Pedro Jorge da Cruz Lino

THE CONTROL OF BAXTER ROBOT, AND ITS INTERACTION WITH OBJECTS USING FORCE SENSITIVE AR10 HANDS, GUIDED BY KINECT

Dissertação de Mestrado em Engenharia Eletrotécnica e de Computadores

09/2017

UNIVERSIDADE DE COIMBRA

Universidade de Coimbra

Departamento de Engenharia Electrotécnica
e de Computadores

Master Thesis

The control of Baxter Robot, and its interaction with objects using force sensitive AR10 hands, guided by Kinect

Date: 11-9-2017

Author: Pedro Jorge da Cruz Lino

Advisor: Hélder Araújo

Advisor: Rui Cortesão

Index

1 Abstract **2**

2 Acknowledgments **2**

3 Preface **3**

4 Introduction to Baxter **4**

5 A first touch on Baxter **5**
- 5.1 Baxter Hardware controllers 5
- 5.2 Baxter Hardware limitations 6
- 5.3 Study of the best Joint Position Control function given by Rethink Robotics . . 7
- 5.4 Study of the Joint Angular Positions and Velocities stability 12

6 Baxter arms and Control Architectures for the end-effector **16**
- 6.1 Control by Position 18
- 6.2 Control by Velocity 23
- 6.3 Control by Torque 28

7 Kinect Sensor and its usefulness with Baxter **32**
- 7.1 Finding objects within an image 32
- 7.2 Object tracking with Kinect using color intervals 33
- 7.3 Object tracking with Kinect using the color histogram 33
- 7.4 Object 3D mapping using Kinect 35

8 Touch sensors **38**

9 AR10 Hands **40**
- 9.1 Controlling the AR10 hands on their own 40
- 9.2 Controlling the AR10 hands with touch sensors 42

10 Baxter Interaction with an Object **43**
- 10.1 Interaction with velocity control 47
- 10.2 Interaction with torque control 48
- 10.3 Comments on the fulfilment of the task, and the controllers performance 49

11 Conclusions **50**
- 11.1 Baxter Robot 50
- 11.2 AR10 robotic hands 51
- 11.3 Soft Sensors 51
- 11.4 Kinect 51
- 11.5 Baxter Robot control and integration with the peripherals 52
- 11.6 Possible future improvements 52

12 Web-graphy and Bibliography **53**

1. Abstract

This work was made in the context of a master thesis, and it aims to explore Baxter Robot capabilities, and explore architectures of control to move its robotic arms. This work also includes the installation of Baxter workstation, for the first time, in our university department (Departamento de Engenharia Electrotécnica e de Computadores da Universidade de Coimbra).

The concrete objective of this thesis is to get Baxter to pick and place objects, as a prove of concept task, in the safest way possible for Baxter to be working side by side with humans. This work takes its greatest emphases on learning how Baxter works, and it will be explained how to take the leap from Baxter given examples, to the study of Velocity and Impedance controllers and how to use their null-space to take advantage of the 7 DOF available on Baxter arms.

To fulfil this pick-and-place task, there will be implement alongside Baxter a Kinect sensor, that will map the position of the object to pick and the site to place, and send this mapped position to Baxter.

To enable a better grasping of objects there were also integrated AR10 robotic hands with soft pressure sensors on the fingers.

Figure 1: Image representative of the robot final build, with peripherals.

2. Acknowledgments

I would like to thank my family and friends and how their work and patience helped me in my journey.

I thank the trust that my advisors, Hélder Araújo and Rui Cortesão deposited in me, and I would also like to thank the professor responsible for the soft sensors Mahmoud Tavakoli, and two of his co-workers Luís Lourenço and Rui Pedro Rocha, that delivered a quite good assistance.

I would particularity like to thank Gonçalo Martins for his help in teaching me the basics of Python and ROS.

3. Preface

I've always been an enthusiast of robotics. This enthusiasm led me to finish a masters in Automation and Control. When searching for a thesis theme, the university had just acquired a Baxter robot, and I was given the chance to be the first to work with it. This seemed to me as the perfect opportunity, to express my dedication and knowledge in getting this "machine" to work.

It wasn't trivial to "build" the workstation from the ground up. It was necessary to install Ubuntu (and all the necessary libraries required), to install the ROS workspace for the robot, to install all the other software required to work with the peripherals (AR10 hands, Kinect, Arduino and Sensors) and, finally, to make them all interact, efficiently, with each other. I also had to learn Python and ROS in the process almost all by myself.

Programming Baxter wasn't trivial either, since passing from the Gazebo simulator to the real thing was not without its "surprises".

Things that proved to be difficult to manage in order to get Baxter to accommodate velocity and torque controllers were:

- the fluctuation/noises of Baxter given joint angular positions and velocities, at each instant;
- the management of the process running the control cycle, that was very sensitive to the adage of more ROS subscribers, since they would decrease its frequency.
- the official *Rethink Robotics* site (the source of many of the information available on Baxter) has its information scattered and it's not always very easy to find a specific subject, topic or page in it;

Adding to these difficulties, the AR10 hands were relatively slow, full of fluctuations and noises, and the tutorials weren't very useful at preparing the user to deal with the "real" hand limitations themselves.

The Kinect sensor, however widely used, was a little tricky to implement in Ubuntu/Linux using ROS indigo, with freenect library and drivers.

Finally, the soft sensors used were a little hard to get right, since they usually leaked liquid metal, and had to be repaired sometimes, until a robust solution was found.

Bottom line, in engineering every problem leads to other problems and questions, and the process of making something work, takes its time. To the best of my work, many of these problems were bypassed or solved throughout the thesis development, and I was satisfied with the final result.

Better results could have been achieved with more time and effort, but I'll leave my work in a proper state, so it can be grabbed and continued by anybody interest in the subject. I encourage people to improve the problems discussed, and find new, better solutions to them, so society can start to integrate safer and more useful robots in our daily lives.

4. Introduction to Baxter

Baxter is an accessible robot from *Rethink Robotics*. Because it was built and designed to be robust and safe to work around, it delivers an overall great solution as a collaborative robot, that can work continuoulsy in a variety of environments. "Baxter is easy to train and fast to deploy" - as stated in the official site [1].

Baxter most evident attributes are two independent force sensing arms, with 7 DOF (Degrees of Freedom) each, and it also includes 3 independent cameras one in the top of the LCD and two in each wrist (the wrists also include an accelerometer and a distance IR laser sensor).

The greatest advantage of using Baxter, and similar robots, is that they can match many human repetitive tasks (if used well), and, therefore, they are a step forward in the automation of processes and in the human-robot interaction and conjoint work.

The Baxter version utilized in this thesis is the: Baxter Research Robot (v1.2.0.57), which does not contain the Intera software [2]. The Intera software was specifically designed to make Baxter more accessible ("plug and play"), and to help workers interact easily with Baxter interface, allowing them, also, to teach him to perform tasks and help with their own job, in their specific environment.

The version used, allows Baxter to be programmed from the ground up, with the additional freedom that derives from it.

In order to program Baxter, it was followed the *Rethink Robotics* tutorials and example codes avaiable for Baxter from the offical site [6]. In order to get the examples to work, there was installed ROS Indigo in Ubuntu 14.04 (as recommended) and used Python as the codding language. The Python processes will be executing/running directly in the robot's computer, via SSH (Secure SHell).

Baxter arm is composed by seven joints and these joints will sometimes be referred to as a vector q of 7 dimensions.

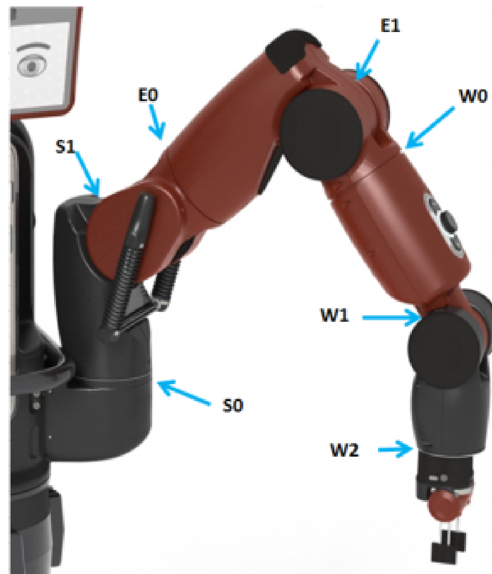


Figure 2: Image of Baxter left Limb/Arm and it's joint names.

5. A first touch on Baxter

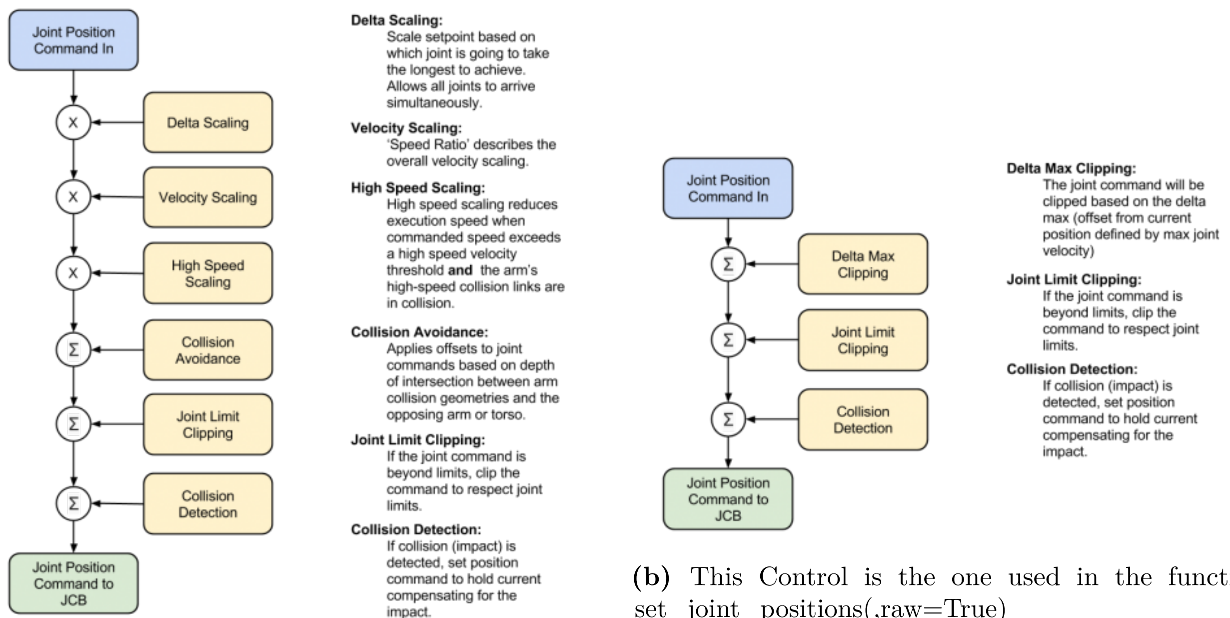
Before jumping to the real robot, some codes were experimented in the Gazebo Baxter simulation. However it was a very accurate simulator, and helped understanding Baxter examples very easily, it wasn't without its limitations. The real implementation was needed to identify real-time problems, like noises associated with measurements, tremors and fluctuations, and force interactions with the robot, in between others (for example: cameras and distance sensors).

After turning on the robot for the first time, and calibrating and taring the arms, there were executed some of the given examples, that had already been tested on the Gazebo simulator. The first thing noticed, in all the examples, was the electrical motors noises, and tremors, that led to suspect that the robot wasn't very precise, even right after being calibrated. This resulted in going through a series of tests to find the best way to program, deal with the robot, and its "real" environment limitations.

5.1. Baxter Hardware controllers

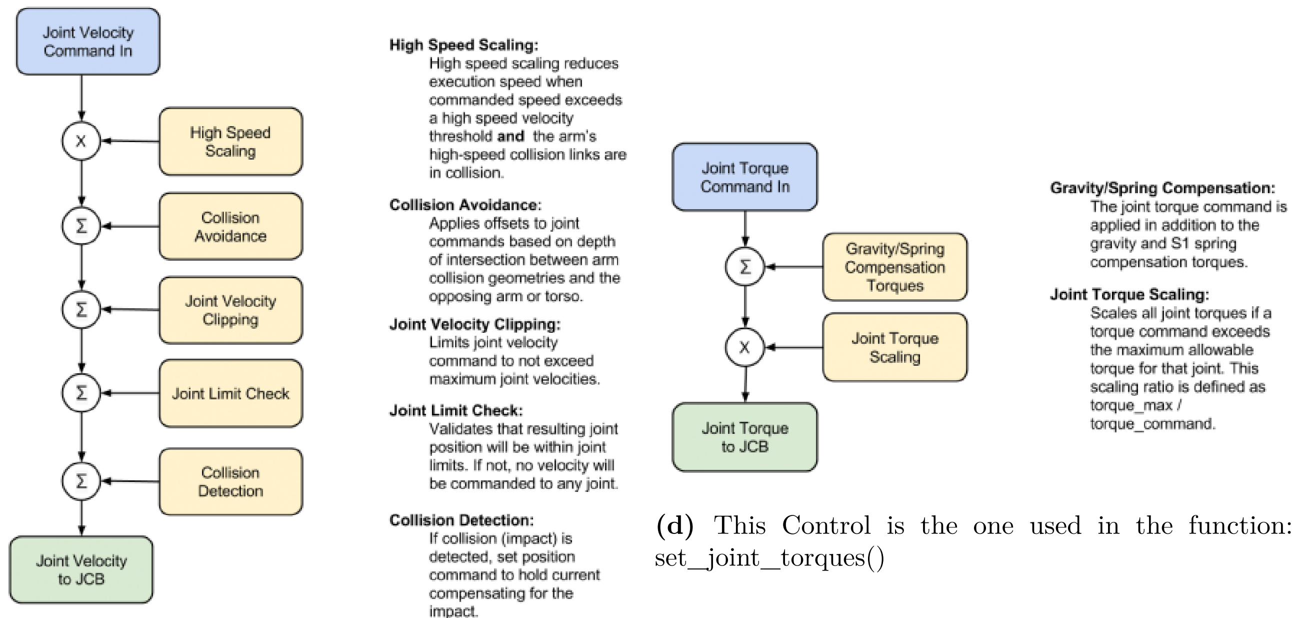
In order to communicate correctly with Baxter, ROS messages must be shared between the Robot Hardware, and a higher level programming language like C++ or Python. In this work, to follow Baxter's example codes, Python was the language chosen.

The Software programed, must, at all times, respect the Hardware restrictions (like ROS frequencies, and sensor resolutions). One of the most important things that must be respected at all times, while testing architectures of control, is the given lower-lever control modes avaiable to control the arms motors movements. These are talked about in *Rethink Robotics* site [5], and the information can be summarized in these images:



(a) This Control is the one used in the functions: move_to_joint_positions(), set_joint_positions(,raw=False)

(b) This Control is the one used in the function: set_joint_positions(,raw=True)



(d) This Control is the one used in the function: set_joint_torques()

(c) This Control is the one used in the function: set_joint_velocities()

Figure 4

5.2. Baxter Hardware limitations

Before advancing further on this thesis, it is important to know some of the robot hardware limitations (taken out from the official site [5]). These limitations will be more evident in the accuracy of the results obtained from the tests that the robot will be exposed throughout the work. These might be in the root of many problems and imprecisions in the plots.

The following items, were taken from *Rethink Robotics* official site:

- (Quote regarding the Torque motor controller) A control rate timeout is also enforced at this motor controller layer. This states that if a new 'JointCommand' message is not received within the specified timeout (0.2 seconds, or 5Hz), the robot will 'Timeout'. When the robot 'Times out', the current control mode is exited, reverting back to position control mode where the robot will command (hold) it's current joint angles.

- (Quote) When commanding joint velocities, if a commanded velocity to one of the joints will result in a joint position that is beyond the joints limits, no joints will be commanded, as all of the command is considered invalid. Reason we do this; an example of a common control method, using the Jacobian for Cartesian control resulting in joint velocity commands. If a single joint hits its limit, the rest of the joints will still be commanded, resulting in obscure and potentially dangerous motions. We recommend either implementing a joint space potential field or joint limit check before submitting the joint velocity commands.

- (Quote) The resolution for the joint sensors is 14 bits (over 360 degrees); so $360/(2^{14}) =$ 0.021972656 degrees per tick resolution.

- (Quote) All of the joints have a sinusoidal non-linearity, giving a typical accuracy on the order of +/-0.10 degrees, worst case +/-0.25 degrees accuracy when approaching joint limits. In addition, there may be an absolute zero-offset of up to +/-0.10 degree when the arm is not calibrated properly. Be sure to tare and calibrate the arms if you're trying to minimize accuracy errors in the joint sensors. (...) The performance of the joint controllers is a separate matter; ultimately, the accuracy of the controller is only limited by the accuracy of the sensors. In a case where joint position controllers are included, we use a threshold to determine when the joint states are "close enough" to the commanded joint angles to call it acceptable. In the baxter_interface for the RSDK, we use a default threshold of: JOINT_ANGLE_TOLERANCE = 0.00872664626 radians (0.5 degrees), which is set in settings.py and used by the limb interface and the joint_position examples. If you want to improve the accuracy, you can write your own controller to adjust the setpoint and to overcome any steady-state error in the internal low-level controllers. Even when the joint controller is slightly off the target position, it always knows exactly how far off it is, via the joint position sensors.

5.3. Study of the best Joint Position Control function given by Rethink Robotics

The simplest way to test the stability of the robot was to run some of the *Rethink Robotics* own code, to make the arm stay at a given pose, for a specific amount of time, and see if the tremors/noises would continue, and if so, what could be in the root of this problem.

The tests that are about to conducted, and presented, will be experimenting the Python class "Limb", from *Rethink Robotics* API reference ([4]), to show Baxter static performance with the functions: move_to_joint_positions() , set_joint_positions(,raw=False) and set_joint_positions (,raw=True). These Python functions, theoretically, do exactly the same thing, that is: send the desired angular position command to the joints motors. The ROS message being used by these functions is: "/robot/limb/(left/right)/joint_command", which properties result in the different nuances:

- move_to_joint_positions() commands the joints to arrive at the desired position at the same time, using a low-pass filter of 0.2 Hz. The function relies on the ROS message property POSITION_MODE, thus uses the controller from image 3a;
- set_joint_positions(,raw=False) simply commands the joints to the specified positions. The function relies on the ROS message property POSITION_MODE, thus uses the controller from image 3a;
- set_joint_positions(,raw=True) commands the joints to the specified positions, providing a much more direct position control on the joints (it can be also more dangerous to use). The function relies on the ROS message property RAW_POSITION_MODE, thus uses the controller from image 3b

The commands carried by the messages must respect the hardware properties of the robot, including the recommended ROS rate of 1kHz. This also means that the end-user/programmer, has to submit to Baxter hardware limitations (stated in section 5.2).

To test the robot response for a static reference, the following tests were run after calibrating and taring the robot, and were all executed in the left limb/arm. The same tests were conducted for the right arm, but the results were similar, since the arms are very similar in structure and behaviour.

The static angular reference given to the left-arm joints (referenced in 2) (s0..w2), at an approximate rate of 1kHz, was:

$$q_{initial}(s0..w2) = \begin{bmatrix} -\pi/4 & -\pi/4 & 0 & \pi/4 & 0 & \pi/2 & 0 \end{bmatrix} \tag{1}$$

which corresponds to an approximate end-effector position, referenced on the robot's root/origin, of:

$$X_{initial}(x, y, z) \simeq \begin{bmatrix} 0.815 & 0.257 & 0.364 \end{bmatrix} \tag{2}$$

it goes without saying that the static behaviour predicts **no** angular velocities on the joints from (s0..w2).

Note: In the following examples the $X_{initial}$ will always be the first X,Y,Z position read by the robot in the beginning of its movement, after being put on the $q_{initial}$ using move_to_joint_positions().

