ISSN 0280-5316
ISRN LUTFD2/TFRT--5888--SE

Intuitive Human Robot Interaction and Workspace Surveillance by means of the Kinect Sensor

Klaudius Werber

Department of Automatic Control
Lund University
August 2011

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* STUDENT THESIS REPORT *Date of issue* August 2011 *Document Number* ISRN LUTFD2/TFRT--5888--SE
Author(s) Klaudius Werber	*Supervisor* Anders Robertsson Automatic Control Lund, Sweden Rolf Johansson Automatic Control Lund, Sweden (Examiner) *Sponsoring organization*

Title and subtitle

Intuitive Human Robot Interaction and Workspace Surveillance by means of the Kinect Sensor (Intuitiv robotprogrammering och arbetsområdesövervakning med Kinect-sensor)

Abstract

This student thesis is part of the ROSETTA project that examines the possibilities of future manufacturing based on close human-robot interaction.
In this report, the capabilities of a robot system that is controlled by the Kinect sensor of the Microsoft XBOX-360 are tested. A setup including two Kinect devices and one robot is presented. Applications for calibration of two Kinect sensors, robot workspace surveillance, face detection and shadowing the user’s arm are introduced and evaluated concerning their effectiveness. Finally, the compatibility of the introduced setup with the objectives of the ROSETTA project is evaluated. It proves that the Kinect can strongly support developing solutions for these objectives. The thesis work was conducted at the Department of Automatic Control at LTH at Lund University, Sweden, in collaboration with the Institute of Integrated Sensor Systems at University of Kaiserslautern, Germany.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 121	*Recipient’s notes*
Security classification		

http://www.control.lth.se/publications/

Contents

1. Introduction

1.1. Motivation

Analyses show that future factories will need more flexible manufacturing systems to meet the expectations in an increasing product variety. Producing goods with high volumes but limited product lifetime will require mass production with frequent production changes. To achieve these demands, in the future humans and robots are supposed to work closer together, to benefit from both the ability of humans to perform tasks that would be difficult to automate, and the efficiency of robots in quickly performing high precision tasks with low automation threshold.[1]

For that reason the European Union funded the ROSETTA project (RObot control for Skilled ExecuTion of Tasks in natural interaction with humans based on Autonomy, cumulative knowledge and learning). Among others the objectives of this project are to make human-like Robots more flexible to program, more intuitive to control and more safe to work with.

These objectives could be achieved by the means of the Kinect Sensor of the Microsoft XBOX-360 (cf. [1]). Evaluating to which extend this sensor might be useful to accomplish the mentioned objectives and giving an example application in which a robot can be controlled using the Kinect Sensor was the task of this student thesis and a small contribution to the ROSETTA project.

1.2. Problem Formulation

The problem description of this thesis was:

> "The student project work aims at integrating the 'Kinect sensor' as a plug-and-play sensor within the open robot control environment (ORCA/Orcinus) and with the Linux based software development platform of the Robotics lab, LTH, Lund University. As a proof of concept a

[1] cf. the ROSETTA project page: http://www.fp7rosetta.org/?q=node/2

small study of demonstrating surveillance of the workspace and 'visual programming/interaction' with implementation for either a stationary or for a mobile robot platform will be made."

My goal for this thesis was to build an example robot application, controlled by the means of the Kinect sensor. This application should, accordingly to the ROSETTA project, be intuitive and feel natural for the user, it should be safe and flexibly set up.

1.3. Setup

This section is supposed to give a rough overview of the application composition. A more detailed description of the used components and methods will be presented in the following chapters.

As displayed in Figure 1.1, two Kinect sensors observe the workspace and send 3D image information to the main computer. This computer performs the sensor fusion of the two Kinect devices, processes the data according to a control program that will be explained later in this report and then sets the parameters and reference values of the Simulink controller via a LabComm connection between the main computer and the Simulink controller. The Simulink controller intercepts the communication between the main robot computer and the axis controller computer and introduces the control values of the main controller. This method to introduce external control to an industrial robot application is called "ExtCtrl"(cf. [2]) and was developed at the Department for Automatic Control at LTH at Lund University. Together, the LabComm connection and the ExtCtrl system form the ORCA/Orcinus open robot control environment (cf. [10]).

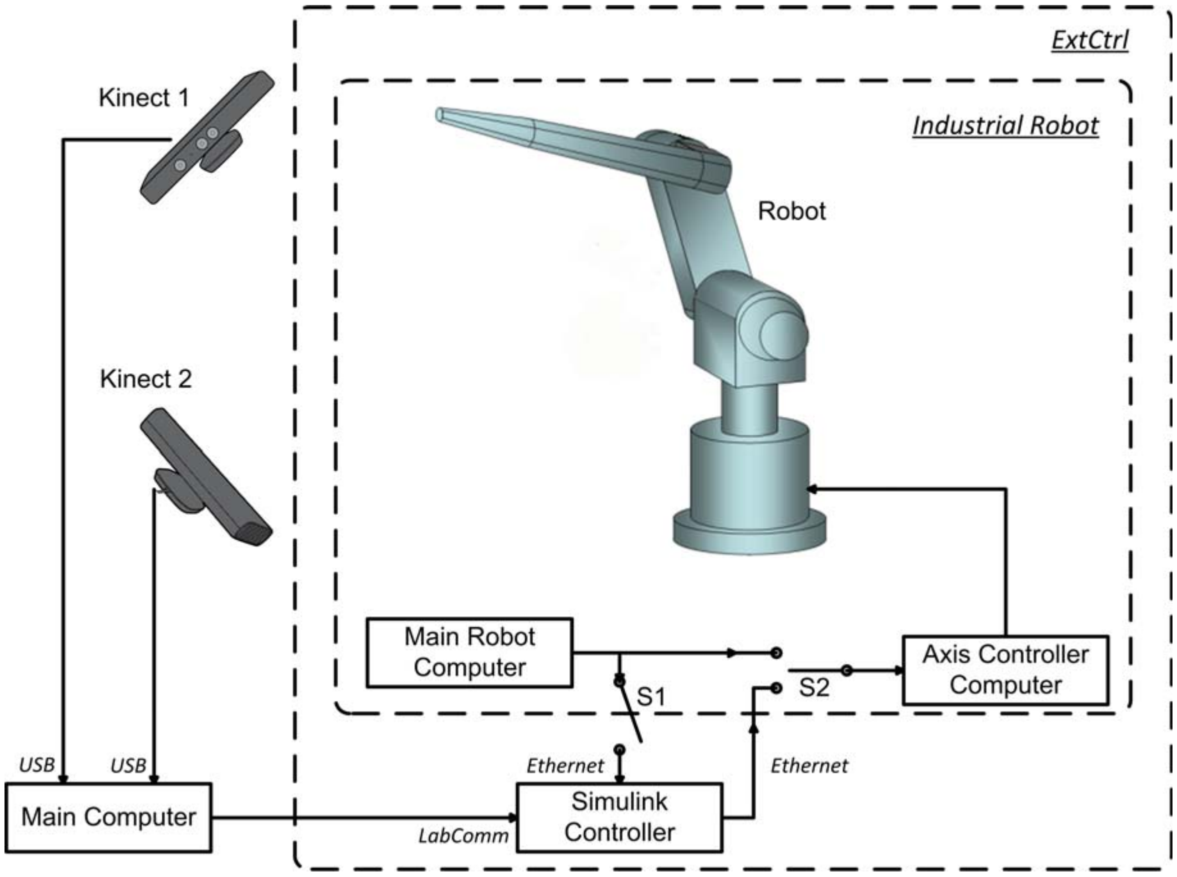


Figure 1.1.: Overall System
The arrows indicate the prevailing direction of the data flow. Data flow in the opposite direction is possible.

2. Components

In this chapter all components of the control application using the Kinect Sensor are listed and closely described.

2.1. Kinect Sensor

The Kinect Sensor (Figure 2.1) was developed by Microsoft and PrimeSense (cf. [3]), an Israeli company that focuses on digital 3D vision sensors for natural interaction.[1] It is a hardware device used to control the Microsoft XBOX-360 game console without any kind of controller that the user has to hold or wear. The official release of Kinect in North America was on November 4th, 2010, in Europe on November 10th, 2010.

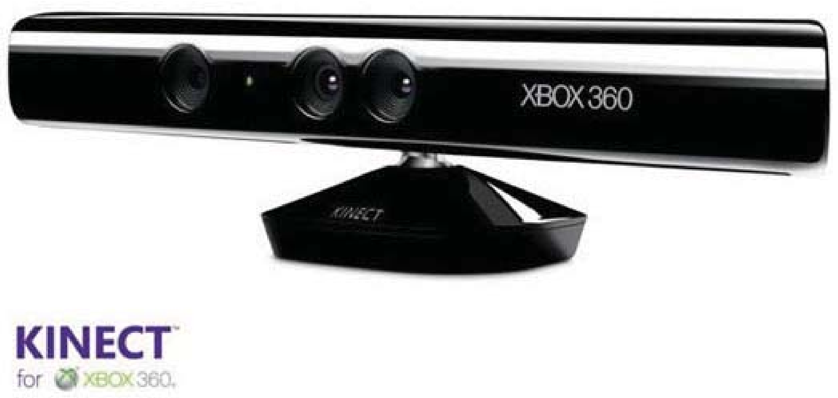


Figure 2.1.: Kinect-Sensor

The Kinect features a RGB camera, an IR depth sensor a multi-array microphone and a motor in the socket for tilting the Kinect. In the application of this thesis the RGB camera was used to calibrate the two Kinect sensors. Apart from that only the depth sensor was employed. These parts are marked in Figure 2.2. Additionally, in Figure 2.2

[1]The term "natural interaction" describes human machine interaction that feels intuitive and natural for the human. It is oriented on the ways humans interact with each other.

the coordinate System used for each Kinect is displayed. As origin the RGB camera was chosen to match depth and image data of one single Kinect. This will be explained more detailed in the section on calibration, Section 3.2.

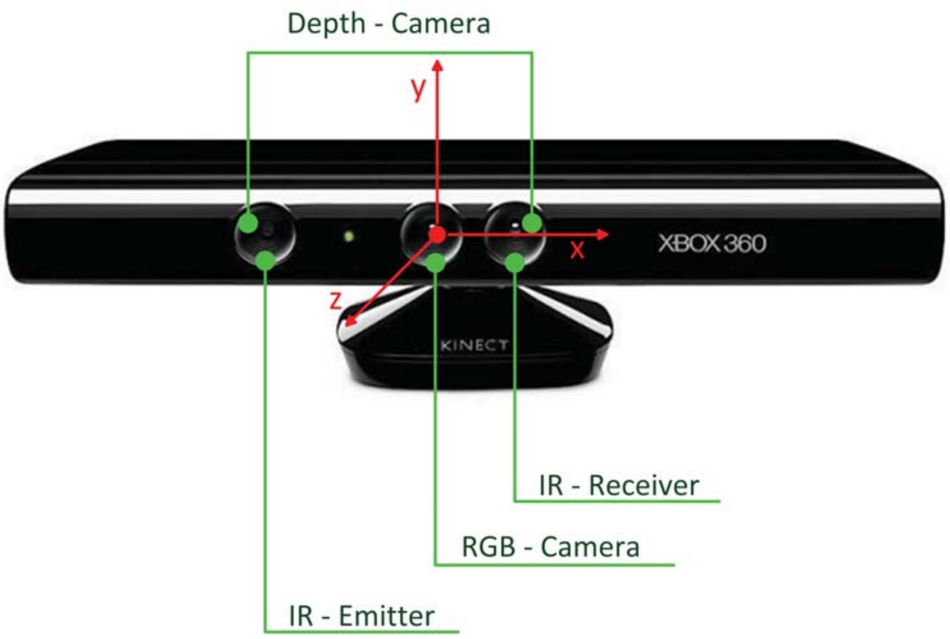


Figure 2.2.: Single parts of the Kinect sensor and the used coordinate system

Several contradictory specifications concerning the Kinect exist. Unfortunately there is no official information on that. The following numbers describing specifications relevant for this thesis are taken from kinectprices.co.uk[2].

- Horizontal field of view: 57 degrees
- Vertical field of view: 43 degrees
- Depth camera: 640x480 pixels at 30 frames/sec
- RGB camera: 640x480 pixels at 30 frames/sec
- Power consumption 2,25 W

Especially the resolution of the depth camera is often said to be only 320x240 pixels instead of 640x480. In Subsection 2.1.2, I will explain, why I decided on the higher resolution.

Kinect uses “Light Coding” to evaluate the depth of the items in view. “Light Coding” is a proprietary variation to “Structured Light” that is explained for example in [4].

[2]http://www.kinectprices.co.uk/kinect-technical-specifications

PrimeSense writes on its FAQ internet side concerning "Light Coding":

> "PrimeSense's technology for acquiring the depth image is based on Light Coding™. Light Coding works by coding the scene volume with near-IR light. The IR Light Coding is invisible to the human eye. The solution then utilizes a standard off-the-shelf CMOS image sensor to read the coded light back from the scene. PrimeSense's SoC chip is connected to the CMOS image sensor, and executes a sophisticated parallel computational algorithm to decipher the received light coding and produce a depth image of the scene. The solution is immune to ambient light, and works in any indoor environment."

This immunity to ambient light of the depth camera can be nicely observed in Figure 2.3. This figure shows the depth and the color image of a person behind a reflective window. While the color image is nearly useless due to the ambient light, the depth image barely seems deteriorated in the window part of the picture in comparison to the non-window parts.

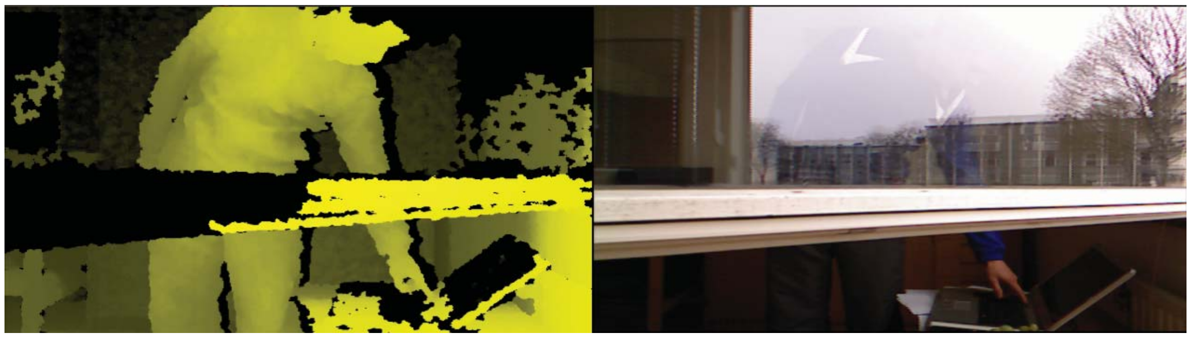


Figure 2.3.: Depth and color image of a person behind a reflective window

2.1.1. Depth Measurement

There is no official information on "Light Coding". So the following considerations are likely assumptions, but not sure facts:

The IR-Light-Emitter (cf. Figure 2.2), a class 1 LASER, either sends out just one light point to one single pixel at a time or, more likely, a pattern of distinct symbols for every pixel at the same time. In both cases, after the light has been dispersed back to the Kinect by an object in view, the IR-Receiver can identify this light to belong to its original pixel of the IR-Emitter, in the first case due to the distinct time point, in the second case due to the distinct symbol.

The following considerations do not need to be carried out in 3D space. For calculating the depth of a point in view, i.e. the value of its z-coordinate, it is sufficient to work with its projection on the xz-plane, that means to neglect its y-coordinate.

Because of the distance between the IR-Emitter and the IR-Receiver ($b = 75\,mm$), for each point in view the angle β, at which the emitter sends out the light, and the angle $(180° - \alpha)$, at which the receiver detects its dispersion differ a little. Just like in stereo vision this angular difference can be used to calculate the depth of the respective point in view. This situation can be seen in Figure 2.4. The viewpoint of Figure 2.4 is located above the displayed plane looking downwards.

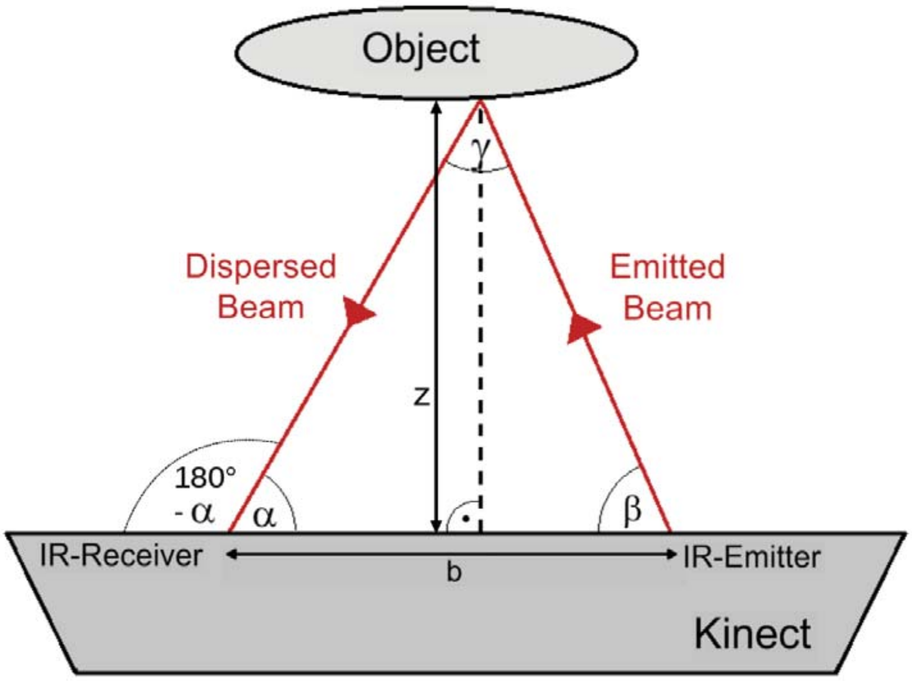


Figure 2.4.: Angular difference of the emitted and the dispersed beam

Summing up the angle α of the dispersed and received beam and the angle β of the emitted beam can be measured. The baseline b is a known constant. The height z of the triangle formed by the two beams and the Kinect baseline is searched for, cf. Figure 2.4.

The angle γ can be calculated by the sum of all angles in a triangle:

$$\gamma = 180° - \alpha - \beta \tag{2.1}$$

The law of sine can be used to calculate the length s of the dispersed beam:

$$s = b \cdot \frac{\sin\beta}{\sin\gamma} \tag{2.2}$$

Using the rectangle of the height, z can be expressed as:

$$z = s \cdot \sin\alpha \tag{2.3}$$

Inserting Equations (2.1) and (2.2) into Equation (2.3) yields:

$$z = b \cdot \sin\alpha \cdot \frac{\sin\beta}{\sin(180^\circ - \alpha - \beta)} = b \cdot \frac{\sin\alpha \cdot \sin\beta}{\sin(\alpha + \beta)} \tag{2.4}$$

This equation returns the depth for every couple (α, β), i.e. for every pixel that the IR-Emitter sends light to and the IR-Receiver detects light from. Then if the depth value (the z-coordinate) is known, for every pixel the x- and the y-coordinate can be calculated as linear functions of the z-coordinate. The cofactors of the depth value depend on the respective angle in vertical and horizontal deflection. These functions arise out of the specifications of the Kinect:

$$x = 2 \cdot \left(\frac{X_{Pix}}{640} - \frac{1}{2}\right) \cdot \tan\left(\frac{57^\circ}{2}\right) \cdot z \tag{2.5}$$

$$y = 2 \cdot \left(\frac{1}{2} - \frac{Y_{Pix}}{480}\right) \cdot \tan\left(\frac{43^\circ}{2}\right) \cdot z \tag{2.6}$$

where (X_{Pix}, Y_{Pix}) are the coordinates of the respective pixel in the depth frame.

2.1.2. Uncertainty in Depth Measurement

The depth value z in Equation (2.4) depends on the angles α, and β. Possible errors of β are due to non ideal positioning of the IR-LASER beam. The main error of α is the quantization error in detecting the dispersed light by the discrete pixel array in the IR-Receiver.

Assuming that the main error of the depth measurement is the quantization error of the detection of the dispersed IR-light and using the law of error propagation, the error of a calculated depth value can be approximated as:

$$\Delta z \approx \frac{\partial z}{\partial \alpha} \cdot \Delta\alpha \tag{2.7}$$

This is only an approximation, because errors due to non ideal positioning of the IR-LASER beam are disregarded.

The derivative of Equation (2.4) with respect to α is:

$$\frac{\partial z}{\partial \alpha} = b \cdot \sin \beta \cdot \left(\frac{\cos \alpha \cdot \sin(\alpha + \beta) - \sin \alpha \cdot \cos(\alpha + \beta)}{\sin^2(\alpha + \beta)} \right) \tag{2.8}$$

The drawback of this equation is that α and β cannot be chosen entirely independently from each other. A lot of combinations (α, β) would lead to impossible (negative) or unlikely (very near or very far) depth values. Thus it makes sense to use Equation (2.4) to substitute β by z in Equation (2.8).

Using the addition theorem

$$\sin(\alpha + \beta) = \sin \alpha \cdot \cos \beta + \sin \beta \cdot \cos \alpha \tag{2.9}$$

the following conversions of Equation (2.4) can be accomplished:

$$\frac{\sin \alpha \cdot \cos \beta + \sin \beta \cdot \cos \alpha}{\sin \alpha} = \frac{b}{z} \cdot \sin \beta \tag{2.10}$$

$$\Leftrightarrow \qquad \cot \alpha + \cot \beta = \frac{b}{z} \tag{2.11}$$

$$\Leftrightarrow \qquad \beta = \operatorname{arccot}\left(\frac{b}{z} - \cot \alpha \right) \tag{2.12}$$

Therefore, by inserting Equation (2.12) into Equation (2.8), $\partial z / \partial \alpha$ can be expressed as a function of z and α.

$$\frac{\partial z}{\partial \alpha} = \frac{\partial z}{\partial \alpha}(z,\, \alpha) \tag{2.13}$$

To derive an expression for the uncertainty $\Delta\alpha$ of the measurement of α, the model of a pinhole camera[3] like in Figure 2.5 is used.

[3]Even though the pixels inside the IR-Receiver probably are arranged in a more sophisticated manner, the dependency between the angle of the incident beam and the density of pixels has to be the same as for a pinhole camera. Otherwise the generated picture would be distorted.

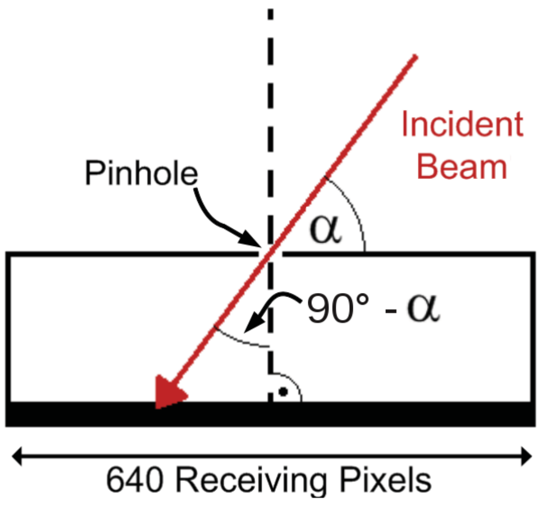


Figure 2.5.: Model of a pinhole camera

Taking into account Figure 2.5 and the technical specifications of the Kinect,

- horizontal field of view: 57°,
- number of pixels in horizontal direction: 640

the following relation is valid. It relates the angular range of the full horizontal field of view of the Kinect with the number of all camera pixels in horizontal direction.

$$2 \cdot \tan\left(\frac{57^\circ}{2}\right) \mathrel{\widehat{=}} 640\,[\text{pixels}] \tag{2.14}$$

Using the relation (2.14), Figure 2.5 and the non-official information that the angle of the incident beam in the IR-Receiver is resolved with the quantisation of 1/8 pixel[4], the uncertainty $\Delta\beta$ of the angle of the received light beam can be derived the following way:

$$\Delta \tan\left(90^\circ - \alpha\right) = \Delta \cot\alpha = \frac{1}{8} \cdot \frac{2 \cdot \tan\left(\frac{57^\circ}{2}\right)}{640} \tag{2.15}$$

Now applying the law of error propagation yields:

$$|\Delta\alpha| = |\Delta \operatorname{arccot}\left(\cot\alpha\right)| = \left|\frac{\partial\left(\operatorname{arccot}\left(\cot\alpha\right)\right)}{\partial\left(\cot\alpha\right)} \cdot \Delta\cot\alpha\right| = \frac{1}{1+\cot^2\alpha} \cdot \frac{\tan\left(\frac{57^\circ}{2}\right)}{4\cdot 640} \tag{2.16}$$

[4]Posted by Dr. Suat Gedikli, research engineer at Willow Garage in the OpenNI Google Group: https://groups.google.com/group/openni-dev/browse_thread/thread/673021887a5eca14

Inserting Equations (2.8), (2.12) and (2.16) in Equation (2.7) leads to a description of the uncertainty Δz of the depth measurement as a function of the depth z itself and the angle of the received light beam α. This function is plotted in Figure 2.6.

$$\Delta z = \Delta z\,(z,\,\alpha) \tag{2.17}$$

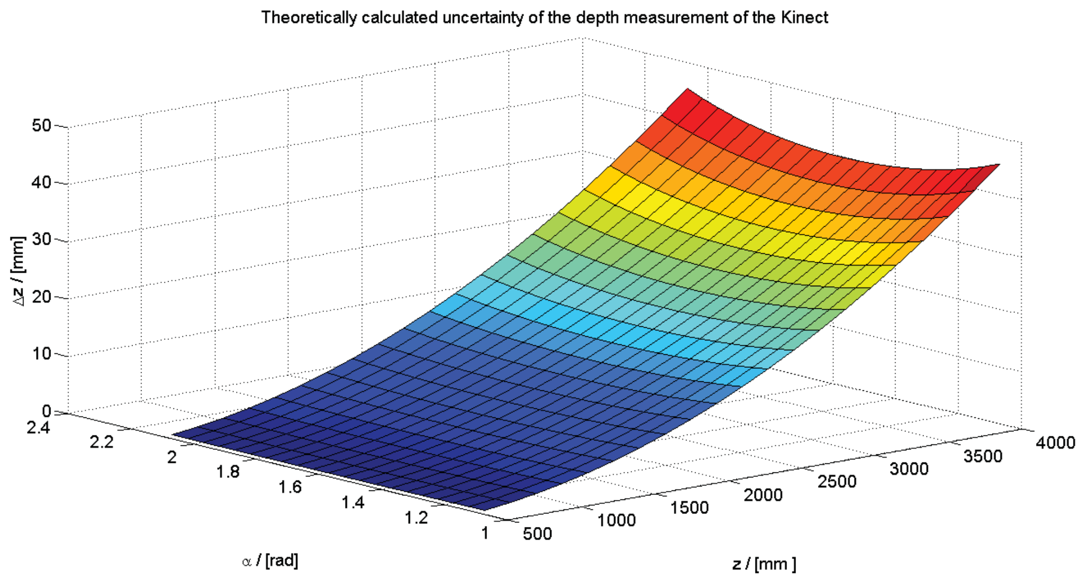


Figure 2.6.: Analytically calculated uncertainty of the depth measurement of the Kinect

To verify the function of Equation (2.17), a small experiment was conducted. The Kinect was positioned facing a plane wall. While recording the x-, y-, and z-coordinates of the point that was seen by a certain pixel(240/180) the Kinect was moved, to change the depth values. This pixel corresponds to the angle $\alpha = 82,09°$. The recorded coordinates are plotted in Figure 2.7. The values of the x- and the y- coordinate are only plotted to show their direct linear dependency on the depth value, the z-coordinate. Solely that one is relevant for the experiment. In total, 4691 depth values were generated that way.

Even though the movement of the Kinect and therefore the change of the depth value was continuous, the measured depth values have to be quantised due to the finite resolution of the Kinect. So whenever two consecutive depth values of Figure 2.7 were not equal, their difference in dependency of their absolute value was recorded. If there were several depth-differences for the same absolute depth, their mean value was chosen. As a result, in Figure 2.8 for every measured depth the mean depth-difference to following non-equal depths is plotted, which is the quantisation and with the approximation of Equation (2.7) the uncertainty of the depth measurement. For comparison in Figure 2.8 the theoretically calculated uncertainty for the same angle ($\alpha = 82,09°$) is displayed, too.

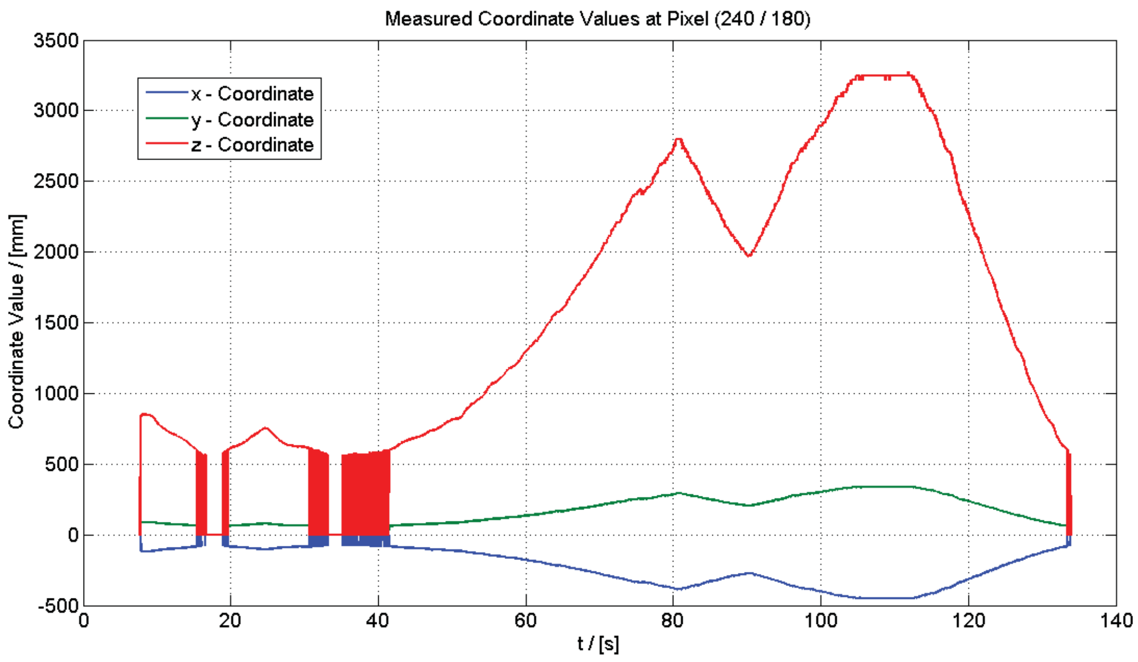


Figure 2.7.: Measured coordinate values at pixel (240 / 180)

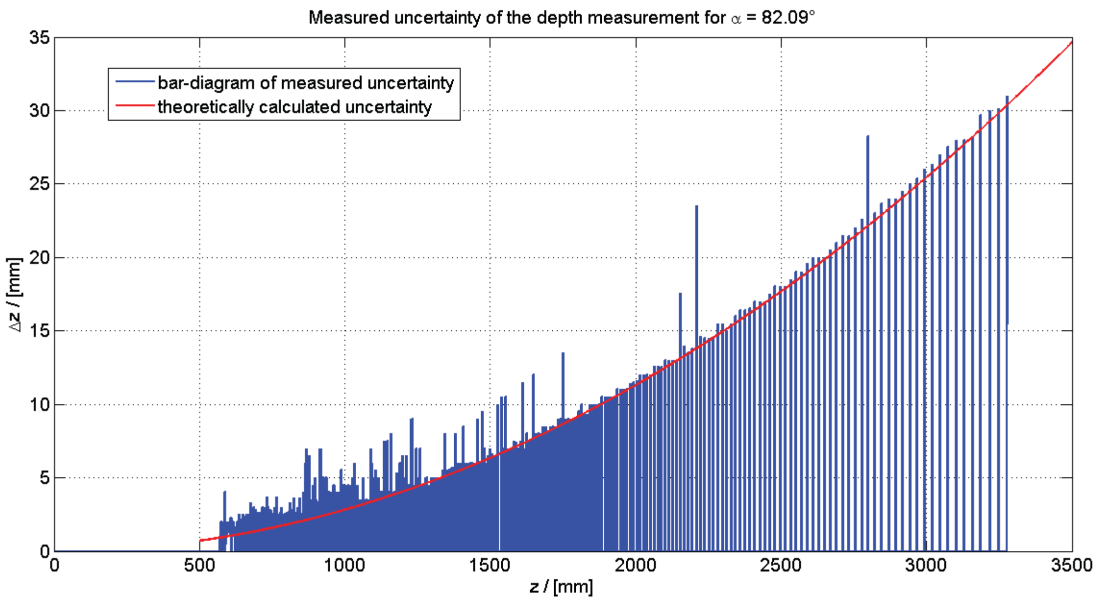


Figure 2.8.: Measured uncertainty of the depth measurement for $\alpha = 82,09°$

From Figure 2.8 can be seen, that both in form and in absolute values the measured and the calculated values for the uncertainty match strongly. This proves that the postulated assumptions were admissible. It also indicates that the non-official information about the technical specifications of the Kinect are correct. The systematically slightly higher values for the measured uncertainty can be explained by the experiment. If the depth value did not change between two consecutive measurements, this zero difference was discarded. But if it may have changed by more than one quantisation step, this difference was regarded. So on average the mean difference can only be greater than or equal to the analytically calculated inaccuracy.

