Imaging Robot

Jonathan Dyssel Stets

Kongens Lyngby 2010
IMM-B.Sc-2010-42

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-B.Sc: ISSN 0909-3192

Summary

This thesis is focuses on designing an interface to control an industrial robot, with the goal of making it applicable for imaging. This can be done by creating two interfaces which will communicate together with a Command File, a file that contains the coordinates and directions of the robot arm. One of the interfaces, programmed in C++, works as a hardware and client interface. This interface controls the robot, light sources and an attached camera. The other interface, programmed in Matlab, controls the calculations of the robot arm coordinates and directions, and includes a graphic representation of how the robot will move. A library of functions for robot motions has been created; and the challenge has been to make the robot move on the surface of a sphere and to secure it from collisions with the surroundings or obstacles. The results show that it has been an adequate solution to create the interface with this architecture and consequently made the robot more accessible and easier to control.

Resumé

Denne opgave fokuserer på hvordan der kan designes et interface til en industrirobot, således at den kan bruges til imaging. Dette er gjort ved at lave to interfaces der kommunikerer sammen ved hjælp af en kommando fil der indeholder koordinater og retninger til robot armen. Det ene interface fungerer som hardware og klient interface, og er programmeret i C++. Dette interface styrer robotten, lyskilder og et, på robotten, fastgjort kamera. Det andet interface, programmeret i Matlab, styrer beregninger af robot armens koordinater og retninger, og kan desuden grafisk repræsentere hvordan robotten vil bevæge sig. Et bibliotek af funktioner for robot bevægelser er blevet opbygget, og udfordringen har været at få robotten til at bevæge sig på overfladen af en kugle og at sikre den fra at kollidere med omgivelserne eller hindringer. Resultatet viser at det har været en fyldestgørende løsning at lave et interface med denne arkitektur og har derfor gjort robotten mere tilgængelig og letter at styre.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring a Bachelor degree in engineering.

The thesis is about creating a user interface for an industrial robot with the purpose of making it applicable for imaging.

Lyngby, December 2010

Jonathan Dyssel Stets

Acknowledgements

I would like to thank my advisor Associate Professor Henrik Aanæs for helping me realise this project and giving me a unique opportunity to work with the robot. I would like to thank my advisor PostDoc Anders Lindbjerg Dahl for an excellent supervision at the friday meetings, and throughout the course. Furthermore I would like to thank Assistant Professor Line Katrine Harder Clemmensen and Professor Rasmus Larsen for guidance and assistance at the Friday meetings. I would also like to give thanks to Jannik Boll Nielsen for assisting parts of the theory and Natalie Gill for helping to revise.

Contents

Chapter 1

Introduction

This chapter will briefly introduce the problem, provide background information on how the system currently operates, and then show why it needs to be improved.

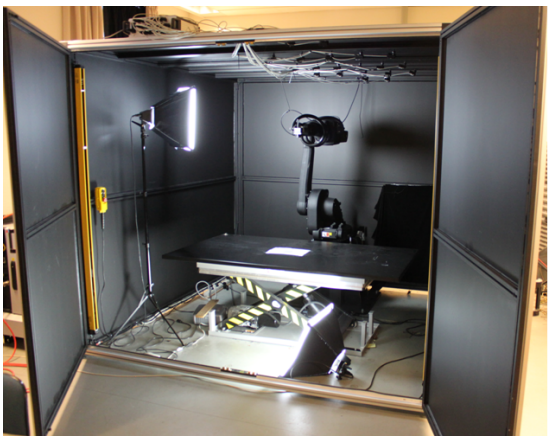
1.1 Background

Testing how well an image algorithm works is crucial before implementing or just doing further work on an algorithm. It is advantageous to determine how the algorithm behaves in a specific environment with for example low or high light intensity or situations where objects are imaged from different angles. To be able to test image algorithms, a lot of data is necessary in order to validate if an algorithm is legit. Large datasets of images therefore need to be gathered and this can be done in various ways. Collecting images of buildings in different lighted situations or with shadows cast and from various angles, could be taken in person with a camera. The newest idea is to recreate a similar scenario, by using small scale model houses, and controlled light sources in the form of high power light emitting diodes (LED's). A robot with a mounted camera can be used to move around the model scenario, capturing a large amount of images, while it is possible to control the shadow cast and light intensity along with the

exact position of the camera. The robot used is known as an industrial robot produced by ABB Robotics.

Figure 1.1: Industrial robot from ABB Robotics. The robot being used for imaging is painted matte black so almost no light is reflected. Source: http://www.manufacturingtalk.com/news/abd/abd224.html

The robot is placed in a closed cage to reduce the light intake, and the walls and the ceiling are painted in a matte black color, to minimize unwanted light reflections. The robot cage is often referred to as the robot cell.



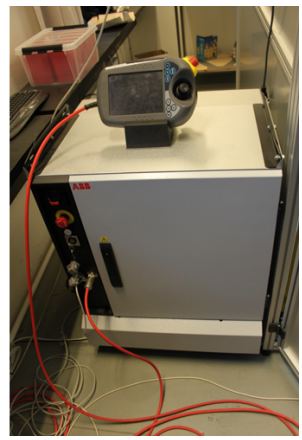


Figure 1.2: The robot cell on the left and the robot computer on the right. The robot cell has an infra-red sensor light grid, that immediately stops the robot from moving if crossed.

1.2 Motivation

When imaging an object, a camera is used to take pictures of the object from multiple directions. To avoid using multiple cameras and manually moving the camera around to different positions, a single camera is mounted on the arm of an industrial robot. Industrial robots are used for multiple purposes in the industry such as assembling products and move objects, which typically requires a quick operation with a great precision and accuracy. By using the industrial robot with a mounted camera, we can obtain a very powerful tool for imaging; since the robot can perform preprogrammed sequences of motions faster and more precise than a human, and without any assistance while its running. Using the robot also makes it possible to generate datasets for testing image related algorithms. Testing algorithms requires datasets that represent many different situations, which could be different lighting, and then afterwards comparing the results. The robot also enables imaging for an almost unlimited amount of data points with a very high precision, which for the human would have been almost impossible, or taken many valuable hours. An example of the robot in use, is the evaluation of interest point detectors independently of image descriptors [1]. Where the robot is used to generate datasets containing 60 scenes of a range of different objects, where each scene have 119 positions of the camera. The key point of this experiment is the repetition of motions and the precise location of the camera.

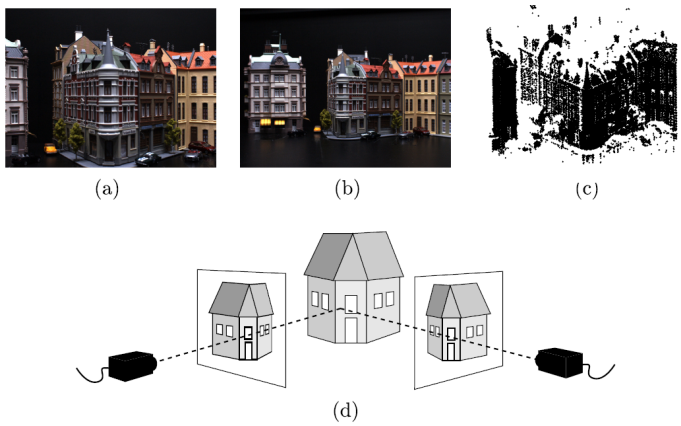


Figure 1.3: Example of a robot application. A scene is created and two images are cached, one close up (a), and one distant from another angle (b). (c) is a reconstructed 3D image of the scene, consisting 3D points. Using the geometric information of the scene from the 3D points, and the known camera positions, corresponding interest points can be found (d). Figure from [1], Figure 1.

Another application of the robot is analysing dairy and meat products. It is possible to analyse the quality of a meat or dairy product by using a laser diode to light up the product, and a imaging sensor that ranges within the near infra-red and short wave infra-red spectral, to capture the results. The size and concentration of the particles in the product will result in differences of how the laser light is absorbed and reflected. Regardless of whether the robot would be carrying either the imaging sensor or the laser light it this would require a complete control of the robot.

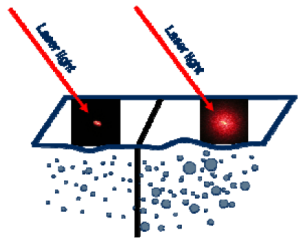


Figure 1.4: Laser beam pointed at the surface of a dairy or meat product. The optical properties of the surface and subsurface helps determining the quality of the food product with no physical contact and therefore no contamination risk. Figure from [7], Figure 1.

The robot has, as earlier mentioned, no user interface designed for the tasks mentioned above. It is therefore fairly complicated for an inexperienced user to operate without understanding the syntax of the existing software and also programming every series of motions the robot has to make. This process is not only time-consuming, but also entertains a high risk: the user could potentially give the wrong instructions to the robot, which could result in damage to either people, the robot or their surroundings. Since the robot is used in the industry to carry relatively heavy objects, the damage it could do to either a person or equipment could be quite extensive. Damages occur when the robot is programmed to do motions that either outreaches its working cell, or when the path between two points in a motion goes straight through an obstacle located within the robot cell. Of course, these errors are predictable, but easy to overlook, and it would therefore be nice if any collisions with objects or obstacles could be detected before running the robot.

Programming an interface for the robot will result in a more effective and flawless system, open up for a wider spectrum of usability, and expand the range of users who can use the robot. Creating a solid interface for the robot, would open the possibilities for various other applications.

1.3 Problem statement

On the background of the needs described in the motivation section, the main goal of this project is to create a functional user interface for the robot. The overall problem statement is therefore as follows:

Create an interface to control the IMM imaging robot, an industrial robot with attached devices for imaging, within aim of making it more accessible.

In short, the interface will be programmed in Matlab and C++. The interface has to be designed so that it can be used for multiple purposes, and needs to be designed so that it is not possible for the user to collide the robot with the surroundings. There is of course many ways to find a solution to the problem, so the problem will be further analysed in the next chapter, and the exact method to solve the problem will be discussed.