For the use of the function: move_to_joint_positions(), the following plots were obtained (measures in millimetres):

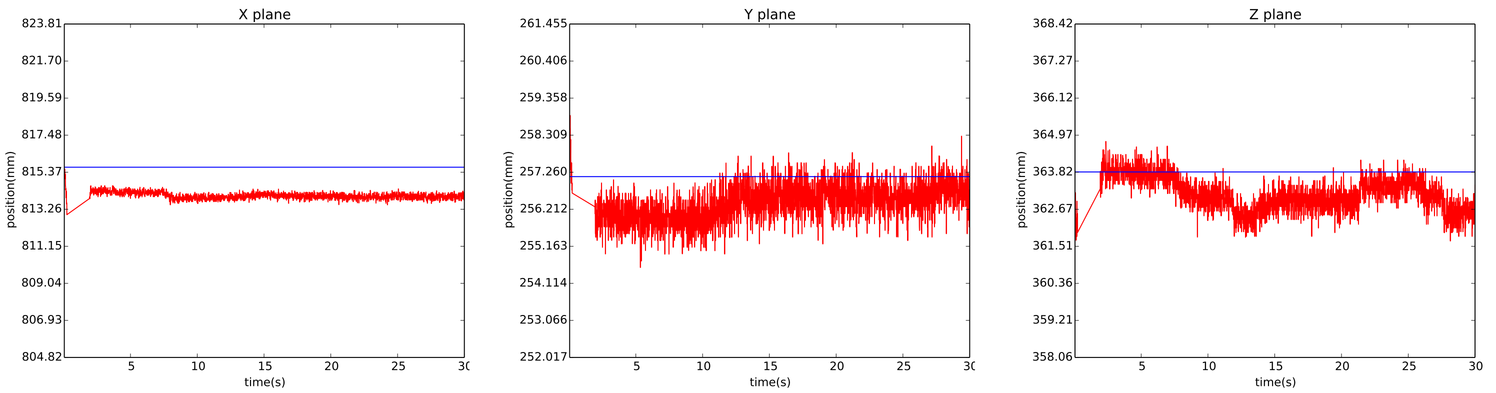


Figure 5: Plots of the Left arm end-effector position in (x,y,z) for move_to_joint_positions()

As it can be noticed in the plots, something strange happened in the interval from the 1st to 3rd seconds. What happened, is that the end-effector position started being published in a different way. This might be explained by the filter being used in the function, and its nuance of blocking Baxter's data flow. Anyhow, it can be seen that the end-effector position, doesn't remain still even with the filter being applied by the function: move_to_joint_positions(). It can also be clearly seen a stationary error, left to correct, in the X plane.

For the use of the function: set_joint_positions(,raw=False), the following plots were obtained (measures in millimetres):

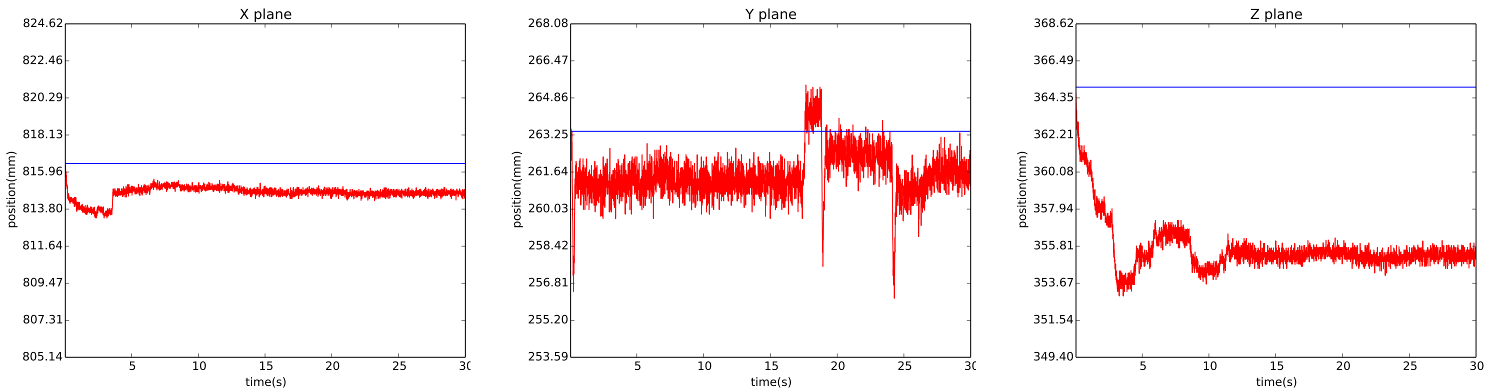


Figure 6: Plots of the Left arm end-effector position in (x,y,z) for set_joint_positions(,raw=False)

It can be seen in the plots that there were left great stationary errors to correct, in the X and Z planes. One can be led to suspect that the function set_joint_positions(,raw=False) gives worse results than the previous move_to_joint_positions() function. This only stands true depending on the implementations.

For the use of the function: set_joint_positions(,raw=True), the following plots were obtained (measures in millimetres):

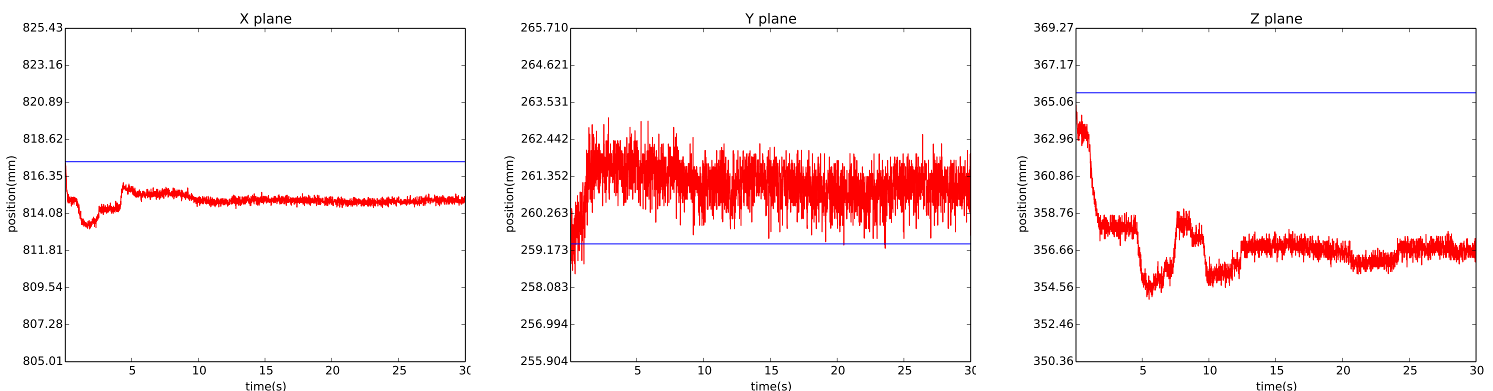


Figure 7: Plots of the Left arm end-effector position in (x,y,z) for set_joint_positions(,raw=True)

From the plots, it can be deduced that, for small movements, there are little differences between the function set_joint_positions() with the attributes raw=True and raw=False.

As shown in the plots, the arm end-effector position has associated a lot of noise. This noise is to be expected since the robot is exposed to the "real" environment. The signal tends to stabilize eventually, but always far away from the reference with a remaining stationary error. It is good to recall that this (X,Y,Z) reference was achieved by moving the arm to a certain initial joint position with the function move_to_joint_positions(), that seems to be the most accurate of them all.

From the previous plots, it could be argued that this reference was not well taken, and that the arm's reference is not the one shown in blue, but should be the one it is tending/stabilizing to. In order to disprove this, the following tests were taken to examine the exact same system when subjected to an outside perturbation. This perturbation will have three stages: **stage one** - a small perturbation will be applied to the side of the arm, not enough to move the motors out of place; **stage two-** a harder perturbation will be applied to the side of the arm, rough enough to move the motors out of place. **stage three-** redo a similar perturbation to the one done in stage one;

It's important to remember that the joints angular positions of reference (from 1) are constantly being applied to the arm.

For the use of the function: move_to_joint_positions(), the following plots were obtained:

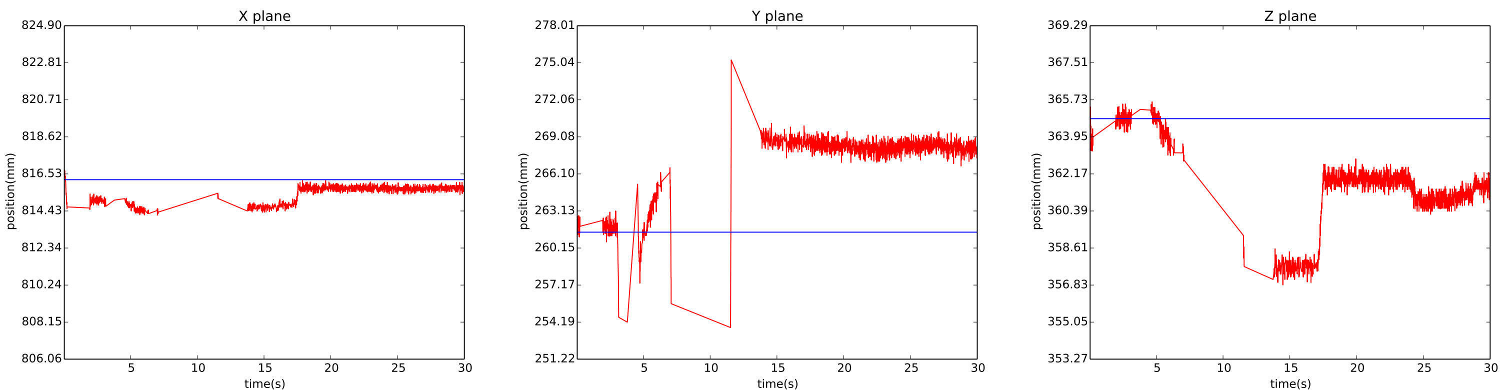


Figure 8: Plots of the Left arm end-effector position in (x,y,z) for move_to_joint_positions()

For the use of the function: set_joint_positions(,raw=False), the following plots were obtained:

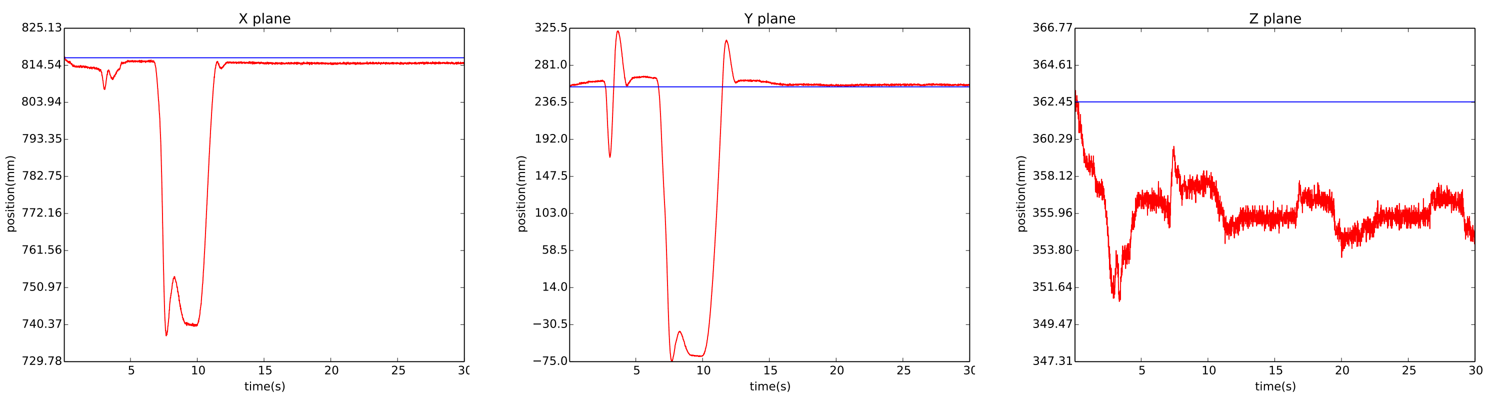


Figure 9: Plots of the Left arm end-effector position in (x,y,z) for set_joint_positions(,raw=False)

For the use of the function: set_joint_positions(,raw=False), the following plots were obtained:

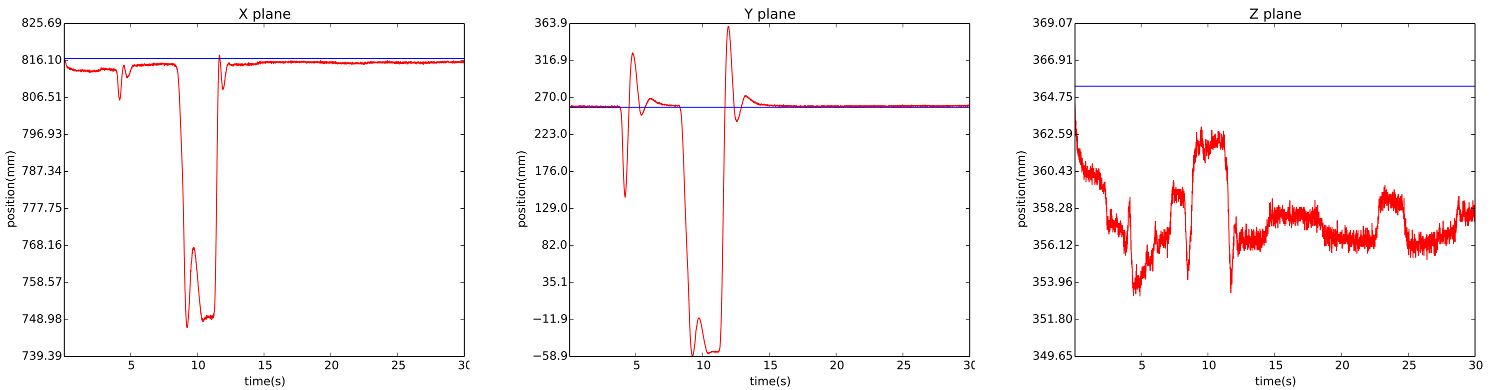


Figure 10: Plots of the Left arm end-effector position in (x,y,z) for set_joint_positions(,raw=True)

As shown in the plots above, the response of the hardware control systems for the perturbations applied should be able to recover to the same end-effector position reference they were stabilizing to, before the three perturbations were applied. It can be seen, however, that the control systems recover to another stability state, that is different from the initial one. This might suggest a deeper problem associated with the joints, and some approaches will be taken to solve this problem in the next sections.

An important result was to discredit move_to_joint_positions() from being the best function to control the position of the joints, since it is not fast enough to deal correctly with perturbations. The observed experimental results, using this function, was bad publishing of the joint positions, velocities and the position of the end-effector, which resulted in an unexpected movement of the arm.

Another interesting result is that, when using the raw=True feature, it can be seen a greater overshoot (more evident in the Y plane) relative to raw=False, but the recovery time is almost the same. One can be led to conclude that the raw=False is a smoother way to obtain almost the same results as raw=True, but with more safely - without the risk of reaching dangerous torques and without the risk of collisions with Baxter itself.

Everything led to the conclusion that using the function: set_joint_positions(,raw=False), was the best way to control the joint angular positions, in a safe way, for most of the applications.

(These tests were also conducted for the right arm, but with very similar results.)

5.4. Study of the Joint Angular Positions and Velocities stability

As shown in the plots, in the previous section, the position of the end-effector is filled with noise, and there might be occurring problems in the joints that cause this (x,y,z) end-effector state and behaviour. To examine closer the angular values responsible for this, and because they were filled with noise, there was applied a low-pass filter (which is an adequate solution for these high-frequency noises), so that the actual signal could be differentiated from the noise. The desired signal is supposed to contain the ROS subscriber message of the joint positions and velocities, and these are crucial for the implementation of the velocity and torque controllers. Therefore, the filtered signals won't solve the end-effector position problem (at least directly), but will help to stabilize the control architectures used in sections 6.1, 6.2 and 6.3.

A good discrete low-pass filter was verified to be the following (applied to the angular positions and to the joint velocities respectively):

$$q_{filtered}[k] = \frac{\alpha h q_{read}[k] + q_{filtered}[k-1]}{1 + \alpha h} \tag{3}$$

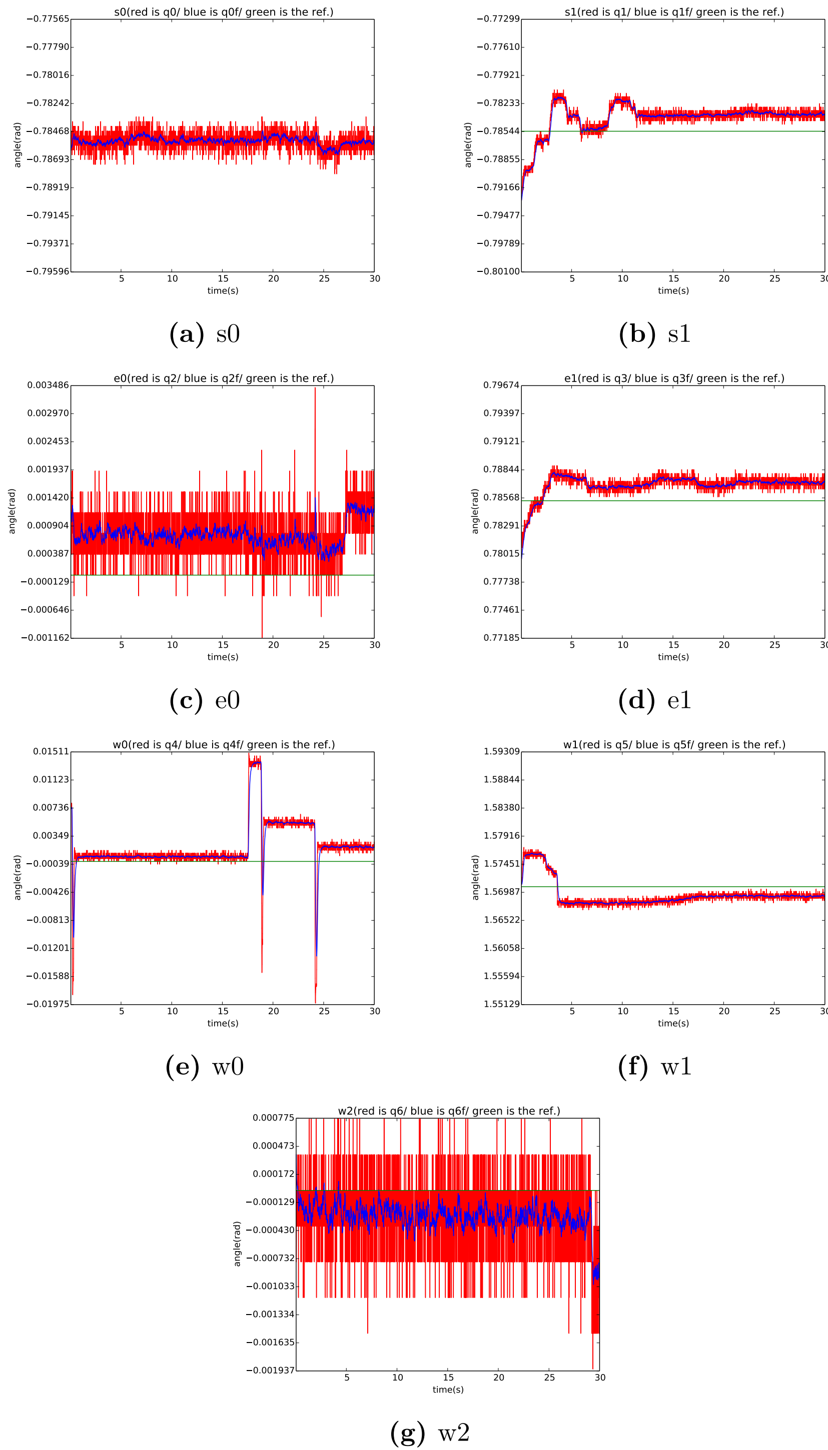
$$\dot{q}_{filtered}[k] = \frac{\alpha h \dot{q}_{read}[k] + \dot{q}_{filtered}[k-1]}{1 + \alpha h} \tag{4}$$

where q is the joint angular position, $\dot{q}$ is the joint angular velocity, α is the cut-off frequency constant, and h is the sample time (commonly used as Δt).

In order not to overload the reader with plots, from now on, I'll make use of the function: **set_joint_positions(,raw=False)**. It was concluded in section 5.3 that this function was able to follow a real-time reference (which the filtered version move_to_joint_positions() can't) and it's less dangerous to use than its version raw=True. The differences in the end-effector positions, by taking this choice, are minimal as you saw in the previous section.

