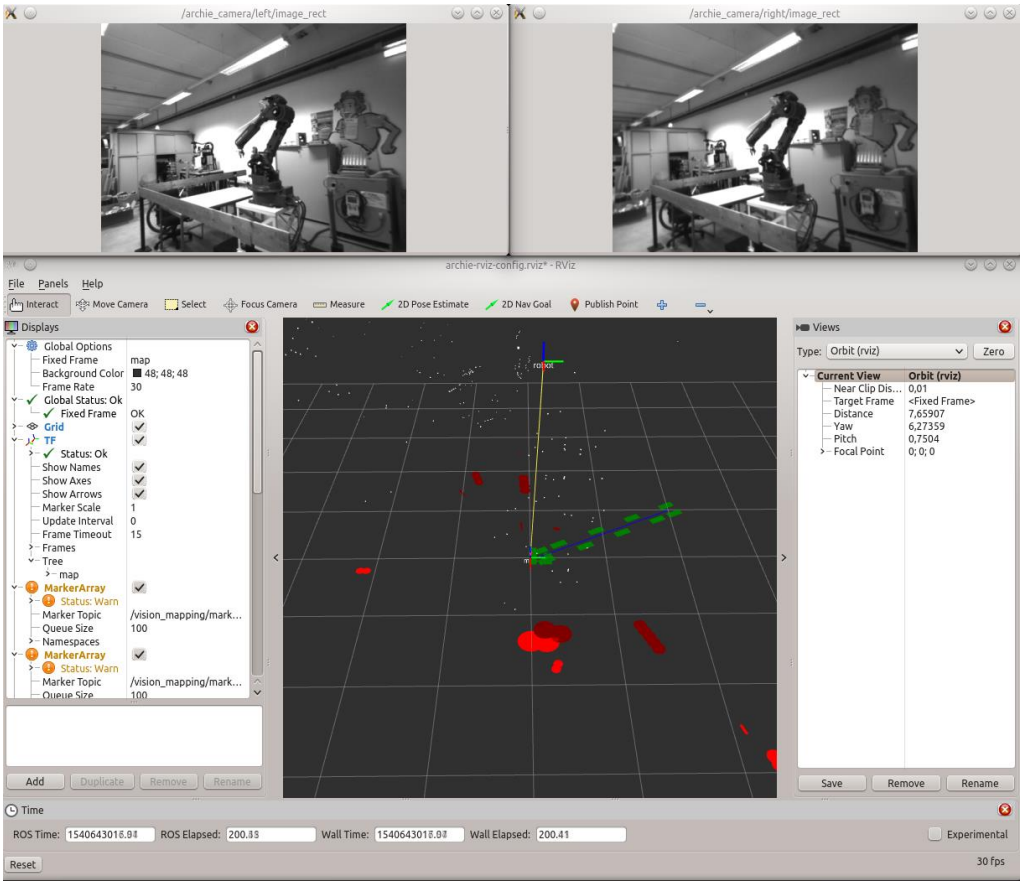

Figure 66: Feedback of the implemented software

The two images, after a rectification process performed by the two image_proc nodes, are then given as input to the stereo_ptam node which will output the point cloud and the robot position result of the processing of the PTAM (Parallel Tracking and Mapping) algorithm. The point cloud is sent on the topic "*/sptam/points*" while the positions on the topic: "*/sptam/robot/pose*".

The point cloud topic and the position topic are read by the vision and mapping node. The user is able, by calling the ROSService "*archie_ai/start_planning*", to send the target position to Archie and to start the planning of the trajectory which lead the robot from its starting position to the new position. It is also possible by calling the ROSService "*archie_ai/load_map*" to load a user-defined map for testing purposes.

The user has also the possibility to call the "*/archie_ai/clearRViz*" service in order to clear the map, the detected obstacles and the planned trajectory and footsteps.

The vision and planning node consist of several algorithms:

- Obstacle detection algorithm which handles the point cloud coming from the stereo_ptam package and detect the obstacle in the environment around the robot.
- The collision avoidance algorithm which takes as input the detected obstacles, the start position and the goal position. As output, it gives a vector of waypoints composing the safe path planned.
- The footstep computation algorithm which computes the position of the feet of the robot and the walk parameters.

The computed footstep and the walk parameters will be sent on the ROS Topic "*/vision_mapping/footsteps*" to the control node. The vision and planning node produces the markers for RViz in order to give to the operator a visual feedback concerning the detected obstacles, the point cloud coming from the SLAM algorithm, the planned trajectory as well as the footstep computed.

From RViz it is also possible to set a new target position which will start a new path planning. A further ROSService, called "*/archie_ai/clear_RViz*", will also give the user the possibility of clearing the map, the detected obstacles, the planned trajectory and footsteps visualized on RViz.

The software is started by running one unique launch file where all nodes with all their configurations parameters are already set up. Furthermore, through the launch file it is also possible to make the various node to execute on different machines on the same network (i.e. the on-board computer and the user's laptop) in order to distribute the computational load on different machines.

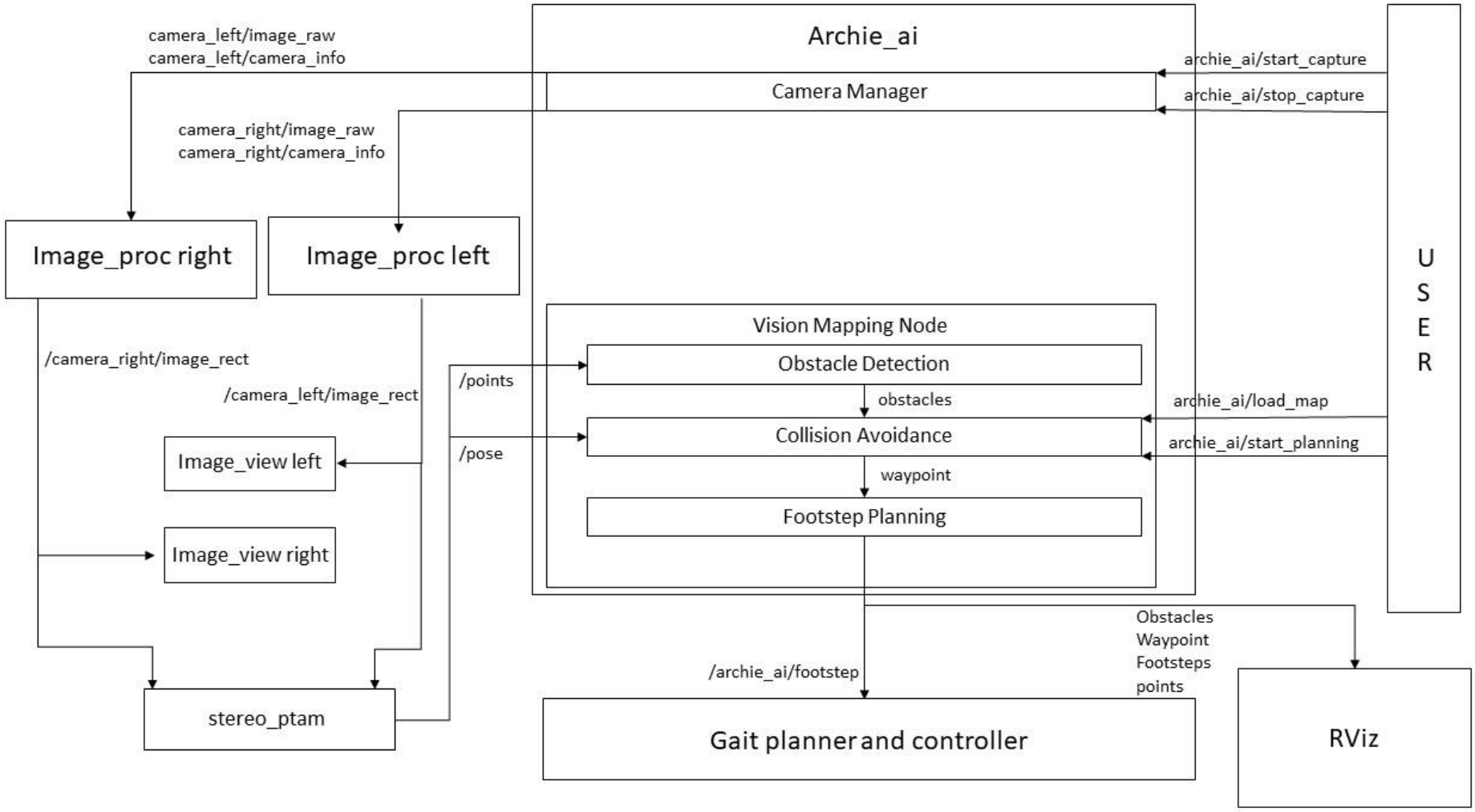

Figure 67: Software architecture

As mentioned before the design of the software changed during the course of this PhD work. The flow diagram of the first version of the software is depicted in Figure 68

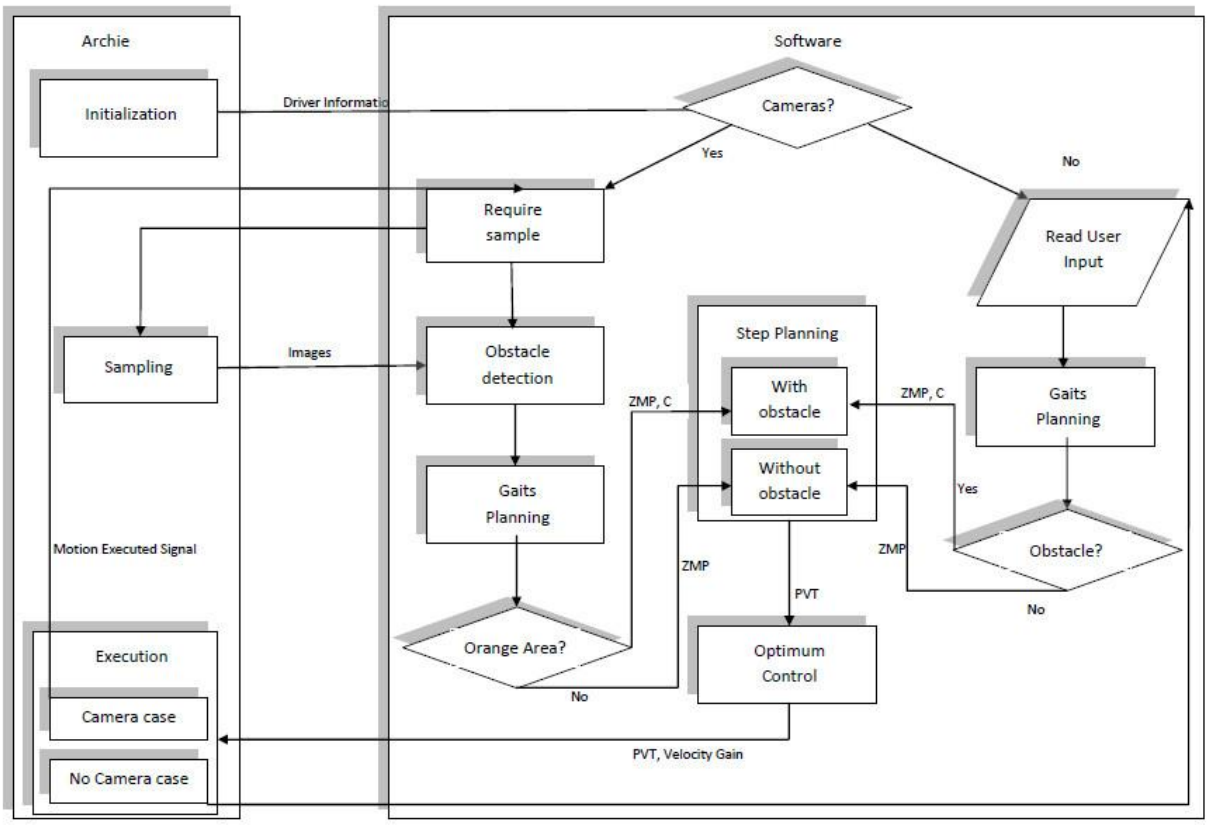

Figure 68: First version of the developed software (Bauer et.al., 2015)

In this first implementation, the software first checked the camera connection. If the cameras were not connected, the software waited for the user to select what type of action Archie had to perform: if he had to step over an obstacle or if no obstacle to step over was present, making the user input the direction to take.

In case the cameras were connected then, the software at first requested a sampling and subsequently waited for the pictures grabbed by the cameras. After receiving them, the software distinguished between areas with no obstacles (green areas), areas where Archie had to step over an obstacle (orange areas) and areas that Archie had to avoid (red areas).

As already mentioned, the gait planning algorithm was initially thought to be part of the software and to be executed after the collision avoidance algorithm. When Archie was in a green area, the artificial potential field step planning method for a regular step was called. Positions and velocities of the joints computed by the algorithm were then sent to the joints' controllers.

When Archie was in an orange area, the function containing the artificial potential field step planning was called, adding the position and height of the obstacle it had to step over. As for the first case, the positions and velocities of the joints were sent to the joint controllers. The gains for the embedded control algorithm resulted from the study of (Schoerghuber, 2014), were sent together with the position and velocities. After the execution, a signal to the software was sent in order to signal the end of the motion and then another sample request to the cameras would had been forwarded.

This version of the software was very simple, and it didn't allow further research and development. The camera stream, for instance was discontinuous and thus, it was not suited for a non-static environment. In order to make it suitable for such an environment, the camera capture strategy should have been entirely replaced. This is not the case of the new version of the software which provides a continuous camera stream.

Furthermore, the control strategy allowed by this software is decentralized. The desired joints and position velocities were sent to the embedded controller and then the software had to wait for a signal from the controllers confirming that the movement was finished. In order to allow the development of a centralized strategy, the new version of the software was implemented, and the gait planning algorithm was moved on the control node.

6.1. Camera manager node

The first node developed is the camera manager. The node is supposed to provide the "driver" functionality for the cameras, i.e. starting and stopping the capture, as well as a necessary data pre-processing. The picture coming from the stereo cameras, as a matter of fact, are interlaced between each other. The incoming data are in format YUYV where Y is the light intensity and U and V the colour information. Each pixel is coded with 16 bit, 8 bit for the Y and 8 bit for U or V. OpenCV converts the data in RGB as soon as it gets it, which results in having useless data (d'Apolito et.al., 2016). The camera manager node thus, grabs and separates the data coming from the stereo cameras in left and right image.

The camera manager is based on the libUVC library (LibUVC, last retrieved 2016), an open-source library allowing the acquisition of the images from USB cameras. By means of this library, the camera

manager node can grab the data from the cameras, separate the two pictures and send it on a ROS topic.

The tests proved that the camera manager is able to stream a 30 fps video stream from both cameras with a resolution of 630x480. Figure 69 shows the stream of the cameras from the ROS tool *"*image_view*"*.

Figure 69: stereo-cameras video stream

6.2. The rectification and the stereo_ptam

The stereo-PTAM is a SLAM algorithm. It provides the position and attitude of the stereo-camera in a map built by the algorithm giving the left and right image of the stereo-cameras.

The main characteristic of this algorithm is that the tracking and the mapping tasks are executed in parallel, speeding up in this way the computation. Its first implementation can be found in (Klein, G. and D. Murray, 2007) and it was thought of as an algorithm for a monocular camera in Augmented Reality (AR) workspaces. It was then used especially in Unmanned Aerial Vehicle (UAV) application like for instance in (Weiss et.al., 2012) and in (Engel et.al., 2012).

The first stereo-cameras application of the PTAM is described in (Pire et.al., 2015). In their method the tracking begins with the first pose and the initialization of the map. The last one is done by means of the triangulation of the features in the first pair of stereo-images. Then the tracking thread minimize the reprojection error between the features in the stereo-images pair and their corresponding points in the map for every frame, thus estimating the current robot pose. Some of the frames which incorporate new points can be strategically selected in order to augment the map. In parallel to the tracking thread, the mapping thread is always trying to minimize the reprojection error by adjusting all points and keyframes in a bundle (Pire et.al., 2015).

The stereo matching between the left and right images is based on the epipolar geometry like most of the stereo algorithms. The epipolar geometry is based on the simple fact that the same point projected on two different camera frames will be in two different places in the picture. (See Figure 70).

As a matter of fact, using just the left image, one would not be able to retrieve the depth information of the X point since every point on the $\overline{OX}$ line projects to the same point on the image plane. Adding the second image, one can triangulate the coordinate of the X point since the points on the $\overline{OX}$ line projects in the second image plane in different places (OpenCV Online Documentation on Epipolar Geometry and Stereo Vision, last retrieved 2017).

The $\overline{OX}$ line, projected in the second image, forms a line which is commonly named as epiline to the point x. The plane containing the camera centres O, O' and X is called Epipolar Plane. The projection of the two centres in the other camera image plane, in Figure 70 indicated as e and e', are called epipole.

Usually, in order to speed up the computation and the search of the corresponding point in the two image frames the search is performed on the epiline instead of on the whole image. The difference between the position of the points in the two image frames is called disparity and it is supposed to be higher the more the closer the point is to the camera. To experience this, it is possible to close one eye and then rapidly open it while simultaneously closing the other one. The closer objects will apparently do a significant jump.

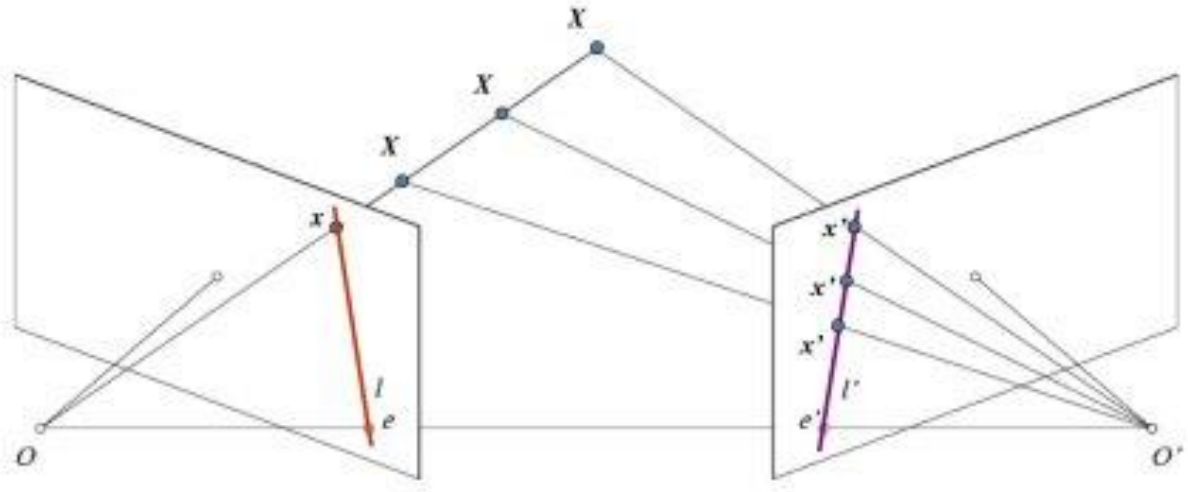

Figure 70: Epipolar geometry (OpenCV Online Documentation on Epipolar Geometry and Stereo Vision, last retrieved 2018)

If there is just a horizontal offset between the two image planes, i.e. the two image planes are coplanar, then the epipolar line is horizontal. This results in a simplification of the search of the matching point. Due to distortion and to an imperfect alignment of the two image centres, a stereo-image pair needs to be rectified before being processed.

The rectification process consists mainly in the correction of the distortion of the image and on the translation of the warping of the images in order to make it look like there is just a horizontal shift between the two image planes.

The stereo-PTAM was tested with publicly available repository and with a robotic platform. All tests were successful showing a small error between the pose localization and the ground truth. Figure 71 shows the results of the tests of the stereo-PTAM with the MIT Stata Data Centre Dataset described in (Fallon et.al., 2012). In the figure, the black points are the points composing the map,

the green line is the trajectory determined by the PTAM algorithm and the blue line instead is the ground truth.

The parallelization of the tracking and mapping task is a huge computational advantage and it has the potential to allow a real time application even when running on an embedded board. This, together with the good performances showed by the tests, made the stereo-PTAM preferable to other SLAM algorithm for Archie.

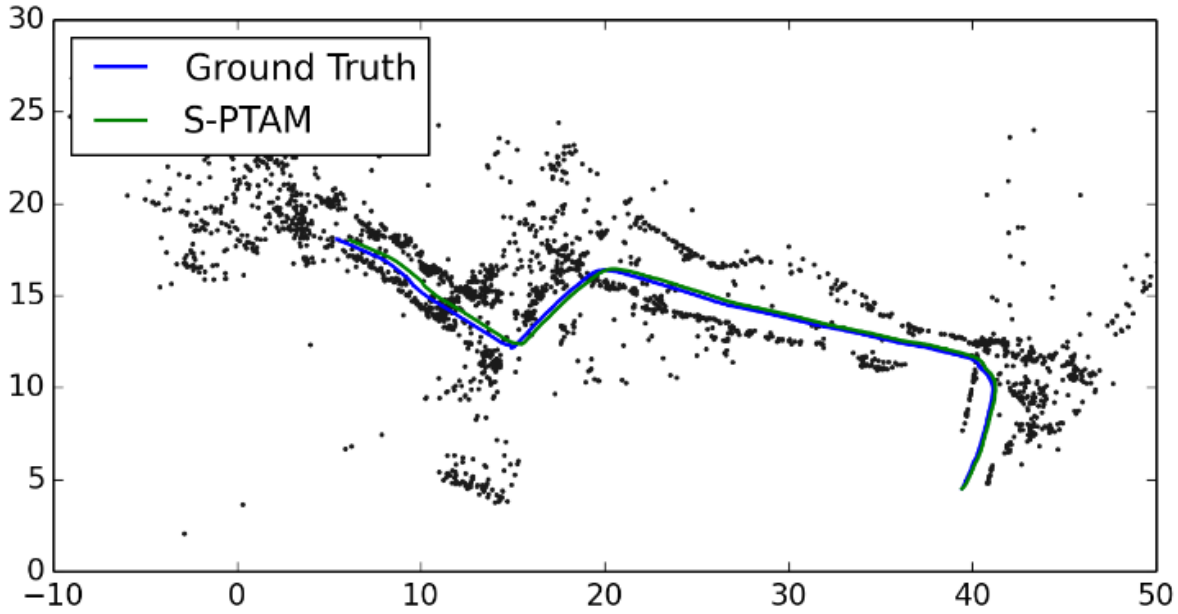

Figure 71: Test of the stereo-PTAM with the MIT Stata Centre Dataset (Pire et.al., 2015)

6.3. Mapping and planning node

The second node is the core of the presented software architecture and contains the obstacle detection algorithm, the path-planning and collision avoidance algorithm and the subsequent footstep computation. As input it takes the point cloud produced by the stereo_ptam node and the position and attitude of the robot in the map created by the PTAM.

For the obstacle detection, the Ground Plane approach is used. It is based on the assumption that everything that is above the Ground Plane is an obstacle (d'Apolito F. and C. Sulzbachner, 2017). In

indoor environment, such as the ones Archie is supposed to operate, this assumption can be considered always valid.

In order to detect the obstacles with such an approach, it is used first a pass-through filter, and finally a Euclidean cluster extraction for the segmentation of the obstacles. After these first processing, the bounding boxes and the centroids of the obstacles are computed, and they are finally approximated as circles. The centres and the radius of the circles approximating the obstacles are the input data of the planning algorithm.

In contrast to the state of the art described in chapter 2, the planning algorithm for Archie is performed in two steps. First, an obstacle avoidance algorithm computes a safe trajectory for the robot from a starting position to an end position, then another algorithm computes the position of the feet during the trajectory. Each one of these two algorithms are implemented as a standalone class.

The reason for such an implementation resides in the fact that Archie is supposed to be used as a test platform for the implementation of advanced control algorithm. This implementation of the footstep planning strategy, as a matter of fact, guarantees the possibility of further development and research by replacing one or more algorithm inside the software with newer ones without removing them completely or changing the full planning strategy.

The obstacle avoidance algorithm is a geometrical path planning based on the binary tree approach. It computes the waypoints as the intersection points between the two sets of tangents to circular obstacles from the robot position and the goal. All these waypoints are then represented in a binary tree structure which is used to choose the shortest path to the end position. The output of the algorithm is the set of points belonging to the shortest trajectory chosen by the algorithm.

It has to be noted that, in contrast with to state of the art, the representation of the environment around the robot is not grid-based and, thus, neither is the obstacle avoidance algorithm. This design choice was done mainly in order to keep the computational load low. Grid based representation and grid-based planning, as a matter of fact, can become computationally heavy if the search space is large. A small indoor demonstrator for this obstacle detection and avoidance strategy was developed using a UAV platform and it is described in (d'Apolito F. and C. Sulzbachner, 2017).

The waypoint list computed by the collision avoidance algorithm is then used to compute the footsteps positions which will then be sent on a ROS topic to the control node.

6.3.1. The obstacle detection algorithm

For the obstacle detection two options were evaluated: The Inverted Cone Algorithm and the Ground Plane approach.

The Ground Plane approach is a simple approach based on the assumption that everything that is above the ground plane is an obstacle (d'Apolito F. and C. Sulzbachner, 2017).

The Inverted Cone algorithm is an obstacle detection algorithm developed in (Manduchi et.al., 2005). This method, as showed in Figure 72, clusters all the points which belong to the same inverted cone by checking the slope between them and their heights. Mathematically, this algorithm is based on two definitions:

- **Definition 1:** 2 points p_1 and p_2 are compatible with each other if they satisfy the 2 following conditions:

  1. $H_{min} < \left|p_{2,y} - p_{1x}\right| < H_{max}$ (71)
  2. $\frac{|p_{2,y} - p_{1,y}|}{\|p_2 - p_1\|} > \sin\theta_{max}$ (72)

  With H_{min}, H_{max}, θ_{max} constant parameter representing minimum and maximum height and the maximum slope. Their values depend on the mechanical configuration of the robot.

- **Definition 2:** 2 points p_1 and p_2 belongs to the same obstacle if they are compatible with each other or if there is a chain of point pairs connecting them.

Contrary to the Ground Plane Approach which can be used just in indoor environment, this approach has reliable performances both in indoor and in outdoor environment. However, it requires a slope map and a dense point cloud. As it is possible to see in Figure 71, the point cloud output from the PTAM algorithm is sparse and on the robot there is no other sensor which can provide data for the obstacle detection. Furthermore, the inverted cone algorithm would have required also a further

segmentation process after the detection. Consequently, due to the sparsity of the input data and, the reduction of the computational load of all the system, the design choice was the Ground Plane approach.