2.1.3. Comparison Kinect - Time of Flight

Already before Kinect was released depth cameras existed based on the time of flight principle (TOF). These TOF-cameras send out an IR-flash that lights up the scene and then measure for every pixel the time it takes the light to return. This time is directly proportional to the depth of each pixel.

$$T = 2 \cdot \frac{D}{c} \tag{2.18}$$

with D the depth of this pixel and c the speed of light. Depending on the maximum time that the light is given for return, different depth ranges can be defined. For short ranges that do not need much time, the frame rate of TOF-cameras can be up to 100 frames/s. However, by allowing a long time till return of the light these cameras can be used for detecting far depths, too. Then the frame rates are respectively lower. TOF-cameras have uncertainties of roughly about 10 mm, independently from the absolute depth within a certain depth range.

Summing up TOF-cameras seem to be the better choice than the Kinect, when high frame rates are needed at short distances, or when comparably high accuracy is needed at points far away from the camera.

The main disadvantage of TOF-cameras is their price. They cost roughly about 5000 Euro, while the Kinect costs about 130 Euro.

2.1.4. Interference between two different Kinect devices

As in this thesis two Kinect units are used in parallel, the possibility of interference between these two Kinect devices is examined.

Given the principle of Light Coding to send out light in a distinct way, so it can be detected again and matched to a certain pixel, the probability that two different Kinect sensors observing the same scene will interfere strongly, even to an extent that makes multiple Kinect use impossible, seems high.

As there is no official information on Light Coding, the only way to determine the extent of interference between two Kinect units is by experiment. In Figure 2.9 three times a flat board is displayed, as seen by one Kinect. In the first case no other Kinect was present, in the second case another Kinect was positioned next to the first Kinect at the exact same angle and in the third case the other Kinect was pointed at the same board, but from another position at an angle 90 degrees different from the angle of the first Kinect. This third case is the arrangement that will be reasonably used in the actual control application, because it offers vision of the scene from two sides.

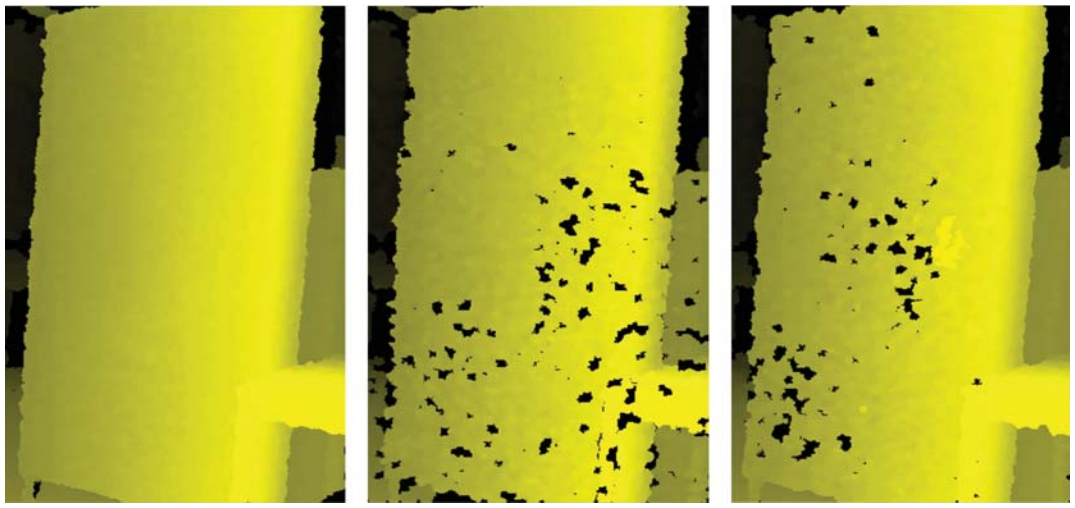


Figure 2.9.: Interference between two Kinect devices
left: no interference at all
middle: full interference, both Kinect sensors at the same angle
right: weakened interference, both Kinect sensors observing the same scene from angles different by $90°$

From Figure 2.9 can be seen that in contrast to first assumption the two Kinect devices do not interfere strongly. They do interfere, but the resultant depth picture is only slightly stronger disturbed than without interference, not distorted as a whole. If the two Kinect devices are not positioned at the same angle this effect is even weaker. Additionally, Figure 2.9 shows that interference between two Kinect sensors does not lead to wrong depth values. At the points of interference the Kinect is not able to calculate a depth value at all. The depths of the other points stay the same as without interference.

So using two Kinect sensors in parallel leads to a little smaller amount of depth data from the single Kinect sensors, but does not tamper with that depth data. Compared with one single Kinect sensor, two Kinect devices together are able to collect a much higher amount of data without losses in accuracy.

2.2. Main Computer

The two Kinect units are connected to the main computer by USB. This computer has to have one USB controller for every plugged in Kinect, because the data stream of two Kinect units would be too big for one single USB controller.

To interface with the Kinect sensors the OpenNI-Framework was used. This is an open source package by PrimeSense. It is intended to make available the new opportunities offered by sensors like Kinect to a larger community, to accelerate new developments in natural interaction.

OpenNI provides a driver for Kinect and an application programming interface (API). It also offers a lot of basic functionality for analysis of the scene watched by Kinect. The functionality that was used in this thesis consists of the following:

Depth generator: The depth generator provides a depth map of the scene as an array of floats, even though the actual depth values are always natural numbers of the unit mm. The position of the points in view can either be described by real-world-coordinates, i.e. the xyz-coordinates of the coordinate system given in Figure 2.2, or by projective coordinates. That means the x- and the y-coordinates are replaced by the pair of numbers addressing the respective pixel in the frame. The z-coordinate stays the same.

Image generator: The image generator provides the RGB-color-image of the scene as an array of tripels of integers between 0 and 255.

Gesture generator: The gesture generator can detect "Click-" or "Wave-" gestures.

Hand generator: The hand generator can track a hand in vision if its initial position is known. To clarify, it cannot detect any hand in vision and track that, but it can decide for a given point whether or not that may belong to a hand and, in case that point does belong to a hand, track its movement.

User generator: The user generator segments the scene in different users and background and labels every pixel with a distinct UserID, indicating to which part of the picture the respective pixel belongs to. The background has the ID "0", every user a different natural number. After a user has been detected, he can do the "Psi-Pose" as in Figure 2.10. Then the user generator calibrates the users body proportions and can afterwards track the position and orientation of some body joints. The joints relevant for this thesis are given here:

- Head (only position)
- Right and left shoulder
- Right and left elbow
- Right and left wrist (only position)
- Right and left hand (only position)

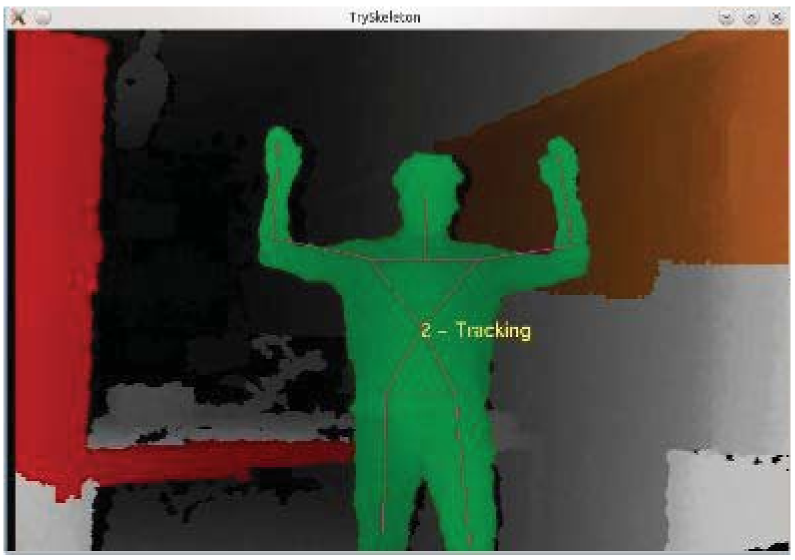


Figure 2.10.: Psi-Pose to calibrate the user generator

Further information on OpenNI can be found in its documantation[5] and in [5].

During the course of this thesis Microsoft also released a software development kit (SDK) for the Kinect[6]. This SDK and the corresponding driver, running exclusively under Windows 7, offers full access to all depth, image, user skeleton and 3D audio data, of the Kinect. Additionally, it allows to move the motors changing the tilt angle of the Kinect. So it offers more access to the basic Kinect functionality then OpenNI.

[5]Included in the OpenNI package, downloadable from https://github.com/OpenNI/OpenNI/tree/unstable
[6]cf. http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/default.aspx

But unlike OpenNI it does not define any hand gestures that can be used as events and it does not calculate the skeleton joint orientations, only the positions. So summing up, the Microsoft Kinect SDK offers more access to the Kinect, but less higher level functionality. Maybe Microsoft expects the private Kinect programming community to find and make public own solutions for these features in the future. For now OpenNI still seems to be the more convenient choice for the application of this thesis.

The main computer has to send and receive control data to and from the Simulink controller. This is done by a LabComm connection, cf. Section 2.4.

2.3. Industrial Robot

In this thesis work three different types of robots were used. The ABB IRB120 (cf. Figure 2.11 and [6]), the ABB IRB140B[7] (cf. Figure 2.11 and [7]) and the ABB FRIDA[8] (cf. Figure 2.12). The IRB120 and IRB140B both are robot arms with six revolute joints. Therefore both of them have six degrees of freedom (DOF). The maximum range of influence of the IRB120 is 580 mm, of the IRB140B 810 mm.

The FRIDA is a new robot having two arms, each with seven revolute joints. So every arm is a seven DOF manipulator itself. It was developed by ABB in close collaboration with the ROSETTA project. It is a light, human like robot that can be put easily in various assembly lines and is considered to be safe for humans to work with in near vicinity.

Figure 2.13 shows the used robot frames: the base frame that is constant and the flange frame that moves with the robot flange. These frames are chosen accordingly to the Denavit-Hartenberg convention, which is the most common convention to describe robot links, joints and coordinate frames. It is explained for example in [8] in Chapter 3. Given a robot with n joints and therefore $n + 1$ links including the base, the links are numbered between 0 and n starting from the base upwards and the joints are numbered between 1 and n starting with the joint attached to the base upwards. In this context upwards means in the direction from the base to the robot flange.

Roughly speaking, this convention then allocates the z-axis of the coordinate frame of each robot link i along the joint axis of the joint $i + 1$ attached to this link in upwards direction. This is displayed in Figure 2.14. The x-axis of each frame i is set according to the common normal of the z-axes of frame i and frame $i - 1$ respective. The y-axis

[7]The post script "B" at IRB140B indicates the used set of gear ratios. Other from that the IRB140B is similar to the IRB140.

[8]At this point in time, only prototypes of FRIDA exist. So most technical specifications are not public yet. Some official information on FRIDA can be found at: http://www.abb.com/cawp/abbzh254/8657f5e05ede6ac5c1257861002c8ed2.aspx

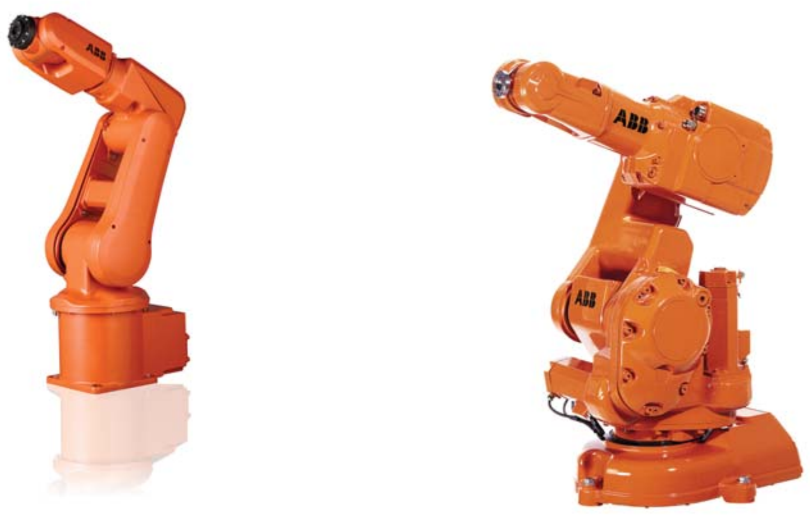


Figure 2.11.: ABB IRB120 (left) and ABB IRB140 (right)

is chosen as necessary to complete a right-handed frame. The origin of each frame i is located at the intersection of its z-axis with the common normal of the z-axes of frame i and $i-1$. This is not necessarily the joint $i-1$, but it is usually in the near neighbourhood of joint $i-1$. This allocation of frames is not entirely unique.

For every joint i the Denavit-Hartenberg convention defines four parameters that fully describe the transformation between frame $i-1$ and frame i (cf. [8]). These parameters are:

link length	a_i:	distance between the z-axes of frame i and frame $i-1$
link offset	d_i:	z-coordinate of the origin of frame i with respect to frame $i-1$
link twist	α_i:	angle between the z-axes of frame $i-1$ and frame i around the x-axis of frame i
joint angle	θ_i:	angle between the x-axes of frame $i-1$ and frame i around the z-axis of frame $i-1$

Depending on the kind of joint - revolute or prismatic - and the respective assembly, for each joint usually three of these parameters are constants. The other one is the joint variable plus a possible offset. This defines the homogeneous transformation matrix of frame i with respect to frame $i-1$ in the following way:

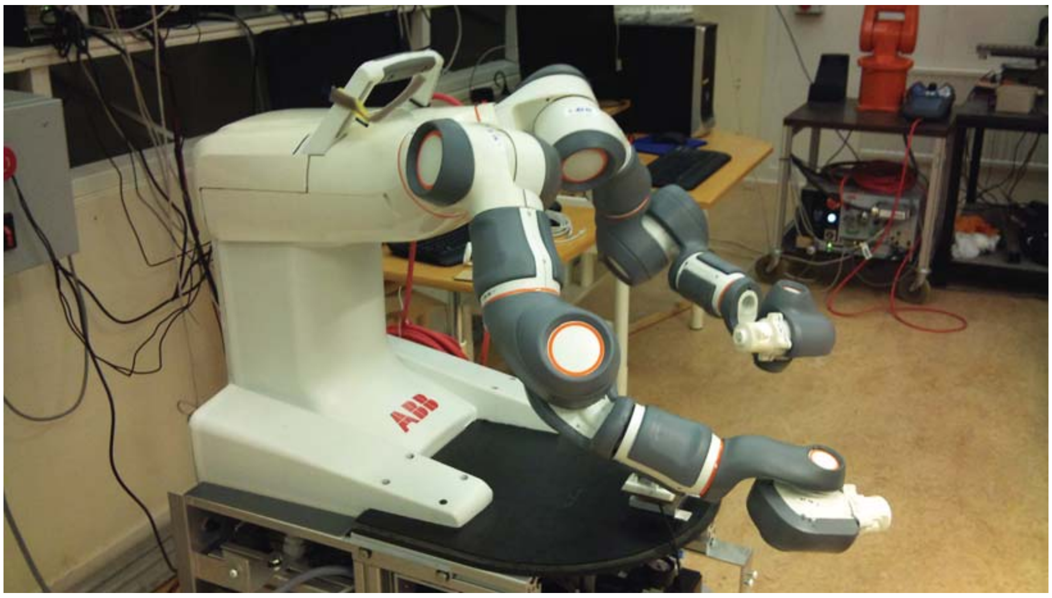


Figure 2.12.: ABB FRIDA

$$\mathbf{A}_i^{i-1} = \mathbf{Rot}_{z_{i-1},\theta_i} \cdot \mathbf{Trans}_{z_{i-1},d_i} \cdot \mathbf{Trans}_{x_i,a_i} \cdot \mathbf{Rot}_{x_i,\alpha_i} = \begin{pmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.19}$$

where c_x and s_x mean $\cos(x)$ and $\sin(x)$. The structure of a homogeneous transformation matrix will be introduced in Section 3.2. Postmultiplying all transformations from the base frame to the robot flange leads to the transformation matrix of the robot flange frame with respect to the base frame. This matrix then depends on all joint variables.

Therefore, given the geometric measures of a robot, it is possible to calculate from the angular position values q_i of each joint i to the position and orientation of the robot flange or the mounted tool relative to the robot's base. This calculation is called the forward kinematics. It is a transformation from the robot joint space description to a cartesian workspace description, for example the homogeneous transformation matrix of the flange frame with respect to the base frame $\mathbf{H}_f^b$. The forward kinematics are a nonlinear problem that always has a unique solution. The block diagrams of the forward kinematics and all other robot transformations that are mentioned in the following are given in Figure 2.15.

The inverse transformation from the cartesian workspace to the robot joint space is called the inverse kinematics. This problem is much more difficult. It is nonlinear, too, but additionally, it is often not unique or the solution cannot be obtained in closed form.

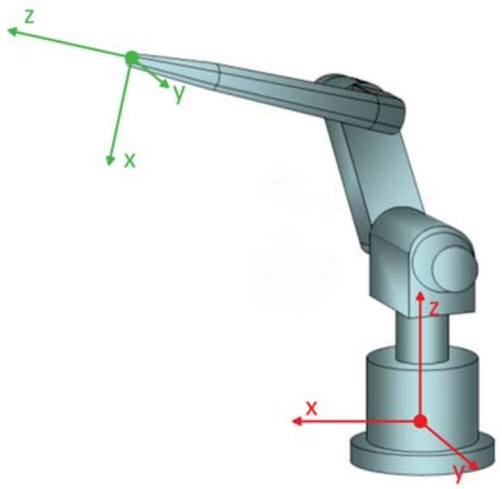


Figure 2.13.: Coordinate frames of the robot base (red) and the robot flange (green)

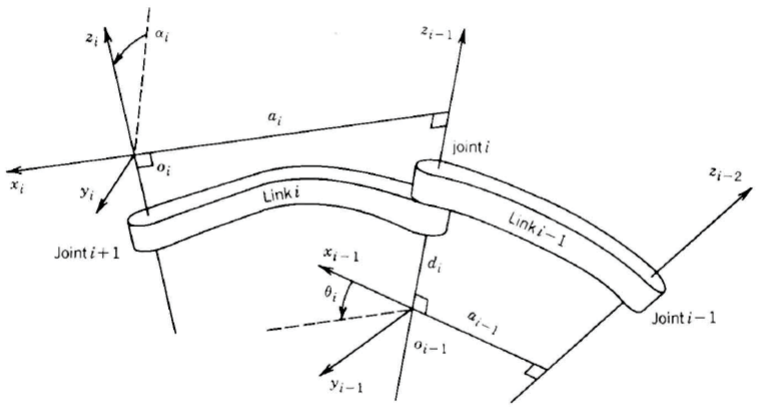


Figure 2.14.: Allocation of frames according to the Denavit-Hartenberg convention, this figure is taken from [8]

For a robot with six ore more DOF inside its workspace, there is always at least one solution for the inverse kinematics.[9]

The transformation relating the joint velocities $\dot{\mathbf{q}}$ and the cartesian linear and angular velocities $\mathbf{v}$, ω of the robot flange or robot tool is called the velocity kinematics. Its inverse problem is called the inverse velocity kinematics. There are configurations, where a movement of the robot in a certain direction is not possible. These points are called kinematic singularities, the inverse kinematics problem has no solution for these configurations. Of course all edges of the robot workspace are singularities, but there are further singularities inside the workspace, for example whenever two axes of a six DOF robot are aligned. In the near neighbourhood of a kinematic singularity even

[9]This is only true inside the workspace and for robots with six or more DOF. At the edges of the workspace even for robots with six ore more DOF the inverse kinematics problem might have no solution.

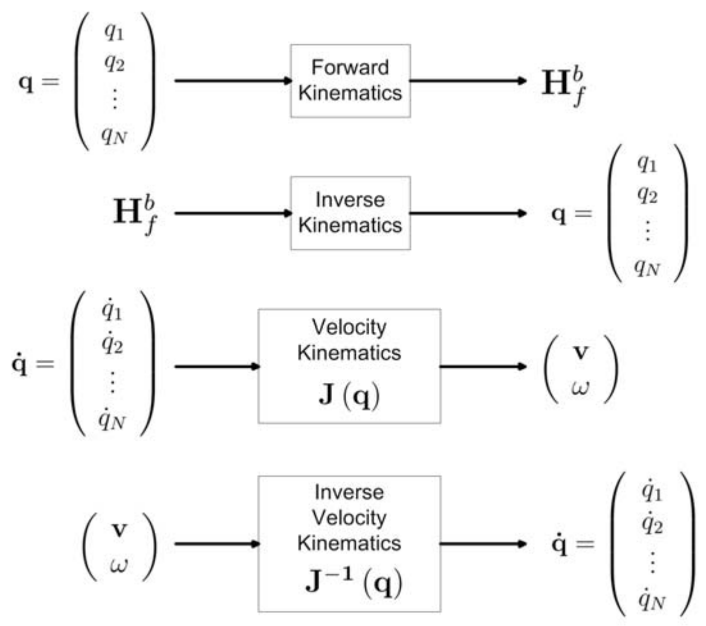

Forward Kinematics
Inverse Kinematics
Velocity Kinematics
Inverse Velocity Kinematics
$\mathbf{q} = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \end{pmatrix}$
$\mathbf{H}_f^b$
$\dot{\mathbf{q}} = \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_N \end{pmatrix}$
$\begin{pmatrix} \mathbf{v} \\ \omega \end{pmatrix}$
$\mathbf{J}(\mathbf{q})$
$\mathbf{J}^{-1}(\mathbf{q})$

Figure 2.15.: Robot Transformations

small cartesian velocities of the robot tool can lead to very high angular velocities of certain joints. So for proper behaviour of the robot in action, these singularities have to be avoided.

The velocity kinematics are usually calculated with the geometric Jacobian $\mathbf{J}$. This is a $(6 \times n)$-matrix, where n is the number of DOF of the robot. The Jacobian $\mathbf{J}$ depends on the respective joint configuration $\mathbf{q}$ itself. So the velocity kinematics can be described as:

$$\begin{pmatrix} \mathbf{v} \\ \omega \end{pmatrix} = \mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} \tag{2.20}$$

This indicates that for small changes of the joint angles the velocity kinematics are approximately a linear problem. Regarding the full workspace they are nonlinear. Due to this local linearity of the velocity kinematics, the inverse velocity kinematics can be calculated for every robot joint configuration by the inverse $\mathbf{J}^{-1}$ of the respective Jacobian $\mathbf{J}(\mathbf{q})$.

As written before, the forward, inverse and velocity kinematics all depend strongly on the geometric dimensions of the robot. However, if they are given, different robot types can be handled in very similar ways on a higher level of abstraction. Then the most important remaining specification of a robot is the number of DOF.

In this thesis for the IRB120 and the IRB140B the forward and inverse kinematics were available. For the FRIDA instead of the inverse kinematics the Jacobian was given. So further geometric information was not relevant.

2.3.1. Robot Controller

The current industrial robot controller by ABB that was used in this thesis is the ABB IRC5 (cf. [9]). It consists of two computers, the main robot computer (MC) and the axis controller computer(AXC), compare Figure 1.1.

The main robot computer is intended to execute the high level robot control like feedforward control and trajectory planning. It sets the reference values and control parameters for the axis controller computer. The axis controller computer performs the low level feedback control separately for each joint. The block diagram of this control is shown in Figure 2.16.

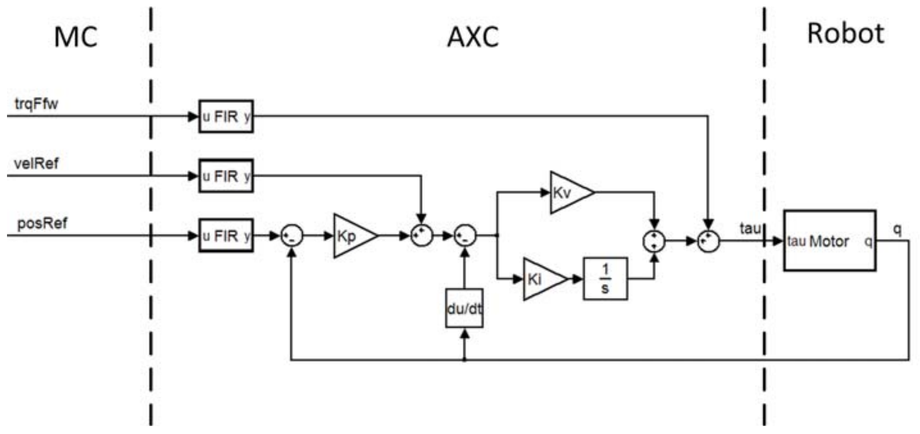


Figure 2.16.: Communication between MC and AXC

2.3.2. ExtCtrl

ExtCtrl is an application to intercept the communication between the main robot computer and the axis controller computer of the IRC5 robot controller of Subsection 2.3.1 and to introduce “External Control”. This way industrial robots can be flexibly applied for experimental use in uncommon setups. The structure of ExtCtrl can be seen in Figure 1.1.

The additional control that is introduced by ExtCtrl is running on the Simulink controller computer. It is an arbitrary Simulink model compiled by Real-Time-Workshop. Among others, the inputs of this model are:

- all reference values of joint positions, velocities and torques and all control parameters that the main robot computer sends to the axis controller,
- all sensory data concerning joint positions, velocities and torques of the robot, that the axis controller sends to the main robot computer
- arbitrary signals that are sent to the Simulink controller by a LabComm connection, cf. Section 2.4. These signals hold the external information, that should be introduced into the robot control.

The outputs of the Simulink model are:

- all signals that the axis controller expects from the main robot computer. The Simulink controller can just pass them through or modify them in an appropriate way.
- arbitrary signals that are sent from the Simulink controller by a LabComm connection, compare Section 2.4.

If the Simulink controller is loaded, but not running the switch S1 in Figure 1.1 is open and the switch S2 in its high position. The communication of the IRC5 is not intercepted. ExtCtrl can run the Simulink controller in two modes: “Submit”, and “Obtain”. In Submit-mode S1 is closed but S2 still in its high position. In this mode the Simulink controller receives the input data by the main robot computer and the axis controller, but its output signals are not sent to the axis controller. This mode can be used to examine and debug new Simulink controllers. In Obtain-mode S1 is closed and S2 in its low position. The Simulink controller fully intercepts the communication of the IRC5 system and may change the signals sent to the axis controller. This way external control is introduced. If the robot is run in ExtCtrl, the robot main computer usually only outputs constant reference values, the actual application is only running on the Simulink controller. All robot experiments of this thesis were conducted in this way.

2.4. LabComm Connection

For the communication between the main computer and the Simulink controller the LabComm protocol is used, cf. Figure 1.1. Information regarding this protocol can be found in [10]. This binary protocol was developed at the Department for Automatic Control of LTH in Lund. It can be used to send an arbitrary amount of primitive data types or arrays. The data to be sent has to be described in a .lc file in the following manner:

```
sample float vref;
sample float T44[16];
```

where `sample` refers to a snapshot of an object. This .lc file has to be saved on both sides of the LabComm connection, the main computer and the Simulink controller. Together with the ORCA/Orcinus protocol [10], also developed at the Department for Automatic Control of LTH in Lund, it can be used for data traffic in both directions. On the side of the main computer, the C-files orca_client.c and orca_client.h offer the functionality to set up a LabComm client, that can encode the data to send and decode the data to receive. On the side of the Simulink controller the tool rtw2orca allows to simply use the sample variables defined in the .lc file as Simulink-input or -output blocks.

3. Control Program

This chapter explains the control program running on the main computer. This was the major effort of this thesis. For some sub-problems there will be given the corresponding code-segment in this chapter. The complete source code can be found in the appendix in Chapter A.

Note: The operating system of the network of the Robotics Lab of the Department for Automatic Control at LTH in Lund is Linux Fedora 14. Unfortunately it was not possible to install a proper driver for the Kinect on this system. The driver which was used (openni-primesense 5.0.0.25[1]) could only handle one Kinect, it returned wrong color values from the RGB camera and it could not detect gestures. On another computer that was not placed in the Robotics Lab and that was running under Linux Kubuntu 10.10 another driver (avin2/SensorKinect 5.0.1.32[2]) could be installed, giving access to all mentioned Kinect functionality and enabling use of multiple Kinect devices. Therefore, due to technical limitations the full application could not be tested as a whole. The single features that are described in the following are all working on one computer, but unfortunately the planned control program including all these features could not be put into effect.

3.1. Informal Description

At the beginning of this thesis the I wrote the following informal description of the robot application to built, as some kind of goal to achieve during this thesis:

Informal Description:

The robot has to be able to fulfill different tasks, e.g.:

- following / mirroring the user's hand
  That can be used as a "learning mode".

[1]Source code, binaries and documentation on: https://github.com/PrimeSense/Sensor
[2]Source code, binaries and documentation on: https://github.com/avin2/SensorKinect

- transporting things to and from the user
- predefined tasks

To handle different tasks there has to be an interactive menu. The user can only gain control to choose tasks in this menu if he performs a focus action which is very unlikely to happen by coincidence.

During any task workspace surveillance has to ensure the robot does not come too close to anything in the workspace.

Exception: The robot has to reach the user for tasks like transporting things to and from the user.

3.2. Calibration

After initialization of the two Kinect sensors they have to be calibrated to find the exact homogeneous transformation matrix $\mathbf{H}$ consisting of the rotation matrix $\mathbf{R}$ and the translation vector $\mathbf{t}$ between them.

$$\mathbf{H} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \tag{3.1}$$

Calibration is done by detecting a set of the same points in the real world by both Kinect sensors individually. Because the two Kinect devices each have a different point of view and therefore use different coordinate systems, the same points are described by different coordinates by the two Kinect sensors. These differences in description of the same points can be used to calculate the homogeneous transformation matrix $\mathbf{H}$. Figure 3.1 shows the setup and the relevant vectors for one calibation point.

The following considerations make use of the notation that a high index to a vector declares the coordinate system in which this vector is expressed and therefore the Kinect from which this vector is seen. So the notations $\mathbf{x}_2^1$ and $\mathbf{x}_2^2$ describe the same vector $\mathbf{x}_2$, but seen from different Kinect sensors.

Then the fact that both Kinect sensors are looking at the same point in Figure 3.1 can be formulated as:

$$\mathbf{x}_1^1 = \mathbf{t}^1 + \mathbf{x}_2^1 = \mathbf{t}^1 + \mathbf{R}_2^1 \cdot \mathbf{x}_2^2 \tag{3.2}$$

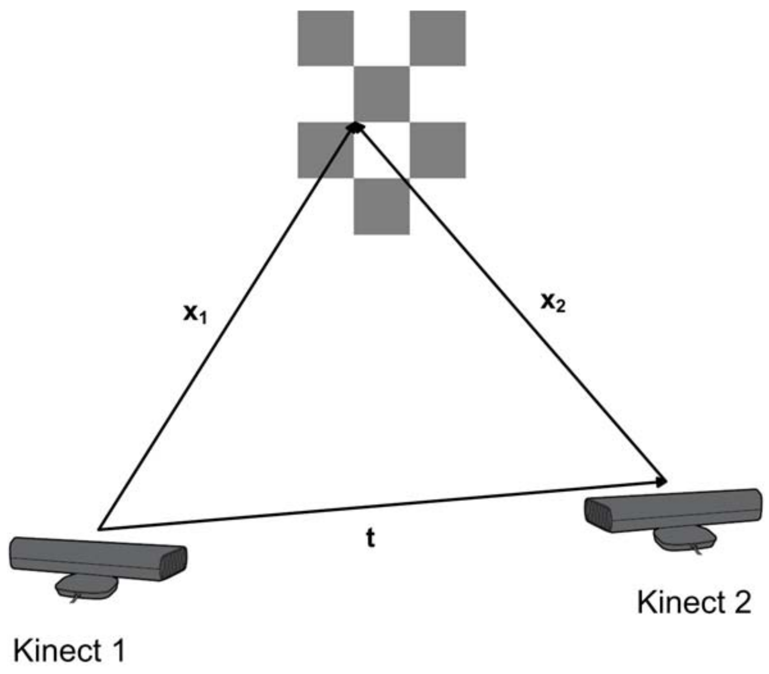


Figure 3.1.: Setup for calibrating the two Kinect devices

where $\mathbf{R}_2^1$ denotes the rotation matrix of the coordinate system of Kinect 2 with respect to the coordinate system of Kinect 1.

