Software Architecture Project

University of Genoa

Robot Arm Teleoperation using Inertial Sensory Data

Report

Students: Marco Demutti, Matteo Dicenzi, Vincenzo Di Pentima, Elena Merlo, Matteo Palmas, Andrea Pitto, Emanuele Riccardo Rosi, Chiara Saporetti, Giulia Scorza Azzara', Luca Tarasi, Simone Voto, Gerald Xhaferaj
Supervisor: Hossein Karami
Year: 2019 - 2020

Contents

Abstract

The aim of this project is to command a robot arm to move from one point to another using inertial sensory data derived from the smartwatch worn by a human.
The goal is to project sensory data to joint commands of robot using inverse kinematics techniques. Since the smartphone provides us with six data, angular velocity and linear acceleration of the moving wrist $(w_x, w_y, w_z, a_x, a_y, a_z)$, and that both human and Baxter arms have seven degrees of freedom, the problem becomes redundant mathematically and a cost function is used to find the best solution of robot joint commands (velocity and position). Some optimisation techniques are hired for finding the most convenient cost function. All the algorithms are tested with simulated robot in CoppeliaSim simulation environment.
GitHub repository: https://github.com/andreabradpitto/SofAr-project

1 Introduction

1.1 Project's goal

The project's goal is to design and implement a software component for the teleoperation of the robot Baxter. The teleoperation works as follows: the human operator moves his arm keeping a smartphone into the hand, and the sensor data from the smartphone's IMU is sent to the software and used to allow Baxter's end effector to follow the trajectory and orientation of the human hand.
The project's original goal, before the Covid-19 emergency, was actually not only to track the end effector's configuration, but also to replicate the motion of the human arm into Baxter's, using Mocap technology; this idea, as well as the objective of using the software on the real robot, was later rejected due to the emergency and the consequent impossibility of accessing the EMARO Lab.

1.2 Hardware and software tools used

The computer on which the software is run should have at least the following characteristics:

- i5 processor, 3.1GHz, 2 cores
- 8GB DDR4 SDRAM
- 256 GB SSD memory
- 103.5 GB dedicated to Ubuntu (in testing 39% memory was in use)

The software was developed and tested on Ubuntu 16.04.
The main software tools used are ROS and CoppeliaSim Edu V4 (which has to be linked with ROS in order for the software to work); the version of ROS used for the development was ROS Kinetic, but the software has been also successfully tested on ROS Melodic on Ubuntu 18.04. Notice that in this report CoppeliaSim is also referred to as V-REP.
Furthermore, an Android app, named CameraImu, has to be installed in order for the IMU data to be sent to the ROS nodes, as explained in the Installation section. The app works best with Android 8.1 or older; earlier versions of the OS may cause frequent freezes and crashes. The app can be installed by downloading and unzipping the apk file found in the git repository. Launching the software on a virtual machine environment causes

instability issues, so it is strongly adviced against.
Some of the ROS nodes were written in the Python programming language, while others (for instance nodes with very intensive matrix computations and the nodes inside the Simulation component) were developed in C++.
Aside from the ROS libraries, the following external software components were also employed:

- *Eigen*, a C++ library for matrix computations.
- *numpy*, a Python library for numerical computations.
- *matplotlib*, a Python library for plotting.
- *pandas*, a Python library for data analysis and manipulation.
- *scipy.io*, a module of the scientific Python library *scipy*, used to interface ROS nodes with mat files.

2 Architecture of the System

The overall system structure has been thought as an ensemble of three main blocks, each one with a specific functionality. We named these three components *Processing*, *IK (Inverse Kinematics)* and *Simulation*. The data flow follows this direction and they are respectively responsible for acquiring data from the human end effector motion, elaborating these data obtaining quantities applicable to the seven Baxter joints and showing how the Baxter motion results through a simulation.

By looking at the UML, in Figure 24, we can easily find these three main components, but we can go a little bit deeper inside the overall system architecture, in order to make the role of each node clearer.
The *Processing* component includes:

- data **acquisition** by IMU sensors embedded in the smartphone, held by the human operator: they deliver angular velocity, linear acceleration measurements and an orientation estimate to the ROS environment;
- data **filtering**: previously computed bias due to the IMU sensors is deleted from the acquired data leading to "calibrated data" (*IMU Calibration* node). Then the effect of gravity on the acquired data is estimated (and subsequently deleted from the linear acceleration vector) by using the orientation estimate and finally a clipping algorithm is applied in order to guarantee that, when the operator arm is held still, the resulting linear acceleration is set to zero (*Gravity Removal & Clipping* node);
- corrected data **publishing**: they are made available for further computations;
- an **orientation calibration** operation: to achieve the superposition of the IMU reference system and the human end effector one (*Orientation Calibration* node);
- **computations of fundamental variables for the Inverse Kinematics** using the corrected smartphone data and joint positions and velocities as feedback inputs (*Forward Kinematics and Data Integration* node);
- **computation of orientation, position and velocity errors** that we want to converge to 0 (*Errors Computation* node).

The *Processing* component outputs are *IK* component inputs together with joint positions as feedback data. In this module the inputs are elaborated obtaining **joint velocities for the Baxter arm joints**, according to the desired motion of the human end effector. Inside the *IK* component there are three sub-blocks and each of them is responsible for the $\dot{q}$ computation in a specific way.

- In the *Analytic IK* service a J matrix is computed by considering six joints of the Baxter arm (instead of seven), it is analytically inverted because of its size (6x6) and later multiplied for the end effector twist in input leading to have the $\dot{q}$ vector.
- In the *Jtransp IK* service the *J transpose algorithm* is used and the resulting $\dot{q}$ are a consequence of the product between J transpose and the error vector.
- In the *CLIK* (*Closed Loop Inverse Kinematics*) sub-component, four different vectors of joint velocities are computed, each one as the sum of three terms. Each term results from one of the three nodes contained in this sub-component: the *Safety task* node computes a partial vector based on the imposition of the joint position and velocity limits; the *CLIK* node computes two different solutions to the tracking problem, using pseudoinverse algorithms contained in the kernel of the previously computed safety term; finally, the *Cost* node applies two different optimizations to the previous terms, yielding four optimized solutions to the inverse kinematics problem.

The resulting $\dot{q}$ vectors of these three different approaches are inputs of the *Weighted Sum* node, where a weighted sum is performed between all the computed solutions, giving as result the final $\dot{q}$ vector. At this point the joint velocities are integrated by an *Integrator* node to obtain the next joint configuration to be tested by the simulator.

The new configuration is therefore sent to the last component *Simulator* which includes:

- the Coppelia environment that just **handles the Baxter graphic representation as a series of configurations** (*Simulation* node);
- a logger which saves the *Integrator* node output data into a specific file with a timestamp, by creating a memory of all the published joint configurations (*Logger* node);

- the *Interface* node that allows the user to handle the simulation by typing commands from the shell. This node also sends information about the simulation state to the other components for the overall system synchronization.

For readability reasons, the names of the nodes in the UML diagram are not always the same as the ones of the real nodes. The mapping for the nodes with discordant names is as follows:

UML node	**Real node**
Imu Calibration	do_calib, apply_calib and computeGravity
Gravity removal and clipping	clipping
Safety tasks	Safety
CLIK	IK
FK/Data Integration	Forward_Kine2
Errors Computation	Errors
Orientation Calibration	calibration2
J Transp IK	J_Transp_server
Analytic IK	jac_mat

3 Description of the System's Architecture

3.1 Module 1: Smartphone

Developers: Marco Demutti, Matteo Dicenzi, Andrea Pitto, Simone Voto

3.1.1 Work coordination

In order to complete each one of the developed tasks, the team has followed a multiple-path approach, which consists of two phases. In the first one, each one developed a solution. In the second, the team analyzed all the solutions and chose the one which led to more suitable results.

3.1.2 Requirements

Hardware requirements:
The module makes use of the **Inertial Measurement Unit (IMU)**, a particular kind of sensor composed by a *three-axis accelerometer* and a *three-axis gyroscope*, by which it is possible to measure angular rates and forces. Here, the IMU also provides the orientation and heading of the smartphone (thanks to a sensor fusion software, embedded in the application required to make the communication between the smartphone and the PC possible).

Software requirements:
The module makes use of the **CameraImu** application (which is available for Android users only).