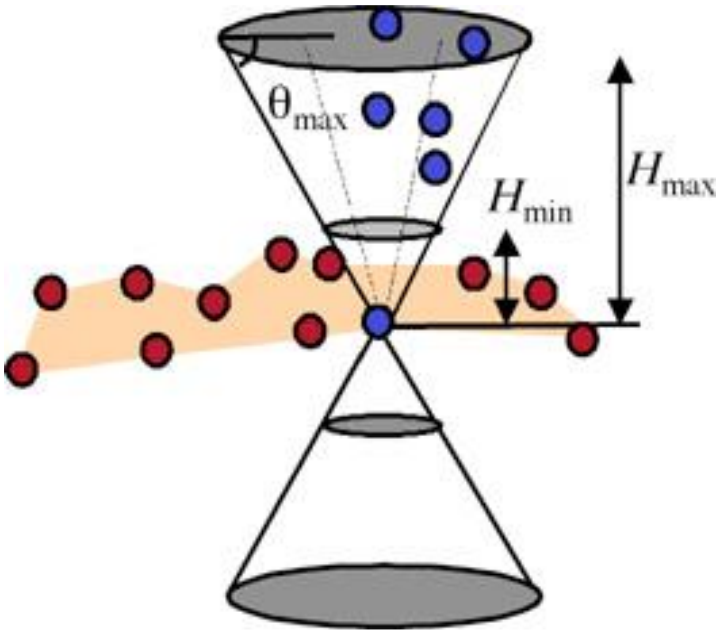

Figure 72: Graphical representation of the Inverted Cone Algorithm developed in (Manduchi et.al., 2005)

The vision and mapping node receives the sparse point cloud computed from the Stereo-PTAM. The coordinates of the points in the incoming data are defined in the left camera frame. The point cloud is then given as input to the pass-through filter which is a simple filtering process along a specific direction. For Archie it is used to filter the input dataset along the "z" direction. Doing so, it is possible to eliminate all the points from the input point cloud which have a "z" coordinate higher than 5 meters. The reason for the implementation of such a filter is not only the down sampling of the input dataset but also the elimination of inaccurate points from the dataset. As a matter of fact, as showed in Figure 73, in the left camera frame the "z" axis is the axis coming out of the camera, the points with a high "z" coordinate in the input dataset are points that are far away from the cameras. These points, due to the small baseline between the cameras, do not have a high accuracy.

The input point cloud is then rotated according to the transformation matrix between the map frame and the left camera frame. Figure 74 shows the reference systems on which the stereo-PTAM is based on. In the figure, the *base_link* frame is the reference system fixed on the robot. The point cloud, thus, has to be rotated from the *camera_frame* to the *base_link* frame and then from the *base_link* frame to the *map* frame, before being added to the map and being visualized. The

difference between the map frame and the base_link frame consists in a rotation of the attitude of the robot and a translation of the robot's position.

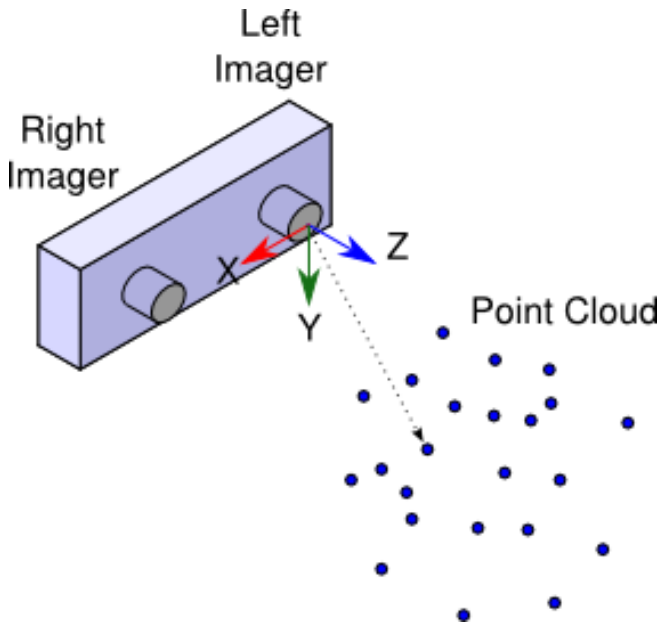

Figure 73: Point cloud defined in the left camera frame (stereo_image_proc online documentation, last retrieved 2018)

Considering (φ, ψ, θ) the euler angles respectively around x,y and z one the can define the rotation matrices:

$$T_x = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{73}$$

$$T_y = \begin{bmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & p_y \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{74}$$

$$T_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{75}$$

$$T = T_x T_y T_z = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & p_y \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{60}$$

With (p_x, p_y, p_z) cartesian coordinates of Archie's position in the map.

The transformation between the *camera_frame* and the *base_link* frame consists first, in a 90° rotation around the x axis and then in another 90° rotation around the z axis. There is no translation between them. The rotation matrices between the *camera_frame* and the *base_link* frame are showed in (59), (60) and (61).

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \tag{76}$$

$$R_z = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{77}$$

$$R = R_z R_x = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \tag{78}$$

Next step of the obstacle detection is then the Euclidean Cluster Extraction. Like every clustering method, this algorithm aims to divide the input point cloud into smaller parts called clusters. The Euclidean Cluster Extraction algorithm generates clusters of points which lie in a sphere of a given radius. As written in (Rusu, 2009), this clustering method can be implemented in a point cloud partitioned using a 3D boxes grid, or more generally, using a tree structure.

In the course of this PhD work, it was used the Kdtree which is a data partitioning structure which allows a fast and efficient nearest neighbour search around a point or points. As described in (Euclidean Cluster Extraction Online Documentation, 2017), the algorithmic steps of the Euclidean Cluster segmentation are the following:

1. Initialize a Kd-Tree which represents the input point cloud;
2. Initialize an empty list of clusters P, and a queue of the points to check Q;
3. **For** every point $\mathrm{p_i}$ belonging to P **do**:
   3.1. Push $\mathrm{p_i}$ to the considered queue Q;
   3.2. **For** every point $\mathrm{p_i}$ belonging to Q **do**:

3.2.1. Look for the set P_k^i of points near p_i included in a sphere of radius $r < d_{th}$;

3.2.2. **For** every neighbour p_i^k belonging to P_k^i **do**

3.2.2.1. Check if it has already been processed,

3.2.2.2. **If** it has not already been processed, **then**

3.2.2.2.1. Push it to Q;

3.3. Push Q to the list of clusters C,

3.4. Re-initialize Q to an empty list;

The Euclidean Cluster Extraction algorithm is usually preceded by a RANSAC algorithm to eliminate the points belonging to the floor or, more generally, to flat surfaces. In this case, though, it was not needed because there were no points belonging to the floor in the conducted experiments. Furthermore, it is usually preceded by a voxel grid representation of the point cloud in order to decrease the computational load. Nevertheless, it was not needed due to the sparsity of the data.

Clusters of points within a 30 cm radius were created. Every cluster represents a detected obstacle. For every cluster, its centroid and bounding box of the detected obstacles are subsequently computed as follows. After the centroids are computed, the generic detected obstacle is rotated in its eigen space and the maximum and minimum values of the points coordinates in each eigen axis is computed. The bounding box structure is filled with the point at the top left of the bounding box and its three dimensions: height, width and depth. The top left point is then rotated in the world frame.

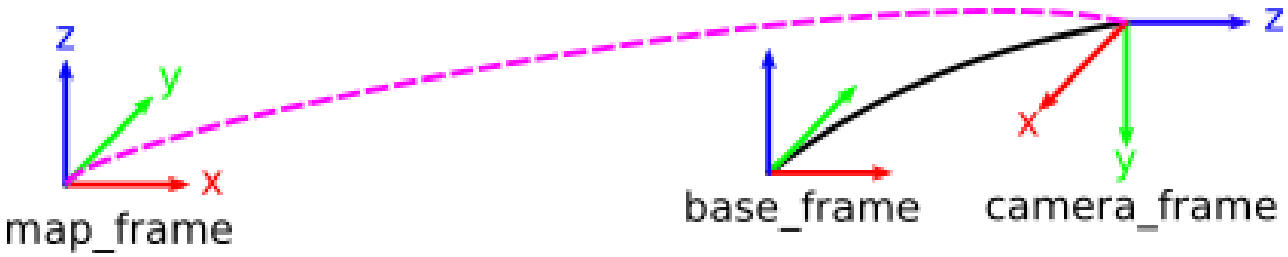

Figure 74: S-PTAM reference systems (S-Ptam Github Documentation, last retrieved 2018)

From the knowledge of the bounding box it is then possible to proceed with the approximation of the obstacles as circles. One drawback of this approximation consists in the fact that the circle approximation can occlude the obstacle-free space. Consider for example an obstacle shaped as an elongated rectangle. The circle approximating it, needs to have a diameter as long as the maximum

size of the obstacle. This results in having an approximating circle that occlude also much of the free space as showed in the first of Figure 75. In order to solve this inconvenient, first the dimensions (width and depth) of the obstacles are checked with relation (79).

$$n = \begin{cases} \frac{width}{depth} & width > depth \\ \frac{depth}{width} & depth > width \end{cases} \tag{79}$$

If the dimensions of the obstacle are comparable (±10%) then the obstacle is approximated as a circle otherwise it is approximated as a series of *n* intersecting circles, as showed in the second of Figure 75. Considering x_m and y_m, the minimum x and y coordinates of the bounding box, the centres of the circles are computed in the eigen space of the obstacle as follow:

$$C_i = \begin{cases} \left(\frac{x_m}{2} + i\frac{depth}{2}, \quad y_m + \frac{depth}{2}, \quad \frac{height}{2}\right) & width > depth \quad i = 1, \dots, n \\ \left(\frac{x_m}{2} + \frac{depth}{2}, \quad y_m + i\frac{depth}{2}, \quad \frac{height}{2}\right) & depth > width \quad i = 1, \dots, n \end{cases} \tag{80}$$

The radius of the obstacle id computed as:

$$r = \begin{cases} depth \cos 45° & width > depth \\ width \cos 45° & width < depth \end{cases} \tag{81}$$

The computed obstacles are then rotated back in the fixed reference frame and inserted in a queue which will send them to the path planning algorithm

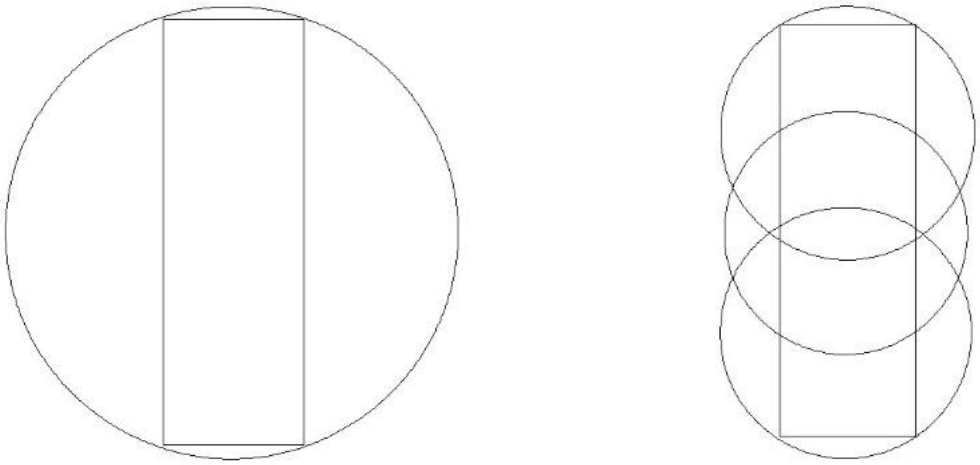

Figure 75: elongated rectangle approximated with a circle and with a series of intersecting circles (d'Apolito, 2018).

The test on the obstacle detection showed that the approach was lightweight and well suited for a real time execution. Furthermore, the representation of elongated obstacles as series of intersecting circles proved itself to be a good approach as it was able to correctly represent the obstacles without occluding the free space. Figure 76 and Figure 77 show two of the tests performed. As it is possible to see, the calculated obstacles represented well the point cloud that they received.

It has to be noted though, that the point cloud computed from the PTAM is sparser than it was expected. Figure 78 shows a case encountered in bad lighting condition. As it is possible to see, there were no points in the point cloud representing many objects. This results in a failure in the detection of these objects. Although the point cloud is denser in case of good lighting, some object is still failed to be represented in the point cloud (see Figure 79)

The reason for this relies most probably in the small baseline of the camera which makes it difficult to have a good stereo matching between the left and right images.

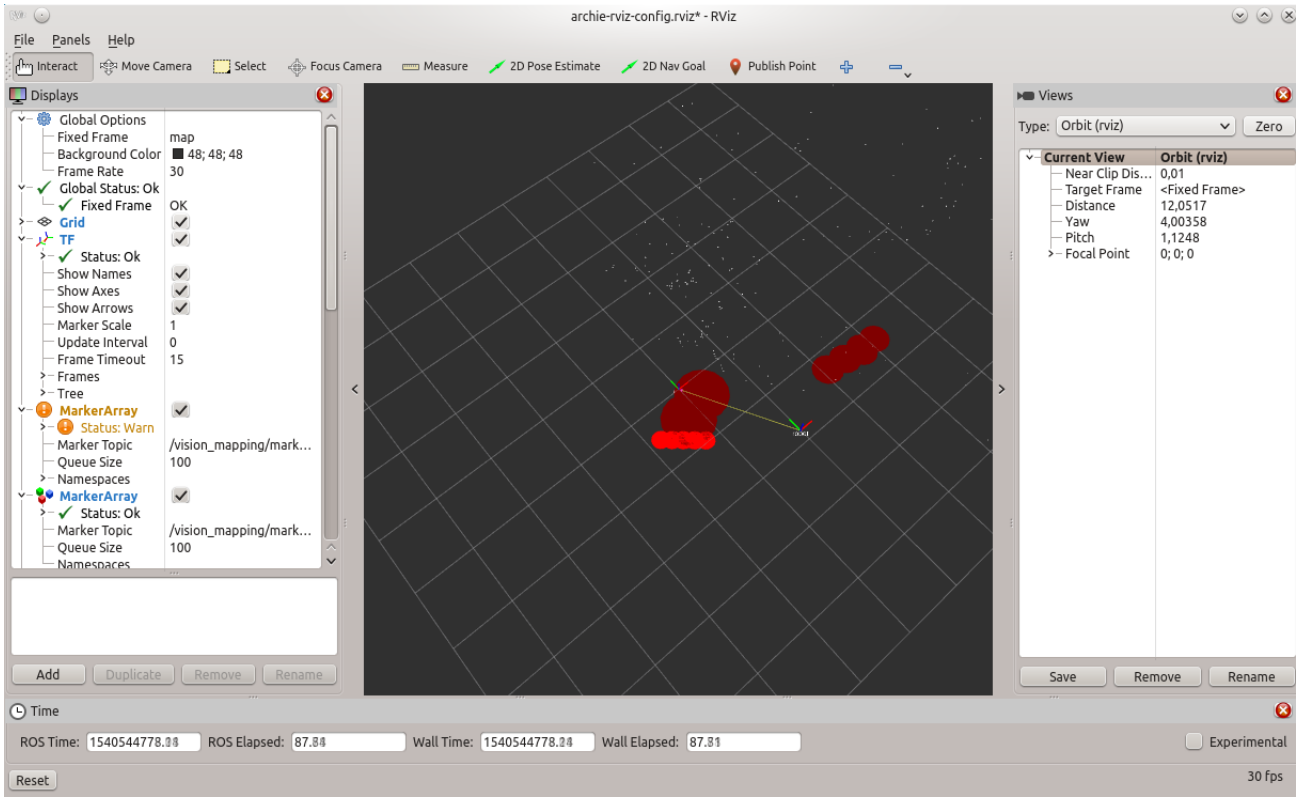

Figure 76: obstacle detection test

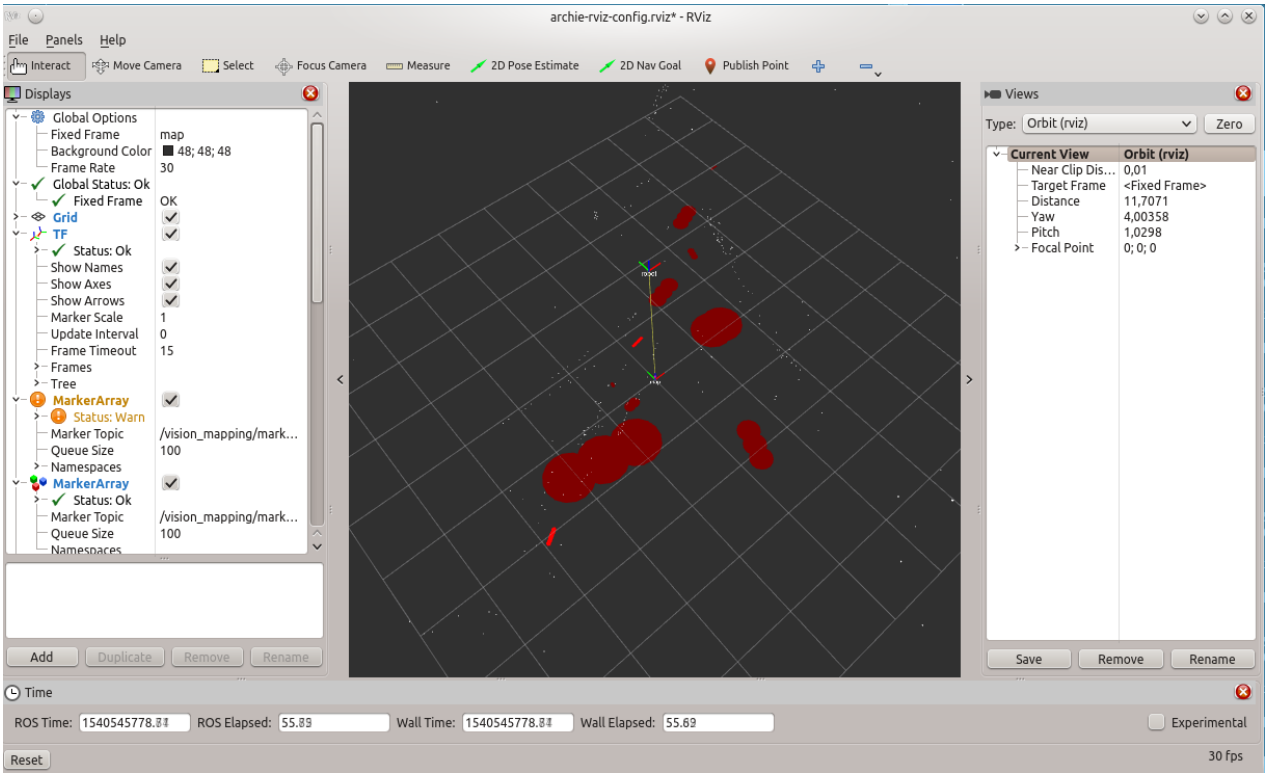

Figure 77: further obstacle detection test

Figure 78: PTAM result with the stereo-cameras in bad lightning condition

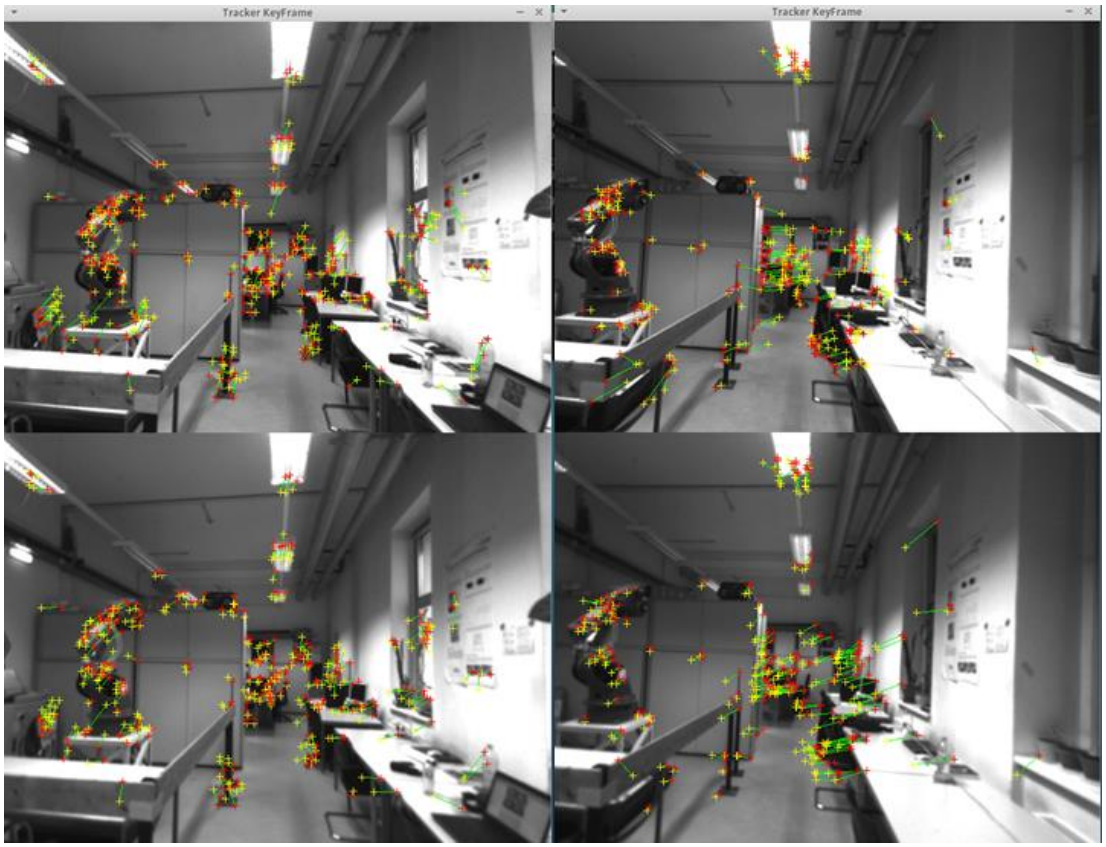

Figure 79: PTAM test result in good lightning condition

6.3.2. The obstacle avoidance algorithm

The obstacle avoidance algorithm is a geometrical algorithm based on the binary tree approach. The binary trees, provided a known environment, give the robot various possible paths among which the shortest path to the goal is chosen. As mentioned before, the obstacles are approximated with circles with different radius, according with the dimension of the obstacle. The dimensions of every obstacle are then increased with the dimension of the robot in order to provide safe trajectories for the robot.

The start position is the root of the binary tree which is then filled with the next points of the trajectory. In the case an obstacle is present between the starting position and the goal, two avoidance waypoints are calculated as the interception points of the two sets of tangents to the circle starting from the starting point and from the goal as depicted in Figure 80. In the figure, the two circles indicate the circle representing the obstacles (the inner circle) and the circle with the diameter increased by half of the robot's hip length (the outer circle).

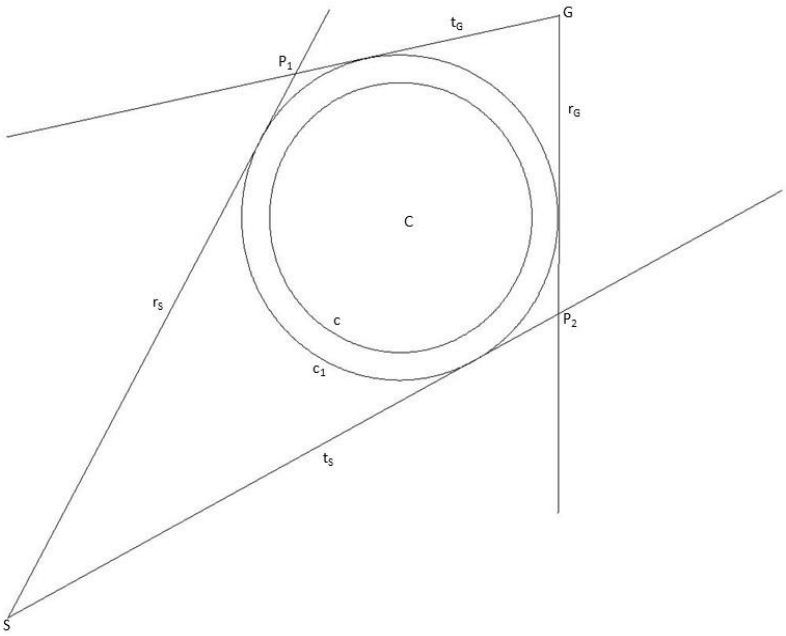

Figure 80: Avoidance waypoint computed by the algorithm (d'Apolito F. and C. Sulzbachner, 2017).

Figure 81 gives a more detail explanation of the algorithm. The point $S = (S_x, S_y)$ is the start position of the robot and $G = (G_x, G_y)$ is the end position.

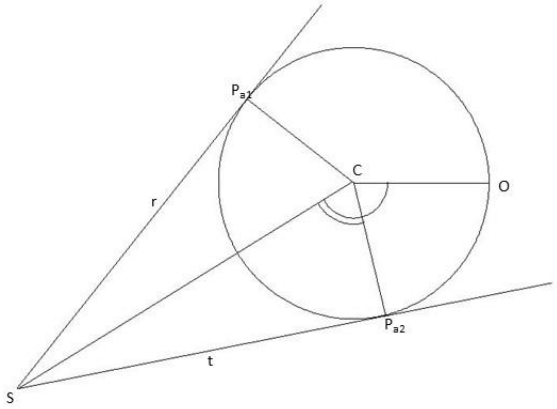

Figure 81: tangent point computation (d'Apolito F. and C. Sulzbachner, 2017).

As described in (d'Apolito et.al., 2017), in order to find the intersection points and, consequently, the avoidance waypoints, one first needs to compute the equation of the 4 tangents lines. It is possible to compute the angle $\widehat{OCS}$ and the angle $\widehat{P_{a2}CS}$ with the (82) and (83):

$$\widehat{OCS} = \tan^{-1}\left(\frac{S_y - C_y}{S_x - C_x}\right) \tag{82}$$

$$\widehat{P_{a2}CS} = \cos^{-1}\left(\frac{\|\overline{CP_{a2}}\|}{\|\overline{SC}\|}\right) \tag{83}$$

Given this two angle and because of the equivalence of the angle $\widehat{P_{a2}CG}$ and of the angle $\widehat{GCP_{a1}}$ it is possible to calculate the angle $\widehat{OCP_{a2}}$ and $\widehat{OCP_{a1}}$ respectively with the sum and the difference of the angle $\widehat{OCS}$ and the angle $\widehat{P_{a2}CS}$. Having these two angles allows to compute the tangent point P_{a1} and P_{a2} with the relation (84) and (85).

$$P_{a1} = \begin{pmatrix} C_x + r \cdot \cos \widehat{OCP_{a1}} \\ C_y + r \cdot \sin \widehat{OCP_{a2}} \end{pmatrix} \tag{84}$$

$$P_{a2} = \begin{pmatrix} C_x + r \cdot \cos \widehat{OCP_{a2}} \\ C_y + r \cdot \sin \widehat{OCP_{a2}} \end{pmatrix} \tag{85}$$

After the computation of the tangent point, one can compute the equation of the tangents to the circle.

$$\begin{aligned} \bar{r_s}&: \quad y = m_1 x + q_1 \\ \bar{t_s}&: \quad y = m_2 x + q_2 \\ \bar{r_g}&: \quad y = m_3 x + q_3 \\ \bar{t_g}&: \quad y = m_4 x + q_4 \end{aligned} \tag{86}$$

With:

$$\begin{aligned} m_1 &= \frac{P_{a1y} - S_y}{P_{a1x} - S_x} \\ q_1 &= -S_x m_1 + S_x \\ m_2 &= \frac{P_{a2y} - S_y}{P_{a2x} - S_x} \\ q_2 &= -S_x m_2 + S_x \\ m_3 &= \frac{P_{a3y} - G_y}{P_{a3x} - G_x} \\ q_3 &= -G_x m_3 + G_x \\ m_4 &= \frac{P_{a4y} - G_y}{P_{a4x} - G_x} \\ q_4 &= -G_x m_4 + G_x \end{aligned} \tag{87}$$

Where P_{a3} and P_{a4} are the tangency points to the circle of the set of tangents originating from the goal position.