If the two descriptions of the point being looked at do not match completely, the error $\mathbf{e}$ between the two descriptions can be expressed as:

$$\mathbf{e} = \mathbf{x}_1^1 - \mathbf{t}^1 - \mathbf{R}_2^1 \cdot \mathbf{x}_2^2 \tag{3.3}$$

Such an error can be defined for every calibration point. If a pair of $\mathbf{R}_2^1$ and $\mathbf{t}^1$ can be found, that minimizes the sum of all quadratic errors, this has to be the best approximation for the real transformation and rotation.

In the following for reasons of better readability the notation depicting the respective coordinate system is skipped again. The vector $\mathbf{x}_i$ means $\mathbf{x}_i^i$ for $i \in [1, 2]$, $\mathbf{t}$ means $\mathbf{t}^1$ and $\mathbf{R}$ means $\mathbf{R}_2^1$.

So as a first step for calibration, a set of suitable calibration points has to be found. Then these points are used do define an error function of the arguments $\mathbf{R}$ and $\mathbf{t}$. The minimizing pair $\mathbf{R}$, $\mathbf{t}$ of this function is the result of the calibration. In this thesis the minimization was done in two ways, using linear least-squares estimation and using nonlinear function approximations. A comparison of these two methods will be presented concluding this chapter.

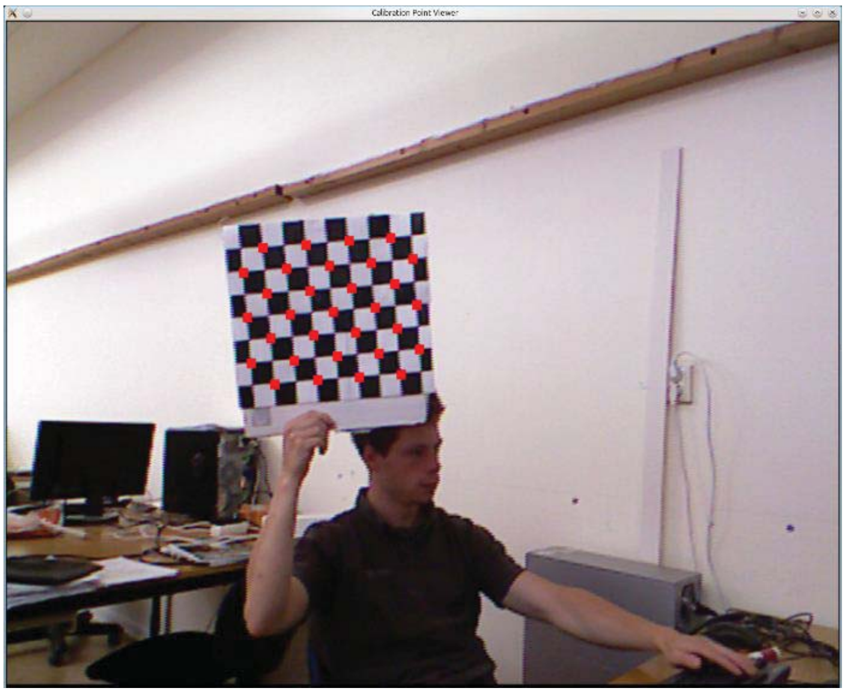


Figure 3.2.: Calibration Board and detected Calibration Points

3.2.1. Calibration Points

Calibration in this thesis was done with a chess pattern drawn on the calibration board to be seen in Figure 3.2. Calibration is the only purpose for which the RGB-camera of the Kinect is used. The function "findCalibrationPoints(•)" (cf. Section A.1 in the appendix) is called for each Kinect. It is used to find points that are at the corner between four different areas in this pattern, where the top left area is black, the top right white, the bottom left white and the bottom right black.[3]

At first this function sums up for every pixel its three color values, to get a single grey value in the range between 0 and 765.

Then for every window of 10x10 pixels its correlation coefficient ρ to an ideal chess pattern corner is calculated. This ideal chess pattern corner consists of four areas, each 5x5 pixels large, where also the top left area is black, the top right white, the bottom left white and the bottom right black. The grey values of this ideal corner are zero for the black parts and 765 for the white ones. So both the mean value μ_2 and the standard deviation σ_2 of this ideal corner are 765/2. The middle value μ_1 and the standard deviation σ_1 of the respective window in the image and the covariance σ_{12} have to be calculated for every single window again.

[3]OpenNI draws a picture of the scene as a mirror would show it. This means right and left are changed in comparison to reality. In this report I follow this convention, so unless explicitly mentioned the real sides are just the other way around.

$$\rho = \frac{\sigma_{12}}{\sigma_1 \cdot \sigma_2} \tag{3.4}$$

where the usual statistical definitions for σ_1, σ_2 and σ_{12} are applied. Note that σ_{12} like a variance is a quadratic expression, whereas σ_1 and σ_2 as standard deviations are the square roots of quadratic expressions.

On the used calibration board there are 31 calibration points. So the function "findCalibrationPoints(•)" selects the 31 points with the highest correlation coefficients in the order from the highest correlation coefficient down to the lower ones. It is possible that one single actual calibration point results in more than one point with a high correlation coefficient. Therefore for every new point to be selected the function evaluates, whether there is another point already selected less then 8 pixels away from the current one. If that is the case, the new point is not regarded.

Finally, the function "findCalibrationPoints(•)" has to sort the selected calibration points in a distinct order, so that every single calibration point seen from one Kinect can be matched to the same one as seen from the other Kinect. Otherwise, identification would be impossible. The rules to sort the calibration points are: Left points are sorted before right points and, if two points have the same horizontal coordinate, the higher point is sorted before the lower point. This is done in projective coordinates because the image generator cannot offer real world coordinates. Hence, due to the different camera transformations that the Kinect RGB-cameras apply to the real world, the calibration points could be projected in a way that would interfere with the rules to sort them. Therefore, due to different projections the calibration points could be still not sorted correctly. This can be avoided by positioning the two Kinect units and the calibration board in approximately the same height, i.e. with the same y-coordinate, and tilting the calibration board slightly around its normal vector like it is done in Figure 3.2.

So the function "findCalibrationPoints(•)" returns the x- and y-coordinates of the projective coordinates of 31 sorted calibration points. The depth generator can match the respective depth to these x- and y-coordinates to complete the triple of projective coordinates for each calibration point. This is the reason, why the origin of the depth coordinate system of the Kinect should be chosen as the RGB-camera. This way projective color and depth data are aligned.

The full triple of projective coordinates can then be transformed to real world coordinates from the point of view of each Kinect. To measure the quality of the set of calibration points, the real world distances between all pairs of consecutive calibration points are calculated and stored in a key vector with 30 elements. So far, all these calculations are performed for both Kinect sensors individually. So as a result there are two such key vectors, that ideally should be exactly the same. In reality they should not differ too much, otherwise that would indicate that the function "findCalibrationPoints(•)" did not return the same calibration points in the same order for both Kinect devices.

To measure the degree of consistency of these two keys, two coefficients are used. The first one is the correlation coefficient ρ between the two keys, defined just like in Equation (3.4), only now applied to the elements of the key vectors. The range of ρ are all real values in the interval $[-1, 1]$. I called the second coefficient ρ_2. Given a pair of vectors $\mathbf{X} = (x_1, x_2, \ldots, x_N)$ and $\mathbf{Y} = (y_1, y_2, \ldots, y_N)$, to which ρ_2 is applied to, it is defined as:

$$\rho_2 = 2 \cdot \frac{\sum_{i=1}^{N} (x_i \cdot y_i)}{\sum_{i=1}^{N} (x_i^2) + \sum_{i=1}^{N} (y_i^2)} \tag{3.5}$$

The range of ρ_2 are all real values in the interval $[0, 1]$.

Unless both these coefficients are at least 0.85, the set of calibration points is discarded and a new set is searched for by calling the function "findCalibrationPoints(•)" for both Kinect sensors again. It is important to check both coefficients because neither of them is sufficient on its own. The correlation coefficient returns high values if the direction of both key vectors is similar, even if the absolute values differ a lot. The coefficient $\rho 2$ returns high values, if the absolute numbers in both key vectors are similar but not in the same order. Only if both coefficients are high one can assume that the two key vectors and therefore the two sets of calibration points are nearly the same.

By practical experience I can say, each coefficient can be low while the other one is high. But if both are high, they are usually very high (~ 0.99). Thus the threshold 0.85 actually could vary a lot without noticeably changing the calibration performance.

During the calibration process, three pictures of the calibration board are taken and the calibration points extracted. For every picture the calibration board should be in a different location. So afterwards there are 93 calibration points available, each of them described by two vectors, one from each Kinect.

The 93 different points are indexed with a superscript bracketed number. Therefore, for every $k \in [1..93]$, $\mathbf{x_1^{(k)}}$ and $\mathbf{x_2^{(k)}}$, both describe vectors to the k^{th} calibration point, $\mathbf{x_1^{(k)}}$ from Kinect 1 and as seen by Kinect 1, $\mathbf{x_2^{(k)}}$ from Kinect 2 and as seen by Kinect 2.

3.2.2. Calibration using linear Least-Squares Estimation

Linear least-squares estimation is a common technique to identify suitable values for an arbitrary number of parameters given an adequately big set of training data. If a linear scalar error equation of the form

$$\varepsilon_k = y_k - \mathbf{g}_k \cdot \mathbf{m} \tag{3.6}$$

can be formulated, where $\mathbf{m}$ is a column vector of the parameters to identify and the scalar y_k and the row vector $\mathbf{g}_k$ are one piece of the set of training data, linear least-squares estimation is applicable.

The idea behind the least-squares algorithm is to minimize the cost function

$$J_l = \sum_{k=1}^{N} \varepsilon_k^2 \tag{3.7}$$

with respect to the parameter vector $\mathbf{m}$. The minimizing parameter vector $\mathbf{m}$ is the solution of the least-squares algorithm. By inserting Equation (3.6) into Equation (3.7) and setting the derivative equal to zero the following expression for $\mathbf{m}$ can be found.

$$\mathbf{m} = \left(\mathbf{G}^T\mathbf{G}\right)^{-1} \cdot \mathbf{G}^T\mathbf{y} \tag{3.8}$$

In this equation $\mathbf{G}$ is a matrix containing all N row vectors $\mathbf{g}_k$ of Equation (3.6) as column vectors and $\mathbf{y}$ is a column vector containing all N scalars y_k of Equation (3.6). Further information concerning the least-squares method can be found for example in [11] in Chapter 4.

Equation (3.3) can be rewritten as:

$$\mathbf{e}^T = \mathbf{x}_1^T - \begin{pmatrix} 1 & \mathbf{x}_2^T \end{pmatrix} \cdot \begin{pmatrix} \mathbf{t}^T \\ \mathbf{R}^T \end{pmatrix} \tag{3.9}$$

which is mainly just the transposed version of Equation (3.3).

If $\quad \mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \quad$ and $\quad \mathbf{R} = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix},$

the coordinate form of Equation (3.9) is:

$$\begin{pmatrix} e_x & e_y & e_z \end{pmatrix} = \begin{pmatrix} x_{1_x} & x_{1_y} & x_{1_z} \end{pmatrix} - \begin{pmatrix} 1 & x_{2_x} & x_{2_y} & x_{2_z} \end{pmatrix} \cdot \begin{pmatrix} t_x & t_y & t_z \\ R_{11} & R_{21} & R_{31} \\ R_{12} & R_{22} & R_{32} \\ R_{13} & R_{23} & R_{33} \end{pmatrix} \tag{3.10}$$

This can be interpreted as three parallel scalar error equations of the form of Equation (3.6) for the three coordinate directions. This means values for **t** and **R** can be identified by three parallel least-squares algorithms with the same matrix **G**, but different vectors **y** and **m**. Because these three least-squares algorithms use the same matrix **G**, they can be calculated concurrently. Then the vectors of Equation (3.6) just become matrices as well. Using the ordered set of calibration points of Section 3.2.1, the solution of this triple Least Squares algorithm is:

$$\mathbf{M} = \left(\mathbf{G}^T\mathbf{G}\right)^{-1} \cdot \mathbf{G}^T\mathbf{Y} \tag{3.11}$$

with $\quad \mathbf{M} = \begin{pmatrix} t_x & t_y & t_z \\ R_{11} & R_{21} & R_{31} \\ R_{12} & R_{22} & R_{32} \\ R_{13} & R_{23} & R_{33} \end{pmatrix}, \qquad \mathbf{Y} = \begin{pmatrix} \mathbf{x}_{1_x} & \mathbf{x}_{1_y} & \mathbf{x}_{1_z} \end{pmatrix}$ and

$\qquad \mathbf{G} = \begin{pmatrix} \mathbf{1} & \mathbf{x}_{2_x} & \mathbf{x}_{2_y} & \mathbf{x}_{2_z} \end{pmatrix},$
(cf. Equation (3.10)).

Where $\mathbf{x}_{i_j}$ is the column vector containing the j-coordinates of all vectors $\mathbf{x}_i^{(k)}$ of the calibration points for $i \in [1, 2]$, $j \in [x, y, z]$ and $k \in [1..93]$.

So all twelve wanted parameters can be calculated using this triple least-squares algorithm. In the control program the matrix inversion of Equation (3.11) is performed with the function "spdmatrixinverse(•)" of the open "alglib" library[4].

[4]online available under http://www.alglib.net/

3.2.3. Calibration using nonlinear Function Optimization

The drawback of the least-squares method is that it disregards the inner dependencies of the rotation matrix **R**. Even though this matrix contains nine elements, a full rotation can be described by only tree independent parameters. Together with the tree parameters for the translation vector **t** the hole transformation matrix **H** between Kinect 1 and Kinect 2 can be expressed by only six parameters instead of twelve as using Least Squares.

One way to express a rotation with only three parameters is with a rotation vector based on the axis/angle representation as introduced in [8] in Chapter 2. The three parameters k_x, k_y and k_z define a vector

$$\mathbf{k} = \begin{pmatrix} k_x \\ k_y \\ k_z \end{pmatrix} \tag{3.12}$$

with the length

$$\theta = \|\mathbf{k}\| = \sqrt{k_x^2 + k_y^2 + k_z^2} \tag{3.13}$$

In axis/angle representation this describes a rotation of θ in radiant around the axis defined by the vector **k**,cf. Figure 3.3.

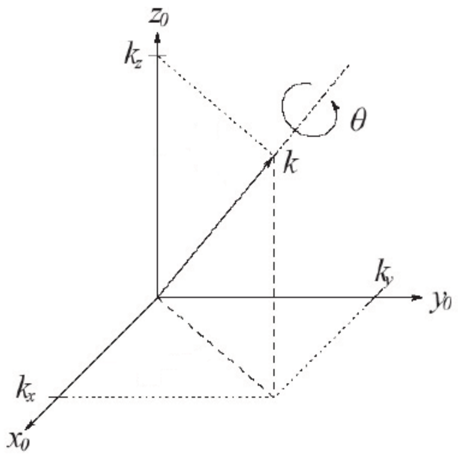


Figure 3.3.: Rotation Vector in Axis/Angle Representation

The corresponding rotation matrix to this rotation vector is:

$$\mathbf{R}_{\mathbf{k},\theta} = \begin{pmatrix} \frac{k_x^2}{\theta^2}(1-\cos\theta)+\cos\theta & \frac{k_xk_y}{\theta^2}(1-\cos\theta)-\frac{k_z}{\theta}\sin\theta & \frac{k_xk_z}{\theta^2}(1-\cos\theta)+\frac{k_y}{\theta}\sin\theta \\ \frac{k_xk_y}{\theta^2}(1-\cos\theta)+\frac{k_z}{\theta}\sin\theta & \frac{k_y^2}{\theta^2}(1-\cos\theta)+\cos\theta & \frac{k_yk_z}{\theta^2}(1-\cos\theta)-\frac{k_x}{\theta}\sin\theta \\ \frac{k_xk_z}{\theta^2}(1-\cos\theta)-\frac{k_y}{\theta}\sin\theta & \frac{k_yk_z}{\theta^2}(1-\cos\theta)+\frac{k_x}{\theta}\sin\theta & \frac{k_z^2}{\theta^2}(1-\cos\theta)+\cos\theta \end{pmatrix} \tag{3.14}$$

This matrix is based on the formulation of the axis/angle representation given in [8].

If we define a cost function of the form

$$J_n = \sum_{k=1}^{N} \|\mathbf{e}_k\|^2 , \tag{3.15}$$

where we insert Equation (3.3) for $\mathbf{e}_k$ and use Equation (3.14) as rotation matrix, we obtain the following problem specific cost function:

$$J_n = \sum_{k=1}^{N} \left\| \mathbf{x}_1^{(k)} - \mathbf{t}^1 - \mathbf{R}(\mathbf{k}) \cdot \mathbf{x}_2^{(k)} \right\|^2 \tag{3.16}$$

Note the difference between the vector $\mathbf{k}$ of the axis/angle representation and the scalar count variable k. If we apply Equation (3.16) to all calibration points ($k \in [1..93]$), the result is a cost function depending only on the six independent parameters t_x, t_y, t_z, k_x, k_y and k_z, whose global minimum with respect to these parameters will indicate the best approximation for the translation and rotation between the two Kinect devices.

Because of the highly nonlinear rotation matrix and the squared error, the cost function of Equation (3.16) is highly nonlinear, too. Therefore, to minimize it nonlinear function optimization methods are necessary. In the control program the Nelder-Mead method is applied.

The Nelder-Mead method is a robust algorithm, that minimizes an arbitrary function f of N independent variables without calculating any derivatives. It uses a N-dimensional simplex[5], that moves to lower values of the function f and then shrinks around a local minimum. In every step the function f is calculated at each corner point. Then the

[5]A simplex is a generalized triangle. The two dimensional simplex is a triangle. A N-dimensional simplex is the easiest N-dimensional geometric body spread out by $N+1$ corner points, that are all connected to each other by edges. Each subset containing N corner points has to form a simplex of the dimension N-1.

corner point with the worst function value is replaced, usually by its reflection around the center of all other points. If no better point than the worst corner point can be found outside the simplex, the simplex is shrunk by moving every corner half the way to the best corner point. Further information about the Nelder-Mead method can be found in [12] in Chapter 8.

Applying the Nelder-Mead algorithm to the function of Equation (3.16) yields six minimizing values for the parameters t_x, t_y, t_z, k_x, k_y and k_z. Hence, for the vector $\mathbf{t}$ all components are already calculated, for the rotation Equation (3.14) has to be applied, to extract the matrix $\mathbf{R}$ out of the parameters k_x, k_y and k_z.

In the program the Nelder-Mead algorithm is performed by the function "nelmin(•)" of the open "asa047" library[6].

3.2.4. Comparison

Given the set of 93 calibration points, both presented calibration algorithms will return proper values for the transformation vector $\mathbf{t}$ and the rotation matrix $\mathbf{R}$.

The least-squares method will analytically find the single minimizing matrix $\mathbf{R}$ and vector $\mathbf{t}$ of its squared linear cost function. However, as written before it treats these twelve parameters as totally independent. That does not just mean a lot of additional effort, to find twelve instead of six parameters, it also disregards the orthogonality characteristics of a rotation matrix. For every rotation matrix the following equation has to hold:

$$\mathbf{R} \cdot \mathbf{R}^T = \mathbf{I} \tag{3.17}$$

Because it is not possible to handle constraints in the least-squares algorithm, these characteristics of a rotation have never been regarded in the derivation of the algorithm. So it is not implied that a rotation matrix calculated by the least-squares algorithm actually fulfills this constraint. On the other hand this is also a means to judge the quality of the calculated matrix. If the product of $\mathbf{R}$ and its transposed in fact is very similar to the identity matrix, even though that has never been implied, the values of this rotation matrix are probably very accurate.

In axis/angle representation the characteristics of a rotation matrix are implied. So if Equation (3.14) is inserted into Equation (3.17), it will always hold.

The Nelder-Mead algorithm is only an approximation, so it will not return an analytical minimum. There is even the danger, that the Nelder-Mead algorithm might get stuck

[6]online available under http://people.sc.fsu.edu/~jburkardt/cpp_src/asa047/asa047.html

in a local side minimum. But this error can be easily detected and worked around by a different starting point.

To decide, which calculation method to use, a snapshot of a person sitting on a table was taken by two Kinect sensors. These two Kinect devices where set about two meters from each other and their orientation differed by an angle of about 90 degrees turned around the y-axis. It was intentional not to use an exact experiment setup, because just that is the task of calibration, to find the exact position and orientation information for a rough setup.

Calibration of the two Kinect sensors led to the following results:

$$\mathbf{H}_{LeastSquares} = \begin{pmatrix} -0.0213 & -0.0202 & 0.9298 & -1385.28 \\ 0.0722 & 0.7023 & 0.0978 & -156.286 \\ -1.0712 & 0.1171 & -0.0169 & 1583.6 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.18}$$

$$\mathbf{H}_{NelderMead} = \begin{pmatrix} -0.0124 & -0.1075 & 0.9941 & -1524.93 \\ 0.0900 & 0.9903 & 0.1060 & -103.366 \\ -0.9959 & 0.0882 & -0.0220 & 1519.57 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.19}$$

For $\mathbf{R}_{LeastSquares}$ being the rotation matrix of $\mathbf{H}_{LeastSquares}$ the orthogonality characteristics defined in Equation (3.17) are checked:

$$\mathbf{R}_{LeastSquares} \cdot \mathbf{R}^T_{LeastSquares} = \begin{pmatrix} 0.8654 & 0.0752 & 0.0048 \\ 0.0752 & 0.5080 & 0.0033 \\ 0.0048 & 0.0033 & 1.1615 \end{pmatrix} \tag{3.20}$$

This product differs strongly from the identity matrix especially regarding the second coordinate. Therefore it is in conflict with Equation (3.17).

The results of the snapshot and the consecutive transformation of the data of Kinect 2 are plotted in Figure 3.4. For clearer representation in this figure only pixels of every forth line and column are shown. So in total this person could be drawn by 16 times as much points.

The influence of the non-ideally orthogonal rotation matrix of the least-squares method can be observed best in subplot 2 of Figure 3.4, where the transformed points of Kinect 2 just do not match the points of Kinect 1 as well as the points in subplot 3 that are transformed using the Nelder-Mead method do. Especially in y-direction, the second coordinate, the picture of the points that are transformed by the Least Squares method

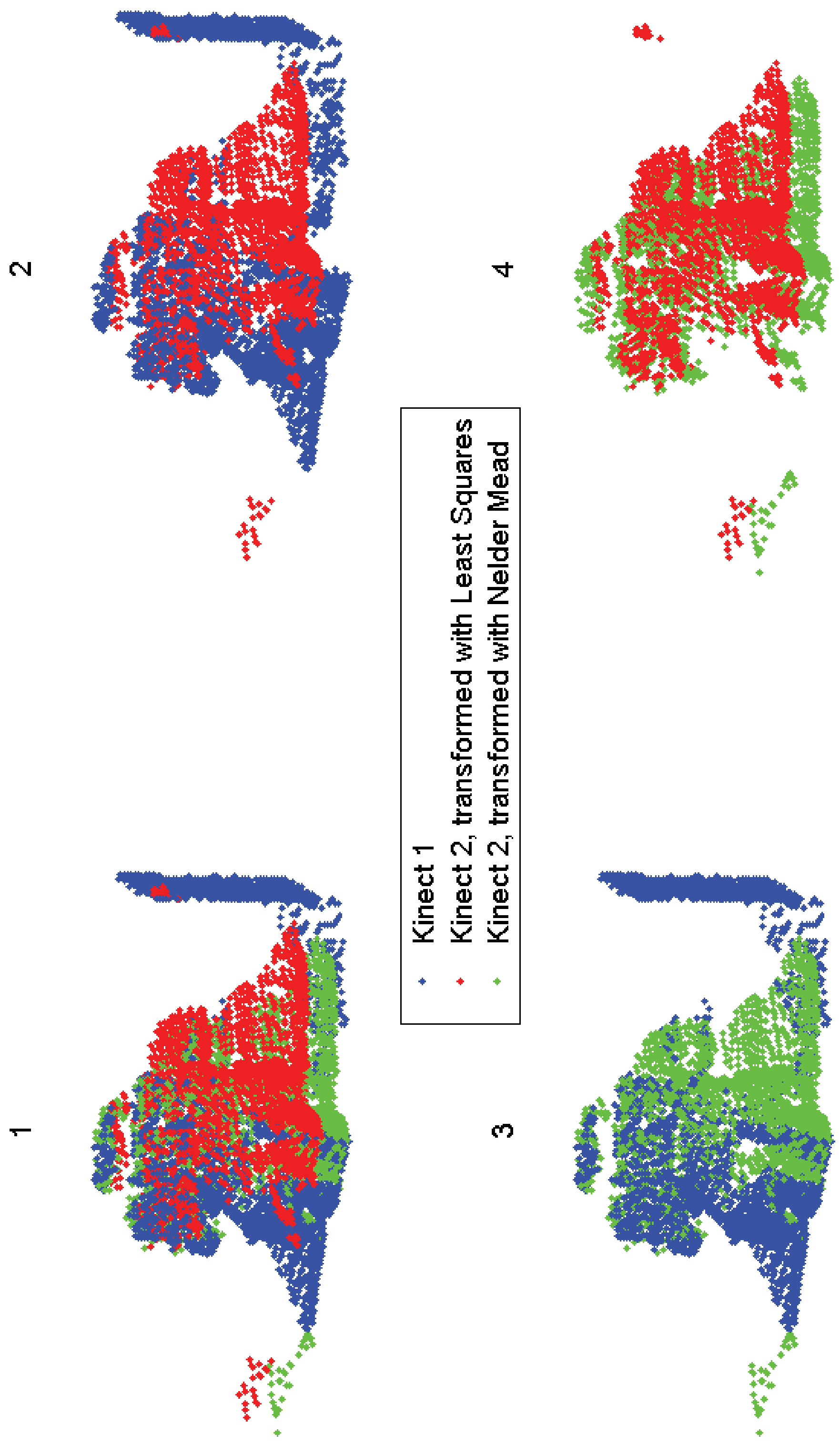


Figure 3.4.: Snapshot of a person sitting on a table, seen by two Kinect sensors

are distorted in comparison to the points of Kinect 1. This agrees with the observation that Equation (3.20) delivered the worst values in the second coordinate.

On the other hand, the points transformed using the Nelder-Mead method lead together with the points of Kinect 1 to a steady, continuous picture of the person. So for the application the calibration calculations are done with the Nelder Mead method.

The main advantage of the sensor fusion of two Kinect devices can be observed in Figure 3.4, too. The person is seen from two sides. The right side is in the shadow of Kinect 1 and the left side in the shadow of Kinect 2. Together the two Kinect sensors deliver a very high range of vision of the scene in the observed workspace.

3.3. Workspace Surveillance

The purpose of workspace surveillance is to protect humans that work next to a robot from possible collisions with the robot. That is especially important if it is a very heavy robot or if the robot is holding a dangerous tool, for example a sharp drill. In the scenario of future manufacturing described in the very beginning of this report, this situation of small local distances between humans and robots is quite common due to the close interaction. Therefore workspace surveillance will be an important aspect of future manufacturing planning.

The experiments of this section were conducted with the FRIDA.

The task a workspace surveillance application has to fulfill is easily described as preventing the robot from moving too close to anything in its neighbourhood except for the pieces it is working on.

In this thesis this was achieved by the Simulink controller sending the coordinates of the robot tool expressed in the robot base frame to the main computer. There the tool position is transformed into a description in the Kinect frame by a constant homogeneous transformation matrix[7] defining position and orientation relations between robot and Kinect. Then for every point that the Kinect - or possibly the multiple Kinect devices

[7]This transformation matrix should be found in a similar way as described in the calibration Section 3.2, by a chess pattern rigidly attached to the robot. Unfortunately as written before, the computer in the Robotics Lab could not handle the RGB-camera of the Kinect, so the calibration of Section 3.2 could not be used. Instead the transformation matrix was found by measuring the geometric distances. This leads to proper results, too, but it is not in the sense of this thesis and the ROSETTA project, because it is not flexible. Once an assembly is measured, no component may be moved any more. Whereas using automatic calibration the setup can be varied arbitrarily without too much effort.

- has or have detected, the distance[8] between that respective point and the robot tool is calculated and compared to a "safe distance" value. In experiment I chose 300 mm as safe distance. If the closest detected point is farther away from the robot tool than safe distance, the robot can move according to its task, else this closest point is regarded as a "dangerous point" and the robot is stopped, by setting the velocity reference to zero, until the sphere with the radius safe distance around the robot tool is clear again.

Of course the Kinect will always detect the robot itself and therefore the main computer would always stop the robot, unless these points belonging to the robot are neglected by the main computer when searching for the closest point to the robot tool. Therefore the Simulink controller additionally to the tool position sends the positions, where the coordinate frames at the second, the third and the fifth joint originate. As written in Section 2.3, these usually are not the exact positions of the joints themselves, but they are very near to these joints. Even though in robot control these position values are rarely used explicitly, they are always calculated as part of the forward Kinematics (cf. Section 2.3). So finding these values does not need any additional computations, they just have to be output and sent to the main computer by the Simulink controller.

The main computer then allocates 19 "tolerance points" on the line segments between the tool and the joints and adds for every tolerance point a tolerance radius, describing the sphere around that tolerance point, that should be ignored, when searching for the closest point to the robot tool. The exact locations and radii of these tolerance points are listed in table 3.1. This way the robot arm, modelled as the combination of 19 spheres with different radii, is cut out of the batch of all points regarded for the search for the closest point.

Two snapshot of two scenes, showing how the main computer matches Kinect and robot data, are given in Figure 3.5. Especially the distribution of the tolerance points along the robot arm can be observed well. In the left picture of Figure 3.5 a dangerous point is detected. It belongs to a human hand. So in the scene of the left picture the robot movement is being suspended.

Except from the robot arm itself there are other pieces that may not be taken into account during the search for the closest point. These are all pieces that are necessarily nearer to the robot tool than safe distance. In the example of this application this was the work piece itself, the work table as a platform for robot and work piece and the robot socket. All these pieces were modelled as cuboids described by their front bottom right and back top left corners. All points included in any of these cuboids are disregarded in the search for the closest point, too. This can be observed in the right picture of Figure 3.5. The work piece (marked) and the table are much nearer to

[8]Actually the squared distance is calculated. The distance between two points in vector space would be calculated following the law of Pythagoras as the square root of the sum of all squared differences of the single vector elements. To spare computation power this square root is not calculated and the following comparison is conducted with the square of the safe distance value.

Tolerance Point	Location	Tolerance Radius / [mm]
1	tool	30
2	1/5 from tool to joint 5	60
3	2/5 from tool to joint 5	60
4	3/5 from tool to joint 5	60
5	4/5 from tool to joint 5	60
6	joint 5	130
7	1/7 from joint 5 to joint 3	100
8	2/7 from joint 5 to joint 3	100
9	3/7 from joint 5 to joint 3	100
10	4/7 from joint 5 to joint 3	100
11	5/7 from joint 5 to joint 3	100
12	6/7 from joint 5 to joint 3	100
13	joint 3	140
14	1/6 from joint 3 to joint 2	100
15	2/6 from joint 3 to joint 2	100
16	3/6 from joint 3 to joint 2	100
17	4/6 from joint 3 to joint 2	100
18	5/6 from joint 3 to joint 2	100
19	joint 2	130

Table 3.1.: Tolerance points

the robot tool than safe distance. But the workspace surveillance application still does not find a dangerous point in this scene, because work piece and table are modelled as cubics, that are allowed to be next to the tool.

This way of modeling all pieces that may be near to the robot tool means a lot of effort that even has to be repeated, whenever there is a change of the setup. But in a real manufacturing scenario instead of cubics CAD-data of all items can be used and integrated into the workspace surveillance program in an automated way. This does not just lower the necessary modeling effort, it also makes this application more flexible for any changes of the setting.

Due to errors in the transformation matrix from the robot base frame to the Kinect frame, due to the approximation to find the robot joints in the origins of the respective robot frames and due to uncertainty in the depth measurement by the Kinect, the tolerance radii and the lengths of the cuboids have to be larger than the actual measures, to avoid false positives. That means the detection of a dangerous point, when there actually is none.

The effects of the third point, the uncertainty of the measurement by the Kinect, can be decreased in another way. In experiment it showed that the depth picture of the Kinect is especially noisy around the edges of the different shapes in vision.