3.1.3 Nodes

- *Apply calib*
- *Do calib*
- *Compute Gravity*
- *Clipping*

This module takes as input the linear acceleration, the angular velocity, and the orientation of the smartphone. The linear accelerometer sensor provides a three-dimensional vector, representing the acceleration along each device axis. Note that gravity is included in the raw acceleration data. Similarly, the angular velocity is a three-dimensional vector, provided by the gyroscope.

The orientation is a four-dimensional vector represented using unit quaternions. Since the software requires Euler angles [Williams01] to remove the gravity from the linear acceleration, the quaternion will be converted into Euler angles.
Through the CameraImu application, these data are published into the ROS topic *android/imu*, as an IMU message. This message consists of a header (which univocally identifies each message) a unit quaternion (which represents the spatial rotations in three dimensions) along with its orientation covariance, the angular velocity vector and its covariance, the linear acceleration vector and its covariance. These raw-data will be subject to the calibration phase, which is performed through the imu calib package (created by dpkoch `https://github.com/dpkoch/imu_calib`).

3.1.4 Do Calibration

Input:

- *IMU message*

Internal working: This is the first node to be executed. It subscribes to the *android/imu* topic, on which incoming IMU data are published. After the first IMU message is received, the user is asked to put the smartphone in a sequence of six fixed predefined positions. Note that each of these measurements is recorded and it will be used to post-process all the received data. After this initial phase, the node computes the calibration parameters, stores them in the specified YAML file, unsubscribes from the ROS topic *android/imu* and it eventually stops its execution.

Output:

- *Calibration parameters*

3.1.5 Apply Calibration

Input:

- *IMU message*
- *Calibration parameters*

Internal working: This is the second node to be executed (still from the same imu calib package). By subscribing to the *android/imu* topic, it reads the incoming IMU message, applies the accelerometer calibration parameters and then it publishes the post-processed data in the new topic called *android/imu corrected*, again as an IMU message. From now on, and until data is incoming, this node will keep publishing data on such topic.

Output:

- *IMU message*

3.1.6 Compute Gravity

Input:

- *IMU message*

Internal working: This is the third node to be executed. It subscribes to the *android/imu corrected* topic, listens to the first fifty incoming IMU messages and computes an approximation of the gravity vector. Finally, it stores the gravity vector into a csv file and stops its execution. It is very important that the smartphone is held fixed, with the z-axis directed along the gravity vector for the whole process.
The importance of this procedure regards the fact that IMU sensors, due to their physical structure, introduce errors when measuring the linear acceleration. So, if we set a fixed gravity value, we would obtain larger errors when performing the gravity suppression.

Output:

- *Gravity estimation*

3.1.7 Clipping

Input:

- *IMU message*

Internal working: Clipping is the last node to be executed in this module, and it is responsible for the clipping and the gravity suppression. It subscribes to the *android/imu corrected* topic, in order to properly receive new IMU messages. This node performs two tasks: at the beginning, it just listens to the first fifty linear accelerations data. Then, it starts reducing noise from such data. In the first phase, the smartphone must be held fixed by the operator. Whenever a new IMU message arrives, the node performs the gravity suppression and saves the result for post processing. After a sufficient number of received IMU messages, the node uses its knowledge in order to compute a threshold, which will be used in the second phase to reduce the noise in the linear acceleration suppression. This noisy phenomenon is introduced by the human operator, which tries to stay fixed, but ends up introducing oscillations. Thanks to the threshold, in the second phase each linear acceleration is filtered (clipping), and this allows us to have exactly zero linear acceleration values when the smartphone is held fixed. Then, the resultant linear acceleration, the orientation and the angular velocity are used to make the new IMU message, which is then published to the *smartphone* topic, to which the node of the next module will subscribe.

Output:

- *IMU message*

3.2 Module 2 - Part 1: Data processing

Developers: Vincenzo Di Pentima, Emanuele Riccardo Rosi, Chiara Saporetti, Giulia Scorza Azzara', Luca Tarasi, Gerald Xhaferaj

3.2.1 Baxter's model

A quick overview on Baxter's model. Reference: [Williams01]

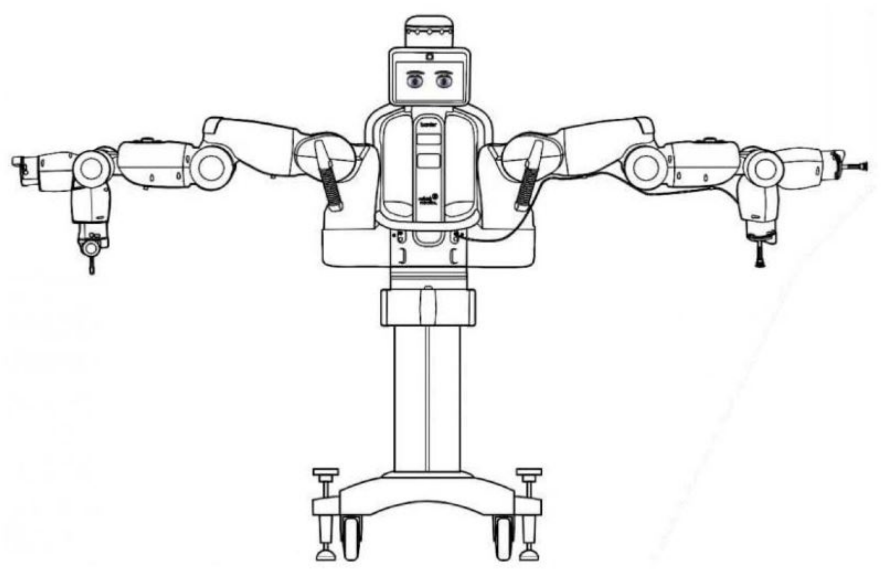


Figure 1: zero pose

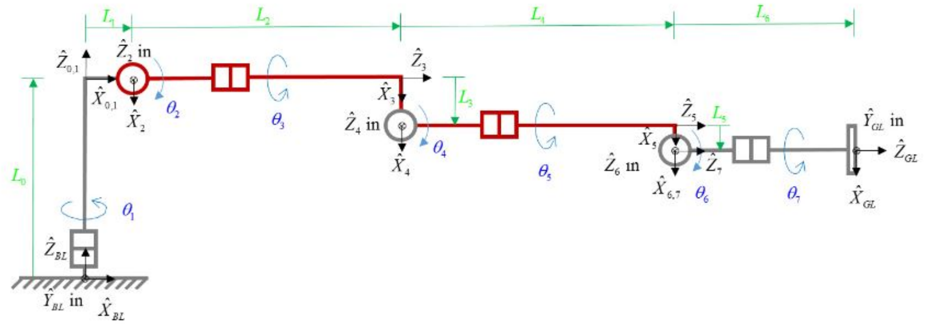


Figure 2: seven-dof left arm kinematics diagram with coordinate frames

From Figure 2, it is possible to derive the Denavit-Hartenberg (DH) parameters, which will be used to compute the transformation matrices of the arm and then the geometric Jacobian.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	L_0	θ_1
2	$-\frac{\pi}{2}$	L_1	0	$\theta_2 + \frac{\pi}{2}$
3	$\frac{\pi}{2}$	0	L_2	θ_3
4	$-\frac{\pi}{2}$	L_3	0	θ_4
5	$\frac{\pi}{2}$	0	L_4	θ_5
6	$-\frac{\pi}{2}$	L_5	0	θ_6
7	$\frac{\pi}{2}$	0	0	θ_7
8	0	0	L_6	0

In the table the L_i, where $i \in [0, 6]$, represent the length of the linkages of the arm.
From now on the $BL\ frame$ in Figure 2 will be referred as $zero\ frame$.

3.2.2 Orientation Calibration

A view on the involved frames during the overall experiment:

- $zero\ frame$, which is positioned on the base of Baxter's arm;
- *end effector frame*, which is positioned on the end effector of Baxter's arm;
- $IMU\ frame$, which is positioned on the smartphone as in Figure 3;
- $Global\ frame$, w.r.t. which the data is measured.

Before feeding the reference linear acceleration and angular velocity to the inverse kinematics modules, the signals must be projected on a frame of our interest, e.g. $zero\ frame$.
In the original problem statement finding the misalignment between *IMU frame and end effector frame* is trivial. In our case, instead, the problem becomes more difficult since we have only one sensor available.