1.3.1 Why is it a problem

The Robot itself is not very difficult to program, since it has driver software with predefined operations. To move the robot from one point to another is done simply, by just sending a set of coordinates to the new position. The robot can either be controlled with an advanced remote control, or by sending commands through a Local Area Network Protocol. This gives a flexibility to make the robot move anywhere within its scope. This flexibility also makes it harder for a potential new user of the robot to use it, and to make it do simple or more advanced tasks without having to go through a lot of trouble learning the robots language and syntax. Another aspect of this, is security. The robots flexibility makes its dangerous to work with, using it wrong can cause a lot of damage to people, machinery and the surroundings. In order to avoid this, a good solution for predicting the robots behaviour is necessary. The more difficult aspect of designing the interface is mostly designing algorithms to calculate the geometry, but implementing this so it actually works in practice is also going to be a challenge.

1.4 Delimitation

The purpose of this project is creating an interface for the robot where the central focus is on the robot and the robot motions. Since this is a practical

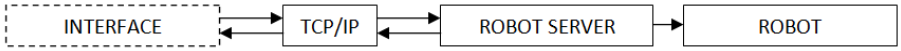


Figure 1.5: Diagram over the connection. The interface sends commands to the robot via a TCP/IP protocol. The robot computer and server receives the commands from the protocol and sends electrical signals to the robot to control the motor-rotations. Programming advanced paths and sending them to the robot, along with predicting the robots behaviour will is a challenge.

project, the main challenges in this project is what method to choose to realise this project, and then implement it. Geometric considerations and how to realise these, are weighted higher than going into details about how specific imaging algorithms works. The project is based on existing software that enables sending coordinates and rotations to the robot, how the software works is studied, but not how the socket is programmed and how the RAPID code is working. RAPID is the software and language used to program the robot.

1.5 Thesis structure

The background, motivation and problem statement have now been stated. The thesis is divided into seven chapters. The next chapter will describe and discuss which method should be used to solve the problem stated above. Afterwards a short introduction to the theory used to solve the problem will be examined and afterwards implemented in a solution. The solution will be tested and evaluated and finally discussed. The thesis will end with suggestions of future work and a conclusion.

The source code of all the software can be found at :

http://www.student.dtu.dk/~s072112/bachelor/

The following should be remembered: The camera mounted on the robot arm, will in this report often be referred to as the "tool", since this camera also could be replaced with other tools such as a laser.

The robot robot cage is often referred to as the robot cell.

Unless other is mentioned, then a right handed coordinate system, with Z axis pointing up is used throughout the thesis, since this is how the robot is defined.

Chapter 2

Choice of Method

This section will discuss the extent of the problem and also subdivide the problem into smaller steps.

2.1 Problem analysis

The final expected outcome of this project, is as stated in the problem statement, to create a user interface to the robot, to make it applicable for a wide range of tasks while simultaneously remaining user friendly for inexperienced users. Creating an interface for the robot, is also a part of a larger problem, which is about testing algorithms and analyse food products. The solution for the specific problem this report is about can also could be used as a tool for solving other problems in the future. The obstacles associated with this task include designing a graphic user friendly interface, that works well with the robot and stabilizing it so that no unexpected errors occur or are mistakenly created by the user. Since this is a rather specific project, there is not a lot of experience in previous research about how to create a good interface for a robot that has to perform imaging. Although there is some similar projects about using robots for imaging, there is a lack of documentation regarding good functional user interfaces. The problems statement describes a problem that could be solved in a variety of ways, since it does not specify what method to use, and exactly how


Figure 2.1: Example of a similar project from University of Waterloo, Canada. The robot is used for multiple purposes, such as human-robot interfaces and interaction[5].

the problem should be solved. The problem is not just one problem, but is really several problems that can be divided into smaller steps or subtasks. Dividing the problem into subtasks, both makes it more manageable and also possible to solve it with partial solutions. When dividing the problem into smaller subtasks, it automatically enables half-way, or partial solutions, which will safeguard the outcome of the project from a completely failing, if something goes wrong in a single subtask. Although a partial solution is not the ultimate goal, it is helpful if something unpredictable arises, turns out to be impossible, or at least more difficult than expected. Another advantage of subdividing the problem, is to go from working on a problem that has rarely been solved, to go to a set of smaller problems that have been worked on before.

2.2 Methodology

On the basis of problem analysis, to design a good interface for the robot, a method used to solve the problem will be discussed in this section. The most important part of the interface, is being able to control the robot motions, but it actually consists of 3 main parts; the robot motion control, the light (LEDs) and the camera. To control the robot, it is necessary to calculate motions for the robot and send them to the robot server which will translate them into stepping motor rotations. The existing robot socket is programmed in C++ and so is the LED and camera control, because of that, it would be obvious to continue programming the all the hardware controls in C++. Calculating the coordinates, the different motions and displaying the graphically, is by far the simplest to do in Matlab and the build in GUI (Graphical User Interface) editor. This could also be done in C++, but programming a GUI in C++ is far more time consuming. Communication between the two interfaces will be

solved by generating a Command File that will contain the information about the coordinates calculated in the Matlab interface, and can be loaded with the C++ interface and send to the robot from there. The connection between the interfaces and the robot, the LEDs and the camera is described in 2.2.

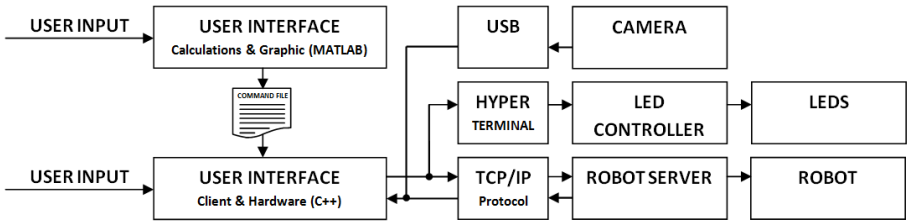


Figure 2.2: The user interface programmed in Matlab calculates the coordinates, and saves it to a Command file. The command file is opened in the user interface programmed in C++, where it sends the coordinates to the robot server via the Local Area Network (LAN). The user interface programmed in C++ also controls the camera and the LEDs.

When using a robot for imaging, there are a set of motions which can be helpful in a variety of situations. When imaging a simple 3 dimensional object, for example a circular motion around the object is used to cover its full surface of the object. When imaging a more complex three dimensional objects, it can be necessary to 'shoot' the object from various angles, to be able to cover all details. This can be done by moving the camera in a spherical motion around the object. Another perspective of imaging is texture and surface scans. This task can be completed by making the robot move on a line or a surface along an object. When completing these tasks, it is important that the robot do not harm anything by colliding with any objects or surroundings. That is why a collision map should be designed to avoid any damages. The method I have chosen to approach the problem, is to divide the interface into smaller subtasks. Each subtask is divided, so that the first tasks are simple, and the later tasks are more complex to implement. That way, experience will be obtained solving the each subtask, and it will give me more experience to solve the harder problems that may occur later on in the process. Dividing the project into smaller tasks, also makes sure that there are success along the way, and this avoids the risk of an outcome without positive results at all. The subtasks are divided, so that each task is a motion that will be implemented in the final user interface, and will make the overall process more manageable.

2.2.1 Subtasks

The conclusion of the discussion is, that the robot interface needs to have a series of basic motions, along with a solid interface design. Therefore, the project will be divided into a series of subtasks listed below.

	Tasks	Risk
1	Design collision map	2
2	Move robot in line	1
3	Move in $\angle$ x circle	3
4	Move robot in a half-sphere.	4
5	Design Client and Hardware Interface	4
6	Design Calculation and Graphic Interface	4

A very strong library of simple functions needs to be programmed, and will be used by the calculation and graphics interface. This is shown in 2.3. As earlier mentioned, programming the interface in Matlab will be a good solution, since Matlab can calculate geometry and visualise the results easily. The subtasks are chosen on the basis of what kind of motions we want the robot to do. The motions are very basic, and they can therefore be used to cover almost any kind of task in the process of imaging an object. The basic motions are described more detailed in the next section.

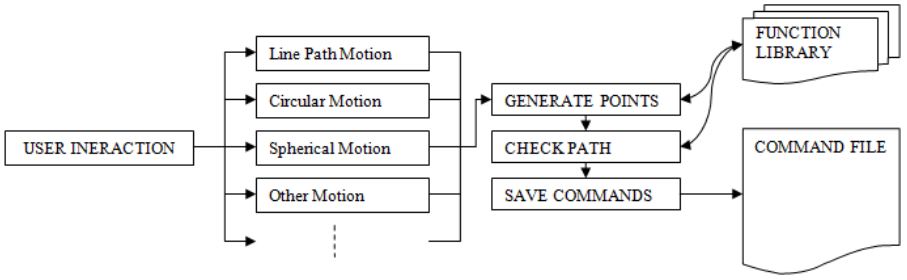


Figure 2.3: Interface Diagram

2.2.2 Subtasks description

1. Design collision map A collision map, containing the coordinates where the robot is not allowed to be within, needs to be designed.

2. Move robot in line Enter start and end coordinate of line, and enter how many images to be taken on the line. Enter a direction the camera is pointing.

3. Move in ∠ x circle This is an extension of **3.**. The angle of the circle can be entered, thereby there can be mad a half circle, a quarter circle or similar.

4. Move robot in half-sphere Enter sphere radius and center. Choose number of data points of the half-sphere from a predefined list. Camera is always pointing in direction of the sphere center.

5. Design Client and Hardware Interface Description of Hardware interface. Expandable. Terminal like program. Robot joggin, read file etc..

6. Design Calculation and Graphic Interface Description of graphic interface. Easy, and powerfull. Few buttons. Simple to program. Simple to operate. Expandable.

Chapter 3

Theory

This section will go through the basic theory needed to create an interface for the robot. This includes how the robot functions, how imaging is done and how the geometric calculations are approached.

3.1 Geometry

3.1.1 Spherical coordinate system

The spherical coordinate system is used in many applications, such as geographic calculations and 3 dimensional computer graphics, and becomes handy when dealing with spherical and circular objects and rotation angles. The spherical coordinate system is introduced in this project because it simplifies calculations of rotations and directions in a three dimensional space.

A point p in the spherical space is described with three coordinates (r, θ, φ) where r is the euclidean distance from the origin o to p, θ is the angle from the z axis to the vector $\bar{OP}$ and φ is the angle from the x axis to the projection of $\bar{OP}$ on the xy-plane.