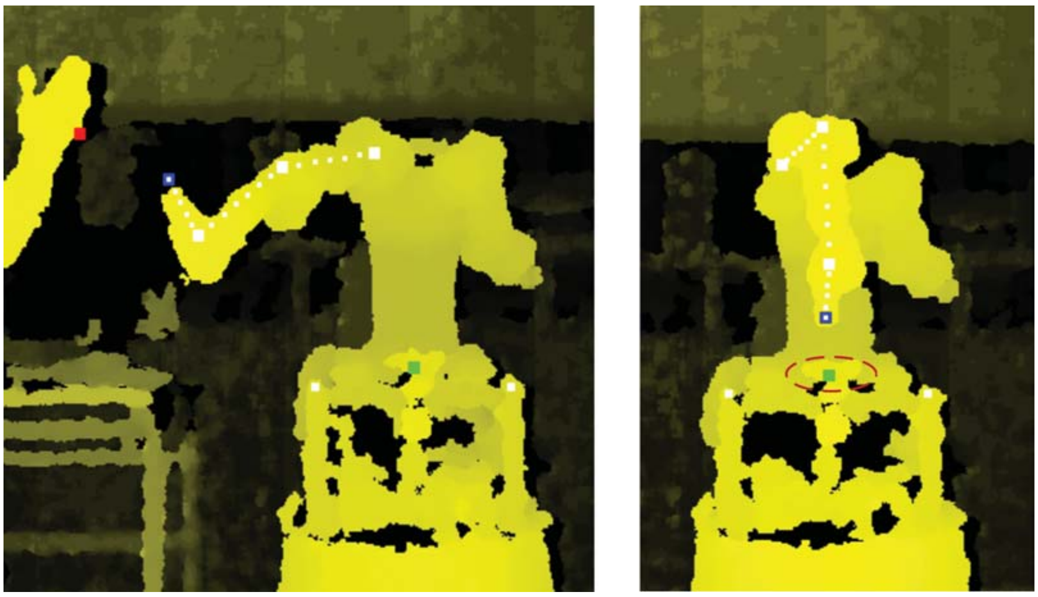


Figure 3.5.: Snapshots of Workspace Surveillance as seen by the Kinect

red point:	**dangerous point**
green point:	**origin of robot base frame (not on the surface)**
blue point:	**robot tool**
big white points:	**robot joints**
small white points:	**tolerance points**

The two medium sized white points were used for calibration and are not relevant for workspace surveillance.

Unfortunately detecting sharp edges of the robot arm is exactly what would be needed for this application. But as errors due to noisy measurement are unlikely to occur twice in a row, the number of false positives can be reduced strongly if only consecutively detected dangerous points are forwarded to the Simulink controller. This means in each picture frame, the main computer only stops the robot if it has detected a dangerous point in both, this current frame and the frame one step before. So the reaction to an object closer to the robot tool than safe distance will happen one frame later, that is 0.033 s at a frame rate of 30 frames/s. But the number of detected false positives is much smaller this way.

During an experiment conducted with the FRIDA while standing still, in 25 minutes 1136 false positives were detected at all, but only 36 consecutive false positives that would have been forwarded to the robot.

As postulated in the informal description the workspace surveillance should be weakened when the robot is transporting things to and from the user. Additionally regarding a robot that is considered safe like FRIDA and that does not operate dangerous tools, it may not be necessary to completely stop all robot movement. Instead the robot velocity should just be decreased the nearer a human gets to the robot. For that reason the angular and linear velocity reference[9] of the robot tool is multiplied by an additional factor between zero and one that depends on the distance of the tool to a near object. This factor $v_{ref,\,rel}$ is calculated by the main computer the following way and then sent to the Simulink controller via the LabComm connection.

$$v_{ref,\,rel} = \min\left(1,\ \frac{\max\left(0,\ d_{closest} - 60\right)}{d_{safe} - 60}\right) \tag{3.21}$$

where $d_{closest}$ is the closest point and d_{safe} is the safe distance. The offset -60 [mm] of $d_{closest}$ is added to make sure the robot will finally stop next to the closest object even if the measured positions hold some uncertainty. The same offset is also added to d_{safe}, to keep the robot velocity continuous when crossing safe distance.

If the robot is supposed to reach the user's hand, Equation (3.21) leads to an exponentially slowed down movement inside the sphere with radius safe distance around the users hand, as the changing of the distance depends on the actual distance itself:

$$v_{tool} = \dot{d}_{closest} = -v_0 \cdot \frac{d_{closest} - 60}{d_{safe} - 60} \tag{3.22}$$

where v_0 is the tool velocity the Simulink controller would apply in the absence of workspace surveillance. Assuming the distance between the robot tool and the users hand is safe distance at time $t = 0$, the solution to this differential equation is:

$$d_{closest}\left(t\right) = 60 + \left(d_{safe} - 60\right) \cdot e^{\frac{-v_0}{d_{safe} - 60} \cdot t} \tag{3.23}$$

For $v_0 = 200\,[mm/s]$ this movement has a time constant of $\tau = 1.2\,\mathrm{s}$.

3.4. Face Detection

As established in the informal description on page 29, the focus action that puts the user who performed it into control over the application, has to be very unlikely to happen by coincidence. Otherwise any person in vision could unintendedly take over the control, which could lead to undefined and even dangerous behaviour of the robot. But even though the focus action has to be safe concerning its unlikeliness to happen by coincidence, it should not be too complicated either, given that the purpose of this whole application is to make human robot interaction more intuitive and therefore easier.

[9]The relations between angular and linear velocity, joint velocities and joint positions in a Simulink controller for FRIDA will be dealt with closely in Section 3.5.

So for the control program of this thesis I chose a combined gesture including hand and head as focus. The user has to perform a simple “Click”-movement with one hand while looking directly at one Kinect at the same time. The thought behind this is that while a simple “Click”-movement might happen by coincidence sometimes, it is rather unlikely that a user, who is looking at a Kinect and thus seeing it, performs this “Click” unintendedly.

This means if the control program is in a state, where no controlling user is present, and it detects a “Click”-movement, it first has to check if the user, who performed the “Click” was looking at the Kinect at the same time. Only if that is the case, the main menu is started.

The face detection in this program is done based on the work presented in [13]. The authors of [13] used a curvature analysis of 3D-data to detect faces in depth images. Their method was to independently detect nose-like and eye-like areas and then try and combine those parts to one or more faces. The nose and the inside corners of the eyes have high curvature values.

There are two major differences between the problem dealt with in [13] and the problem of this application. Firstly, they allowed partly occlusions and arbitrary orientations of the faces in the pictures. In this application, as a strait face directly looking at the Kinect is a requirement in the interest of safety, rotated faces may not be detected. Secondly, even though they did not publish the resolution of the 3D-pictures they presented as examples in [13], guessing from the look of these pictures, these seemed to have a much higher resolution than the face-excerpts of pictures of the Kinect.

Because of the comparably low pixel-resolution of the Kinect, the eye area that can be detected by curvature analysis is only a small amount of single pixels and therefore very vulnerable to noise. Therefore in this application only the nose is detected. To make this detection more distinct also the first derivatives of the face surface are taken into account. On the comparably rough grid over the face that the Kinect offers, the nose is the only area, that has a high curvature but a comparably low gradient[10].

In [13] as measures for the curvature the Gaussian and the Mean curvature were applied. Both of them are rotation invariant measures, which was important for the problem specification of [13]. In this application to detect a pixel belonging to a nose the second derivative in horizontal direction f_{xx} is sufficient. This is exactly the direction in which a strait nose is curved.

When the control program has to do a face detection after a “Click” has been detected, at first the region of interest has to be found. This is done by the means of the OpenNI user generator. In the area of the depth picture where the “Click” event happened, a

[10]Other areas in the depth picture of a face that have high curvature values are areas at the edge of the face. But all these areas have much higher gradients than the nose.

pixel with a scene label different from zero (meaning background) is searched for. The first found scene label different from zero is used as userID. Then the region of interest is chosen as a rectangle, that includes all pixels of the depth picture that are labelled with the userID and that are not farther than 30 cm under the highest pixel labelled with this userID. This implies, the head has to be the topmost part of the body. All following operations are only done in this region of interest.

The depth values then are smoothed by a 7x7-moving window filter and the first derivatives in x- and y-direction (f_x, f_y) and the second derivative in x-direction (f_{xx}) are calculated with the central differential quotient. Without smoothing the derivatives would be too corrupted by noise.

For every pixel in the region of interest the function "faceDetection(•)" then evaluates, whether it is a "nose pixel". For every nose pixel the following must hold:

- $|f_x| < 0.6$
- $|f_y| < 0.5$
- $f_{xx} > 0.1$

The threshold values were adjusted by evaluating their effect on single non-thresholded arrays of f_x, f_y and f_{xx}. Two examples for the relation between f_x, f_y and f_{xx} and the distribution of nose pixels can be seen in Figures 3.6 and 3.7.

Finally the function "faceDetection(•)" tries to find a rectangle with the following characteristics:

- number of pixels = floor $\left(\frac{60 \cdot 1630^2}{\text{depth of face in mm}}\right)$
- $\frac{\text{horizontal length}}{\text{vertical length}} \leq 0.4$
- horizontal length in pixels ≥ 2
- "nosity" $= \frac{\text{number of nose pixels}}{\text{number of pixels}} \geq 0.34$

These characteristics are supposed to uniquely describe a strait nose. The first item uses a relation between different depths and the measured nose area 60 pixels for a depth of 1630 mm. The second item describes that a nose is much longer than wide. If the function "faceDetection(•)" finds a rectangle fulfilling all these characteristics, this rectangle is considered a strait nose and the function evaluates true. The user is granted control over the application. If it cannot find such a rectangle, it returns false. The "Click"-event stays disregarded.

Figures 3.6 and 3.7 display the derivatives and the nose pixels of the region of interest. In Figure 3.6 there is a face looking at the Kinect, its nose is correctly detected. In Figure 3.7 there is a face too, but it is looking at a point slightly right of the Kinect. Even though in this case there is about the same amount of nose pixels, these nose pixels cannot be abstracted to a rectangle fulfilling the aforementioned characteristics.

In experiment 15 times this face detection was performed with a strait face looking at a Kinect and 15 times with a face present but not looking at the Kinect. Five times a false negative occurred, i.e. the strait face looking at the Kinect was not detected. Two times a false positive occurred, i.e. the face in vision that was not looking at the Kinect was falsely detected as looking at the Kinect. This experimental data is not representative for two reasons, firstly the amount of 15 test runs for each situation is not large enough, secondly for all test runs I used my own face that had been already used for choosing the threshold values for the nose pixels, too. So training data and test data are not independent in this case. In general the performance of this face detection application probably is a little worse. If these results are used anyway, the resulting probability of a failure on demand (PFD) of this face detection is:

$$\text{PFD} = \frac{\text{number false positives}}{\text{number of ``Clicks'' without a strait face}} = \frac{2}{15} = 0.1333 \tag{3.24}$$

which means that the risk of receiving control unwillingly with this face detection is only 13.33% of the risk without the face detection. With more training data and an automated learning method that choses appropriate threshold values the performance of this face detection probably could be further increased.

3.5. Shadowing the user's hand

One objective of this application was to enable the robot to follow the user's hand. This means the robot should imitate the translations of the user's hand in a reasonable manner and copy the orientation of the user's lower arm. The experiments for this section were conducted with the FRIDA because of the most human-like proportions of this robot type.

As written in Section 2.2 the OpenNI user generator can determine the position of the user's hands and shoulders. Additionally it can calculate the rotation matrix of the user's elbows. This is applied for this objective. A hand transformation matrix as seen from the Kinect is defined for the left hand containing the rotation matrix of the user's left elbow and as translation the vector from the user's left shoulder to his left hand. This homogeneous transformation matrix is sent to the Simulink controller via a LabComm connection.

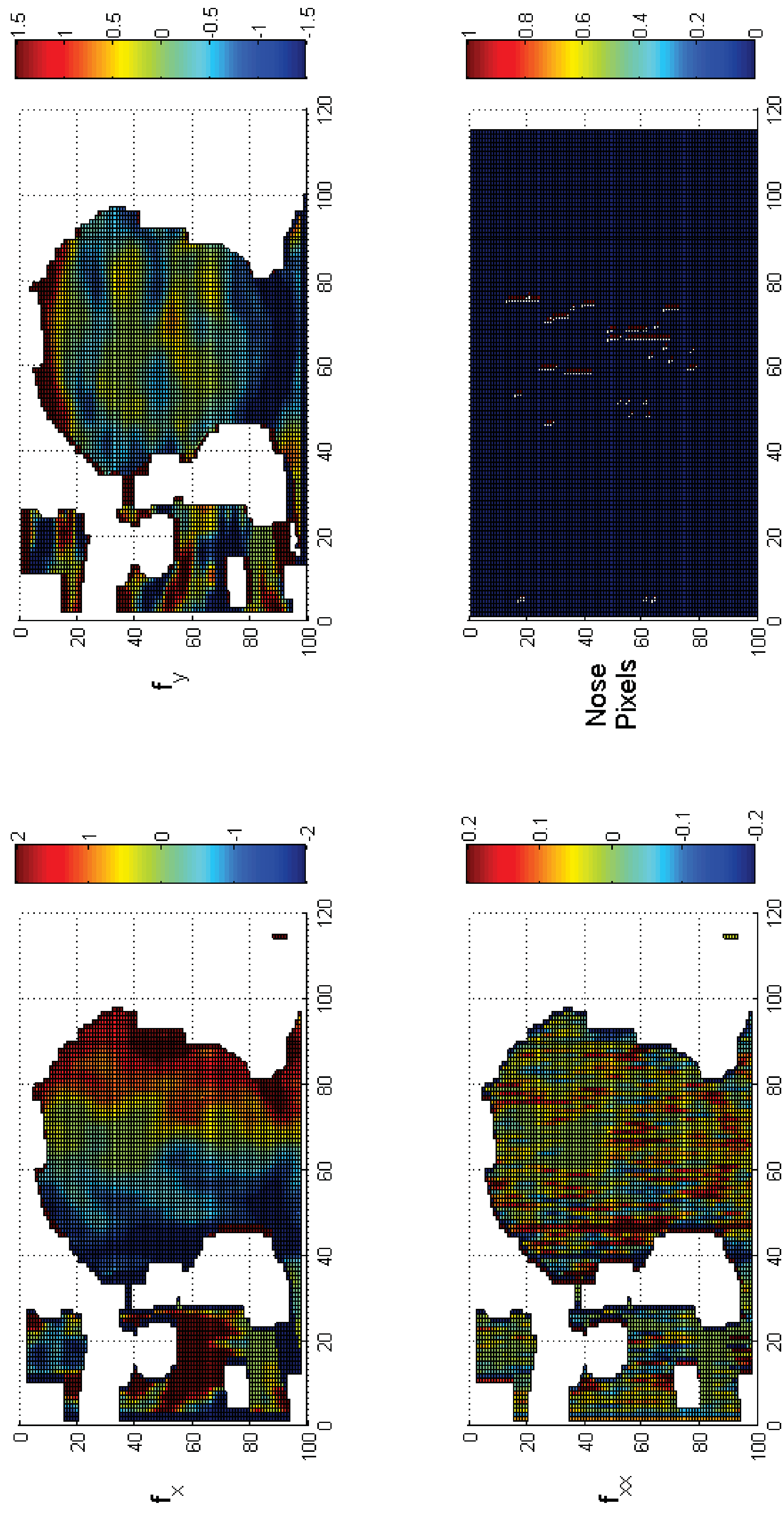


Figure 3.6.: Derivatives and nose pixels of a strait face

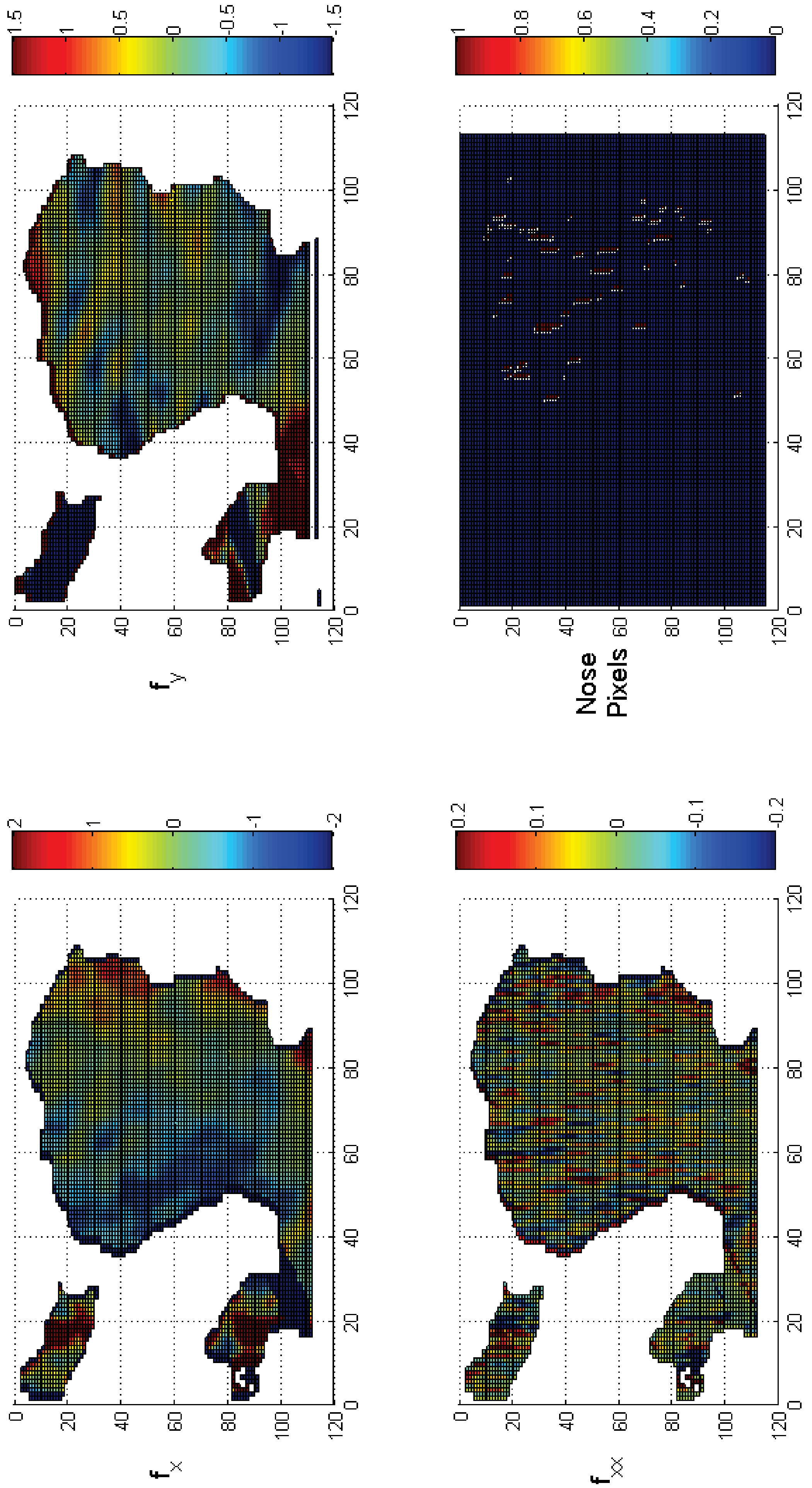


Figure 3.7.: Derivatives and nose pixels of a non-strait face

The origin of the base frame of FRIDA usually is set in the middle of the bottom end of the front torso. To match the user's shoulder, for this application it was chosen to be located at the shoulder of FRIDA, cf Figure 3.8. Under these conditions the tasks of the Simulink controller are to transform the homogeneous transformation matrix sent by the main computer from the Kinect coordinate frame to the robot base frame and to deduce the necessary joint position and velocity reference values for the axis computer.

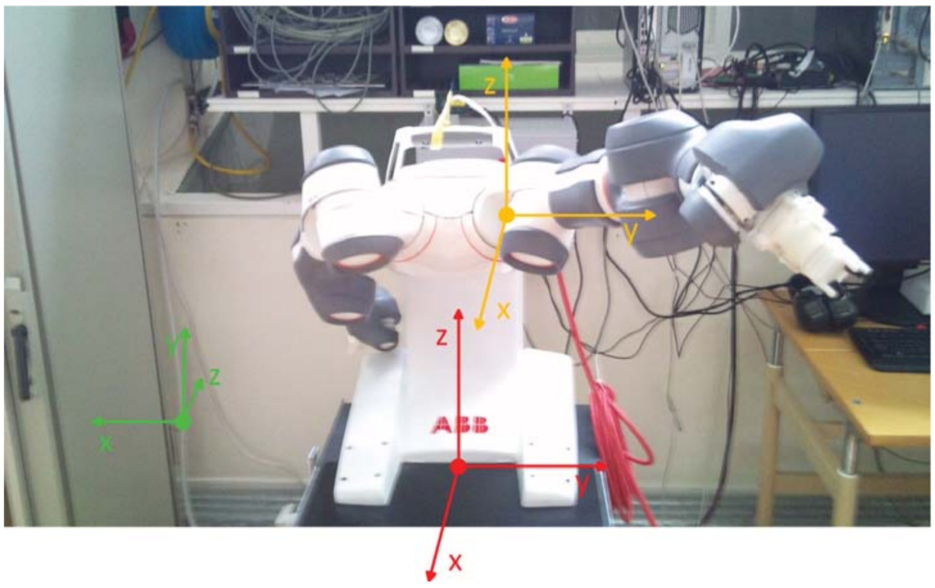


Figure 3.8.: Relevant Coordinate Frames:
red: usual robot base frame
orange: robot base frame originating n the left shoulder of FRIDA
green: Kinect coordinate frame (does not originate there)

The different coordinate frames of this experiment can be seen in Figure 3.8. The Kinect in this experiment has a similar view point of the scene as the one, from which Figure 3.8 is taken. The user is supposed to be standing next to FRIDA facing the Kinect. Therefore the necessary transformation from the Kinect frame to the base frame is:

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.25}$$

which is premultiplied to the transformation matrices expressed in Kinect coordinate system that the main computer sends to the Simulink controller. The matrix **T** was found by expressing the Kinect coordinate axes in robot base frame coordinates. In this transformation the translation between the Kinect frame and the robot base frame is neglected, because the translatory position of the users's hand is expressed relatively to the user's shoulder anyway. So an absolute position information is not necessary. This makes this assembly very flexible. Neither it is important where the Kinect is positioned relative to the robot, nor where the user stands relative to the Kinect. Only the orientations matter. The result of this transformation is the homogeneous transformation matrix $\mathbf{H}_{hand}$ expressed in the robot base frame that the robot flange should adapt. So if the inverse kinematics that transform this matrix into the respective joint values were available the problem would already be solved. But for the FRIDA only the Jacobian was given.

To deduce the necessary joint angles using the Jacobian of the FRIDA a position feedback loop was introduced, compare Figure 3.9. The homogeneous transformation matrix of the actual flange position $\mathbf{H}_{flange}$ is calculated from the actual joint position values $\mathbf{q}_{act}$ of the robot using the forward kinematics. From both matrices $\mathbf{H}_{hand}$ and $\mathbf{H}_{flange}$ the included actual translation and rotation is deduced. Then the difference between these multiplied by a proportional gain is regarded as the linear and angular velocity references $\mathbf{v}_{ref}$ and ω_{ref}. As written in Section 2.3 the inverse velocity kinematics - in this case this is the inverse Jacobian - transform these velocities in cartesian space into joint velocities. Therefore by the inverse Jacobian $\mathbf{J^{-1}}(\mathbf{q}_{act})$ the joint velocity reference values $\dot{\mathbf{q}}_{ref}$ can be calculated. Integrating these over time results in the joint position reference values $\mathbf{q}_{ref}$.

This was only a rough outline of the Simulink controller model. Several aspects have to be examined a little closer:

3.5.1. Difference of two Rotations

To compare the actual and the wished for translation and orientation of the robot flange - as in every feedback control loop -, the difference is calculated. However, while the difference between two translations, both represented as a three dimensional vector, of course is the three dimensional vector containing the scalar differences of each coordinate, the meaning of the difference between two rotations is not the mathematical difference of the two corresponding rotation matrices.

Firstly, in matrix representation a rotation is a multiplicative operation. This means if two different rotations are performed, the final rotation is described as the product of the two primary rotation matrices, not as the sum as it would be for two translations. Hence, to gain the difference between two rotations the quotient has to be taken, in matrix representation this means a multiplication with the inverse of the divisor matrix. The resultant rotation matrix in this case is supposed to describe the rotation that is left to turn the actual flange frame into the orientation of the hand. This corresponds to a postmultiplication of the rotation matrix $\mathbf{R}_{hand}$ included in the homogeneous transformation matrix $\mathbf{H}_{hand}$ by the inverse of the actual flange rotation matrix $\mathbf{R}_{flange}$ included in $\mathbf{H}_{flange}$:

$$\mathbf{R}_{diff} = \mathbf{R}_{hand} \cdot \mathbf{R}_{flange}^{-1} \tag{3.26}$$

Secondly, as the description of the cartesian angular velocity ω needed as input for the inverse Jacobian is a three dimensional vector, an equivalent vector description of this difference rotation matrix $\mathbf{R}_{diff}$ has to be found. The norm of the vector ω describes the velocity of a rotation in a plane to which ω is normal. So this plane rotates around the vector ω. This corresponds to the definition of the rotation vector of the axis/angle representation introduced in Subsection 3.2.3, cf. Figure 3.3. Therefore expressing the rotation $\mathbf{R}_{diff}$ by its rotation vector in axis/angle representation yields the wished for structure of the angular position difference.

The formulas to calculate the rotation vector from a rotation matrix are given in [8]:

$$\theta = \arccos\left(\frac{\operatorname{trace}(\mathbf{R}) - 1}{2}\right) \tag{3.27}$$

$$\mathbf{k}_{rot} = \frac{\theta}{2\sin(\theta)} \cdot \begin{pmatrix} R_{3,2} - R_{2,3} \\ R_{1,3} - R_{3,1} \\ R_{2,1} - R_{1,2} \end{pmatrix} \tag{3.28}$$

where the same structure of the rotation matrix $\mathbf{R}$ is assumed as in Equation 3.10.

When running this application, due to numerical inaccuracies there is the danger that the argument of the arccos in Equation (3.27) is slightly outside the domain of the arccos, $[-1, 1]$. This would lead to a NaN result that could crash the whole application. So that argument has to be explicitly limited to the interval $[-1, 1]$. The range of the arccos is the interval $[0, \pi]$. $\theta = \pi$ would be a singularity of Equation (3.28). So to avoid too high values of the elements of the rotation vector, θ has to be limited, too. In the control program the limit is 0.936π. This implies that the coefficient of the vector in Equation 3.28 cannot get higher than 8.4.

$\theta = 0$ is a removable discontinuity of Equation (3.28). In fact $\theta = 0$ - corresponding to the rotation matrix being the identity matrix - is the goal of this control, it means no further rotation necessary any more. The frames of the hand and the robot flange are aligned. So it is reasonable to set $\mathbf{k}_{rot} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ in this case. This also is the limes of Equation (3.28) for $\theta \to 0$.

Multiplying the vector $\mathbf{k}_{rot}$ by an appropriate gain finally yields the angular velocity reference ω_{ref}.

3.5.2. Pseudoinverse of the Jacobian

The Jacobian as defined in Equation (2.20) is a $(6 \times n)$ matrix, where n is the number of DOF of the respective robot, for the FRIDA $n = 7$. Therefore the Jacobian of FRIDA is not a square matrix and cannot be inverted regularly. Instead the right pseudoinverse of the Jacobian is used. The right pseudoinverse $\mathbf{J}^+$ of the Jacobian $\mathbf{J}$ is the $(n \times 6)$ matrix for which the following equation holds:

$$\mathbf{J} \cdot \mathbf{J}^+ = \mathbf{I} \tag{3.29}$$

$$\mathbf{J}^+ = \mathbf{J}^T \left(\mathbf{J}\mathbf{J}^T\right)^{-1} \tag{3.30}$$

In this application the pseudoinverse was calculated by Singular Value Decomposition.

3.5.3. Avoiding Singularities at the Edges of the Workspace

As written in Section 2.3, singularities of the robot workspace are a huge problem when dealing with inverse velocity kinematics. Singularities can occur inside the robot workspace and they are always at all edges of the robot workspace. Detecting and working around singularities inside the workspace requires deep analysis of the inverse velocity kinematics. This was not done as part of this thesis. But the singularities at the edges of the robot workspace can be avoided much easier by preventing all robot joints from reaching their extremal position values.

The FRIDA is a 7-DOF robot. This means after fulfilling the constraints of the linear and angular velocity references, each needing three DOF, there is still one DOF left in joint space. This can be used to decrease the absolute joint values and therefore move the joints away from the edges of their range.

Given the right pseudoinverse $\mathbf{J}^+$ of the Jacobian, all possible solutions of the inverse velocity problem are:

$$\dot{\mathbf{q}} = \mathbf{J}^+ \begin{pmatrix} \mathbf{v} \\ \omega \end{pmatrix} + \left(\mathbf{I} - \mathbf{J}^+\mathbf{J}\right)\mathbf{b} \tag{3.31}$$

where $\mathbf{b} \in \mathbb{R}^n$ is an arbitrary vector. This can be regarded as the solution to the minimizing problem

$$\min_{\dot{\mathbf{q}}} \dot{\mathbf{q}}^T \mathbf{R} \dot{\mathbf{q}} \tag{3.32}$$

subject to the condition

$$\begin{pmatrix} \mathbf{v} \\ \omega \end{pmatrix} = \mathbf{J}\dot{\mathbf{q}} \tag{3.33}$$

where $\mathbf{R}$ is a positive semidefinite weighting matrix. For example the solution with $\mathbf{b} = \mathbf{0}$ in Equation (3.31) corresponds to $\mathbf{R} = \mathbf{I}$ in Equation 3.32. That would be the solution with minimal joint velocity. As $\mathbf{b}$ is arbitrary, it is possible to use a vector depending on the actual joint position.

$$\mathbf{b} = \mathbf{b}(\mathbf{q}) \tag{3.34}$$

In this thesis I used:

$$\mathbf{b} = 2 \cdot \mathrm{sign}\left(\frac{\mathbf{q}_{mid} - \mathbf{q}}{\mathbf{q}_{span}}\right) \cdot \left(\frac{\mathbf{q}_{mid} - \mathbf{q}}{\mathbf{q}_{span}}\right)^2 \tag{3.35}$$

where $\mathbf{q}_{mid}$ is the vector containing all joint middle positions and $\mathbf{q}_{span}$ containing the full range of all joints. Additionally, I multiplied the elements of $\mathbf{b}$ for the joints 1, 2, 5 and 6 with a higher gain, because in experiment these joints tended most to reaching the edge of their range.

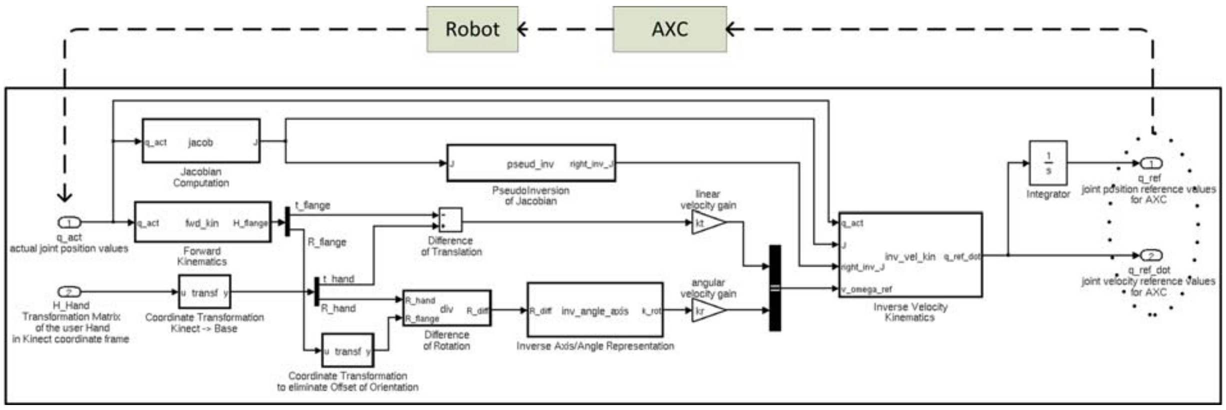
With $\mathbf{b}$ chosen as in Equation (3.35), Equation (3.31) can be regarded as the solution to a minimizing problem as in Equation (3.32) with a variable weighting matrix $\mathbf{R}$ depending on the actual joint position $\mathbf{q}$. The joints that are relatively farthest away from their middle point are punished hardest.

This solution leads to the tendency of the robot controller to decrease the absolute joint positions, because the expression $\mathbf{J}^+\mathbf{J}$ usually describes a kind of "watery" weakened identity matrix. So the diagonal elements of this expression usually are slightly smaller then one and the non-diagonal elements slightly different from zero. This means the diagonal elements of $\mathbf{I}-\mathbf{J}^+\mathbf{J}$ are usually positive. So if the absolute value of the respective element of the vector $\mathbf{b}$ is relatively high, this leads to the second summand of the Equation (3.31) holding a different sign in the respective coordinate than $\mathbf{q}$.

This way of calculating the inverse velocity kinematics showed good performance in experiment. To protect the robot from extremal joint positions, additional saturation blocks where introduced into the Simulink controller with saturation limits of 98% of the respective joint range. However, these saturation blocks had to get active only very rarely in experiment.

3.5.4. Additional Tool Rotation

The described control for the left hand was implemented and tried out in experiment. It showed, that the position of the human hand was copied correctly by the robot, but the orientation needed one further rotation. The flange of the robot arm had to be rotated around the y-axis of the robot flange frame by 90 degrees. This is most likely caused by a rigid offset of the elbow coordinate frame of the user that the OpenNI user generator adds to the actual Kinect coordinate frame.