Input:

- $control\ signal$, which allows the calibration algorithm to start;
- $quaternion$, which maps the rotation from $IMU\ frame$ to $Global\ frame$;

Internal working: given the initial orientation matrix of *end effector* w.r.t. *zero frame* of Baxter and knowing how the frame is placed on the phone's IMU (Figure 3), it is possible to compute the orientation matrix from *zero frame* to *Global frame*, which will be important to compute the orientation matrix from *end effector frame* to *IMU frame*.
The operator is asked to orient the smartphone in a manner such that the orientation matrix from *end effector frame* to *IMU frame* is an identity matrix. Of course since the operator cannot be infinitely precise some error is introduced. After setting up, the operator sends the control signal via the *Interface* node and the algorithm, using basic orientation matrix algebra, computes the output. Check the third instruction in subsection 4.4.2 for the detailed procedure that the operator has to do.

Output: orientation matrix from *zero frame* to *Global frame*.

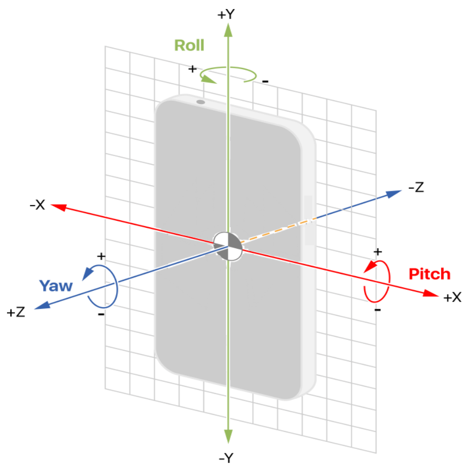


Figure 3: smartphone reference frame

3.2.3 Forward Kinematics and Data Integration

Input:

- *control signal*, which allows to start reading smartphone's data;
- *IMU message*;
- q, current configuration of Baxter's arm;
- $\dot{q}$, current joint velocities of Baxter's arm;
- *orientation matrix* from *zero frame* to *Global frame*.

Internal working: given the initial conditions the module computes the initial orientation and position of *end effector* w.r.t. *zero.* After the initial setting, it waits for the control signal that allows to start reading *IMU messages.*
The first *IMU message* is very important since it allows to compute the orientation of *IMU frame* w.r.t. *end effector frame* using the input *orientation matrix.*
The *linear acceleration* and *angular velocity* are first projected on *zero frame.*
The *linear acceleration* is then integrated, allowing to compute the target *linear velocity* and the target *position*, assuming to use as initial conditions, the initial position of Baxter's *end effector* and its initial velocity.
The current configuration q is used at each step to compute the current geometry of Baxter's arm and then derive the *Jacobian.*
The current joint velocities $\dot{q}$ are used to compute the current velocity of Baxter's arm.
To sum up, at each step the module computes the current geometry and kinematics of Baxter's arm, integrates the data coming from smartphone and sends out all the data needed for inverse kinematics.

Output:

- *Jacobian matrix*, used to compute inverse kinematics;
- *reference signals*, which include:
    1. reference *linear acceleration*;
    2. reference *linear velocity*;
    3. reference *angular velocity*;

- *data vector*, used to compute the errors.

  It includes:

  1. reference *linear velocity*;
  2. reference *position*;
  3. reference orientation matrix of *end effector frame* w.r.t. *zero frame*;
  4. current *linear velocity* of Baxter's arm;
  5. current *position* of Baxter's arm;
  6. current orientation matrix of *end effector frame* w.r.t. *zero frame*.

3.2.4 Errors Computation

Input: *data vector*, as defined in the output of *FK/Data Integration* node.

Internal working: computes the positional error, the velocity error and the misalignment from current orientation to target orientation.

Output: *errors vector*, which contains the quantities mentioned in internal working.

3.3 Module 2 - Part 2: Inverse Kinematics

3.3.1 Safety, Inverse Kinematics and Optimization

This module of the project leads to the solution of the Inverse Kinematics problem based on pseudoinverse methods; the solution is made of three consecutive steps: *safety tasks*, *inverse kinematics task* and *cost function* for the optimization. The basic idea to solve this problem is to give different priorities to the safety constraints (which get the highest priority) and to the tracking task, while the last node is just used to optimize the solutions coming from the others. In particular, the safety tasks include joint angle and joint velocity limits, since the robot cannot perform certain motions and cannot move at certain speeds. The algorithm's main structure is based on [Control01], [Control02].
The final $\dot{q}$ resulting from the whole algorithm is the sum of the contributions of the three terms specified before. Each block is implemented as a service and is a client to the other services. Below the input/output variables and descriptions of each block are listed.

3.3.1.1 Safety

Input:

- q feedback from integrator block
- previous $\dot{q}$ from weighted sum block

Internal working:
Even if the main goal of the inverse kinematics problem is the tracking of the human arm, the safety task has the maximum priority, since it's more important that the robot doesn't exceed limits in angle and velocity rather than it follows the desired motion.
The behavior we would like to have is that when the robot is approaching a limit, a safety task should be activated in order to avoid undesired motions. On the contrary, when the robot is far from this situation, the tracking task is the only active one, while the safety one is disabled.
In the inverse kinematics problem, we have the relation linking the joint velocities $\dot{q}$ and the end effector velocity $\dot{x}$, by means of the Jacobian matrix J ($\dot{x} = J\dot{q}$). An important aspect to be clarified here is what we have defined as J matrix and $\dot{x}$ for the safety task. In the end, after all the computations (omitted for sake of simplicity), we obtain $J = I$ (identity matrix) and $\dot{x} = -K(q - q_{safe}), K > 0$.

When the robot's arm is in a safe position, we would like the safety task to be off. Hence we tried to minimize $||A(\bar{x} - J\dot{q})||^2$, where A is a diagonal 7x7 matrix containing sigmoidal values. As a consequence, if the current q_i is in a safe position, the corresponding sigmoidal value a_i is equal to zero (safety task is off), while if q_i is approaching a joint limit angle, the corresponding sigmoidal value a_i will progressively approach 1 (safety task is on, with the maximum priority).
Regarding the speed limits, an approximation has been used, with an error of order $O(DT)$, where DT is the step of the simulation and the formula is $\dot{q} = -DT \times K(\dot{q} - \dot{q}_{safe}) + \dot{q}_{prev}, K > 0$, where $\dot{q}_{safe}$ is a safe joint velocity.
For sake of simplicity the two safety tasks are merged into a single task, since when adjusting the joint position, the corresponding joint velocity will approach zero and so both the tasks are controlled.
The joint limits values have been obtained from the Baxter model in CoppeliaSim, while the speed limit of each joint has been set to 1 radps (absolute value), which is considered to be adequate given the possible range of speeds of a human arm.

Output:

- partial $\dot{q}$ to inverse kinematics block

3.3.1.2 Inverse Kinematics

Input:

- partial $\dot{q}$ from safety block (obtained by service call)
- previous $\dot{q}$ from weighted sum block
- J matrix from forward kinematics block
- $errors\ vector$ from errors computation block
- $reference\ signals$ from forward kinematics block

Internal working:
At this stage we have $\dot{q} = \dot{q}_{safety}$, which corresponds to $\dot{q} = 0$ when the robot is in a safe position. To implement the inverse kinematics algorithm, we have computed at each simulation step the partial $\dot{q}$, using the following formulas, from [Control01] and [Control02]:

$$\boldsymbol{\rho}_0 = \mathbf{0}, \quad \boldsymbol{Q}_0 = \boldsymbol{I},$$

$k = 1, \ldots, p$, where p is the total number of priority levels:

$$\begin{aligned}
\boldsymbol{W}_k &= \boldsymbol{J}_k \boldsymbol{Q}_{k-1} (\boldsymbol{J}_k \boldsymbol{Q}_{k-1})^{\#, \boldsymbol{A}_k, \boldsymbol{Q}_{k-1}} \\
\boldsymbol{Q}_k &= \boldsymbol{Q}_{k-1} (\boldsymbol{I} - (\boldsymbol{J}_k \boldsymbol{Q}_{k-1})^{\#, \boldsymbol{A}_k, \boldsymbol{I}} \boldsymbol{J}_k \boldsymbol{Q}_{k-1}) \\
\boldsymbol{\rho}_k &= \boldsymbol{\rho}_{k-1} + \left(\boldsymbol{Q}_{k-1} (\boldsymbol{J}_k \boldsymbol{Q}_{k-1})^{\#, \boldsymbol{A}_k, \boldsymbol{I}} \boldsymbol{W}_k (\dot{\bar{\boldsymbol{x}}}_k - \boldsymbol{J}_k \boldsymbol{\rho}_{k-1}) \right),
\end{aligned}$$