Subsequently, one can compute the intersection points Cartesian components by means of the two systems of linear equations composed by the pairs of lines $r_s - t_g$ and $t_s - r_g$ with the relation (88)

$$
\begin{aligned}
P_{1x} &= \frac{q_4 - q_1}{m_1 - m_4} \\
P_{1y} &= m_1 P_{1x} + q_1 \\
P_{2x} &= \frac{q_3 - q_2}{m_2 - m_3} \\
P_{2y} &= m_2 P_{2x} + q_2
\end{aligned}
\tag{88}
$$

In case one or more of the angular coefficients are infinite or 0, it is possible to use respectively the x or y coordinate of the starting or the target position and substitute it in the equation of the line whose angular coefficient is not singular to compute the other coordinate. Once the intersection points P_1 and P_2 are computed, they are stored in the binary tree (d'Apolito F. and C. Sulzbachner, 2017).

The binary tree developed for the algorithm stores the coordinates of the avoidance waypoints, the centroid of the obstacle to be avoided, the length of the path which is incrementally computed and an id.

The steps of the collision avoidance algorithm are the following:

1. Check if the paths planned reached the target position in any of the leaves in the tree.
2. **If** not, **then**
3. **For** every leaf **do**:
    3.1. Check if one of the obstacles is in the way to reach the goal.
    3.2. **If** more than one obstacles are present, **then**
        3.2.1. Choose the obstacle to considered for the computation of the pair of avoidance waypoints.
    3.3. Compute the pair of avoidance waypoints and their ids and store them in the tree.
    3.4. **If** the computed points are inside the obstacle area, **then**
        3.4.1. delete the computed points.
4. Check if the planning succeeded for all the leaves.
    4.1. **If** in all the leaves the planning failed, **then**

4.1.1. Abort the planning.

4.2. **If** not, **then**

4.2.1. go to step 1

Figure 82 depicts the flow diagram of the collision avoidance algorithm.

The choice of the obstacle to consider, is based on two factors:

- The distance between the robot position and the obstacle's centre.
- The distance between the line joining the robot position and the target position, and the obstacle's centroid.

For every obstacle, the algorithm computes the distance between the segment joining the starting and the target position. The resolution method for the computation of the distance is explained in (Distance line segment resolution method, last retrieved 2018). As described, the main focus of this calculation relies in the determination of whether or not the perpendicular through the obstacle centroid is inside the segment between starting and target position or not. One can describe the line between the start and the target position with the parametric relation:

$$\bar{l} = \begin{bmatrix} S_x + (G_x - S_x)t \\ S_y + (G_y - S_y)t \\ 0 \end{bmatrix} \tag{89}$$

The distance is minimum if the relation (90) is satisfied:

$$t = -\frac{\begin{bmatrix} S_x - C_x \\ S_y - C_y \\ 0 \end{bmatrix} \begin{bmatrix} G_x - S_x \\ (G_y - S_y) \\ 0 \end{bmatrix}}{\left\| \begin{bmatrix} (G_x - S_x) \\ (G_y - S_y) \\ 0 \end{bmatrix} \right\|^2} \tag{90}$$

The (90) can be rewritten as the (91).

$$t = -\frac{(S_x - C_x)(G_x - S_x) + (S_y - C_y)(G_y - S_y)}{(G_x - S_x)^2 + (G_y - S_y)^2} \tag{91}$$

If $0 < t < 1$, then the minimum distance between the point C and the line $\bar{l}$ falls into the segment delimited by the points S and C. In this case the distance can be computed with the normal equation of the distance between a point and a line showed in (92).

$$d = \frac{|(G_x - S_x)(S_y - C_y) - (G_y - S_y)(S_x - C_x)|}{\sqrt{(G_x - S_x)^2 + (G_y - S_y)^2}} \tag{92}$$

If $t < 0$ or $t > 1$ then the minimum distance between the obstacle centre and the $\overline{SC}$ segment is the minimum between the Euclidean distance between the obstacle centroid and the boundary point of the segment. Once the distance is computed, the closest obstacle to the obstacle position is chosen.

Figure 83 depicts the flow diagram of the algorithm for the decision of the obstacle to avoid.

Once all the paths planned reached the goal, the search for the leaf with the shortest path length is performed. Every leaf in the tree is identified with the id and level parameters which are used later to search the waypoints in the tree leading to the shortest path. The leaves id and level are showed in Figure 84.

The id is calculated and assigned to a leaf taking into consideration the id and the level of the parent:

$$x_i = (x_{i-1} \cdot l_{i-1}) + k \tag{93}$$

With

$$k = \begin{cases} 0 & left\ leaf \\ 1 & right\ leaf \end{cases} \tag{94}$$

Giving this relation, it is possible, given the id of the leaf with the shortest path length, to find the parent leaves and consequently all the waypoint composing the trajectory.

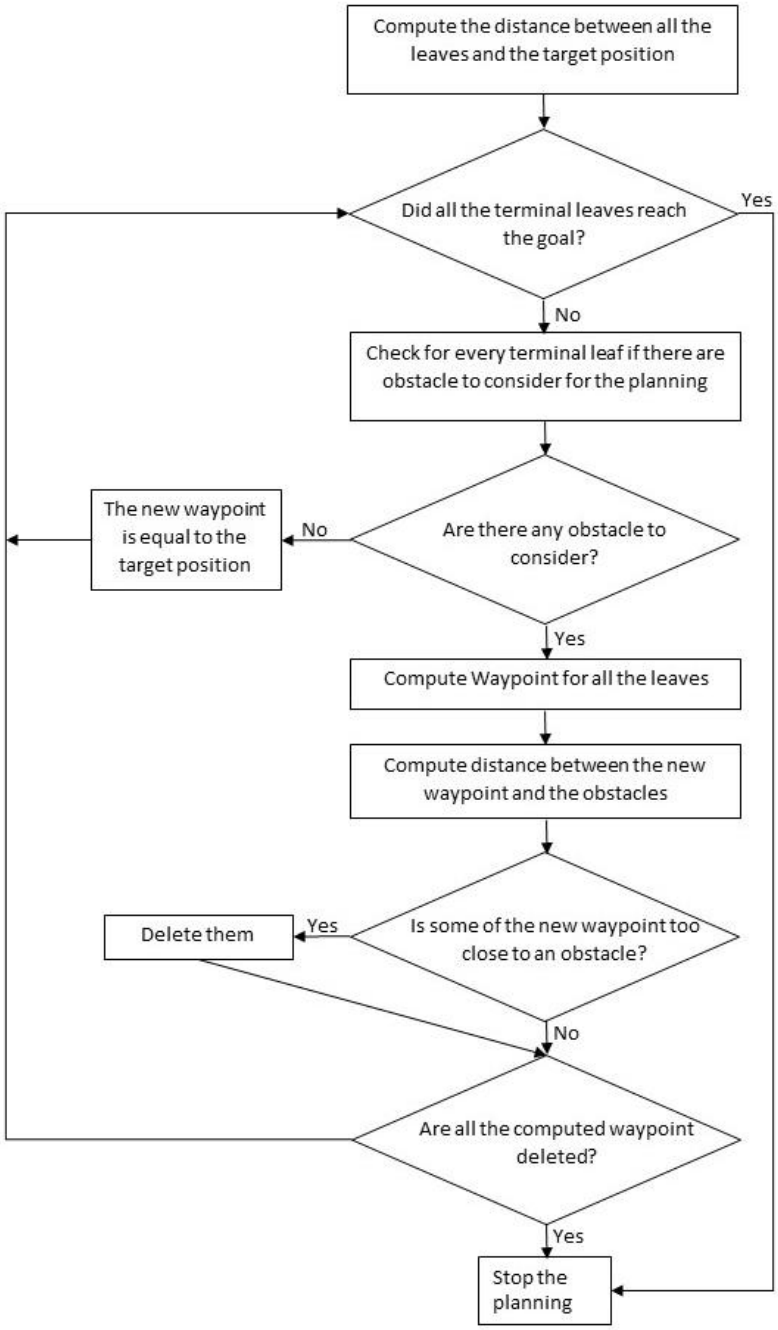

Figure 82: Flow diagram of the collision avoidance algorithm

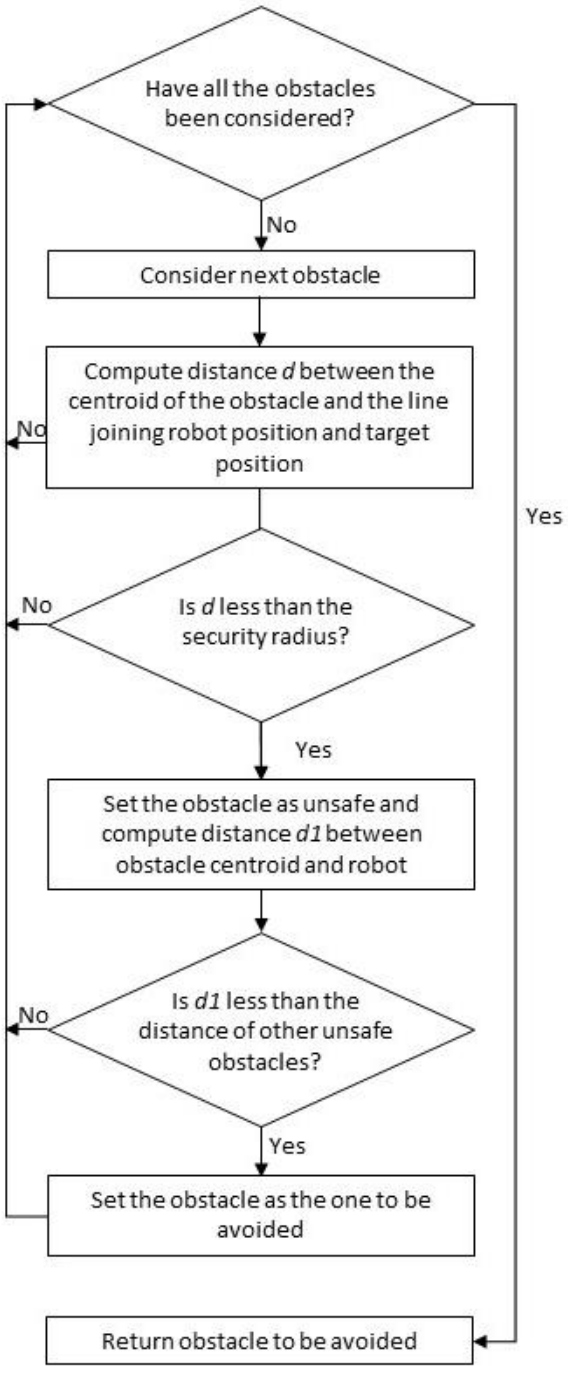

Figure 83: Flow diagram of the algorithm for the decision of the obstacle to avoid

The search of the shortest path is made possible by the incremental computation of the path length performed for every waypoint computation. Every time a new waypoint is computed, the length of the path between the waypoint stored in the parent leaf and the newly computed one is added to the length of the path of the parent leaf.

As a matter of fact, the tree stores also the information of the length of the path along with the id and the level of the leaf. First, a search is done on the tree for the leaf which reached the target with the shortest path length. Second, basing on the level and id of this leaf, all the waypoint contained in the leaves that leads to it are added to a vector.

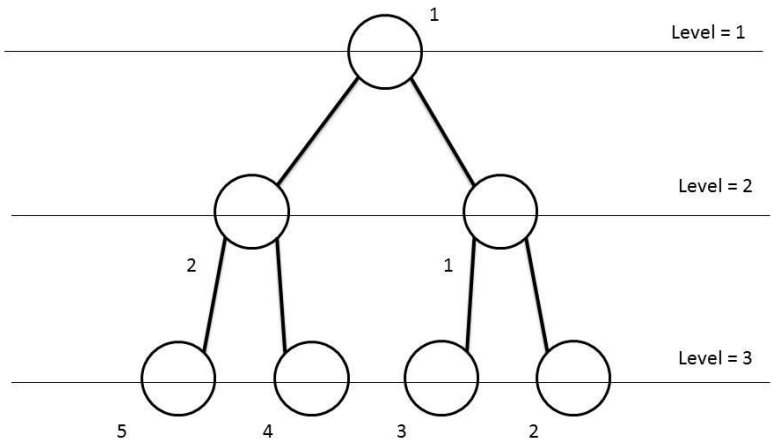

Figure 84: Level and id in the tree structure. The levels are written on the right side. The ids are the number next to the leaves (d'Apolito F. and C. Sulzbachner, 2017).

Given the relations (93) and (94), it is possible to define the algorithmic steps to accomplish such a task:

1. Initialize an empty vector of length equal to the level of the leaf with the shortest path.
2. Fill the first element of the vector with the id of the root of the tree (i.e. level equal to 1 and id equal to 1).
3. Fill the last element of the vector with the id and level of the leaf with the shortest path.
4. **For** all the other elements in the vector **do**:
   4.1. Compute the id of the leaf using equation (87) and (88) and insert it in the vector of ids.
5. **For** every element of the vector, starting from the second **do**:
   5.1. Check the id of the leaf stored in the vector.
   5.2. Using equation (87) and (88), check if the next waypoint is in the right leaf or in the right leaf.
      5.2.1. **If** left, **then**
         5.2.1.1. Go left and insert the waypoint in the vector of waypoints.
      5.2.2. **If** right, **then**

5.2.2.1. Go right and insert the waypoint in the vector of waypoint.

6. Return the vector of waypoint.

The flow diagram of the algorithm is depicted in Figure 85

The collision avoidance algorithm proved itself lightweight as the computation time required is always under 100 ms and a near-optimal path has always been found. Consider for instance the user-defined maps showed in Figure 86. In these cases, it was present in the map an elongated obstacle represented as a series of intersecting circles. As it is possible to see, the path output of the collision avoidance in these cases goes parallel to the longest dimension of the obstacle, proving that such a representation can correctly represent the obstacles without obstructing the free space.

In order to prove the capabilities of the collision avoidance algorithm a few other tests are showed in Figure 87 to Figure 95 for user generated maps with an increasing number of obstacles. For all the test cases presented the start position is (0,0).

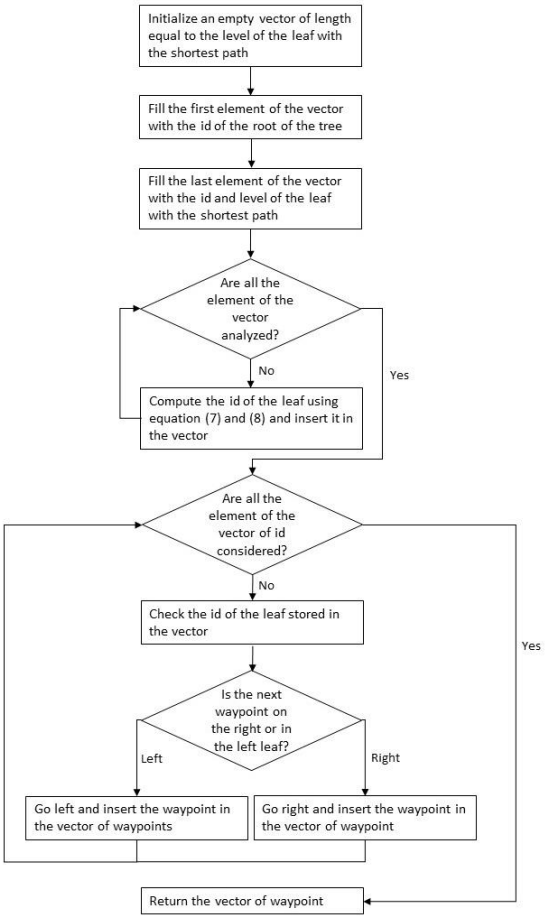

Figure 85: algorithmic steps for the search of the waypoint belonging to the shortest path

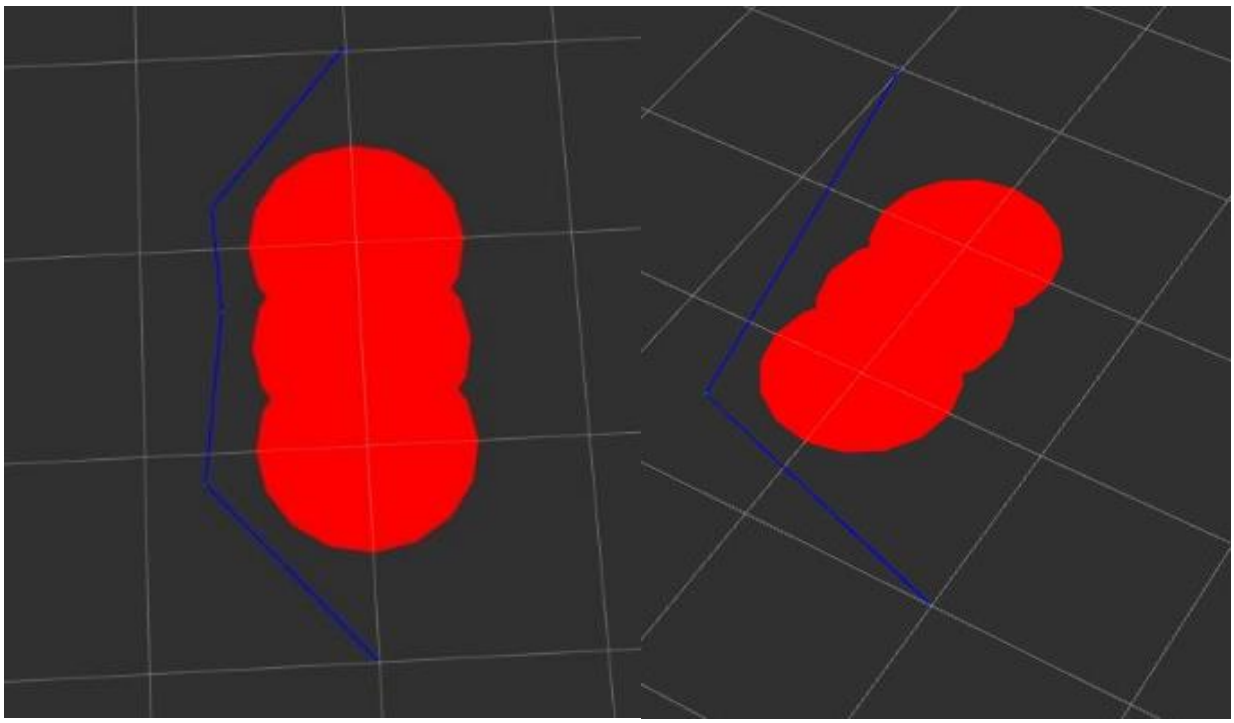

Figure 86: First and second test case on the collsion avoidance algorithm

The map in Figure 87 was created with obstacles in position (1,1), (1.5,1), (2,1), (0,3) and (3,0). The starting position is (0,0) and the target position is (4,0). The map in Figure 88 instead has the obstacles in position (1,1), (1.5,1), (2,1), (0,3). The target position is set as (2,3).

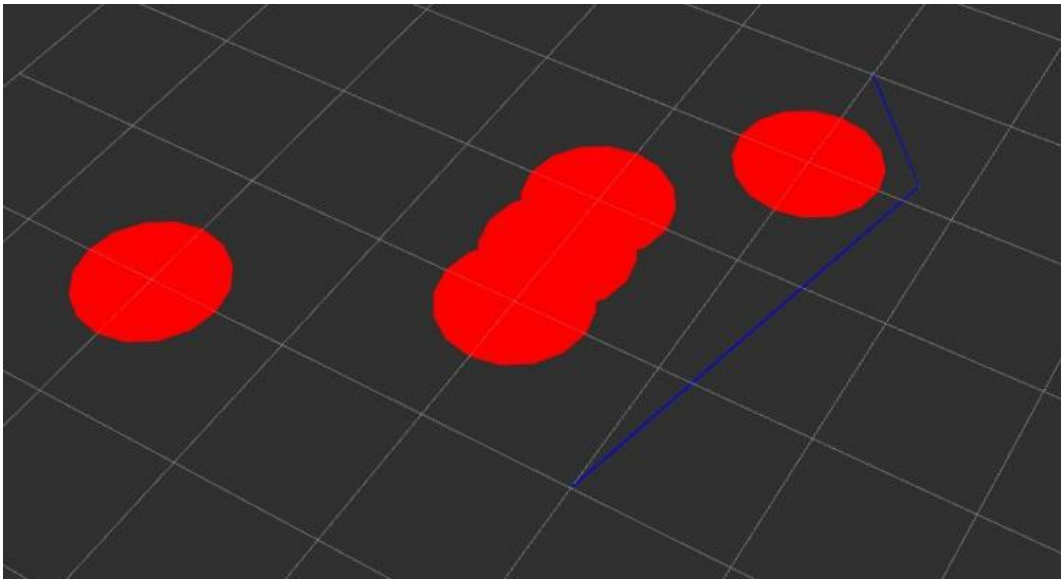

Figure 87: Third test case for the collision avoidance algorithm

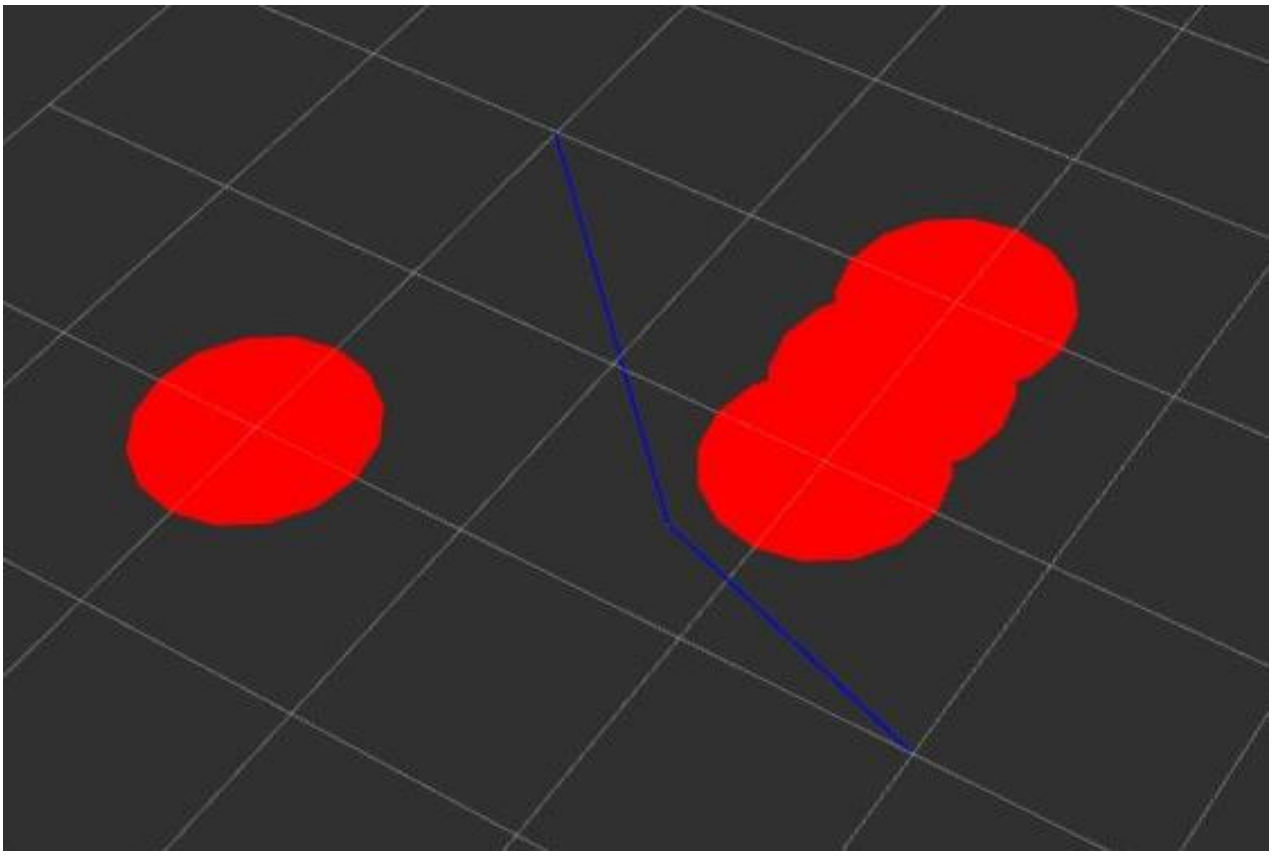

Figure 88: Fourth test case for the collision avoidance algorithm

In Figure 89, the obstacles' positions in the map are (1,1), (1,1.5), (1,2), (0,3), (2,2), (2,3) and (4,2). The target position is set to (4,4). In Figure 90, instead, the obstacles are in the positions (1,1), (1.5,1), (3,1), (3,2), (2,2.5), (-0.5,1.5). The target position is set to be (4,3).

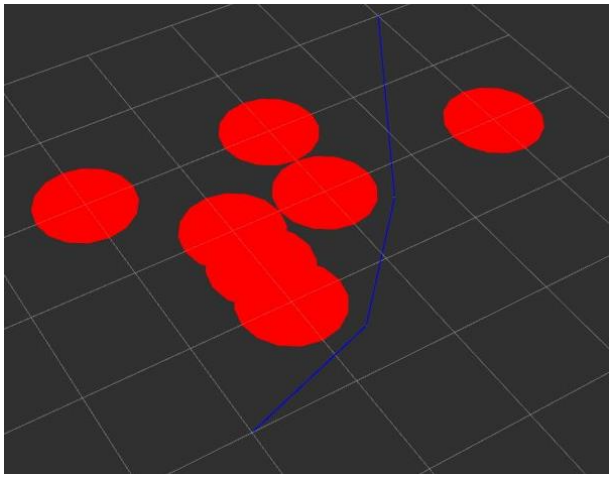

Figure 89: Fifth test case for the collision avoidance algorithm

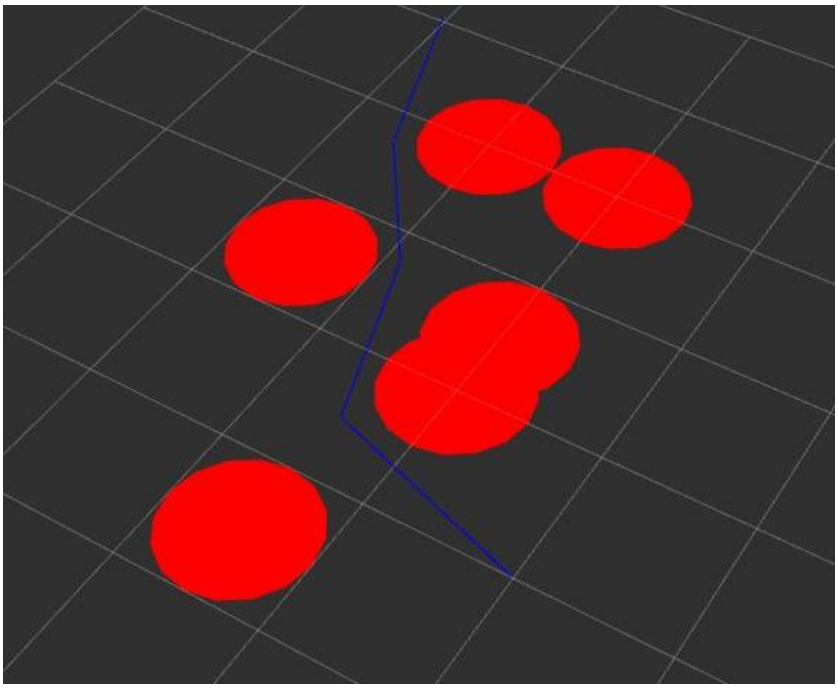

Figure 90: Sixth test case for the collision avoidance algorithm

In the test case of Figure 91, the obstacles are in position (1,1), (1.5,1), (2,1), (2,2), (3,2), (2,4), (3,0), (0.5,-1), (1,-1), (1.5,-1). The target position is set to (5,2).