Robot
AXC
q_act
jacob
J
Jacobian Computation
J
pseud_inv
right_inv_J
PseudoInversion of Jacobian
1
q_act
actual joint position values
q_act
fwd_kin
H_flange
Forward Kinematics
t_flange
R_flange
Difference of Translation
linear velocity gain
kt
2
H_Hand
Transformation Matrix of the user Hand in Kinect coordinate frame
u transf y
Coordinate Transformation Kinect -> Base
t_hand
R_hand
R_hand
div
R_diff
R_flange
Difference of Rotation
R_diff
inv_angle_axis
k_rot
Inverse Axis/Angle Representation
angular velocity gain
kr
u transf y
Coordinate Transformation to eliminate Offset of Orientation
q_act
J
right_inv_J
v_omega_ref
inv_vel_kin
q_ref_dot
Inverse Velocity Kinematics
1/s
Integrator
1
q_ref
joint position reference values for AXC
2
q_ref_dot
joint velocity reference values for AXC

Figure 3.9.: Structure of Simulink controller and feedback loop, even though this schematic is strongly oriented on the Simulink model for reasons of simplicity many components are omitted

3.5.5. Chirp

In initial state the robot flange has a random pose. The joint position references for this pose are saved throughout the whole program run. After the “Shadowing”-mode is finished, the robot should adopt this pose again. At both these switching points the user’s arm probably is in a different pose than the robot. So by just switching between theses poses jumps of the joint position references could be caused. To avoid this a four second “chirp”-phase is introduced between these two states “Shadowing” and “Not-Shadowing”, that makes the transition smoother.

The chirp-block is located at the very end of the Simulink model, it selects which set of joint position references is sent to the axis controller computer, the saved set that is the initial pose, the calculated set that describes the pose of the user's hand or a mixture of both. During chirp-phase the transition from one set of references to the other one is conducted along a sine function in the interval $[-\pi/2, \pi/2]$. The sine was chosen as basis function for this chirp-movement, because it offers a transition from one point to another in a finite span of time with a derivative of zero in both end points. The output during this phase is calculated by the following algorithm:

$$\mathrm{arg} = \left(\frac{t - t_0}{T_{chirp}} - \frac{1}{2}\right) \cdot \pi \tag{3.36}$$

$$\mathrm{val} = \frac{\sin(\mathrm{arg}) + 1}{2} \tag{3.37}$$

$$\mathbf{q}_{ref} = \mathbf{q}_1 + \mathrm{val} \cdot (\mathbf{q}_2 - \mathbf{q}_1) \tag{3.38}$$

where t_0 is the time point, when the chirp-phase began and T_{chirp} is the time this phase takes on the whole. Then this algorithm describes a chirp from input $\mathbf{q}_1$ to input $\mathbf{q}_2$. This does not necessarily mean all joints perform a sinusoidal movement during this phase, because the calculated values describing the user's hand may change in the meantime. However, as the derivative of the sine for the arguments $-\pi/2$ and $\pi/2$ is zero, the output joint position reference is differentiable in both the starting point and the ending point of this chirp-phase. Hence, smooth transitions between "Shadowing" and "Not-Shadowing" of the robot are achieved.

3.5.6. Shadowing two Arms

This control developed for the left arm of human and robot can easily be adjusted to the right arm too. The only two differences that have to be made, are to send the homogeneous transformation matrix of the right user hand to the Simulink controller instead of the left one and to choose the right shoulder of FRIDA as the origin of the robot base frame.

Then both robot arms can even be controlled in parallel. The main computer has to send the transformation matrix of both hands to the Simulink controller that treats the two arms as individual robots with their respective base frame.

For safety reasons in this two-arm control an additional confidence control was introduced, because it may not always be possible, that the user can show both his arms in front of the Kinect. If the OpenNI user generator cannot determine the position or orientation of a tracked user joint, it still outputs the most likely guess but also indicates the unreliability of this data by setting a confidence variable to zero that is adjoined to the data of any joint position and orientation. Usually this variable would be one.

This information is passed on to the Simulink controller using the redundancy of a homogeneous transformation matrix. The last line of a transformation matrix is always $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$. So by changing this line further information can be sent. In this control a “-1” in the first column of the last line indicates corrupted data of the translation vector, in the second column it indicates corrupted data of the rotation matrix. If this symbol is detected during the program run by the Simulink controller, the respective velocity reference - linear, angular or both - of the respective robot arm is set to zero, so the robot stays in the last confidently known position of this motion mode. After the confidence is high again, for the next two seconds the gain of this velocity is lowered to a third of its usual value. That is done to protect the robot from too sudden movement, because the respective position of the user arm might have changed a lot during the time it could not be detected properly, which would lead to high velocity references.

4. Results

Most of the results have already been presented in Subsection 2.1.2 and Chapter 3. One further experiment conducted during this thesis is described here.

4.1. Dynamic Behaviour

The dynamic behaviour of this application is decided mainly by three aspects: the loss of time during detection by the Kinect, the time needed for calculation by the computers and the mechanical inertia of the robot.

The time needed by the computers depends strongly on the respective computational load and cannot be determined generally.

To measure the influence of the two other aspects an experiment was conducted: On the ABB robot IRB140B an artificial hand was mounted. This hand was tracked by the Kinect using the OpenNI hand generator. This setup can be seen in Figure 4.1. Then the robot performed a sinusoidal movement with the amplitude 50 mm and stepwise rising frequencies between 0.1 Hz and 3.5 Hz. The tracked hand positions in 3D-real world coordinates were sent to the Simulink controller computer via a LabComm connection, where they were saved together with the robot data.

The single components in this experiment are composed in a different order than in the actual application. In the application the Kinect detects a movement, the computers perform the necessary calculations, and as a result the robot moves in an appropriate manner, cf. Figure 4.2. In this experiment the robot moves following a sinusoidal signal of the Simulink controller. This movement is seen by the Kinect and the calculated coordinates are sent back to the Simulink controller, cf. Figure 4.3.

Assuming linear transfer functions of all involved blocks this different order in this experiment will lead to the same dynamic behaviour as the order of the actual application. This is a result of the rule that linear blocks in block algebra are commutable.

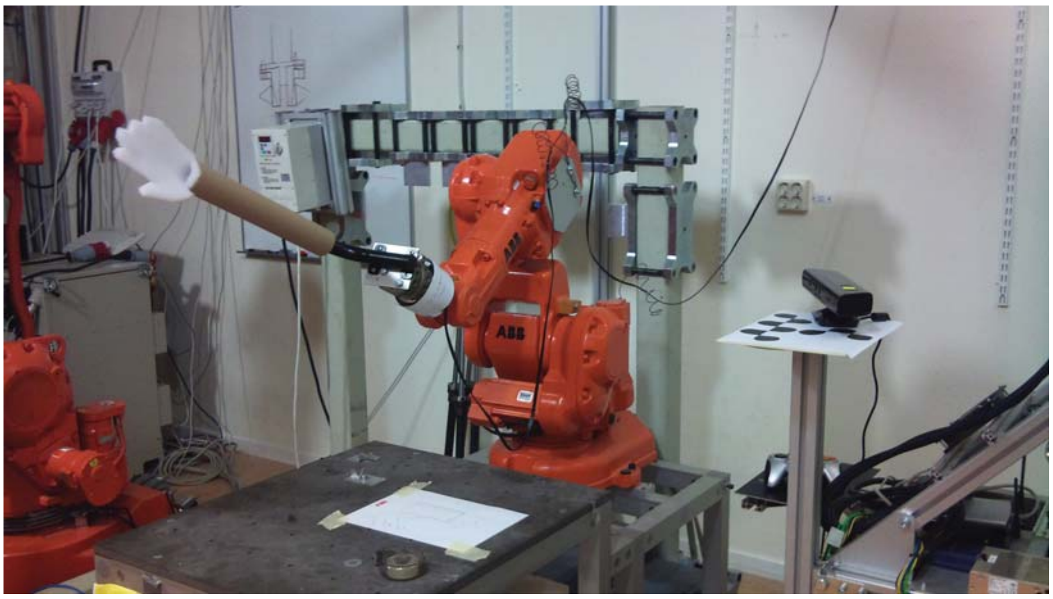


Figure 4.1.: Experiment setup: ABB IRB140B with mounted hand and Kinect

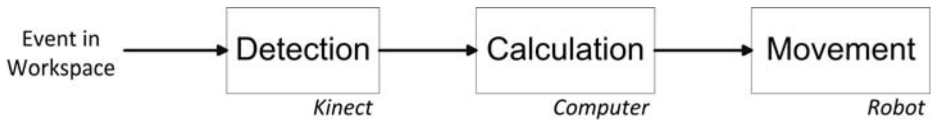


Figure 4.2.: Order of components in actual application

The computational requirements of the program ”Handrecorder”, that extracted the position of the artificial hand out of the Kinect data, are not very severe.[1] Therefore the time needed by this program is accounted to the time, the Kinect needs for detection. The third block calculation by the computers is not supposed to take any time in this experiment. This allocation is reasonable, because any other application at first needs to extract the important information of the Kinect data, too. Therefore this extraction belongs to the detection by the Kinect just like the calculations that actually happen inside the Kinect.

[1]The code of this small program also can be found in the appendix.

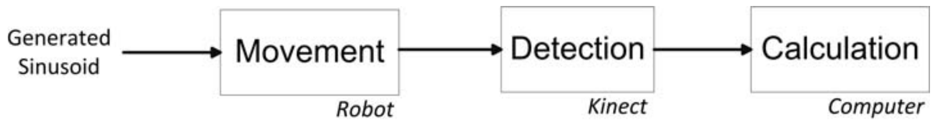


Figure 4.3.: Order of components in this experiment

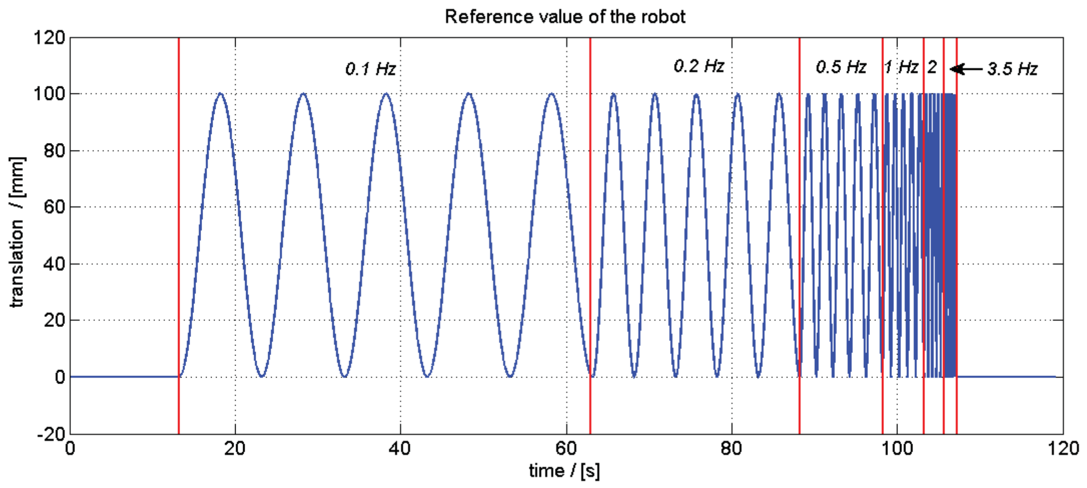


Figure 4.4.: Signal for position reference of the robot

For the direction of the sinusoidal movement the vertical direction, that is the y-coordinate in Kinect space, was chosen, because this direction could be discriminated best from the other ones. The measurement itself is not affected by this decision, because as written in Section 2.1 for a certain pixel all coordinates depend linearly on the depth value, the z-coordinate, of this pixel.

The function of the sinusoidal signal was

$$s(t) = 50\ mm \cdot (1 - \cos(2\pi \cdot f_{Hz} \cdot t)), \tag{4.1}$$

where for f_{Hz} in rising order the values $f_{Hz} = [0.1, 0.2, 0.5, 1, 2, 3.5]$ were inserted, each one for five periods. This signal can be seen in Figure 4.4. Frequencies higher than 3.5 Hz could not be tested because the robot could not move any faster in the used extctrl mode.

The cosine was chosen to avoid jumps of the robot velocity. The change of the frequency happened always after five full periods. For the sine this is the moment with the highest velocity. Therefore the jump in velocity would be the maximum possible jump, too. The derivative of the cosine is zero after a full period. So the frequency can be changed without any discontinuity. To avoid a jump of the position at the very first period the bracketed expression $(1 - \cos(\bullet))$ is necessary.

Three signals were logged: the generated sinusoid that was sent to the robot as position reference, the actual position value of the robot and the actual position value of the hand detected by the Kinect[2]. In these signals the offsets were eliminated and the measured

[2]The robot was performing a solely linear movement, so the relative movement of the robot flange and of the rigidly mounted hand were the same.

frequency / [Hz]	Delay in Robot / [s]	Delay in Kinect / [s]	total Delay / [s]
0.1	0.012	0.048	0.060
0.2	0.012	0.046	0.058
0.5	0.012	0.041	0.053
1	0.012	0.042	0.054
2	0.012	0.055	0.067
3.5	0.013	0.068	0.081

Table 4.1.: Delays in robot and Kinect

signal of the Kinect was additionally smoothed by a moving average filter regarding always five samples. The resulting signals are plotted in Figure 4.5.

In all distinct points, the extrema and the zero crossings, the time differences between these curves were measured. All measurements are listed in the appendix. Table 4.1 shows the delays of the robot and of the Kinect measurement for each frequency. These delays are also plotted in Figure 4.6.

Looking at Figure 4.5 and at the delay times, it is reasonable to model both the robot and the Kinect as mere dead time elements. At no frequency a difference between the amplitude of the signals of the position reference and the actual position of the robot could be observed and the time delay of the robot was very constantly over the frequencies 12 ms. Usually a robot is modelled as a PT_N-element of a high order, due to its various masses that have to be accelerated. However a PT_N-element would imply different amplitude gains for different frequencies. So at least for the relatively slow velocities at the frequencies of this experiment a dead time element is the better choice. So the transfer function of the robot is:

$$G_{Robot}(s) = e^{-s \cdot 0.012} \tag{4.2}$$

The amplitude of the detected signal by the Kinect rises a little for frequencies higher or equal to 1 Hz. This effect is probably due to the inertia of the artificial hand and the compliance of the arm of the artificial hand. For higher frequencies this arm might bent a little and therefore amplify the movement of the robot flange. There is actually no theoretical motivation to model the Kinect as a PT_N-element, because PT_N-elements describe energy storage elements, while the Kinect only deals with information. Hence, regarding that the different time delays for the different experiment frequencies do not differ too greatly (29% between the largest and the smallest delay), it seems permissible to describe the Kinect as a dead time element with the average delay time 50 ms.

$$G_{Kinect}(s) = e^{-s \cdot 0.050} \tag{4.3}$$

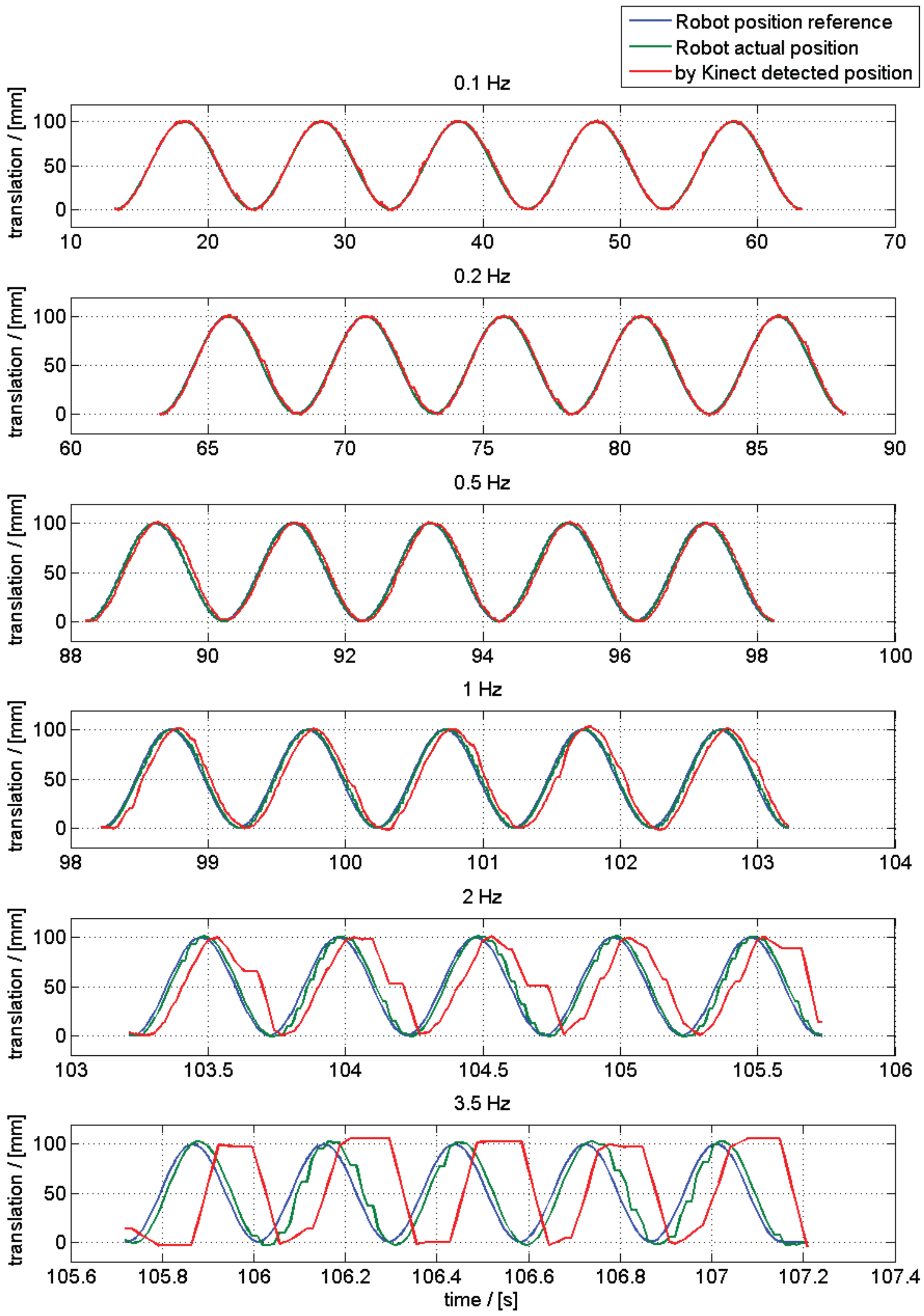


Figure 4.5.: Measured signals of the experiment

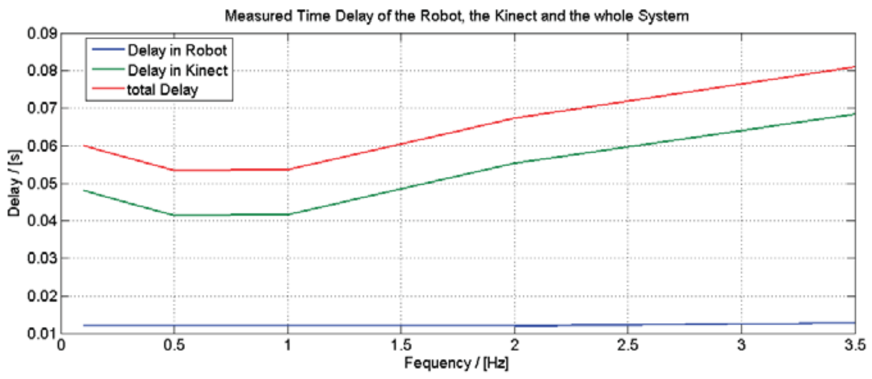


Figure 4.6.: Delays in robot and Kinect

This means the whole system is a time delay element, too. Its modelled delay time is 62 ms.

$$G_{whole}(s) = e^{-s \cdot 0.062} \tag{4.4}$$

Figure 4.7 shows the Bode diagram of the robot, the Kinect and the whole system. It can be seen that for small frequencies the whole system is almost a constant with the gain 1 and that for frequencies up to about the 3.5 Hz of the experiment the phase delay stays small (ca. 15° in this Bode diagram, 16.2°, if the measured time delay for 3.5 Hz is used). For higher frequencies dead time elements quickly loose phase. Additionally in this case the course of the delay times indicates a further rise of the delay times for higher frequencies[3]. This would lead to an even quicker fall of the phase. So this system should not be used for much higher frequencies than the frequencies of this experiment.

Another observation of the signals in Figure 4.5 is that for high frequencies the signal of the detected hand position by the Kinect more and more looses its sinusoidal form. For $f = 3.5$ Hz it looks strongly sampled with a sampling step of around 0.066 s. This is about the time of two frames at a frame rate of 30 frames/s. This might indicate, that the hand generator needs two depth frames from the Kinect, to detect a new hand point. If that is true, this would limit the possible frequency range to about the frequencies of this experiment. Following Shannon's Sampling Theorem the theoretically highest frequency in this case would be 7.5 Hz.

[3]This is also the reason, why the Bode diagram in Figure 4.7 was not drawn any further. The validity of predictions for the phase delays of higher frequencies would be very limited

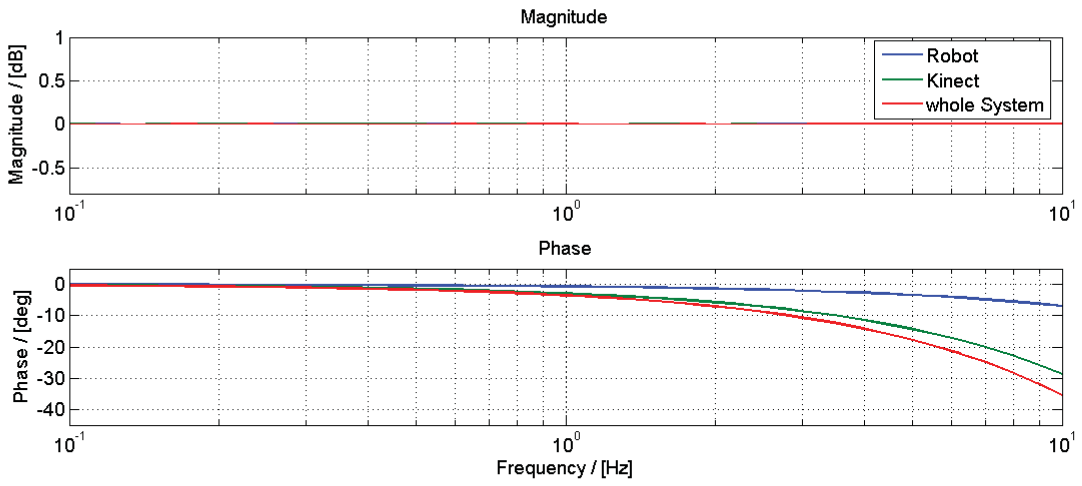


Figure 4.7.: Bode diagram of robot, Kinect and the whole system

5. Discussion

In this chapter an error analysis is presented and the extent to which this Kinect application fulfills the expectations of the ROSETTA project is evaluated.

5.1. Error Analysis

Possible error sources are listed in this section.

Numerical errors: The calculations for the calibration of the two Kinect devices are performed in `double` accuracy. All other computation are done in `float` accuracy, to spare computation power.

If the angular difference between the user's and the robot's arm in the shadowing application is less than 10^{-6} [rad], it is set to zero, to avoid the lack of definition of the inverse axis/angle transformation for an angular difference of zero. If it is higher than 0.936π [rad] it is set to 0.936π [rad], to exclude too high angular velocity reference values near the discontinuity of the inverse axis/angle transformation at an angular difference of π.

Statistical errors: The depth picture of the Kinect is noisy. Inside a steady plane the depth value may jump between the two adjoining depth quantization steps. Their distance depends on the actual depth, cf. Figure 2.6. At the edges of steady objects in vision the depth values can vary strongly over time. They can be accounted to the object in the foreground (low depth value), to the background (high depth value) or not be detected at all (depth value zero). These errors due to noisy measurement by the Kinect in most cases vary between two frames.

Max-, Min- errors: The joint position reference values of the Simulink controller are limited in both directions to 98% of the range of the respective joints of the robot, to prevent the robot from out-of-range joint position references. If a limitation gets active, the user's position or orientation cannot be shadowed completely accurately any more. This state is observed and outputted by the Simulink controller. So the user should vary his arm position shortly and, if possible, move into the first position again following a different trajectory.

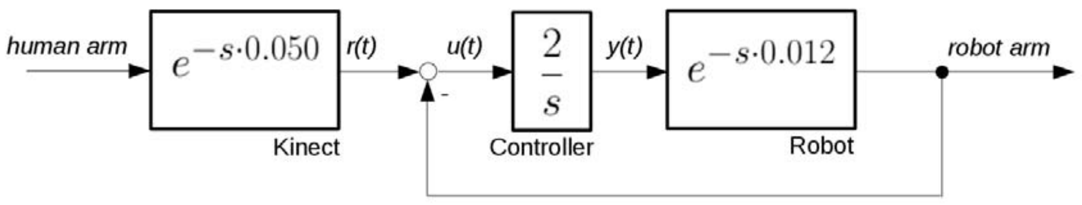


Figure 5.1.: Simplified schematic of the control of the shadowing application

Accuracy: The accuracy of the depth measurement by the Kinect depends on the horizontal angle of each respective pixel and the absolute depth, cf. Subsection 2.1.2.

Dynamic lagging: Due to the feedback loop in the Simulink controller in the shadowing application further delays apart from the delay elements arise. With a velocity gain of 2, which was used most of the time in the experiments, the Simulink controller contributes together with the robot and the Kinect sensor to an I-controlled feedback loop, cf. Figure 5.1.

$$G_{Shadowing} = e^{-s\cdot 0.050} \cdot \frac{2 \cdot e^{-s\cdot 0.012}}{s + 2 \cdot e^{-s\cdot 0.012}} \tag{5.1}$$

Neglecting the inner delays this feedback loop is a PT$_1$-element with a time constant of 0.5 seconds. This means the position error asymptotically tends to zero. The step response is not regarded here, because in the control program any jumps are avoided, to protect the robot. Instead in Figure 5.2 the ramp response is displayed that shows the practically relevant ramp error of 0.5 mm in response to a ramp of 1 mm/sec. This consideration is a strong simplification, as it disregards the inner delays shown in Figure 5.1 and Equation (5.1), the differences between angular and linear velocity and the fact that this system is a MIMO-system and not a SISO system. The resultant ramp error is valid nevertheless.

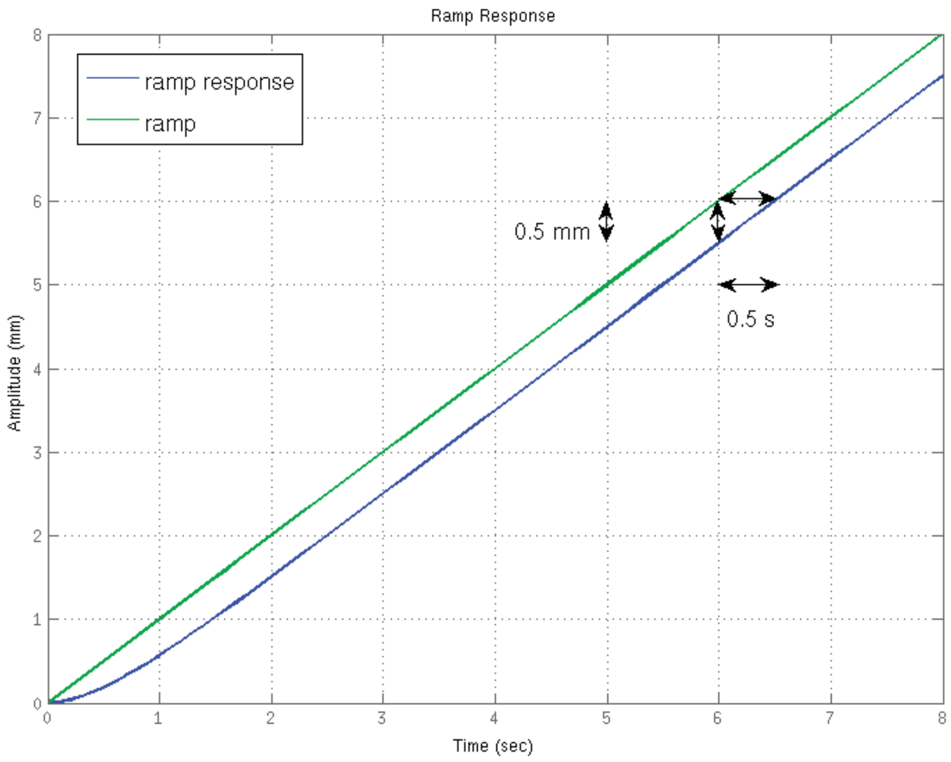


Figure 5.2.: Ramp response of the shadowing application

5.2. Conclusion

As written in Section 1.2 out of the postulated objectives of the ROSETTA project mainly the following three ones are relevant for this thesis:

1. The interaction has to be intuitive and perceived as natural.
2. The robots must be safe.
3. The application has to be flexible.

Concerning 1: Controlling a robot by showing it the exact same movement that the robot just has to repeat is the most intuitive form of executing control possible. It is comparable to the way, how children learn very basic forms of behavior by imitating their parents. Therefore, this form of control must be perceived as natural by the human operator, too.

Manoeuvring the robot arm with one's own arm feels very direct and accurate, as the effect of one's action can be directly observed. If the robot controlled by the shadowing application has to reach a certain position in its range, the overall feedback loop includes the operator's vision as sensor, brain as controller and arm as actuator in the exact same way as real human movement would do.

This first goal of the ROSETTA project is achieved very well by this Kinect application.

Concerning 2: As presented in Section 3.3, the Kinect can be used for workspace surveillance, which is an intuitive and flexible form of a safety application.

While todays safety measurements are usually a trait-off to efficiency - for example a dead-man switch of a robot requires constantly one hand of the user - this workspace surveillance runs latently in the background and only visibly acts if it has to stop or slow down a robot.

To guarantee full functionality when required, safety systems usually are composed as simply as possible. They have to be programmed very robustly and the computers they are running on have to be capable of these programs in real time. This means, if the Kinect which is computationally highly demanding is integrated into a safety application, the architecture of hardware and software has to be chosen and maintained most carefully.

Concerning 3: The only experiment, for which some measurements had to be accomplished in advance, is the workspace surveillance. But in Section 3.3 an alternative way was presented to find the transformation between the robot base frame and the Kinect frame, based on calibration.

So all necessary transformations can be found by calibration, additionally translations were defined relatively to other objects, not absolutely. Therefore actual positions are not relevant, but the position relations between objects. All together, this leads to a very flexible setup. The position of the Kinect device, the robot or the human can be varied without causing a lot of effort, to adapt the application to the new setting.

Two more aspects are evaluated, the usability of this Kinect application for force control and the range of possible velocities:

Force control: Implementing force control was not part of this thesis. But this Kinect application does not seem to be very applicable for force control either, because the haptic feedback for the operator is missing, unlike as for example with a haptic device. So in the domain of force control the big advantages of natural and intuitive motion control disappear, because without haptic feedback force control will not be perceived as natural.

Possible velocities: The experiment concerning the dynamic behavior of the Kinect showed that the Kinect can follow the robot at all velocities, that the safety system in extctrl-mode allowed $v \leq\sim 1.1[m/s]$. In practical use, for safety reasons neither a robot nor a human should move faster. So the tracking capabilities of the Kinect are proven to be sufficient for the requirements of the scenario of the ROSETTA project.

One further observation was made during this thesis work. The FRIDA robot was designed to have not just human-like proportions, but also movement characteristics oriented on the human movement patterns. During my experiments I could observe, that this goal has been reached. As written in Subsection 3.5.3, when moving the FRIDA only singularities at the edges of the robot workspace are avoided by the Simulink controller. Singularities within the robot workspace are not prevented explicitly. So as long as FRIDA was controlled to perform random non-human-like movement, it hit inner singularities a lot of times, which always led to abortion of the control program. But if FRIDA is used with the same inverse velocity kinematics in the shadowing application, therefore performing human movement, inner singularities are very rare.