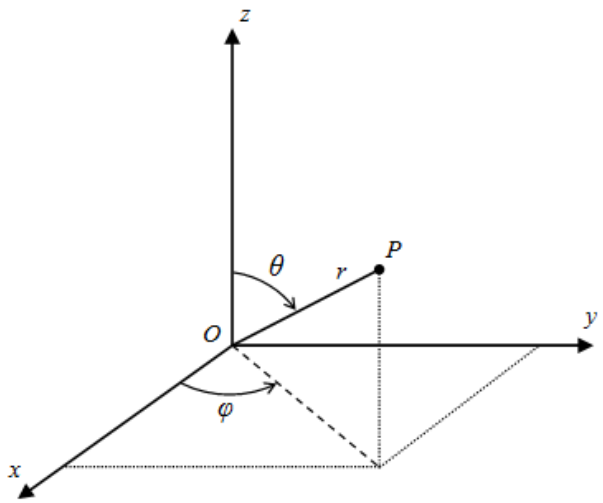


Figure 3.1: The spherical coordinate system represented in a Cartesian coordinate system.

The transformation between the two systems can easily be calculated using simple trigonometric expressions. Transforming from the spherical coordinate system (r, θ, φ) to the Cartesian coordinate system (x, y, z) can be done as follow:

$$\begin{aligned} x &= r\sin\theta\cos\varphi \quad \theta \in [0, \pi] \\ y &= r\sin\theta\sin\varphi \quad \varphi \in [0, 2\pi] \\ z &= r\cos\theta \quad r \in [0, +\infty] \end{aligned} \tag{3.1}$$

Reversing the transformation can be done with the expressions below:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \theta &= atan2(z, sqrt(x^2 + y^2)) \\ \varphi &= atan2(y, x) \end{aligned} \tag{3.2}$$

Where atan2 is a special version of *arctan* that can iterate and check for specific cases. atan2 can be written as an if statement, described with the following equations:

$$\begin{cases} arctan(\frac{x}{y}) & x > 0 \\ \pi + arctan(\frac{x}{y}) & y \geq 0, x < 0 \\ -\pi + arctan(\frac{x}{y}) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases} \tag{3.3}$$

The transformations stated above will become handy in the further work, and is from [4] and [3].

3.1.2 Distributing points on a sphere

Distributing points evenly on a sphere is a well known mathematical challenge, and there exists a number of approaches of how to obtain the best distribution of points among the surface of a sphere.

1. Equal area points Introduced in [12], the method distributes points, so that there is an equal area between the points.

2. Geodesic sphere A geodesic sphere is also known from 3D graphics, and it is essentially a sphere created by triangles. It begins by creating an icosahedron and then dividing each side of the triangles of the icosahedron into sub-triangles. The more triangles the icosahedron is divided into, the more points distributed. This algorithm provides an almost exact distribution of points on the sphere, however, only supports a finite row of numbers.

3. Move and fit points Described in [8]. Generate the number of wanted points, and distribute them randomly on the sphere. The algorithm now moves the points around until they have an almost to even distance from one point to its nearest neighbours. This algorithm supports any number of points, and gives a close to even distribution.

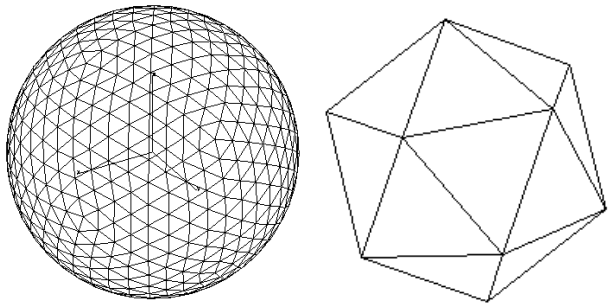


Figure 3.2: Left side shows a icosahedron, and right side shows a geodesic sphere. Starting with the icosahedron and dividing each triangle in three new triangles, leads to the geodesic sphere.

4. Spiral Method The basic principle behind this method is to create a spiral of points surrounding and projected on to the surface of a sphere using the spherical coordinate system. This method can use any number of points, and works well for a large number of points, as well as smaller amount of points. Two different approaches to this can be found in the appendix.

I have chosen to evaluate the approach introduced by Rakhmanov, Saff and Zhou in [11] pp. 9-10. This method described is fairly simple and very permissive to any number of points. This method also has the advantage of generating the points in an order that can be adapted directly as a robot path, since the robot will move in the spiral path, and does not have to enter the circle radius as any point.

The first step is to create the counting-variable h_k which will generate a series of points between -1 and 1 with even spacing in between,

$$h_k = -1 + \frac{2(k-1)}{(N-1)} \quad 1 \leq k \leq N \tag{3.4}$$

where N is the number of points. With the counting-variable h_k, the first spherical coordinate θ can be generated using arccos().

$$\theta_k = arccos(h_k) \tag{3.5}$$

Notice that the spiral then starts from the bottom and moves up the $z - axis$

since θ_k runs from π to 0.

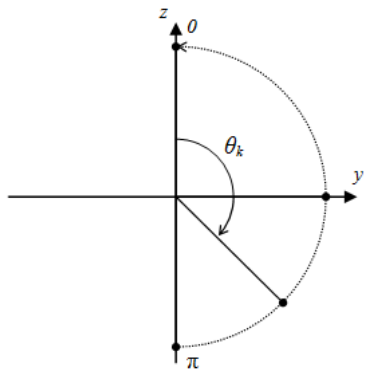


Figure 3.3: θ starting in π and moving up to 0.

Now φ is calculated:

$$\varphi_k = (\varphi_{k-1} + \frac{3.6}{N}\frac{1}{\sqrt{1-h_k^2}})(mod 2\pi) \quad 2 \leq k \leq N-1, \varphi_1 = \varphi_N = 0 \quad (3.6)$$

The N amount of points are now distributed on the surface of the sphere.

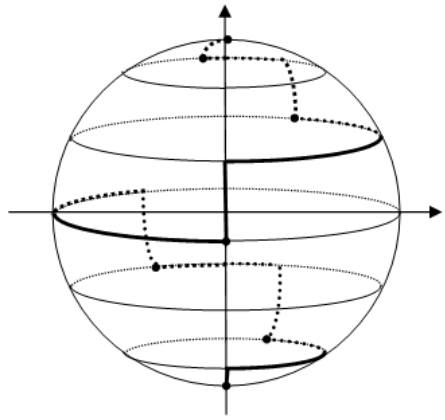


Figure 3.4: The points are distributed on the surface of the sphere following a spiral path.

3.1.3 Point in Square

A fairly simple method exists to determine if a point is within a square. It constitutes as the following:

Let a square be defined by four corner points $A = (x_A, y_A, z_A)$, $B = (x_B, y_B, z_B)$, $C = (x_C, y_C, z_C)$ and $D = (x_D, y_D, z_D)$ and $P = (x_P, y_P, z_P)$. Notice that all five points are in the same plane.

Calculating the the area A of the four triangles ABP, BCP, CDP and DAP, then adding them together and comparing it with the area of the square will indicate whether the point is inside or outside of the square. see 3.5. If the total area of the triangles combined is larger than the area of the square, this means that the point P is outside the square. Conversely, if this means that if the two areas are equal, the point P must be either inside the square or on the borderline.

$$\begin{aligned} A_{ABP} + A_{BCP} + A_{CDP} + A_{DAP} = A_{ABCD} &\quad \rightarrow \text{P is in square} \\ A_{ABP} + A_{BCP} + A_{CDP} + A_{DAP} > A_{ABCD} &\quad \rightarrow \text{P outside square} \end{aligned} \tag{3.7}$$

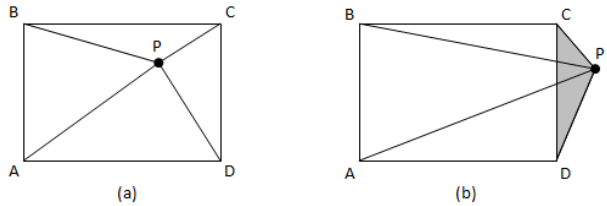


Figure 3.5: (a) Point inside square, the area of the four triangles adds up to the area of the square. (b) Point outside square, the area of the four triangles adds up to a larger area than the square.

3.1.4 Obtaining a plane from 3 points

To determine a plane in the 3 dimensional vector space, 3 points are needed. Three points are given, $A = (x_A, y_A, z_A)$, $B = (x_B, y_B, z_B)$ and $C = (x_C, y_C, z_C)$. To write up the equation of the plane, the normal vector $(\vec{n})$ to the desired plane must be be determined. This will be found by the crossproduct of $\vec{AB}$ and $\vec{AC}$. The normal vector $\vec{n}$ and one of the four points can now be inserted into the scalar equation of the plane:

$$\vec{n} = \vec{AB} \times \vec{AC} \tag{3.8}$$

$$P: \ x_n(x - x_A) + y_n(y - y_A) + z_n(z - z_A) = 0 \tag{3.9}$$

The equation of the plane, comes from the statement that the following equation should be satisfied for all points R in the plane:

$$(R - A) \cdot n = 0 \tag{3.10}$$

3.2 Quaternions and Euler angles

The robot tool orientation , or rotation, is represented in quaternions, which, in the 3D space, is the most compact way to express an orientation [9] pg. 24. Quaternions defines an element in $\mathbb{R}^4$, a 4 dimensional vector space, and has the notation

$$q = (q_0, q_1, q_2, q_3) \tag{3.11}$$

where q_0, q_1, q_2 and q_3 are scalars,and also called the components of the quaternion [6]. Representing the quaternion in $\mathbb{R}^3$, is done with a vector part

$$\vec{q} = \vec{i}q_1 + \vec{j}q_2 + \vec{k}q_3 \tag{3.12}$$

so that the quaternion is defined as the sum of the scalar part q_0 and the vector part:

$$q = q_0 + \vec{q} = q_0 + \vec{i}q_1 + \vec{j}q_2 + \vec{k}q_3 \tag{3.13}$$

Using quaternions for the robots' rotations provides some major advantages. For example, the mathematics will facilitate the obtainment of a smooth interpolation is a lot easier because of the mathematics. As earlier mentioned, quaternions is a very compact way to represent rotations. Finally there is no danger for gimbal lock to occur, since quaternions operate in a 4 dimensional space [2]. Gimbal lock and interpolation will be described later on in this chapter.