Continuing with the study of the arm joints, for a static reference (the same as in 1), but now looking at the angular joint positions and velocities, the following plots were obtained:

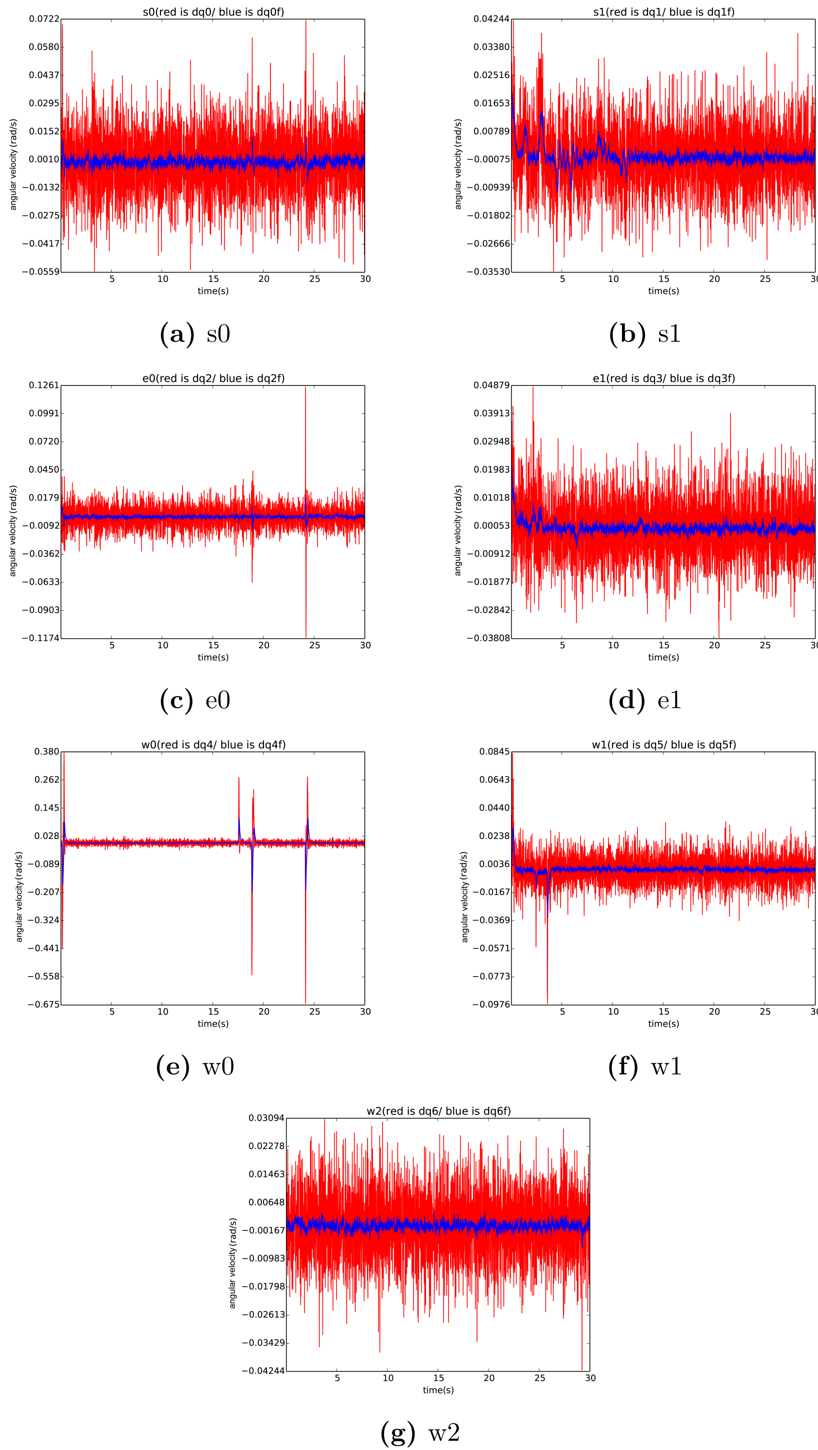
The plots in this page show three different signals: the **green** is the angular static reference, the read signal in **red**, and the filtered red signal in **blue**. The low-pass filter used had a cut-off frequency of 100Hz.



(a) s0 **(b)** s1 **(c)** e0 **(d)** e1 **(e)** w0 **(f)** w1 **(g)** w2

Figure 11: Plots of the Left arm joints (s0..w2) angular positions (referenced in 2), in radians

The plots in this page show two different signals: the read signal in **red**, and the filtered red signal in **blue**. Because the arm is supposed to be still, the velocities on the joints should be zero. The low-pass filter used had a cut-off frequency of 100Hz.



(a) s0 **(b)** s1

(c) e0 **(d)** e1

(e) w0 **(f)** w1

(g) w2

Figure 12: Plots of the Left arm joints (s0..w2) angular velocities (referenced here 2), in rad/s

From the plots in the previous pages, it can be seen that the filter took away a lot of undesired noise from the read measurements. However, low-pass filters tend to delay the signal received, in this case, that delay was negligible.

The filter only actuates on the read values, and, by itself, cannot mitigate the ever so slightly movements of the joints, even for a static reference.

It was observed, that, unfortunately, some joints keep a static angular error from the desired reference. This can only be blamed on the hardware motor controllers, since the same reference is being given at an almost constant rate of 1kHz. Altering the JOINT_ANGLE_TOLERANCE to a smaller value, to increase the precision of these controllers, resulted in irrelevant improvements; in fact other problems started to emerge, like: the joints never reached the desired position with set_joint_positions(,raw=false); the joints took too long to reach the positions with move_to_joint_positions(); even in the best case, with set_joint_positions(,raw=true), where the desired positions were achieved, there weren't noticeable any great improvements in the joint positions precision and noises.

In some extreme cases, the noises associated with the joints, admitted errors of 0.5 degrees, even after the robot had been calibrated and tared. This seems to be a small error for a single joint, however, there are 7 joints, meaning that this effect can, and will, be escalated to the end-effector position (and might be the cause of the oscillations seen in 9). This problem will also escalate with the time of execution, since, with time operating, the robot will lose its calibration.

The limitations in section 5.2, should account for some imprecisions in the control architectures studied in the next sections.

In the next sections there will be implemented control architectures, which will face real-time problems, and even with the applied filter, there should be expected vibrations of some sort through the arms, or slight deviations from the desired positions. This imprecisions are not a "surprise", in the sense of the word. References to this issues, can be found in articles like the IEEE article [9], and the relative price of this robot compared to other, **more precise**, ones (that should give a hint about why these problems/imprecisions happen). Even with some complications, satisfactory results were obtained.

6. Baxter arms and Control Architectures for the end-effector

The arm geometry:

Baxter robot has two great arms with 7 DOF each. To take advantage of these degrees of freedom, one can control the angular positions of the 7 joints, as well as their velocity and torque, to output a certain position of the end-effector, and "theoretically", with zero steady state error. This end-effector might be holding some tool (which can be a gripper or a hand) attached to Baxter's cuffs, and it's worth controlling with the best precision possible. The end-objective of this control, is to attach the AR10 hand to the end-effector, and control its position and orientation at all times.

The easiest way to control the end-effector in a desired (x, y, z) position is through inverse kinematics (IK). Baxter tutorial examples, provides us with an inverse kinematic example based on Baxter's URDF (Unified Robot Description Format). This approach, however, is quite slow, not immune to singularities, and cannot always solve for positions in obvious reach of the arm. Therefore, for a real-time control of the arm, without human teaching, another approach is needed.

To control the end-effector Cartesian coordinate, it must be generated an error vector. This error vector can be calculated using the difference between the actual position of the end-effector and the desired (x,y,z) position, referenced on Baxter's root. This control will make the arm reach for that point or the closest point that minimizes that (quadratic) error.

One easy way to describe the end-effector position depending on the state of the seven joints, is through the Denavit–Hartenberg (DH) parameters. The specific DH parameters for Baxter are:

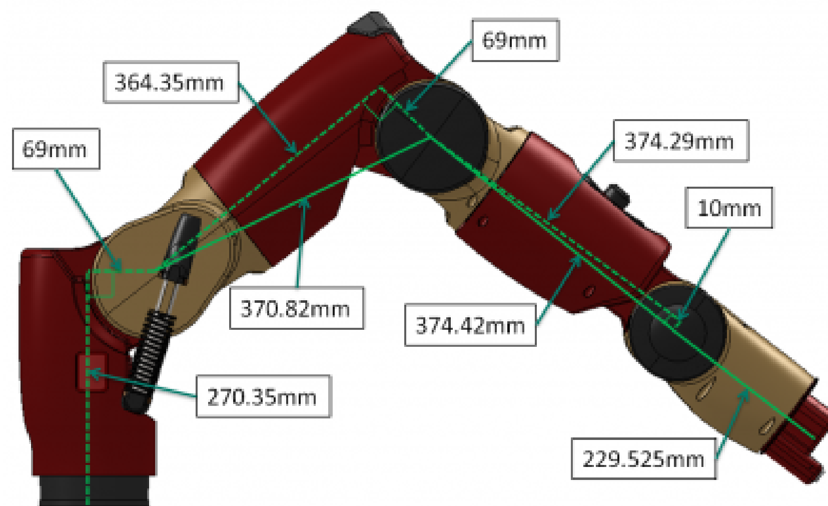


Figure 13: Baxter generic arm, dimensions.

	θ_i	d_i	a_i	α_i
Shoulder 0 (T_1^0)	θ_0	0.27035	0.069	$-\pi/2$
Shoulder 1(T_2^1)	θ_1	0	0	$\pi/2$
Elbow 0 (T_3^2)	θ_2	0.36435	0.069	$-\pi/2$
Elbow 1 (T_4^3)	θ_3	0	0	$\pi/2$
Wrist 0 (T_5^4)	θ_4	0.37429	0.01	$-\pi/2$
Wrist 1 (T_6^5)	θ_5	0	0	$\pi/2$
Wrist 2 (T_7^6)	θ_6	0.229525	0	0

Figure 14: Table of Baxter's Denavit–Hartenberg (DH) parameters. This measurements are in meters (d_i, a_i) and in radians (θ_i, α_i)

The geometric transformation T_7^0 (dimensions 4x4), provides information of both the end-effector Cartesian position (in x,y,z) and orientation (in a rotation matrix).

Note: Because the arm origin has an offset from Baxter's root, the approximate values should be added to the given end-effector position for the left arm of: $[+0.0652, +0.26089, +0.117422]$ and $[+0.0652, -0.26089, +0.117422]$ for the right arm.

The Jacobian (J) for the end-effector of the arm can be obtained by deriving this end-effector (x,y,z values) in relation to all the angles joints (from s0 to w2).

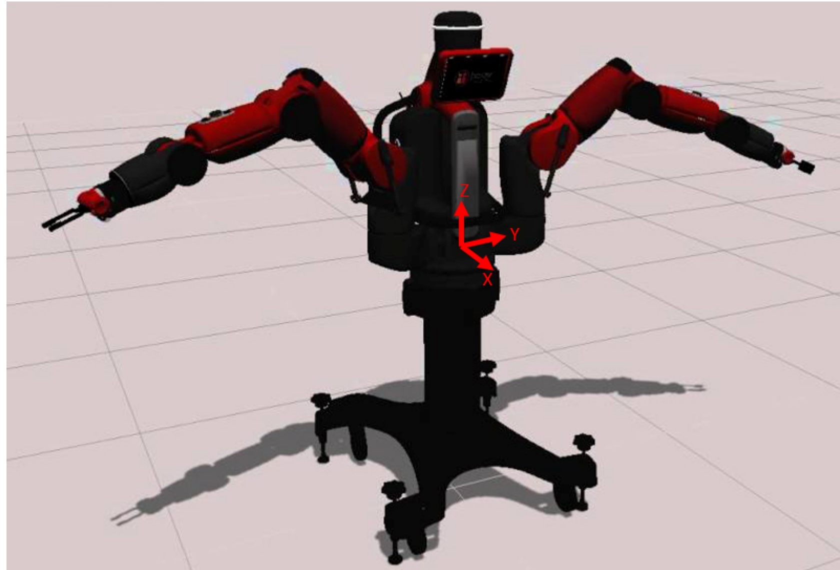


Figure 15: Baxter on Gazebo simulator, and Axis visualisation

Baxter publishes the end-effector position and orientation, and this information can be extracted in the function: endpoint_pose(). This function can be found in the Python class "Limb" given by *Rethink Robotics*. Other very usefull functions to calculate the Jacobian and Inertia Matrices, based on Baxter's URDF files, are given in an additional Python library from Orcos [7] called "baxter_pykdl".

Because of hardware dependencies/restrictions, some of the input values for the control, sourced from the ROS messages related to the "end point" (or end-effector), are published at a lower frequency (100Hz) than the "joint states" (1000Hz). These limitations can sometimes result in real time problems. Because of this, in order to source some input values more correctly and faster, to obtain the velocity of the end-effector, there was used the relation: $\dot{X} = J\dot{q}$.

The control architectures will base themselves on the minimization of the Cartesian error vector, this error vector is made by differencing the **actual** position and/or velocity from the **desired** position and/or velocity, at each instant (in case of the impedance control it can be added a desired acceleration). In addition to this, other error vectors can be minimized, because the robot is redundant, and allows many configurations for the same end-effector position (this will be explained in more detail on the next sections).

The following subsections will explore the control architectures: **Control by Position, Control by velocity and Control by Torque**. The order these architectures will be presented, was chosen in order to contextualize the reader and to escalate the complexity of the architectures slowly.

Note: In the control architectures explained in the next sections, there weren't taken in account additional loads (weights) applied on arm's the end-effector. In the case of adding these weights to the end-effector, its recommend to tune the gains to compensate for the loads, and still keep the desired behaviour.

6.1. Control by Position

In between Baxter's examples, there is one in which the robot reads the angular positions of the joints from a file, and tries to comply with the angular positions and time schedule listed in that file. This file is supposed to be originated from running another example in which a human teaches Baxter a trajectory, by taking control of the arm in "**Zero-G** mode".("**Zero-G** mode" is a mode in which Baxter's arms behave like they had no weigh, so they can be easily posed by a user, who can obtain that control by grabbing Baxter's cuffs) **Note:** Through out this thesis this **Zero-G** mode will have to be suppressed at all instants, so it won't override the control architecture running.

In order to control the robot, it was made use of the function set_joint_positions(,raw=False) (used previously in section 5.4), to adapt the example, to use our own desired trajectory and schedule. This was the first approach to take, because it was the simplest to derive from the given examples, and it was the simplest to start testing and implementing control architectures for the end-effector, in this case Position Control.

Controlling a robot end-effector position is a mathematical problem that can be solved using geometry. Some simple trajectories such as lines and ellipses are usually used to test the tracking/compliance capabilities of a robot, to obey the desired commands.

Two approaches can be taken doing this tests. In one hand, the desired trajectory to be followed can be know beforehand, and implemented so, on the other hand, one might want the robot to follow a certain, arbitrary, trajectory based on the minimization of the end-effector position error.

I'll begin with the simplest approach, that is: for a beforehand knowledge of the trajectory. Only after, I'll resume with the error based approach.

The mathematical problem context involved in the process, will be explained as follows:

To map a desired velocity in Cartesian space to the angular joint space (and vice-versa), the following physical relation is needed:

$$\dot{X} = J\dot{q} \quad \equiv \quad \dot{q} = J^{+}\dot{X} \qquad (J^{-1} \cong J^{+}) \tag{5}$$

In the equation, let X be the (x,y,z) Cartesian position vector and $\dot{X}$ the Cartesian velocity vector; J is the Jacobian matrix with dimensions 3x7, and $\dot{q}$ is the angular velocity of the joints (from s0 to w2) that respect the relation/equation. **Note:** The almost equivalent sign was used because Baxter is redundant and J cannot be directly inverted.

The Jacobian matrix can be separated into two distinct matrices, the top one related to the Cartesian velocities, and the bottom one related to the angular velocities like so:

$$J = \begin{bmatrix} J_v \\ J_w \end{bmatrix} \tag{6}$$

Even with the use of only one of these subdivisions of J, the equations hold true. This fact will be widely used along this thesis, in order to work with the **null-space** of some equations.

In order to invert the Jacobian matrix (**that is not square, because the robot is redundant**), it was used the Moore-Penrose pseudo-inverse relation. In this relation, the solution will be approximated via the least squares method.

J^{-1} will be, from now on, called J^{+}.

$$J^{+} = J^{T}(JJ^{T})^{-1} \tag{7}$$

Now, in order to control the angular position of the joints, that respect a given end-effector position, a discrete approximation of the angular velocity can be made, by transforming the previous equations into:

$$q[k+1] = q[k] + \Delta t J_v^{+}\dot{X}_v \qquad k \in [0, 1, 2...N] \tag{8}$$

In the equation, $q[0]$ needs to start in the initial joint configuration ($q_{initial}$), Δt is around 0,001 seconds (respecting the 1kHz ROS frequency), and $\dot{X}_v$ is the desired Cartesian velocity at each instant, compliant with Δt. This approximation, however being the easiest to derive from Baxter examples, has its limitations, and those will be studied.

The following tests will be experimenting a reference Cartesian velocity (in meters by seconds), for 30 seconds, of:

$$\dot{X}_v = \begin{bmatrix} 0 \\ -0.01 \\ -0.01 \end{bmatrix} \quad (9)$$

with a **perturbation**, to see if the robot can comply with a Cartesian line, even when disturbed.

The robot started with a initial joint position of:

$$q_{initial}(s0..w2) = \begin{bmatrix} -\pi/4 & -\pi/4 & 0 & \pi/4 & 0 & \pi/2 & 0 \end{bmatrix}^T \quad (10)$$

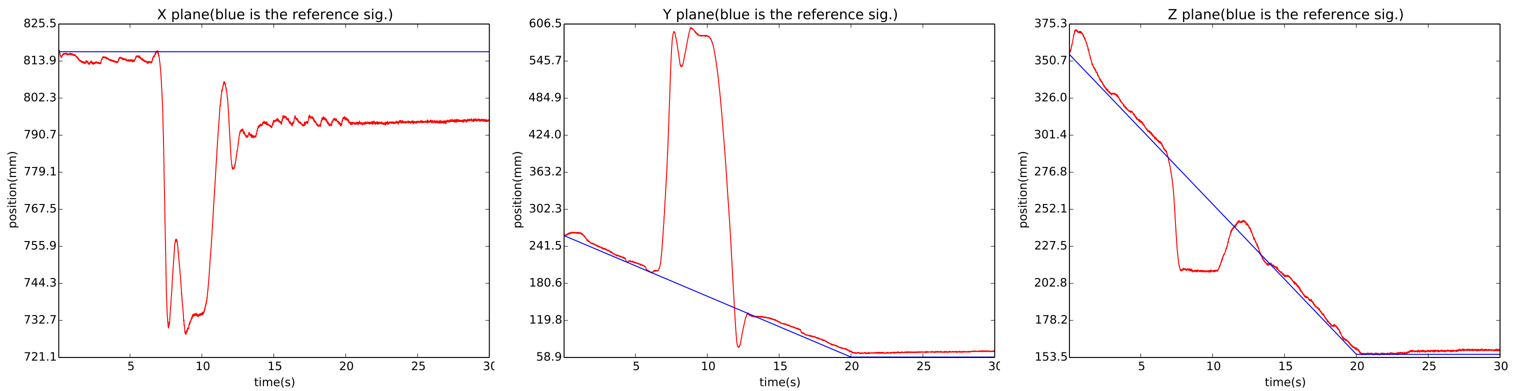


Figure 16: Plots of the end-effector position for a line trajectory on the left arm, with raw=False position control, with a perturbation.

As it can be seen in the plots, the control of the end-effector is being well applied, corresponding to the end-effector being able to follow the reference signal in **blue**, relatively well. In fact the controller is even able to recover partially from a strong perturbation but, not without stationary-regime errors. Unfortunately, it can still be seen a slight deviation from the reference signal at each instant, that might evolve into bigger deviations with time.

This same problem occurred when applying sinusoidal trajectories. For example, another test made with this architecture, was to apply a circular trajectory as the reference signal. The circle was of radius R=0.05 meters and placed on a plane. These circles were made by starting on the circle edge (keeping a small initial error).

From now on in this thesis, this X plane circle trajectory, will be repetitively used, because it's a good example to see the tracking capabilities of the controllers in all axis. This trajectory, of radius 5cm, and of frequency 0.08Hz, gave the following plots for this case:

$$\dot{X} = \begin{bmatrix} 0 \\ -R0.5sin(0.5t) \\ R0.5cos(0.5t) \end{bmatrix} \tag{11}$$

$$q_{initial}(s0..w2) = \begin{bmatrix} -\pi/4 & -\pi/4 & 0 & \pi/4 & 0 & \pi/2 & 0 \end{bmatrix}^T \tag{12}$$

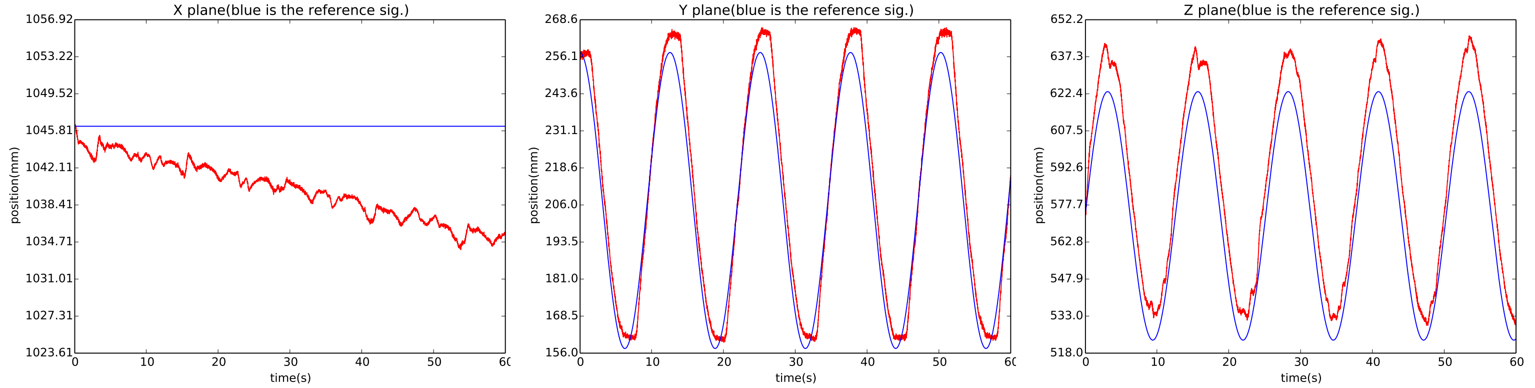


Figure 17: Plots of the end-effector position for a circular trajectory on the left arm, with raw=False position control.

