The Handtracking Robot
Project Report
Technical University of Munich

Nirnai Rao
Jens Petit
Oguz Can Oguz

February 2018

Contents

Abstract

This is the report for the final project of the course *Multi-Sensory Based Dynamic Robot Manipulation* offered at the Institute of Cognitive Science.

The project consists of an industrial robot which imitates the movement of the hand of a human operator. The hands position and movement is estimated using a Kinect sensor and an inertial measurement unit. The sensors information are fused with a Kalman filter.

The report starts with a brief outline of the project before explaining in depths the components. Chapter 2 deals with the robot model before introducing the robot controller. In the following chapter, the Kinect and the image processing framework is presented. Chapter 4 describes the inertial measurement unit and Kalman filter. Finally, the real world performance of the proposed setup is evaluated with measurements. A short conclusion summarizes the report.

Chapter 1

Project Description

1.1 Motivation

Natural ways to interact with robots are gaining more importance as robots start to leave their traditional "cages" in factories. Task which require human robot interaction are very likely to be the new normal in the future. Therefore, the goal of this project is to implement a natural way to control the end-effector of a six degree of freedom industrial robot.

1.2 Problem Formulation

The task seems very simple for a human: imitate the movement of another hand. Implementing this in an industrial robot however, requires a lot of advanced knowledge. The problem can be separated in two tasks: first, estimate the hand position and movement. Second, control the end-effector pose to the desired position and track the given trajectory.

In this project we use an Inertial Measurement Unit (IMU) to measure the orientation as well as accelerations of the hand. A Microsoft Kinect consisting of an RGB-D camera gives the position. Subsequently, the information are fused and smoothened with a Kalman Filter. In Figure 1.1, the use case is illustrated. The information flow is shown in project Figure 1.2 where q represents the joint states, τ the control torque and x the pose of the hand with the subscript d representing the desired state. To control the six degrees of freedom of the robot, a model-based adaptive operational space controller is used. It is able to estimate some unknown dynamic parameters of the robot.

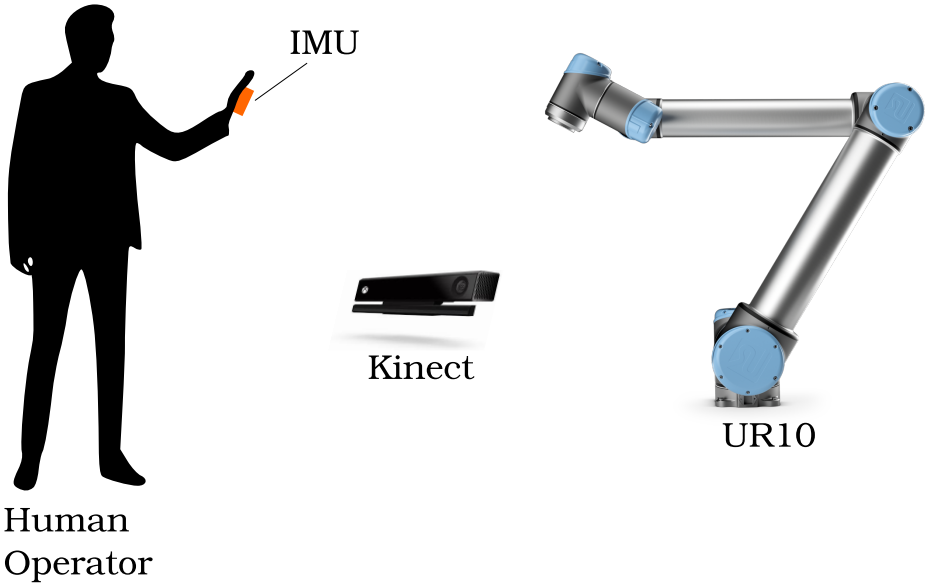

Figure 1.1: Exemplary use case.