The existing software uses an alternative rotation format, namely Euler angles, instead of quaternions. Euler angles is another way of describing rotations in a 3 dimensional space. The RAPID software has a function to receive robot rotations in Euler angles[10], so no conversion is necessary to calculate. RAPID is the software and language used to program the robot.

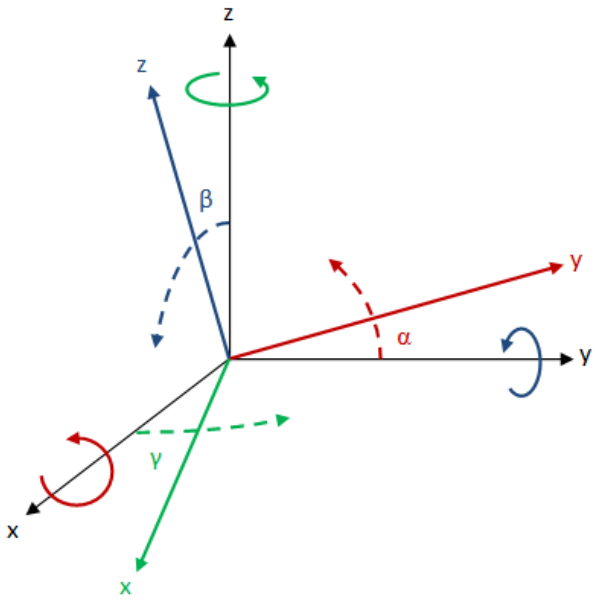


Figure 3.6: Euler angles in a Cartesian coordinate system.

Euler angles are described with tree parameters, which are rotations of respectively the x-, y- and z-axis. The following applies to the Euler angles:

Rotating about the x-axis, by the angle α brings y into z.
Rotating about the y-axis, by the angle β brings z into x.
Rotating about the z-axis, by the angle γ brings x into y.

Unlike the quaternions, Euler angles does not avoid gimbal lock. To understand

how the gimbal lock, we first need a short introduction to how the robot works.

3.3 How the robot works

3.3.1 Coordinate Systems

Since the robot has multiple rotation axis, a few different coordinate systems are introduced. The main coordinate is called the Base coordinate system. This has its zero at the base of the robot, where it is mounted to the floor. The coordinates in this system, describe the robots TCP (tool centre point). The TCP is the point of the mounted tool, in this case a camera is used for imaging. The TCP is visualised with a small metal pointer located close to the camera lens.

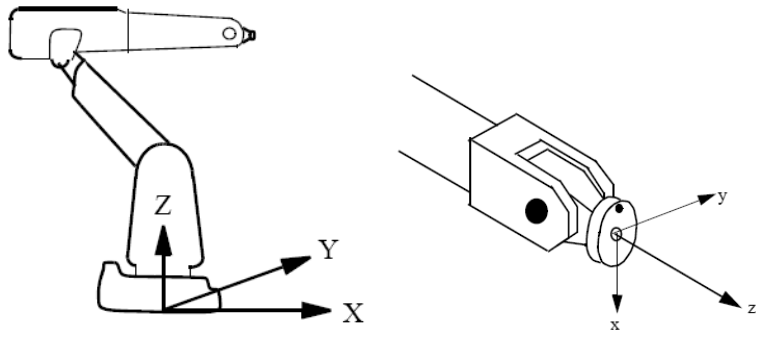


Figure 3.7: Illustrations show the two important coordinate systems. The base coordinate system describing the location, and the wrist coordinate system describing the direction.

Another important coordinate system that needs to be introduced, is the Wrist coordinate system. This decides the direction in which the tool is pointing. In this case, it is deciding in which direction the camera lens is pointing. The coordinate system is expressed in Euler angles.

The robot also allows for work in other user defined coordinate systems. The connection between the base coordinate system, the wrist coordinate system and user coordinate system is described in the world coordinate system. This coordinate system, is used as a common coordinate system to transform from one system to another.

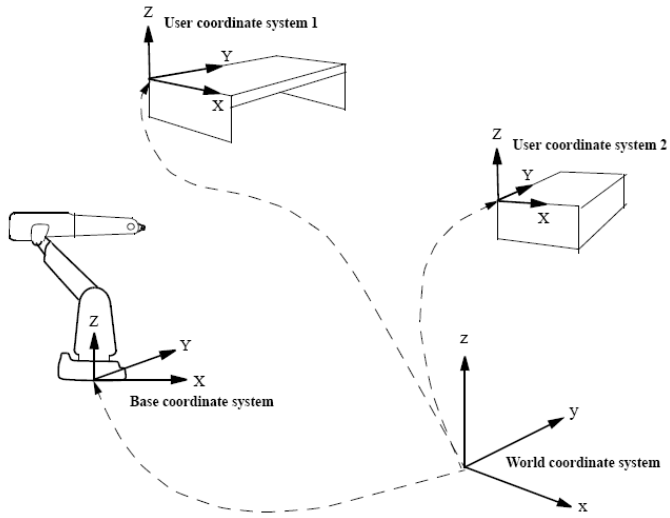


Figure 3.8: The relation between the Base coordinate system, the wrist coordinate system and other user defined systems, is called the world coordinate system.

The world coordinate system is typically used when multiple robots are working together in the same environment. Therefore, the world coordinate system is not used further in this report, but should not be forgotten if another robot should be implemented in the system.

3.3.2 Interpolation

There are a few ways of choosing the interpolation of the movement. The interpolation defines how to move from one point to another. This can be done in a few different ways.

Joint Interpolation This joint interpolation is used when the moving path does not need to be too accurate. The advantage of this motion is that it can be done quite quickly, with very few operations. The disadvantage of this interpolation, is that the path of the robot is less predictable, and consequently there will be a bigger risk of unpredicted collision. This movement gives a linear motion in the axis space, but a non-linear movement in the base coordinate system, and the direction of the tool is not specified.

Linear Interpolation The linear interpolation allows the robot to move in a straight line from one point to another. Moving the robot this way, slows the process because it has to move non linearly in the axis space, in order to move linearly in the base coordinate system. Another disadvantage is that there will be a higher risk of a Gimbal lock, since the motions of the axis are rather unpredictable. The advantage of this motion, is that the path is very predictable, and collision with the surroundings can therefore easily be predicted too. The orientation of the tool, is constant during the motion, but can also be specified.

Circular Interpolation The circle interpolation makes the robot move in a circle path defined by three points; start point, destination point, and a circle point. The circle point is the supportive point that sets the curve of the circle path. This path has the advantage to be predictable, but slow as the linear interpolation, since the rotations of the axis are non linear. The direction of the tool can either be set to or be constant during the motion.

Looking at the three different types of the interpolations, the linear interpolation and the circular interpolation, will be helpful in the design process. Since the main goal of the result is not focused on speed, but accuracy, we have to discard the joint interpolation, to avoid any unpredicted damages that may occur.

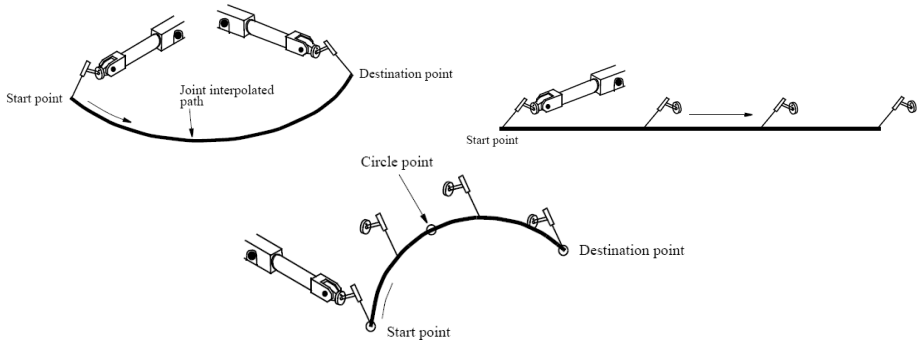


Figure 3.9: Paths of the 3 different interpolations.

3.3.3 Tool direction

As earlier mentioned, the direction of the tool mounted on the robot arm is described with Euler angles. In 3.10 it is showed how the camera depends on each angle. Since the z-axis goes through the camera lens, this axis will decide how the captured image is rotated. It is therefore only the rotations about the

x and y-axis that interferes with the direction of the camera. To simplify the calculations of the tool, the γ rotation is neglected. To calculate the rotations of α and β, the spherical coordinate system will be used.

Figure 3.10: Camera direction in Euler angle system.

Comparing 3.2 and 3.10 a method from translating between the coordinate systems can be done very simple:

$$\begin{aligned} \alpha &= 180 - \varphi \\ \beta &= -\theta \end{aligned} \tag{3.14}$$

3.3.4 Gimbal Lock

Gimbal lock is a state that occurs in a 3 dimensional space, where 2 of the 3 axes are aligned and consequently limit the motion of the axis from a 3 dimensional space to a 2 dimensional space, because one degree of freedom is lost. In air- and spacecraft design where gyroscopes are used, is it very important to avoid "locking" the gimbals because the unpredictability of the system could potentially result in fatal consequences. The gimbal lock problem can be fixed by adding on a 4th gimbal to the system, giving it an extra degree of freedom and still allowing to move in 3 dimensions when 2 of the gimbals are aligned.

A gyroscope consists of three rings , where each ring represents a rotation around an axis. Illustrated in 3.11 are two gyroscopes, the right one has the outer ring representing the x axis pointing out of the paper, the y axis is the middle ring

and pointing to the right, the z axis is the inner ring and is pointing up just as in 3.6. The right gyroscope has the y axis rotated so that the x and the z axis are aligned. This results in loss of one degree of freedom, and "locks" the gyroscope so is has to move out of this state in order to function normally again [2].