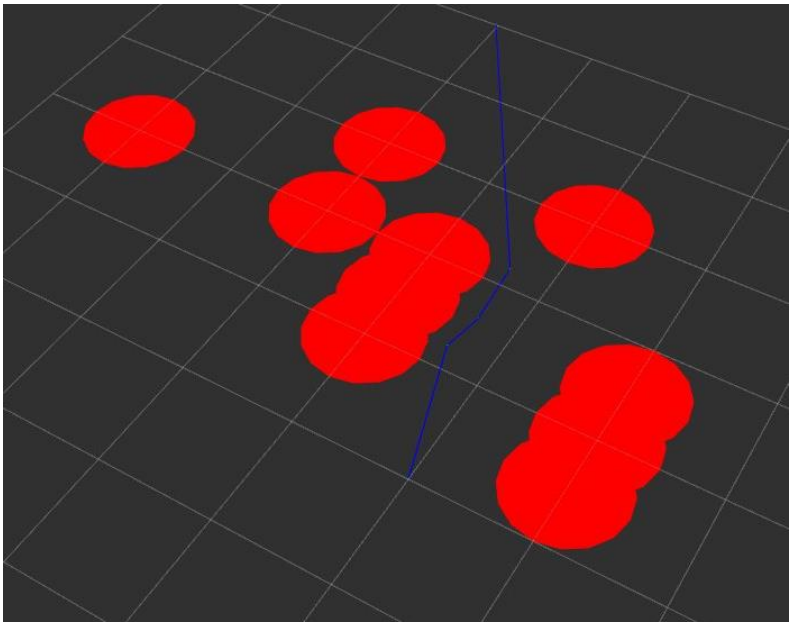

Figure 91:Seventh test case for the collision avoidance algorithm

In Figure 92, Figure 93, Figure 94 and Figure 95 the obstacles' positions are the same and the generated map is the same as in Figure 91 with the addition of the obstacle in position (0,-2). In Figure 92 the target position is (4,2), in Figure 93 is (1,-3), in Figure 94 is (5,1) and in Figure 95 the target position is set to (3,5).

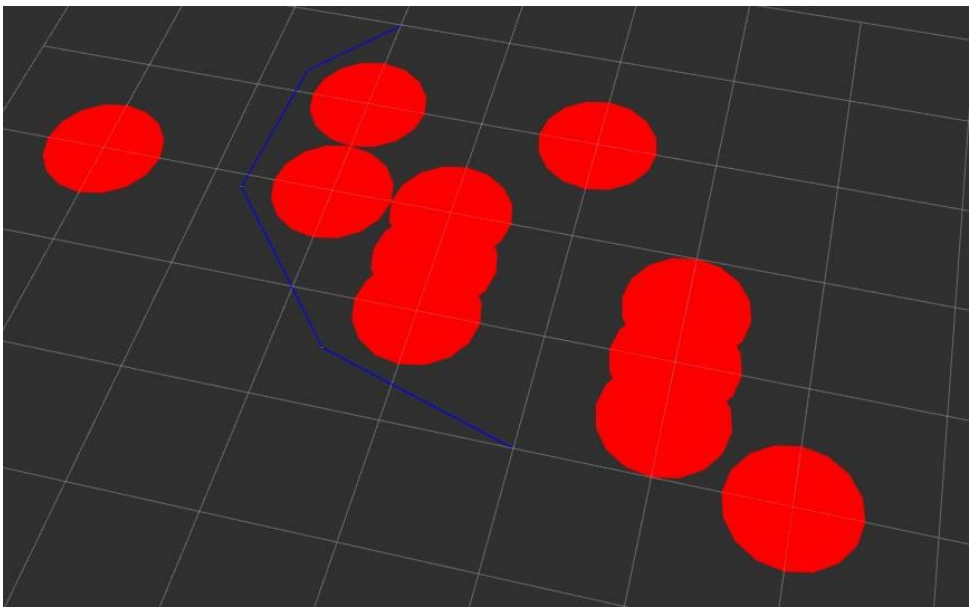

Figure 92:Eighth test case for the collision avoidance algorithm

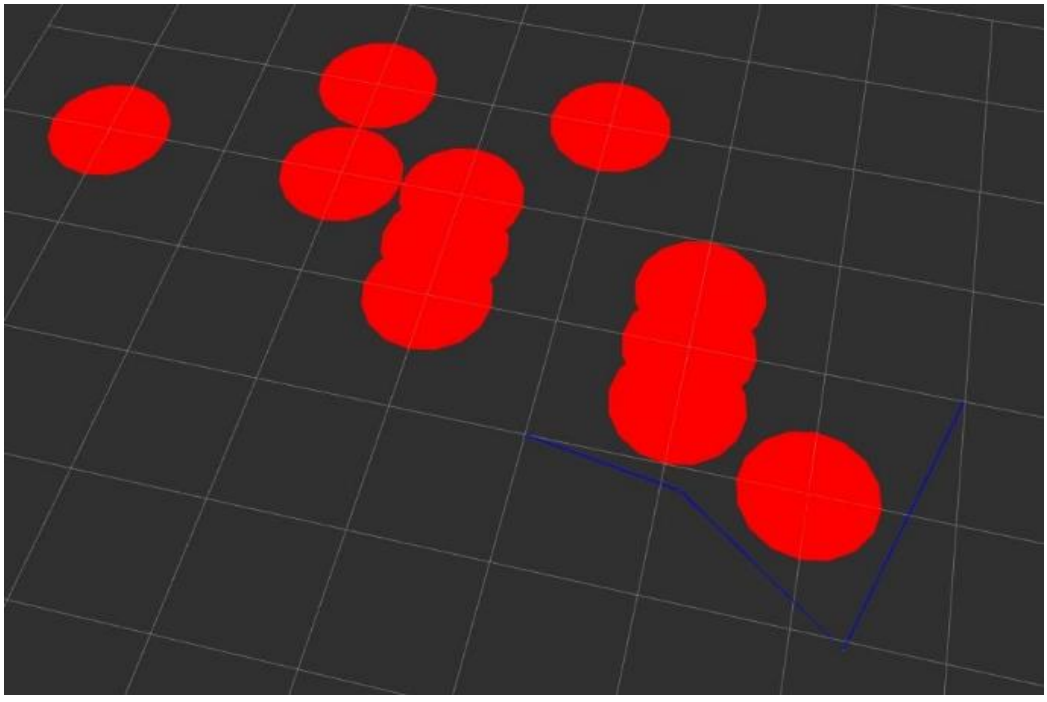

Figure 93: Ninth test case for the collision avoidance algorithm

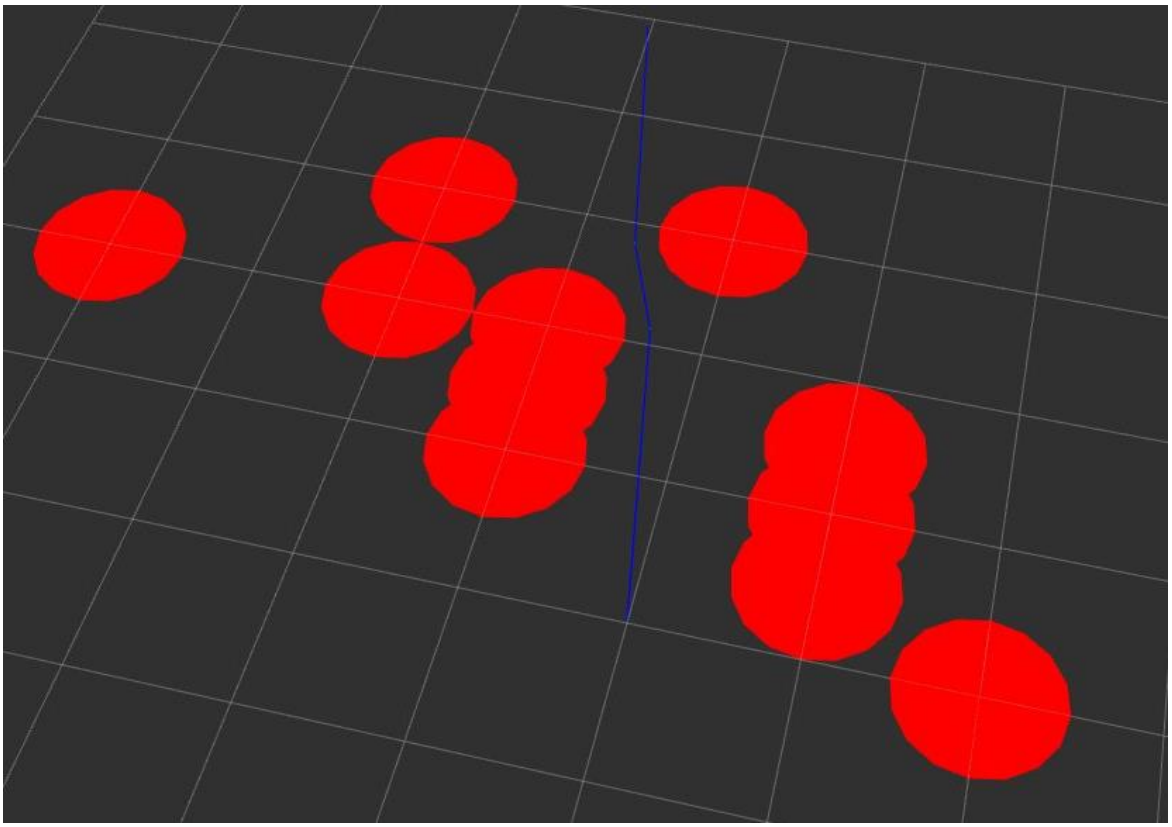

Figure 94: Tenth test case for the collision avoidance algorithm

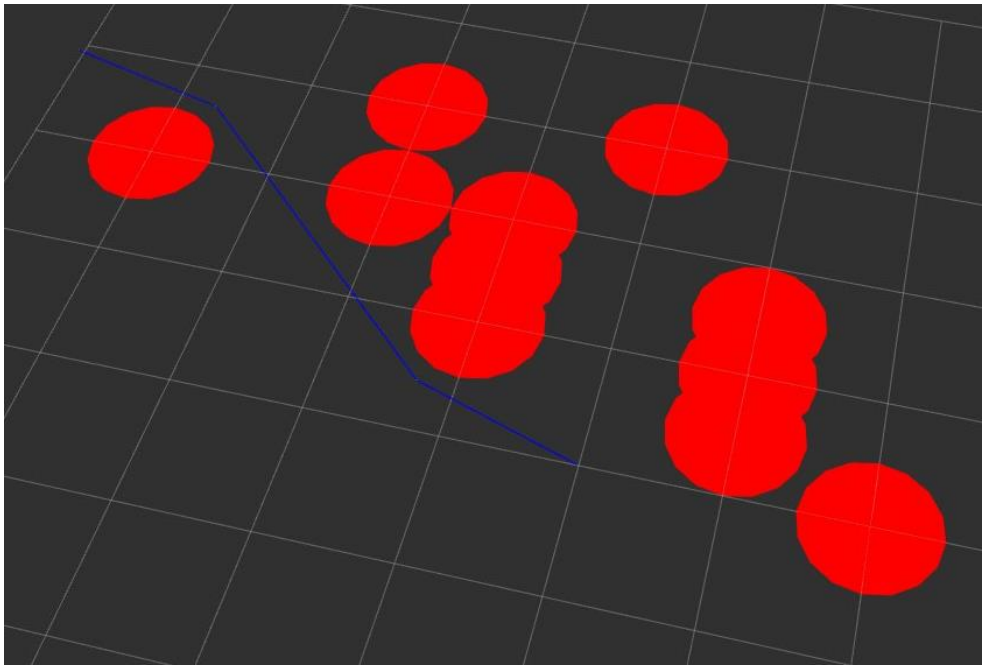

Figure 95: Eleventh test case for the collision avoidance algorithm

In all the test cases presented the obstacle avoidance was always successful in finding a near-optimal path even in cases of cluttered environments. Something interesting can be noted in Figure 93. In this test case, the obstacle in position (0,-2) is avoided with a not-optimal path. The planned trajectory first goes almost half a meter away from the obstacle and then comes back towards the target. This results from the way the points composing the trajectory are computed from the algorithm. Such a waypoint is, as a matter of fact, the intersection point between the tangent to the obstacle starting from the previous waypoint and from the target position. This may require some adjustments of the algorithm for this particular type of scenario.

6.3.3. The footstep computation

The next step of the footstep planning and collision avoidance strategy is the footstep computation. As described in (Kajita et.al., 2014), the foot placement during a walk can be defined by the walk parameter i.e. step width and length, respectively s_y and s_x. Consider for instance, the walk parameter showed in Table 9.

Step number	1	2	3	4	5
Step length s_x	0.0	0.4	0.4	0.4	0.0
Step width s_y	0.25	0.25	0.25	0.25	0.25

Table 9: step width and step lenght for 5 consecutive steps

Step width and step length are called walk parameters. From their knowledge it is possible to derive all the footstep coordinates during the walk. As a matter of fact, the n^{th} footstep can be calculated from the walk parameter as:

$$\begin{bmatrix} p_x^{(n)} \\ p_y^{(n)} \end{bmatrix} = \begin{bmatrix} p_x^{(n-1)} + s_x^{(n)} \\ p_y^{(n-1)} - (-1)^n s_y^{(n)} \end{bmatrix} \tag{95}$$

For changing the direction of motion, one just need to introduce one walk parameter s_θ which indicates the heading of the feet.

The computation of the footstep position becomes:

$$\begin{bmatrix} p_x^{(n)} \\ p_y^{(n)} \end{bmatrix} = \begin{bmatrix} p_x^{(n-1)} \\ p_y^{(n-1)} \end{bmatrix} + \begin{bmatrix} \cos s_\theta^{(n)} & -\sin s_\theta^{(n)} \\ \sin s_\theta^{(n)} & \cos s_\theta^{(n)} \end{bmatrix} \begin{bmatrix} s_x^{(n)} \\ -(-1)^n s_y^{(n)} \end{bmatrix} \tag{96}$$

Equation (96), introduced in (Kajita et.al., 2014), was adapted for Archie in the course of this PhD work, since the reference system of the map is centred on the ground projection of the centre of mass initial position. First, the position of the centre of mass is computed as follows:

$$\begin{bmatrix} c_x^{(n)} \\ c_y^{(n)} \end{bmatrix} = \begin{bmatrix} c_x^{(n-1)} \\ c_y^{(n-1)} \end{bmatrix} + \begin{bmatrix} \cos s_\theta^{(n)} & 0 \\ 0 & \sin s_\theta^{(n)} \end{bmatrix} \begin{bmatrix} s_x^{(n)} \\ s_x^{(n)} \end{bmatrix} \tag{97}$$

With the centre of mass position, it is possible to calculate the footstep position with the relation (98).

$$\begin{bmatrix} p_x^{(n)} \\ p_y^{(n)} \end{bmatrix} = \begin{bmatrix} c_x^{(n)} \\ c_y^{(n)} \end{bmatrix} + \begin{bmatrix} \sin s_\theta^{(n)} & 0 \\ 0 & \cos s_\theta^{(n)} \end{bmatrix} \begin{bmatrix} (-1)^n s_y^{(n)} \\ -(-1)^n s_y^{(n)} \end{bmatrix} \tag{98}$$

As written in (d'Apolito et.al., 2016) the procedure for computing the footsteps' positions from the waypoint list can be put in algorithmic form by making the following assumptions:

- Archie starting pose is characterized by having its feet next and parallel to each other.
- The first footstep is the right one.
- A turn is done "on the spot", i.e. without advancing (d'Apolito, 2018)

Based on this assumption, the footsteps are computed as follows:

1. Set the first footstep position to be half the robot's hip distant on the y axis from the origin. Sets its heading to 0 and set also all its walk parameters to 0. Store, in the walk parameter vector, the walk parameter of the first step.
2. Set the second walk parameter in order to move along the y axis. Set its heading to 0 and push it in the walk parameter vector.
3. Compute the length of the waypoints' vector.
4. **For** every set of two consecutive waypoints **do**:
   4.1. Compute the distance between the two considered waypoints.
   4.2. Compute the number of footsteps necessary to move from the first waypoint to the second.
   4.3. Compute the heading of the line between the two considered waypoints.
   4.4. **If** the calculated heading is different from the heading of the line between the set of waypoints **then**:
      4.4.1. Set the step length s_x of the next two footsteps to 0 and the step width s_y of the next two footsteps to the maximum step width. The step heading s_θ of the next two footsteps, must be set as the heading of the line between the two waypoints. Push them finally in the vector of walk parameters.
   4.5. **For** each walk parameter **do**:
      4.5.1. Set the step length s_x to the maximum step length. Set s_y to the maximum step width and s_θ as the computed heading and push them in the walk parameter vector.
   4.6. Set the step length s_x of the last footstep as 0. Set the step width s_y of the last footstep as the maximum step width and the step heading s_θ as the computed heading.
5. Calculate the length of the walk parameter vector.
6. **For** each walk parameter in the vector **do**:
   6.1. Compute the footsteps using equation (92) and push it in the footstep vector.
7. Return the footsteps vector.

The flow diagram of this algorithm is shown in Figure 96. After the positions of the feet are computed, they are given as input to the walking pattern generator which computes the trajectory that the COM of the robot will have to follow during the walk between the two given positions.

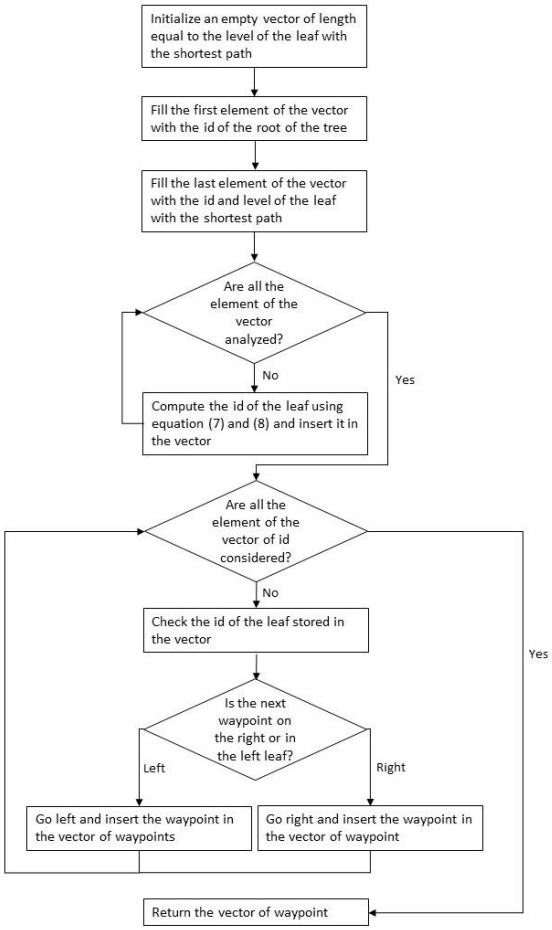

Figure 96: flow diagram of the computation of the footsteps

Figure 97 to Figure 106 show the results of the footstep computation for the user generated map showed in the previous chapter.