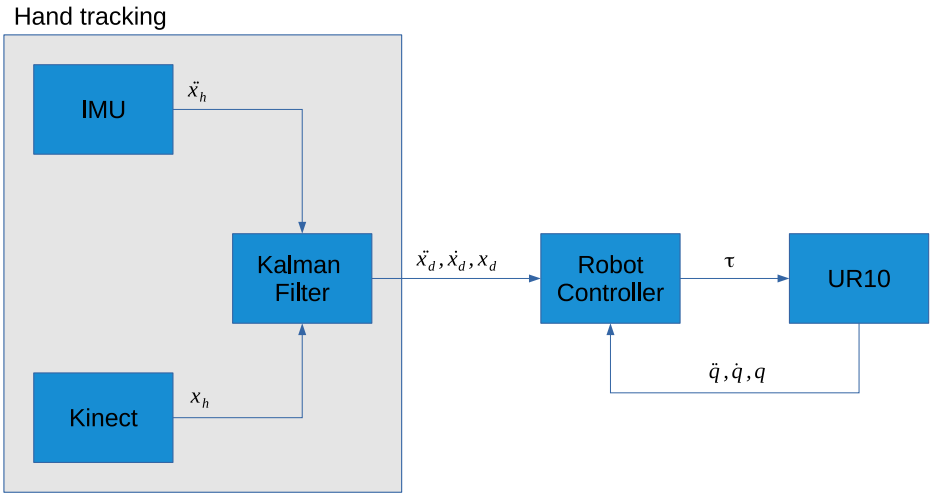

Figure 1.2: Information flow between the components.

Chapter 2

Robot Model and Controller

Responsible for this chapter is Oguz Can Oguz. Our method of control is a model based reference trajectory PD torque control, to which the prerequisite is to have the model of the robot at hand. Therefore, the robot modeling and controller are highly unrelated. The robot has to be modeled as accurately as possible for the controller to be stable, while the inaccuracies of the model has to be compensated by the controller.

2.1 Robot Model

2.1.1 Kinematic Model

The robot we have is an UR10 model from Universal Robots. The robot has 6 rotational joints which will be used to control the position and orientation of the end effector. In the project, we modeled the robot based on the parameters available on the official site of the Universal Robots, with few modifications 2.1.

Our modifications consist of defining the $x - axis$ of all links starting from x_2 in the opposite direction and adding a 90° offset to both second and fourth joints, so the starting position with all joint angle being equal to zeros would give us the default position of the real robot. However the real robot in the lab, was defined without the offsets in the DH table and was given offsets later on, so we had to change our angle inputs with regards to said offsets in the controller. In the end our DH table has the form in 2.1.

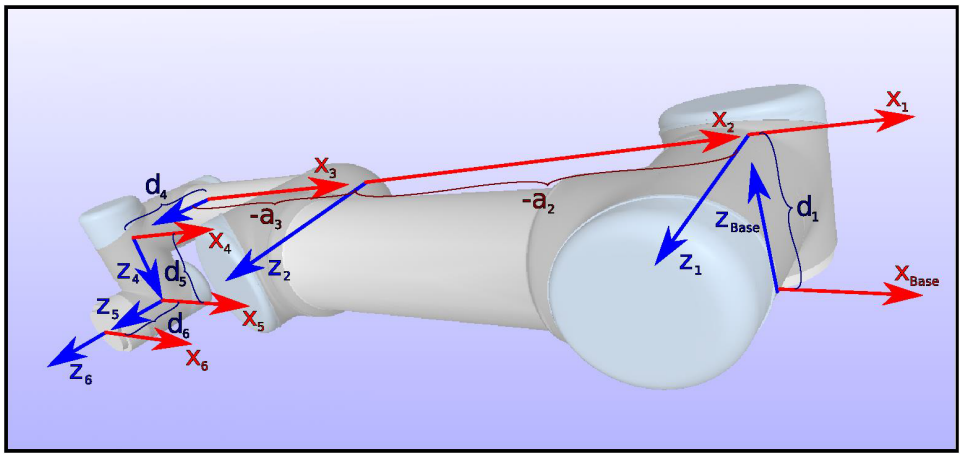

Figure 2.1: Official DH Model

i	θ	d	a	α
1	q_1	L_1	0	$\frac{\pi}{2}$
2	$q_2 + \frac{\pi}{2}$	0	L_3	0
3	q_3	0	$L_5 - L_{11}$	0
4	$q_4 - \frac{\pi}{2}$	L_2	0	$-\frac{\pi}{2}$
5	q_5	L_{11}	0	$\frac{\pi}{2}$
6	q_6	$L_4 + L_{12}$	0	0

Table 2.1: DH-table for the robot.