In this example, it can be evidently seen that there is a growing error in the X plane, and it can also be seen that, however following the reference, the robot overshoots too much on the sides of the circle, were the derivative is higher.

One solution to try to minimize the observed flaws, is to implement a PID controller that outputs a desired velocity vector, based on two very convenient errors that can be computed at all times (the position and velocity error vectors), and an extra approximate error, made from the position error, that can be integrated in time.

The control equation can be expanded to:

$$\dot{X}_v = Kp(X_{desired} - X_{actual}) + Kd(\dot{X}_{desired} - \dot{X}_{actual}) + Ki\int_0^t (X_{desired} - X_{actual})\delta t \tag{13}$$

Note: The K_p, K_d and K_I are positive scalars applied to the values of a square identity matrix, however, depending on our intentions, some of the diagonal values can be "over-controlled" or "under-controlled" by taking more or less scalar importance/weight than the rest.

$\dot{X}_{actual}$ can be obtained at all times by $\dot{X}_{actual} = J_v\dot{q}_{actual}$, because Baxter also publishes the joints velocities. $\dot{X}_v$ can also be plugged in directly onto the equation:

$$q[k+1] = q[k] + \Delta t J_v^+ \dot{X}_v \qquad k \in [0, 1, 2, 3...] \tag{14}$$

When applying this controller, using the same equation as before, but now for a line trajectory, using the gains $K_P = 2$, $K_D = 0.4$ and $K_I = 0.5$, the following plots were obtained:

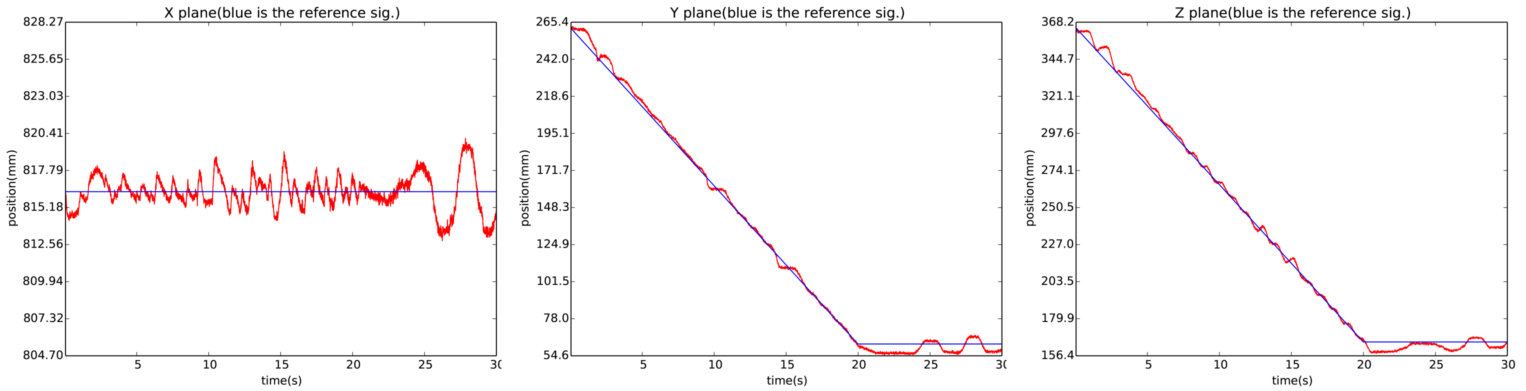


Figure 18: Plots of the end-effector position for a line trajectory on the left arm, with raw=False position control.

However it does not seem like a bad controller, and better gains can lead to an even more accurate following of the reference, this controller is not robust against ruff environments. If per say, the robot faces a wall or a person on its **rigid** movement (which is characteristic of a velocity controller), the robot won't deal nicely with the error generated in the process, and it will degenerate the control to become unstable, as it can be seen on the plots below:

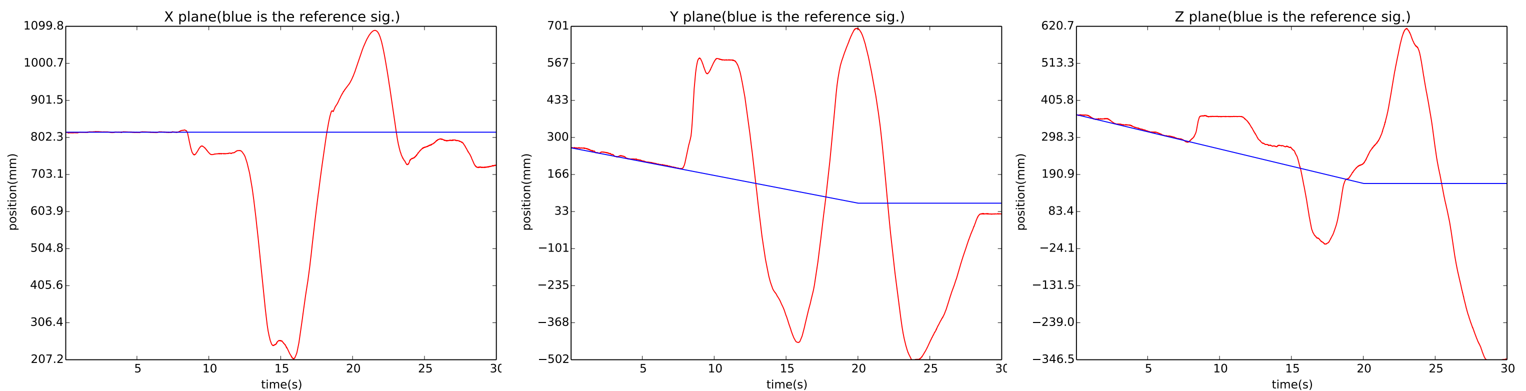


Figure 19: Plots of the end-effector position for a line trajectory on the left arm, with raw=False position control, with a perturbation.

This effect becomes even worse, when used its variation with "raw=True". This is one of the greatest limitations of this discrete velocity controller, that relies heavily on Baxter maintaining the desired joint angular position with precision, and fast enough. Other limitations were observed when trying to apply a null-space behaviour to this controller.

The good news are the limitations of this controller, can be overcame by an even better controller, very similar to this one, which is the velocity controller.

6.2. Control by Velocity

This is the section where the effects of acting directly on Baxter's joints velocities will be examined. The function used to this end was: set_joint_velocities(). The physical relation (from 6.1) can be reused for the control:

$$\dot{X}_v = Kp(X_{desired} - X_{actual}) + Kd(\dot{X}_{desired} - \dot{X}_{actual}) + Ki\int(X_{desired} - X_{actual})\delta t \quad (15)$$

$$\dot{q} = J_v^+ \dot{X}_v \quad (16)$$

For the circular trajectory on the X plane (the same trajectory as in figure 17), with the gains $K_P = 5$, $K_D = 1$ and $K_I = 0.1$, the following end-effector results were obtained:

$$q_{initial}(s0..w2) = \begin{bmatrix} -\pi/3 & 0 & -\pi/3 & 0 & 0 & \pi/2 & 0 \end{bmatrix}^T \quad (17)$$

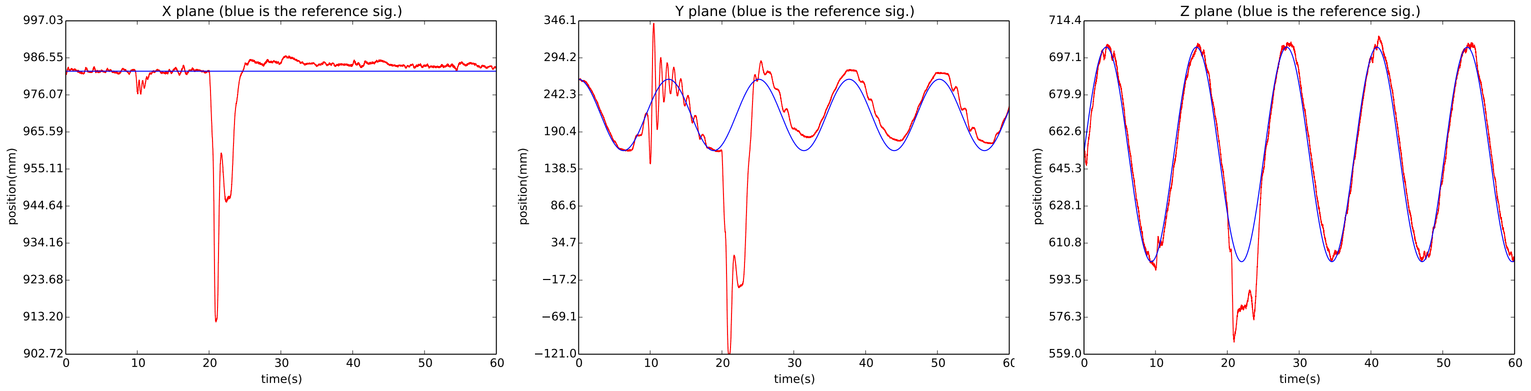


Figure 20: Plots of the end-effector position for a circular trajectory on the left arm, with velocity control, and with perturbations.

It can be seen from the plots that the process starts following the desired trajectory very well until it is softly disturbed by a perturbation at around the 10 seconds, and, however with some vibration, the process almost recovers completely. At around 20 seconds another, more abrupt, perturbation is applied to the robot, and it can also recover from it quite well.

One possible cause for the vibrations on the Y plane is the layout of the arm, since it's almost all stretched in the X direction/axis, and the Coriolis effect generated by the movement of the joints can be significant.

The null-space

Doing some job alongside Baxter, might call for a specific arm layout, that does not interfere with the environment nor with the workers. Because the arm is redundant, the available degrees of freedom can be used to position the arm in a specific orientation, without disrupting Baxter end-effector control. This can be done by expanding the control equation, and finally taking advantage of Baxter 7 DOF arms by controlling the null-space. The 7 degrees of freedom allow us to control the robot end-effector position (x,y,z) and orientation (wx,wy,wz), and leave one degree of freedom alone. This last one, is the one responsible for the redundancy, as it allows different arm joint layouts, that respect the same position and orientation (except in very specific

situations, in which the end-effector is said to be in a singularity). The robot can favour a joint layout, if this joint layout is given in an error vector $\dot{q}_0$.

Note: Using this type of controllers, the arm never reaches the singularity problems that happen in an Inverse Kinematics approach.

For example, using geometry, if it's desirable for the arm layout to be tending to the pose:

$$q_{desired}(s0..w2) = \begin{bmatrix} 0 & 0 & -\pi/2 & \pi/2 & 0 & 0 & 0 \end{bmatrix}^T \tag{18}$$

and the actual pose is q_{actual}, then an error vector can be generated as: $\dot{q}_0 = (q_{desired} - q_{actual})$. This error can be minimized (by the least squares) with a constant gain and a damped behaviour (desired velocity =0) as:

$$\dot{q}_0 = K_{Pq}(q_{desired} - q_{actual}) + K_{Dq}(0 - \dot{q}_{actual}) \tag{19}$$

And the equation can be expanded by adding a desired velocity based on this error, in the null-space:

$$\dot{q} = J_v^+ \dot{X}_v + (I_7 - J_v^+ J_v)\dot{q}_0 \tag{20}$$

Note: Once again, K_{Pq} and K_{Dq} are positive scalars applied to the values of a square identity matrix, and, depending on our intentions, some of the diagonal values can be neglect (put to zero), or controlled by taking more or less scalar importance than the rest.

The null-space used here, acts on the Cartesian velocity part of the Jacobian (J_v), since the equations are using the pseudo-inverse of that same part to control the end-effector position with $\dot{X}_v$ (the main task).

From the equation, $\dot{q}$ will be directly applied to the joints, respecting the hardware limitations 5.2.

Taking on the previous example (image 20), and applying the null-space using $Kpq = 0.8$ and $Kdq = 0.1$, and lowering Kp to 3 (so less vibrations are generated), the following results were obtained:

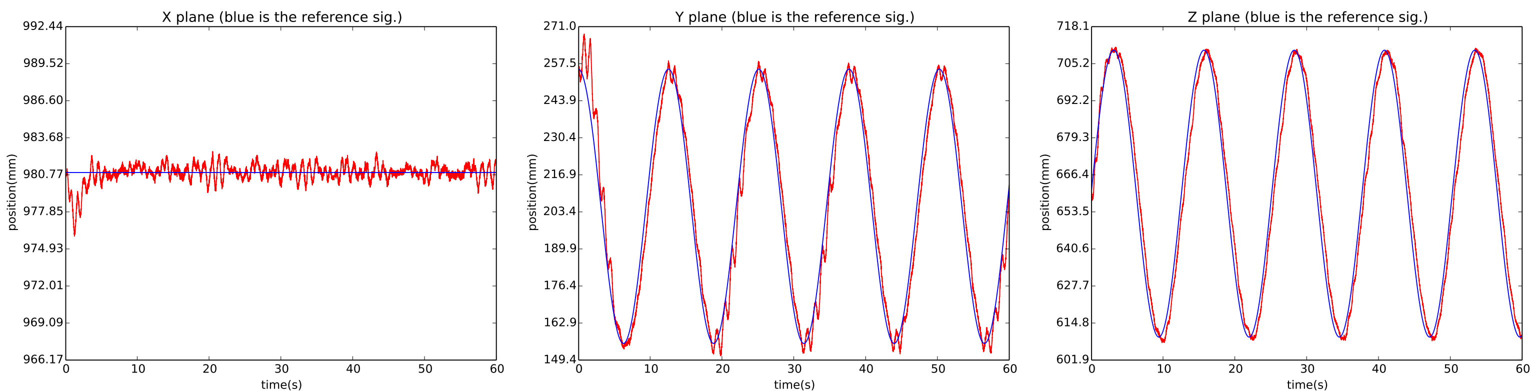


Figure 21: Plots of the end-effector position for a circular trajectory on the left arm, with velocity control, and with null-space arm layout control.

In the plots, it can be seen the arm following the reference signal quite well, however, the oscillations in the Y axis remained. It can also be seen that the position of the end-effector was but slightly oscillating at the beginning (more evidently in the X and Y planes), because of the initial arm adjustment to the desired configuration. This shouldn't be expected from the null-space, but this is where small imprecisions on the values used to build the null-space,

in addition to the error of $\dot{q}_0$, can disturb the independence of the equations, resulting in a deviation of the end-effector. The oscillations can also be originating from the physical motor's strings.

The plots can't show what actually happened differently from the previous example, but in fact, the arm is now turned left, reaching for the desired layout: $q_{desired}$, even though the end-effector remains trying to reach the objective/reference $\dot{X}_v$.

Controlling the cuff orientation:

One way to control the end-effector orientation is through the use of the part of the Jacobian related to the angular velocities (J_w). This part can be controlled in the null-space, without interrupting or disturbing the "main" control task that is: follow the circle trajectory. Having a good orientation of the end-effector might be more desirable than having a good layout of the arm. This is the case that is going to be studied. In order to achieve this, the previous null-space will now act on the end-effector orientation null-space. This way, the 3 controllers can be acting simultaneously and hierarchically, publishing just one velocity for each joint, at all times.

To get this controller to damp the movement and converge constantly to the desired angular position $\dot{X}_w$ without a stationary error, a PID was used, similarly to what has been done previously to control $\dot{X}_v$. This PID was proven to work with the relatively small axis errors used. This has been made by adding the terms to the equations:

$$\dot{X}_w = K_P w Orientation_{error} + K_D w(0 - J_w \dot{q}_{actual}) + K_I w \int_0^t (Orientation_{error}) dt \tag{21}$$

$$\dot{q} = J_v^+ \dot{X} + (I_7 - J_v^+ J_v)(J_w^+ \dot{X}_w + (I_7 - J_w^+ J_w)\dot{q}_0) \tag{22}$$

The $Orientation_{error}$ was obtained, through the quaternions given by the function from the Python class "Limb": endpoint_pose() at each instant. An error vector can be obtained by using, directly, the relations between the desired quaternion and the actual quaternion ($Orientation_{error}$). For safety (avoiding abrupt movements), the methodology for the calculation of $Orientation_{error}$ was the following:

Let a quaternion be represented by $Q = \begin{bmatrix} qv & qw \end{bmatrix}^T$, and $qv = \begin{bmatrix} qx & qy & qz \end{bmatrix}^T$

Another way of representing quaternions is through a rotation matrix (R), and, reciprocally, an axis-angle representation (θ,$Axis$).

A more detailed explanation is that, at every given moment, the end-effector orientation differs from the desired orientation by some specific rotation matrix R. The following rotation matrix relations can be derived (considering the rotation matrix properties):

$$RR_{actual} = R_{desired} \quad \equiv \quad R = R_{desired} R_{actual}^T \tag{23}$$

The rotation matrix can be obtained by using the quaternions as follows (for the $R_{desired}$ similarly):

$$R_{actual} = \begin{bmatrix} 1 - 2qy_a^2 - 2qz_a^2 & 2qx_a qy_a - 2qz_a qw_a & 2qx_a qz_a + 2qy_a qw_a \\ 2qx_a qy_a + 2qz_a qw_a & 1 - 2qx_a^2 - 2qz_a^2 & 2qy_a qz_a - 2qx_a qw_a \\ 2qx_a qz_a - 2qy_a qw_a & 2qy_a qz_a + 2qx_a qw_a & 1 - 2qx_a^2 - 2qy_a^2 \end{bmatrix} \tag{24}$$

R is a 3 by 3 rotation matrix, that can be mapped to the axis-angle notation. This allows us to make our desired error vector by:

$$\theta = acos(0.5(R(1,1) + R(2,2) + R(2,2) - 1)) \tag{25}$$

$$Axis = \frac{1}{sin(\theta)} \begin{bmatrix} (R(3,2) - R(2,3)) \\ (R(1,3) - R(3,1)) \\ (R(2,1) - R(1,2)) \end{bmatrix} \tag{26}$$

$$Orientation_{error} = \theta Axis \tag{27}$$

Note: This error cannot actually become zero, it only tends to a 0/0 indetermination.

If the desired end-effector orientation corresponds to the quaternion (which can be mapped to $R_{desired}$):

$$Q_{desired} = \begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \end{bmatrix}^T \tag{28}$$

this desired orientation can be fed to the controller from image (22), with $K_P w = 5$ $K_d w = 0.5$ and $K_I w = 1$, and the following results for a circular trajectory were obtained:

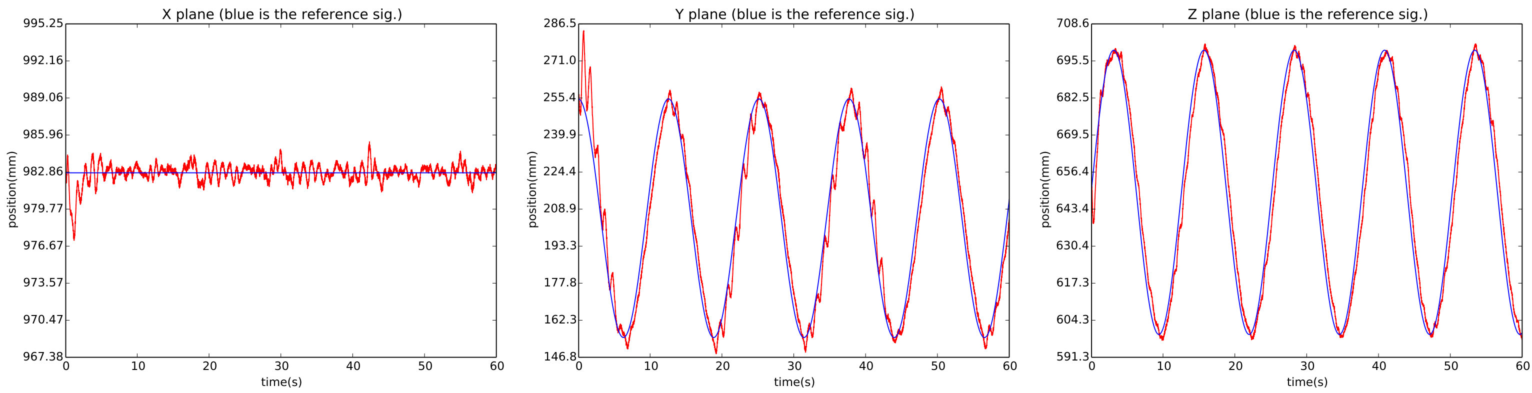


Figure 22: Plots of the end-effector position for a circular trajectory on the left arm, with velocity control, and with null-space desired configuration and orientation control.