Then two different methods arise because there are two ways to compute $\dot{x}_e$, which is the twist of the end effector. The first method (IK_1) uses the linear velocity of the target, resulting from the integration of the linear acceleration in the Forward Kinematics block: $\dot{x}_e = \dot{x}_g + kerr$.
The second one (IK_2) uses the linear acceleration and an algorithm of the second order, based on the following formula from [Control03] (where J_A represents the tracking Jacobian):

$$\ddot{\boldsymbol{q}} = \boldsymbol{J}_A^{\#}(\ddot{\boldsymbol{x}}_d - \dot{\boldsymbol{J}}_A\dot{\boldsymbol{q}} + \boldsymbol{K}_D\dot{\boldsymbol{e}} + \boldsymbol{K}_P\boldsymbol{e})$$

One problem is that we don't get the angular velocity (that would be used in $\ddot{x}$ term) from the sensors, so we cannot use the above formula directly. We can solve this problem by using a hybrid method, which involves ω_g for the angular component and a_g for the linear one. Then we could approximate the $\ddot{q}$ with a finite difference, ending up with the following formula:

$$v_e = DT(a - \dot{J}_L\dot{q} + K_v\dot{\eta} + K_p\eta) + J_L\dot{q}$$

where a is the reference acceleration, η is the linear error, K_v and K_p are positive gains.
Again, by introducing this approximation, we'll have an error of order $O(DT)$ for the IK_2 method, which is the same order as the one in IK_1 algorithm. Therefore we expect the two algorithms to work with similar precision.

Output:

- $\dot{q}$ to cost function block

3.3.1.3 Optimization

Input:

- $\dot{q}$ from inverse kinematics block (obtained by service call)
- q feedback from integrator block

Internal working: The $\dot{q}$ coming from the safety and the tracking tasks have to be optimized. Since we used two methods to do that, we finally end up with four solutions: the tracking $\dot{q}$ from the IK_1 method optimized in two different ways and the one from IK_2, again optimized in two different ways. One optimization choice has been to minimize the $\dot{q}$, exploiting the formula found in [Control01]. The other optimization choice has been to get close to a preferred configuration $\dot{q}$ (initial pose).

Output:

- optimized $\dot{q}$ to weighted sum block

3.3.2 Analytic 6dof Jacobian matrix for IK

Input:

- q current configuration of Baxter's arm
- $errors\ vector$ from errors computation
- $reference\ signals$ from forward kinematics

Internal working: This algorithm uses the Jacobian matrix defined in its analytic form, with the assumption that one joint of the arm is fixed. In this way we obtain a 6x6 matrix instead of a 6x7 one.
By construction Baxter has a seven-dof arm composed of a two-dof shoulder, two-dof elbow and three-dof wrist (with an additional kinematic offset). It is known that, according to Pieper's principle, for a robot arm where three consecutive coordinate frames share the same origin, the inverse kinematics can be solved in closed form. Since this doesn't happen in Baxter, a possible simplification for the IK problem could be to lock the first joint of the elbow and to consider a null offset in the wrist.
Locking the third joint of the manipulator (namely setting $\theta_3 = 0$) has two main advantages, the first being that, since θ_2 and θ_4 rotate about axes that remain always parallel, we have simpler kinematics equations that use their sum. Secondly, the IK problem is based on a mxn jacobian with m=n=6, so it is simpler to solve with respect to the kinematically redundant case where m=6 and n=7.
For the calculation of the Jacobian matrix we use the reduced form (from seven to six elements) of the joint positions given in input, and DH parameters through which we find the transformation matrices, and from those extract the geometric vectors. The pseudoinverse of the Jacobian is then obtained via

- singular value decomposition of J: $J = U\Sigma V^{\mathrm{T}}$
- multiplication for a bell shaped function in order to smooth near singularity behaviour $\Sigma = diag\{\frac{\sigma}{\sigma^2+p^2}\}$
- calculation of the pseudoinverse of J $J = V\Sigma U^{\mathrm{T}}$

The final desired velocity of the joints is computed as: $\dot{q} = J^{\#}(\dot{x}+KErrors)$, where K is the control gain and $\dot{x}$ is the reference twist. Finally, the vector of joint velocities is sent as a service by setting the third element to zero.

Output:

- $\dot{q}$ joints velocities

3.3.3 Jacobian Transpose IK

To perform the Inverse Kinematics of the Baxter arm, this module implements the Jacobian Transpose technique.

Inputs:

- *Baxter Jacobian matrix*, from Forward Kinematics;
- *Error vector*, from errors computation.

Additional user-defined Parameters:
In addition to the external inputs, the module also requires the user to set some parameters:

- the **sampling time** (Δt), used in the control and sensors' measures, in our case corresponds to a $\Delta t = 0.01$ s, since the system needs to be accurate and fast in reproducing the user's movements;
- the **saturation**, described by the variables MIN and MAX, is necessary to avoid sending harmful or unfeasible velocities to the robot. In our case they are set respectively to -1 and 1, because they were considered to be adequate given the possible range of speeds of a human arm;
- the **weights** (W_p), strictly related to the algorithm and with the same dimension of the error vector.

Internal Working:
Since both inputs change continuously according to the arm's movements, they are communicated to other modules by publishers. For this reason, the Jacobian Transpose module contains two subscribers to the topics '*Jacobian*' and '*errors*'. All the necessary computations for the IK are performed by the handler of a server that answers to the *weighter*'s periodical requests.

It's important to point out that the module also needs data availability flags to operate correctly. Whenever new data arrives, in the callback functions the flags are set to *true* and, after they are used or not them all are available at the request time, they are set again to $false$. This prevents to introduce an error in the motion whenever data is not updated: instead of returning a wrong answer, being it referred outdated data, the module only communicates the problem.

The Algorithm:
After the complete explanation of all the necessary parameters, the algorithm itself, performed in the module by a dedicated function (jtransp), needs to be clarified.
It approaches the Inverse Kinematics problem by applying the Lyapunov direct Method, finding a function relating joint velocities and the error able to guarantee the error to go asymptotically to zero. Choosing the following V(e) as Lyapunov function as [Siciliano02]:

$$V(e) = \frac{1}{2}\, e^T K\, e \Rightarrow \dot{V} = \; e^T K\, \dot{x}_d - \; e^T K\, \dot{x}_e = e^T K\, \dot{x}_d - \; e^T K\, J(q)\, \dot{q}$$

Choosing $\dot{q} = J(q)^T K e \Rightarrow \dot{V} = \; e^T K\, \dot{x}_d - \; e^T K\, J(q)\, J(q)^T K e$

With the assumption of J being full rank and K to be a positive definite matrix, the conditions $V > 0$ and $\dot{V} < 0$ are perfectly satisfied in the case of $\dot{x}_d = 0$, implying, for Lyapunov, the asymptotical convergence to $e = 0$. But this is not the case of the Baxter's arm: tracking implies that the desired end effector's velocity changes during time. In the best case, if the movements of the operator are much slower than the control, it could be considered as "quasi-static" scenario.
If this is not the case, the Jacobian Transpose solution will be able anyway to keep the error bounded, dependently on K's norm.

The choice of $\dot{q}$ can be also explained by the attempt of solving the Inverse Kinematics using the **Gradient Descent** (or Gradient Method for nonlinear systems): this becomes clear by considering the minimization of the following cost function F.

$$F = \frac{1}{2}\, (x_d - x_e)^T\, (x_d - x_e) \rightarrow \Delta q = -\alpha\, \left((x_d - x_e)^T\, \frac{\delta x(q)}{\delta q} \right)^T$$
$$\Delta q = \alpha J(q)^T\, (x_d - x_e) = \alpha\, J(q)^T e$$

The common choice for the gain matrix K is a diagonal matrix and there are two main possibilities for defining it:

1. Using a matrix with fixed values
2. Using elements dependent on the error

The usage of fixed values is simpler, but the latter, used in [Buss], showed better performances. In fact, in this way is possible to have the most fitting value for each situation: when the error is big, the gain increases, in order to recover rapidly, instead, if the error is small, the gain decreases, to avoid wide oscillations. Fixed values instead require to be chosen according to a trade-off between oscillations and slow dynamics.