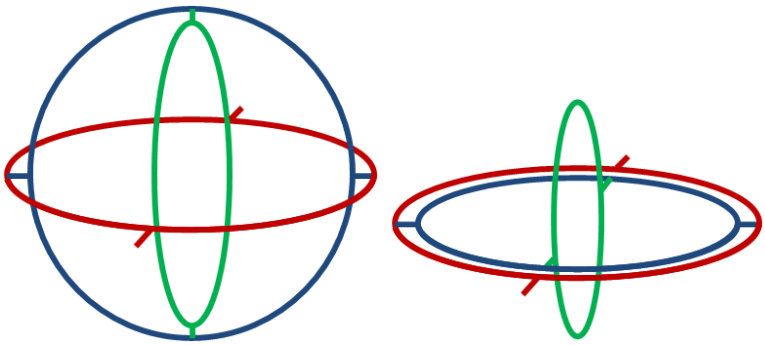


Figure 3.11: Gimbal lock phenomena. Left gyroscope shows the gimbals with 3 degrees of freedom. Right gyroscope shows how two gimbals are aligned, and therefore a loss of one degree of freedom.

In robotics, a gimbal lock is also known as singularity or "wrist flip" and can, similar to the air and spacecraft design, have fatal consequences for the robot motion. A robot arm can get into singularity if two axis aligns. For example, if the 1st and the 3rd axis aligns, the robot arm will then need to quickly and suddenly make the 2nd axis spin 360° or 180° to maintain its direction. This high velocity motion could in some situations be very dangerous and happen very abruptly.

To avoid damages to surroundings and equipment, a standard for robot and robot software manufactures has been made to prevent this singularity. This is also the case for the ABB industrial robot used to this project. The software will not allow the robot to enter this singularity; rather, it will warn the user when approaching singularity and then eventually stop.

Chapter 4

Implementation

This section describes how the theory, described in the previous section, is implemented in the final design.

4.1 Hardware and Software Architecture

When working with a robot where it is possible to switch the tool and being able to control lights, it is very important that the individual pieces of hardware do not depend upon each other. It should be possible to operate the robot, without switching the light or the camera on, and equally possible to test the lights or the camera without running the robot. This approach will also create the ability to switch the tool on the robot any time, without allow for changing the whole hardware architecture. This way of designing hardware is of course very common in, for example, a laboratory where testing has a high value. This ability is particularly useful when using an expensive device, like a robot, to operate a range of different jobs. This architecture is referred to as Cell Control Architecture[13], where every resource or device contributes to the system.

The design of the software of course needs to match the hardware architecture. The design therefore needs to be adaptable to its changes in the hardware architecture as well as functional and useful. The software is consequently designed

in multiple layers, so that individual functions in the software can be altered, if needed. The software is split into two pieces, where the first piece is the client and hardware control software programmed in C++, used to control all the hardware pieces such as Robot socket, LED control and Camera control. Most of this software was preprogrammed before this project began, and modified to fit the needs of this project. The hardware interface can capture and store images from the camera, turn the LEDs on or off, and finally connect to the robot via a TCP/IP protocol. To make the hardware interface easier to operate, a user interface is attached in form of a terminal like application. This terminal is used to load and execute a command file containing information about how the system should react.

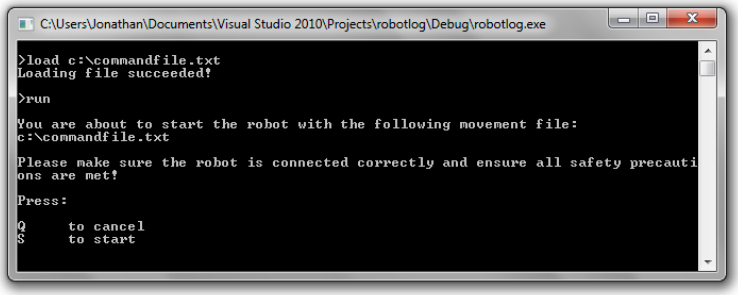


Figure 4.1: Screenshot of the client and Hardware User Interface terminal application.

The other piece of software is the calculation and graphic user interface. This is where all calculations are done along with a graphic representation of how the system reacts. The software is programmed in Matlab and the Matlab GUI Layout Editor which moves Matlab from being a console application to have an actual graphic user interface. Matlab is an excellent tool for making advanced calculations simple, and providing a quick graphic representation. The interface can calculate the motions, coordinates and tool rotations that the robot should perform, and also test the for collision with obstacles. The calculated motions can then be saved to a command file, and later loaded into the hardware control software. The calculation and graphic user interface are programmed so that it has a main window with two tables and a plot. The first table contains the coordinates and rotations of the motion and the second table contains the obstacles. The plot shows a graphic representation of the coordinates, rotations and the obstacles.

4.2 Design process and Implementation

The process of designing the interface was divided into smaller steps to be able to evaluate the process as the program was designed. After considerations about how the final project should turn out, the calculative functions were designed and tested. Along with the design of the geometric functions, the hardware interface were created and tested. The GUI was then designed and the calculative functions were integrated in the GUI. Finally the GUI was tested and evaluated along with the Hardware interface. The design and implementation of the geometric calculative functions are described in this section.

4.2.1 Geometric Functions

Circular Motion The scripting behind the generate circle function, is designed so that it can be used for multiple purposes. This means that can not only generate a circle, but also part of a circle. The function syntax is as follow,

$$\text{genCircle}([\text{C}], [\text{N}], \text{R}, \text{DP}, \text{C}, \text{D}) \tag{4.1}$$

A circle of DP number of data points with the radius R will be created in the plane with the normal vector N. C is the circumference of the circle, and ranges from $0 < C \leq 2\pi$. D is the displacement, and decides the rotation of the circle around the center point. 4.2 shows the plot of a circle with the parameters points = genCircle([0 0 0],[0 0 1],10,20,2*pi,0).

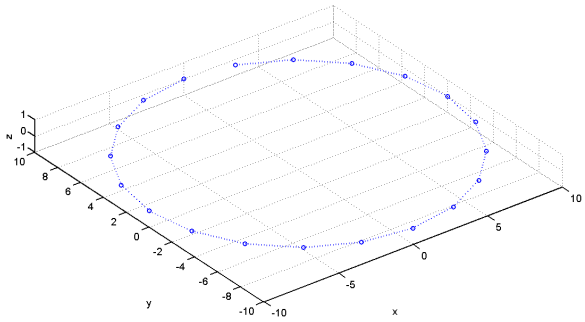


Figure 4.2: Example of a circle plotted.

Spherical Motion How to distribute points evenly on a sphere is described in the theory section, and is implemented in a Matlab script with the following syntax,

$$\text{genSphere}([\text{C}], 0, \text{R}, \text{DP}, \text{SDP}, 0) \tag{4.2}$$

Like the circle, [C] is the center of the sphere, R is the radius and DP are the number of data points. Described in the theory section, the algorithm starts from the bottom, where $\theta = 2\pi$, of the circle and creates a sphere shaped spiral along the z-axis with DP amount of data points. SDP indicates where to start logging the data points, so in order to create a half-sphere, SDP should be set to half of DP/2. The two zeros are not used in the function, but are variables open for implementing the direction of the sphere. The direction of the sphere could be stated by spherical coordinates, where θ would be the tilt of the sphere, and φ could be the rotation along the revolution axis. 4.3 shows the plot of a sphere with the following parameters genSphere([0 0 0],0,10,100,1,0).

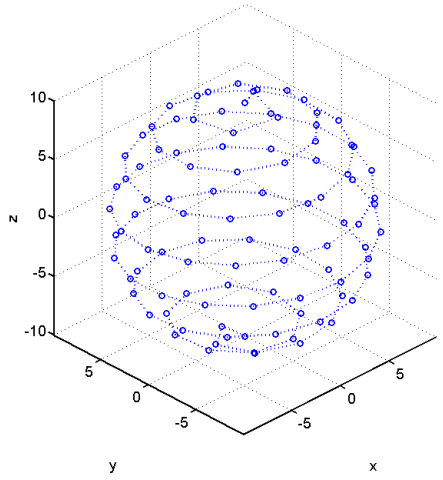


Figure 4.3: Example of a sphere plotted.

Plane Grid Motion The plane grid function is basically distributing a number of points on a plane, bounded the square stretched out by two vectors in the plane. The vectors and plane is show in 4.4.

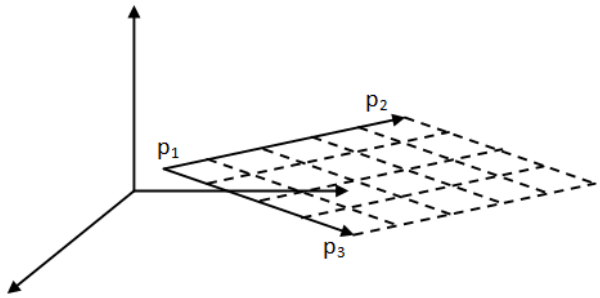


Figure 4.4: The plane grid is created by two vectors that is expanded between three points.

Following syntax is used in the Plane Grid Motion function,

$$\text{genPlaneGrid}([\text{P}_1], [\text{P}_2], [\text{P}_3], \text{DP2}, \text{DP3}) \tag{4.3}$$

where $P1, P2$ and $P3$ are points as shown in 4.4, DP2 and DP3 is the number of data points on respectively $P_1\vec{P}_2$ direction and $P_1\vec{P}_3$ direction. The total number of datapoints is therefore the product of DP2 and DP3. 4.5 shows an example of a plane grid with input genPlaneGrid([0 0 0],[0 10 0],[10 0 0],10,10).

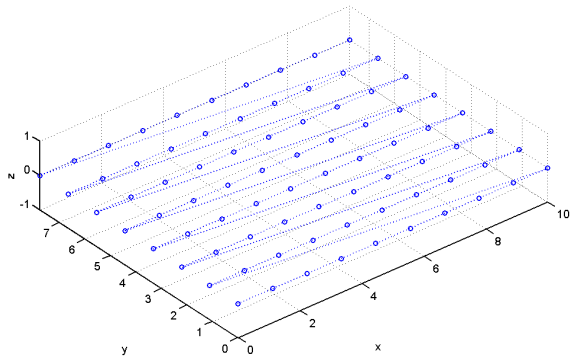


Figure 4.5: Example of a plane plotted.

Tool Rotation and Direction The direction of the tool on the robot arm is described as rotations in spherical coordinates. When a motion-series is generated, such as a circular motion, the best way to calculate the direction of the tool is by calculating the direction of two points. In the circle example, the two points could be the center of the circle and and a point on the circle path.