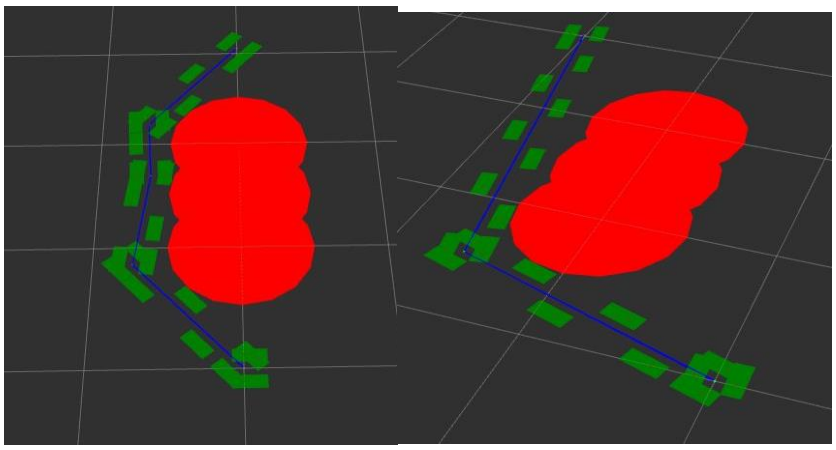

Figure 97: Footstep computation for first and second test case

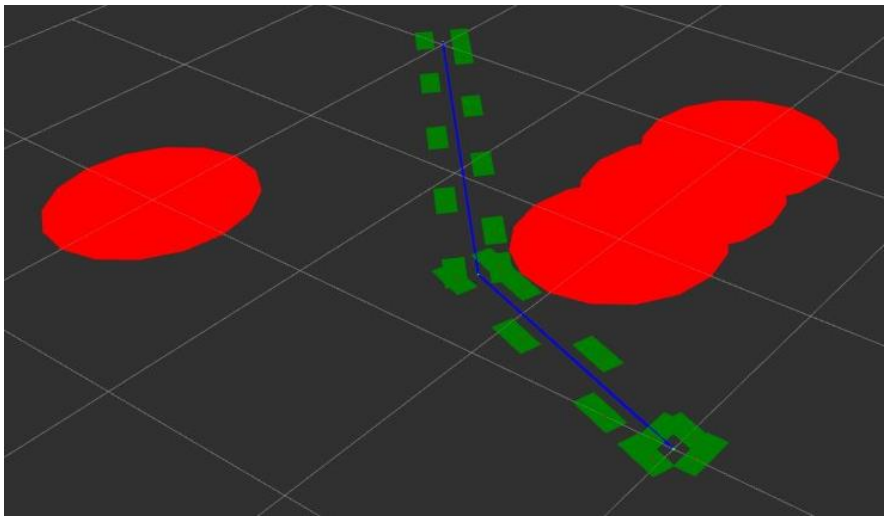

Figure 98: Footstep computation for third test case

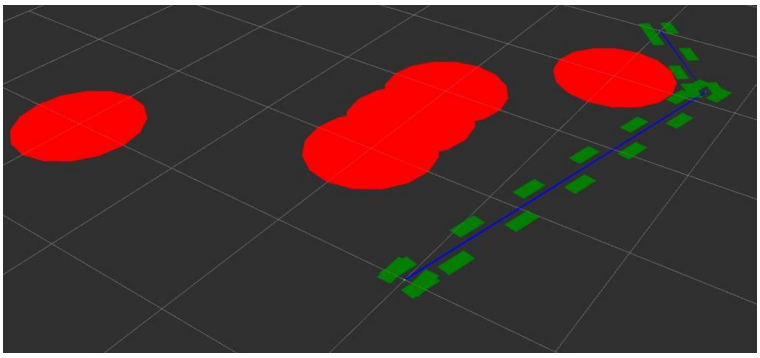

Figure 99: Footstep computation for the forth test case

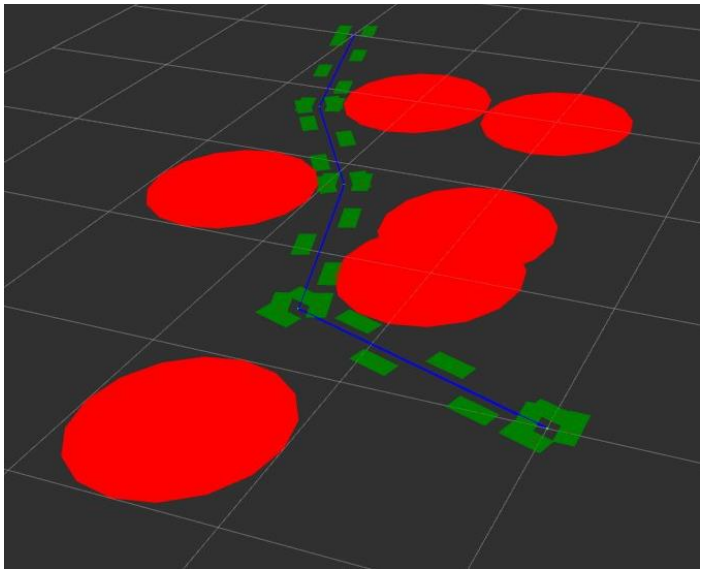

Figure 100: Footstep computation for the fifth test case

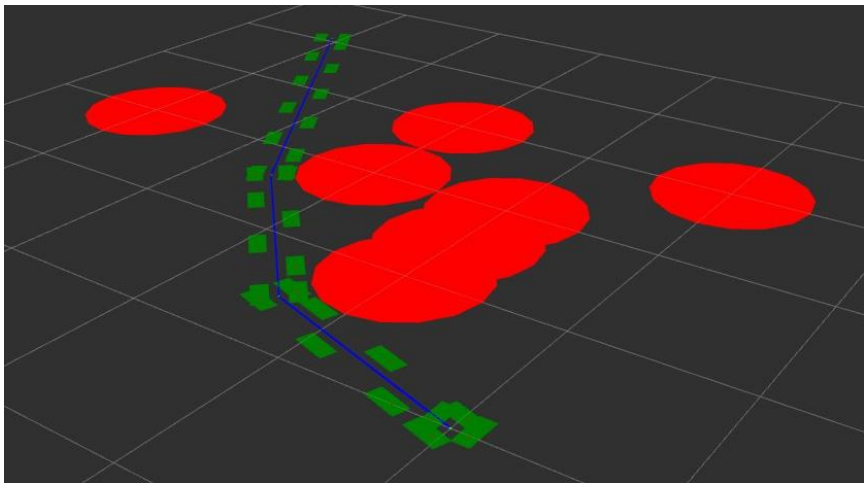

Figure 101: Footstep computation for the sixth test case

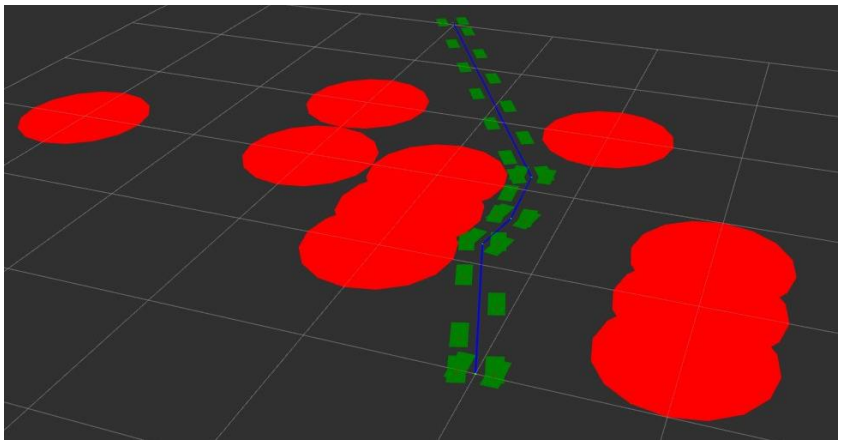

Figure 102: Footstep computation for the seventh test case

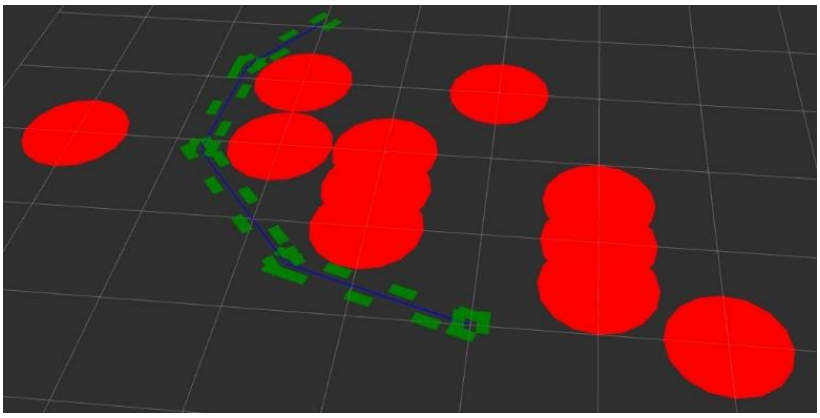

Figure 103: Footstep computation for the eight test case

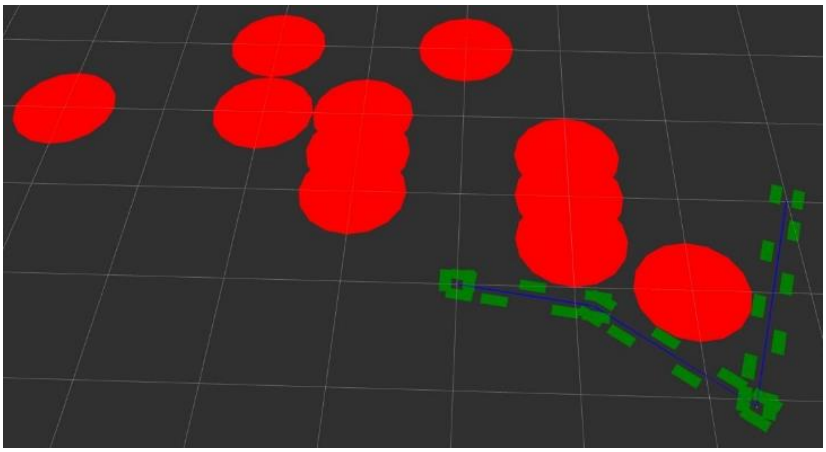

Figure 104: Footstep computation for the ninth test case

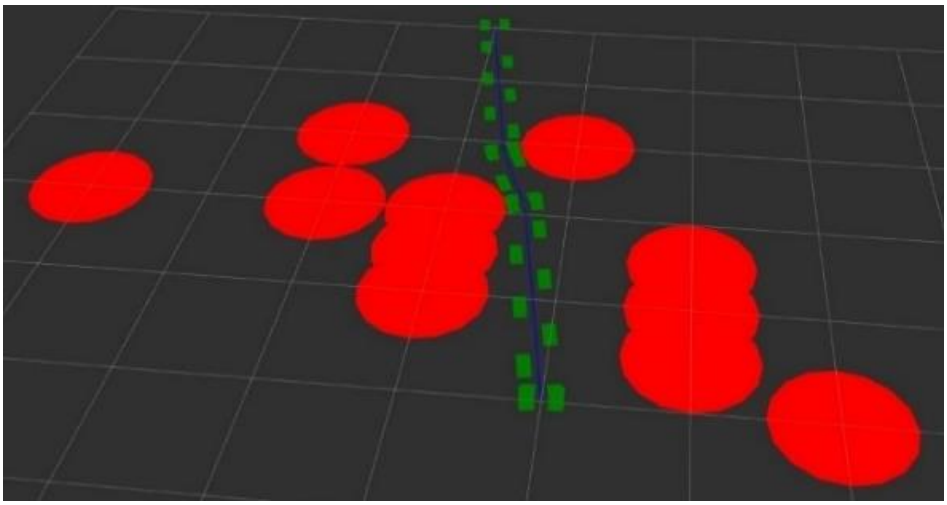

Figure 105: Footstep computation for the sixth test case

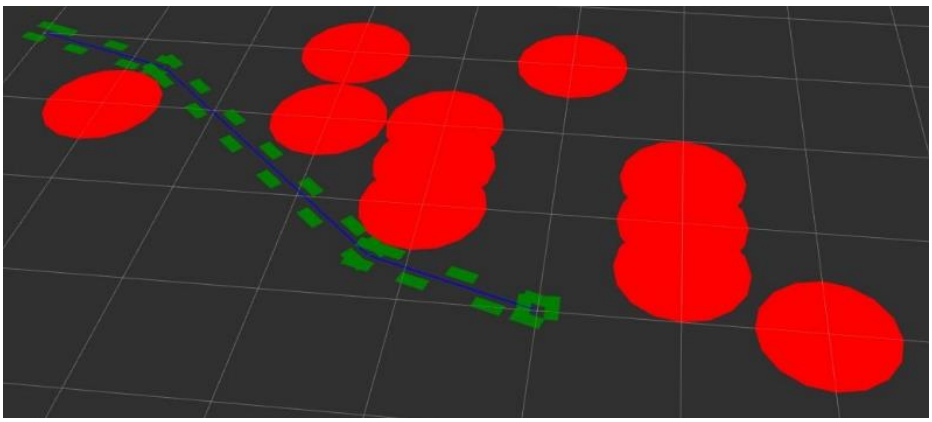

Figure 106: Footstep computation for the eleventh test case

The test showed the developed approach was lightweight. The decision to increase the obstacle dimension of the dimension of half-length of Archie's hip proved itself to be the best one for its computational simplicity and for its performance. No footstep was ever found inside the area of an obstacle. This is because the trajectory is always parallel to the obstacle.

It has to be noted that, currently, the direction changing is done on the spot in one step. It can be considered acceptable for small direction changing but this approach will need to be calibrated for bigger direction changes when the new control system of the robot will be available.

6.4. Collision avoidance tests with real scenario maps

The whole new software architecture was tested together on the on-board computer and with the stereo-camera in order to evaluate its real time performance.

The computational load of all the modules together was too high for the board. For this reason, some nodes of the software had to be executed on an external laptop connected to the wireless network dedicated for the robot. The major part of the computational power of the on-board computer was reserved for the camera driver which has to separate the left to the right image for a 30 fps video stream. Although the stereo-PTAM was chosen for the low computational power it requires, it was the second most heavy application. However, the distribution of the computational load between the on-board computer and an external laptop, make a real time usage of the software possible. Figure 107 and Figure 108 show the result of the footstep planning for the map created by the obstacle detection showed in Figure 76 and Figure 77

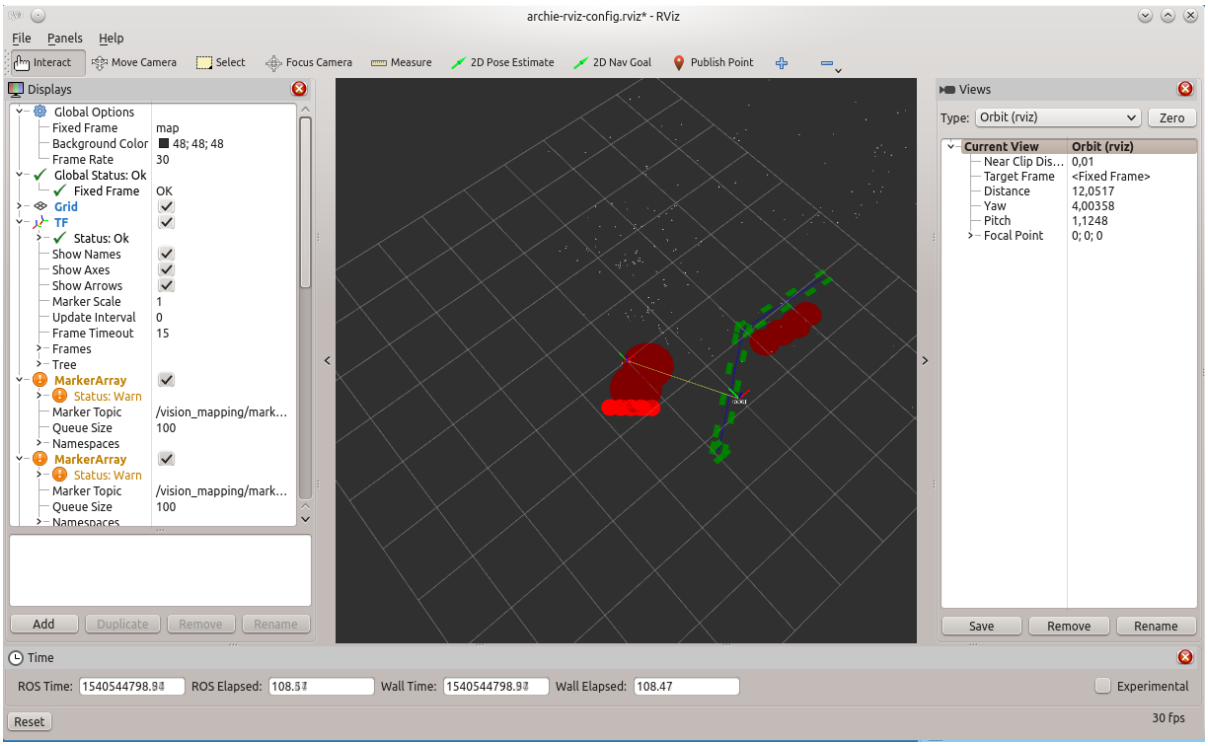

Figure 107: footstep planning for the first detection example

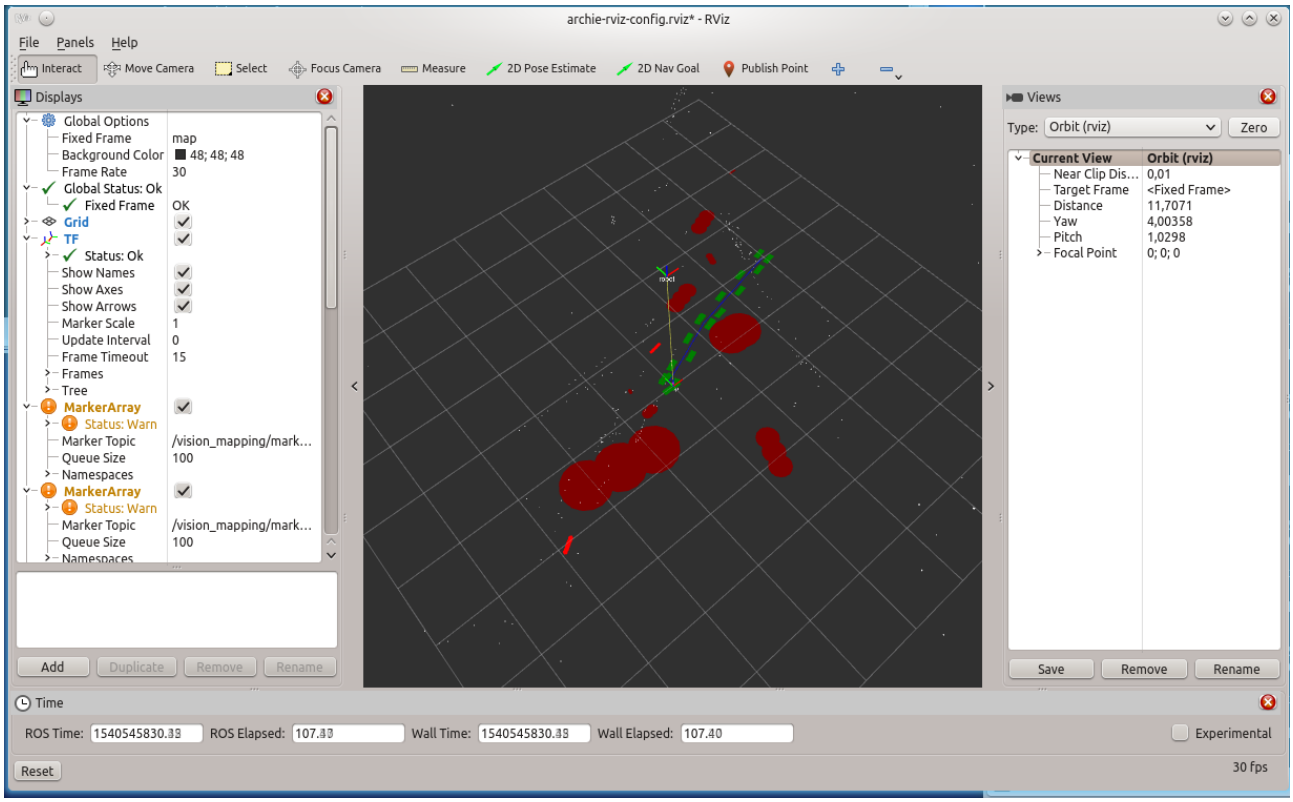

Figure 108: Footstep planning for the second detection example

7. Conclusion and future work

In this PhD dissertation, it is described the development of a footstep planner for the cost oriented humanoid robot Archie. Archie is currently in development at the Technical University of Vienna and it currently has basic human-like walking capability.

For the development of a footstep planner, the environment sensing system, i.e. the system composed of the sensors which analyse the working environment of the robot, has a particular importance. Many humanoid robots currently developed have environment sensing systems composed of many sensors in order to retrieve as many data as possible describing the work environment of the robot.

The humanoid robot ASIMO for instance, has a complex environment sensing system composed of visual sensors in the head and laser sensor on the hip pointed to the ground in order to detect obstacles up to 2 meters from the robot. Furthermore, ASIMO's environment sensing system is composed of ultrasonic sensor on the front and on the back of the robot. The humanoid robot ATLAS instead, is equipped with stereo-cameras and laser range finders. THORMANG 3 from Robotis is equipped with cameras and LIDAR. Even a commercial toy robot such as NAO is equipped with several sensors in the head which allow it to perceive the environment around him.

Currently only two footstep planners for humanoid robots are available open source. The first one is the one described by (Garimort et.al., 2011) and in (Hornung et.al., 2012). The planner in question is based on a grid map representation of the environment. This representation is then used, in the first implementation of the footstep planner, by the D* algorithm to find a collision-free path between the start and the goal. In the second version, the ARA* is used in order to improve the performance of the footstep planner in case a re-planning is necessary due to modifications in the environment around the robot.

The second footstep planner is the one described in (Stumpf et.al., 2016). It is based on the footstep planner by (Garimort et.al., 2011) but it has an improved implementation in order to make it more adaptable to various kind of humanoid robot platform. For the world perception they also use a grid map approach.

In the beginning of the development, as described in (Byagowi, 2010) Archie was a full 31 DOF humanoid robot in which upper and lower body were developed. Currently only the lower body is

built i.e the two legs, the hip and a torso. All the joints are actuated by brushless and brushed DC motors and every motor is connected to an embedded controller. The control strategy is, thus, decentralized. Furthermore, due to the lack of an on-board computer, the Linux software implemented for operating the robot runs on an external laptop. No sensors for observing the environment or for the balance are present.

The advanced control algorithm under evaluation receives as input the feedback from the motors and from an IMU mounted on the hip. It aims to correct deviation from a stable gait trajectory computed with symmetry relations between the Denavit-Hartenberg parameters of the joints.

In order to pass from a decentralized control strategy to a centralized one, i.e in order to develop an advanced control algorithm for the robot, the addition of an on-board computer is necessary. The on-board computer has to be connected with all the motors and the sensors in order to receive the necessary input data. On this on-board computer, also the footstep planner developed for this thesis can run in order to increase the level of autonomy of the robot.

The on-board computer selected for the robot is the Nvidia Jetson TK1, chosen for its high computational power compared with the price. The Ubuntu OS runs on it and it has full ROS compatibility. Furthermore, a USB hub and a Wi-Fi dongle were selected. The USB hub was added because the Nvidia Jetson TK1 has only one USB port and more were necessary. The Wi-Fi dongle was added in order to allow the connection between the robot and an external computer through a router dedicated for Archie. This router creates a local network where both the robot and an external computer can connect allowing them to exchange data between each other and, more generally, to divide the computational load between multiple machines.

The stereo-cameras chosen for Archie are the LI-USB30-V024STEREO form Leopard Imaging. Their dimension, cost orientation and compatibility with the Jetson TK1 made them the best candidate. They are connected to the on-board computer via USB 3.0.

The IMU sensors for the measurement of the attitude of the hip and the new embedded controllers in development were also shortly described in this dissertation. They will all be connected to the on-board computer via serial connection.

There are three main challenges in the development of a footstep planner for Archie. The first one is the cost orientation which sets a limit to the number of sensor and thus to the density of the data describing the environment.

Secondly, it must be considered that Archie is supposed to be a research-oriented application. The main aim of the development of the robot is to make it a test bed for advanced control algorithm. One possible advanced control algorithm is currently in evaluation and changes are still possible. For this reason, the software must be implemented in a modular way in order to make the substitution or the removal of one of the modules possible without having to re-implement the whole software.

The third challenge is the computational load of the software. In order to increase the autonomy level of the robot, the software should run on the new on-board computer. Thus, the computational load of the software should be kept as low as possible.

The footstep planner has been developed using the ROS framework. The main components of the developed footstep planner are:

- The camera manager.
- The vision and mapping node.

The camera manager is the component responsible for the connection with the stereo-cameras and therefore, provides the driver functionality for the cameras and a first pre-processing of the incoming data. The data coming from the cameras, as a matter of fact, store left and right image in the same data structure. Using open source tools such as the one provided by OpenCV for grabbing the image would result in having useless data. Thus, the camera manager divides the incoming data in left and right images before any other data processing is performed.

The left and right images are then sent to two different image_proc nodes for the rectification. The image_proc package is an image rectification ROS tool. The rectification can be described as the process by which the distortions in an image are removed and, in the stereo-camera case, the image centres are aligned.

The rectified images are then sent to the stereo-PTAM node which analyses the frame and output the position of the robot in the environment and the sparse point cloud used for the tracking. These data are then the input of the vision_mapping node. The main characteristics of the PTAM algorithm is the parallelization of the tracking and the mapping thread which makes it less computational expensive than other SLAM approaches.

The vision_mapping node is the core of the developed footsteps planner. It provides the obstacle detection, the collision avoidance algorithm and the footsteps computation. The obstacles are detected using the ground plane approach which is considers as obstacles everything with a positive vertical coordinate. The inverted cone algorithm was evaluated too but, in order to keep the computational load of the software low, it was decided to use the ground plane approach.

Contrary to the two footstep planner available online, the footstep planner developed for Archie is not based on a grid map. The map is built approximating the obstacles as circles. Such a representation choice was done in order to keep the computational load low. As a matter of fact, the circle representation allows the use of simple geometrical planning algorithm. Furthermore, the grid representation can become heavy in terms of memory storage and collision-free path search in case of big environments.

However, one drawback of the circle representation though, is the representation of elongated obstacles. As a matter of fact, the circle representing an elongated obstacle would have a radius equal to the maximum dimension of the obstacle. This would occupy much of the free space around the object. In order to solve this inconvenience, elongated obstacles are represented as a series of intersecting circles with centres along the maximum dimension of the obstacle.

The computation of a collision free trajectory for a humanoid robot usually is a 3D path planning algorithm which computes directly the position of the footsteps as well as the trajectory of the COM. In the footstep planner developed for Archie, a collision avoidance algorithm computes first a collision free trajectory between the starting position of the robot and the target position and, then computes the footsteps with a simple geometrical algorithm. Such an implementation was chosen because Archie is supposed to be used as a test bed for the implementation of advanced control algorithms. The design of the footstep planner had to be flexible and modular.

The developed collision avoidance algorithm is a geometrical algorithm which computes the avoidance waypoint as the intersection of the two sets of tangents to circular obstacles from the starting position of the robot and from the target position. The obstacles' radius is previously increased by half the hip dimension of the robot. The waypoints computed are then stored in a binary tree structure along with the length of the path. This structure is later used to choose the shortest path to the target position. If more than one obstacle is present, the obstacle to avoid is chosen depending on the distance between the centre of the obstacle and the segment joining robot position and end position. Furthermore, the distance of the obstacle centre from the robot

position is considered. The planning stops when at least one of the leaves of the tree reached the target position. To every leaf in the tree is associated an identifier with a known mathematical relation in order to make the search of the shortest path easier.

The collision-free trajectory outputted by the algorithm described is the safe trajectory that the COM of the robot has to follow. From this trajectory, the footstep positions are computed by means of geometrical relations and of the walk parameter which describes step heading, step width and step length.

The developed footstep planner uses ROS tools in order to interact with an operator. As a matter of fact, the image_view package is used to visualize the stream of the stereo-cameras and RViz is utilized to visualize the point cloud, the robot positions, the circular obstacles, the collision-free trajectory planned and the footsteps computed. Through RViz, the operator can also set the target position for the robot to reach while avoiding the obstacles detected. In order to test the collision avoidance algorithm, the operator can also define a new map or clear the current map by means of an implemented ROSService.

All the functionalities of the software and the single modules were tested. The camera manager node works without issues and it is able to stream a 30 fps video from the cameras with a resolution of 640x480.

The obstacle detection and the subsequent circle approximation work. However, the point cloud output from the stereo-PTAM is sparser than it was expected and too much depending on the lighting condition. This is most probably due to the small baseline of the stereo-camera.

The collision avoidance algorithm and the subsequent footstep computation also works without any complications. The computations are fast (less than 100 ms of computational time required) and the planned path is close to optimal. Such a low computational time makes the algorithm also usable in case of non-static environment. In case modification in the environment are detected, a re-planning can be issued. It has to be noted that the representation of the elongated obstacles as a series of intersecting circles proved itself to be reliable as the planned trajectory is parallel to the maximum dimension of the obstacle. The design choice to increase the dimension of the obstacles with half the hip length of the robot was also a good choice because, since the trajectory is always tangent to the obstacles, it was never found a planned footstep inside the area occupied by the obstacles.

The software was also tested live in order to evaluate the real time capabilities of the new software architecture. The division of the data coming from the camera into left and right images is the most computationally expensive operation of the software. The stereo-PTAM is the second most computational expensive. The computational load of the camera manager and of the stereo-PTAM made the execution of the vision_mapping node od of RViz on an external laptop necessary.

As future work, in order to make everything run on-board the robot, the camera manager operations should be moved on the GPU of the board in order to leave more of the computational capacity of the CPU free. Moreover, it is also advised to mount another board in parallel since the advance control algorithm node has to run.

From the detection point of view, as mentioned before, the point cloud is too sparse. Other SLAM algorithms should be tested in order to see if it is possible to have a denser point cloud. In alternative, the stereo-cameras needs to be integrated with other sensors, such as a LIDAR for instance. Furthermore, with a denser point cloud, a more robust method for approximating the obstacles into circles may be required. A possible more robust method is to compute the median line of the point cloud as described in (Huang et.al., 2013) and then set the centres of the intersecting circles along the medial line.

References

Arbulú, M.; Kaynov, D.; & Balaguer, C. (2010): *The Rh-1 full-size humanoid robot: Control system design and Walking pattern generation.* Proceedings of the 13th International Conference on Climbing and Walking Robots. InTech.

Arduino dimensions Adafruit.https://blog.adafruit.com/2011/02/28/arduino-hole-dimensions-drawing/ - last retrieved 2018.

Arduino Due. https://store.arduino.cc/usa/arduino-due - last retrieved 2018.

Arduino Micro. https://store.arduino.cc/usa/arduino-micro - last retrieved 2018.

A algorithm Stanford website description*. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html - last retrieved 2017.

Bauer, J.; Kopacek, P.; d'Apolito, F.; Kraeuter, L.; Dorna, I. (2015): *Cost Oriented (Humanoid) Robots*. Proceeding of the 16th International Conference on "Technology, Culture and International Stability", September 24-27, 2015, Sozopol, Bulgaria. Published by Elsevier ScienceDirect IFAC-PapersOnLine, Volume 48 (2015), Issue 24; pp. 173-177. N.Y DOI 10.1016/j.ifacol.2015.12.078.

Boston Dynamic website Atlas description. http://www.bostondynamics.com/robot_Atlas.html - last retrieved 2017.

Byagowi, A. (2010): *Control System for a Humanoid Robot*. PhD Dissertation, TU Wien.

Chitta, S.; Marder-Eppstein, E.; Meeussen, W.; Pradeep, V.; Rodríguez Tsouroukdissian, A.; Bohren, J.; Coleman, D.; Magyar, B.; Raiola, G.; Lüdtke, M.; Fernandez Perdomo, E. (2017): *ros_control: A generic and simple control framework for ROS*. The Journal of Open Source Software, pp.456-456. DOI 10.21105/joss.00456

Correl, N. (2011): *Introduction to Robotics #4: Path-Planning*. http://correll.cs.colorado.edu/?p=965 - last retrieved 2017.

cv_bridge online documentation: http://wiki.ros.org/cv_bridge - last retrieved 2018.

Daniali, M. (2013): *Walking Control of a Humanoid Robot*. PhD Dissertation, TU Wien.

d'Apolito, F.; Mehmeti, X.; Kopacek, P. (2016): *Control of a Cost Oriented Humanoid Robot*. Proceedings of the 17th International Conference on "Technology, Culture and International

Stability", October 26-28, 2016, Duerres, Albania. Published by Elsevier ScienceDirect IFAC-PapersOnLine, Volume 49 (2016), Issue 29, pp. 18-23. N.Y DOI 10.1016/j.ifacol.2016.11.064.

d'Apolito, F.; Sulzbachner, C. (2017, October). *Obstacle avoidance system development for the Ardrone 2.0 using the tum_ardrone package*. In Workshop on "Research, Education and Development of Unmanned Aerial Systems (RED-UAS)", October 3-5, 2017, Linkoeping, Sweden. Published by IEEE, pp. 49-54. N.Y. DOI 10.1109/RED-UAS.2017.8101642.

d'Apolito F. (2018): *Obstacle Detection and Avoidance of a Cost-Oriented Humanoid Robot*. Proceedings of the 2018 IFAC International Conference on "Technology, Culture and International Stability", September 13-15, Baku, Azerbaijan. Published by Elsevier ScienceDirect IFAC-PapersOnLine, Volume 51 (2018), Issue 30, pp. 198-203. N.Y. DOI 10.1016/j.ifacol.2018.11.286.

Dezfouli, S. (2013): *Motion Creating System Design and Implementation for a Biped Humanoid Robot*. PhD Dissertation, TU Wien.

Distance line segment resolution method. https://math.stackexchange.com/questions/2248617/shortest-distance-between-a-point-and-a-line-segment - last retrieved 2018.

EDIMAX EW-7811UN Product Description. http://www.edimax.eu/edimax/merchandise/merchandise_detail/data/edimax/global/wireless_adapters_n150/ew-7811un/ - last retrieved 2018

ELinux Online Description Nvidia Jetson TK1. https://elinux.org/Jetson/Network_Adapters#Wifi_adapters_tested_on_Jetson_TK1 - last retrieved, 2018

Engel, J.; Sturm, J.; Cremers, D. (2012, October): *Camera-based navigation of a low-cost quadrocopter*. Proceedings of the 2012 IEEE/RSJ International Conference on "Intelligent Robotics System (IROS)", October 7-12, 2012, Vilamoura, Portugal. N.Y. DOI 10.1109/IROS.2012.6385458.

Euclidean Cluster Extraction Online Documentation. http://pointclouds.org/documentation/tutorials/cluster_extraction.php - last retrieved 2017.

Fallon, M. F.; Johannsson, H.; Kaess, M.; Rosen, D. M.; Muggler, E.; Leonard, J. J. (2012): *Mapping the MIT stata center: Large-scale integrated visual and RGB-D SLAM*. RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras.

Garimort, J.; Hornung, A.; Bennewitz, M. (2011): *Humanoid Navigation with Dynamic Footstep Plans*. Proceedings of the IEEE International Conference on "Robotics and Automation (ICRA)", May 9-13, 2011, Shanghai, China. N.Y. DOI 10.1109/ICRA.2011.5979656.

geometry online documentation. http://wiki.ros.org/geometry - last retrieved 2018

González-Fierro, M.; Hernández-García, D.; Nanayakkara, T.; Balaguer, C. (2015): *Behavior sequencing based on demonstrations: a case of a humanoid opening a door while walking*. Advanced Robotics, Volume 29, Issue 5, pp. 315-329. DOI 10.1080/01691864.2014.992955.