It is also possible to use a common value for all the elements, but in this way it's not possible to assign greater or smaller importance to different components of the error vector: for this reason this approach has not been considered. In the module, the gains, depending on the related error norm (translational or rotational), have different weights, defined by the user in W_p).

Output:

- *Joint Velocities*, able to make the error converge to zero.

3.3.4 Weighter

Input:

- *control signal*, which allows the node to start publishing.
- The responses from the Cost, Jacobian Transpose IK and Analytical IK services, i.e. the six computed joint velocities $\dot{q}$.

Internal working: selects the best solution among the ones received from the services specified in the **Input** section. The algorithm works as follows:

1. If CLIK first order solution optimized for joint speed is available, choose it.
2. Otherwise, if the analytical solution is available, choose it.
3. Otherwise, if the Jacobian transpose solution is available, choose it.
4. Otherwise, output the same velocity as in the previous simulation step.

Steps 2 to 4 handle the case in which one or more data is missing.
Indeed the original idea for this node, that is to produce a weighted sum of the various inverse kinematics solutions, works only if weight 1 is given to one of the solutions and 0 is given to the others: this is because the six inverse kinematics solutions are computed with different techniques, so in general a weighted sum of them does not constitute a solution for the inverse kinematics problem. Therefore the best algorithm among six was choosen (as illustrated in the Testing section) to be always used if available.
The chosen solution is then saturated to satisfy the speed limit of each joint

of 1 radps (absolute value), which is considered to be adequate given the possible range of speeds of a human arm.
The operation of this node has to be started, and may be paused or stopped, by the input control signal. In the case of a stop signal, the initial condition are reset (that is, $\dot{q}$ is set to zero).

Output: $final\ \dot{q}$, which is the joint velocities vector that will be actually employed in the current step of the simulation.

3.3.5 Integrator

Input:

- *control signal*, which allows the node to start publishing.
- $\dot{q}$ vector from Weighter node.

Internal working: the node integrates $\dot{q}$ using the trapezoidal rule (except for the first step, in which Euler integration is used, since at that step only one data sample is available, while the trapezoidal rule needs two).
Afterwards, the node saturates the configuration vector to the joint limits, to ensure that it represents a feasible configuration for Baxter.
The operation of this node has to be started, and may be paused or stopped, by the input control signal. In the case of a stop signal, the initial condition are reset (that is, q is set to zero).

Output: q, the joint configuration vector obtained from the integration.

3.4 Module 3: Simulation

Developers: Elena Merlo and Matteo Palmas

All the produced code was made on Ubuntu 16.04, with the CoppeliaSim Edu V4 and ROS kinematics, which is the advised configuration. So they are required in order to make everything work properly.

Input: q, data encoding the configurations for all the seven Baxter arm joints.

Internal working: this node is able to receive some data encoding the configurations for all the seven Baxter arm joints: by reaching specific sequential configurations with a time interval depending on the chosen frequency we are able to simulate the motion of this robot arm. The input data for this component are also saved and stored inside a document, a txt file together with a timestamp for knowing when each requested configuration has been sent to the simulator.

Output: none.

The UML in Figure 4 shows how this module is organized in case of the so-called "test configuration", i.e. the module setup when it is considered independently from the other blocks. As we can appreciate we needed to work inside the ROS environment developing *Publisher_ROS_VREP* and *Logger* nodes, respectively responsible for sending and storing the dummy inputs to the core component belonging to the V-REP environment (*Simulation*) where we operated, too.

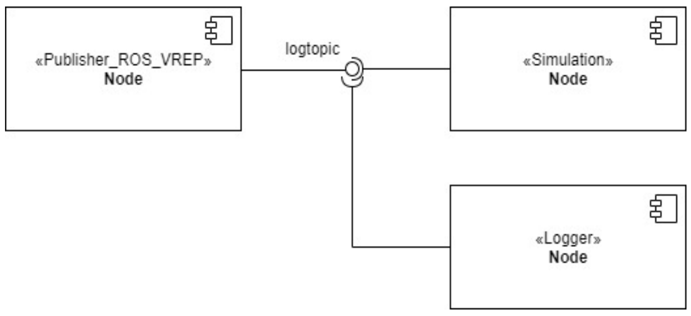


Figure 4: UML of V-REP module in its "test configuration"

3.4.1 The ROS environment

The ROS environment development was concentrated in writing a publisher and a subscriber to handle communications with the simulation, mimic the interaction with the other components within the project and create a history saving data passed to the Simulator during the whole testing period.

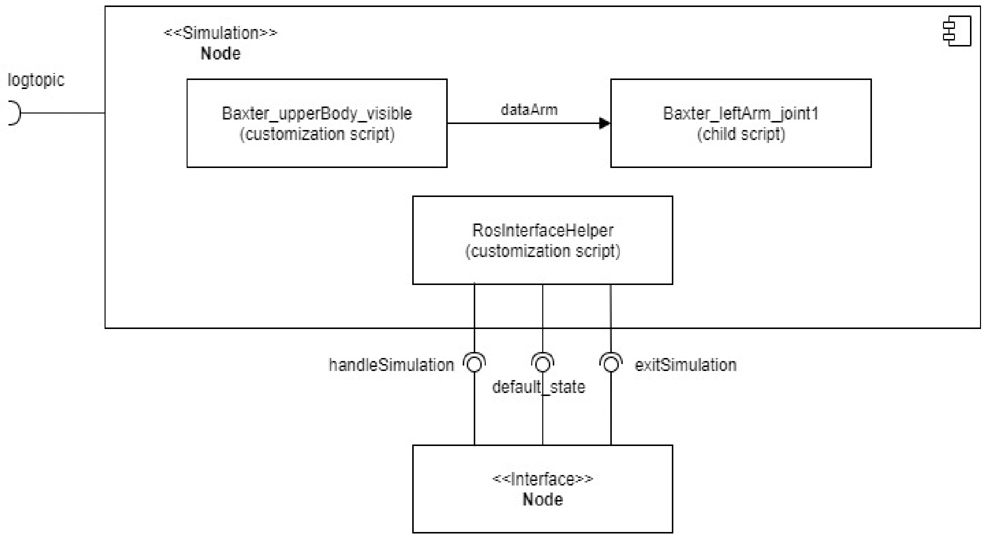
The publisher component is a dummy one that only occurs in the "test configuration" case by generating some values mimicking each joint position value. When all the modules are attached together (the so-called "simulation configuration"), then these input values are made available and sent to the V-REP environment by the Inverse Kinematics block.
This script has the form of a ROS publisher, that sends to a specific topic (*logtopic*) a *JointState* message (that belongs to sensor_msgs class). This

kind of message provides many fields to fill, with the possibility to leave the useless ones empty.
This component has a publication rate set to 100 Hz according to the project specifications.
Inside this script we can find some code portions we can decide to comment according to which kind of position values we want to be published. The first possibility is to send at every cycle the very same value for each joint, the second one is to send randomic values, our choice to study the simulation behaviour during a testing phase without any other module connected; these values become more consistent once all the modules are merged together to perform a live simulation.

The position message published on *logtopic* is read by the *Simulation* node that allows the robot arm to reach the desired position encoded by the subscribed data.
Simultaneously the very same topic is subscribed by the *Logger* node, which saves the read data into a specific file (logger.txt, located in a specific folder *baxter_scene* to be easily found) with a timestamp, generating a history of the module itself by creating a memory of all the published messages. This script is a ROS subscriber that reads a *JointState* message from *logtopic.*

3.4.2 The V-REP environment

In Figure 5 we show deeper inside the Simulation component and how the data are handled by this node is explained, according to the V-REP internal code (lua scripts).



<<Simulation>>
Node
logtopic
Baxter_upperBody_visible (customization script)
dataArm
Baxter_leftArm_joint1 (child script)
RosInterfaceHelper (customization script)
handleSimulation
default_state
exitSimulation
<<Interface>>
Node

Figure 5: UML of the Simulation node (inner view)

In the V-REP environment, the original scene we started from worked as a synchronous script, used to inform each arm joint the hardwired pose it should be at and informing the main script when the objective was obtained. This type of setup was not convenient for our objectives, so we opted for a different approach:

- **one asynchronous customization script** is used in order to handle the input from ROS environment and send the position command to the correct arm;
- **the left arm threaded child script** is modified in order to read the message by the customization script and let the desired position being reached by each arm joint.