2.1.2 Dynamic Model

In the previous part we explained about the DH model required for kinematic computations and control. However, we are controlling the robot by directly specifying the torque commands to be applied to the robot joints and to control the robot successfully we need a dynamical model as well. There are a few extra requirements for the dynamical model. In the kinematic model, we only needed the lengths of the joints and the angle offsets. In the dynamical modeling, we require the positions for center of masses w.r.t. the kinematic model, the masses, the inertial components and the gravity vector. Unfortunately, there is not a way to correctly obtain all of them. The center of mass positions and masses are declared to the public by Universal Robots and we can determine the gravity vector but the inertial components are

missing. On top of that, the robot we are using is modified by a gripper on the end effector, which also changes the dynamical model. These inaccuracies have to be addressed in the controller design.

The robot kinematic parameters for the center of masses are defined as the table 2.2, where L_12 is defined as the end effector gripper length. The masses are directly taken from the Universal Robot website.

i	θ	d	a	α
1	q_1	L_6	0	0
2	$q_2 + \frac{\pi}{2}$	L_7	L_8	0
3	q_3	L_9	L_10	0
4	q_4	$\frac{L_2}{2}$	0	0
5	q_5	$\frac{L_{11}}{2}$	0	0
6	q_6	$\frac{(L_4+L_{12})}{2}$	0	0

Table 2.2: DH-table for the robot.

2.2 Robot Controller

Our project description was to accurately mimic hand movements with the robot. In order to achieve that there are two parts that need to be done uncorrelated to each other. The trajectory of the hand has to be estimated and the estimated trajectory has to be followed by the robot. The controller is responsible for having the robot follow the trajectory supplied by the trajectory estimator, which is in our case a kalman filter. In this section, we are going to assume that the estimation is accurate and any problems from the sensors are filtered and compensated beforehand, providing the controller with an accurate desired trajectory.

2.2.1 Controller Fundamentals

Given the dynamic model of our robot,

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) \tag{2.1}$$

where M, C and G are the Inertia, Coriolis and Gravity matrices respectively we are trying to find the correct τ to track our desired trajectory. Note that the Coriolis matrices for a dynamical system are not unique, but

in equation 2.1, we want the unique $C(q, \dot{q})$ that validates the following equation:

$$N = \dot{M}(q) - 2C(q, \dot{q}) \tag{2.2}$$

Above, N is defined as a skew-symettric matrix, and there is only one Coriolis matrix that fulfills that equation. We can also express the dynamical equation in the following way,

$$Y(q, \dot{q}, \ddot{q})\Theta = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) \tag{2.3}$$

where Y is the Regressor that expresses the joint angle, velocity and acceleration relations and $Theta$ are the parameters not depending on the angles. Now assuming that we have a reference trajectory velocity $\dot{q}_r$ and acceleration $\ddot{q}_r$, we can build the error model with reference Regressor as following:

$$\begin{aligned} M(q)\ddot{q}_r + C(q, \dot{q})\dot{q}_r + G(q) &= Y_r(q, \dot{q}, \dot{q}_r, \ddot{q}_r)\Theta & (2.4)\\ M(q)(\ddot{q} - \ddot{q}_r) + C(q, \dot{q})(\dot{q} - \dot{q}_r) &= \tau - Y_r\Theta & (2.5)\\ Sq &= \dot{q} - \dot{q}_r & (2.6)\\ M(q)\dot{S}q + C(q, \dot{q})\dot{S}q &= \tau - Y_r\Theta & (2.7) \end{aligned}$$

The reference Regressor $Y_r\Theta$ compensates for the torques that are produced by our robot model, generated from following the reference trajectory. So we only need to concern ourselves with finding an appriopriate τ that reduces the remaining error from the reference trajectory at each time step. With passivity based Lyapunov analysis, it can be shown that using a K_d gain on the error, while designing the reference trajectory for PD control is minimizes the error at each step and is suitable for trajectory tracking.

$$\begin{aligned} \tau &= -K_dSq + Y_r\Theta & (2.8)\\ Sq &= \dot{q} - \dot{q}_r) & (2.9)\\ \dot{q}_r &= \dot{q}_{des} - K_p\Delta q & (2.10)\\ \ddot{q}_r &= \ddot{q}_{des} - K_p\Delta\dot{q} & (2.11)\\ \Delta q &= q - q_{des} & (2.12)\\ \Delta\dot{q} &= \dot{q} - \dot{q}_{des} & (2.13) \end{aligned}$$

The controller defined with the equations above is a starting template and has to be modified for the specifics of our task and model.