Hashimoto, K.; Hattori, K.; Otani, T.; Lim, H. O.; Takanishi, A. (2014): *Foot placement modification for a biped humanoid robot with narrow feet*. The Scientific World Journal, 2014.

Honda Asimo gallery. http://asimo.honda.com/gallery/ - last retrieved 2017

Honda Motors Co., Ltd (2007): *ASIMO Technical Information*. http://asimo.honda.com/downloads/pdf/asimo-technical-information.pdf - last retrieved 2018

Hornung, A.; Dornbush, A.; Likhachev, M.; Bennewitz, M. (2012): *Anytime Search-Based Footstep Planning with Suboptimality Bounds*. Proceedings of the IEEE-RAS International Conference on "Humanoid Robots (HUMANOIDS)", November 29 - December 12, 2012, Osaka, Japan. N.Y. DOI 10.1109/HUMANOIDS.2012.6651592.

Huang, H.; Wu, S.; Cohen-Or, D.; Gong, M.; Zhang, H.; Li, G.; Chen, B. (2013): *L1-medial skeleton of point cloud*. ACM Transaction on Graphics.

I, Bioloid blog. http://mike-ibioloid.blogspot.co.at/2015/11/robotis-thor-mang-3.html - last retrieved 2018

image_pipeline online documentation. http://wiki.ros.org/image_pipeline?distro=indigo - last retrieved 2018.

Kajita, S.; Hirukawa, H.; Harada, K.; Yokoi, K. (2014): *Introduction to Humanoid Robots*. Springer Tracts in Advanced Robotics.

Klein, G.; Murray, D. (2007): *Parallel tracking and mapping for small AR workspaces. In Mixed and Augmented Reality*. Proceedings of the 6th IEEE International Symposium on "Mixed and Augmented Reality", November 13-16, 2007, Nara, Japan. N.Y. DOI 10.1109/ISMAR.2007.4538852.

Kohlbrecher, S.; Stumpf, A.; Romay, A.; Schillinger, P.; Stryk, O.; Conner, D. (2016): *A comprehensive software framework for complex locomotion and manipulation tasks applicable to different types of humanoid robots*. Frontiers in Robotics and AI. Published online under DOI 10.3389/frobt.2016.00031.

Kokaj, A. (2018): *Sensing and Signal Processing Based Control of Balancing a Humanoid*, PhD Dissertation, TU Wien

Kopacek, P. (2013). *Development trends in robotics*. Elektrotechnik und Informationstechnik, 130 (2013), Vol.2, p.42-47, Published online under DOI 10.1007/s00502-013-0129-1, Springer Verlag, Wien, 2013.

Kuo, A.D. (2007): *The six determinants of gait and the inverted pendulum analogy: A dynamic walking perspective.* Human movement science, Volume 26, Issue 4, pp. 617-656. DOI 10.1016/j.humov.2007.04.003. Elsevier

Leopard Imaging. http://shop.leopardimaging.com/product.sc?productId=209 - last retrieved 2015.

Lee, B. J.; Stonier, D.; Kim, Y. D.; Yoo, J. K.; & Kim, J. H. (2008): *Modifiable walking pattern of a humanoid robot by using allowable ZMP variation*. IEEE Transactions on Robotics, Volume 24, Issue 4, pp. 917-925. DOI 10.1109/TRO.2008.926859. IEEE.

LibUVC. https://github.com/ktossell/libuvc - last retrieved 2016

Manduchi, R.; Castano, A.; Talukder, A.; Matthies, L. (2005): *Obstacle Detection and Terrain Classification for Autonomous Off-road Navigation.* Autonomous Robots, Vol. 18, Issue 1, pp. 81–102. DOI 10.1023/B:AURO.0000047286.62481.1d. Springer.

Milford M.; Schulz, R. (2014): *Principles of goal-directed spatial robot navigation in biomimetic models*. Philosophical Transactions of The Royal Society B: Biological Sciences. DOI 10.1098/rstb.2013.0484.

Motor drive online shop page. https://www.aliexpress.com/item/24V-BLDC-Motor-controller-BLD-70-12V-30VDC-70W-Brushless-DC-Motor-Driver-Controller/32845166025.html?spm=a2g0s.9042311.0.0.OZvrvA - last retrieved 2018

MPU 6050. https://playground.arduino.cc/Main/MPU-6050 - last retrieved 2018

My Point Cloud blog. https://taylorwang.wordpress.com/2012/04/06/collision-free-path-planning-using-potential-field-method-for-highly-redundant-manipulators/ - last retrieved 2017.

NAO online description. https://www.ald.softbankrobotics.com/en/cool-robots/nao/find-out-more-about-nao - last retrieved 2017

NASA Robonaut website. https://robonaut.jsc.nasa.gov/R2/ - last retrieved 2017

Nvidia. http://www.nvidia.de/object/jetson-tk1-embedded-dev-kit-de.html - last retrieved 2015

OpenCV Online Documentation on Epipolar Geometry and Stereo Vision. http://docs.opencv.org/master/da/de9/tutorial_py_epipolar_geometry.html - last retrieved 2017

pcl_ros online documentation. http://wiki.ros.org/pcl_ros - last retrieved 2018

Pire, T.; Fischer, T.; Civera, J.; De Cristóforis, P.; & Berlles, J. J. (2015): *Stereo parallel tracking and mapping for robot localization*. Proceedings of the 2015 IEEE/RSJ International Conference on "Intelligent Robotics System (IROS)", September 28 – October 2, 2015, Hamburg, Germany. N.Y. DOI 10.1109/IROS.2015.7353546.

Quigley, M.; Conley, K.; Gerkey, B. P.; Faust, J.; Foote, T.; Leibs, J.; Berger, E.; Wheeler, R.; Ng, A. (2009): *ROS: an open-source robot operating system*. ICRA Workshop on Open Source Software.

Robotis website Thormang description. http://www.robotis.us/thormang/ - last retrieved 2017

Robotis website Bioloid description. http://www.robotis.us/bioloid-1/ - last retrieved 2017

Rusu, R.; (2009): *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD Dissertation, TU Muenchen.

rviz online documentation. http://wiki.ros.org/rviz - last retrieved 2018

rviz_visual_tools online documentation. http://wiki.ros.org/rviz_visual_tools - last retrieved 2018.

Schoerghuber, M. (2014): *Controller design of a humanoid robot*. Master thesis, TU Wien

Siciliano, B.; Sciavicco, L.; Villani L.; Oriolo, G. (2009): *Robotics: Modelling, Planning and Control*. Springer Advanced Textbook in Control and Signal Processing.

S-Ptam Github Documentation. https://github.com/lrse/sptam - last retrieved 2018

stereo_image_proc online documentation. http://wiki.ros.org/stereo_image_proc - last retrieved 2018

Stumpf, A.; Kohlbrecher, S.; Stryk, O.; Conner, D. (2016): *Open Source Integrated 3D Footstep Planning Framework for Humanoid Robots*. Proceedings of the IEEE-RAS International Conference on Humanoid Robots, November 15-17, 2016, Cancun, Mexico. N.Y. DOI 10.1109/HUMANOIDS.2016.7803385. IEEE.

TP-Link TL-WR841N Online specification. https://www.tp-link.com/uk/products/details/cat-9_TL-WR841N.html - last retrieved 2018

Vadakkepat, P.; Goswami, D. (2008): *Biped locomotion: stability, analysis and control*. International Journal on Smart sensing and intelligent systems.

Weiss, S.; Achtelik, M. W.; Lynen, S.; Chli, M.; Siegwart, R. (2012): *Real-time onboard visual-inertial state estimation and self-calibration of mavs in unknown environments*. Proceedings of the 2012 IEEE International Conference in “Robotics and Automation (ICRA)”, May 14-18, 2012, Saint Paul, MN, USA. N.Y. DOI 10.1109/ICRA.2012.6225147. IEEE.

Appendix

Appendix A: Scilab simulation of joint movements during the gait

A1: Stepping.sce

This file computes the joint movements during the gait using the relation explained in Chapter 4.

```
clear
close
clc

//Initializations steps
numSteps = 12;
stepsFrontal = zeros(1,numSteps);
stepSagittal = zeros(1,numSteps);

//Initial DH Parameteres
a1 = 12;
a2 = 26;
a3 = 31;
a4 = 0;
a5 = 0;
a6 = 18;
a7 = 0;
a8 = 0;
a9 = 31;
a10 = 26;
a11 = 12;
a12 = 3.5;

d1 = 0;
d2 = 0;
d3 = 7.5;
d4 = 0;
d5 = 0;
d6 = 5;
d7 = 5;
d8 = 0;
d9 = 7.5;
d10 = 0;
d11 = 0;
d12 = 0;

theta0 = zeros(numSteps,1);
theta1 = zeros(numSteps,1);
theta2 = zeros(numSteps,1);
theta3 = zeros(numSteps,1);
theta4 = zeros(numSteps,1);
theta5 = zeros(numSteps,1);
theta6 = zeros(numSteps,1);
theta7 = zeros(numSteps,1);
theta8 = zeros(numSteps,1);
theta9 = zeros(numSteps,1);
theta10 = zeros(numSteps,1);
theta11 = zeros(numSteps,1);
```

```
x = zeros(numSteps,16);
y = zeros(numSteps,16);
z = zeros(numSteps,16);

theta0(1,1) = 0*%pi/180;
theta1(1,1) = 0*%pi/180;
theta2(1,1) = 0*%pi/180;
theta3(1,1) = 0*%pi/180;
theta4(1,1) = 0*%pi/180;
theta5(1,1) = 0*%pi/180;
theta6(1,1) = 0*%pi/180;
theta7(1,1) = 0*%pi/180;
theta8(1,1) = 0*%pi/180;
theta9(1,1) = 0*%pi/180;
theta10(1,1) = 0*%pi/180;
theta11(1,1)= = 0*%pi/180;

//Starting point
x0 = 0;
y0 = 0;
z0 = 3.5;

stepsFrontal(1,1:numSteps/2) = linspace(16.5,6.5,numSteps/2);
stepsFrontal(1,numSteps/2+1:numSteps) = linspace(6.5,16.5,numSteps/2);
stepsSagittal(1,numSteps/2+1:numSteps) = linspace(0,15,numSteps/2);

//Joints movements calculations
for i = 1:numSteps

   //Movements on x and y of the COM
   deltaY = stepsFrontal(1,i);
   deltaX = stepsSagittal(1,i);

   //Theta1 computation from x movement
   th1 = asin(deltaX/(a2+a3));

   //Theta0 computation from x and y movements
   k1 = a1+(a2+a3)*cos(th1);
   k2 = (a6/2-d3-deltaY);
   k3 = (d3+a6/2-deltaY);

   th01 = atan(2*((2*k1+sqrt(4*k1*k1-4*k2*k3))/(2*k2)));
   th02 = atan(2*((2*k1-sqrt(4*k1*k1-4*k2*k3))/(2*k2)));

   //theta angles
   theta1(i,1) = th1;
   theta0(i,1) = min(abs(th01),abs(th02));
   theta4(i,1) = -theta0(i,1);
   theta2(i,1) = 0;
   theta3(i,1) = -theta1(i,1);
   theta5(i,1) = 0;
   theta6(i,1) = 0;
   theta7(i,1) = 0;

   //Right leg and hip positions
   x1 = x0;
   x2 = x0 + a2*sin(theta1(i,1));
   x3 = x2 + a3*sin(theta1(i,1)+theta2(i,1));
   x4 = x3;
   x5 = x4 - d6*sin(theta1(i,1)+theta2(i,1)+theta3(i,1));
   x6 = x5 + a6*cos(theta4(i,1)+theta0(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1))*sin(theta5(i,1));
   x7 = x6 + d6*sin(theta1(i,1)+theta2(i,1)+theta3(i,1));
   x8 = x7 +
d3*cos(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1))*sin(theta5(i,1)+theta6(i,1));
```

```
y1 = y0-a1*sin(theta0(i,1));
   y2 = y1 - a2*cos(theta1(i,1))*sin(theta0(i,1));
   y3 = y2 - a3*cos(theta1(i,1)+theta2(i,1))*sin(theta0(i,1));
   y4 = y3 + d3*cos(theta0(i,1));
   y5 = y4 + d6*sin(theta4(i,1)+theta0(i,1));
   y6 = y5 + a6*cos(theta4(i,1)+theta0(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1))*cos(theta5(i,1));
   y61 = y5 + (a6/2)*cos(theta4(i,1)+theta0(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1))*cos(theta5(i,1));
   y7 = y6 - d6*sin(theta4(i,1)+theta0(i,1));
   y8 = y7 +
d3*cos(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1))*cos(theta5(i,1)+theta6(i,1));

   z1 = z0 + a1*cos(theta0(i,1));
   z2 = z1 + a2*cos(theta1(i,1))*cos(theta0(i,1));
   z3 = z2 + a3*cos(theta1(i,1)+theta2(i,1))*cos(theta0(i,1));
   z4 = z3 + d3*sin(theta0(i,1));
   z5 = z4 - d6*cos(theta4(i,1)+theta0(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1));
   z6 = z5 + a6*sin(theta4(i,1)+theta0(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1));
   z7 = z6 + d6*cos(theta4(i,1)+theta0(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1));
   z8 = z7 + d3*sin(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1));

   //Theta angle for left leg
   hn = z8;
   hn1 = hn - y8*tan(2*theta0(i,1)) - a11 - a12;
   rSq = deltaX*deltaX + hn1*hn1;
   r = sqrt(rSq);
   gammaAngle = -acos((a9*a9+a10*a10-rSq)/(2*a9*a10));
   theta9(i,1) = %pi - gammaAngle;
   theta8(i,1) = acos((a9*a9+rSq-a10*a10)/(2*a9*r))+atan(deltaX/hn1);
   theta10(i,1) = -(theta9(i,1) + theta8(i,1));
   theta11(i,1) = 0;

   //Left leg positions
   x9 = x8 + a9*sin(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1));
   x10 = x9 + a10*sin(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1));
   x11 = x10 + a11*sin(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1)+theta10(i,1));
   x12 = x11 + a12*sin(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1)+theta10(i,1));
   x13 = x12 - 3;
   x14 = x12 + 19;

   y9 = y8 + a9*sin(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1));
   y10 = y9 +
a9*sin(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1));
   y11 = y10 +
a11*sin(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1)+theta10(
i,1));
   y12 = y11 +
a12*sin(theta4(i,1)+theta0(i,1)+theta7(i,1)+theta11(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(
i,1)+theta10(i,1));
   y13 = y12 - 3;
   y14 = y12 + 3;

   z9 = z8 - a9*cos(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1));
   z10 = z9 -
a10*cos(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1));
   z11 = z10 -
a11*cos(theta4(i,1)+theta0(i,1)+theta7(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9(i,1)+theta10
(i,1));
   z12 = z11 -
a12*cos(theta4(i,1)+theta0(i,1)+theta7(i,1)+theta11(i,1))*cos(theta1(i,1)+theta2(i,1)+theta3(i,1)+theta8(i,1)+theta9
(i,1)+theta10(i,1));
   z13 = z12;
   z14 = z12;

   x(i,:) = [0,x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14];
   y(i,:) = [0,y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14];
   z(i,:) = [0,z0,z1,z2,z3,z4,z5,z6,z7,z8,z9,z10,z11,z12,z13,z14];
end
```

```
//Plots
figure(1)
for i = 1:numSteps/2
   plot(y(i,:),z(i,:),'o');
   plot(y(i,:),z(i,:));
end
a = gca();
a.isoview = "on";
a.x_label.text ="y";
a.y_label.text = "z";

figure(2)
for i =1:numSteps/2
   plot(x(i,8:16),z(i,8:16),'o');
   plot(x(i,8:16),z(i,8:16));
end
a1 = gca();
a1.isoview = "on";
a1.x_label.text ="x";
a1.y_label.text = "z";

figure(3)
for i = numSteps/2+1:numSteps
   plot(x(i,:),z(i,:),'o');
   plot(x(i,:),z(i,:));
end
a2 = gca();
a2.isoview = "on";
a2.x_label.text ="x";
a2.y_label.text = "z";

figure(4)
a3=get("current_axes");
for i = 1:numSteps
   param3d(x(i,:),y(i,:),z(i,:));
end
a3 = gca();
a3.isoview = "on";
a3.x_label.text ="x";
a3.y_label.text = "y";
a3.z_label.text = "z";
```

Appendix B: Camera Driver and vision_mapping node

B1. Camera manager node

B1.1 main_archievision.cpp

This files initialize the ROS node of the camera driver and calls the loop by means of which the first image pre-processing is performed.

```
/*
 * main_archievision.cpp
 *
 *  Created on: Dec 19, 2015
 *      Author: Francesco d'Apolito
 */

#include "ros/ros.h"
#include <iostream>
#include <libuvc/libuvc.h>
#include <libuvc/libuvc_config.h>
#include "opencv2/opencv.hpp"
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <vector>
#include "CameraDriver.h"

using namespace cv;
using namespace std;

int main(int argc, char **argv)
{
    //Ros initializations
    ros::init(argc, argv, "archie_vision");
    ROS_INFO("Started Archie Vision Node");
    int ret = 0;
    ros::NodeHandle node;
    ros::NodeHandle camera_node;

    //Camera Driver Initialization
    Camera_Driver::CameraDriver camDriv(node,camera_node);

    //Camera driver loop
    ret = camDriv.loop();

    return ret;
}
```

B1.2 CameraDriver.h

```
/*
 * CameraDriver.h
 *
 *  Created on: Apr 7, 2016
 *      Author: Francesco d'Apolito
 */

#ifndef CAMERADRIVER_H_
#define CAMERADRIVER_H_

#include "ros/ros.h"
#include <iostream>
#include <libuvc/libuvc.h>
#include <libuvc/libuvc_config.h>
#include "opencv2/opencv.hpp"
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <vector>
#include <sensor_msgs/Image.h>
#include <image_transport/image_transport.h>
#include <tf/tfMessage.h>
#include <tf/transform_broadcaster.h>
#include <cv_bridge/cv_bridge.h>
#include <boost/thread/mutex.hpp>
#include <sensor_msgs/CameraInfo.h>
#include <camera_info_manager/camera_info_manager.h>
#include "archie_ai/start_capture.h"
#include "archie_ai/stop_capture.h"

using namespace std;
using namespace cv;

namespace Camera_Driver
{

class CameraDriver {

private:

    uvc_context_t *ctx;
    uvc_device_t *dev;
    uvc_device_handle_t *devh;
    uvc_stream_ctrl_t ctrl;
    uvc_error_t res;
    uvc_device_t **list;
```

```
    ros::NodeHandle node;
    ros::NodeHandle camera_node;

    ros::Time begin;
    ros::Time lastPublished;
    ros::Time t;

    boost::shared_ptr<image_transport::ImageTransport> it;
    image_transport::CameraPublisher imageL_pub;
    image_transport::CameraPublisher imageR_pub;

    tf::TransformBroadcaster br;
    tf::Transform transform;

    std::string cameraName;
    boost::shared_ptr<camera_info_manager::CameraInfoManager> cinfo_;
public:
    CameraDriver(ros::NodeHandle nh, ros::NodeHandle camera_nh);
    virtual ~CameraDriver();
    int loop();
    bool cameraOn(archie_ai::start_capture::Request &req,
archie_ai::start_capture::Response &resp);
    bool cameraOff(archie_ai::stop_capture::Request &req,
archie_ai::stop_capture::Response &resp);
    std::string cameraLeftChannel;
    std::string cameraRightChannel;

    ros::ServiceServer start_service;
    ros::ServiceServer stop_service;

    bool isRunning;
    bool streaming;
    bool found;
    bool stopReceived;
    bool contextInitialized;

};

}
#endif /* CAMERADRIVER_H_ */
```

B1.3 CameraDriver.cpp

This file handles the requests to start and stop the camera stream. Furthermore it separates the incoming data into left and right image as explained in Chapter 6.

```
/*
 * CameraDriver.cpp
 *
 *  Created on: Apr 7, 2016
 *      Author: Francesco d'Apolito
 */

#include "CameraDriver.h"

pthread_mutex_t lockImgQueLeft = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lockImgQueRight = PTHREAD_MUTEX_INITIALIZER;
std::vector<cv::Mat> imgQueLeft;
std::vector<cv::Mat> imgQueRight;

cv::Mat left1;
cv::Mat right1;

void cb(uvc_frame_t *frame, void *ptr)
{
    frame->frame_format = UVC_FRAME_FORMAT_YUYV;
    std::cout << "Frame format" << std::endl;
    std::cout << frame->frame_format << std::endl;
    uvc_frame_t *greyLeft;
    uvc_frame_t *greyRight;
    uvc_error_t retLeft;
    uvc_error_t retRight;

    /* We'll convert the image from YUV/JPEG to gray8, so allocate space */
    greyLeft = uvc_allocate_frame(frame->width * frame->height);
    greyRight = uvc_allocate_frame(frame->width * frame->height);
    if (!greyLeft)
    {
        ROS_DEBUG("unable to allocate grey left frame!");
        return;
    }
    if (!greyRight)
    {
        ROS_DEBUG("unable to allocate grey right frame!");
        return;
    }
```

```
    IplImage *cvImg, *cvImg2;
    cvImg = cvCreateImage(cvSize(greyLeft->width, greyLeft->height),
IPL_DEPTH_8U, 1);
    cvImg2 = cvCreateImage(cvSize(greyRight->width, greyRight->height),
IPL_DEPTH_8U, 1);
    cvImg->imageData = (char*) greyLeft->data;
    cvImg2->imageData = (char*) greyRight->data;

    cv::Mat left (greyLeft->height, greyLeft->width, CV_8UC1, greyLeft->data);
    cv::Mat right (greyRight->height, greyRight->width, CV_8UC1, greyRight-
>data);
    left1 = left.clone();
    right1 = right.clone();

    pthread_mutex_lock(&lockImgQueLeft);
    imgQueLeft.push_back(left1);

    pthread_mutex_unlock(&lockImgQueLeft);
    pthread_mutex_lock(&lockImgQueRight);
    imgQueRight.push_back(right1);
    pthread_mutex_unlock(&lockImgQueRight);

    cvReleaseImage(&cvImg);
    cvReleaseImage(&cvImg2);

    uvc_free_frame(greyLeft);
    uvc_free_frame(greyRight);
}

namespace Camera_Driver
{

//Constructor
CameraDriver::CameraDriver(ros::NodeHandle nh, ros::NodeHandle camera_nh):
    node(nh,"archie_ai"),
    camera_node(camera_nh,"archie_camera"),
    cinfo_(new camera_info_manager::CameraInfoManager(camera_node)),
    it(new image_transport::ImageTransport(camera_node)),
    imageL_pub(it->advertiseCamera("left/image_raw",1)),
    imageR_pub(it->advertiseCamera("right/image_raw",1))
{
    streaming = false;
    isRunning = false;
    found = false;
    stopReceived = false;
    contextInitialized = false;

    start_service =
node.advertiseService("/archie_camera/start_capture",&CameraDriver::cameraOn,t
his);
```

```
    stop_service = node.advertiseService("/archie_camera/stop_capture",
&CameraDriver::cameraOff,this);
}

//Destructor
CameraDriver::~CameraDriver()
{

}

//Callback of the request to turn on the camera: Start the stream if the
request is received
bool CameraDriver::cameraOn(archie_ai::start_capture::Request &req,
archie_ai::start_capture::Response &resp)
{
    bool ret = false;
    //if(req.capture)
    //{
        ROS_INFO("Received request for turning camera on\n");
        res = uvc_init(&ctx, NULL);

        if (res < 0)
        {
            uvc_perror(res, "uvc_init");
            ret = false;
        }
        else
        {
            contextInitialized = true;
            ROS_INFO("UVC initialized");
            res = uvc_get_device_list(ctx, &list);
            if (res < 0)
            {
                uvc_perror(res, "uvc_get_device_list");
                ret = false;
            }
            else
            {
                /* Locates the first attached UVC device, stores in dev */
                res = uvc_find_device(ctx, &dev,0, 0, NULL); /* filter
devices: vendor_id, product_id, "serial_num" */

                if (res < 0)
                {
                    uvc_perror(res, "uvc_find_device"); /* no devices found */
                    ret = false;
                }
                else
                {
                    ROS_INFO("Device found");
```

```
                    found = true;
                    /* Try to open the device: requires exclusive access */
                    res = uvc_open(dev, &devh);

                    if (res < 0)
                    {
                        uvc_perror(res, "uvc_open"); /* unable to open device
*/
                        ROS_INFO("Unable to open device");
                        ret = false;
                    }
                    else
                    {
                        ROS_INFO("Device opened");
                        uvc_print_diag(devh, stderr);
                        isRunning = true;
                        ret = true;
                    }
                }
            }
        }
    //}
    begin = ros::Time::now();
    lastPublished = begin;
    resp.status = 1;
    return ret;
}

//Callback for the request to turn of camera: stop the stream
bool CameraDriver::cameraOff(archie_ai::stop_capture::Request &req,
archie_ai::stop_capture::Response &resp)
{
    stopReceived = true;

    //if(req.stop)
    //{
        ROS_INFO("Received stop request\n");
        // TODO Auto-generated destructor stub
        if(isRunning)
        {
            if(streaming)
            {
                /* End the stream. Blocks until last callback is serviced */
                uvc_stop_streaming(devh);
                ROS_INFO("Done streaming.");
            }
            /* Release our handle on the device */
            uvc_close(devh);
            ROS_INFO("Device closed");
        }
```

```
        if(found)
        {
            /* Release the device descriptor */
            uvc_unref_device(dev);
        }
        if(contextInitialized)
        {
            /* Close the UVC context. This closes and cleans up any existing
device handles,
            * and it closes the libusb context if one was not provided. */
            uvc_exit(ctx);
            ROS_INFO("UVC exited");
        }
    //}
    resp.res = 1;
    return true;
}

int CameraDriver::loop()
{
    while(!isRunning)
    {
        ros::spinOnce();
    }
    if(isRunning)
    {
        res = uvc_get_stream_ctrl_format_size(devh, &ctrl,
UVC_FRAME_FORMAT_YUYV, 640, 480, 30);

        /* Print out the result */
        uvc_print_stream_ctrl(&ctrl, stderr);

        if (res < 0)
        {
            uvc_perror(res, "get_mode"); /* device doesn't provide a matching
stream */
        }
        else
        {
            /* Start the video stream. The library will call user function cb:
            *   cb(frame, (void*) 12345)
            */

            res = uvc_start_streaming(devh, &ctrl, cb, (void*)12345, 0);

            if (res < 0)
            {
                uvc_perror(res, "start_streaming"); /* unable to start stream
*/
            }
```

```
            else
            {
                ROS_INFO("Streaming...");
                streaming = true;
                uvc_set_ae_mode(devh, 0); /* e.g., turn on auto exposure */

                while(!stopReceived)
                {
                    t = ros::Time::now();
                    if(t.toSec() - lastPublished.toSec() < 1)
                    {
                        //Transform from camera frame to base link PTAM frame
                        transform.setOrigin(tf::Vector3(0.0,0.0,0.5));
                        tf::Quaternion q =
tf::createQuaternionFromRPY(M_PI/2,0,0);
                        transform.setRotation(q);

br.sendTransform(tf::StampedTransform(transform,ros::Time::now(),"base_link","
cam_left"));

                        //Camera_info message
                        sensor_msgs::CameraInfoPtr ci(new
sensor_msgs::CameraInfo(cinfo_->getCameraInfo()));
                        //Image left message
                        pthread_mutex_lock(&lockImgQueLeft);
                        if(!imgQueLeft.empty())
                        {
                            int size = imgQueLeft.size();
                            ROS_DEBUG("size of imageQueLeft = %d. getting
lock\n",size);
                            std_msgs::Header head_left;
                            head_left.frame_id = "cam_left";
                            sensor_msgs::ImagePtr msg =
cv_bridge::CvImage(head_left, "mono8", imgQueLeft.at(size-1)).toImageMsg();
                            imageL_pub.publish(msg,ci);
                            imgQueLeft.clear();
                        }
                        pthread_mutex_unlock(&lockImgQueLeft);

                        //Image right message
                        pthread_mutex_lock(&lockImgQueRight);
                        if(!imgQueRight.empty())
                        {
                            int size = imgQueRight.size();
                            ROS_DEBUG("size of imageQueRight = %d. getting
lock\n",size);
                            std_msgs::Header head_right;
                            head_right.frame_id = "cam_right";
                            sensor_msgs::ImagePtr msg =
cv_bridge::CvImage(head_right, "mono8", imgQueRight.at(size-1)).toImageMsg();
```

```
                        imageR_pub.publish(msg,ci);
                        imgQueRight.clear();
                    }
                    pthread_mutex_unlock(&lockImgQueRight);
                    lastPublished = t;

                }
                ros::spinOnce();
            }
        }
    }
    return res;
}

}
```