After this broad description of the various components of this module, we can proceed with the presentation of the workflow inside it.
Starting from the “test configuration”: the dummy publisher generates an array of seven floats and sends them as a *JointState* message to the V-REP environment where the asynchronous customization script, when the simulation is started, sends it to the arm as a position signal to be imposed to each left arm joint (*dataArm*).
The live-simulation configuration acts the same as the already described modality, except for the dummy publisher that is replaced by the *IK* component, as seen in Figure 6.

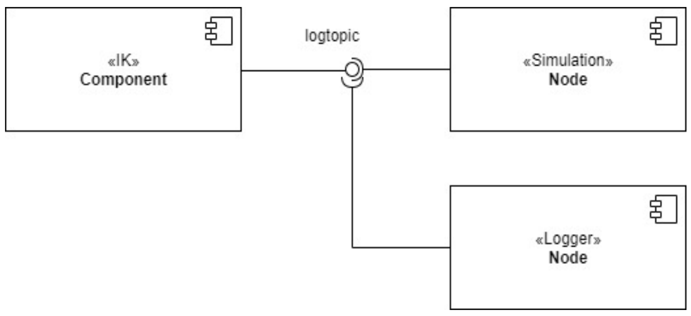


Figure 6: UML of V-REP module in its "simulation configuration"

3.4.3 User Interface

As it is evident from Figure 5, the Simulation node has another possibility to face the ROS environment, and that is thanks to an interface that allows the user to handle the simulation by typing commands from the shell. The relation between ROS and V-REP is again a publisher-subscriber one: there are 3 different topics linking the two sides (*handleSimulation*, $default\_state$, *exitSimulation*) one for each task the interface can perform.

There are five main commands:

1. **Start**: to initiate the simulation in CoppeliaSim without the user actually interacting with the simulator.
2. **Pause**: to momentarily stop the simulation which will restart from the last reached configuration after a start command.
3. **Stop**: to take the simulation back to time 0.
4. **Set_default**: to easily set the starting configuration of the simulation robot so that it could match the human.
5. **Exit**: to orderly kill all the running ROS nodes and close the CoppeliaSim environment.

Some ulterior notes can be made:

- There is a **help** command to remind the usage of the various commands.
- The **Set_default** should not be done after a **Pause** as it creates discrepancies.

- There is also a **Calibration** command useful which triggers the Orientation Calibration node (for more details see 3.2.2). By adding this new command we can use only one UI handling the whole system, conditioning the total synchronization.

After having exposed all the channels the two environments ROS and V-REP use to communicate one another, in Figure 7 we present a rqt_graph showing that linkage and how the flow of information works (look at the arrows tip) during the "test configuration" phase. Inside each square we can find an involved topic: three of them are for handling the simulation interface and the fourth one is responsible for sending the new data for the arm motion from the ROS side.
Please note that sim_ros_Interface is what we named *Simulation* node, while the other names in circles are the scripts working as explained above. As we already mentioned, the *publisher_ROS_VREP* component is replaced by the *IK* block during a live simulation.

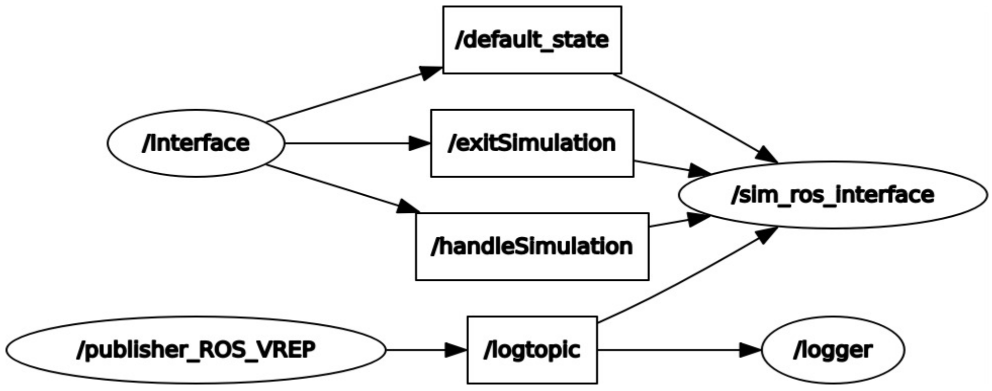


Figure 7: rqt_graph of V-REP module in its "test configuration"

3.4.4 Launch files

In order to run the whole module we need a launch file able to start a master and all the interested nodes. In particular two versions of this file were created, the first is *baxter_test.launch*, which starts the module with the dummy publisher already discussed, and the second is *baxter_sim.launch* which is the one which will perform in the live version of the project.
To avoid the user intervention in resolving resource conflicts by modifying each script with its specific addresses for every installation, we decided to make this module plug and play. To do this two bash scripts are developed which will be called by the launch file itself:

1. *coppelialauncher.sh* allows to find the path of the CoppeliaSim package and that of our scene (*baxter_sofar.ttt*) inside the user resources and let the simulator open and run the correct environment;

2. *logger_launcher.sh* with the same code let the Subscriber run and the *logger.txt* file to be found inside a specific folder (*baxter_scene*).

4 Installation

4.1 Preparing your computer

4.1.1 Installing the simulation environment

Download Coppeliasim Edu V4: go to https://www.coppeliarobotics.com/downloads, click on "Choose a different platform", choose Ubuntu 16.04 and click "Download CoppeliaSim Edu". Go do Downloads and unzip the folder in the same location as the catkin workspace folder.

4.1.2 Installing the libraries

Install the following libraries: numpy, scipy, matplotlib, pandas (you can use pip list in terminal, beforehand, to check if these are already installed):

1. Type **python -m pip install numpy** in terminal to install numpy. Check SciPy.org for more information.
2. Type **python -m pip install -U matplotlib** in terminal to install matplotlib. Check matplotlib.org for further information.
3. Type **sudo apt-get install python-pandas** in terminal to install pandas. Check pandas.pydata.org for more information.
4. Type **sudo apt-get install python-scipy** in terminal to install scipy. Check scipy.org for more information.
5. Type **sudo apt-get install ros-kinetic-cmake-modules** in terminal to install the cmake modules.

4.2 Installing the package

Hereafter is described how to properly install our package. Download the project from github and put it into your catkin workspace. In order to do so, open a terminal and type:
cd catkin_ws/src
where catkin_ws shall be understood as the absolute path to your catkin workspace. Then type:
git clone https://github.com/andreabradpitto/SofAr-project
Then, to install the package, type:
cd SofAr-project
./install.sh

The install file performs a catkin_make multiple times and builds all the pyhon files.

4.3 Preparing your smartphone

Install the CameraImu app on your smartphone: go to
SofAr-project/Smartphone/smartphone and unzip
org.ros.android.android_tutorial_camera_imu_1.0.apk
in order to install the CameraImu app on your smartphone. Warning: the app works best with Android 8.1 or older; earlier OS versions may cause frequent freezes or crashes.

4.4 Running the package

There are two possibilities in order to test our work :

1. Simulation without the smartphone by using a predefined trajectory to follow.

2. Simulation with the smartphone.

4.4.1 Simulation without the smartphone

It allows to test the project by sending Baxter a set of pre-determined input data for the trajectory. The data files were created through MATLAB simulations and the input trajectory we use as an example here is a half circle of the end effector on the horizontal plane. In order to try this simulation open a terminal and type:
cd catkin_ws/src/SofAr-project
Then type:
./launcher_test.sh
The launcher_test file activates all the ros nodes, topics and services needed by the simulation. It also launches the CoppeliaSim environment with the correct scene.

Use the user interface to handle the simulation (see chapter 3.3.3 to know more about this user interface): on the shell a message invites the user to write a command on it. To start the half circle simulation write on the interface **start** and enjoy Baxter move! The simulation can be repeated by writing **stop** and again **start**. Whenever you wish to exit the simulation write **exit**.

4.4.2 Simulation with the smartphone

In order to try this simulation open a terminal and type:
cd catkin_ws/src/SofAr-project

Start the CameraImu app in your smartphone. Then you have to type the IP address of your computer into the app's Master URI textbox (i.e. http://YOUR_IP _ADDRESS:11311, as the default port is 11311). In order to find your IP address, type hostname -I in terminal if you are using a Unix system, or type ipconfig in command prompt if using Windows.
When you have finished, click "connect" and place your mobile on a horizontal surface.