5.3. Future Prospectives

The final step, which is missing from this thesis for technical reasons, is to control a robot using sensor fusion of two Kinect sensors. But in the future even more devices could be introduced into this control. A net of several Kinect devices could be allocated around and observe a whole assembly line including several possibly mobile robots, cf. Figure 5.3

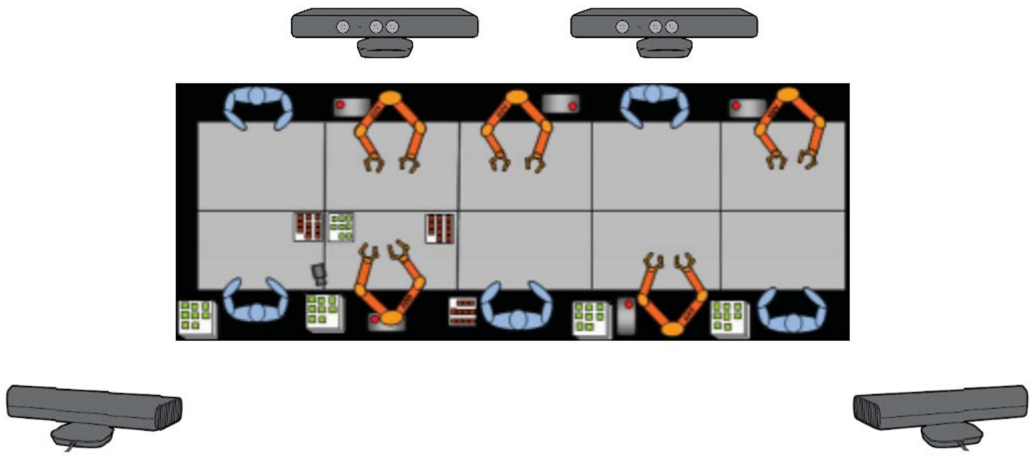


Figure 5.3.: Assembly line[1] observed by net of Kinect sensors

The main challenge in this extended setup would be to reasonably allocate control paths. In this assembly line several humans would be present controlling the robots. Each controlled robot has to be allocated to one controlling human. In this context it should not matter, which of the Kinect devices actually is observing the human or the robot. So the perspective of the control has to be shifted from a Kinect-centered point of view as in this thesis to a more human-centered point of view. That also corresponds with another objective of the ROSETTA project.

For the workspace surveillance another approach could be taken, especially in this extended setup. To better deal with the objects in vision that have to be ignored for the workspace surveillance, when setting up the assembly line, a depth map could be recorded, including all rigid components of the scene. This recorded depth map then could be used as background that is subtracted from every current depth map during the program run. The resulting depth map would only include the humans and the robots currently present in the assembly line.

Additionally, the Kinect can be used for robot training. Like proposed in [14] learning of new movements could be based on action primitives that are combined to full movements by action grammar rules.

The field of applications of the Kinect is large and due to active research quickly expanding. Integrating it into current robot systems can lead to a similar boost in robotics research, as it happened in 3D-image processing.

[1]The schematic of the assembly line is taken from the ROSETTA project description on the website of the Department of Automatic Control of Lund University: http://www.control.lth.se/Research/Robotics/ROSETTA.html

A. Source Code

In this chapter the full source code of this application is displayed. The structure of this chapter is oriented on the structure of Chapter 3. If not mentioned otherwise, the code is written in C++.

A.1. Calibration, including Initialisation of two Kinect devices

```
//---------------------------------------------------------------------------
// Includes
//---------------------------------------------------------------------------
#include <XnOpenNI.h>//Needed for OpenNI
#include <XnLog.h>
#include <XnCppWrapper.h>

#include <math.h>

#include <linalg.h>//Used for Matrix Inversion
#include <ap.h>

#include "asa047.h" //Used for Nelder-Mead-Algorithm

#include <unistd.h>

#include <iostream>


//---------------------------------------------------------------------------
// Defines
//---------------------------------------------------------------------------
#define SAMPLE_XML_PATH "./SamplesConfig.xml"
#define NUM_OF_SENSORS 2

//---------------------------------------------------------------------------
// Macros
//---------------------------------------------------------------------------
#define CHECK_RC(rc, what) \
  if (rc != XN_STATUS_OK) \
  { \
  printf("%s failed: %s\n", what, xnGetStatusString(rc)); \
  return rc; \
  }
```

```
#define CHECK_RC_CONTINUE(rc, what) \
  if (rc != XN_STATUS_OK) \
  { \
  printf("%s failed: %s\n", what, xnGetStatusString(rc)); \
  }

//---------------------------------------------------------------------------
// Code
//---------------------------------------------------------------------------

using namespace xn;

const double PI = 4.0*atan(1.0);

Context context;

struct DepthRgbSensors
{
  char name[80];
  DepthGenerator depth; //OpenNI objects
  DepthMetaData dmd;
  ImageGenerator image;
  ImageMetaData imd;
};

DepthRgbSensors sensors[NUM_OF_SENSORS];

int nbrCalibPoints = 10; //Just a initial proposal
//the actual number of Calibration Points
//has to be entered as Command Line Argument
float* pCalibPoints[2];



float dist(XnPoint3D p1, XnPoint3D p2) {
  //returns the Euclidian Distance between p1 and p2
  float ret = (p1.X-p2.X)*(p1.X-p2.X)+(p1.Y-p2.Y)*(p1.Y-p2.Y)+(p1.Z-p2.Z)*(p1.Z-p2.Z);
  return sqrt(ret);
}

float correlation(int nbr, float* vec1, float* vec2) {
  // returns the correlation coefficient of vec1 and vec2
  float e1=0, e2=0, s1=0, s2=0, s12=0;
  for(int i=0; i<nbr; i++) {
    e1 += *(vec1+i);
    e2 += *(vec2+i);
  }
  e1/=nbr;
  e2/=nbr;
  for(int i=0; i<nbr; i++) {
    s1 += (*(vec1+i)-e1)*(*(vec1+i)-e1);
    s2 += (*(vec2+i)-e2)*(*(vec2+i)-e2);
    s12+= (*(vec1+i)-e1)*(*(vec2+i)-e2);
  }
  return s12 / sqrt(s1) / sqrt(s2);
}

float corr2(int nbr, float* vec1, float* vec2) {
  // returns the coefficient corr2 of vec1 and vec2
  float nom=0, den=0;
  for(int i=0; i<nbr; i++) {
    den += (*(vec1+i))*(*(vec1+i));
    den += (*(vec2+i))*(*(vec2+i));
    nom += (*(vec1+i))*(*(vec2+i));
  }
  return 2*nom / den;
}

int matrixMult(int n1, int m1, double* mat1, int n2, int m2, double* mat2, int n3, int m3, double* mat3){
  // multiplies mat1 with mat2 and stores the result in mat3
  if (m1!=n2) {
    printf("Input matrix dimensions do not agree!\n");
    return -1;
  }
  if ((n1!=n3)||(m2!=m3)) {
    printf("Output matrix does not have the right size!\n");
    return -1;
  }

  double Mat1[n1][m1], Mat2[n2][m2], Mat3[n3][m3];

  for (int j=0; j<n1; j++) {
    for (int i=0; i<m1; i++) {
      Mat1[j][i] = *mat1;
      mat1++;
    }
  }

  for (int j=0; j<n2; j++) {
    for (int i=0; i<m2; i++) {
      Mat2[j][i] = *mat2;
      mat2++;
    }
  }

  for (int j=0; j<n3; j++) {
    for (int i=0; i<m3; i++) {

      double ret = 0;
      for (int k=0; k<m1; k++) {
        ret += Mat1[j][k]*Mat2[k][i];
      }
      Mat3[j][i] = ret;

    }
  }

  for (int j=0; j<n3; j++) {
    for (int i=0; i<m3; i++) {
      *mat3 = Mat3[j][i];
      mat3++;
    }
  }

  return 1;
}

void rotationMatrix(double kx, double ky, double kz, double* ret) {
  //calculates the rotatation matrix  corresponding to the rotation vector (kx ky kz)' and stores it in ret
  double theta2 = kx*kx + ky*ky + kz*kz;
  double theta = sqrt(theta2);
  double si = sin(theta);
  double co = cos(theta);
  double fac = 1-co;

  *ret   = kx*kx/theta2*fac + co;
  *(ret+1) = kx*ky/theta2*fac - kz/theta*si;
  *(ret+2) = kx*kz/theta2*fac + ky/theta*si;
  *(ret+3) = ky*kx/theta2*fac + kz/theta*si;
  *(ret+4) = ky*ky/theta2*fac + co;
  *(ret+5) = ky*kz/theta2*fac - kx/theta*si;

  *(ret+6) = kz*kx/theta2*fac - ky/theta*si;
  *(ret+7) = kz*ky/theta2*fac + kx/theta*si;
  *(ret+8) = kz*kz/theta2*fac + co;
}

double fun(double x[6]) {
  // Error function to be minimized by Nelder Mead Algorithm
  // returns the function value depending on the six parameters included in x
  // x[6] = {tx, ty, tz, kx, ky, kz}
  double R[3][3];
  rotationMatrix(x[3], x[4], x[5], &R[0][0]);

  double ret = 0;

  for(int i=0; i<3*nbrCalibPoints; i++) {

    double x1[3] = {*(pCalibPoints[0]++), *(pCalibPoints[0]++), *(pCalibPoints[0]++)};
    double x2[3] = {*(pCalibPoints[1]++), *(pCalibPoints[1]++), *(pCalibPoints[1]++)};

    matrixMult(3, 3, &R[0][0], 3, 1, &x2[0], 3, 1, x2);

    for(int j=0; j<3; j++) {
      x2[j] = x1[j]-x[j]-x2[j];
      ret += x2[j]*x2[j];
    }

  }

  pCalibPoints[0] -= 9*nbrCalibPoints;
  pCalibPoints[1] -= 9*nbrCalibPoints;

  return ret;
}


int findCalibrationPoints(const XnRGB24Pixel* pImage, XnPoint3D* ret) {
  // detects $nbrCalibPoints calibration points in the color image by the Kinect
  // and stores their projective x- and y- coordinates in ret

  XnPoint3D stack[5*nbrCalibPoints]; // there has to be enough memory space for all critical points
  float corrstack[5*nbrCalibPoints];

  int grayscale[480][640]; // getting the grayvalue of each pixel
  for(int j=0; j<480; j++) {
    for(int i=0; i<640; i++) {

      grayscale[j][i] = pImage->nBlue + pImage->nGreen + pImage->nRed;
      pImage++;

    }
  }

  // Using Correlation, compare to ideal Window 10x10 with four quarters [black white; white black]
  float mu2 = 765 / 2;
  float si2 = mu2;

  int count = 0;

  for(int j=10; j<470; j++) {
    for(int i=10; i<630; i++) {
```

```
    float si1 = 0, si12 = 0, mu1 =0, corr;

    for(int v=-5; v<5; v++) {
  for(int u=-5; u<5; u++) {
    mu1 += grayscale[j+v][i+u];
  }
    }

    mu1 /= 100;

    for(int v=-5; v<5; v++) {
      for(int u=-5; u<5; u++) {

        si1 += (grayscale[j+v][i+u]-mu1)*(grayscale[j+v][i+u]-mu1);
        if (((v<0)&&(u<0))||((v>=0)&&(u>=0))) {
          //deciding whether the ideal Window has a black or white pixel at this spot
          si12 -= mu2 * (grayscale[j+v][i+u]-mu1);
        } else {
        si12 += mu2 * (grayscale[j+v][i+u]-mu1);
        }

      }
    }

    si1 = sqrt(si1/100);
    si12 /= 100;

    corr = si12 / si1 /si2;

    if (corr>0.8) {
      stack[count].X = i;
      stack[count].Y = j;
      corrstack[count] = corr;
      count++;
    }
  }
}

if (count <= nbrCalibPoints) {
  // not enough candidates for Calibration points were found
  for (int i=0; i<count; i++) {
    ret->X = stack[i].X;
    ret->Y = stack[i].Y;
    ret->Z = stack[i].Z;
    ret++;

  }

  printf("Found %d calibration points!\n", count);
  return count;

} else {
  // select the $nbrCalibPoints Calibration Points with the highest correlation coefficient
  for (int i=0; i<nbrCalibPoints; i++) {

    int max;
    float maxcorr = 0.8;

    for (int j=0; j<count; j++) {
      if (corrstack[j] > maxcorr) {
        max = j;
        maxcorr = corrstack[j];
      }
    }
```

```
      if (maxcorr == 0.8) {
        printf("Used %d calibration points!\n", i);
        return i;
      }

      for (int j=1; j<=i; j++) {
        // checks that no other Calibration Point less than 8 pixels away has already been selected
        int dist2 = (int) ((ret-j)->X - stack[max].X)*((ret-j)->X - stack[max].X)
                        + ((ret-j)->Y - stack[max].Y)*((ret-j)->Y - stack[max].Y)
                        + ((ret-j)->Z - stack[max].Z)*((ret-j)->Z - stack[max].Z);
        if (dist2 < 64) {
          corrstack[max] = 0;
          i--;
          goto mark2;
        }
      }

      ret->X = stack[max].X;
      ret->Y = stack[max].Y;
      ret->Z = stack[max].Z;
      corrstack[max] = 0;
      ret++;

mark2: continue;

    }
  }

  //Sort the Calibration Points into a distinct order
  ret -= nbrCalibPoints;

  for(int j=0; j<nbrCalibPoints-1; j++) {
    for (int i=nbrCalibPoints-2; i>=j; i--) {
      if(((ret+i)->X > (ret+i+1)->X) || (((ret+i)->X == (ret+i+1)->X) && ((ret+i)->Y > (ret+i+1)->Y))) {

        XnPoint3D swap;
        swap.X = (ret+i)->X;
        swap.Y = (ret+i)->Y;
        swap.Z = (ret+i)->Z;
        (ret+i)->X = (ret+i+1)->X;
        (ret+i)->Y = (ret+i+1)->Y;
        (ret+i)->Z = (ret+i+1)->Z;
        (ret+i+1)->X = swap.X;
        (ret+i+1)->Y = swap.Y;
        (ret+i+1)->Z = swap.Z;

      }
    }
  }

  printf("Found %d calibration points!\n", count);

  return nbrCalibPoints;
}


int main(int argc, char* argv[])
{

  printf("Program 2Kinects: Initialization\n");
  if(argc != 0) { // Read the number of Calibration Points
    nbrCalibPoints = atoi(argv[1]);
  }
```

```
printf("Number of Calibration Points: %d\n", nbrCalibPoints);

XnStatus nRetVal = XN_STATUS_OK;

// Getting Sensors information and configure all sensors
nRetVal = context.Init();

NodeInfoList devicesList;
nRetVal = context.EnumerateProductionTrees(XN_NODE_TYPE_DEVICE, NULL, devicesList);
CHECK_RC(nRetVal, "Enumerate");

int dev_cnt=0;
for (NodeInfoList::Iterator it = devicesList.Begin(); it != devicesList.End(); ++it, ++dev_cnt)
{
  // Create the device node
  NodeInfo deviceInfo = *it;
  nRetVal = context.CreateProductionTree(deviceInfo);
  CHECK_RC(nRetVal, "Create Device");

  // create a query to depend on this node
  Query query;
  query.AddNeededNode(deviceInfo.GetInstanceName());

  XnMapOutputMode outputMode;
  outputMode.nXRes = 640;
  outputMode.nYRes = 480;
  outputMode.nFPS = 30;

  //Copy the device name
  xnOSMemCopy(sensors[dev_cnt].name,deviceInfo.GetInstanceName(),
    xnOSStrLen(deviceInfo.GetInstanceName()));
  // now create a depth generator over this device
  nRetVal = context.CreateAnyProductionTree(XN_NODE_TYPE_DEPTH, &query, sensors[dev_cnt].depth);
  CHECK_RC(nRetVal, "Create Depth");

  nRetVal = sensors[dev_cnt].depth.SetMapOutputMode(outputMode);
  CHECK_RC(nRetVal, "Set Depth Output Mode");

  // now create a image generator over this device
  nRetVal = context.CreateAnyProductionTree(XN_NODE_TYPE_IMAGE, &query, sensors[dev_cnt].image);
  CHECK_RC(nRetVal, "Create Image");

  nRetVal = sensors[dev_cnt].image.SetMapOutputMode(outputMode);
  CHECK_RC(nRetVal, "Set Image Output Mode");

  // set origin of depth frame onto RGB-camera
  if(sensors[dev_cnt].depth.IsCapabilitySupported("AlternativeViewPoint")) {
    nRetVal = sensors[dev_cnt].depth.GetAlternativeViewPointCap().SetViewPoint(sensors[dev_cnt].image);
    CHECK_RC(nRetVal, "match Depth and RGB points of view");
  } else printf("Capability not supported\n");

  if (sensors[dev_cnt].depth.GetAlternativeViewPointCap().IsViewPointAs(sensors[dev_cnt].image)) {
    printf("depth alternative view\n");
  }

}

printf("%d sensors found\n", dev_cnt);

context.SetGlobalMirror(true);

//Start generating
nRetVal = context.StartGeneratingAll();
CHECK_RC(nRetVal, "Start generating all");
```

```
// array for all calibration points seen from both Kinects
XnPoint3D calibPoints[2][3*nbrCalibPoints];
float key[2][nbrCalibPoints-1];
int round = 0;

while(!xnOSWasKeyboardHit()) {
  while(!xnOSWasKeyboardHit()) {

    printf("Measurement Round %d: Put the Calibration Board in a new steady position!\n", round+1);
    sleep(5);

    bool keyok = true;

    while(!xnOSWasKeyboardHit()) { //Find Calibration Points

      context.WaitAnyUpdateAll();

      int all[2];

      for(int i=0; i<2; i++) {
        sensors[i].depth.GetMetaData(sensors[i].dmd);
        sensors[i].image.GetMetaData(sensors[i].imd);
      }

      for(int i=0; i<2; i++) {

        for (int j=nbrCalibPoints*round; j<nbrCalibPoints*(round+1); j++) {
          calibPoints[i][j].X=0;
          calibPoints[i][j].Y=0;
          calibPoints[i][j].Z=0;
        }

        const XnRGB24Pixel* pImage = sensors[i].imd.RGB24Data();

        // Function call
        all[i] = findCalibrationPoints(pImage, &calibPoints[i][nbrCalibPoints*round]);

        // Find depth values of Calibration points
        for (int j=nbrCalibPoints*round; j<nbrCalibPoints*round+all[i]; j++) {

          calibPoints[i][j].Z = *(sensors[i].dmd.Data()+640*((int)calibPoints[i][j].Y)
                                +(int)calibPoints[i][j].X);
          int chg = 4;
          while(calibPoints[i][j].Z==0) {
            switch (chg%4) {
              case 0: calibPoints[i][j].Z = *(sensors[i].dmd.Data()
                      +640*((int)calibPoints[i][j].Y)+(int)calibPoints[i][j].X+chg/4);
                      break;
              case 1: calibPoints[i][j].Z = *(sensors[i].dmd.Data()
                      +640*((int)calibPoints[i][j].Y+chg/4)+(int)calibPoints[i][j].X);
                      break;
              case 2: calibPoints[i][j].Z = *(sensors[i].dmd.Data()
                      +640*((int)calibPoints[i][j].Y)+(int)calibPoints[i][j].X-chg/4);
                      break;
              case 3: calibPoints[i][j].Z = *(sensors[i].dmd.Data()
                      +640*((int)calibPoints[i][j].Y-chg/4)+(int)calibPoints[i][j].X);
                      break;
            }
            chg++;
          }
        }
      }

      // Check that both Kinects found all Calibration points, if not repeat
      if((all[0]==nbrCalibPoints) && (all[1]==nbrCalibPoints)) break;

    }

    for(int i=0; i<2; i++) {
      // judge calibration points

      // Get Description in 3D real world coordinates of respective Kinect Frame
      sensors[i].depth.ConvertProjectiveToRealWorld(nbrCalibPoints,
               &calibPoints[i][nbrCalibPoints*round], &calibPoints[i][nbrCalibPoints*round]);

      // Calculate key for set of Calibration points
      for (int j=0; j<nbrCalibPoints-1; j++) {
        key[i][j] = dist(calibPoints[i][j+nbrCalibPoints*round], calibPoints[i][j+nbrCalibPoints*round+1]);
        if (key[i][j] < 40) keyok = false;
      }

    }

    // measuring the similarity of the two keys
    float coef = corr2(nbrCalibPoints-1, &key[0][0], &key[1][0]);
    float corr = correlation(nbrCalibPoints-1, &key[0][0], &key[1][0]);

    printf("Correlation Coefficient: %f\n\n", corr);
    printf("other Coefficient: %f\n\n", coef);

    // Threshold for key coefficients
    if((coef > 0.85) && (corr > 0.85) && keyok) break;

    printf("Measurement not accurate enough. Repeat this round!\n");

  }

  if (++round == 3) break;

}

//Calibration with Least Squares

printf("Least-Squares\n");

double rot[2][3][3]; //first array-coordinate: 0:Least Squares, 1:Nelder-Mead
double trans[2][3];

double G[3*nbrCalibPoints][4], GT[4][3*nbrCalibPoints], y[3][3*nbrCalibPoints];

for (int i=0; i<3*nbrCalibPoints; i++) {
  G[i][0] = 1;
  G[i][1] = calibPoints[1][i].X;
  G[i][2] = calibPoints[1][i].Y;
  G[i][3] = calibPoints[1][i].Z;
  for (int j=0; j<4; j++) {
    GT[j][i]=G[i][j];
  }
  y[0][i] = calibPoints[0][i].X;
  y[1][i] = calibPoints[0][i].Y;
  y[2][i] = calibPoints[0][i].Z;
}

double help1[4][4];

matrixMult(4, 3*nbrCalibPoints, &GT[0][0], 3*nbrCalibPoints, 4, &G[0][0], 4, 4, &help1[0][0];

// matrix inversion using the open library alglib
```

```
alglib::real_2d_array help1a;
double sort[16];
for (int j=0; j<4; j++) {
  for(int i=0; i<4; i++) {
    sort[j*4+i] = help1[j][i];
  }
}

help1a.setcontent(4, 4, sort);

alglib::ae_int_t info;
alglib::matinvreport rep;

alglib::spdmatrixinverse(help1a, info, rep);

for (int j=0; j<4; j++) {
  for (int i=0; i<4; i++) {
    help1[j][i] = help1a[j][i];
  }
}


for (int i=0; i<3; i++) {

  double help2[4];

  matrixMult(4, 3*nbrCalibPoints, &GT[0][0], 3*nbrCalibPoints, 1, &y[i][0], 4, 1, &help2[0]);

  double help3[4];

  matrixMult(4, 4, &help1[0][0], 4, 1, &help2[0], 4, 1, &help3[0]) != 1;

  //Result of Least Squares
  trans[0][i] = help3[0];
  rot[0][i][0] = help3[1];
  rot[0][i][1] = help3[2];
  rot[0][i][2] = help3[3];

}

// Calibration with Nonlinear Function Optimization: Nelder-Mead

float X0[3*nbrCalibPoints][3];
float X1[3*nbrCalibPoints][3];

for (int i=0; i<3*nbrCalibPoints; i++) {
  X0[i][0] = calibPoints[0][i].X;
  X0[i][1] = calibPoints[0][i].Y;
  X0[i][2] = calibPoints[0][i].Z;

  X1[i][0] = calibPoints[1][i].X;
  X1[i][1] = calibPoints[1][i].Y;
  X1[i][2] = calibPoints[1][i].Z;
}

pCalibPoints[0] = &X0[0][0];
pCalibPoints[1] = &X1[0][0];

printf("\nNelder-Mead\n");

//Nelder Mead Algorithm included in open library asa047

//arguments for nelder(.)
int icount;
```

```
  int ifault;
  int kcount;
  int konvge;
  int n = 6; // x[6] = {tx, ty, tz, kx, ky, kz}
  int numres;
  double reqmin  = 1.0E-12;
  double *start = new double[n];
  double *step = new double[n];
  double *xmin = new double[n];
  double ynewlo;

  //  start vector
  start[0] = -280.;
  start[1] = 0.;
  start[2] = 0.;
  start[3] = 0.1;
  start[4] = 0.;
  start[5] = 0.;

  // starting step size
  step[0] = 10.0;
  step[1] = 1.0;
  step[2] = 10.0;
  step[3] = 0.01;
  step[4] = 0.05;
  step[5] = 0.01;

  konvge = 10;
  kcount = 5000;

  // starting function value
  ynewlo = fun (start);

  // Nelder Mead Algorithm
  nelmin (fun, n, start, xmin, &ynewlo, reqmin, step, konvge, kcount, &icount, &numres, &ifault);

  // extracting calibration information from minimizing vector
  trans[1][0] = xmin[0];
  trans[1][1] = xmin[1];
  trans[1][2] = xmin[2];

  rotationMatrix(xmin[3], xmin[4], xmin[5], &rot[1][0][0]);

  delete [] start;
  delete [] step;
  delete [] xmin;

  context.Shutdown();
  return 0;
}
```

The resultant rotation matrices and translation vectors calculated with both, the least squares and the Nelder-Mead algorithm are saved in the arrays `rot` and `trans`.

A.2. Workspace Surveillance

A.2.1. LC-file “getT44setvref.lc”

```
sample float vref[3];
sample float T44[16];
sample float x2[4];
sample float x3[4];
sample float x5[4];
```

A.2.2. Client Program for Main Computer

```
#include <XnOS.h>            //Needed for OpenNI
#include <XnCppWrapper.h>
#include <XnFPSCalculator.h>

#include <math.h>

#include <sys/socket.h>      //Needed for LabComm
extern "C" {
#include <orca_client.h>
#include "getT44setvref.h"
}
#include <pthread.h> //Needed for LabComm decoder
#include <sys/time.h>

#define SAMPLE_XML_PATH "./SamplesConfig.xml"

#define CHECK_RC(rc, what) \
    if (rc != XN_STATUS_OK) \
    { \
        printf("%s failed: %s\n", what, xnGetStatusString(rc)); \
        return rc; \
    }

using namespace xn;

Context context;       //OpenNI objects
DepthGenerator depth;
DepthMetaData dmd;

HandsGenerator hand;

//Position to gain simulated Control.
//In this example a hand that has been at this position is regarded as controlling hand
//The robot will not stop, but only slow down near it.
XnPoint3D startp = xnCreatePoint3D(0.0f, 0.0f, 600.0f);

//Position of controlling hand
XnPoint3D handpoint = xnCreatePoint3D(0.0f, 0.0f, 0.0f);;
```

```
struct orca_client *orca;

//Homogeneous transformation matrix of robot tool expressed in robot base frame
float T44[4][4] = {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}};

//Robot joint positions expressed in robot base frame
float x2[4] = {0, 0, 0, 1};
float x3[4] = {0, 0, 0, 1};
float x5[4] = {0, 0, 0, 1};

//Rigid Transformation from robot base frame to Kinect frame
float HKR[4][4] = {{0, -1, 0, 0}, {0, 0, 1, -61}, {-1, 0, 0, 2240}, {0, 0, 0, 1}};

//Objects that are allowed to be near the robot tool:
//tabel, workpiece, robot socket
const int nbrQuaders = 3;
int quaders[nbrQuaders][2][3] = {{{-225, -300, 1790}, {225, -55, 2240}},
                                 {{-70, -61, 1870}, {75, 10, 1970}},
                                 {{-210, -60, 2090}, {210, 0, 2240}}};

orca_client_channel_t *signals2Rob;
orca_client_channel_t *signalsFromRob;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int saveDist = 300;


// Handcallbacks

void XN_CALLBACK_TYPE HandCreate(HandsGenerator& generator, XnUserID nId, const XnPoint3D* pPosition,
                                 XnFloat fTime, void* pCookie) {
  handpoint.X = pPosition->X;
  handpoint.Y = pPosition->Y;
  handpoint.Z = pPosition->Z;
  printf("%d %f %f %f\n", nId, handpoint.X, handpoint.Y, handpoint.Z);
}

void XN_CALLBACK_TYPE HandUpdate(HandsGenerator& generator, XnUserID nId, const XnPoint3D* pPosition,
                                 XnFloat fTime, void* pCookie) {
  handpoint.X = pPosition->X;
  handpoint.Y = pPosition->Y;
  handpoint.Z = pPosition->Z;
  printf("%d %f %f %f\n", nId, handpoint.X, handpoint.Y, handpoint.Z);
}


void XN_CALLBACK_TYPE HandDestroy(HandsGenerator& generator, XnUserID nId, XnFloat fTime, void* pCookie)
{
  handpoint.X = 0;
  handpoint.Y = 0;
  handpoint.Z = 0;
  printf("%d 0 0 0 %f\n", nId, fTime);
  hand.StartTracking(startp);
}

/* This function which continuously receives data from the robot must be started from a thread,
 because the function will not end until the connection to the robot is shut down. */

void* handler_thread(void* ptr) {
  orca_client_channel_t* signalsFromRob = (orca_client_channel_t*) ptr;
  labcomm_decoder_run(signalsFromRob->decoder);
}
```

```
// The following functions are called, when the respective signal is received from the robot. */

void handle_T44(getT44setvref_T44* v , void* context) {
  pthread_mutex_lock(&mutex);
  for (int i=0; i<12; i++) {
    *(&T44[0][0]+i) = v->a[i];
  }
  pthread_mutex_unlock(&mutex);
}

void handle_x2(getT44setvref_x2* v , void* context) {
  pthread_mutex_lock(&mutex);
  for (int i=0; i<3; i++) {
    *(&x2[0]+i) = v->a[i];
  }
  pthread_mutex_unlock(&mutex);
}

void handle_x3(getT44setvref_x3* v , void* context) {
  pthread_mutex_lock(&mutex);
  for (int i=0; i<3; i++) {
    *(&x3[0]+i) = v->a[i];
  }
  pthread_mutex_unlock(&mutex);
}

void handle_x5(getT44setvref_x5* v , void* context) {
  pthread_mutex_lock(&mutex);
  for (int i=0; i<3; i++) {
    *(&x5[0]+i) = v->a[i];
  }
  pthread_mutex_unlock(&mutex);
}

//float arguments
int matrixMult(int n1, int m1, float* mat1, int n2, int m2, float* mat2, int n3, int m3, float* mat3) {
...[This function is exactliy the same as the function matrixMult(.) of the Calibration Program,
just with float arguments]...
}

int main(int argc, char **argv) {

  // Initialize the Kinect

  XnStatus rc;

  EnumerationErrors errors;
  rc = context.InitFromXmlFile(SAMPLE_XML_PATH, &errors);
  CHECK_RC(rc, "Initialize from Xml-file");

  rc = context.FindExistingNode(XN_NODE_TYPE_DEPTH, depth);
  CHECK_RC(rc, "Find existing node depth");

  rc = hand.Create(context);
  rc = context.FindExistingNode(XN_NODE_TYPE_HANDS, hand);
  CHECK_RC(rc, "Find existing node hands");

  XnCallbackHandle(h);
  rc = hand.RegisterHandCallbacks(HandCreate, HandUpdate, HandDestroy, NULL,h);

  // Initialize the LabComm Connection

  //Connection To Robot
```

```
char* ip2Robot = argv[1];
int portRobot = atoi(argv[2]);

orca = orca_client_new_tcp(ip2Robot, portRobot);

if (!orca) {
  printf("Connection to host %s: port %d failed\n", ip2Robot ,portRobot);
  return 0;
}

char *channel2Rob = "vref";
char **channel2Rob_p = &channel2Rob;

orca_client_selection_t selection2Rob;
selection2Rob.n_0 = 1;
selection2Rob.a = channel2Rob_p;
signals2Rob = orca_client_select_input_tcp(orca ,&selection2Rob);

labcomm_encoder_register_getT44setvref_vref(signals2Rob->encoder);

//Connection From Robot

char *channelFromRob[4];
channelFromRob[0] = "T44";
channelFromRob[1] = "x2";
channelFromRob[2] = "x3";
channelFromRob[3] = "x5";

orca_client_selection_t selectionFromRob;

selectionFromRob.n_0 = 4;
selectionFromRob.a = channelFromRob;
signalsFromRob = orca_client_select_output_tcp (orca, &selectionFromRob);
labcomm_decoder_register_getT44setvref_T44(signalsFromRob->decoder, handle_T44, 0);
labcomm_decoder_register_getT44setvref_x2(signalsFromRob->decoder, handle_x2, 0);
labcomm_decoder_register_getT44setvref_x3(signalsFromRob->decoder, handle_x3, 0);
labcomm_decoder_register_getT44setvref_x5(signalsFromRob->decoder, handle_x5, 0);


// Create an independent thread that deals with receiving from the robot
pthread_t thread ;
int iret1 = pthread_create(&thread, NULL ,handler_thread, (void*)signalsFromRob);

// Start generating

XnFPSData xnFPS; // Frames per Second Counter
rc = xnFPSInit(&xnFPS, 180);
CHECK_RC(rc, "FPS Init");

rc = context.StartGeneratingAll();
CHECK_RC(rc, "Start Generating all");

printf("Initialization successful!\n");


// main loop

bool frm1 = true; //shows whether in the previous frame was a dangerous point

hand.StartTracking(startp);

while (!xnOSWasKeyboardHit()) {

  xnFPSMarkFrame(&xnFPS);
```

```
rc = context.WaitOneUpdateAll(depth);   //Update Depth Map
CHECK_RC(rc, "Update");

depth.GetMetaData(dmd);
const XnDepthPixel* pDepth = dmd.Data();

//Save all detected points as real world points
//could also handle points from a second Kinect
XnPoint3D points[480][640];

for (int j=0; j<480; j++) {
  for (int i=0; i<640; i++) {
    points[j][i].X = i;
    points[j][i].Y = j;
    points[j][i].Z = *pDepth;
    pDepth++;
  }
}

depth.ConvertProjectiveToRealWorld(307200, &points[0][0], &points[0][0]);

float joint[3][4];

// Transform joint and tool(here called flange) position into description in Kinect frame
pthread_mutex_lock(&mutex);
float flange[4] = {T44[0][3], T44[1][3], T44[2][3], 1};
matrixMult(4, 4, &HKR[0][0], 4, 1, &x2[0], 4, 1, &joint[0][0]);
matrixMult(4, 4, &HKR[0][0], 4, 1, &x3[0], 4, 1, &joint[1][0]);
matrixMult(4, 4, &HKR[0][0], 4, 1, &x5[0], 4, 1, &joint[2][0]);
pthread_mutex_unlock(&mutex);

matrixMult(4, 4, &HKR[0][0], 4, 1, &flange[0], 4, 1, &flange[0]);

// Tollerance area

float vec1[3] = {(joint[2][0]-flange[0])/5, (joint[2][1]-flange[1])/5, (joint[2][2]-flange[2])/5};

float tollpoints[19][4];
for (int i=0; i<5; i++) {
  for (int j=0; j<3; j++) {
    tollpoints[i][j] = flange[j]+(i)*vec1[j];
  }
  tollpoints[i][3] = 60*60; //tollerance
}

tollpoints[0][3] = 30*30;

for (int j=0; j<3; j++) {
  tollpoints[5][j] = joint[2][j];
}
tollpoints[5][3] = 130*130;

for (int j=0; j<3; j++) {
  vec1[j] = (joint[1][j]-joint[2][j])/7;
}

for (int i=6; i<13; i++) {
  for (int j=0; j<3; j++) {
    tollpoints[i][j] = joint[2][j]+(i-5)*vec1[j];
  }
  tollpoints[i][3] = 100*100;
}

tollpoints[12][3] = 130*130;
```

```
    for (int i=13; i<19; i++) {
      for (int j=0; j<3; j++) {
        tollpoints[i][j] = joint[1][j]+(i-12)*vec1[j];
      }
      tollpoints[i][3] = 100*100;
    }

    tollpoints[18][3] = 130*130;

    int ret = saveDist*saveDist;
    int retpix = -1;

    XnPoint3D* ppoint = &points[0][0];

    for (int i=0; i<307200; i++, ppoint++) {

      if(ppoint->Z < 1) continue;   //discard points with depth value zero

      float help = 0;
      float add;

      add = abs(ppoint->X - flange[0]);
      if(add > ret) continue;    // If one coordinate is already farther away than ret
      help += add*add;  // the whole vector can't be closer

      add = abs(ppoint->Y - flange[1]);
      if(add > ret) continue;
      help += add*add;

      add = abs(ppoint->Z - flange[2]);
      if(add > ret) continue;
      help += add*add;

      if (help<ret) {
        //if a point is nearer than the tolerance radius to a tolerance point, it is discarded
        for (int j=0; j<19; j++) {
          float dist2 = (ppoint->X-tollpoints[j][0])*(ppoint->X-tollpoints[j][0]);
          dist2 += (ppoint->Y-tollpoints[j][1])*(ppoint->Y-tollpoints[j][1]);
          dist2 += (ppoint->Z-tollpoints[j][2])*(ppoint->Z-tollpoints[j][2]);
          if (dist2 < tollpoints[j][3]) goto mark1;
        }
        //if a point is included in one of the  allowed objects, it is discarded
        for (int j=0; j<nbrQuaders; j++) {
          if ((ppoint->X > quaders[j][0][0]) && (ppoint->X < quaders[j][1][0]) &&
              (ppoint->Y > quaders[j][0][1]) && (ppoint->Y < quaders[j][1][1]) &&
              (ppoint->Z > quaders[j][0][2]) && (ppoint->Z < quaders[j][1][2])) goto mark1;
        }
        ret = (int) help;
        retpix = i;

      }

mark1: continue;

    }

    float vint = 1;

if (ret < saveDist*saveDist) {    //dangerous point
      if(!frm1) {            //consecutive dangerous points
        ret = sqrt(ret);
        ppoint -= (307200 - retpix);
        float dist2 = (ppoint->X-handpoint.X)*(ppoint->X-handpoint.X);
        dist2 += (ppoint->Y-handpoint.Y)*(ppoint->Y-handpoint.Y);
        dist2 += (ppoint->Z-handpoint.Z)*(ppoint->Z-handpoint.Z);

        if (dist2 < 10000) {  // dangerous point belongs to controlling hand
          vint = (float)((60<ret)?(ret-60):0)/(float)(1.0 * (saveDist-60));
        } else   {
          vint = 0;
        }
      }
      frm1 = false;
    } else frm1 = true;

    //Type according to .lc-file
    getT44setvref_vref vref;

    for (int i=0; i<3; i++) {
      vref.a[i] = vint;
    }

//send vref to Simulink controller via LabComm connection
    labcomm_encode_getT44setvref_vref(signals2Rob->encoder, &vref);

    printf("FPS: %f\n", xnFPSCalc(&xnFPS));

  }
  context.Shutdown();
}
```