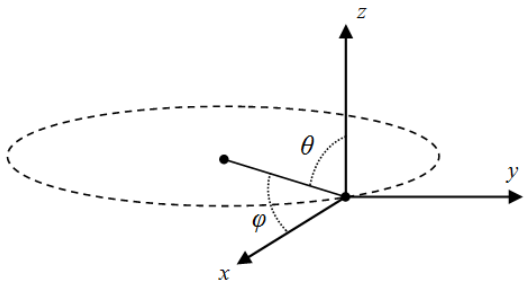


Figure 4.6: The direction of the tool pointing toward the circle center, moving on a circular path.

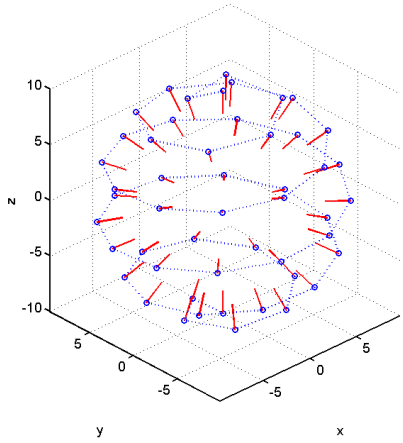


Figure 4.7: The direction of the tool pointing towards center of the sphere.

The direction of the tool is in this case calculated by finding the vector from the tool position on the circle, and the circle center point, and then converting this to spherical coordinates. The function is therefore as simple as,

$$\text{genDirections}([A], [B]) \tag{4.4}$$

where A and B are two points, and outputs the rotations, in degrees, necessary for the tool be placed in A and point toward B. In order to plot the directions, the rotations are converted back to cartesian coordinates. 4.7 shows an example of the tool moving in a sphere always pointing toward the center of the sphere. The little red lines shows the direction of the tool.

Collision map The collision map is designed so that it can detect the collision between a square plane bound by four points, and a path between two points. The idea is that every path between two points in the command file will be checked for collision with squared plans entered in the collision map. The pseudo code is as follow:

Code 4.1: collisionPlaneDetect()

```
INPUT: 4 points (p1,p2,p3,p4) in space representing a square
INPUT: 2 points (p5,p6) representing a path.

Calculate plane, P, from p1, p2 and p3
Calculate line, l, from p5 and p6

if angleBetween(P,l) = 0
  output = 0; %Plane and line are parallel, no collision.
elseif pointInPlane(P,p5) || pointInPlane(P,p6)
  output = 1; %path points are in plane, collision.
else
  intsect = intersectionBetween(P,l);
  d1 = distanceBetween(intsect,p5) + distanceBetween(intsect,p6);
  d2 = distanceBetween(p5,p6);
  if  d1 != d2
    %intersection with plane is not between p5 and p6. no collision
    output = 0;
  else
    XY = pointInSquare(p1[x y],p2[x y],p3[x y],p4[x y],intsect[x y]);
    YZ = pointInSquare(p1[y z],p2[y z],p3[y z],p4[y z],intsect[y z]);
    XZ = pointInSquare(p1[x y],p2[x y],p3[x y],p4[x y],intsect[x y]);
    if XY || YZ || ZX
      output = 1; %intersection w. plane is inside square.
    else
      output = 0; %intersection w. plane is outside square.
    end
  end
end
```

The function pointInSquare() is used to determine if a point, the intersection point, is within the borders of a square in a 2 dimensional system. The function uses the method described in the theory, where the area of the square is calculated, and the area of the four triangles are formed with the square corners and the intersection point.

4.2.2 Graphical User Interface

The graphic user interface is, as earlier mentioned, designed in Matlab GUI Layout Editor, and consists of a main window that shows a plot with the overall layout of the motion map and the collision map. From here it is possible to add new motions or obstacles, these will open in new windows. 4.8 shows the main window of the GUI.

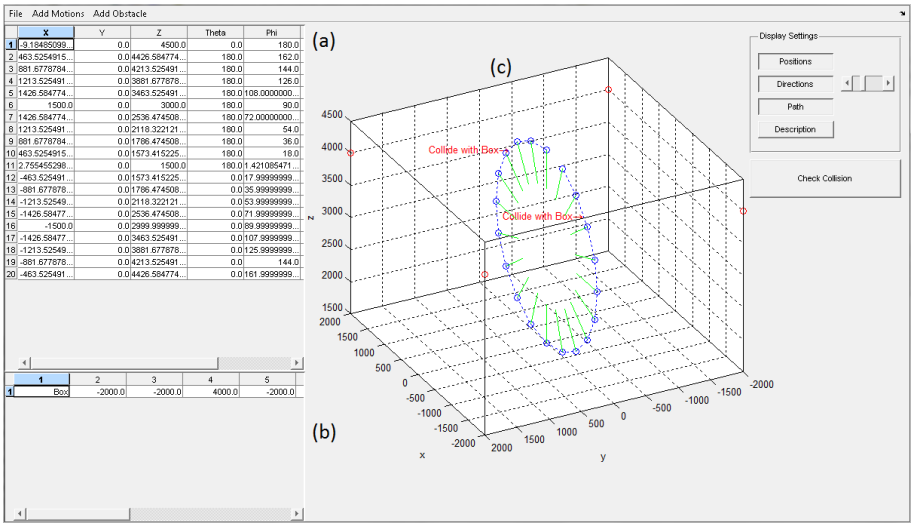


Figure 4.8: Screenshot of the Matlab program main window. A circle is added to the motion map (a) and a box is added to the collision map (b). The circle and the box is display along with where the circle path collides with the box (c).

The GUI is designed with base in the calculative functions, where the input boxes on the GUI for the most part is linked directly to the input parameters of the functions to generate the motion path, and the direction. This means, that the user can directly adjust these parameters and obtain exactly the wanted motion, and get it visualised simultaneously. An example of generating a sphere is showed in 4.9.

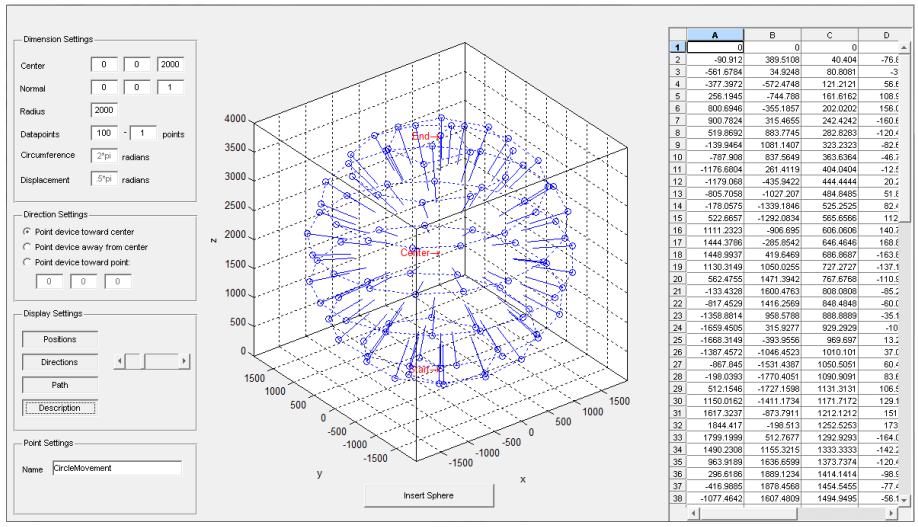


Figure 4.9: Screenshot of the Matlab 'insert sphere' window. Where the settings for the sphere dimensions and directions are on the left side, and the coordinates are listed on the right side. The sphere is inserted by clicking the 'Insert Sphere' button.

Chapter 5

Validation and Evaluation

The previous chapter, describes how all the functions were implemented, and how the interface was created. In this chapter, the validation of the project will be evaluated on the basis of tests and results.

5.1 Test and results

To test the functions, a few experiments have been made to check if the outcome is as expected. The collision detection, geometric functions, graphical user interface and hardware interface is tested. This is basically running through the whole procedure necessary to run the robot.

Collision Detection The collision detection and the geometric functions are tested in A.3. A set of different coordinates are send in to the function collisionPlaneDetect(), and the expected output is listed. The function collisionPlaneDetect() is tested for sending a path through the square plane, a path that lays below the square plane and does not intersect, a path touching the square plane and a path outside the square plane. The results show that the expected result agrees with the output of the function.

Geometric Functions and Directions In order to validate the geometric functions, they also must be tested. The only way to validate the outcome of the geometric functions, is to plot some examples of the functions. The plots are available in A.3, and the outcome shown is as expected.

Graphical User Interface Now the implementation of the geometry and the collision detection can be tested, to see how well they function together. Three circles are added to the motion map, and plane crossing the 3 circles diagonally is inserted as an obstacle into the collision map, see 5.1.

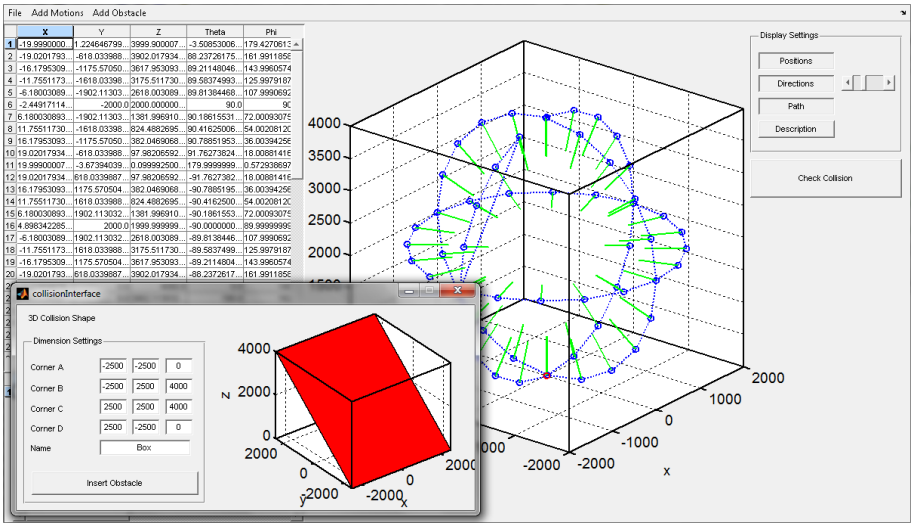


Figure 5.1: A motion map consisting 3 circles has been created. Now a squared plane is added to the collision map.