Then, on the second terminal, type:
./launcher.sh
and follow the calibration instructions that are prompted on the terminal in this order:

1. First:
   Place the phone on an even horizontal surface.
   Press enter to proceed.

2. Second:
   If you want to launch the sensor calibration process type "yes" and press enter.
   If you press yes you will read on the terminal:
   Orient IMU with X+ axis up and press Enter

   And another terminal is opened, on which you can read the linear acceleration registered by your mobile. Take your smartphone and orient it with one of its sides parallel to the ground until you see that the linear acceleration on X is around -10. Repeat the calibration for all of the axes by following the same concept. Note: on the terminal of the acceleration it is also possible to check if the app is still running. In case it stops close it and open it again.

3. Third:
   With your hand, hold the phone in the calibration position and maintain the position.
   Press enter to proceed.

   Take the smartphone in your right hand and raise your arm so that it is perfectly horizontal with respect to the ground and at an angle of more or less 45 degree with respect to the plane going parallel to your chest. The smartphone should be horizontal, have the display oriented towards you and the external camera on the left side, near your thumb.

Keep the arm raised in the position. When Coppeliasim user interface appears there are a few commands you can run. In order to perform the simulation write **calibration** and then **start**. Watch Baxter as it tries to follow the movement of your arm!

5 System Testing and Results

5.1 System Testing

In order to test the application as a whole, it is sufficient to follow the instructions in Section 4.4.2; to test the math and simulation packages only, follow the instructions in Section 4.4.1.

5.2 Results

5.2.1 Introduction to the results

Because of the unresolved issues encountered in producing a fully working system, it is useful to look at the results two types of tests: tests that employ all the modules of the system and tests that employ only the math and simulation modules.

5.2.2 Results for tests of the full system

The simplest test performed with the whole system is the test where the operator stays still. In this case the simulation works since the acceleration, shown in Figure 8, is clipped to a zero value.

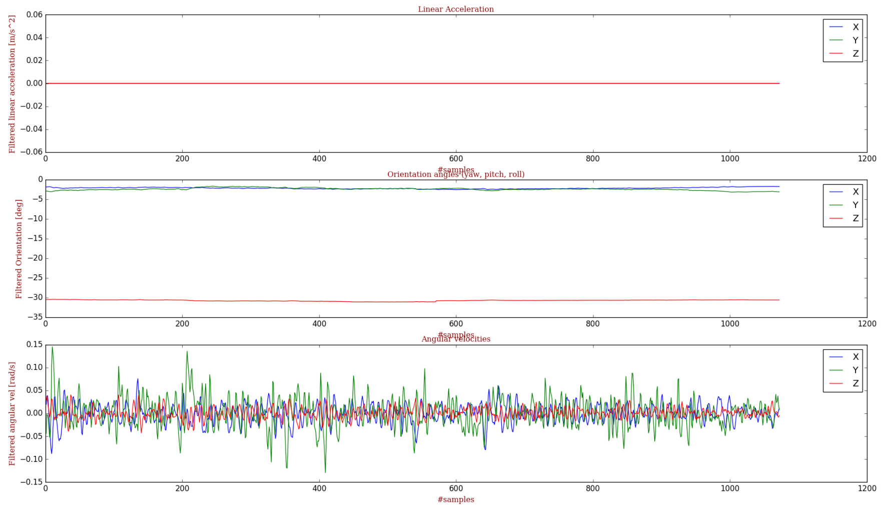


Figure 8: Plot of smartphone data for still reference trajectory.

In the tests with moving reference trajectories the error is not guaranteed to converge to zero.
Due to the non-observability characteristic of the system, as we don't have any measurements of Cartesian position, an estimation system could not be provided.
Another affecting factor is the poor sensor quality in general, as well as the inability to perform a state-of-the-art calibration (we do not own the necessary tools). This leads to have limited control over the precision of the signal, that appears noisy, with non-zero mean and generally unreliable, especially during transition phases of the operator's gestures.
For instance, we consider the test where the shoulder of the human (joint 1 of Baxter) rotates from its original position to its limit in 2 seconds (this test has been named "halfcircle test"). The resulting smartphone data is shown in Figure 9.

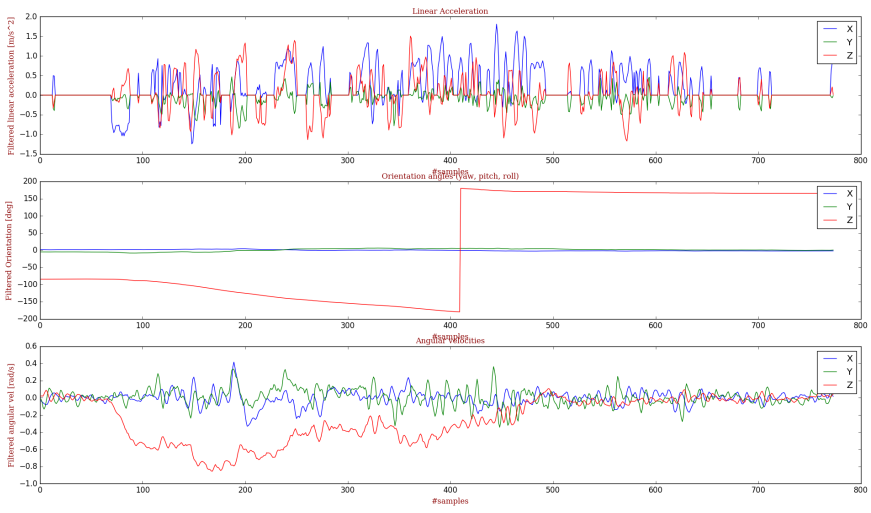


Figure 9: Plot of smartphone data for halfcircle test.

As it can be seen, the acceleration data is very perturbed and, indeed, it was verified that the error does not converge.

5.2.3 Results for tests of math and simulation modules

These tests have been performed extensively during the production of the code, with the following objectives:

- Test the functionality of the math and simulation modules, using each inverse kinematics algorithm separately.
- Compare the algorithms for inverse kinematics and decide which one presented the least number of cons and the largest number of pros (this is the so-called **offline** phase).

The tests were performed by using mock reference signals generated by a MATLAB simulation. For each test, the following fake data was provided:

1. The acceleration required by the reference trajectory, generated by differentiating it twice.
2. Quaternions, obtained by running a MATLAB simulation imposing only the position tracking and then extracting the rotation matrices generated in the MATLAB simulation.
3. Angular velocity, approximated during the MATLAB simulation by using the inverse strap-down equation.

First we consider the halfcircle test.
The figures from 10 to 15 show, respectively, the linear and rotation errors for the second pseudoinverse algorithm with joint optimization (called CLIK2 from here on), the transpose algorithm and the analytic algorithm.