B2: Vision and mapping node

B2.1 main_planning.cpp

This file initializes the planning node and starts the planner loop where the obstacle detection and path planning are performed

```
#include <ros/ros.h>
#include <cv_bridge/cv_bridge.h>
#include "PlanningNode.h"
#include "PathSearchTree.h"

int main(int argc, char **argv)
{
    //ROS Node intialization
    ros::init(argc, argv, "archie_planning");
    ros::NodeHandle nh;
    ROS_INFO("Started Archie planning Node");

    PlanningNode planner(nh);

    //Calling the planner loop
    planner.loop();

    ROS_INFO("Exited from loop, exiting cleanly... \n");

    //When the planner is finished, return
    return 0;

}
```

B2.2 PlanningNode.h

```
#ifndef PLANNINGNODE_H
#define PLANNINGNODE_H

#include <ros/ros.h>
#include "HelperVariable.h"
#include <geometry_msgs/Pose.h>
#include "archie_ai/start_planning.h"
#include "archie_ai/new_map.h"
#include "archie_ai/clearMapAndPlanning.h"
#include "PathSearchTree.h"
#include "ObstacleDetection.h"
#include <cv_bridge/cv_bridge.h>
#include "opencv2/opencv.hpp"
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <pcl_ros/point_cloud.h>
#include <sensor_msgs/PointCloud2.h>
#include <rviz_visual_tools/rviz_visual_tools.h>
#include <geometry_msgs/PoseStamped.h>

class PlanningNode
{
private:
    ros::NodeHandle nh;

    std::string pclChannel;
    std::string pclPubChannel;
    std::string markerObsChannel;
    std::string markerPlanningChannel;
    std::string markerFootChannel;
    std::string markerFootPolygonChannel;
    std::string goalTopic;
    std::string poseTopic;
    std::string footstepTopic;

    ros::Subscriber pcl_sub;
    ros::Subscriber goal_sub;
    ros::Subscriber pose_sub;
    ros::Publisher footstep_pub;
    ros::Publisher pcl_pub;     //just for visualization in rviz
    ros::Publisher markerObs_pub;
    ros::Publisher markerPlanning_pub;
    ros::Publisher markerFoot_pub;
    ros::Publisher markerFootPolygon_pub;
```

```
    std::vector<obstacles> obst;
    std::vector<ObsList> obstacleList;
    std::vector<geometry_msgs::Pose2D> waypoints;

    sensor_msgs::PointCloud2Ptr pclMsgBack;

    tf::TransformBroadcaster br1;
    tf::Transform transform1;

public:

    geometry_msgs::Pose archiePose;
    geometry_msgs::Pose2D archiePose2D;
    geometry_msgs::Pose2D endPosition;
    PathSearchTree *planTrees;
    ObstacleDetection *obstDetector;
    bool running, goalAvailable;
    bool pclAvailable, obstAvailable;
    geometry_msgs::Pose2D posArchie;
    geometry_msgs::Pose2D goal;
    tf::Vector3 translation;

    rviz_visual_tools::RvizVisualToolsPtr vis_toolObs;
    rviz_visual_tools::RvizVisualToolsPtr vis_toolPlanning;
    rviz_visual_tools::RvizVisualToolsPtr vis_toolFoot;
    rviz_visual_tools::RvizVisualToolsPtr vis_toolFootPolygon;

    ros::ServiceServer start_planning;
    ros::ServiceServer new_map;
    ros::ServiceServer clearRViz;

    PlanningNode(ros::NodeHandle node);
    bool planningCb(archie_ai::start_planningRequest &req,
archie_ai::start_planningResponse &res);
    bool new_mapCb(archie_ai::new_mapRequest &req, archie_ai::new_mapResponse
&res);
    bool clearMapAndPlanCb(archie_ai::clearMapAndPlanningRequest &req,
archie_ai::clearMapAndPlanningResponse &res);
    void loop();
    void pclCb(const sensor_msgs::PointCloud2ConstPtr pclMsg);
    void poseCb(geometry_msgs::PoseStampedConstPtr pose);
    void goalCb(geometry_msgs::PoseStamped g);
};

#endif // PLANNINGNODE_H
```

B2.3 PlanningNode.cpp

This file are the functions that receives the incoming data from the S-PTAM. The point cloud is then given here as input to the obstacle detector class. This file also receives the target position of the target by the user and sends it, together with the pose received from the S-PTAM to the planner. This file also serves as interface between the developed software and RViz, by sending to it the markers which represent detected obstacles and planned trajectory.

```
#include "PlanningNode.h"
#include "WalkingGeneration.h"
#include <tf/LinearMath/Quaternion.h>
#include <visualization_msgs/Marker.h>
#include <visualization_msgs/MarkerArray.h>
#include <string>
#include <stdlib.h>
#include <string.h>
#include <archie_ai/footstepArray.h>

//Constructor
PlanningNode::PlanningNode(ros::NodeHandle node): nh(node, "vision_mapping")
{
    ROS_INFO("Planning node running\n");

    //Topic names intialization
    pclChannel = nh.resolveName("/sptam/point_cloud");
    poseTopic = nh.resolveName("/sptam/robot/pose");
    pclPubChannel = nh.resolveName("/archie_camera/pointcloud2");
    footstepTopic = nh.resolveName("vision_mapping/footsteps");
    markerObsChannel = nh.resolveName("/vision_mapping/markers_obs");
    markerPlanningChannel = nh.resolveName("/vision_mapping/marker_planning");
    markerFootChannel = nh.resolveName("/vision_mapping/marker_feet");
    markerFootPolygonChannel =
nh.resolveName("/vision_mapping/marker_feet_polygon");
    goalTopic = nh.resolveName("/move_base_simple/goal");

    //Publisher and Subscriber intialization
    pcl_sub = nh.subscribe(pclChannel, 1, &PlanningNode::pclCb, this);
    pose_sub = nh.subscribe(poseTopic, 1, &PlanningNode::poseCb, this);
    goal_sub = nh.subscribe(goalTopic,1,&PlanningNode::goalCb,this);
    footstep_pub = nh.advertise<archie_ai::footstepArray>(footstepTopic,1);
    pcl_pub = nh.advertise<sensor_msgs::PointCloud2>(pclPubChannel,1);
```

```
markerObs_pub =
nh.advertise<visualization_msgs::MarkerArray>(markerObsChannel,1);
    markerPlanning_pub =
nh.advertise<visualization_msgs::MarkerArray>(markerPlanningChannel,1);
    markerFoot_pub =
nh.advertise<visualization_msgs::MarkerArray>(markerFootChannel,1);
    markerFootPolygon_pub =
nh.advertise<visualization_msgs::MarkerArray>(markerFootPolygonChannel,1);

    //RViz visual tools initialization
    vis_toolObs.reset(new
rviz_visual_tools::RvizVisualTools("/map",markerObsChannel));
    vis_toolPlanning.reset(new
rviz_visual_tools::RvizVisualTools("/map",markerPlanningChannel));
    vis_toolFoot.reset(new
rviz_visual_tools::RvizVisualTools("/map",markerFootChannel));
    vis_toolFootPolygon.reset(new
rviz_visual_tools::RvizVisualTools("/map",markerFootPolygonChannel));

    //Service publisher for starting the planning, insert a user created map
and clear the Rviz view
    start_planning =
nh.advertiseService("vision_mapping/start_planning",&PlanningNode::planningCb,
this);
    new_map = nh.advertiseService("/vision_mapping/new_map",
&PlanningNode::new_mapCb,this);
    clearRViz =
nh.advertiseService("/vision_mapping/clearRviz",&PlanningNode::clearMapAndPlan
Cb, this);

    //pclMsgBack = new sensor_msgs::PointCloud2Ptr;

    running = true;
    goalAvailable = false;
    pclAvailable = false;
    obstAvailable = false;

    planTrees = new PathSearchTree();
    obstDetector = new ObstacleDetection();

    posArchie.x = 0;
    posArchie.y = 0;

    transform1.setOrigin(tf::Vector3(0, 0, 1.5));
    tf::Quaternion q1 = tf::createQuaternionFromRPY(0,0,0);
    transform1.setRotation(q1);
}
```

```
//Callback for the pose. It receives and memorize the pose from the S-PTAM
void PlanningNode::poseCb(geometry_msgs::PoseStampedConstPtr pose)
{
    posArchie.x = pose->pose.position.x;
    posArchie.y = pose->pose.position.z;

    translation = tf::Vector3(pose->pose.position.x, pose->pose.position.z,
abs(pose->pose.position.y));

    transform1.setOrigin(translation);
    tf::Quaternion q1 = tf::createQuaternionFromRPY(0,0,0);
    transform1.setRotation(q1);
}

//Callback for the target position of the planning. It receives and memorize
the target position
void PlanningNode::goalCb(geometry_msgs::PoseStamped g)
{
    goal.x = g.pose.position.x;
    goal.y = g.pose.position.y;
    goal.theta = 0;
    goalAvailable = true;
}

//Callback for the start_planning service. As soon as the user call this
service the planning start
bool PlanningNode::planningCb(archie_ai::start_planningRequest &req,
archie_ai::start_planningResponse &res)
{
    goal.x = req.gx;
    goal.y = req.gy;
    goal.theta = 0;
    goalAvailable = true;
    return true;
}

//Callback for the service new map. The user calls it together with a list of
obstacles.
//It memorize the obstacles as soon as it gets them
bool PlanningNode::new_mapCb(archie_ai::new_mapRequest &req,
archie_ai::new_mapResponse &res)
{
    if(!req.ox.empty() && !req.oy.empty() && req.ox.size() == req.oy.size())
//Check if the input of the user is correct first
    {
        if(!obstacleList.empty())  //If the map is not empty then clear
        {
            obstacleList.clear();
        }
```

```
        for(unsigned int i = 0; i < req.ox.size(); i++)  //Fill the obstacle
list with the new obstacles
        {
            ObsList o;
            o.obstX = req.ox.at(i);
            o.obstY = req.oy.at(i);
            o.radius = 0.5;
            o.height = 1;
            ROS_INFO("Received %f %f %f %f \n", o.obstX, o.obstY, o.radius,
o.height);
            obstacleList.push_back(o);
        }
        obstAvailable = true;
        return true;
    }
    else
    {
        return false;
    }
}

//Callback for the service clear map and plan. As soon as it is received it
clears all the markers in RViz
bool PlanningNode::clearMapAndPlanCb(archie_ai::clearMapAndPlanningRequest
&req, archie_ai::clearMapAndPlanningResponse &res)
{
    bool ret = false;
    obstAvailable = false;
    pclAvailable = false;
    goalAvailable = false;

    if(!obstacleList.empty())  //If the map is not empty then clear
    {
        obstacleList.clear();
    }
    if(!planTrees->obstToAvoid.empty())  //If the map is not empty then clear
    {
        planTrees->obstToAvoid.clear();
    }
    if(!planTrees->obstToStepOn.empty())  //If the map is not empty then clear
    {
        planTrees->obstToStepOn.clear();
    }

    bool ret1 = vis_toolObs->deleteAllMarkers();
    bool ret2 = vis_toolPlanning->deleteAllMarkers();
    bool ret3 = vis_toolFoot->deleteAllMarkers();
    bool ret4 = vis_toolFootPolygon->deleteAllMarkers();
```

```
    if(ret1 && ret2 && ret3 && ret4)
    {
      ret = true;
    }

    res.res = ret;
    return ret;
}

//Loop of the planning node
void PlanningNode::loop()
{
    std::string obstString = "obst";
    //Color Obstacles
    float rObstacle = 1.0f; //0.0f;
    float gObstacle = 0.0f; //1.0f;
    float bObstacle = 0.0f; //0.0f;
    float aObstacle = 1.0; //1.0;

    //Color Points
    float rPoints = 0.0f;
    float gPoints = 1.0f;
    float bPoints = 0.0f;
    float aPoints = 1.0;

    //Color Points Feet
    float rCenterFeet = 0.0f;
    float gCenterFeet = 1.0f;
    float bCenterFeet = 0.0f;
    float aCenterFeet = 1.0;

    //Color Points Feet
    float rPointsFeet = 1.0f;
    float gPointsFeet = 0.0f;
    float bPointsFeet = 0.0f;
    float aPointsFeet = 1.0;

    //Color Lines Trajectory Planned
    float rLinesT = 0.0f;
    float gLinesT = 0.0f;
    float bLinesT = 1.0f;
    float aLinesT = 1.0;

    while(running)
    {
        //Call for incoming data
        ros::spinOnce();

        visualization_msgs::MarkerArray markArray;
        unsigned int jkl = 0;
```

```
        //The software can be used in two mods: tests with map loaded by the
user and real time usage
        if(!pclAvailable && obstAvailable)  //Map input by the user
        {
            //No need to clear the previous obstacle list because it will be
constant
            for(unsigned int jj = 0; jj < obstacleList.size(); jj++)
            {
                //Creating the indexing of the markers
                stringstream st;
                st << jkl;
                std::string numObst = st.str();
                visualization_msgs::Marker marker;

                ObsList obsToAdd;
                obsToAdd = obstacleList.at(jj);

                //Header
                marker.header.frame_id = "/map";
                marker.header.stamp = ros::Time::now();
                marker.type = visualization_msgs::Marker::CYLINDER;
                marker.ns = obstString+numObst;
                marker.id = jkl;
                marker.action = visualization_msgs::Marker::ADD;

                //Pose and Orientation
                marker.pose.position.x = obsToAdd.obstX;
                marker.pose.position.y = obsToAdd.obstY;
                marker.pose.position.z = 0;
                marker.pose.orientation.x = 0.0;
                marker.pose.orientation.y = 0.0;
                marker.pose.orientation.z = 0.0;
                marker.pose.orientation.w = 1.0;

                //Header
                marker.scale.x = 2*obsToAdd.radius;//1.0;
                marker.scale.y = 2*obsToAdd.radius;//1.0;
                marker.scale.z = 0;

                //Colors
                marker.color.r = rObstacle;
                marker.color.g = gObstacle;
                marker.color.b = bObstacle;
                marker.color.a = aObstacle;

                marker.lifetime = ros::Duration();

                markArray.markers.push_back(marker);

                jkl = jkl + 1;
```

```
            }
            markerObs_pub.publish(markArray);
            obstAvailable = false;

        }
        else if(pclAvailable && obstAvailable) //Map created from pcl
        {
            //Delete the previous obstacle list first
            obstacleList.clear();

            pcl::toROSMsg(*obstDetector->cloud_filtered2,*pclMsgBack);
            pcl_pub.publish(pclMsgBack);

            //So then we can fill it with the new obstacles coming from the
analysis of the pcl
            for(unsigned int ii = 0; ii < obst.size(); ii++)
            {
                int numObstacleDivided = obst.at(ii).c.size();
                printf("Obstacle divided in %i points \n",
numObstacleDivided);
                for(unsigned int jj = 0; jj < numObstacleDivided; jj++)
                {
                    stringstream st;
                    st << jkl;
                    std::string numObst = st.str();
                    visualization_msgs::Marker marker;

                    //Creating the obstacle to insert in the obstacle list
                    ObsList obsToAdd;
                    obsToAdd.obstX = obst.at(ii).c.at(jj).x;
                    obsToAdd.obstY = obst.at(ii).c.at(jj).y;
                    obsToAdd.radius = obst.at(ii).radius;
                    obsToAdd.height = obst.at(ii).bbox.height;
                    obstacleList.push_back(obsToAdd);

                    //Header
                    marker.header.frame_id = "/map";
                    marker.header.stamp = ros::Time::now();
                    marker.type = visualization_msgs::Marker::CYLINDER;
                    marker.ns = obstString+numObst;
                    marker.id = jkl;
                    marker.action = visualization_msgs::Marker::ADD;

                    //Pose and Orientation
                    marker.pose.position.x = obsToAdd.obstX;
                    marker.pose.position.y = obsToAdd.obstY;
                    marker.pose.position.z = 0;
                    marker.pose.orientation.x = 0.0;
                    marker.pose.orientation.y = 0.0;
                    marker.pose.orientation.z = 0.0;
```

```
                    marker.pose.orientation.w = 1.0;

                    //Header
                    marker.scale.x = 2*obsToAdd.radius;
                    marker.scale.y = 2*obsToAdd.radius;
                    marker.scale.z = 0;//obsToAdd.height;

                    //Colors
                    marker.color.r = rObstacle;
                    marker.color.g = gObstacle;
                    marker.color.b = bObstacle;
                    marker.color.a = aObstacle;

                    marker.lifetime = ros::Duration();

                    markArray.markers.push_back(marker);

                    jkl = jkl + 1;
                }
            }
            markerObs_pub.publish(markArray);
            pclAvailable = false;
            obstAvailable = false;
        }
        if(goalAvailable) //If a target position is received
        {
            //Planning
            bool resPlanning = planTrees->plan(posArchie, goal, obstacleList);
            if(resPlanning) //If the planning was successful
            {
                //Initialization of marker for the planned trajectory
                visualization_msgs::Marker waypoints, trajectoryPlanned;
                visualization_msgs::MarkerArray waypointsArray,
trajectoryPlannedArray;

                waypoints.header.frame_id = "/map";
                waypoints.header.stamp = ros::Time::now();
                waypoints.ns = "waypoints";
                waypoints.action = visualization_msgs::Marker::ADD;
                waypoints.pose.orientation.w = 1.0;
                waypoints.scale.x = 0.01;
                waypoints.scale.y = 0.01;
                waypoints.id = jkl + 1;
                waypoints.type = visualization_msgs::Marker::POINTS;
                waypoints.color.r = rPoints;
                waypoints.color.g = gPoints;
                waypoints.color.b = bPoints;
                waypoints.color.a = aPoints;
                waypoints.lifetime = ros::Duration();
```

```
                trajectoryPlanned.header.frame_id = "/map";
                trajectoryPlanned.header.stamp = ros::Time::now();
                trajectoryPlanned.ns = "trajectoryPlanned";
                trajectoryPlanned.action = visualization_msgs::Marker::ADD;
                trajectoryPlanned.pose.orientation.w = 1.0;
                trajectoryPlanned.scale.x = 0.01;
                trajectoryPlanned.id = jkl + 2;
                trajectoryPlanned.type =
visualization_msgs::Marker::LINE_STRIP;
                trajectoryPlanned.color.r = rLinesT;
                trajectoryPlanned.color.g = gLinesT;
                trajectoryPlanned.color.b = bLinesT;
                trajectoryPlanned.color.a = aLinesT;
                trajectoryPlanned.lifetime = ros::Duration();

                ROS_DEBUG("There are %d waypoints \n", planTrees-
>waypoints.size());

                for(unsigned int kk = 0; kk < planTrees->waypoints.size();
kk++)
                {
                    ROS_DEBUG("Waypoint number %d \n", kk);
                    geometry_msgs::Point p;
                    p.x = planTrees->waypoints.at(kk).wP.x;
                    p.y = planTrees->waypoints.at(kk).wP.y;
                    p.z = 0;

                    waypoints.points.push_back(p);
                    trajectoryPlanned.points.push_back(p);
                }
                //Sending to RViz the markers of the planning
                waypointsArray.markers.push_back(waypoints);
                trajectoryPlannedArray.markers.push_back(trajectoryPlanned);
                markerPlanning_pub.publish(waypointsArray);
                markerPlanning_pub.publish(trajectoryPlannedArray);

                //Computation of the position of the foot
                WalkingGeneration *walkPrimitive = new WalkingGeneration;
                walkPrimitive->footStepPoseComputation(planTrees-
>waypoints,posArchie);

                //Initialization of the markers of the footsteps
                visualization_msgs::MarkerArray footPointCentersMarker;
                visualization_msgs::Marker centerFoot;
                for(unsigned int n = 0; n < walkPrimitive-
>feetCentersPose.size(); n++)
                {
                    stringstream st;
                    st << n;
                    std::string numStep = st.str();
```

```
                    //Header
                    centerFoot.header.frame_id = "/map";
                    centerFoot.header.stamp = ros::Time::now();
                    centerFoot.type = visualization_msgs::Marker::CUBE;
                    centerFoot.ns = "Foot"+numStep;
                    centerFoot.id = n;
                    centerFoot.action = visualization_msgs::Marker::ADD;

                    //Pose and Orientation
                    centerFoot.pose.position.x = walkPrimitive-
>feetCentersPose.at(n).x;
                    centerFoot.pose.position.y = walkPrimitive-
>feetCentersPose.at(n).y;
                    centerFoot.pose.position.z = 0;
                    tf::Quaternion q;
                    q.setRPY(0,0,walkPrimitive->feetCentersPose.at(n).theta);
                    centerFoot.pose.orientation.x = q.x();
                    centerFoot.pose.orientation.y = q.y();
                    centerFoot.pose.orientation.z = q.z();
                    centerFoot.pose.orientation.w = q.w();

                    //Header
                    centerFoot.scale.x = 0.225;
                    centerFoot.scale.y = 0.1;
                    centerFoot.scale.z = 0;

                    //Colors
                    centerFoot.color.r = rCenterFeet;
                    centerFoot.color.g = gCenterFeet;
                    centerFoot.color.b = bCenterFeet;
                    centerFoot.color.a = aCenterFeet;

                    centerFoot.lifetime = ros::Duration();

                    footPointCentersMarker.markers.push_back(centerFoot);


                }
                markerFootPolygon_pub.publish(footPointCentersMarker);
                archie_ai::footstepArray footstepMsg;
                footstepMsg.header.stamp = ros::Time::now();
                footstepMsg.header.frame_id = "/map";
                visualization_msgs::Marker footPointMarker;
                visualization_msgs::MarkerArray footPointArray;
                footPointMarker.header.frame_id = "/map";
                footPointMarker.header.stamp = ros::Time::now();
                footPointMarker.ns = "feetPlanned";
                footPointMarker.action = visualization_msgs::Marker::ADD;
                footPointMarker.pose.orientation.w = 1.0;
                footPointMarker.scale.x = 0.01;
```

```
                footPointMarker.scale.y = 0.01;
                footPointMarker.id = jkl + 3;
                footPointMarker.type = visualization_msgs::Marker::POINTS;
                footPointMarker.color.r = rPointsFeet;
                footPointMarker.color.g = gPointsFeet;
                footPointMarker.color.b = bPointsFeet;
                footPointMarker.color.a = aPointsFeet;
                footPointMarker.lifetime = ros::Duration();
                for(unsigned int kk = 0; kk < walkPrimitive->feetPose.size();
kk++)
                {
                    ROS_DEBUG("Footstep number %d \n", kk);
                    geometry_msgs::Point p;
                    p.x = walkPrimitive->feetPose.at(kk).x;
                    p.y = walkPrimitive->feetPose.at(kk).y;
                    p.z = 0;
                    geometry_msgs::Pose2D fs;
                    fs = walkPrimitive->feetPose.at(kk);
                    footstepMsg.poses.push_back(fs);
                    footPointMarker.points.push_back(p);
                }

                //Sending the markers of the footsteps to RViz and the
footsteps to the control node
                footstep_pub.publish(footstepMsg);
                footPointArray.markers.push_back(footPointMarker);
                markerFoot_pub.publish(footPointArray);

            }
            goalAvailable = false;
        }
    }
}
```

B2.4 ObstacleDetection.h

```
#ifndef INVERTEDCONEALGORITHM_H
#define INVERTEDCONEALGORITHM_H

#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include "HelperVariable.h"
#include <pcl/visualization/pcl_visualizer.h>
#include <tf/tfMessage.h>
#include <tf/transform_broadcaster.h>

class ObstacleDetection
{

public:

    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered2;

    ObstacleDetection();
    ~ObstacleDetection();
    std::vector<obstacles> newPCLAvailable(pcl::PointCloud<pcl::PointXYZ>
cloud, tf::Vector3 trans);
    std::vector<obstacles>
clusterAndApprox(pcl::PointCloud<pcl::PointXYZ>::Ptr filteredCloud);
    void passthroughFilter(pcl::PointCloud<pcl::PointXYZ>::Ptr inputCloud,
pcl::PointCloud<pcl::PointXYZ>::Ptr outputCloud, std::string axis, float
minimumAxis, float maximumAxis);
    void rotateVoxelGrid(pcl::PointCloud<pcl::PointXYZ>::Ptr in,
pcl::PointCloud<pcl::PointXYZ>::Ptr out, tf::Vector3 trans);

};

#endif // INVERTEDCONEALGORITHM_H
```