2.2.2 Controller Modifications

The first problem is that, our desired trajectories are generated in the operational space, whereas our controller was designed for desired trajectories in joint space. We can try to translate the desired trajectory into joint space using inverse kinematics but that is not appropriate for cases where the inverse kinematics does not have one solution. On the other hand, we can try to define the trajectory error in the operational space using the forward kinematics, as forward kinematics always has one solution. However, we need to change our controller to work in the operational space. Defining the controller in operational space is actually straightforward. We have to define the errors in operational space first.

$$
\begin{aligned}
Sx &= \dot{x} - \dot{x}_r && (2.14)\\
\dot{x}_r &= \dot{x}_{des} - K_p \Delta x && (2.15)\\
\ddot{x}_r &= \ddot{x}_{des} - K_p \Delta \dot{x} && (2.16)\\
\Delta x &= x - x_{des} && (2.17)\\
\Delta \dot{x} &= \dot{x} - \dot{x}_{des} && (2.18)
\end{aligned}
$$

Then we need to find the transformation from the operational space velocities and joint space velocities, that is defined by the generalized pseudoinverse of the Jacobian J^+.

$$
\begin{aligned}
Sq &= J^+ Sx && (2.19)\\
\dot{q}_r &= J^+ \dot{x}_r && (2.20)\\
\ddot{q}_r &= J^+ \ddot{x}_r + \frac{(J^+)}{dt} \dot{x}_r && (2.21)\\
\frac{(J^+)}{dt} &= -J^+ \frac{(J)}{dt} J^+ && (2.22)
\end{aligned}
$$

The next problem is that we want to have our desired orientation given in euler angles, but the geometric Jacobian does not describe the time derivative of our forward kinematics anymore. So we have to find the derivative of our forward kinematics, which is the analytical Jacobian which we will denote as J_a.

$$J_a(q, \Phi) = T_A^{-1}(\Phi) J(q) \qquad (2.23)$$

In the equation above, $T_A(\Phi)$ describes the transformation from the change in pose defined by the euler angles to linear and angular velocity. Note that there is a different $T_A(\Phi)$ for each euler angle convention.

As the third problem, we have to compensate for the inaccuracies of our dynamic model. The solution is to use adaptive control, which modifies the Θ of the Regressor and changes it with an estimate $\hat{\Theta}$. The theta estimate changes during the movement, to so that it compensates for the reference trajectory torque.

$$\tau = -K_d Sq + Y_r \hat{\Theta} \tag{2.24}$$

$$\dot{\hat{\Theta}} = -\Gamma^{-1} Y_r^T Sq \tag{2.25}$$

$$\tag{2.26}$$

The variable Γ in 2.26, is a squared weighting matrix to be tuned.

Now we have a working operational space controller, but we have 6 Degrees of Freedom(Dof) for controlling in 6 dimensional operation space, which is sufficient if we linearly independent joint configurations. Unfortunately it is not the case as joint configurations frequently become linearly dependent which are called singular configurations or singularities, causing the determinant of Jacobian to be zero making it non invertible.

With only position control, it is feasible to create a manipulability map and only work in the regions with high manipulability however that manipulability mapping is not so intuitive with pose control and it is not feasible to expect the desired trajectory to only be in regions with high manipulability.