A.2.3. Excerpt of Simulink Controller Model

Matlab source code of user defined function block “jacob”:

```
function [T44_fkine, J0, Jn, x_2, x_3, x_5] = jacob(q)
% Calculation of forward kinematics and jacobian in base frame and flange frame

n_joints = 7;

q_offs = [0;0;pi/2;0;0;0;0];
q = q+q_offs;

shoulderLen = [...];
shoulderOffs = [...];
elbowOffs = [...]; %% These FRIDA parameters are not public yet
wristOffs = [...];
handOffs = [...];
upArmLen = [...];
lowArmLen = [...];
handLen = [...];

%          alpha        a         theta       d       Rev=0/Pris=1
dhp = [ -pi/2   shoulderOffs   q(1)   shoulderLen   0
         pi/2  -shoulderOffs   q(2)        0        0
        -pi/2   elbowOffs      q(7)    upArmLen     0
         pi/2  -elbowOffs      q(3)        0        0
        -pi/2   wristOffs      q(4)    lowArmLen    0
         pi/2   handOffs       q(5)        0        0
```

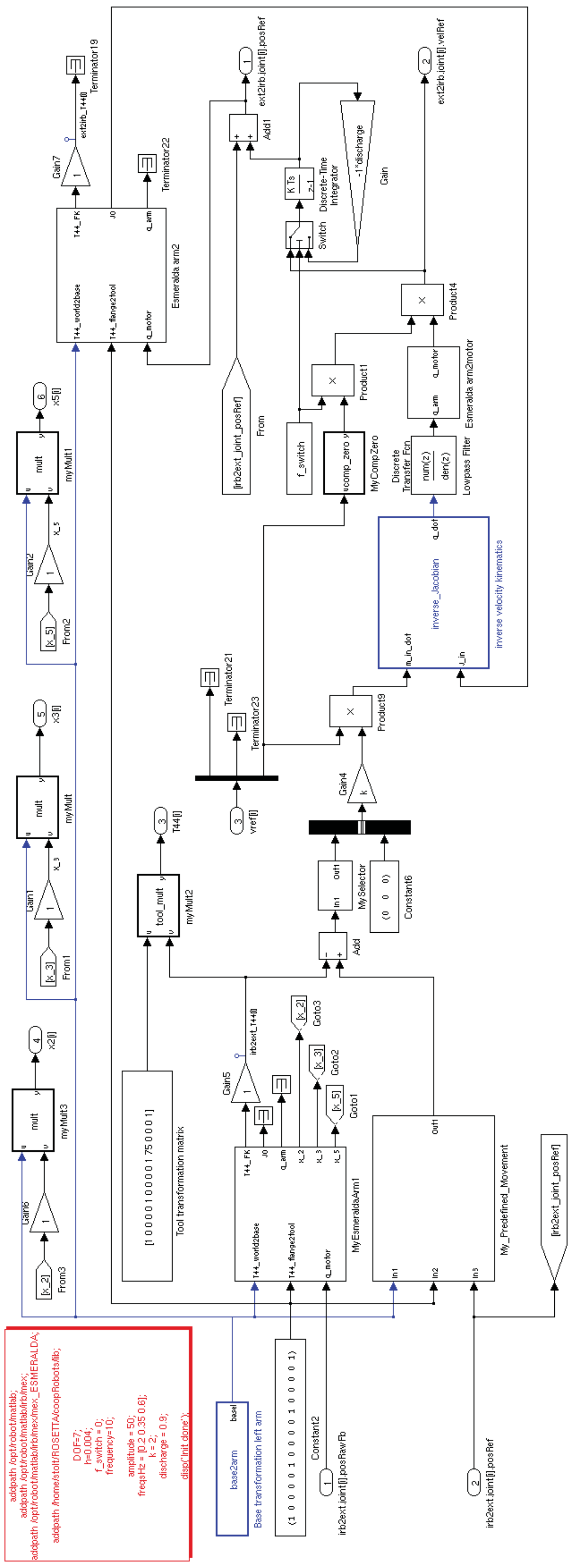

```
addpath /opt/robot/matlab;
addpath /opt/robot/matlab/irb/mex;
addpath /opt/robot/matlab/irb/mex/mex_ESMERALDA;

addpath /home/stolt/ROSETTA/coopRobots/lib;

DOF=7;
h=0.004;
f_switch = 0;
frequency=10;

amplitude = 50;
freqsHz = [0.2 0.35 0.6];
k = 2;
discharge = 0.9;

disp('Init done');
```

Figure A.1.: Simulink controller model for “Workspace Surveillance”

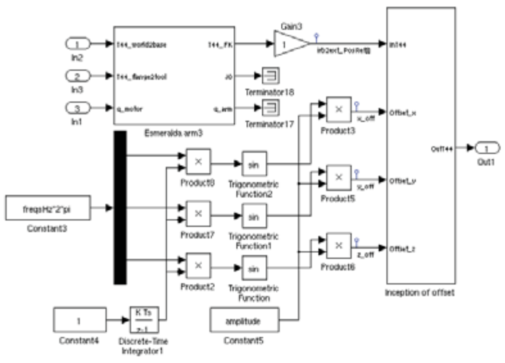


Figure A.2.: Simulink model for block "My predefined Movement"

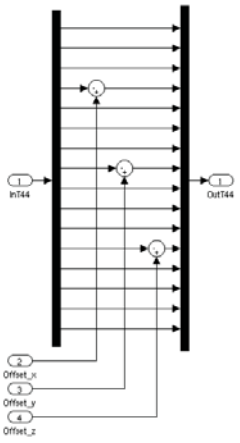


Figure A.3.: Simulink model for block "Inception of Offset"

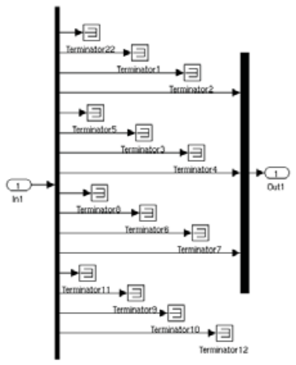


Figure A.4.: Simulink model for block "mySelector"

```
         0      0              q(6)     handLen     0 ];

base = eye(4);

trans = zeros(4,4,n_joints);

for j=1:n_joints
  t=[cos(dhp(j,3)) -sin(dhp(j,3))*cos(dhp(j,1))  sin(dhp(j,3))*sin(dhp(j,1))  dhp(j,2)*cos(dhp(j,3))
     sin(dhp(j,3)) cos(dhp(j,3))*cos(dhp(j,1))   -cos(dhp(j,3))*sin(dhp(j,1)) dhp(j,2)*sin(dhp(j,3))
     0             sin(dhp(j,1))                 cos(dhp(j,1))                dhp(j,4)
     0             0                             0                            1                      ];
  trans(:,:,j) = t;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% jacobn
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Jn_tmp = zeros(6,n_joints);
U = eye(4); %Alternatively transformation to TCP
for j=n_joints:-1:1
  U = trans(:,:,j) * U;

  if dhp(j,5) == 0
    % revolute axis
    d = [  -U(1,1)*U(2,4)+U(2,1)*U(1,4)
           -U(1,2)*U(2,4)+U(2,2)*U(1,4)
           -U(1,3)*U(2,4)+U(2,3)*U(1,4)];
    delta = U(3,1:3)';  % nz oz az
else
      % prismatic axis
      d = U(3,1:3)';    % nz oz az
      delta = zeros(3,1);  %  0  0  0
  end
  Jn_tmp(:,j) = [d; delta];
end
Jn_tmp = Jn_tmp(:,[1 2 4 5 6 7 3]);  % Compensate for weird joint order
Jn = reshape(Jn_tmp', 6*n_joints,1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% fkine
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x_2 = zeros(4,1);
x_3 = zeros(4,1);
x_5 = zeros(4,1);

T = base;
for i=1:n_joints,
T = T * trans(:,:,i);
    if (i==2)
        x_2 = T(:,4);
    elseif (i==4)
        x_3 = T(:,4);
    elseif (i==6)
        x_5 = T(:,4);
    end
end
%T = T * robot.tool;

T44_fkine = reshape(T',16,1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% tr2rot
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

R = T(1:3,1:3);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% jacob0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

J0_tmp = [R zeros(3,3); zeros(3,3) R] * Jn_tmp;
J0 = reshape(J0_tmp', 6*n_joints,1);
```

Matlab source code of user defined function block “inverse velocity kinematics”:

```
function q_dot  = inverse_Jacobian(m_in_dot, J_in)
%#eml

%Pseudoinverse Jacobian

J=zeros(6,7);

for j=1:6
    for i=1:7
        J(j,i) = J_in((j-1)*7+i);
    end;
end;

[U,S,V] = svd(J);

for i=1:6
    S(i,i) = S(i,i)^(-1);
end;

Jinv = V*S'*U';

%Joint velocities
q_dot = Jinv*m_in_dot;
```

Matlab source code of user defined function block “myCompZero”:

```
function y = comp_zero(u)
%#eml

eps =1e-3;
dim = length(u);
help = ones(1,dim);

for i=1:length(u)
    if (abs(u(i))<eps)
```

```
        help(i) = 0;
    end
end

y = help';
```

A.3. Face Detection

The face detection is formulated as a function. The input variable is the position, where a “Click”-gesture has been dtected. The return value is a boolean. It is `true` if a strait face looking at the Kinect is present and `false` otherwise.

```
bool faceDetection(const XnPoint3D* pEndPosition)   {

  bool ret = false; //return value

  XnVector3D pEndProj;
  g_Depth.ConvertRealWorldToProjective(1, pEndPosition, &pEndProj);

  pEndProj.X = (int) pEndProj.X;
  pEndProj.Y = (int) pEndProj.Y;

  XnUserID user = static_cast <XnUserID> (0);

  xn::DepthMetaData dmd;
  g_Depth.GetMetaData(dmd);

  xn::SceneMetaData smd;

  //Find user pixel in the area of the detected gesture
  for(int j=1; j<33; j++) {

    g_User.GetUserPixels(0, smd);
    const XnLabel* pLabels = smd.Data();

    user = static_cast <XnUserID> (*(pLabels + (int)(pEndProj.Y*640+pEndProj.X)));

    if (user != static_cast <XnUserID> (0)) break;
    //if the examined pixel was labeled as background(UserID 0), go to another pixel near to it

    // search algorithm for labeled pixels

    int chg =1+j/2;

    switch (j%4) {
      case 0: pEndProj.X += chg;
              break;
      case 1: pEndProj.Y += chg;
              break;
      case 2: pEndProj.X -= chg;
              break;
      case 3: pEndProj.Y -= chg;
```

```
          break;
  }

}

printf("Gesture recognized: %s performed by User: %d\n", strGesture, user);

// FACE DETECTION

if (user) {

  //transformation ratio projective -> real world
  const float transfr = 8.6834e-4;   //tan(FoVH/2)/640 ~ tan(FoVV/2)/480

  const XnDepthPixel* pDepth = dmd.Data();
  const XnLabel* pLabels = smd.Data();

  // Get region of interest
  int x=1, y=1, up=0, down=0, right=640, left=0, faceDepth;

  for (int i=0; i<480*640; i++, x++, pLabels++) {

    if(x==641) {  //New line of depth image
      x=1;
      y++;
    }

    if(!(*pLabels)) continue; //cut away background

    if(!(*(pDepth+i))) continue;  //disregard pixels with depth value zero

    if(!up) {
      //This is the topmost user pixel defining the vertical position of the roi
      up = y;
      faceDepth = (int) *(pDepth+i);
      //Only the upper 30 cm of the Body are taken into account
      down = y + (int)(300.0 / (*(pDepth+i)*2.0*transfr));
    }

    if( y == down) break; //bottom of roi reached

    if(right > x) right = x; // horizontal expansion of roi
    if(left < x)  left  = x;

  }

  //Depth points in roi
  XnPoint3D face[down-up][left - right + 1];

  for (int j = up; j<down; j++) {
    for (int i = right; i<=left; i++) {
      face[j-up][i-right].X = i;
      face[j-up][i-right].Y = j;
      face[j-up][i-right].Z = *(pDepth+(j-1)*640+i);
    }
  }

  //Smoothing (moving Window 7x7)

  float help[down-up][left - right + 1];

  for (int j = 3; j<down-up-3; j++) {
    for (int i = 3; i<left-right-2; i++) {
        float nom = 0;
        int den = 0;

        for (int jj = -3; jj<=3; jj++) {
          for (int ii = -3; ii<=3; ii++) {

            if (face[j+jj][i+ii].Z < .1) {
              help[j][i] = 0;
              goto mark1;  //Discard the pixel if only one pixel of window has depth value zero
            }

            nom += face[j+jj][i+ii].Z;
            den++;

          }
        }

        if (den) {
          help[j][i] = nom/den;
        }
mark1:  continue;

      }
    }

    for (int j = 1; j<down-up-1; j++) {
      for (int i = 1; i<left-right; i++) {
        face[j][i].Z=help[j][i];
      }
    }

    // get real world coordinates
    g_Depth.ConvertProjectiveToRealWorld((down-up)*(left-right +1), *face, *face);

    // Derivatives, central differential quotient

    //First Derivatives:

    int fin[2] = {down-up-2, left-right-1}; // array sizes

    float fx[fin[0]][fin[1]];
    float fy[fin[0]][fin[1]];

    //Second Derivatives:

    float fxx[fin[0]][fin[1]];

    for (int j = 1; j<=fin[0]; j++) {
      for (int i = 1; i<=fin[1]; i++) {

        int hx=face[j][i+1].X-face[j][i-1].X, hy = face[j+1][i].Y-face[j-1][i].Y;

        fx[j-1][i-1] = (face[j][i+1].Z-face[j][i-1].Z) / hx;
        fy[j-1][i-1] = (face[j+1][i].Z-face[j-1][i].Z) / hy;
        fxx[j-1][i-1] = (face[j][i+1].Z-2*face[j][i].Z+face[j][i-1].Z)/(0.25*hx*hx);

      }
    }

    // array for nose pixels
    bool N[fin[0]][fin[1]];

    for (int j = 0; j<=fin[0]; j++) {
      for (int i = 1; i<=fin[1]; i++) {
        if ((abs(fx[j][i])<0.6) && (abs(fy[j][i])<0.5) && (fxx[j][i]>0.1)) {
          N[j][i] = true;   // nose pixel
        } else {
          N[j][i] = false;  // no nose pixels
        }
      }
    }

    // Nose Check

    int Npix = (int) (60. * 1630. * 1630. / faceDepth / faceDepth);   //Nose Area
    // at depth 1630 the nose should have the area of 60 pixels
    int lxmax = (int) sqrt(Npix*0.4);  // max horizontal nose expansion

    int nosity[fin[0]][fin[1]][lxmax];
    //number of nose pixel in every possible nose rectangle

    for (int j=0; j<fin[0]; j++)  {
      for (int i=0; i<fin[1]; i++)  {
        for (int k=0; k<lxmax; k++)  {
          nosity[j][i][k] = 0;
        }
      }
    }

    for (int j = fin[0]/4; j<=2*fin[0]/3; j++)  {  // area to search for noses
      for (int i = fin[1]/3; i<=2*fin[1]/3; i++)  {

        if (N[j][i])  {
          // every rectangle that includes this nose pixel is incremented
          for (int lx=1; lx<=lxmax; lx++) {
            for (int u=0; u<=Npix/lx; u++) {
              nosity[j-u][i+1-lx][lx-1]++;
            }
          }
        }

      }
    }

    for (int j=0; j<fin[0]; j++)  {
      for (int i=0; i<fin[1]; i++)  {
        for (int k=1; k<lxmax; k++)  {   // nose has to be at least two pixels wide

          if (nosity[j][i][k]>=Npix*0.34) {
            printf("Nose at: x = %d, y = %d; lx = %d; nosity: %f\n",
                          right + i, up +j, k+1, nosity[j][i][k]/(float)Npix);
            ret = true;
          }

        }
      }
    }

    printf("User: %d\n", user);

    if (ret) {
      printf("FACE !!!\n");
    } else  {
      printf("NO FACE !!!\n");
    }

  }
  return ret;
}
```

A.4. Shadowing the user's hand

A.4.1. LC-file "kinectT44s.lc"

```
sample float handT44l[16];
sample float handT44r[16];
```

A.4.2. Client Program for Main Computer

```
#include <XnOpenNI.h>        //Needed for OpenNI
#include <XnCppWrapper.h>

#include <sys/socket.h>     //Needed for LabComm
extern "C" {
  #include <orca_client.h>
  #include "kinectT44s.h"
}

xn::Context g_Context;      //OpenNI objects
xn::DepthGenerator g_Depth;
xn::UserGenerator g_User;

XnBool g_bNeedPose = FALSE;
XnChar g_strPose[20] = "";

bool rel = true; //output distance from shoulder to hand as
//translation instead of absolute value

#define SAMPLE_XML_PATH "./SamplesConfig.xml"

#define CHECK_RC(nRetVal, what) \
  if (nRetVal != XN_STATUS_OK) \
  { \
    printf("%s failed: %s\n", what, xnGetStatusString(nRetVal));\
    return nRetVal; \
  }

// Callback: New user was detected
void XN_CALLBACK_TYPE User_NewUser(xn::UserGenerator& generator, XnUserID nId, void* pCookie) {
  printf("New User %d\n", nId);
  // New user found
  if (g_bNeedPose) {
    g_User.GetPoseDetectionCap().StartPoseDetection(g_strPose, nId);
  } else {
    g_User.GetSkeletonCap().RequestCalibration(nId, TRUE);
  }
}

// Callback: An existing user was lost
void XN_CALLBACK_TYPE User_LostUser(xn::UserGenerator& generator, XnUserID nId, void* pCookie) {
  printf("Lost user %d\n", nId);
}

// Callback: Detected a pose
void XN_CALLBACK_TYPE UserPose_PoseDetected(xn::PoseDetectionCapability& capability, const XnChar* strPose,
                                            XnUserID nId, void* pCookie) {
  printf("Pose %s detected for user %d\n", strPose, nId);
  g_User.GetPoseDetectionCap().StopPoseDetection(nId);
  g_User.GetSkeletonCap().RequestCalibration(nId, TRUE);
}

// Callback: Started calibration
void XN_CALLBACK_TYPE UserCalibration_CalibrationStart(xn::SkeletonCapability& capability, XnUserID nId,
                                                       void* pCookie) {
  printf("Calibration started for user %d\n", nId);
}

// Callback: Finished calibration
void XN_CALLBACK_TYPE UserCalibration_CalibrationEnd(xn::SkeletonCapability& capability, XnUserID nId,
XnBool bSuccess, void* pCookie) {
  if (bSuccess) {
    // Calibration succeeded
    printf("Calibration complete, start tracking user %d\n", nId);
    g_User.GetSkeletonCap().StartTracking(nId);
  } else {
    // Calibration failed
    printf("Calibration failed for user %d\n", nId);
    if (g_bNeedPose) {
      g_User.GetPoseDetectionCap().StartPoseDetection(g_strPose, nId);
    } else {
      g_User.GetSkeletonCap().RequestCalibration(nId, TRUE);
    }
  }
}

int main(int argc, char **argv) {

  XnStatus nRetVal = XN_STATUS_OK;

  // Initialize the Kinect
  xn::EnumerationErrors errors;
  nRetVal = g_Context.InitFromXmlFile(SAMPLE_XML_PATH, &errors);
  CHECK_RC(nRetVal, "Init From Xml-File");

  nRetVal = g_Context.FindExistingNode(XN_NODE_TYPE_DEPTH, g_Depth);
  CHECK_RC(nRetVal, "Find depth generator");
  nRetVal = g_Context.FindExistingNode(XN_NODE_TYPE_USER, g_User);
  if (nRetVal != XN_STATUS_OK) {
    nRetVal = g_User.Create(g_Context);
    CHECK_RC(nRetVal, "Find user generator");
  }

  XnCallbackHandle hUserCallbacks, hCalibrationCallbacks, hPoseCallbacks;
  if (!g_User.IsCapabilitySupported(XN_CAPABILITY_SKELETON)) {
    printf("Supplied user generator doesn’t support skeleton\n");
    return 1;
  }
  g_User.RegisterUserCallbacks(User_NewUser, User_LostUser, NULL, hUserCallbacks);
  g_User.GetSkeletonCap().RegisterCalibrationCallbacks(UserCalibration_CalibrationStart,
           UserCalibration_CalibrationEnd, NULL, hCalibrationCallbacks);

  if (g_User.GetSkeletonCap().NeedPoseForCalibration()) {
    g_bNeedPose = TRUE;
    if (!g_User.IsCapabilitySupported(XN_CAPABILITY_POSE_DETECTION)) {
      printf("Pose required, but not supported\n");
      return 1;
    }
    g_User.GetPoseDetectionCap().RegisterToPoseCallbacks(UserPose_PoseDetected, NULL, NULL, hPoseCallbacks);
    g_User.GetSkeletonCap().GetCalibrationPose(g_strPose);
  }

  g_User.GetSkeletonCap().SetSkeletonProfile(XN_SKEL_PROFILE_ALL);

  //Memory space for Homogeneous Transformation matrices of left and right arm
  float T44l[16];
  float T44r[16];

  for (int i=0; i<16; i++) {
    T44l[i] = 0;
    T44r[i] = 0;
  }

  //Initialize LabComm connection

  char* ip2Robot = argv[1]; // host name and port number are entered
  int portRobot = atoi(argv[2]); // as command line arguments

  struct orca_client *orca;
  orca = orca_client_new_tcp(ip2Robot, portRobot);

  if (!orca) {
    printf("Connection to host %s: port %d failed\n", ip2Robot ,portRobot);
    return 1;
  }

  char *channel[2];
  channel[0] = "handT44l";
  channel[1] = "handT44r";

  orca_client_selection_t selection2Rob;

  selection2Rob.n_0 = 2;
  selection2Rob.a = channel;

  orca_client_channel_t *signals2Rob;
  signals2Rob = orca_client_select_input_tcp(orca ,&selection2Rob);

  labcomm_encoder_register_kinectT44s_handT44l(signals2Rob->encoder);
  labcomm_encoder_register_kinectT44s_handT44r(signals2Rob->encoder);

  int count = 0;

  nRetVal = g_Context.StartGeneratingAll();
  CHECK_RC(nRetVal, "StartGenerating");

  printf("Initialization successful, starting main loop!\n");

  // Main Loop
  while (!xnOSWasKeyboardHit()) {
```

```
xn::SceneMetaData sceneMD;
xn::DepthMetaData depthMD;

g_Context.WaitOneUpdateAll(g_Depth); //Update the depth map

XnUserID aUsers[1]; // right now only one user is allowed
XnUInt16 nUsers = 1;
g_User.GetUsers(aUsers, nUsers);

for (int i = 0; i < nUsers; ++i) {
  if (g_User.GetSkeletonCap().IsTracking(aUsers[i])) {

    //Tracked Skeleton Joint Positions: left and right shoulder and hand
    XnSkeletonJointPosition jhl, jsl, jhr, jsr;

    //get positions as 3 element vectors
    g_User.GetSkeletonCap().GetSkeletonJointPosition(aUsers[i], XN_SKEL_LEFT_HAND, jhl);
    g_User.GetSkeletonCap().GetSkeletonJointPosition(aUsers[i], XN_SKEL_RIGHT_HAND, jhr);

    if (rel) {
      g_User.GetSkeletonCap().GetSkeletonJointPosition(aUsers[i], XN_SKEL_LEFT_SHOULDER, jsl);
      g_User.GetSkeletonCap().GetSkeletonJointPosition(aUsers[i], XN_SKEL_RIGHT_SHOULDER, jsr);
    }

    //Tracked Skeleton Joint Orientations: left and right elbow
    XnSkeletonJointOrientation jel, jer;
    XnSkeletonJointOrientation* pjel = &jel;
    XnSkeletonJointOrientation* pjer = &jer;

    //get the orientation as pointer to a 3x3 rotation matix
    g_User.GetSkeletonCap().GetSkeletonJointOrientation(aUsers[i], XN_SKEL_LEFT_ELBOW, *pjel);
    g_User.GetSkeletonCap().GetSkeletonJointOrientation(aUsers[i], XN_SKEL_RIGHT_ELBOW, *pjer);

    float* phelp[2];
    phelp[0] = &jel.orientation.elements[0];
    phelp[1] = &jer.orientation.elements[0];

    // Save the rotation matrices of the elbows in the Transformation matrices
    for (int j=0; j<12; j++) {
      if ((j+1)%4) {
        T44l[j] =  *(phelp[0]++);
        T44r[j] =  *(phelp[1]++);
      }
    }

    if (rel) {
      // save distance between shoulder and hand in transformation matrices
      T44l[3]  = jhl.position.X - jsl.position.X;
      T44l[7]  = jhl.position.Y - jsl.position.Y;
      T44l[11] = jhl.position.Z - jsl.position.Z;

      T44r[3]  = jhr.position.X - jsr.position.X;
      T44r[7]  = jhr.position.Y - jsr.position.Y;
      T44r[11] = jhr.position.Z - jsr.position.Z;

    } else {
      // save absolute position of hand in transformation matrices
      T44l[3]  = jhl.position.X;
      T44l[7]  = jhl.position.Y;
      T44l[11] = jhl.position.Z;

      T44r[3]  = jhr.position.X;
      T44r[7]  = jhr.position.Y;
      T44r[11] = jhr.position.Z;
```

```
        }

        //last line of a homogeneous transformation matrix
        T44l[12] = T44r[12] = 0;
        T44l[13] = T44r[13] = 0;
        T44l[14] = T44r[14] = 0;
        T44l[15] = T44r[15] = 1;

        // Confidence Coding
        if (jsl.fConfidence*jhl.fConfidence < 0.5) T44l[12] = -1;
        if (jsr.fConfidence*jhr.fConfidence < 0.5) T44r[12] = -1;

        if (jel.fConfidence< 0.5) T44l[13] = -1;
        if (jer.fConfidence< 0.5) T44r[13] = -1;
      }
    }

    //Type to be sent defined by .lc file
    kinectT44s_handT44l sendl;
    kinectT44s_handT44r sendr;

    for(int i=0; i<16; i++) {
      sendl.a[i] = T44l[i];
      sendr.a[i] = T44r[i];
    }

    // Send data via LabComm connection
    labcomm_encode_kinectT44s_handT44l(signals2Rob->encoder, &sendl);
    labcomm_encode_kinectT44s_handT44r(signals2Rob->encoder, &sendr);

    if (count==0) {
      //output every 10th frame to the console
      printf("\n %4.3f %4.3f %4.3f %4.0f      %4.3f %4.3f %4.3f %4.0f\n
                %4.3f %4.3f %4.3f %4.0f      %4.3f %4.3f %4.3f %4.0f\n
                %4.3f %4.3f %4.3f %4.0f      %4.3f %4.3f %4.3f %4.0f\n
                %4.3f %4.3f %4.3f %4.0f      %4.3f %4.3f %4.3f %4.0f\n",
            T44l[0] , T44l[1] , T44l[2] , T44l[3],     T44r[0] , T44r[1] , T44r[2] , T44r[3],
            T44l[4] , T44l[5] , T44l[6] , T44l[7],     T44r[4] , T44r[5] , T44r[6] , T44r[7],
            T44l[8] , T44l[9] , T44l[10], T44l[11],    T44r[8] , T44r[9] , T44r[10], T44r[11],
            T44l[12], T44l[13], T44l[14], T44l[15],    T44r[12], T44r[13], T44r[14], T44r[15]);
      count = 10;
    }
  count--;
  }
  g_Context.Shutdown();
}
```

A.4.3. Excerpt of Simulink Controller Model

Matlab source code of user defined function block “Decode Confidence Messages”:

```
function [y, st, sr] = decode(u)
%#eml

if (u(13) < -0.5)
```