B2.5 ObstacleDetection.cpp

This file contains the code for the obstacle detection and approximation as explained in Chapter 6. The input point cloud if first filtered with a passthrough filter and then the Euclidean cluster extraction is performed. The dimensions of the clusters produced in this way are then computed and the obstacle is represented as a circle or as a series of intersecting circles.

```
#include "ObstacleDetection.h"
#include <pcl/filters/passthrough.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/common/common.h>
#include <pcl/common/transforms.h>
#include <ros/ros.h>
#include <pcl/filters/voxel_grid.h>
#include <string>
#include <cmath>

//Constructor
ObstacleDetection::ObstacleDetection()
{

}

//Destructor
ObstacleDetection::~ObstacleDetection()
{

}

//Function for the rotation of the point cloud from camera frame to base link
frame. It takes in input input point cloud in,
//output point cloud out and translation between the two reference systems
void ObstacleDetection::rotateVoxelGrid(pcl::PointCloud<pcl::PointXYZ>::Ptr
in, pcl::PointCloud<pcl::PointXYZ>::Ptr out, tf::Vector3 trans)
{
    Eigen::Affine3f transform_2 = Eigen::Affine3f::Identity();
    Eigen::Affine3f transform_3 = Eigen::Affine3f::Identity();
    float theta = M_PI/2;
    float thetaz = M_PI/2;
    pcl::PointCloud<pcl::PointXYZ>::Ptr outInter (new
pcl::PointCloud<pcl::PointXYZ> ());
```

```
    //Translation between the two reference systems
    transform_2.translation() << trans[0], trans[1], trans[2];

    //From the fixed reference system and the camera reference system it is
needed to rotate of 90 deg around x
    transform_2.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitX()));
    pcl::transformPointCloud (*in, *outInter, transform_2);

    //Then there is a rotation of 90° around z
    transform_3.rotate (Eigen::AngleAxisf (thetaz, Eigen::Vector3f::UnitZ()));

    //Set the output point cloud as the results
    pcl::transformPointCloud (*outInter, *out, transform_3);
}

//Passthrough filter. Filter the point cloud eliminating all the point which
have a coordinate on the input axis not in the interval set by maximum and
minimum
void ObstacleDetection::passthroughFilter(pcl::PointCloud<pcl::PointXYZ>::Ptr
inputCloud, pcl::PointCloud<pcl::PointXYZ>::Ptr outputCloud, std::string axis,
float minimumAxis, float maximumAxis)
{
    pcl::PassThrough<pcl::PointXYZ> pass;
    pass.setInputCloud (inputCloud);
    pass.setFilterFieldName (axis);
    pass.setFilterLimits (minimumAxis, maximumAxis);
    //pass.setFilterLimitsNegative (true);
    pass.filter (*outputCloud);
}

//Function for Euclidean Cluster Extraction and circle approximation
std::vector<obstacles>
ObstacleDetection::clusterAndApprox(pcl::PointCloud<pcl::PointXYZ>::Ptr
filteredCloud)
{
    //initializations
    pcl::PointCloud<pcl::PointXYZ>::Ptr fCloud;
    pcl::PCA<pcl::PointXYZ> pca;
    pcl::PointCloud<pcl::PointXYZ> fCloud1;
    fCloud = filteredCloud;
    std::vector<obstacles> obstDetected;

    // Creating the KdTree object for the search method of the extraction
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ>);
    tree->setInputCloud (fCloud);
```

```
    //Euclidean Cluster Extraction
    std::vector<pcl::PointIndices> cluster_indices;
    pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
    ec.setClusterTolerance (0.3); // 2cm
    ec.setMinClusterSize (5);
    ec.setMaxClusterSize (25000);
    ec.setSearchMethod (tree);
    ec.setInputCloud (fCloud);
    ec.extract (cluster_indices);

    ROS_INFO("I have %d indices \n", cluster_indices.size());

    int j = 0;
    for (std::vector<pcl::PointIndices>::const_iterator it =
cluster_indices.begin (); it != cluster_indices.end (); ++it) //For evey
cluster
    {
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
pcl::PointCloud<pcl::PointXYZ>);
        pcl::CentroidPoint<pcl::PointXYZ> centroid;
        pcl::PointXYZ c1;
        //ROS_INFO("Obstacle %d \n", j);

        //Centroid computation
        for (std::vector<int>::const_iterator pit = it->indices.begin (); pit
!= it->indices.end (); ++pit)
        {
            cloud_cluster->points.push_back (fCloud->points[*pit]);
            centroid.add(fCloud->points[*pit]);
        }
        cloud_cluster->width = cloud_cluster->points.size ();
        cloud_cluster->height = 1;
        cloud_cluster->is_dense = true;
        centroid.get(c1);

        //Bounding box computation: width, depth and height
        pcl::PointXYZ min_pt;
        pcl::PointXYZ max_pt;
        pca.setInputCloud(cloud_cluster);
        pca.project(*cloud_cluster, fCloud1);

        pcl::PointXYZ proj_min;
        pcl::PointXYZ proj_max;
        pcl::getMinMax3D (fCloud1, proj_min, proj_max);

        pca.reconstruct (proj_min, min_pt);
        pca.reconstruct (proj_max, max_pt);

        Eigen::Vector4f t = pca.getMean();
```

```
        ROS_INFO("centroid %f %f %f \n", c1.data[0], c1.data[1], c1.data[2]);
        ROS_INFO("minVector %f %f %f \n", min_pt.x, min_pt.y, min_pt.z);
        ROS_INFO("maxVector %f %f %f \n", max_pt.x, max_pt.y, max_pt.z);

        double width = proj_max.x-proj_min.x;
        double depth = proj_max.y-proj_min.y;
        double height = proj_max.z;
        double width1, depth1, height1;
        if(width < 0)
        {
            width1 = -width;
        }
        else
        {
            width1 = width;
        }
        if(depth < 0)
        {
            depth1 = -depth;
        }
        else
        {
            depth1 = depth;
        }
        if(height < 0)
        {
            height1 = -height;
        }
        else
        {
            height1 = height;
        }
        printf("Obstacle detected. width, depth height: %f %f %f \n", width1,
depth1, height1);
        double diffWidth = width1 - depth1;
        double diffDepth = depth1 - width1;
        printf("percentage width percentage depth %f %f \n", diffWidth,
diffDepth);
        int numObsDivided;
        if( diffWidth > 0.01) //Elongated obstacle width bigger than depth
        {
            numObsDivided = ceil(width1/depth1);
            printf("diffWidth > 0.1, numObsDivided %d \n ", numObsDivided);
            obstacles obs;
            std::vector<geometry_msgs::Pose2D> centers;
```

```
            //Centres computation
            for(int ii = 0; ii < numObsDivided; ii++)
            {
                pcl::PointXYZ c;
                pcl::PointXYZ cRotated;
                geometry_msgs::Pose2D centre;
                c.x = fmin(proj_min.x, proj_max.x)+(ii+1)*depth/2;
                c.z = height/2;
                c.y = fmin(proj_min.y, proj_max.y)+depth/2;
                printf("center computed: %f %f \n", c.x, c.y);
                pca.reconstruct(c,cRotated);
                centre.x = cRotated.x;
                centre.y = cRotated.y;
                printf("center computed after rotation: %f %f \n", cRotated.x,
cRotated.y);
                centers.push_back(centre);
            }
            obs.bbox.x = max_pt.x;
            obs.bbox.y = max_pt.y;
            obs.bbox.z = max_pt.z;
            obs.bbox.xm = min_pt.x;
            obs.bbox.ym = min_pt.y;
            obs.bbox.zm = min_pt.z;
            obs.bbox.width = width;
            obs.bbox.depth = depth;
            obs.bbox.height = height;
            obs.c = centers;
            obs.isCircle = true;
            obs.radius = (depth/2)*cos(M_PI/4);
            printf("radius: %f \n", obs.radius);
            obstDetected.push_back(obs);
        }
        else if(diffDepth > 0.01)  //Elongated obstacle. Depth bigger than
width
        {
            numObsDivided = abs(ceil(depth/width));
            printf("diffDepth > 0.1, numObsDivided %d \n ", numObsDivided);
            obstacles obs;
            std::vector<geometry_msgs::Pose2D> centers;

            //Centres computation
            for(int ii = 0; ii < numObsDivided; ii++)
            {
                pcl::PointXYZ c;
                pcl::PointXYZ cRotated;
                geometry_msgs::Pose2D centre;
                c.x = fmin(proj_min.x, proj_max.x)+width/2;
                c.z = height/2;
                c.y = fmin(proj_min.y, proj_max.y)+(ii+1)*width/2;
                printf("center computed: %f %f \n", c.x, c.y);
```

```
                pca.reconstruct(c,cRotated);
                centre.x = cRotated.x;
                centre.y = cRotated.y;
                printf("center computed after rotation: %f %f \n", cRotated.x,
cRotated.y);
                centers.push_back(centre);
            }
            obs.bbox.x = max_pt.x;
            obs.bbox.y = max_pt.y;
            obs.bbox.z = max_pt.z;
            obs.bbox.xm = min_pt.x;
            obs.bbox.ym = min_pt.y;
            obs.bbox.zm = min_pt.z;
            obs.bbox.width = width;
            obs.bbox.depth = depth;
            obs.bbox.height = height;
            obs.c = centers;
            obs.isCircle = true;
            obs.radius = (width/2)*cos(M_PI/4);
            printf("radius: %f \n", obs.radius);
            obstDetected.push_back(obs);
        }
        else //The obstacle can be represented as a single circle
        {
            obstacles obs;
            std::vector<geometry_msgs::Pose2D> centers;
            geometry_msgs::Pose2D center;
            center.x = c1.data[0];
            center.y = c1.data[1];
            printf("center computed: %f %f \n", center.x, center.y);
            centers.push_back(center);
            obs.bbox.x = max_pt.x;
            obs.bbox.y = max_pt.y;
            obs.bbox.z = max_pt.z;
            obs.bbox.xm = min_pt.x;
            obs.bbox.ym = min_pt.y;
            obs.bbox.zm = min_pt.z;
            obs.bbox.width = width;
            obs.bbox.depth = depth;
            obs.bbox.height = height;
            obs.c = centers;
            printf("no Obstacle division needed. Size of the center vector %lu
\n", obs.c.size());
            obs.isCircle = true;
            obs.radius = fmax(obs.bbox.width, obs.bbox.depth);
            printf("radius: %f \n", obs.radius);
            obstDetected.push_back(obs);
        }
        j++;
    }
```

```
    return obstDetected;
}

//This function is called by the point cloud callback in PlanningNode.cpp.
Receives the point cloud and filters it
std::vector<obstacles>
ObstacleDetection::newPCLAvailable(pcl::PointCloud<pcl::PointXYZ> cloud,
tf::Vector3 trans)
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr pntCloud(new
pcl::PointCloud<pcl::PointXYZ>);
    *pntCloud = cloud;
    ROS_INFO("Inverted con class. Cloud: width = %d , height = %d \n",
pntCloud->width, pntCloud->height);
    ROS_INFO("Cloud size: %d \n", pntCloud->size());
    pcl::PointCloud<pcl::PointXYZ> cloud_out;

    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered1(new
pcl::PointCloud<pcl::PointXYZ>);
    cloud_filtered2.reset(new pcl::PointCloud<pcl::PointXYZ>);

    //Calling passthrough filter
    passthroughFilter(pntCloud, cloud_filtered1, "z", 0.0, 5.0);

    //Point cloud rotation
    rotateVoxelGrid(cloud_filtered1, cloud_filtered2, trans);

    //ROS_INFO("Cloud filtered width and height: %d  %d \n", cloud_filtered2-
>width, cloud_filtered2->height);

    //Clustering and approximation
    std::vector<obstacles> obss = clusterAndApprox(cloud_filtered2);

    return obss;
}
```

B2.6 WalkingGeneration.h

```
#ifndef FOOTCOMPUTATION_H
#define FOOTCOMPUTATION_H
#include "HelperVariable.h"
#include <ros/ros.h>

class WalkingGeneration
{
public:
  float sx, sy;
  float px, py;
  geometry_msgs::Pose2D footPose;
  geometry_msgs::Pose2D footCenterPose;
  std::vector<geometry_msgs::Pose2D> s;
  std::vector<geometry_msgs::Pose2D> feetPose;
  std::vector<geometry_msgs::Pose2D> feetCentersPose;
  std::vector<geometry_msgs::Pose> hipCenters;

  WalkingGeneration();
  ~WalkingGeneration();
  void footStepPoseComputation(std::vector<waypoint> wPs,
geometry_msgs::Pose2D inPos);

};

#endif // FOOTCOMPUTATION_H
```

B2.7 WalkingGeneration.cpp

In this file the function which computes the footsteps is contained. The computation is performed for every couple of waypoints in the planned trajectory. First the walk parameters are computed and the, according to the method explained in Chapter 6, the footsteps are computed.

```
#include "WalkingGeneration.h"

//Constructor
WalkingGeneration::WalkingGeneration()
{
    ROS_INFO("Walking pattern generator started \n");

    px = 0;
    py = -0.1;
}

//Destructor
WalkingGeneration::~WalkingGeneration()
{

}

//Footsteps computation function. It is called in PlanningNode.cpp and it
takes in input the waypoint from the
//obstacle avoidance and the position of the robot coming from the S-PTAM
void WalkingGeneration::footStepPoseComputation(std::vector<waypoint> wPs,
geometry_msgs::Pose2D inPos)
{
    //Initializations
    unsigned int size = wPs.size();
    int index = 2;
    ROS_INFO("number of waypoint %d \n", size);
    geometry_msgs::Pose2D s0;
    geometry_msgs::Pose2D s1;

    //First foot position
    px = py + wPs.at(0).wP.x;//inPos.x;
    py = py + wPs.at(0).wP.y;//inPos.y;
```

```
    //Insertion of first foot position in the footstep vector
    footPose.x = px;
    footPose.y = py;
    footPose.theta = 0;
    feetPose.push_back(footPose);

    //First position of the centre of mass
    geometry_msgs::Pose hCn0;
    hCn0.position.x = wPs.at(0).wP.x;
    hCn0.position.y = wPs.at(0).wP.y;
    hCn0.position.z = 1.3;
    hipCenters.push_back(hCn0);

    //Second footsteps computation
    footCenterPose.x = px + 0.0375;
    footCenterPose.y = py;
    footCenterPose.theta = 0;
    feetCentersPose.push_back(footCenterPose);

    //First walk parameter
    s0.x = 0;
    s0.y = 0;
    s0.theta = 0;
    s.push_back(s0);

    //Second walk parameter
    s1.x = 0;
    s1.y = maximum_footstep_lenghtY;
    s1.theta = 0;
    s.push_back(s1);

    for(unsigned int ii = 0; ii < size-1; ii++)  //For every waypoint
    {
        //Compute heading of previous waypoint and current
        ROS_INFO("Considering Waypoint number %d and %d \n", ii, ii+1);
        ROS_INFO("Positions: %f %f %f %f %f %f \n", wPs.at(ii).wP.x,
wPs.at(ii).wP.y, wPs.at(ii).direction, wPs.at(ii+1).wP.x, wPs.at(ii+1).wP.y,
wPs.at(ii+1).direction);
        float distWPs = sqrt((wPs.at(ii+1).wP.x -
wPs.at(ii).wP.x)*(wPs.at(ii+1).wP.x - wPs.at(ii).wP.x) + (wPs.at(ii+1).wP.y -
wPs.at(ii).wP.y)*(wPs.at(ii+1).wP.y - wPs.at(ii).wP.y));
        int numSteps = (int)(round(distWPs/maximum_footstep_lenghtX));
        ROS_INFO("The distance between the two is %f and the number of steps
are %d \n",distWPs, numSteps);
        float theta = wPs.at(ii+1).direction;
        ROS_INFO("The waypoint number %d has theta : %f \n",ii+1, theta);
        float headDiff = theta - wPs.at(ii).direction;
```

```
        int numBeginningStep = s.size();
        if(headDiff > 0.01 || headDiff < -0.01) //If is a change of heading
between this waypoint and the previous one
        {
            //If it has to turn right and the current foot is the right one
            //or if it has to turn left and the current foor is the left one
            //Then three footsteps needs to be added because. One with the
previous heading and two with the new one
            if(((headDiff > 0) && (numBeginningStep%2==0)) || (headDiff < 0)
&& (numBeginningStep%2!=0))
            {
              //Walk parameters for the turn
              geometry_msgs::Pose2D snHeadChange;
              snHeadChange.x = 0;
              snHeadChange.y = maximum_footstep_lenghtY;
              snHeadChange.theta = wPs.at(ii).direction;

              geometry_msgs::Pose2D snHeadChange1;
              snHeadChange1.x = 0;
              snHeadChange1.y = maximum_footstep_lenghtY;
              snHeadChange1.theta = theta;

              geometry_msgs::Pose2D snHeadChange2;
              snHeadChange2.x = 0;
              snHeadChange2.y = maximum_footstep_lenghtY;
              snHeadChange2.theta = theta;

              s.push_back(snHeadChange);
              s.push_back(snHeadChange1);
              s.push_back(snHeadChange2);
            }
            else //In this case, just two footsteps with the new heading needs
to be added
            {
              //Walk parameters for the turn
              geometry_msgs::Pose2D snHeadChange;
              snHeadChange.x = 0;
              snHeadChange.y = maximum_footstep_lenghtY;
              snHeadChange.theta = theta;

              geometry_msgs::Pose2D snHeadChange1;
              snHeadChange1.x = 0;
              snHeadChange1.y = maximum_footstep_lenghtY;
              snHeadChange1.theta = theta;

              s.push_back(snHeadChange);
              s.push_back(snHeadChange1);
            }
        }
        unsigned int gg;
```

```
        //Compute walk parameter for every steps until the next waypoint
        for(gg = 0; gg < numSteps; gg++)
        {
            if(gg != numSteps-1)
            {
              geometry_msgs::Pose2D sn;
              sn.x = maximum_footstep_lenghtX;
              sn.y = maximum_footstep_lenghtY;
              sn.theta = theta;
              s.push_back(sn);
              ROS_INFO("s computed: %f %f %f %d \n",
sn.x,sn.y,sn.theta,s.size());
            }
            else
            {
              float rest = (distWPs/maximum_footstep_lenghtX)-numSteps;
              geometry_msgs::Pose2D sn;
              sn.x = maximum_footstep_lenghtX + rest*maximum_footstep_lenghtX;
              sn.y = maximum_footstep_lenghtY;
              sn.theta = theta;
              s.push_back(sn);
              ROS_INFO("s computed: %f %f %f %d \n",
sn.x,sn.y,sn.theta,s.size());
            }

        }

        //Walk parameter for last footstep
        geometry_msgs::Pose2D sLast;
        sLast.x = 0;
        sLast.y = maximum_footstep_lenghtY;
        sLast.theta = theta;
        s.push_back(sLast);

        index = index+gg;
        ROS_INFO("New Index %d , s size %d \n", index, s.size());
    }

    //Computation of footsteps based on the walk parameter
    ROS_INFO("Second Loop. Total number of steps: %d \n", s.size());
    for(unsigned int jj = 1; jj < s.size(); jj ++)
    {
        ROS_INFO("Considering footstep number %d \n", jj);
        ROS_INFO("s vector for the step: %f %f %f \n",s.at(jj).x, s.at(jj).y,
s.at(jj).theta);

        geometry_msgs::Pose hCn;
        hCn.position.x = hipCenters.at(jj-1).position.x +
cos(s.at(jj).theta)*s.at(jj).x;
```

```
        hCn.position.y = hipCenters.at(jj-1).position.y +
sin(s.at(jj).theta)*s.at(jj).x;
        hCn.position.z = 1.3;
        hipCenters.push_back(hCn);

        geometry_msgs::Pose2D pn;
        pn.x = hipCenters.at(jj).position.x + pow(-
1,jj)*sin(s.at(jj).theta)*s.at(jj).y;
        pn.y = hipCenters.at(jj).position.y - pow(-
1,jj)*cos(s.at(jj).theta)*s.at(jj).y;
        pn.theta = s.at(jj).theta;
        ROS_INFO("Computed foot pose: %f %f %f \n", pn.x, pn.y, pn.theta);
        feetPose.push_back(pn);

        geometry_msgs::Pose2D cn;
        cn.x = pn.x + cos(s.at(jj).theta)*0.0375;
        cn.y = pn.y + sin(s.at(jj).theta)*0.0375;
        cn.theta = s.at(jj).theta;
        feetCentersPose.push_back(cn);
    }
}
```

B2.8 HelperVariable.h

This file contains the definition of some variable used by the planning node.

```
#ifndef HELPERVARIABLE_H
#define HELPERVARIABLE_H

#include "geometry_msgs/Pose2D.h"
#include "geometry_msgs/Pose.h"
#include <pcl/common/pca.h>

#define maximum_footstep_lenghtX 0.3
#define maximum_footstep_lenghtY 0.1
#define tSup 1
#define grAcc 9.80665
```

```
struct BoundingBox
{
    float x;                                    //maximum x
    float y;                                    //maximum y
    float z;                                    //maximum z
    float xm;                                   //minimum x
    float ym;                                   //minimum y
    float zm;                                   //minimum z
    float width;                                //width of the obstacle
    float depth;                                //depth of the obstacle
    float height;                               //height of the obstacle
};


struct obstacles
{
    std::vector<geometry_msgs::Pose2D> c;       //Coordinates of the
obstacle
    BoundingBox bbox;                           //Bounding box of the
obstacle
    bool isCircle;                              //Is it representable as a
circle?
    float radius;                               //the radius of the
obstacle or of the series of obstacle is...
};

struct ObsList
{
    double obstX;                               //X coordinates of the
obstacle
    double obstY;                               //Y coordinate of the
obstacle
    double radius;                              //radius of the obstacle
    double height;                              //height of the obstacle
};


struct waypoint
{
    geometry_msgs::Pose2D wP;
    bool obsToStepOn;
    ObsList obs;
    float direction;
};


#endif // HELPERVARIABLE_H
```

B3: Services and messages

B3.1 foorstepArray.msg

This is the ROS message which stores the computed footsteps that is sent from the planning node to the control node.

```
Header header

geometry_msgs/Pose2D[] poses    #Array of footsteps
```

B3.2 clearMapAndPlanning.srv

This is the ROS service used to clear the markers on RViz

```
#Request: clear all

---
#Response: ok or no
uint8 res
```

B3.3 start_capture.srv

This service is used to start the stream from the camera

```
# request: start capture

---
#response: ok or no
uint8 status
```

B3.4 stop_capture.srv

This service is used to stop the camera stream

```
# request: stop capture

---
#response: ok or no
bool res
```

B3.5 new_map.srv

This is the service used to insert a user defined map. It has to be called with a list of obstacles positions

```
# request: new_map
float32[] ox
float32[] oy
---
#response: ok or no
uint8 status
```

B3.6 start_planning.srv

This service allows the user to start the planning sensing as input the end position

```
# request: start planning
float32 gx
float32 gy
---
#response: ok or no
uint8 status
```

Europass Curriculum Vitae

Personal information

First name(s) / Surname(s)	**Francesco d'Apolito, MSc**
Address	Landgutgasse, 1100, Wien
Mobile	(+43) 66488390625
E-mail(s)	Francesco.dApolito@ait.ac.at
Nationality	Italian
Date of birth	January 3rd, 1986
Gender	Male

Occupational field

Junior Scientist

Work experience

Dates	December 2017 – present
Occupation or position held	Junior Scientist
Main activities and responsibilities	Autonomous UAV Navigation, Airborne Collision Avoidance, Data Fusion Algorithms
Name and address of employer	Austrian Institute of Technology Giefinggasse 4, 1210, Wien

Dates	September 2015 – November 2017
Occupation or position held	Research Fellow
Main activities and responsibilities	Autonomous UAV Navigation
Name and address of employer	Austrian Institute of Technology Donau-CityStrasse 1, 1220, Wien

Dates	September 2012 – January 2014
Occupation or position held	Consultant
Main activities and responsibilities	Image Processing, GPU computing, On-board scientific software.
Name and address of employer	Planetek Italia s.r.l Via Massaua, 12, 70123 Bari, (BA) (Italy)

Projects

Dates	June 2016 – present
Project name	ROPA
Description	Autonomous navigation and collision avoidance for UAVs, multi-sensor data fusion for conflict detection
Name and address of the employer	Austrian Institute of Technology Giefinggasse 4, 1210, Wien

Dates	September 2015 – May 2016
Project name	RPA-AI
Description	Self-separation and Collision Avoidance algorithm development for UAVs
Name and address of the employer	Austrian Institute of Technology Donau-CityStrasse 1, 1220, Wien

Dates	August 2013 – January 2014
Project name	Solar Orbiter Data Processing Unit
Description	Implementation of the algorithm for the on-board data processing software for the Solar Wind Analyzer (SWA) suite.
Name and address of the employer	Planetek Italia s.r.l Via Massaua, 12, 70123, Bari, (BA) (Italy)
Dates	July 2011 - January 2012
Project name	AMALIA Team Rover GNC
Description	Participation to the Google Lunar X-Prize contest. Path planning and obstacle avoidance algorithm development for the lunar rover AROV.
Name and address of employer	University of Rome "La Sapienza", School of Aerospace Engineering

Publications

Schoerghuber, M.; d'Apolito, F.; Kopacek, P. (2014): Design and Optimisation of the Joint Controllers of a Humanoid Robot. *International Journal of Automation Austria*, Vol. 22, 2, 1-12.

Bauer, J.; Kopacek, P.; d'Apolito, F.; Kraeuter, L.; Dorna, I. (2015), Cost Oriented (Humanoid) Robots. *Proceeding of the 16th International Conference on Technology, Culture and International Stability (TECIS)*

d'Apolito, F.; Mehmeti, X.; Kopacek, P. (2016), Control of a Cost Oriented Humanoid Robot. *Proceedings of the 17th International Conference on Technology, Culture and International Stability (TECIS)*

d'Apolito, F., & Sulzbachner, C. (2017, October). Obstacle avoidance system development for the Ardrone 2.0 using the tum_ardrone package. In *Research, Education and Development of Unmanned Aerial Systems (RED-UAS), 2017 Workshop on* (pp. 49-54). IEEE.

d'Apolito, F. (2018). Obstacle Detection and Avoidance of a cost-oriented humanoid robot. *IFAC-PapersOnLine*, *51*(30), 198-203.

d'Apolito, F., & Sulzbachner, C. (2018, September). Collision Avoidance for Unmanned Aerial Vehicles using Simultaneous Game Theory. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)* (pp. 1-5). IEEE.

Education and training

Dates	March 2014 – January 2019
Title of qualification awarded	PhD, Mechanical and Mechatronic Engineering
Principal subjects / occupation skills covered	• Humanoid robots • Mechatronic systems • Resources Efficiency • End of life management Engineering management and Product development
Name and type of organization providing education and training	TU Wien, Faculty of mechanical and industrial engineering, Institute of Intelligent Handling devices and Robotics
Dates	September 2009 - January 2012
Title of qualification awarded	Master degree, Astronautical Engineering
Principal subjects / occupational skills covered	• Space Robotics. • Astronautical propulsion: nuclear, ion and plasma thrusters. • Spacecraft thermal stresses computation and spacecraft thermal control systems. • Astrodynamics and space flight mechanics. • Electronics for space application. • Structure and material for space application. • Participation to the AMALIA Team rover design **Thesis title**: "Algoritmo di guida e controllo per il rover lunare AROV" (Guidance and control algorithm for the lunar rover AROV)
Name and type of organisation providing education and training	University of Rome "La Sapienza", School of Aerospace Engineering

Dates	September 2005 – July 2009
Title of qualification awarded	Bachelor degree, Aerospace Engineering
Principal subjects / occupational skills covered	• Aerodynamics • Structural Mechanics and Applied Mechanics • Numerical Analysis • Structures and materials for aerospace application. • Aeronautic and space propulsion. **Thesis title**: "Analisi sperimentale nelle gallerie del vento al plasma" (Experimental analysis in plasma wind tunnels)
Name and type of organisation providing education and training	University of Rome "La Sapienza", Faculty of Aerospace Engineering

Personal skills and competences

Mother tongue(s): Italian

Other language(s):

	Understanding				Speaking				Writing	
	Listening		Reading		Spoken interaction		Listening		Reading	
English	C1	Proficient user	C1	Proficient user	C1	Proficient user	C1	Proficient user	C1	Proficient user
German	B1	Basic User	B1	Basic User	B1	Basic User	B1	Basic User	B1	Basic User

Technical skills and competences	Artificial Intelligence, Robotics, Aerospace Flight Mechanics, Space Robotics, Path Planning and Obstacle Avoidance Algorithms, Autonomous System, Sensor Fusion, Mechatronics, Computer Vision and Image Processing, CUDA Programming, Astrodynamics, Spacecraft Thermal Stresses Computation, Numerical Analysis, Simulations, Propulsion Systems, Control Engineering, Spacecraft Operations Planning, Space Structures, Basic knowledge of Astrophysics.
Computer skills and competences	*ROS, OS Linux and windows, Matlab, Simulink, Qt, C/C++, Fortran, Microsoft Office, GMAT, STK, Java.*
Artistic skills and competences	*Experience in acting.*
Driving licence(s)	B

References

Em. o.Univ.Prof. Dr.techn.Dr.hc.mult. Peter Kopacek
email: e-mail: peter.kopacek@tuwien.ac.at
tel: (+43) 1 58801 31800

Dr. Maria Pappalepore, legal representative Planetek Italia s.r.l

email: pappalepore@planetek.it

tel: (+39)0809644200

Fabio Curti, associate professor at University of Rome “La Sapienza”

email: fabio.curti@uniroma1.it
tel: (+39) 0688346423