To overcome this difficulty, we have to first understands what happens to the controller gets close to a singularity. As the determinant of our Jacobian goes to zero, the determinant of the inverse goes to infinity. So while transforming the operational space errors to joint space errors using the inverse Jacobian, we are setting joint space errors very high, which in turn makes our torques very high. We would like to limit this behaviour and dampen our joint space errors near singularities, so that we move very little in the axis with little manipulability. In order to achieve this, we used a method called "Damped Inverse Jacobian", which changes the generalized pseudo-inverse of our Jacobian J^+ to $J^+_{a\lambda}$.

$$J_a^+ \rightarrow J_{a\lambda}^+ = J^T (JJ^T + \lambda I)^{-1} \tag{2.27}$$

This damped pseudo-inverse minimizes the expression $\|Jq - x\| + \lambda\|q\|$. This method does not solve the problem when we are in a singular configuration or when the desired pose is a singular configuration, but it helps to avoid getting to those singular configurations, at the cost of introducing inaccuracies into our tracking. The lambda should be chosen small enough to no affect the performance away from the singularities but big enough to dampen our velocity so that we do not reach the singular configuration. For our task, we found a 0.2 a suitable parameter for λ.

The one remaining problem is that derivating the term lambda took to much computational power and time. So we made an approximation based on the definition. If we are away from a singular configuration, the damped pseudo-inverse is almost the same as the normal pseudo-inverse, so we used that instead. Near singular configurations, the approximation no longer holds, but the acceleration should be very small and the effect of the change in Jacobi on acceleration should be even smaller so we did not account for that term.

To summarize the final version of the controller is defined as following:

$$\begin{aligned} \tau &= -K_d Sq + Y_r \hat{\Theta} && (2.28)\\ Sq &= J_{a\lambda}^{+} Sx && (2.29)\\ \dot{q}_r &= J_{a\lambda}^{+} \dot{x}_r && (2.30)\\ \ddot{q}_r &= J_{a\lambda}^{+} \ddot{x}_r - (J_a^{+} \frac{(J_a)}{dt} J_a^{+} \dot{x}_r) && (2.31) \end{aligned}$$

Chapter 3

Kinect and Image Processing

Responsible for this chapter is Nirnai Rao.

3.1 Overview Kinect

The Microsoft Kinect 2 consists of multiple sensors. For our purposes we will be only focusing on the RGB Color Camera and Depth Sensor. The Color Camera has a resolution of *1930 x 1080* and runs at 30Hz (fps). The Depth Camera has a resolution of *512 x 424*, a minimal depth of 50cm and a maximal depth of 4.5m. The difference in resolution doesn't allow for a direct pixel mapping from color image to depth image. A process called registration, is necessary to allow for such a mapping.

Since this project is developed on the Robot-Operating-System (ROS) framework, it is necessary to have some sort of communication between ROS and the Kinect drivers.

Both features, namely the communication with ROS and registration of the two cameras have already been implemented by an open source project called IAI-Kinect2 [5], which can be found on GitHub. IAI-Kinect2 is a ROS-Package, which is divided into three sub packages.

The main package is the bridge package 3.1, which establishes a connection to the kinect driver and publishes the data as ROS-Topics. This Node only is very power and bandwidth efficient, and thus only publishes data, when a client trys to subscribe any of the topics published by it. To run the package, use following command:

```
1    $ roslaunch kinect2_bridge kinect2_bridge.launch
```

After doing so, topics are publshed in three qualities. HD, Quarter-HD and SD. The HD topics are called as follows:

```
iai_kinect2
├── iai_kinect2
├── kinect2_bridge
├── kinect2_calibration
├── kinect2_registration
└── kinect2_viewer
```

Figure 3.1: IAI-Kinect2 directory structure

```
1   /kinect2/hd/camera_info
2   /kinect2/hd/image_color
3   /kinect2/hd/image_color/compressed
4   /kinect2/hd/image_color_rect
5   /kinect2/hd/image_color_rect/compressed
6   /kinect2/hd/image_depth_rect
7   /kinect2/hd/image_depth_rect/compressed
8   /kinect2/hd/image_mono
9   /kinect2/hd/image_mono/compressed
10  /kinect2/hd/image_mono_rect
11  /kinect2/hd/image_mono_rect/compressed
12  /kinect2/hd/points
```

The two topics `/kinect2/hd/image_color_rect` and `/kinect2/hd/image_depth_rect` are automatically registered in the `/kinect2/hd/points` topic, thus these are the three data stream used in this project.