No big differences can be detected between the previous plots and the plots from image 21, the only difference noticeable, is that the vibrations increased slightly in the Y axis (these were possibly inherited from the null-space arm layout control). The unnoticeable difference is that the end-effector is now turned 90º degrees.

In order to visualise better the evolution orientation error, the end-effector quaternion was transformed to Euler angles (rotation in the X, Y and Z axis). The corresponding plots containing the orientation error of the rotation of the wrist can be seen here:

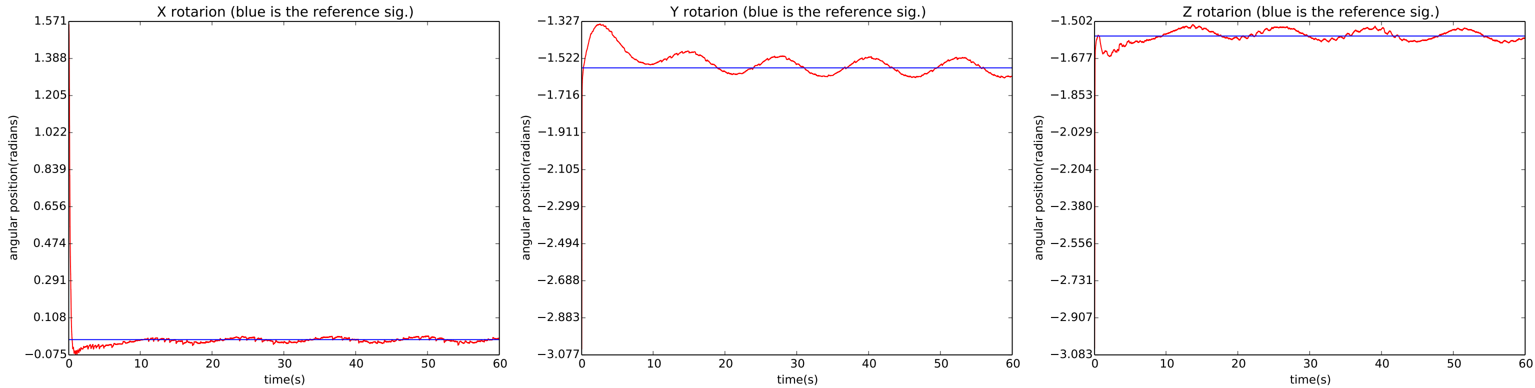


Figure 23: Plots of the end-effector orientation for a circular trajectory on the left arm, with velocity control, and with null-space desired configuration and orientation control.

It might be argued that the plots above don't seem to follow the reference well enough, but they aren't strictly supposed to. Because the position control was prioritized over the orientation control of the end-effector, the position control might interfere with the orientation control. This wouldn't have happened, if the controllers were both placed in the main task, however, it was preferred to have bad orientation over bad positioning of the end-effector. Anyway, the orientation errors were negligible, and, because of the integral component, they are decaying with time.

Note: The remaining vibration seen in the orientation plots, after the big "chunk" of the orientation error has been corrected, can be correlated with the oscillations generated by the main trajectory.

Continuing with the same control equation(22) (using the same gains), but now following a line trajectory with perturbations, the following results were obtained :

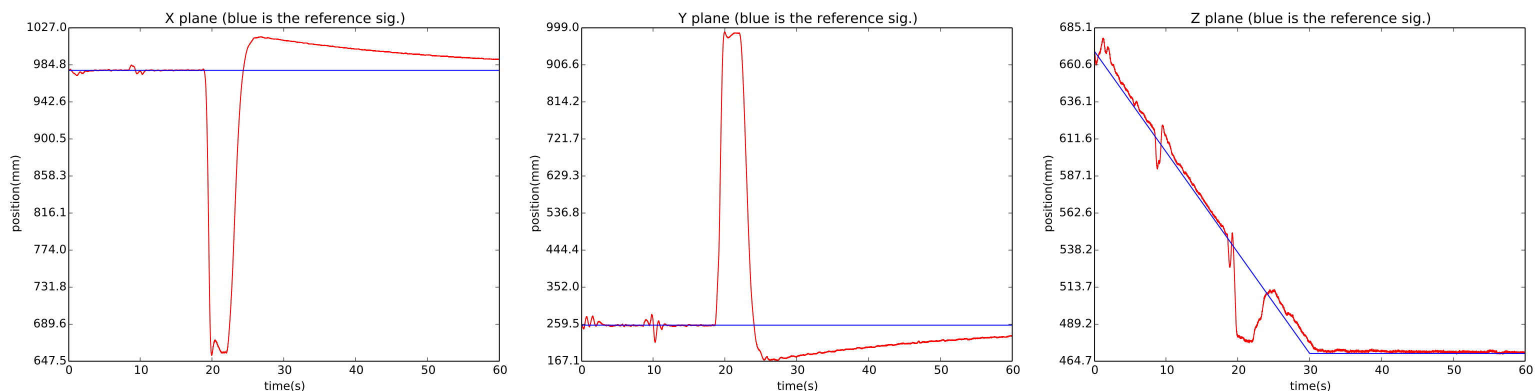


Figure 24: Plots of the end-effector position for a line trajectory on the left arm, with velocity control, with null-space desired configuration and orientation control, and with perturbations.

As it can be seen in the previous plots, the full control equation, is taking effect as well for a line trajectory. Three perturbations were applied to the arm, the first two were applied along

the arm, and the last one directly on the end-effector. Even subjected to the perturbations, the control of the layout and the end-effector position and orientation were maintained. It can also be seen, more precisely, the PID taking control and correcting the stationary errors. The orientation plots were very similar to the ones in figure 23.

6.3. Control by Torque

The objective from the beginning was to control Baxter with a torque (τ) controller, that can publish to all joints the torques needed to comply with a reference based on the end-effector position and orientation, as well as comply with the arm layout reference. The objective was set, because, by controlling the robot via torque, people can interact with the arm without it being rigid (contrary to all the other previous controllers). The controllers used on this mode had frequencies in the range: 200-1000Hz.

The algebraic physical relation between torque and force is:

$$\tau = J^T F \tag{29}$$

The equation can be similarly manipulated as it was for the velocity controller, with slight changes. For example, one of the advantages of using this controller, is that it can intake a desired Cartesian acceleration for the end-effector:

$$\tau = J^T(\Lambda(\ddot{X}_{desired} - \dot{J}\dot{q}_{actual}) + F_{desired}) \tag{30}$$

where $\Lambda = (JM^{-1}J)^{-1}$, and M is the 7 by 7 inertia matrix in joint space. M can be obtained, similarly as is obtained the Jacobain(J), using Orcos ([7]) "baxter_pykdl" library functions.

Because $\dot{J}\dot{q}_{actual}$ is usually just a source of noise, and it's almost zero (for small movements of the arm), adjustments to the equation can be made, without disturbing the equation general effect.

In order to run our main task in the Cartesian coordinate system, the equation can be rewritten as:

$$\tau = J^T(\Lambda_v(\ddot{X}_{desired}) + F_{desired}) \tag{31}$$

where $\Lambda_v = (J_v M^{-1} J_v)^{-1}$.

It's important to know, while using the equation above, that when $\ddot{X}_{desired} = 0$, the impedance controller "falls" back into a simple torque controller.

Everything is in place to control the end-effector force $F_{desired}$ with a PID controller:

$$F_{desired} = K_P(X_{desired} - X_{actual}) + K_D(\dot{X}_{desired} - \dot{X}_{actual}) + K_I \int_0^t (X_{desired} - X_{actual})\delta t \tag{32}$$

For the previous circular trajectory (in figure 17), with R=5cm,

$$q_{initial} = \begin{bmatrix} -\pi/3 & 0 & \pi/3 & 0 & 0 & 0 & 0 \end{bmatrix}^T \tag{33}$$

$$X_{desired} = X_{initial} + \begin{bmatrix} 0 & R(cos(0.5t) - 1) & Rsin(0.5t) \end{bmatrix}^T \tag{34}$$

$$\dot{X}_{desired} = \begin{bmatrix} 0 & -R0.5sin(0.5t) & R0.5cos(0.5t) \end{bmatrix}^T \tag{35}$$

$$\ddot{X}_{desired} = \begin{bmatrix} 0 & -R0.25cos(0.5t) & -R0.25sin(0.5t) \end{bmatrix}^T \tag{36}$$

and using the gains: $K_p = 130$, $K_d = 20$ and $K_I = 10$, the following plots were obtained:

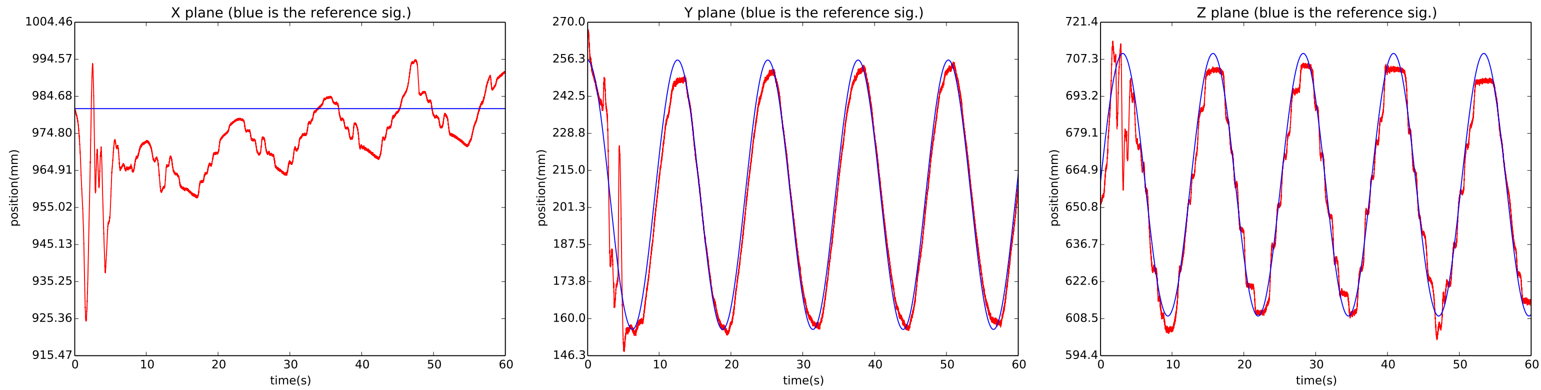


Figure 25: Plots of the end-effector position for a circular trajectory on the left arm, with torque control.

As it can be seen in the plots, the controller is following the reference trajectory, but not without its flaws. It was challenging tuning the controller to follow the reference signal, without succumbing to the joint noises. However, the noises on the first 5 seconds can be traced to the adjustment of the arm layout. It is good to keep in mind, that, because the arm is not rigid, it is now susceptible to the effect of "environment" forces (like gravity, and Coriolis effect), and little noises become very significant to the output result. Because of this, there was applied a 30Hz low-pass filter, to take away some of this noises coming from the read joint velocities $\dot{q}_{actual}$ (which is already in use in the plots above).

A problem associated with this control, as it is, is that the arm has no restrictions to hold some of the joints angular positions in place, since the robot is redundant. This is were the null-space control can be used to solve this problem, even if the gains are low. In the case of using torque control, the equation for the null-space must be changed to accommodate the dynamically consistent null-space matrix:

$$\tau = J^T(\Lambda_v(\ddot{X}_{desired}) + F_{desired}) + (I - J_v^T \Lambda_v J_v M^{-1})\tau_0 \tag{37}$$

where τ_0 can be controlled analogous to $\dot{q}_0$:

$$\tau_0 = K_{Pq}(q_{desired} - q_{actual}) + K_{Dq}(0 - \dot{q}_{actual}) \tag{38}$$

Note: As it happens with all the null-space control approaches, they take in account the arm geometry, but they do not take in account the arm physical angle limits. This might result in the arm trying to follow a trajectory that will get the arm "stuck" on its joint limits. If for some reason the arm gets "unstuck" from one of these positions, it might do some wild movements to readjust itself to the equation inputs/restrictions, therefore it's recommended to do a good tuning of the controller gains beforehand.

For initial joint angles:

$$q_{desired} = \begin{bmatrix} 0 & 0 & -\pi/2 & \pi/2 & 0 & 0 & 0 \end{bmatrix}^T \tag{39}$$

and for the gains $K_{Pq} = 2$ $K_{Dq} = 1$, the following plots were obtained:

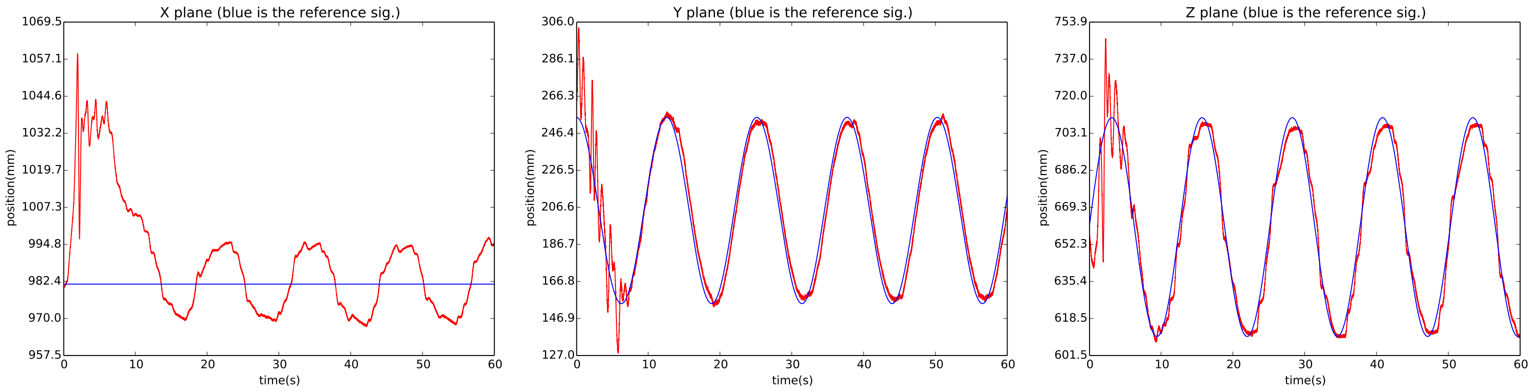


Figure 26: Plots of the end-effector position for a circular trajectory on the left arm, with impedance control, and null-space control.

No great differences can be observed between these plots and the previous ones from image 25, however the reference signal is still being followed, as intended, while the arm layout took its time to adjust itself (within the first 5 seconds). The torques generated through the arm that control the null-space weren't that dangerous, because the gains used do not allow great torques.

In order to implement the the orientation control of the end-effector, as similarly done on the velocity controller (27), the same $Orientation_{error}$ can be obtained, and the **final** control equation can be expanded to:

$$F_w = K_P w Orientation_{error} + K_D w (0 - J_w \dot{q}_{actual}) + K_I w \int_0^t (Orientation_{error}) dt \tag{40}$$

$$\tau = J_v^T (\Lambda_v (\ddot{X}_{desired}) + F_v) + (I - J_v^T \Lambda_v J_v M^{-1})(J_w^T F_w + (I - J_w^T \Lambda_w J_w M^{-1})\tau_0) \tag{41}$$

In the equation above: $F_v \equiv F_{desired}$.

This way, the 3 controllers are working simultaneously and hierarchically, without overlapping, as before for velocity.

The following plots prove that this control is fairly robust, and can posteriorly be used for more demanding tasks (for example, attaching the AR10 hand to the end-effector).

The plots were made using $K_P w = 0.8$, $K_D w = 0.5$ and $K_I w = 0.1$

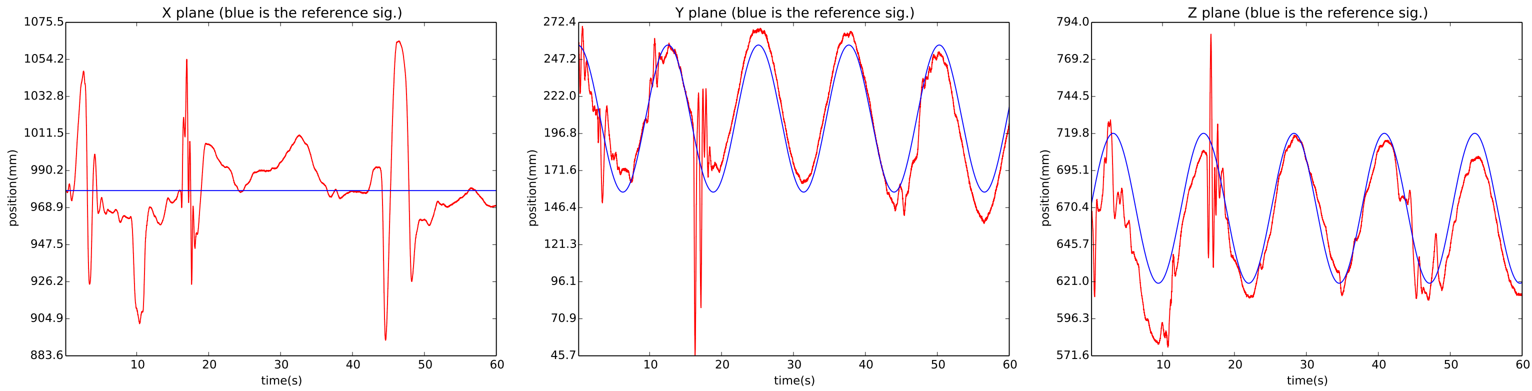


Figure 27: Plots of the end-effector position for a circular trajectory on the left arm, using the torque controller, with perturbations; where the end-effector, the null-space orientation and the arm layout are being controlled.

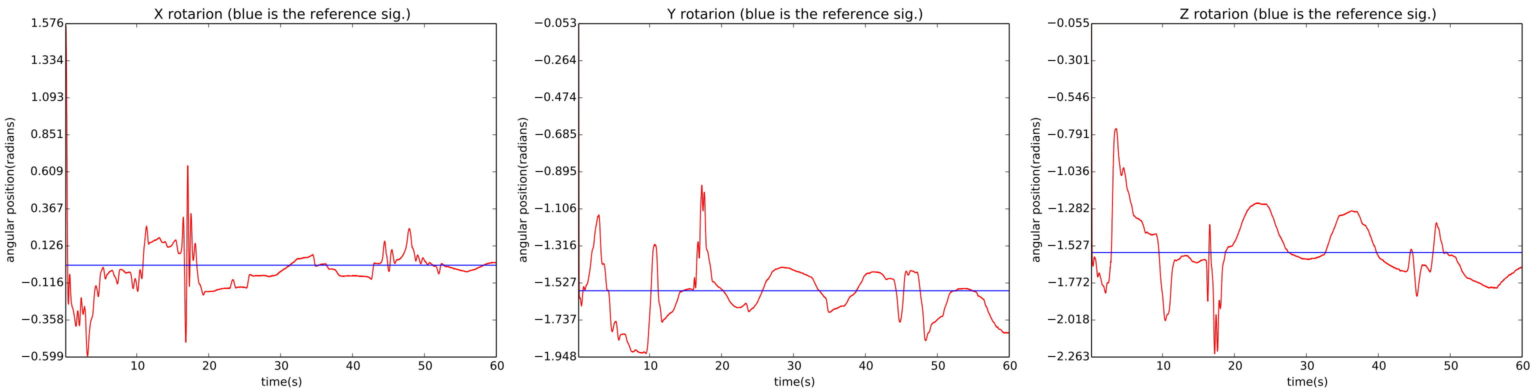


Figure 28: Plots of the end-effector orientation, for a circular trajectory on the left arm, using the torque controller, with perturbations; where the end-effector, the null-space orientation and the arm layout are being controlled.

In the previous plots, after the adjustment to the orientation of the end-effector (first 5 seconds), 3 types of perturbations were applied. The first perturbation (around 9 seconds) was applied on the side of the arm (joint e0 to be more precise), the second (around 17 seconds) was a direct bump on the end-effector, and the third (around 45 seconds) was a push on the X direction applied on the end-effector. This last one, was to show that the controller can act individually in one axis, leaving the Y and Z tracking almost intact. These last plots seem to show the effectiveness of the controller since it was able to recover from all these perturbations, while still trying to follow the reference signal.

Having established good controllers for the arm, the following sections will take advantage of this controllers, to get Baxter robot to conduct useful tasks.