Now the collision detection algorithm is executed, and should find a total of 6 collisions with the inserted obstacle (4 unique collisions and 2 overlapping collisions). The interface passes the test, and the result is showed in 5.2. It is worth to mention, that the speed of calculating the collisions is quite slow, and takes around 5 seconds for a dataset of 60 Motion points and 1 obstacle. Now the motion map can be saved to a Command File, that will contain the information about the coordinates and directions. The Command File is listed in A.3.

Graphical User Interface The Command File generated before, can be opened in the hardware interface and then send to the robot which will perform the motions. 5.3 shows the Command File loaded in the terminal.

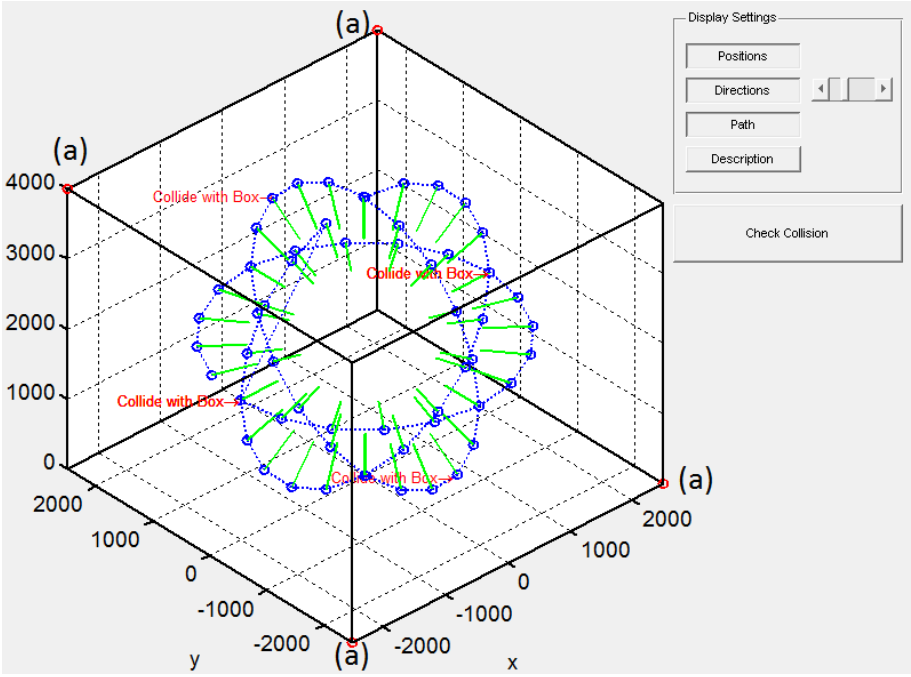


Figure 5.2: The collision detection is execute and the algorithm finds 6 collisions with the motion map, where 4 is unique collisions and 2 are overlapping collisions.

```
+-------------------------------------+
| Welcome to Robo-Command-List Reader |
| Build v1.0 - 28/10-2010             |
+-------------------------------------+

Available commands: load, run, show, home/clear, exit/quit.
Type 'help' for help.
Type 'help command' for detailed help about a specific command.

>load c:/rlog.txt
Loading file succeeded!

>show

 :: DESCRIPTION ::
-----------------------
c:/rlog.txt
This is a Robot Command File.
Moving in circle 20 points.
-----------------------

 :: MOTION COMMANDS ::
----------------------
 -- START --
[(P001/20)(Circle)]  (x,y,z)=(-1414,-1000,3000)  (Rx,Ry,Rz)=(-1447,-600,0)
[(P002/20)(Circle)]  (x,y,z)=(-908,-1260,3260)  (Rx,Ry,Rz)=(-1258,-509,0)
[(P003/20)(Circle)]  (x,y,z)=(-313,-1397,3397)  (Rx,Ry,Rz)=(-1026,-457,0)
[(P004/20)(Circle)]  (x,y,z)=(313,-1397,3397)  (Rx,Ry,Rz)=(-774,-457,0)
[(P005/20)(Circle)]  (x,y,z)=(908,-1260,3260)  (Rx,Ry,Rz)=(-542,-509,0)
[(P006/20)(Circle)]  (x,y,z)=(1414,-1000,3000)  (Rx,Ry,Rz)=(-353,-600,0)
[(P007/20)(Circle)]  (x,y,z)=(1782,-642,2642)  (Rx,Ry,Rz)=(-198,-713,0)
[(P008/20)(Circle)]  (x,y,z)=(1975,-221,2221)  (Rx,Ry,Rz)=(-64,-836,0)
[(P009/20)(Circle)]  (x,y,z)=(1975,221,1779)  (Rx,Ry,Rz)=(64,-964,0)
[(P010/20)(Circle)]  (x,y,z)=(1782,642,1358)  (Rx,Ry,Rz)=(198,-1087,0)
[(P011/20)(Circle)]  (x,y,z)=(1414,1000,1000)  (Rx,Ry,Rz)=(353,-1200,0)
[(P012/20)(Circle)]  (x,y,z)=(908,1260,740)  (Rx,Ry,Rz)=(542,-1291,0)
[(P013/20)(Circle)]  (x,y,z)=(313,1397,603)  (Rx,Ry,Rz)=(774,-1343,0)
[(P014/20)(Circle)]  (x,y,z)=(-313,1397,603)  (Rx,Ry,Rz)=(1026,-1343,0)
[(P015/20)(Circle)]  (x,y,z)=(-908,1260,740)  (Rx,Ry,Rz)=(1258,-1291,0)
[(P016/20)(Circle)]  (x,y,z)=(-1414,1000,1000)  (Rx,Ry,Rz)=(1447,-1200,0)
[(P017/20)(Circle)]  (x,y,z)=(-1782,642,1358)  (Rx,Ry,Rz)=(1602,-1087,0)
[(P018/20)(Circle)]  (x,y,z)=(-1975,221,1779)  (Rx,Ry,Rz)=(1736,-964,0)
[(P019/20)(Circle)]  (x,y,z)=(-1975,-221,2221)  (Rx,Ry,Rz)=(-1736,-836,0)
[(P020/20)(Circle)]  (x,y,z)=(-1782,-642,2642)  (Rx,Ry,Rz)=(-1602,-713,0)
 -- END --
--------------------

>
```

Figure 5.3: A Command File is opened in the hardware interface, and is not ready to send to the robot.

A set of coordinates and directions has now been generated, checked for collisions and loaded into the hardware interface. The motions are now ready to be sent to the robot, which is done by typing "run" in the terminal window. The robot right now has two setting for light control, either specified lights or iteration of all lights. The images captured by the camera will be saved with the file name stated in the Command File.

5.2 Validation

The test results show that the methods created works properly, but calculating time is not impressive for the collision map. The algorithms used to test for collision detection definitely needs to be improved, and maybe expanded to a certain extend. Currently the collision detection limits the program because it can only detect collision between a line, stretched between two points, and a squared object. Extending the collision detection to enable polygons would be a valuable improvement, and could be done by using a Point in Polygon (PIP) algorithm. The general geometric functions works as they should, and no actual improvements are necessary. Although, the algorithm used to generate the sphere could be modified, so that its possible to create parts of a sphere, like a quarter sphere. Further work could be done on the sphere function, so that its possible to rotate it. The connection between the two interfaces works well, but implementing both interfaces in on application would of course be advantageous but also difficult. The layout of the interfaces is made fairly simple, and should be accessible for users with a technical background. The positive results from the tests shows that the method used to create a user interface for the robot is valid.

Chapter 6

Discussion

This chapter will discuss the overall process along with some future work.

A user interface for the industrial robot has been designed, tested and proven to be valid. The problem, as stated in the problem statement, has been narrowed down to a very specific solution. To solve the problem, the following has been implemented:

The client and hardware interface The hardware interface was designed upon an already existing code, which of helped to obtain faster results, but also limited the program to keep the existing structure. This did not turn out to be poorly, since the existing code worked well with the architecture of the interface, even though some modifications needed to be done. There are some minor bugs in the hardware interface, which includes the singularity problem, a time out error and a circuit break error. Singularity, as earlier mentioned, returns an error message, which the software is not programmed to handle. The time out error looses the connection to the robot if it has been in standby for too long, and the circuit break error occurs, when the user breaks the safety circuit, either with one of the stop buttons, or by crossing the infra-red light grid. All these will cause an error in the program that will interrupt the process and may cause the program to terminate. These are not crucial errors, since the program in worst case, needs to be restarted. Besides that, the program is fully usable.

The calculative and graphic interface The calculative interface was designed from scratch using the Matlab Layout Editor and ordinary Matlab scripts. All scripts works individually, but is combined so that they work together in larger functions. Because of this structure, it is possible to test every single one of them, and use every single one for multiple purposes. After having a small library of calculative functions, it was fairly easy to implement the GUI upon these functions. The GUI has been tested and works well, and is also able to communicate with the hardware interface using a Command file. There are no definite problems with the calculative interface, but the performance of some of the algorithms could certainly stand improvements. The interface is designed to be expandable, and it is therefore easy to implement future functions in the interface.

It is of course always possible to improve the existing interface. Some of the major things that could be changed or improved include implementation of the third Euler angle γ. γ is, as mentioned in the theory chapter, neglected because it only describes the rotation of the image, see 3.10. This is not really necessary, since the first two Euler angles α and β is enough to describe the direction of the camera. But implementing the the third coordinate γ, could in some cases simplify the rotations for α and β. This would make the interface even more flexible for mounting other tools, where rotation of the γ coordinate matters. Another feature to implement, is being able to record a set of coordinates using the jog feature on the robot, and create a path from these points. This would require recording the points from the robot and sending them back to the calculative interface. The collision detection does not work for all shapes, and the ability to enter any polygon in space, and detect collision with a path, would be very useful, furthermore, being able to enter the dimensions of the mounted tool, and detect if the collides even though it is only a matter of a few millimetres. Another function that would be helpful to implement is enabling distribution of points on a polygon. Distributing points on a polygon would be a powerful tool for imaging, where odd shaped surfaces is scanned.

Despite all the changes that could be made, the hardware and client interface along with the calculative and graphic interface operates well as a tool for controlling the robot, which is why they fulfil the requirements of the problem analysis and solves the problem.