3.2 Color Detection

A easy to implement approach for object detection in a color image, is to track the color of the object. It would also be feasible to track the shape or find the object directly in the depth image. Both of these options are either less robust or have unreasonable complexity for this task. To find the object via its color in the image, the open source library OpenCV was used [1]. This library makes it easy to manipulate images and run well studied computer vision algorithms. To filter color in the RGB space, in which the images lie after revieving them, can be done, but is tedious. It would be necessary to find all RGB combinations, which span a desired range of color and then filter each of the three channels. The task gets much easier after converting the image into the HSV (Hue-Saturation-Value) space. The Hue of a color is a unique value, where as the other two values describe intensity and saturation. Thus it is only necessary filter a range of hue values. Median

Figure 3.2: RGB-Image

Figure 3.3: Filtered Image

Figure 3.4: Color Detection

filtering is necessary as next step to anly find object where this color is dominant. After these two filtering steps we are left with a binary image, where the desired regions are white and the rest is black 3.3.

3.3 Image Position Estimate

The result of the previous step give as a region of pixels, where the object might be. To get an good position estimate it is import which part of the region is chosen, to look up in the pointcloud. For that an we want to detect the boundary of the found region and fit a shape to it, which resembles the object the most. In our case this would be a rectangle, as you can see in 3.2. For detecting the edges any edge detector in OpenCV can be use. In our case we went for the canny edge detector. After finding the edges, we can find a contour of the object and then fit a rectangle closest to this contour. All of these steps are necessary so that we can pin down one desired pixel, where we are sure, the object is in.

3.4 Cartesian Position Estimate

From the rectangle the easiest approach is to find the center point and lookup the catesian coordinates at that pixel. This is error prone though since the fitted rectangle doesn't necessary cover the object in the center. This would lead to a large error in the z coordinate, if the IR ray did not hit the object at that position. To reduce the probability to have bad z readings, we decided to look at all pixels in the rectangle and took the median of that value. This helped us eliminate outliers and gave us much more robust estimate. After experimental validation this method worked quite well, but was still jumpy in a small radius around the object itself. To reduce this high frequency noise, we introduced a low pass filter. One very good low pass filter is the exponential moving average. It is a infinite impulse response (iir) filter, which makes it possible to implement the filter without having a data buffer. The general idea of this filter is to weight the data with an exponential function and then average over the weighted datapoints.

Tunable parameters are the decay rate of the exponential function and the window length after which to cut of the window. A nice property of this filter is that it can be boiled down to one tuning parameter when using a different formula to calculate the filtered value.

$$\hat{x}(t) = \alpha \cdot \hat{x}(t-1) + (\alpha - 1) \cdot x_{meas} \tag{3.1}$$

Equation 3.1 leaves us with α as tunable parameter. α incorporates the information of window length and decay rate. Large α for example means a fast decay, but a long window length, whereas small alphas mean the opposite. Thus larger α means more filtering. This also introduces a delay