7. Kinect Sensor and its usefulness with Baxter

Kinect it's a very compact match up of sensors that can give, at each instant, a RGB-D image. It also contains a microphone and a tilt motor, but these weren't used in this work. The advantages of integrating Kinect with Baxter, are that it can map points in a 3D Cartesian space, using its depth sensor. In the previous sections the controllers have been using a 3D (x,y,z) point as an input, now, using Kinect, a point can be fed to the controller, at all times, associated with, for example, a specific object. This is not a trivial process, since these points have to mapped from the Kinect axes to match with the coordinate system of Baxter. Another problem, is also to find the desired object in the workspace with the sensory information available. The Kinect version used for this work was Kinect v1.0.

To facilitate the integration of Kinect with Python, so it would be directly compatible with Baxter code, the "libfreenect" library was used. (To facilitate the manipulation of images, there were also used OpenCV functions)

7.1. Finding objects within an image

There are many subjective ways of choosing the best algorithm to find certain features in an image. There are ones that fit better than others depending on the specific situation and application. In our case, two "main" constraints should be respected:

- the time to process and find the object in the image should be minimal, and the results should be received by the controller efficiently, in a way they do not delay their cycle;
- the method must be robust, in order to be the most independent possible of the light being applied to environment under observation;

These are the reasons behind the methods chosen, and those will be explained in the following sections.

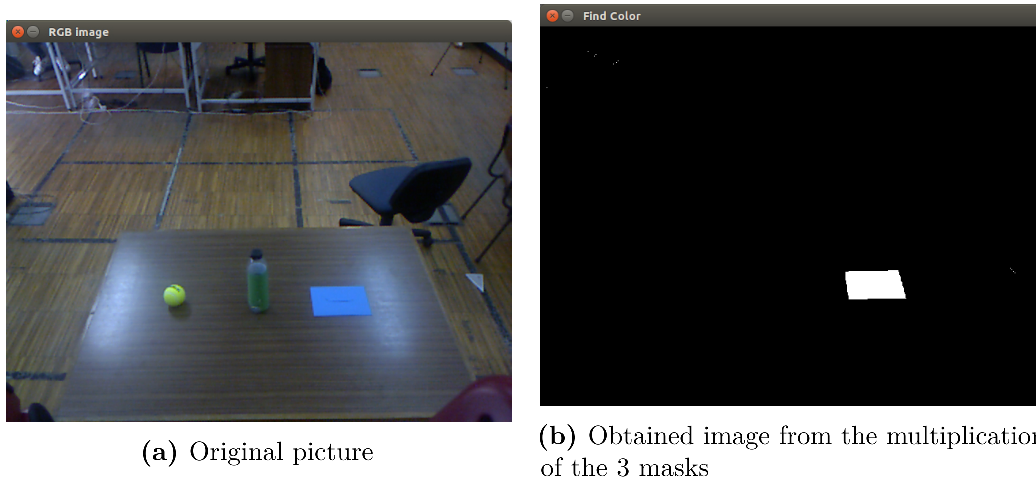
To better track an object and its orientation, methods to find features (such as pattern and symbol recognition in between others) should have been preferable to the methods used, however, the "main" focus of this thesis wasn't to find the best of this methods, but to show an application where Baxter integrates the Kinect sensor in order to find and interact with objects in the workspace.

There will be shown two methods to find in a given image a desired color, this color is supposed to be the color of the desired object. The essential difference between the algorithms/methods shown, is that one generates a binary masks using the pixel color value, and the other generates the masks using the pixel percentage value. These are the color intervals and color histogram methods, respectively.

7.2. Object tracking with Kinect using color intervals

A common and easy approach to find portions of a picture within a certain color range is using color intervals. Kinect sensor provides us with an 8-bit RGB image, meaning that the color of the object will always be in a certain range of these Red, Green and Blue quantities, that vary between 0 and 255. The paradigm is to find in the picture all the pixels in which Red, Green and Blue values are within a certain range. This information can be, posteriorly, turned into a binary mask with the target/object pixels turned to 1(white).

To find the desired color in the image, the intervals must contain that color. As an example, if the objective is to find a specific shade of blue, the intervals can be chosen to be: R=[0:140], G=[0:140], B=[200:255]. Then, using OpenCV "inRange()" function, the three binary masks resulting from these three intervals can be multiplied together. The results associated with these intervals can be seen in the images:



(a) Original picture

(b) Obtained image from the multiplication of the 3 masks

Figure 29

This method is not without its limitations. If the overall ambient light provides more or less intensity (for example in environments exposed to sunlight, during the day and night periods), all the pixels RGB values will have an offset increase/decrease accordingly. This effect should be followed by the intervals set up, but they cannot always cope with the differential.

A better method that resists this effect is the color histogram, that depending on the application, should be preferable to this one.

The greatest advantage of the color intervals method is that it can detect all the colors, even in the gray scale range, the same cannot be said for the color histogram method.

7.3. Object tracking with Kinect using the color histogram

A common approach to find portions of a picture within a certain color ratio, is the color histogram method. This method finds the ratio of Red, Green and Blue on a specific pixel. As an example, let $Ratio_R$ be the ratio of Red in the pixel (i,j) from the image:

$$Ratio_R(i,j) = \frac{R(i,j)}{R(i,j) + G(i,j) + B(i,j)} \tag{42}$$

A problem arises when the pixel (i,j) is completely black, which results in a division by zero, but that problem can be easily bypassed, because the numerator will also be zero.

A clear limitation of this method is that all the pixels in the gray scale will converge to the same value. Therefore, depending on the circumstances, it might be preferable to use color intervals method.

Note: Because the Baxter robot model used for the tests was red, it is not recommended to use this method to search for red, or near red pixels, in order not to be confused with Baxter's arms.

The greatest advantage of this method is that it doesn't depend on the ambient light intensity, **as long as this light is mostly white** (so it doesn't disturb the color ratios of the object), and **as long as it it doesn't "flood" the object with the color of the light** (which can neutralise the object natural color).

In order to look for a specific color in the RGB picture using this method, it must be known in advance, two of the desired ratio ranges of the object being looked for. The third ratio range can be deduced by the remaining ratio that respects the truncated limits 0 to 1. By definition, the ratio of a color must be between 0 and 1 (inclusive), and our desired values in between. Three of these intervals can form a sub-volume of the normalized RGB space that can be parsed and filtered from the image. Because this normalized RGB space derives from images using discrete values between 0 and 255, the RGB space can be compared to an histogram with 255x255x255 maximum bins (thus the method's name).

As an example, if searching for a green covered bottle, the values of the ratios can be:

$$P_{Bmin} = 0.3125 \qquad P_{Bmax} = 0.46875 \qquad P_{Gmin} = 0.375 \qquad P_{Gmax} = 1.0 \tag{43}$$

The Red values can then be deduced by respecting the normalization (of course that this ratio must always be truncated to the space [0,1]).

$$P_{Rmax} = 1 - P_{Bmin} - P_{Gmin} \qquad P_{Rmin} = 1 - P_{Bmax} - P_{Gmax} \tag{44}$$

While trying to find a green bottle, the "volume" of colors that was intended to extract from the RGB space forms a shape of a parallelepiped, that can be represented in cyan in the picture:

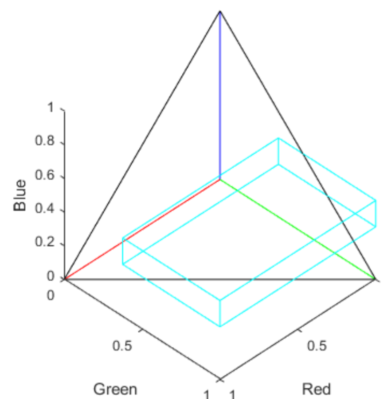
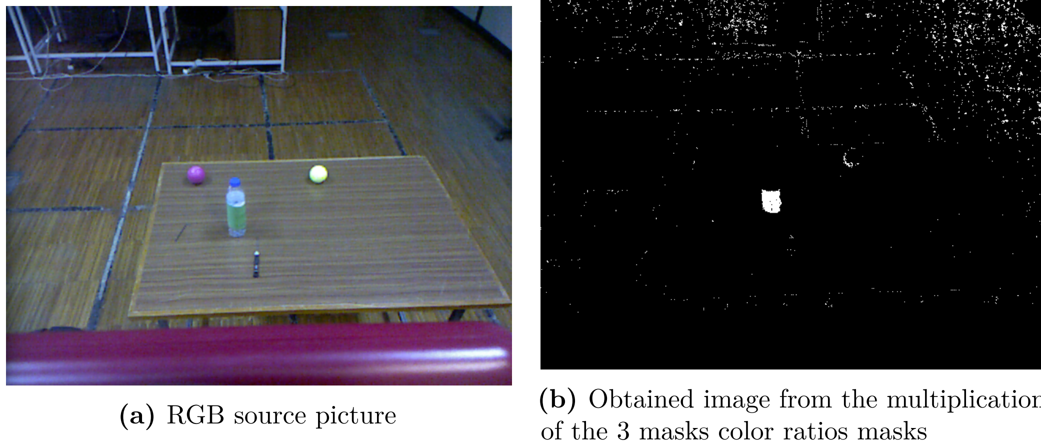


Figure 30: Image representing a 3 dimensional space formed by the components R,G and B corresponding to the axis. The volume of the chosen color is limited by the black pyramid, since the values can never leave it, respecting the ratio interval [0,1]

The cyan parallelepiped is the respective volume that was sub-selected/filtered from the image, as it is also the respective portion of pixels belonging to the object.

Now, having the "volume" of the colours being looked for, OpenCV functions can be used to obtain the desired mask of pixels corresponding to the object. A fast way to do this, is to find the desired mask, by multiplying each of the 2 binary masks (blue and green) that have pixels with colors ratios within the defined limits. The pixels being looked for, can be narrowed down, by also multiplying the resulting mask, for the red mask. An example picture, obtained with the 3 masks multiplication, can be seen in the image:



(a) RGB source picture

(b) Obtained image from the multiplication of the 3 masks color ratios masks

Figure 31

7.4. Object 3D mapping using Kinect

The natural way of using Kinect to find the (x,y,z) position of a pixel referenced in Kienct's origin, is to use the PointCloud2 ROS message published by Kinect. This message can start being published from running "roslaunch", with "freenect-registered-xyzrgb.launch" (which was the driver used to read data from Kinect, using ROS indigo). Unfortunately, the full process of capturing the (x,y,z) + (r,g,b) + (i,j) information associated with each pixel using the PointCloud2, was very time consuming, therefore, it was preferable to use "libfreenect", which is a library for python, that can capture Kinect images in a running loop, independent from Baxter's ROS environment. Using "libfreenect" allowed to launch the Baxter process, containing the controllers, alongside the Kinect processing, by launching just one main process, that would be parent to the peripherals. The Baxter python process can then communicate with this Kinect process sharing a Queue(). The error between the information captured from the method that will be used, relative to the one using the Pointcloud2, was very low (in the order of 1mm). Both methods worked very similarly, and both could achieve the final desired results. It was made the choice of choosing the method that favoured: compactness, simplicity and speed.

Following up from the previous section, after obtaining the desired binary mask using one of the previous methods, this mask must be filtered from noises. Erosions and dilations can be applied on that mask, to filter undesired noises, but in order to find the desired object in the mask (which should correspond to the biggest white spot), the one white spot with the greatest area must be sub-selected. After this, its geometric center/momentum can be found, to get the object most reliable position in (i,j) coordinates from the image. (OpenCV provides functions that do just that, like "contourArea()" and "moments()")

This (i,j) point/pixel can then be placed over the registered depth image (33b), which can then be translated to the object distance from Kinect in millimetres. In the case that Kinect is not detecting our one pixel distance properly, it was considered a 5 by 5 pixels square window centred around that (i,j) pixel. That window of 25 pixels/distances can be averaged to give a more reliable distance results for the object. This is more critical if the surface being looked for is not plain.

In the pictures on the left, there can be seen the masks obtained from filtering (with erosions and dilations, and selection of the greatest "island"), the corresponding place of object being searched for in the images.

In the pictures on the right, there can be seen the overlapping between the 5 by 5 window (with its center in the center of the object) and the registered depth image (with a blend of 50%).

(a) Image (b) after small filtering, with erosion and dilation

(b) Blend between distance image and desired position window

Figure 32: Adaptation of image 29a, filtered in order to find the center of the blue envelope

(a) Image (b) after filtering, with erosion and dilation

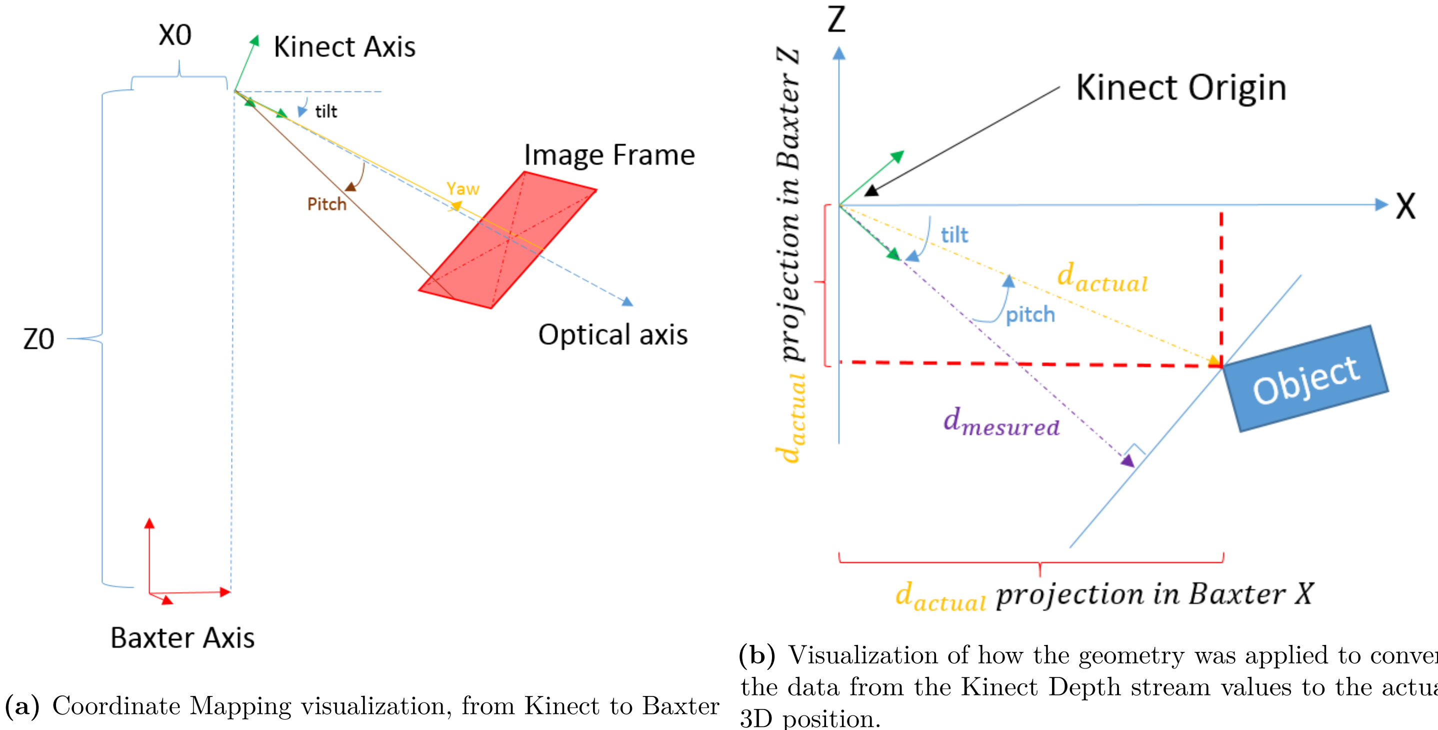
(b) Blend between distance image and desired position window

Figure 33: Adaptation of image 31a, filtered in order to find the center of the green covered bottle

After obtaining the correct pixel corresponding to the center of the object, that pixel can be overlapped with the registered depth image to produce a 3D (x,y,z) point, using geometry.

It is known from the official Microsoft site, that Kinect sensor images capture pixels in a spread (FOV) of +/- 57º degrees horizontally and +/- 43º degrees vertically, from the optical axis. These correspond to the yaw and pitch angles, shown in image 34a, respectively. It's also known that Kinect can tilt a maximum of -27º degrees downwards, but because of Baxter's shape, and the working space in front of it, a plastic piece was inserted beneath the Kinect that adds, approximately, -22º degrees to the -27º degrees tilt down. This configuration, however good enough, makes the camera capture a little portion of Baxter's LCD screen (as seen in 31a), that might be undesirable, and to reduce this effect, the Kinect tilt down was instead set to -10.8º degrees, approximately (making a total of -32.8º of tilt).

In order to more acculturate describe the field of view (FOV) of the RGB image, the yaw and pitch angles were changed to +/- $\simeq 32.5^o$ and +/- $\simeq 26.5^o$, respectively. These angles were chosen, because they generated, almost exactly, the same values given by the ROS PontCloud2 message, when generating (x,y,z) registered points (with approximation errors of 1mm).



(a) Coordinate Mapping visualization, from Kinect to Baxter

(b) Visualization of how the geometry was applied to convert the data from the Kinect Depth stream values to the actual 3D position.

Figure 34

The way to map the the pixel (i,j) to (x,y,z) coordinates, that was found best, was:

let [N,M] be the image size in pixels [640,480] respectively, $d_{mesured}$ the Depth information related to the (i,j) pixel, and the offsets $X0 \simeq 0.01m$, $Z0 \simeq 0.95m$ taken from the previous image 34a:

$$yaw = -32.5(-1 + 2i/N) \quad pitch = -26.5(-1 + 2j/M) \quad tilt = -(22 + 10.8) \tag{45}$$

Because the Kinect is tilted, and because of its properties shown in 34b, adjustments must be made to accurately calculate the depth of each pixel, and how that transforms to Baxter's

root 3D coordinate system:

$$X_{Baxter} = X0 + d_{mesured}\frac{cos(pitch+tilt)}{cos(pitch)}$$

$$Y_{Baxter} = d_{mesured}tan(yaw) \tag{46}$$

$$Z_{Baxter} = Z0 + d_{mesured}\frac{sin(pitch+tilt)}{cos(pitch)}$$

In order to get Kinect communicating with Baxter, without interrupting its processing, Kinect should be running in a separate Python process while doing the 3D mapping. This can be achieved by using a shared **Queue** between Baxter and Kinect, that delivers the last (X,Y,Z) reliable position, of the where and when the desired object was detected.

8. Touch sensors

The AR10 hands, however specific for Baxter, lack the capacity of perceiving touch. This makes it challenging to do repetitive pick-and-place tasks without the force feedback that the hand is applying on the object it is trying to grasp/pick. This might even result in not picking the object at all. Even if the object is picked, there is no way to know if it's being squished too hard, or too less (so it might fall). The use of Soft electronics can be very well applied to this situation, in which the best of two worlds can be combined: getting more grasp at the fingers of the AR10 hands and get a flexible (soft) electronic pressure sensor on those fingers, to give feedback of the pressure they apply on things.

Such sensor can be made by applying a fabrication technique that traps a conductor inside a flexible material. In this case the conductor will be an alloy of Indium-Gallium, and the flexible material will be layers of PDMS (*Polydimethylsiloxane*).

Figure 35: Sample image of one of the sensors utilized.

The disposition/shape of the trapped conductor, was specifically made to perceive a pressure when a force is applied perpendicular to it. The resistance of the metal increases with the expression:

$$R = \rho \frac{\ell}{A} \tag{47}$$

where A is the cross-sectional area of the resistor, ℓ is its length, and ρ is its electrical resistivity (constant through the alloy).

Looking at the equation, it should be obvious to understand that, by stretching the liquid metal, its volume remains and, therefore, the cross section area(A) decreases, and ℓ increases resulting in an increment of electrical resistance(R).

One of the restrictions of this sensor, because it is made of a very conductive alloy, is that its change in resistivity is very low, in the order of $1 \rightleftharpoons 5\Omega$, for natural and stretched, respectively. This means that if our sensor of measurement is based on the Voltage, there should be caution while applying electrical current through the sensor, in order not to overheat it.

An electric circuit that suits our needs is the following, considering the ideal OpAmp:

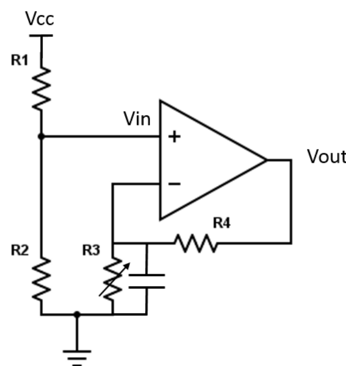


Figure 36: Circuit used to read the sensor, that doesn't overflow it with too much current, and takes advantage of its full range.

The circuit used, even though working on the region where the OpAmp is saturated, works in a way that doesn't damage the OpAmp, nor allows too much current to flow through R3, while still being sensitive to its resistance.