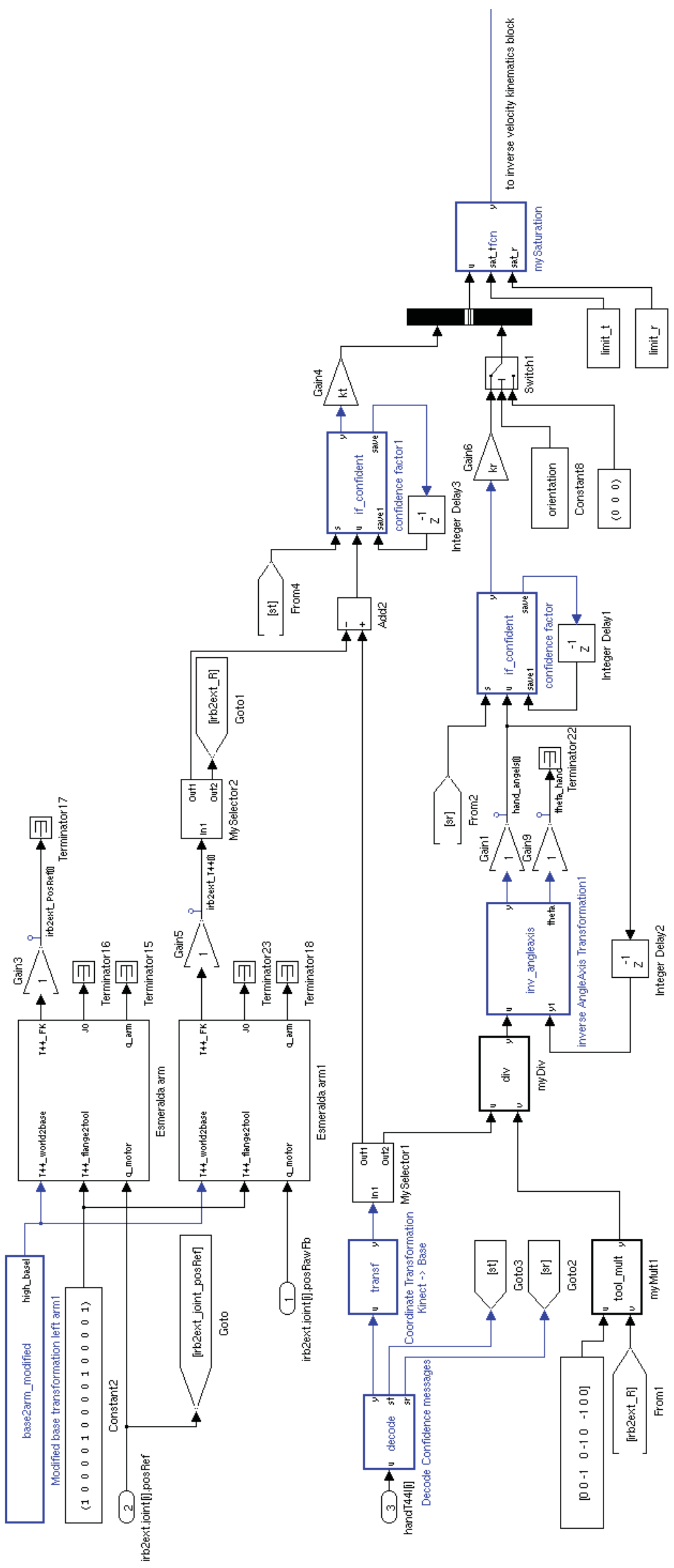
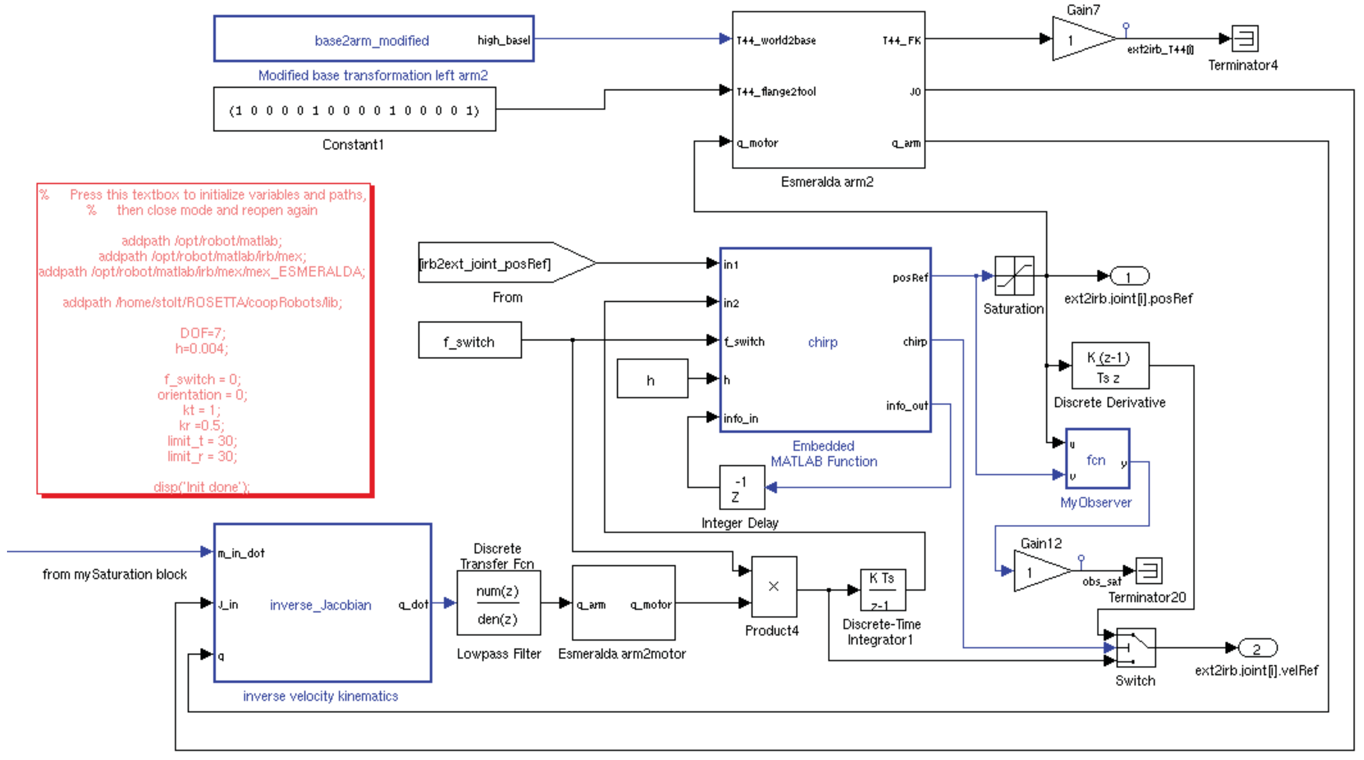


Figure A.5.: Simulink controller model for "Shadowing the user's hand" part1



```
%     Press this textbox to initialize variables and paths,
   %     then close mode and reopen again

         addpath /opt/robot/matlab;
       addpath /opt/robot/matlab/irb/mex;
addpath /opt/robot/matlab/irb/mex/mex_ESMERALDA;

   addpath /home/stolt/ROSETTA/coopRobots/lib;

               DOF=7;
               h=0.004;

            f_switch = 0;
          orientation = 0;
               kt = 1;
               kr =0.5;
             limit_t = 30;
             limit_r = 30;

          disp('Init done');
```

Figure A.6.: Simulink controller model for "Shadowing the user's hand" part2

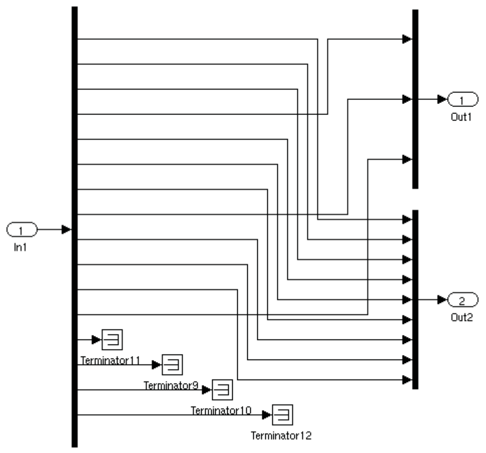


Figure A.7.: Simulink model for block "mySelector"

```
    st = 0;
    u(13) = 0;
else
    st = 1;
end

if (u(14) < -0.5)
    sr = 0;
    u(14) = 0;
else
    sr = 1;
end

y = u;
```

Matlab source code of user defined function block "Coordinate transformation Kinect -> Base":

```
function y = transf(u)
%#eml

T1 = [0 0 -1 0; -1 0 0 0; 0 1 0 0; 0 0 0 1];
T2 = eye(4);

for j=1:3
   for i=1:4
       T2(j,i) = u((j-1)*4+i);
   end;
end;

T2 = T1*T2;

y = [T2(1,:) T2(2,:) T2(3,:) T2(4,:)]';
```

Matlab source code of user defined function block “Inverse AngleAxis Transformation”:

```
function [y, theta]  = inv_angleaxis(u, y1)
%#eml

eps = 1e-6;

R = eye(3);
for j=1:3
   for i=1:3
       R(j,i) = u((j-1)*3+i);
    end;
end;

arg_acos = (R(1,1)+R(2,2)+R(3,3)-1)/2;

if (arg_acos < -0.98)   % This ensures, the absolute value of the factor theta/2/sin(theta)
    arg_acos = -0.98;   % cannot get greater than 8.3969
end

if (arg_acos > 1)
    arg_acos = 1;
end

theta = acos(arg_acos);

if (theta < eps)
    y = [0 0 0]’;
else
    y = theta/2/sin(theta)*[R(3,2) - R(2,3), R(1,3) - R(3,1), R(2,1) - R(1,2)]’;
end
```

Matlab source code of user defined function block “confidence factor”:

```
function [y, save] = if_confident(s, u, save1)
%#eml

if (s < 0.5)
    y = [0 0 0]’;
    save = 500;
elseif (save1 > 0)
    y = u/3;
    save = save1-1;
else
    y = u;
    save = -1;
end
```

Matlab source code of user defined function block “mySaturation”:

```
function y = fcn(u, sat_t, sat_r)
%#eml

% symmetric saturation limit is the second(translation) and third(rotation) input

dim = length(u);
help = zeros(1, dim);

for i=1:3
    if (u(i) < -sat_t)
        help(i) = -sat_t;
    elseif (u(i) > sat_t)
        help(i) = sat_t;
    else
        help(i) = u(i);
    end
end

for i=4:6
    if (u(i) < -sat_r)
        help(i) = -sat_r;
    elseif (u(i) > sat_r)
        help(i) = sat_r;
    else
        help(i) = u(i);
    end
end

y = help';
```

Matlab source code of user defined function block “inverse velocity kinematics”:

```
function q_dot  = inverse_Jacobian(m_in_dot, J_in, q)
%#eml

%extreme joint angles
q_max = [170.6 46 87.8 294.1 133.8 169.7 170.9]'*pi/180;
q_min = [-170.8 -145.6 -125.6 -290.4 -94.8 -189 -171]'*pi/180;

%middle joint angles
q_mid = (q_max + q_min)/2;

%half of joint range
q_span = (q_max - q_min)/2;

%Pseudoinverse Jacobian
J=zeros(6,7);

for j=1:6
    for i=1:7
        J(j,i) = J_in((j-1)*7+i);
    end;
end;
```

```
[U,S,V] = svd(J);

for i=1:6
    S(i,i) = S(i,i)^(-1);
end;

Jinv = V*S'*U';

%Joint velocities

b = (q_mid-q)./q_span;
b = sign(b).*b.^2;

b(1) = b(1)*1.5;
b(2) = b(2)*2;
b(5) = b(5)*2;
b(6) = b(6)*1.5;

q_dot = Jinv*m_in_dot + 2.0 * (eye(7)-Jinv*J)*b;
```

Matlab source code of user defined function block “chirp”:

```
function [posRef, chirp, info_out] = chirp(in1, in2, f_switch, h, info_in)
%#eml

time = info_in(1) + h;
last_chg = info_in(2);
last_state = info_in(3);

timespan = 4;

if (f_switch == last_state)
    info_out = [time, last_chg, last_state]';
    if (time - last_chg >= timespan)       % no chirp
        chirp = 0;
        if (f_switch < 0.5)
            posRef = in1;
        else
            posRef = in2;
        end
    else                                   % chirp in progress
        chirp = 1;
        arg = ((time - last_chg)/timespan - 1/2)*pi;
        val = (sin(arg) + 1)/2;

        if (f_switch < 0.5)
            posRef = in2 + val*(in1-in2);
        else
            posRef = in1 + val*(in2-in1);
        end
    end
else                                       % chirp just started
    info_out = [time, time-h, f_switch]';
    chirp = 1;
    arg = (h/timespan - 1/2)*pi;
    val = (sin(arg) + 1)/2;
    if (f_switch < 0.5)
        posRef = in2 + val*(in1-in2);
    else
        posRef = in1 + val*(in2-in1);
    end
end;
```

Matlab source code of user defined function block “MyObserver”:

```
function y = fcn(u,v)
%#eml
% observes, in which coordinates inputs u ans v differ

tol = 0.1;
dim = length(u);
out = 0;

for i=1:dim
    if (abs(u(i)-v(i)) > tol)
        out = out + i* 10^(dim-i);
    end
end

y = out;
```

A.5. HandRecorder for Dynamic Behaviour Experiment

A.5.1. LC-file “kinectxyz.lc”

```
sample float kinectxyzt[4];
```

A.5.2. Client Program for Main Computer

```
#include <XnOS.h>               //Needed for OpenNI
#include <XnCppWrapper.h>
```

```
#include <math.h>


#include <sys/socket.h>        //Needed for LabComm
extern "C" {
  #include <orca_client.h>
  #include "kinectxyz.h"
}

using namespace xn;

#define SAMPLE_XML_PATH "./SamplesConfig.xml"

Context g_context;             //OpenNI objects
DepthGenerator g_depth;
HandsGenerator g_hand;

XnPoint3D startp = xnCreatePoint3D(0.0f, 0.0f, 1000.0f);

struct orca_client *orca;      // for LabComm connection
orca_client_channel_t *signals2Rob;
kinectxyz_kinectxyzt kinectxyzt;
orca_client_selection_t selection2Rob;
char *channel = "kinectxyzt";
char **channel_p = &channel;


// Handcallbacks

void XN_CALLBACK_TYPE HandCreate(HandsGenerator& generator, XnUserID nId,
                                 const XnPoint3D* pPosition, XnFloat fTime, void* pCookie) {

  kinectxyzt.a[0] = pPosition->X;     // Send detected hand position to Simulink controller
  kinectxyzt.a[1] = pPosition->Y;
  kinectxyzt.a[2] = pPosition->Z;
  kinectxyzt.a[3] = fTime;

  labcomm_encode_kinectxyz_kinectxyzt(signals2Rob->encoder, &kinectxyzt);

  printf("%d %f %f %f %f\n", nId, kinectxyzt.a[0], kinectxyzt.a[1], kinectxyzt.a[2], kinectxyzt.a[3]);
}

void XN_CALLBACK_TYPE HandUpdate(HandsGenerator& generator, XnUserID nId,
                                 const XnPoint3D* pPosition, XnFloat fTime, void* pCookie) {

  kinectxyzt.a[0] = pPosition->X;     // Send detected hand position to Simulink controller
  kinectxyzt.a[1] = pPosition->Y;
  kinectxyzt.a[2] = pPosition->Z;
  kinectxyzt.a[3] = fTime;

  labcomm_encode_kinectxyz_kinectxyzt(signals2Rob->encoder, &kinectxyzt);

  printf("%d %f %f %f %f\n", nId, kinectxyzt.a[0], kinectxyzt.a[1], kinectxyzt.a[2], kinectxyzt.a[3]);
}

void XN_CALLBACK_TYPE HandDestroy(HandsGenerator& generator, XnUserID nId, XnFloat fTime, void* pCookie)
{
  printf("%d 0 0 0 %f\n", nId, fTime);
  g_hand.StartTracking(startp);    //Look for a new hand at the start point
}


int main(int argc, char* argv[]) {
  XnStatus rc;
```

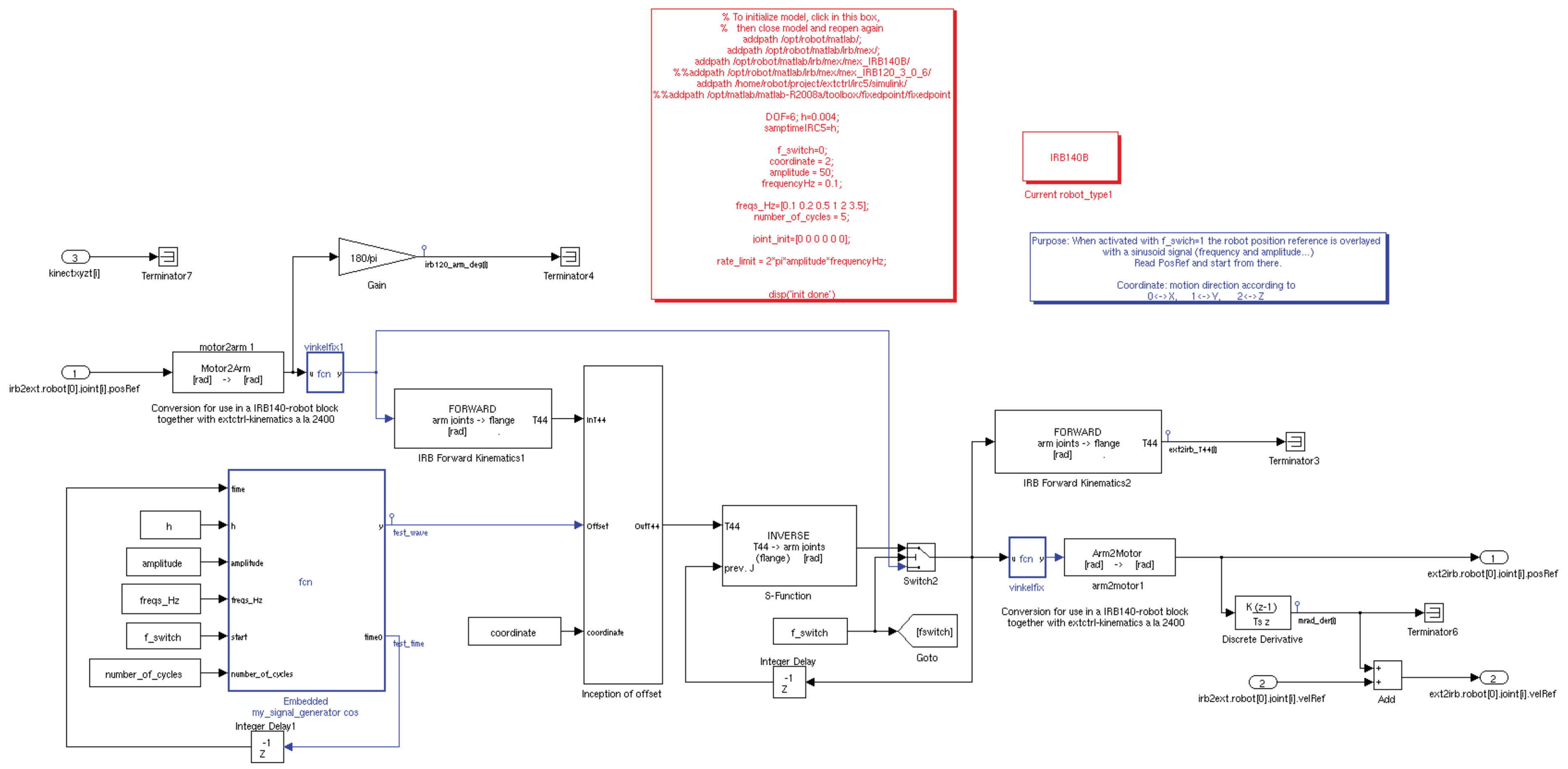
```
// Initialize the Kinect

EnumerationErrors errors;
rc = g_context.InitFromXmlFile(SAMPLE_XML_PATH, &errors);
if (rc == XN_STATUS_NO_NODE_PRESENT) {
 XnChar strError[1024];
  errors.ToString(strError, 1024);
  printf("%s\n", strError);
  return (rc);
} else if (rc != XN_STATUS_OK) {
  printf("Open failed: %s\n", xnGetStatusString(rc));
  return (rc);
}

rc = g_context.FindExistingNode(XN_NODE_TYPE_DEPTH, g_depth);

if (rc != XN_STATUS_OK) {
  printf("Finding existing node for depth failed: %s\n", xnGetStatusString(rc));
  return (rc);
}

rc = g_hand.Create(g_context);
rc = g_context.FindExistingNode(XN_NODE_TYPE_HANDS, g_hand);

if (rc != XN_STATUS_OK) {
  printf("Finding existing node for hands failed: %s\n", xnGetStatusString(rc));
  return (rc);
}

XnCallbackHandle(h);

rc = g_hand.RegisterHandCallbacks(HandCreate, HandUpdate, HandDestroy, NULL,h);

rc = g_context.StartGeneratingAll();

if (rc != XN_STATUS_OK) {
  printf("Starting Generation failed: %s\n", xnGetStatusString(rc));
  return (rc);
}

// Setting up Labcomm Connection

char* ip2Robot = argv[1];   // robot host and port are command line arguments
int portRobot = atoi(argv[2]);

orca = orca_client_new_tcp(ip2Robot, portRobot);

if (!orca) {
  printf("Connection to host %s: port %d failed\n", ip2Robot ,portRobot);
  return 1;
}

selection2Rob.n_0 = 1;
selection2Rob.a = channel_p;
signals2Rob = orca_client_select_input_tcp(orca ,&selection2Rob);

labcomm_encoder_register_kinectxyz_kinectxyzt(signals2Rob->encoder);

// Start the program loop

g_hand.StartTracking(startp);

while(!xnOSWasKeyboardHit()) {
  rc = g_context.WaitOneUpdateAll(g_depth);
```

```
    if (rc != XN_STATUS_OK) {
      printf("Failed updating data: %s\n", xnGetStatusString(rc));
    }
  }

  g_context.Shutdown();
}
```

A.5.3. Excerpt of Simulink Controller Model

Matlab source code of user defined function block “my_signal_generator”:

```
function [y,time0] = fcn(time,h,amplitude,freqs_Hz,start,number_of_cycles)
%#eml

number_fqs=length(freqs_Hz);
time_thres = number_of_cycles./freqs_Hz;
freq_select = 1;
internal_switch = 1;
if (start<0.5)
    time0=0;
else
    time0=time+h;

    time_thres = number_of_cycles./freqs_Hz;
    for i=number_fqs+1-(1:number_fqs)
        time_thres(i)=sum(time_thres(1:i));
    end;

    for i=2:number_fqs
        if((time0>time_thres(i-1))&&(time0<time_thres(i)))
            freq_select=i;
        end
    end

    if(time0>time_thres(number_fqs))
        internal_switch = 0;
    end
end

y = internal_switch*amplitude*(1-cos(2*pi*freqs_Hz(freq_select)*(time0-time_thres(freq_select))));
```



```
% To initialize model, click in this box,
%   then close model and reopen again
addpath /opt/robot/matlab/;
addpath /opt/robot/matlab/irb/mex/;
addpath /opt/robot/matlab/irb/mex/mex_IRB140B/
%%addpath /opt/robot/matlab/irb/mex/mex_IRB120_3_0_6/
addpath /home/robot/project/extctrl/irc5/simulink/
%%addpath /opt/matlab/matlab-R2008a/toolbox/fixedpoint/fixedpoint

DOF=6; h=0.004;
samptimeIRC5=h;

f_switch=0;
coordinate = 2;
amplitude = 50;
frequencyHz = 0.1;

freqs_Hz=[0.1 0.2 0.5 1 2 3.5];
number_of_cycles = 5;

joint_init=[0 0 0 0 0 0];

rate_limit = 2*pi*amplitude*frequencyHz;


disp('init done')
```

Purpose: When activated with f_swich=1 the robot position reference is overlayed with a sinusoid signal (frequency and amplitude...)
Read PosRef and start from there.

Coordinate: motion direction according to
0<->X, 1<->Y, 2<->Z

Figure A.8.: Simulink controller model for “Handtrack”

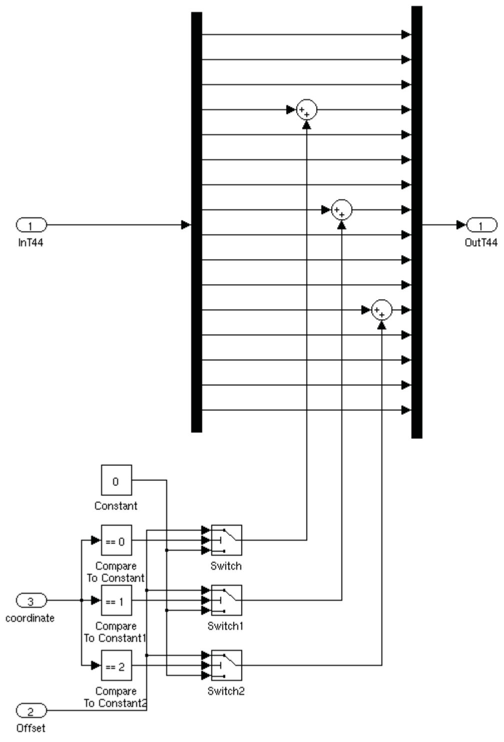


Figure A.9.: Simulink model for block "Inception of Offset"

B. Results from the Dynamic Behaviour Experiment

Period	Phase	computer (generated) / [s]	robot (measured) / [s]	Kinect (measured) / [s]
1	0°	13.224		
	90°	15.724	15.736	15.736
	180°	18.224	18.236	
	270°	20.724	20.736	20.808
2	0°	23.224	23.236	
	90°	25.724	25.736	25.78
	180°	28.224	28.236	
	270°	30.724	30.736	30.8
3	0°	33.224	33.236	
	90°	35.724	35.736	
	180°	38.224	38.236	
	270°	40.724	40.736	40.86
4	0°	43.224	43.236	
	90°	45.724	45.736	45.736
	180°	48.224	48.236	
	270°	50.724	50.736	50.8
5	0°	53.224	53.236	
	90°	55.724	55.736	55.736
	180°	58.224	58.236	
	270°	60.724	60.736	60.8

Table B.1.: Time points of measured signals for f = 0.1 Hz

Period	Phase	computer (generated) / [s]	robot (measured) / [s]	Kinect (measured) / [s]
1	0°	63.224	63.236	
	90°	64.474	64.486	64.528
	180°	65.724	65.736	65.77
	270°	66.974	66.986	67.103
2	0°	68.224	68.236	
	90°	69.474	69.486	69.52
	180°	70.724	70.736	
	270°	71.974	71.986	72.02
3	0°	73.224	73.236	73.34
	90°	74.474	74.486	74.51
	180°	75.724	75.736	
	270°	76.974	76.986	77.03
4	0°	78.224	78.236	
	90°	79.474	79.486	79.51
	180°	80.724	80.736	80.8
	270°	81.974	81.986	82.03
5	0°	83.224	83.236	83.25
	90°	84.474	84.486	84.51
	180°	85.724	85.736	85.8
	270°	86.974	86.986	87.06

Table B.2.: Time points of measured signals for f = 0.2 Hz

Period	Phase	computer (generated) / [s]	robot (measured) / [s]	Kinect (measured) / [s]
1	0°	88.224	88.236	
	90°	88.724	88.736	88.768
	180°	89.224	89.236	89.268
	270°	89.724	89.736	89.797
2	0°	90.224	90.236	90.3
	90°	90.724	90.736	90.765
	180°	91.224	91.236	91.284
	270°	91.724	91.736	91.77
3	0°	92.224	92.236	
	90°	92.724	92.736	92.76
	180°	93.224	93.236	
	270°	93.724	93.736	93.77
4	0°	94.224	94.236	
	90°	94.724	94.736	94.779
	180°	95.224	95.236	95.268
	270°	95.724	95.736	95.77
5	0°	96.224	96.236	96.324
	90°	96.724	96.736	96.765
	180°	97.224	97.236	
	270°	97.724	97.736	97.773

Table B.3.: Time points of measured signals for f = 0.5 Hz

Period	Phase	computer (generated) / [s]	robot (measured) / [s]	Kinect (measured) / [s]
1	0°	98.224	98.236	
	90°	98.474	98.486	98.523
	180°	98.724	98.736	98.772
	270°	98.974	98.986	99.018
2	0°	99.224	99.236	
	90°	99.474	99.486	99.525
	180°	99.724	99.736	
	270°	99.974	99.986	100.015
3	0°	100.224	100.236	100.269
	90°	100.474	100.486	100.528
	180°	100.724	100.736	100.789
	270°	100.974	100.986	101.021
4	0°	101.224	101.236	
	90°	101.474	101.486	101.532
	180°	101.724	101.736	101.784
	270°	101.974	101.986	102.036
5	0°	102.224	102.236	102.288
	90°	102.474	102.486	102.527
	180°	102.724	102.736	102.78
	270°	102.974	102.986	103.035

Table B.4.: Time points of measured signals for f = 1 Hz

Period	Phase	computer (generated) / [s]	robot (measured) / [s]	Kinect (measured) / [s]
1	0°	103.224	103.236	
	90°	103.349	103.361	103.405
	180°	103.474	103.486	103.536
	270°	103.599	103.611	103.698
2	0°	103.724	103.736	103.764
	90°	103.849	103.861	103.903
	180°	103.974	103.986	104.028
	270°	104.099	104.111	104.211
3	0°	104.224	104.236	104.268
	90°	104.349	104.361	104.411
	180°	104.474	104.486	104.532
	270°	104.599	104.611	104.7
4	0°	104.724	104.736	104.796
	90°	104.849	104.861	104.917
	180°	104.974	104.986	105.024
	270°	105.099	105.111	105.177
5	0°	105.224	105.236	105.288
	90°	105.349	105.361	105.407
	180°	105.474	105.486	105.528
	270°	105.599	105.611	105.691

Table B.5.: Time points of measured signals for f = 2 Hz

Period	Phase	computer (generated) / [s]	robot (measured) / [s]	Kinect (measured) / [s]
1	0°	105.724	105.736	
	90°	105.795	105.807	105.895
	180°	105.867	105.879	105.95
	270°	105.938	105.950	106.026
2	0°	106.010	106.022	106.056
	90°	106.081	106.093	106.156
	180°	106.153	106.165	
	270°	106.224	106.236	
3	0°	106.295	106.307	106.39
	90°	106.367	106.379	106.458
	180°	106.438	106.450	106.54
	270°	106.510	106.522	106.612
4	0°	106.581	106.593	106.645
	90°	106.653	106.665	106.725
	180°	106.724	106.736	106.8
	270°	106.795	106.82	106.878
5	0°	106.867	106.879	106.92
	90°	106.938	106.950	107.01
	180°	107.010	107.022	
	270°	107.081	107.093	107.178

Table B.6.: Time points of measured signals for f = 3.5 Hz

C. References

[1] *Microsoft-XBOX webpage* http://www.xbox.com/en-US/kinect [visited July 2011]

[2] Denis Störkle: *Documentation External Control - Extctrl,* not yet published, but soon available under http://www.control.lth.se/Publications.html (2011)

[3] *Primesense homepage* http://www.primesense.com/ [visited July 2011]

[4] Daniel Scharstein, Richard Szeliski: *High-Accuracy Stereo Depth Maps Using Structured Light,* IEEE Computer Society Conference on Computer Vision and Pattern Recognition, volume 1, pages 195-202, (June 2003)

[5] *OpenNI User Guide* online available under http://www.openni.org/images/stories/pdf/OpenNI_UserGuide_v3.pdf [visited July 2011]

[6] *ABB IRB120 data sheet* online available under http://www.abb.com/product/seitp327/be2eef38406eaca4c125762000319182.aspx [visited August 2011]

[7] *ABB IRB140 data sheet* online available under http://www.abb.com/product/seitp327/b3bdd2d1da6c8c07c12570c9003fe5eb.aspx [visited August 2011]

[8] Mark W. Spong, Seth Hutchinson, Mathukumalli Vidyasagar: *Robot Modeling and Control,* Wiley&Sons (2006)

[9] *ABB IRC5 data sheet* online available under http://www.abb.com/product/seitp327/f0cec80774b0b3c9c1256fda00409c2c.aspx [visited August 2011]

[10] Anders Blomdell, Isolde Dressler, Klas Nilsson and Anders Robertsson: *Flexible Application Development and High-performance Motion Control Based on External Sensing and Reconfiguration of ABB Industrial Robot Controllers,* ICRA Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications, Anchorage, Alaska, (June 2010)

[11] Calyampudi Radhakrishna Rao: *Linear statistical inference and its applications,* Wiley (1973)

[12] John H. Mathews and Kurtis K. Fink: *Numerical Methods Using Matlab, 4th Edition,* Prentice-Hall (2004)

[13] Alessandro Colombo, Claudio Cusano, Raimondo Schettini: *3D face detection using curvature analysis,* Pattern Recognition Society (2005)

[14] Volker Krüger, Dennis L. Heryog, Sanmohan Baby, Aleš Ude and Danica Kragic: *Learning Actions from Observations,* IEEE Robotics & Automation Magazine, volume 17, number 2, pages 30-43, (June 2010)