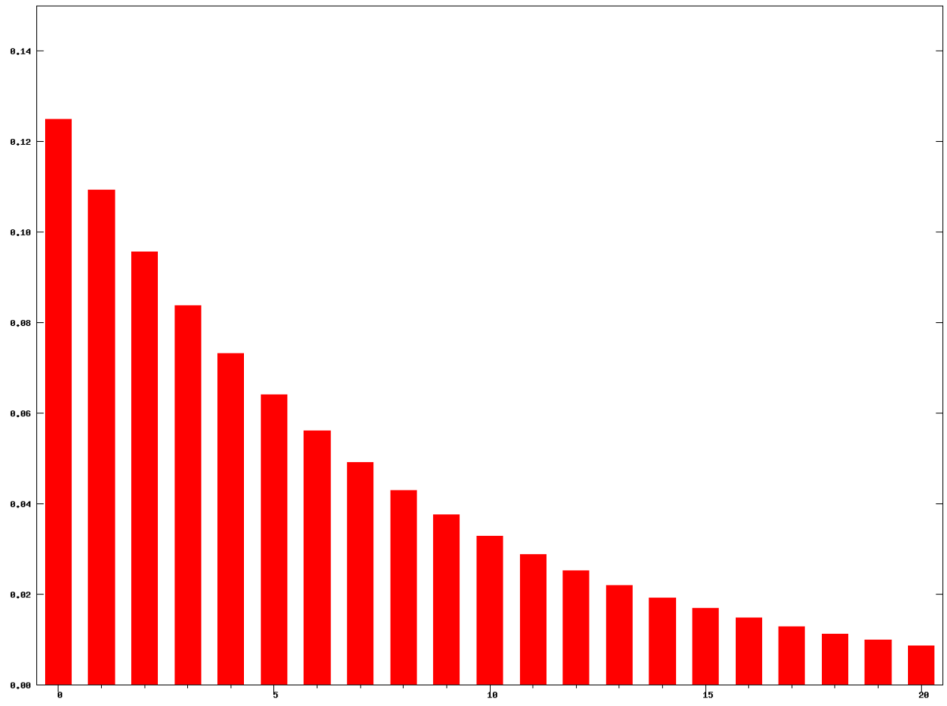

Figure 3.5: Exponential weights

which is critical for this application. If the delay of the position measurement gets to large, it might cause a lot of harm. The Kalman-Filter fuses multiple measurements together and trys to come up with the best estimate while considering all information. If the datapoints don't match sequencially anymore, this will introduce unexpected behavior. When all parameters are tuned right though, this process gives a very smooth position estimate.

Chapter 4

Inertial Measurment Unit and Sensor Fusion

Responsible for this chapter is Jens Petit.

4.1 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) consists of a combination of accelerometers, gyroscopes and magnetometers with the purpose of estimating an objects orientation as well as its relative change in pose. In this project the device *MTi-30 AHRS* from the manufacturer *Xsens* is used. Using the earth's magnetic field as a reference, it outputs drift free roll, pitch and yaw orientation plus linear accelerations as well as angular velocities [2].

For establishing the communication between ROS and the sensor, there exists the package `xsens_driver` [6]. It connects the sensor in such a way that it publishes a rostopic called `/imu/data` with its data and a frequency of around 200 Hz. Furthermore, the proprietary software of the device manufacture, the so-called *mtmanager* allows for a quick graphical interpretation of the sensor's data.

The orientation of the sensor is given with respect to the so-called ENU coordinate system, which x-axis point to the east, z-axis up and y-axis to the north. Also the linear accelerations and angular velocities are given in this coordinate system.

4.1.1 Calibration of IMU

To transform the information of the IMU in the robot base frame, the transformation between the ENU frame and the base has to be known. To solve this issue, the IMUs coordinate system can be overlayed with the robot base system and this orientation be saved as the transformation between the ENU system and the robot's base.

4.2 Kalman Filter

The controller needs as its input a desired state consisting of a pose in cartesian space and its two time derivatives. To estimate all these states, filtering noise and fusing the kinect position data with the IMU data, a Kalman Filter is used. The concept of this filter was introduced by [3] and has gained widespread application in real-time systems. The notation for this chapter is taken from [4].

4.2.1 General description

In general, the Kalman filter aims to model a linear process which is governed by the law

$$x_k = Ax_{k-1} + w_{k-1} \tag{4.1}$$

where x_k models a state at timestep k, A is the state transition matrix relating the state of the previous timestep to the next and w_k represents the process noise which is normally distributed

$$p(w) \sim \mathcal{N}(0, Q)\,. \tag{4.2}$$

with the process noise covariance matrix Q. The measurement is expressed as

$$z_k = Hx_k + v_k \tag{4.3}$$

with the matrix H relating the states of the process to the measurement space z_k and v_k being the measurement noise which is also gaussian distributed

$$p(v) \sim \mathcal{N}(0, R) \tag{4.4}$$

where R is the measurement noise covariance matrix.