In order to get the digital feedback from this circuit, and integrate it with Python (via Serial port), it was included an **Arduino Mega 2560** which contains a lot of independent analogue input ports.

Arduino Mega 2560 has a 5V direct current (DC) source that I used as Vcc, and the circuit elements that I used were: $R1 = R2 = 1k\Omega$, $R4 = 60\Omega$, and the OpAmp used was the MCP6281. These elements were chosen to respect the circuit low resistivity, in order not to overflow it with current, and overheat it. The capacitor was inserted in the mesh in order to act as a low-pass filter to remove high-frequency variations of the sensor resistivity.

Using the circuit, any pressure being applied on the sensor R3, can be parsed using the Vout value. Arduino allows us to convert the analogue measurement of Vout, to a digital value within the discrete range 0 to 1024, correspondent to the range 0V to 5V. In order to be certain that a concrete touch was felt, it was used a filter that averaged the most recent 500 samples read from Vout. This average can give, with more certainty, the stable value of the resistivity of the sensor in its normal state -the reference state. When applying pressure to the sensor, Vout should increase, and, if above a certain threshold level, and if that level is sustained more than

once, the program running should "understand" that a "touch" has been perceived. The touch measurements do not enter the average in order not to affect the perceived reference state.

Note: This was the process used to detect a touch signal from the sensor. The focus of this thesis was not to get the sensor to accurately measure real pressures being applied to it, however it is capable of doing so.

Arduino can then send this pressure state to a Python process dedicated to read Arduino's Serial information, and this process can itself, share a Queue() with Baxter main control process. Baxter main control process can then, posteriorly send the information of "touch" to the AR10 hand/s.

9. AR10 Hands

The AR10 hand from *Activ8 Robots*, was given its name, because its movements are governed by using **10 Firgelli Servos (PQ12)**, that act like linear actuators, which are controlled by a **Pololu Maestro Mini 24 Servo Controller**.

In order to get the best frequency out of Pololu Maestro, its settings were altered so that the full loop, that goes through the control of each servo, tries to comply with a frequency of 1kHz. This enables a good communication with the Python process that reads the values from each finger encoder, and commands them to a desired position.

After testing the Pololu given examples for Python, the hands started showing off their limitations. They are relatively slow to reach their goal positions, and, sometimes, the goal commands need to be sent several times to obtain the desired response.

An annoying fact, was that, sometimes, the hands switched the serial ports they were being read from, and they were assigned to. It was also very hard to distinguish the right from the left hand, as they only diverge in the serial string number. These last problems don't occur when using the Pololu interface, however, using it, it's not a viable solution, since the hands need/should to be running in a Python process. This is not an unnatural approach, since there are given code examples to calibrate and control the hands with Python.

Some of the problems of the AR10 hand can be traced to its hardware, because the potentiometers used are filled with noises, and these problems are talked about in the AR10 Manual.

The AR10 hands purpose, in this work, is to grasp an object that cannot be as easily grasped by an ordinary gripper.

9.1. Controlling the AR10 hands on their own

The 10 servo motors of the hand, encode their digital position using a potentiometer. This potentiometer values vary from 600 to 0, approximately, but the hand digital position commands vary from 4450 to 8000, respectively to the potentiometer. **Note:** To be more intuitive to interpret how the command and the potentiometer readings relate to each other, the signal values that range from 4450 to 8000 were re-mapped from 600 to 0, respectively.

When the hand is fully closed, the servos are fully stretched, this corresponds to the encoder reading: 600; and the command to reach that desired position: 8000. These range of values aren't exactly the same for all the servos, but if the values are too high or too low, the Pololu Maestro will **truncate** them.

In order to exemplify how fast the hand is to comply with the given commands, in the next plot, it will be shown the process of: getting the hand to open completely from a neutral state, and, after that, getting it to close completely. To get the hand to move uniformly, the same command was sent to all the servos, thus, for simplification, the plot presented is relative just to one servo. Also in the plot, the **blue** signal is the servo encoder position, and the **red** signal is the reference signal (command):

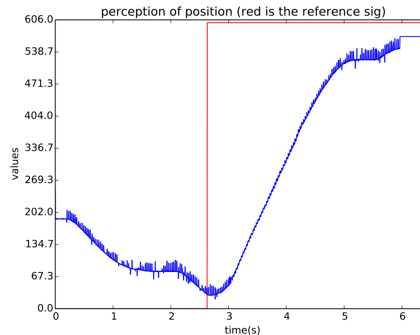


Figure 37: Normalized measurements from the sensors and command signal, for a simple open-close cycle of one servo.

In order to get the hand to open, the command of 4450 had to be sent to the servos. When the encoders read a value near from 0 (in this case the threshold was 40, because of the noises associated with the encoder), the reference command being sent changed to 8000. It can be detected a delayed response from the hand to the command, since the command started being sent as around 2.75 seconds, but it only took effect around 2.9 seconds. After the encoder reached around 600 (in this case the threshold was also 40), commands stooped being sent to the hand, and, to guarantee that the hand stopped moving at the end of the code segment, there was added a 0.5 waiting period. This period was necessary to add, because the hand could still be trying to reach its final command.

In this previous example, the only way to get a force sensing, was to force the hand to grab something, and figure out the difference between the command sent and the actual feedback encoder position. If these two remain stagnant for a while, and not in the hand limits, there could possibly be deduce that some kind of touch/encounter, stopped the hand from grasping further. This is not a very efficient way to sense touch, because, judging from the response time of the hands, one could be squishing the object too hard by the time the touch is actually felt. With the integration of "dedicated" touch/pressure sensors, the process closing the hand can be stopped almost at same moment it feels a threshold pressure on the fingers - just enough to get a good grip of the object.

9.2. Controlling the AR10 hands with touch sensors

The sensors form section 8, can be integrated on the tip of the fingers, to "answer" the signal of: "when to stop the grasping?". For a normal open-close cycle, continuing from figure 37, with a touch felt in the closing sub-cycle, the following plot was obtained. As before, the **blue** signal stands for the hand encoder position, and the **red** signal stands for the sent reference):

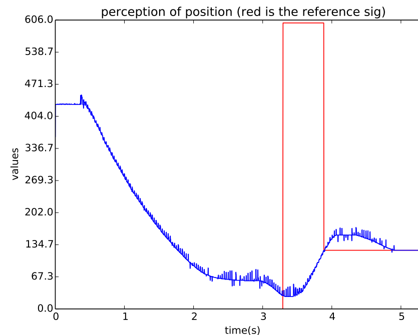


Figure 38: Normalized measurements form the sensors and command signal, for a grasping movement of the hand, with a touch perceived.

Because the hand started from a fully closed position, the servo tested took longer to reach the desired, fully opened, position (this took 3.9 seconds). After this, the hand was commanded to fully close, but, while grasping, a touch was perceived and the reference signal changed, exactly, to that same point where it was perceived. By doing this, the whole hand will try to regenerate the state where the object, it's trying to grasp, was "felt" for the last time. It can be seen in the plot that this state was achieved with a delay of almost 1 second (4th to the 5th second), and this delay can be traced back to the speed that the hands are limited to comply with the given commands. To guarantee that the object isn't dropped, a little more grasp was applied to the finger's encoders giving them -500 units (within the 4450 to 8000 range). After the finger reached a stability threshold, it must be commanded to stop, and hold that position, this is where the 0.5 seconds of waiting was useful, to guarantee that the hand stopped in the actual commanded/desired position, and not in previous ones.

The hand was noticed not to be very fast accepting commands, therefore a fast pick and place task should be best handled to grippers instead of the AR10 hand. The hands have the advantage of complying with many different configurations that are flexible for grabbing different types of objects, in different configurations, with a firm grip. Therefore, the decision between using a gripper or a hand, should be made accordingly to the task at hand. Furthermore, Baxter has its own grippers from *Rethink Robotics*, that are fully compatible with the robot, having no need for more Python processes.

10. Baxter Interaction with an Object

This is the section where all the knowledge gathered in all the previous sections, converges to the purpose of grabbing an object and land it safely in a landing site. In the image below, it can be seen a symbolic data flow schematic of the processes used, to govern the task of grabbing an object:

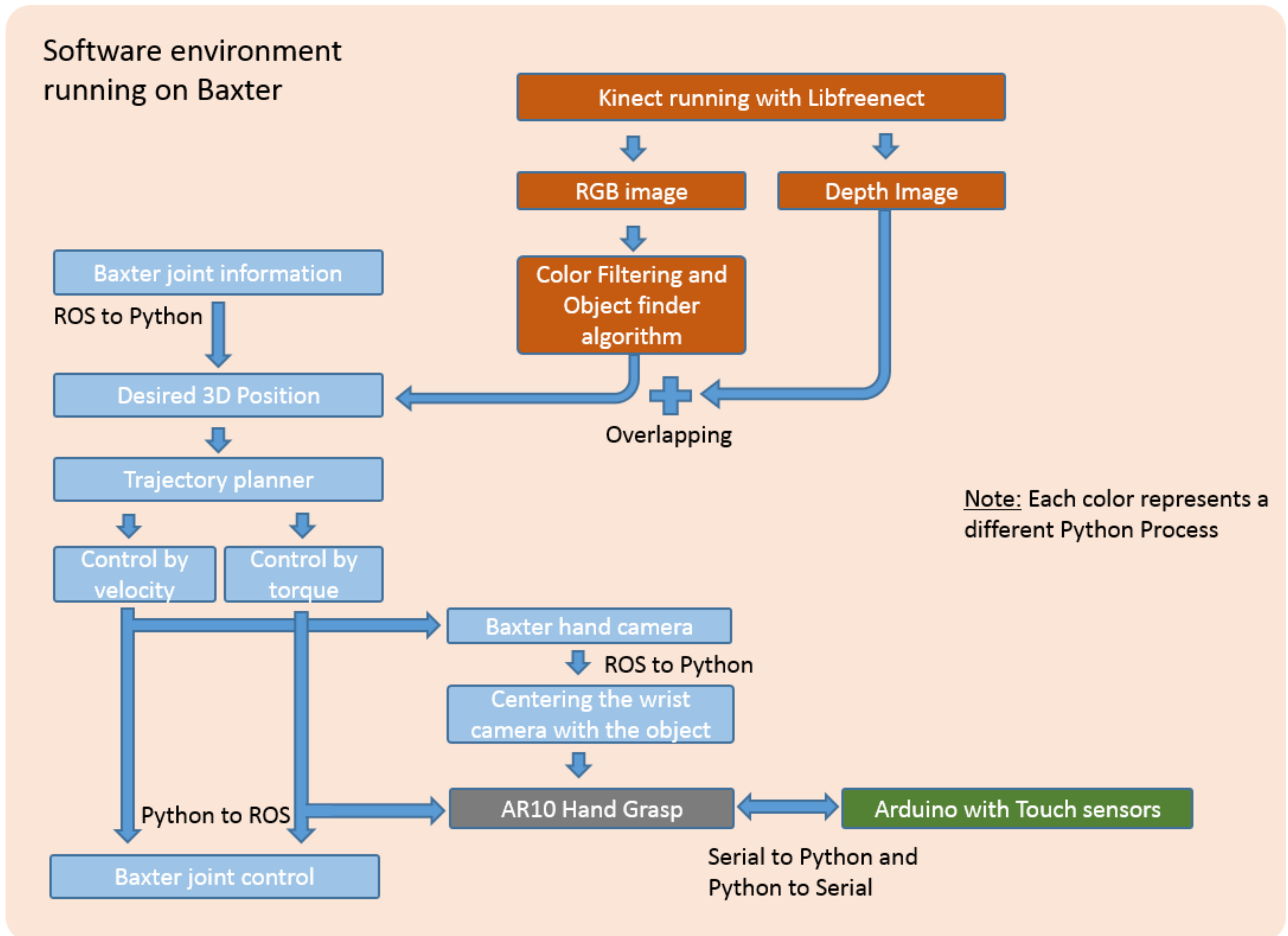


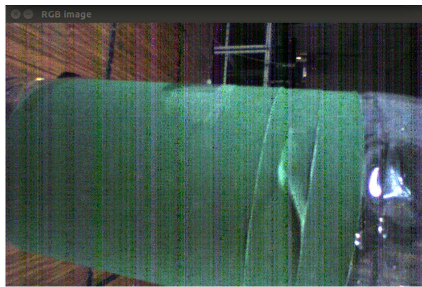
Figure 39: Data flow schematic

The object for the tests will be a green covered water bottle. This object was chosen because it's mostly a cylinder, it's deformable, it's easy to grasp by the hand, and the color is uncommon in our lab, so it can be easily found.

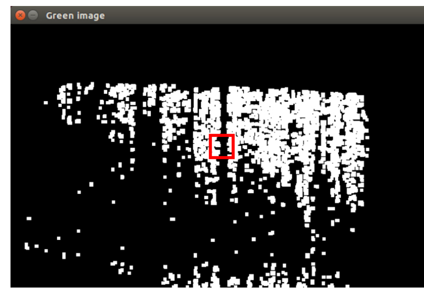
The technique used to obtain the "center" of this water bottle, was previously explained in the section 7.4. One inconvenience that comes by using Kinect on Baxter's head, is that Baxter's arms tend to occult the desired object (even controlling the null-space, and the arm layout), but this is not a great problem, since the task involved static objects. So, even if the position of the object is calculated only once, it gives a reliable reference of where the object might be "now", and, that's enough to get the hand to position itself very near from object.

when using the **velocity controller**, after reaching the proximity of the last position sent from Kinect, one can take advantage of Baxter's own cameras on the wrists. Trusting that the position and orientation controls of the end-effector are working fine, it can be known with precision how the axis of the camera is rotated and oriented, in relation to the origin axis. This means that images of the object can be obtained in front of the hand, and, by finding the object center, using the same methods used for Kinect, the distance from that center to the center of the image can be used to generate an error vector. This error vector can be minimized by using a small Cartesian velocity or force that does the fine adjustment of positioning the hand, right in front of the bottle. Adding to this, Baxter has an infra-red (IR) distance sensor on the wrist, that can give us the distance from that object, and this can be used to find the right distance to grab the object. (It can almost be said that Baxter has an RGB-D sensor in its wrists, for close distances).

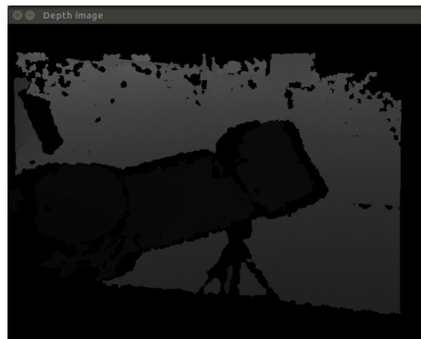
Adding the camera to the whole control process is not without its disadvantages. Because the camera is running on Baxter, and its data is arriving from the same "data bus" which transports all Baxter's ROS messages, using the camera adds to the cycle a delay of approximately 4 milliseconds. **This delay degrades the control frequency, and, in the case of the torque controller, makes it very hard to maintain a smooth movement of the arm/s**.



(a) Baxter camera right ahead from the bottle, twisted 90° to do an appropriate grabbing



(b) Processing the green in the image to find the bottle approximate center



(c) Baxter arm obstructing the object

Figure 40

Note: These pictures were taken without the hand in place, to maximize the interpretations of the pictures.

In any case, even without the wrist camera, the position given by Kinect is in coordinates (x,y,z), and it can be sent to the velocity or torque controller, so that the AR10 hand can position itself to grab the bottle.

In the specific case of velocity control, even if the 3D mapping from the Kinect was just "averagely" good, and only gave an "averagely" good approximation of the object position; the

Kinect information should place the hand right in front of the bottle, and thus the hand camera. If the bottle can be found in the images from the camera, fine adjustments can be done to grab the bottle, that's supposed to be right in front of the hand (as explained before).

While reaching for the object position, the orientation of the hand wrist plays and important role on grabbing the bottle efficiency. This can be controlled by controlling the end-effector orientation. If it's desirable that the arm avoids certain trajectories in the workspace (such as trajectories that get the arm stuck on its own joint limits), the layout of the arm can be controlled by using the null-space. The approach taken, will take advantage of every degree of freedom available in Baxter's arms. Furthermore, in case of a perturbation being applied to the arm, the controllers used can recover smoothly back to the desired trajectory.

Note: One should be careful not to reach the joint limits on the velocity control, because it might result in the arm stopping indefinitely.

Note: In the case of torque control, one should be careful tuning the gains of the arm.

To make the orientation control take effect slowly (through 10 seconds), an acceleration curve was added, so that there is an additional gain to the velocity and torque controller as shown in the equations:

$$K_e(t) = 1 - e^{-\frac{t}{2.5}} \tag{48}$$

$$F_w \equiv \dot{X}_w = K_P w \cdot K_e(t) \cdot Orientation_{error} - K_D w \cdot J_w \dot{q}_{actual} + K_I w \int_0^t (Orientation_{error}) dt \tag{49}$$

When the Kinect finds the object, there is a great chance that the object is far away enough from the end-effector, to generate a great error vector. In order to make this trajectory smoother, a linear and parabolic trajectories were made to interpolate the points between the arm starting position and the ending position. The parabolic trajectories had the advantage of reaching a certain position faster than the linear trajectories, this was sometimes usefull to get in position to grab the object. The gains were also adjusted to enable less aggressive movements of the arm (Kp and Kd). The configuration of the arm was kept sideways (as in 18), and the end-effector desired orientation was also kept (as in 28).

After reaching the best position to grab the bottle, grabbing will be done by compressing the 4 fingers of the hand until a touch is perceived from the sensor in the middle finger. When the touch is perceived, the fingers will stop in that position. A good solution was to use a shared Queue between the Arduino process, the Baxter process and the Hands process, that is responsible for sending a reliable touch signal to close the hand, just enough, to feel a certain pressure, while grabbing the bottle. **Note:** The thumb will be, from the beginning, closed in a way that the hand resembles a claw, making it easier to grab something.

After grabbing the object, the way to place it in a desired position, was achieved was by, then again, searching for a coloured landing site (blue in this case). The object can be placed safely on the landing site, by switching the desired position to be just above that of the desired site; then the hand can be opened smoothly, so that the bottle falls into place. After this, the arm can be raised just slightly, to separate the hand from the (placed) bottle.

To find the green bottle in the image, it was used the color histogram method, and to find the blue envelope, it was used the color interval method, these were previously explained in section 7.4.

The task in study, is divided in sub-cycles, respectively:

- **(before 0 seconds)**- in this sub-cycle, the robot goes to its starting default position. Just before going for the next sub-cycle, the Kinect is searching for the object so it can feed the object position to Baxter, before exiting.
- **(0 to 30 seconds)**- in this sub-cycle, the robot is reaching for the 3D position of the object, using a linear or parabolic trajectory, and one of the two controllers helps to follow this trajectory.
- **(30 to 37 seconds)**- in this sub-cycle, the process that handles the hand closing task, starts running, and closes the hand until a touch is felt from the middle finger sensor, or until the hand is fully closed.

  If using the velocity controller, the wrist camera can be taken advantage of (with chosen resolution of 640x400) to do fine adjustments to the hand position, reaching for the object 2D/3D position. When within a certain threshold of pixels (usually in the center of the object), the program, responsible for the task, can then proceed to the subtask that closes the hand. (This fine adjustment, might result on overpassing the 37 seconds deadline. However, the sub-cycle will only end when the center of the object is found, and the hand can start the grasping throughout around 7 seconds.)

  At the end of the 37th second, Kinect will deliver the position of the landing site to Baxter, just before before exiting this sub-cycle.
- **(37 to 90 seconds maximum)**- in this sub-cycle, similarly as in the beginning, the robot will use the controller to reach to the desired position. When just above this site, the robot opens the hand, and rises a little up in the Z axis, so it clears the hand from touching the bottle. The robot is supposed to end this last sub-cycle in 30 seconds. If the robot doesn't finish the task before the 90 seconds limit, or if the task was finished in the 30 seconds, the robot passes to the next sub-cycle.
- **(90 plus or less seconds)**- the robot returns to its starting default position.

Note: The robot should be controlled to move within its workspace.

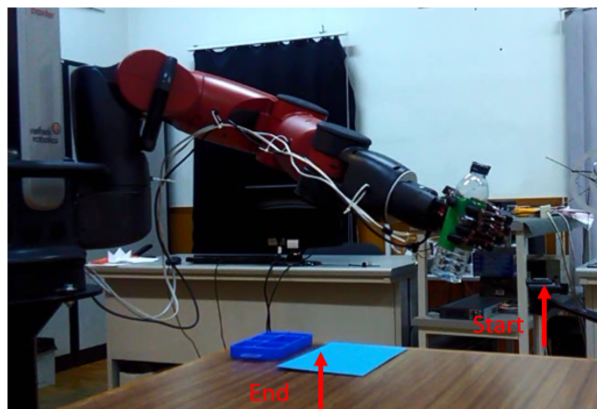


Figure 41: Image of the working space used for the examples. The "start" and "end", are associated with the bottle initial and final position

10.1. Interaction with velocity control

In the following examples, the velocity controller can be seen in action, performing the task of picking and placing a water bottle in a desired landing site.