Chapter 7

Conclusion

A user interface has been created for the robot, a calculative and graphical user interface programmed in Matlab, and a hardware and client interface programmed in C++. Using Matlab to calculate the geometry and collision detection offers a good solution and operates well and also holds the potential of implementing additional functions. Using a C++ programmed terminal application to control the client and hardware is a good solution, and is made simple, so that it is also relatively easy to extend with extra features. The program is made so that it is possible to do further work, and expand the use of the program. Implementing the interface has made the robot more accessible and easier to control.

Appendix A

Test Functions

A.1 Collision Detection

Code A.1: testCollision()

```
clear all
clc

p1 = [0 0 0];
p2 = [0 10 1];
p3 = [10 10 1];
p4 = [10 0 0];

%Path through plane
p5 = [5 5 0]; p6 = [5 5 1];
collisionPlaneDetect( p1,p2,p3,p4,p5,p6 )
%exepected collision

%Path below plane
p5 = [5 5 0]; p6 = [5 5 0.1];
collisionPlaneDetect( p1,p2,p3,p4,p5,p6 )
%not expected collision

%Path ends on corner of plane
p5 = [10 10 1]; p6 = [5 5 0.1];
collisionPlaneDetect( p1,p2,p3,p4,p5,p6 )
%expected collision
```

```
23
24  %Path outside plane area
25  p5 = [100 100 1]; p6 = [10 10 00];
26  collisionPlaneDetect( p1,p2,p3,p4,p5,p6 )
27  %not expected collision
```

Result:
ans =
1.0000e+000 5.0000e+000 5.0000e+000 5.0000e-001
ans =
0 5.0000e+000 5.0000e+000 5.0000e-001
ans =
1 10 10 1
ans =
0 -1.2500e+000 -1.2500e+000 -1.2500e-001

A.2 Geometric Functions and Directions

Code A.2: testCollision()

```
1   clc
2   clear all
3   %odd shape plane
4   points = genPlaneGrid( [0 0 10],[0 5 15],[15 0 20],15,20 );
5   %%normal grid
6   %points = genPlaneGrid( [0 0 0],[0 10 0],[10 0 0],10,10 );
7   %%normal circle
8   %points = genCircle([0 0 0],[0 0 1],10,20,2*pi,0);
9   %%circum circle
10  %points = genCircle([5 5 5],[1 1 1],10,20,(3/2)*pi,0);
11  %%small sphere
12  %points = genSphere([0 0 0],0,10,10,1,0);
13  %%medium sphere
14  %points = genSphere([0 0 0],0,10,100,1,0);
15  %%large sphere
16  %points = genSphere([0 0 0],0,10,1000,1,0);
17  %%half sphere
18  %points = genSphere([0 0 0],0,10,100,50,0);
19  %%Circle w. directions
20  % points = genCircle([5 5 5],[0 0 1],10,20,2*pi,0);
21  % for n=1:20
22  %         directions(n,:) = genDirections(points(n,:),[5 5 5]);
23  %         tempDir = sph2car(deg2rad(directions(n,1)),...
24  %deg2rad(directions(n,2)),2);
25  %         dirX(n) = tempDir(1);
26  %         dirY(n) = tempDir(2);
27  %         dirZ(n) = tempDir(3);
28  %         plot3([points(n,1) points(n,1)+dirX(n)],[points(n,2)...
```

```
% points(n,2)+dirY(n)],[points(n,3) points(n,3)+dirZ(n)],'r-')
%         hold on
% end
%%Sphere w. directions
% points = genSphere([0 0 0],0,10,50,1,0);
% for n=1:50
%         directions(n,:) = genDirections(points(n,:),[0 0 0]);
%         tempDir =...
% sph2car(deg2rad(directions(n,1)),deg2rad(directions(n,2)),2);
%         dirX(n) = tempDir(1);
%         dirY(n) = tempDir(2);
%         dirZ(n) = tempDir(3);
%         plot3([points(n,1) points(n,1)+dirX(n)],...
%[points(n,2) points(n,2)+dirY(n)],...
%[points(n,3) points(n,3)+dirZ(n)],'r-')
%         hold on
% end

plot3(points(:,1),points(:,2),points(:,3),'o:')
xlabel('x')
ylabel('y')
zlabel('z')
grid on;
axis equal;
box off;
rotate3d on

 format compact, format short e
 set(0,'defaultaxesfontsize',14,'defaultaxeslinewidth',2,...
     'defaultlinelinewidth',2,'defaultpatchlinewidth',2)
```

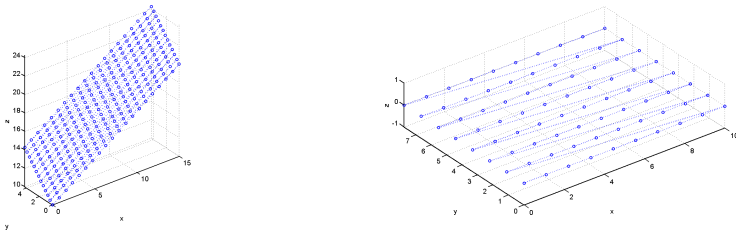


Figure A.1: odd shape plane and normal grid

A.3 Graphical User Interface

Below is a Command File generated, containing the data of a circle with the following dimensions:
center = [0,0,2000];

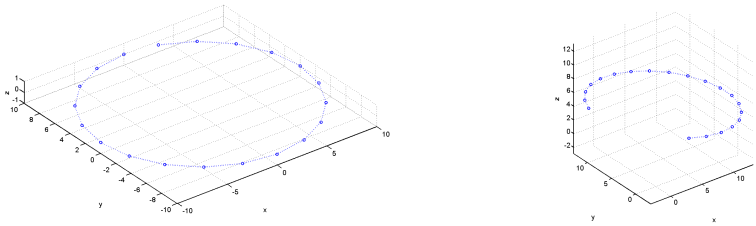


Figure A.2: normal circle and circum circle

Figure A.3: small sphere and medium sphere

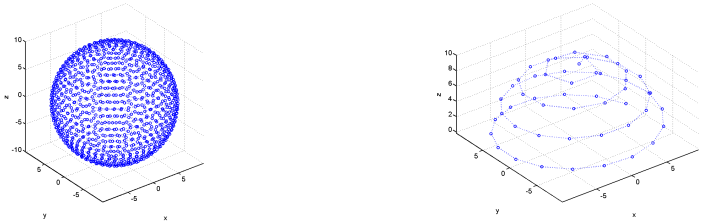


Figure A.4: large sphere and half sphere

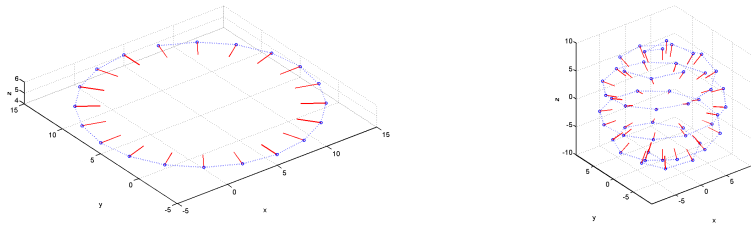


Figure A.5: Circle w. directions and Sphere w. directions

normal = [0,1,1];
datapoints = 20;
radius = 2000;
circum = 2*pi;
displace = (1/2)*pi;

This is a Robot Command File.
Moving in circle 20 points.

START
(P001/20)(Circle) -1414 -1000 3000 -1447 -600 0
(P002/20)(Circle) -908 -1260 3260 -1258 -509 0
(P003/20)(Circle) -313 -1397 3397 -1026 -457 0
(P004/20)(Circle) 313 -1397 3397 -774 -457 0
(P005/20)(Circle) 908 -1260 3260 -542 -509 0
(P006/20)(Circle) 1414 -1000 3000 -353 -600 0
(P007/20)(Circle) 1782 -642 2642 -198 -713 0
(P008/20)(Circle) 1975 -221 2221 -64 -836 0
(P009/20)(Circle) 1975 221 1779 64 -964 0
(P010/20)(Circle) 1782 642 1358 198 -1087 0
(P011/20)(Circle) 1414 1000 1000 353 -1200 0
(P012/20)(Circle) 908 1260 740 542 -1291 0
(P013/20)(Circle) 313 1397 603 774 -1343 0
(P014/20)(Circle) -313 1397 603 1026 -1343 0
(P015/20)(Circle) -908 1260 740 1258 -1291 0
(P016/20)(Circle) -1414 1000 1000 1447 -1200 0
(P017/20)(Circle) -1782 642 1358 1602 -1087 0
(P018/20)(Circle) -1975 221 1779 1736 -964 0
(P019/20)(Circle) -1975 -221 2221 -1736 -836 0
(P020/20)(Circle) -1782 -642 2642 -1602 -713 0
END

Bibliography

[1] Henrik Aanæs, Anders Lindbjerg Dahl, and Kim Steenstrup Pedersen. On recall rate of interest point detectors.

[2] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation. *Technical Report DIKU-TR-98/5*.

[3] Matlab Documentation. http://www.mathworks.com/help/techdoc/ref/cart2sph.html.

[4] Per W. Karlsson and Vagn Lundsgaard Hansen. *Matematisk Analyse 2*. Institut for Matematik, Danmarks Tekniske Universitet, 1998.

[5] Jonathan Kofman. Intelligent human-machine systems / optomechatronic systems laboratories. *http://www.eng.uwaterloo.ca/(TILDE)jkofman/*.

[6] Jack B. Kuipers. Quaternions and rotation sequences. *Geometry, Integrability and Quantization*.

[7] Rasmus Larsen. Center for imaging food quality.

[8] Move and Fit Points. http://local.wasp.uwa.edu.au/(tilde)pbourke/geometry/spherepoints/.

[9] ABB AB Robotics Products. *Technical Reference Manual - RAPID Kernel*.

[10] ABB AB Robotics Products. *Technical Reference Manual - RAPID Overview*.

[11] Edward B. Saff and Arno B. J. Kuijlaars. Distributing many points on a sphere. *The Mathematical Intelligencer*, 19(1):5–11, 1997.

[12] Spherical. http://sitemason.vanderbilt.edu/page/hmbads#code.

[13] Phil Webb and Craig Johnson. Measurement assisted robotic assembly of fabricated aero-engine components. *Assembly Automation*.