4.2.2 Process and measurement model

The states for one cartesian coordinate direction we aim to model are the position s_k, velocity v_k, acceleration a_k and an acceleration offset Ω_k which occurs due to the fact that is impossible to remove perfectly the earths gravity from the linear acceleration measurement. The complete process model is then given as

$$\begin{bmatrix} s_k \\ v_k \\ a_k \\ \Omega_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 & 0 \\ 0 & 1 & \Delta t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_{k-1} \\ v_{k-1} \\ a_{k-1} \\ \Omega_{k-1} \end{bmatrix} + w_{k-1}. \tag{4.5}$$

which assumes a movement with constant acceleration. As we want to able to follow arbitrary trajectories of the human hand, the model needs more flexibility. This is given through the noise term

$$w_k = \begin{bmatrix} \Delta t^2/2 \\ \Delta t \\ 1 \\ 0 \end{bmatrix} a_k, \quad a_k \sim \mathcal{N}(0, \sigma_a) \tag{4.6}$$

which assumes that we induce normally distributed acceleration into our model with the accelerations variance σ_a^2 as a parameter. Consequently, the process noise covariance matrix is given as

$$E[w_k w_k^T] = Q = \begin{bmatrix} \Delta t^4/4 & \Delta t^3/2 & \Delta t^2/2 & 0 \\ \Delta t^3/2 & \Delta t^2 & \Delta t & 0 \\ \Delta t^2/2 & \Delta t & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \sigma_a^2. \tag{4.7}$$

For relating the states with the measurement after (4.3), we use

$$\begin{bmatrix} s_k^m \\ a_k^m \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} x_k + v_k. \tag{4.8}$$

It results from the fact that the measured position s_k^m is directly given through the Kinect and it corresponds to our state s_k. For the acceleration we are able to measure the acceleration a_k^m which corresponds to the real acceleration plus an offset as

$$a_k^m = a_k + \Omega_k. \tag{4.9}$$

Also, the measurements contain noise which is modelled through the individual variances of the sensors

$$R = \begin{bmatrix} \sigma^2_{\text{kin}} & 0 \\ 0 & \sigma^2_{\text{imu}} \end{bmatrix}. \tag{4.10}$$

4.2.3 State estimation

Having described the process to model and how it relates to the sensor readings, we can address the state estimation process. The state estimation of the Kalman filter consists of two steps. First, the update step. It projects the old state estimate ahead using the model assumptions given through the transition matrix A and the process noise covariance Q. The equations are given as

$$\hat{x}_k^- = A\hat{x}_{k-1} \tag{4.11}$$

$$P_k^- = AP_{k-1}A^T + Q \tag{4.12}$$

where $\hat{x}_k$ denotes the estimated state and P_k the estimated covariance. The minus signs indicates that these are the estimates after the predict step which have not yet been updated through the sensor measurements.

In the second step, the results from the predict step are updated through the measurements. Consequently, it is called the update step with the three calculations

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \tag{4.13}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \tag{4.14}$$

$$P_k = (I - K_k H)P_k^- \tag{4.15}$$

where K_k is the Kalman gain. The predict and update step are repeated in a loop to give continuous state estimations for the controller.

4.2.4 Angular velocities and acceleration

A very similar approach as for the linear state estimates is followed for the angular velocities and accelerations. However, here the orientation is directly send to the robot because it is already preprocessed in the IMU with advanced filtering techniques.

Chapter 5

Conclusion

The project report gives a short overview of the projects three main components: the robot model and controller, the image processing workflow and finally the sensor fusion with the Kalman filter. Key topics as well as problems are discussed and solutions presented with their mathematical foundation.

An extension to the project would be to make it more robust and even more dynamical. Also, instead of tracking a coloured object, tracking only the hand of the human operator could be interesting in the future. The point cloud of the Kinect gives enough information to actually estimate the hand pose.

References

Literature

[1] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000) (cit. on p. 13).

[2] Xsens Technologies B.V. *MTi 10-series and MTi 100-series*. Version MT0605P. Oct. 2013. URL: https://www.scribd.com/document/326266454/MTi-Usermanual (cit. on p. 17).

[3] R. E. Kalman. "A New Approach to Linear Filtering And Prediction Problems". In: *ASME Journal of Basic Engineering* (1960) (cit. on p. 18).

[4] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. Tech. rep. Chapel Hill, NC, USA, 1995 (cit. on p. 18).

[5] Thiemo Wiedemeyer. *IAI Kinect2*. https://github.com/code-iai/iai_kinect2. Accessed June 12, 2015. University Bremen: Institute for Artificial Intelligence, 2014 – 2015 (cit. on p. 12).

Online sources

[6] URL: http://wiki.ros.org/xsens_driver?distro=groovy (visited on 02/20/2018) (cit. on p. 17).