Complete Control equation used:

$$\begin{gathered}
\dot{q} = J_v^{+}\dot{X}_v + (I - J_v^{+}J_v)(J_w^{+}\dot{X}_w + (I - J_w^{+}J_w)\dot{q}_0) \\
\dot{X}_v = 3(X_{desired} - X_{actual}) - 1J_v\dot{q}_{actual} + 0.1\int_0^t (X_{desired} - X_{actual})\delta t \\
\dot{X}_w = 5K_e(t)Orientation_{error} - 0.5J_w\dot{q}_{actual} + 1\int_0^t (Orientation_{error})dt \\
\dot{q}_0 = 0.8(q_{desired} - q_{actual}) - 0.1\dot{q}_{actual}
\end{gathered} \tag{50}$$

$$q_{desired} = \begin{bmatrix} 0 & 0 & -\pi/2 & \pi/2 & 0 & 0 & 0 \end{bmatrix}^T \quad q_{initial} = \begin{bmatrix} -\pi/4 & -\pi/3 & 0 & \pi/3 & 0 & 0 & 0 \end{bmatrix}^T \tag{51}$$

The linear and parabolic trajectories, taken for both torque and velocity, used to smooth out the movement of the arm towards the target/bottle, were the following:

$$X_{desired}(t) = X_{initial} + (X_{desired} - X_{initial})\frac{t}{30}\begin{bmatrix} -\frac{t}{30} + 2 & -\frac{t}{30} + 2 & 1 \end{bmatrix}^T \tag{52}$$

and when reaching for the landing site:

$$X_{desired}(t) = X_{initial} + (X_{desired} - X_{initial})\frac{t}{30} \tag{53}$$

where the time elapsed since the beginning of the trajectory is t, and it should be truncated at the end of the trajectory time-out.

These trajectories also helped the hand arriving at the desired position for grabbing the bottle, without tipping it over.

The results obtained were:

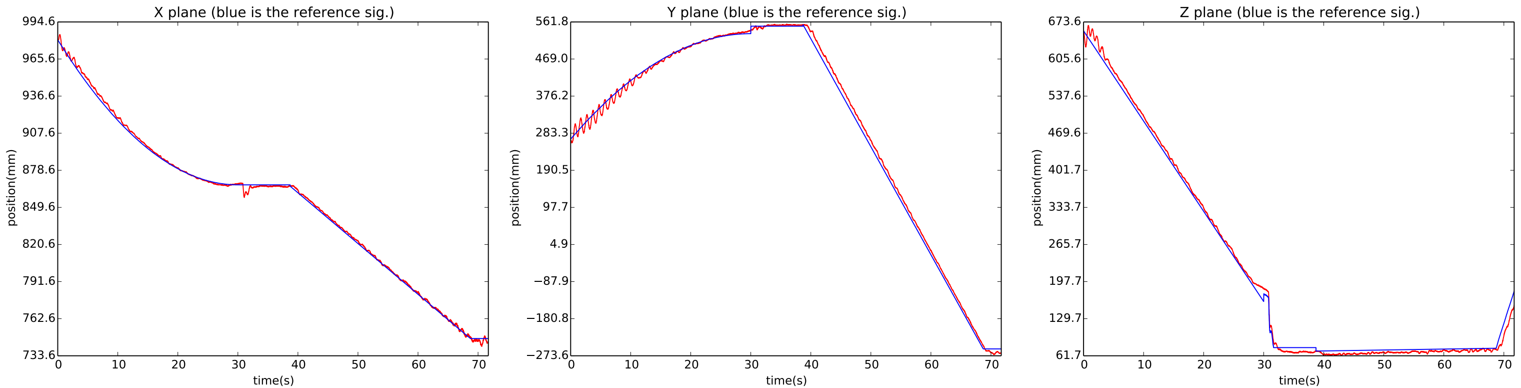


Figure 42: End-effector positions through one cycle of pick and place, using velocity control.

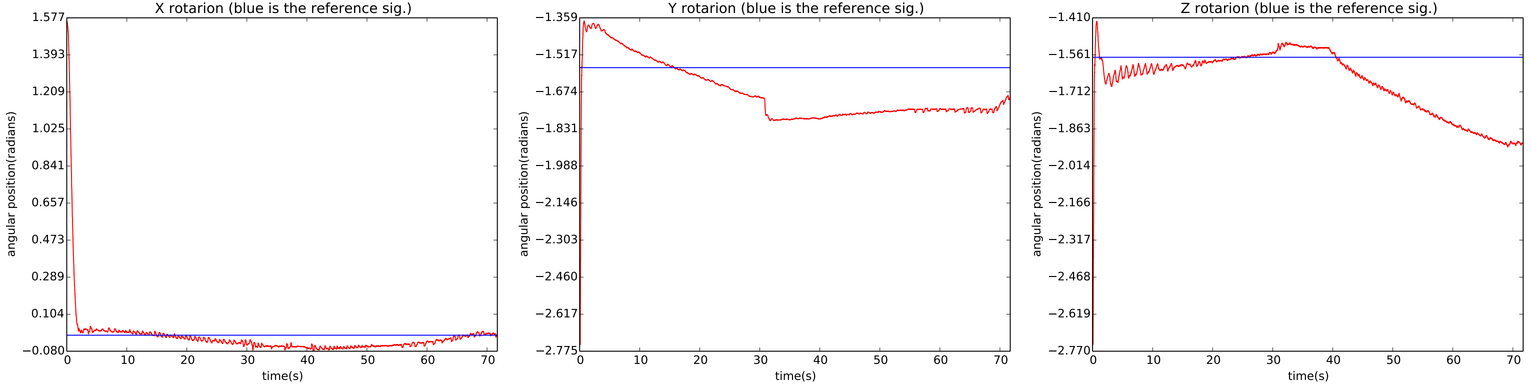


Figure 43: End-effector orientation through one cycle of pick and place, using velocity control.

By looking at the plots, it can be seen the remarkable tracking of the reference by the end-effector position controller. The orientation control of the end-effector, however without a very good "aspect" (a more detailed explanation of this effect, can be found near figure 23), gets the job done, and, at any moment, doesn't compromise the orientation of the bottle.

10.2. Interaction with torque control

In the following examples, the torque controller can be seen in action, performing the task of picking and placing a water bottle in the workspace.

The torque controller architecture is more difficult to implement than the velocity one (in some ways, because the arm is not rigid, and is facing external forces). Through testing, the solution that was found to be the most efficient way of controlling the position, orientation and layout of the arm, was to put the orientation and position control of the end-effector in the main task; the layout of the arm was posteriorly controlled in the null-space of the Cartesian part of the Jacobian. (Using the full Jacobian would have been more intuitive, but it was seen to give worst results. The null-space of the equation, the way it is, will act respecting only the end-effector position restriction). This method, prioritizes the orientation and position simultaneously, relative to the layout of the arm running in the null-space. The liabilities of using this method, is that the orientation control might slightly degrade the end-effector position control; and the layout of the arm, might slightly interfere with the end-effector orientation.

In order to remove the shaking of the arm, a low-pass filter of 30Hz was applied to the read joint velocities values. The noise associated with these measurements, was found to be the source of the problem. (A more detailed explanation of this filter is in section 3).

Complete Control equations used:

$$\tau = J_v^T F_{v_desired} + J_w^T F_{w_desired} + (I - J_v^T(\Lambda_v J_v^T M^{-1})\tau_0$$

$$F_{v_desired} = 150(X_{desired} - X_{actual}) - 20J_v\dot{q}_{actual} + 10\int_0^t (X_{desired} - X_{actual})\delta t$$
$$F_{w_desired} = 5.5K_e Orientation_{error} - 1J_w\dot{q}_{actual}) + 0.1\int_0^t (Orientation_{error})dt \quad (54)$$

$$\tau_0 = 1(q_{desired} - q_{actual}) - 0.75\dot{q}_{actual}$$

$$q_{desired} = \begin{bmatrix} 0 & 0 & -\pi/2 & \pi/2 & 0 & 0 & 0 \end{bmatrix}^T \quad q_{initial} = \begin{bmatrix} -\pi/4 & -\pi/3 & 0 & \pi/3 & 0 & 0 & 0 \end{bmatrix}^T \quad (55)$$

The results obtained were:

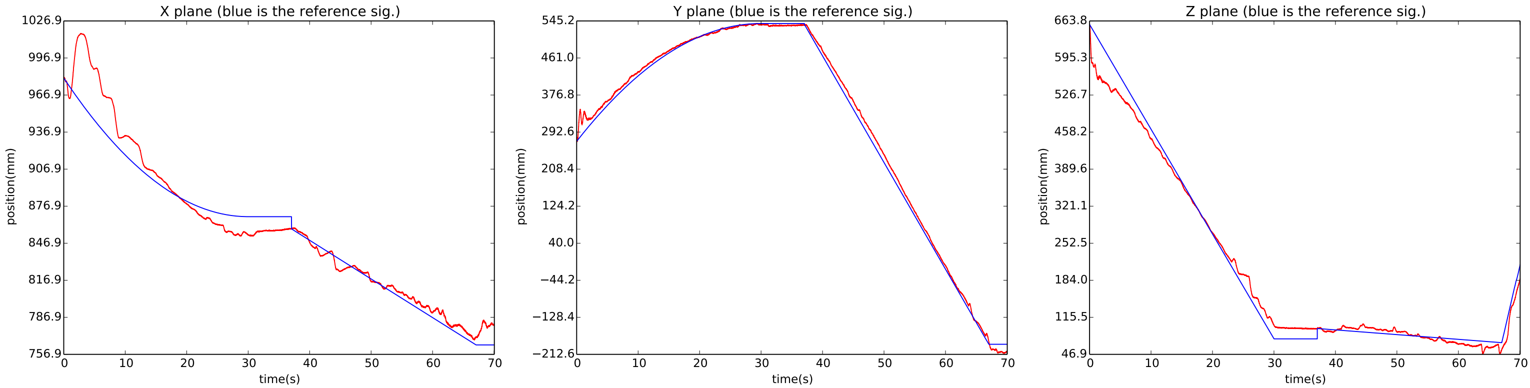


Figure 44: End-effector positions through one cycle of pick and place, using torque control.

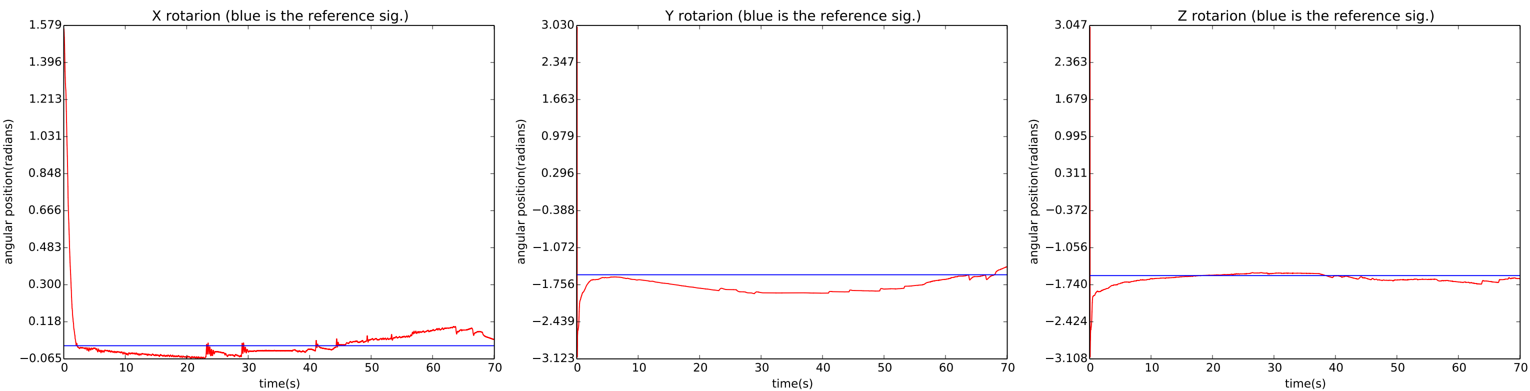


Figure 45: End-effector orientation through one cycle of pick and place, using torque control.

Looking at the plots, it can be seen the end-effector position controller tracking the reference quite well, considering the initial adjustment of the arm layout. By using the orientation control in the main task, it can be achieved better orientation results than the ones achieved with the velocity controller.

10.3. Comments on the fulfilment of the task, and the controllers performance

The task of grabbing a desired object in space and placing it safely on a desired surface was fulfilled successfully, and the controllers used were up to the task. The previous plots show that the controllers were working correctly, following the desired reference trajectories, while also respecting the null-spaces references being applied.

All the individual sub-cycles can be distinguished, in their corresponding time slots, with the exception of the time slot, in the velocity controller, where the wrist camera is making the adjustments to the end-effector.

The velocity and torque controllers, are successful examples that manage to prove that, the not so trivial task of "grabbing a bottle in the workspace without squishing it too hard, and placing it on the landing site", can be done with Baxter robot and its peripherals, in a safe

manner.

11. Conclusions

Being the first person to work with Baxter robot, with the specific soft-sensors and the AR10 robotic hands, in my department, was a very challenging process. Even more challenging, was to integrate everything together, running simultaneously (including the Kinect), at a pace that didn't disturb Baxter controllers frequency.

Many of the problems that I set myself to solve, like: applying the torque and velocity controllers to the arms; integrating the AR10 hands with the soft sensors; and integrating the Kinect object searching and 3D mapping, gave satisfactory results. The solutions found were applied in a manner that allowed Baxter to pick and place a bottle in the workspace, successfully, several times. This task was a prove of concept and I hope that it can inspire other researchers to do better, more complex, tasks using the concepts presented.

Some conclusions were taken along the thesis, and aren't specifically explained in this section. Many of these conclusions were taken about specific tests and examples, that had the final goal of applying the velocity and torque controllers, and, finally, complete the task of picking and placing the bottle/object. Some of these conclusions that can be taken out from this thesis, might be proven inaccurate in the future with further, deeper, study of the subjects treated, and with better implementation methods than the ones used. I was led to suspect that the majority of the conclusions taken hold true.

The work done shows promising implementations where Baxter robot can start to help in the industry and in households, side by side with humans, as a servant, safe, robot. From this thesis, the reader can take away not only good controllers that can comply with diverse trajectories, but also can command the Baxter's arms to remain still at a given position. If for some reason the arm should be flexible, there were also shown a wide range of ways to control this flexibility.

The following subsections, are about the general conclusions taken at each individual part of this thesis (the parts in question are in the titles).

11.1. Baxter Robot

Baxter robot is very human friendly, and I encourage it to be used in human working environments. In none of my tests I got hurt from interacting with the Robot, and, when in **Zero-G** mode, I could control the arms very easily with my hands. Even when recovering from perturbations, using the velocity and torque controllers, the arms returned smoothly to their position, without the use of dangerous torques.

The architectures of control tested and used along this thesis, may have real world implementations, and might even be better alternatives to the use of Inverse Kinematics while using Baxter (for example, the layout of the arm can be controlled, and the arm can be operating without being rigid)

However, I was also led to conclude that Baxter might not be the best robot to do precision tasks. The noises associated with the movement of the joints and the end-effector positions represent some of the liabilities of using Baxter robot. I do suspect that better results could have been achieved in the controllers if these noises weren't so abundantly prominent. Other thing that I was disappointed with, was the relatively low-frequencies that the robot operates on.

The most critical frequencies were the ones responsible for the publishing of the joint states, and the subscribing of the updated joint states, compromising the control cycles. These frequencies weren't also very stable, and depended on the task that the robot was doing.

Anyhow, the processes presented on this thesis show that Baxter, even with its limitations and flaws, can fulfil many tasks, and, depending on the applications, can even exceed many robots at these tasks. Many of Baxter limitations can be bypassed by good implementations and programming.

11.2. AR10 robotic hands

The AR10 hands, however resembling a human hand, don't act as fast as one. Their servos are slow, and don't always respond simultaneously. If I was to give the command for the hand to close, sometimes, some servos would "close" first and the others would follow up shortly after. This "shortly after" can be very significant, depending on the task, and I would not recommend the AR10 hands for precision tasks. There was also much noise associated with the potentiometers, that would encode the digital servo positions. I wouldn't also recommend the AR10 hands for tasks that might involve vibrating the hand in a way that can start loosing its nuts.

However, I must say that the hand itself gets the job done, as it can pose better than a claw gripper.

11.3. Soft Sensors

The soft sensors used were made by trial and error, until a reasonable solution was found. They were not specifically made for using with the AR10 hands, but they can fit nicely in the tip of the fingers. They are fair, stable potentiometers, that can sense pressure being applied to them, and the concept shows promise. However, better shapes can be devised for the sensors (in order to increase their efficiency), and make them more durable (without spilling the liquid metal), they are good enough solutions to give the AR10 hands a "sensation" of pressure, and a better grip.

11.4. Kinect

Using the Kinect alongside Baxter was a great add-on to the capabilities of the robot, as it allows Baxter to have depth information of the workspace in front of it. Maybe in the future, solutions involving coloured LIDAR's (Light Detection And Ranging) sensors might give Baxter a better understanding of its evolving environment, that extents to the full reach of its arms.

The image processing made with Kinect images was satisfactory, however, the filtering of the images and the object tracking gave noisy results, probably resulting from unstable results coming from the pixels's color and depth information (as the light being applied wasn't changing, nor the objects were moving).

The (x,y,z) mapping of points wasn't easy either, and I can't guarantee that it will work the same way with all the Kinects, however, the method used gave very close results from reality, and from the point cloud.

11.5. Baxter Robot control and integration with the peripherals

The "pick and place" task that culminates the usage of all the peripherals integrated with Baxter (like the Kinect, Arduino, AR10 hands, the soft sensors and circuits, and the control computer running Linux) was executed fairly well.

It is very unfortunate that the AR10 hands are advertised to be compatible with Baxter, but the only compatible thing they have is the holes to put the screws. Because of this, controlling the hand must be done separately from controlling Baxter, as the data exchanged with the hand is made through a serial port. Anyhow, the link established between the two, using Python, makes them interact reasonably well.

Arduino never gave any big issues, and it integrated the sensors just fine. In fact, Arduino used can integrate 16 analogue inputs, and can make use of many more sensors, including soft sensors.

Overall, I would recommend the integration of all the sensors, as they were implemented.

As for the controllers, the results achieved make way for many, varied implementations. In this thesis they were shown to work along side the sensors and actuators (peripherals) fairly well, and they were able to complete the prove of concept task (the "pick and place" task) documented previously.

11.6. Possible future improvements

If I were to continue my work with Baxter, I would suggest starting by improving the following points:

- Find better gains for each controller, in order to tune them better for any specific task.
- Improve the tracking capacities of each of the implemented controllers.
- Find a way to decrease Baxter robot noises, and improve the controllers cycle frequencies.
- Add voice commands to trigger Baxter behaviours. This could be a step forward turning Baxter into a more collaborative robot.
- Improve the soft sensors used, and find room for different ones, like capacitive sensors to perceive in close range different materials, like metal objects, and human skin.
- Study the AR10 hands better in order to improve our knowledge of the fingers positions at each instant. Implementation of filters and predictive controllers might give better solutions on this subject.
- Improve Baxter vision of its surroundings, and its object recognition, identification and detection techniques.

12. Web-graphy and Bibliography

References

[1] *http://www.rethinkrobotics.com/smart-collaborative-difference/*

[2] *http://www.rethinkrobotics.com/intera/*

[3] *https://www.active8robots.com/robots/ar10-robotic-hand/*

[4] *http://api.rethinkrobotics.com/baxter_interface/html/index.html*

[5] *http://sdk.rethinkrobotics.com/wiki/Arm_Control_Modes*

[6] *http://sdk.rethinkrobotics.com/wiki/Workstation_Setup*

[7] *http://sdk.rethinkrobotics.com/wiki/Baxter_PyKDL*

[8] *http://sdk.rethinkrobotics.com/wiki/Hardware_Specifications*

[9] **Sven Cremer, Lawrence Mastromoro, and Dan O. Popa** *On the Performance of the Baxter Research Robot* 2016.

[10] **Matthew M. Williamson** *Series Elastic Actuators*

[11] Baxter Hardware Specification Architecture Datasheet

[12] **Yang, Chenguang, Ma, Hongbin, Fu, Mengyin** *Advanced Technologies in Modern Robotic Applications*

[13] **Tsuneo Yoshikawa** *Foundations of Robotics Analysis and Control*

[14] **Mark W. Spong, Seth Hutchinson, and M. Vidyasagar** *Robot Dynamics and Control*