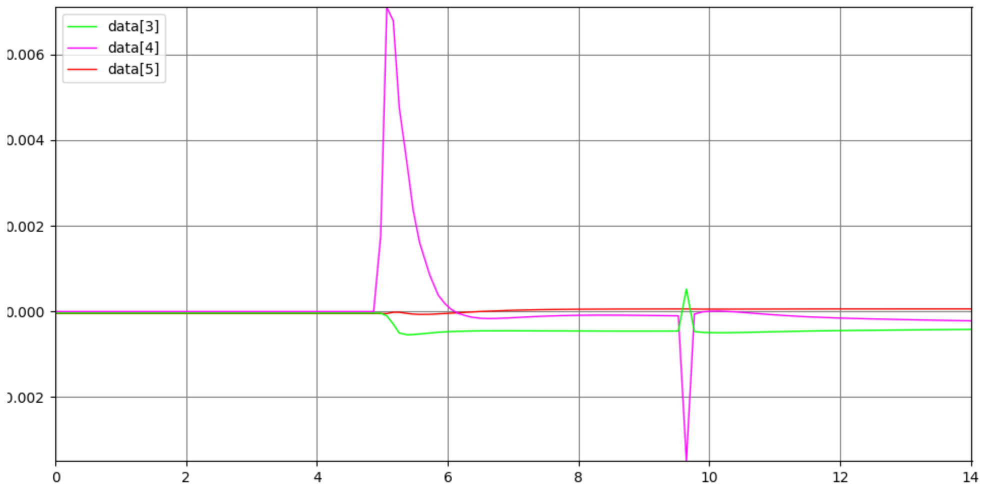


Figure 10: Positional error for CLIK2 on halfcircle test

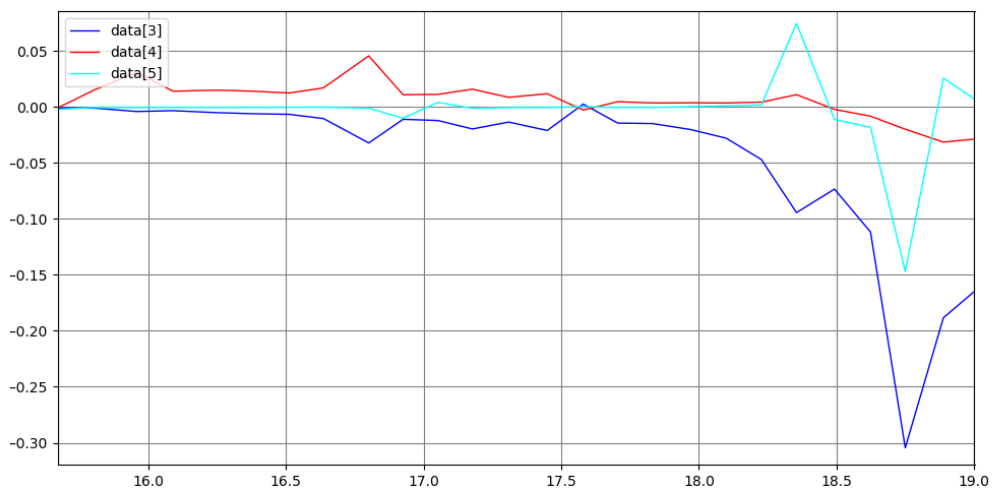


Figure 11: Positional error for transpose algorithm on halfcircle test

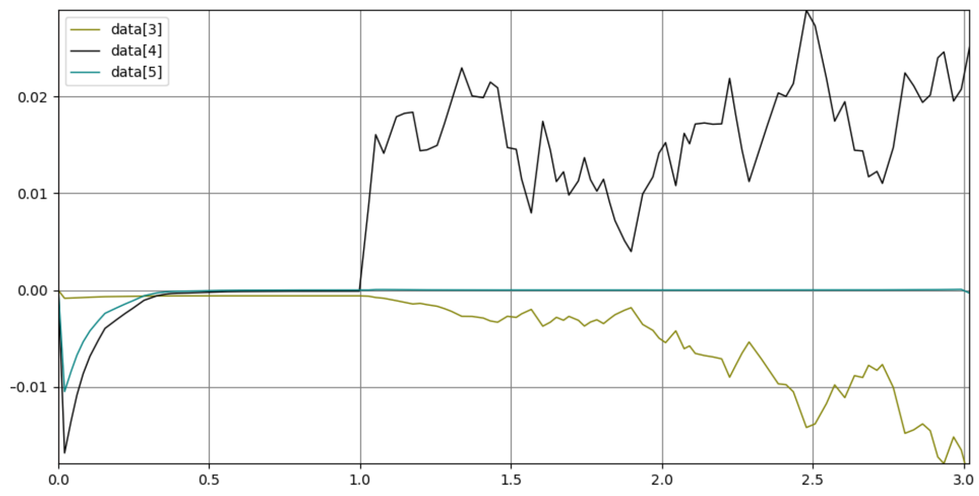


Figure 12: Positional error for analytic algorithm on halfcircle test

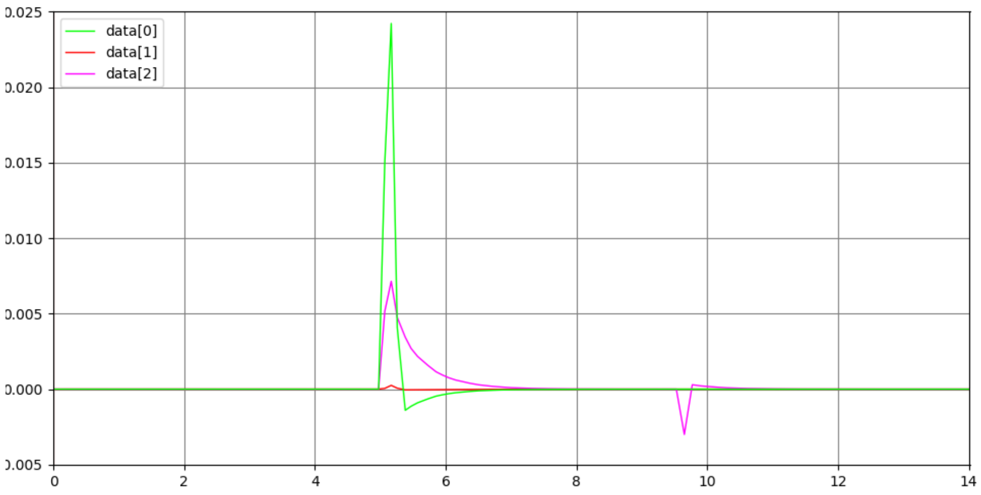


Figure 13: Angular error for CLIK2 on halfcircle test

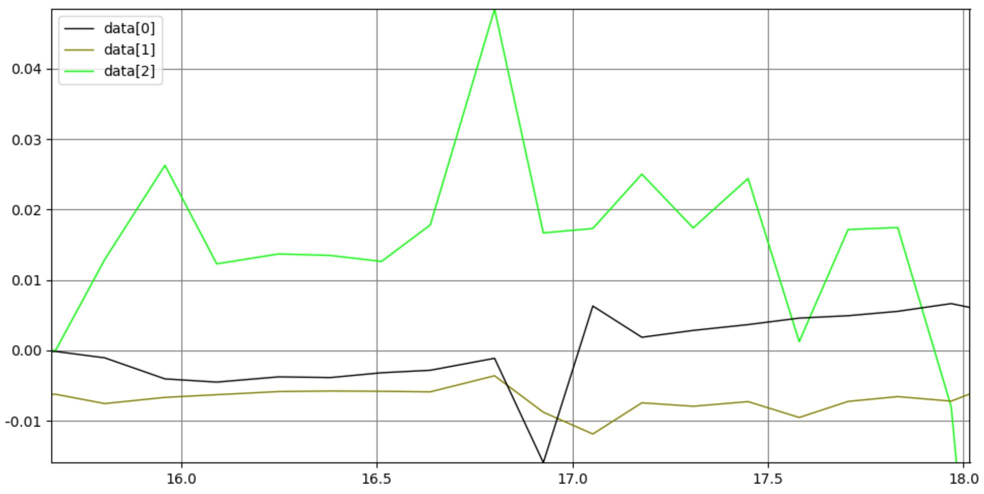


Figure 14: Angular error for transpose algorithm on halfcircle test

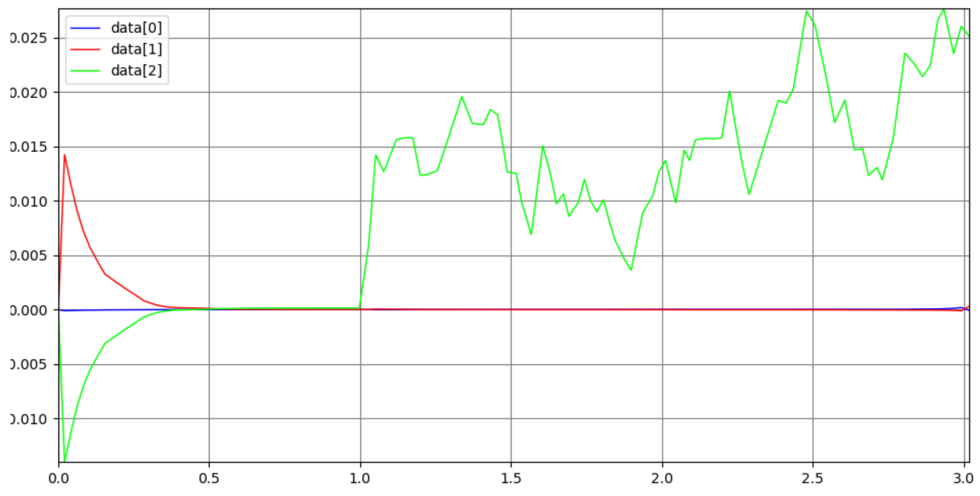


Figure 15: Angular error for analytic algorithm on halfcircle test

As it can be seen, with this trajectory all three algorithms perform well (in particular, CLIK2 is the one that presents the least number of oscillations). The first peak of the errors is due to the reference trajectory, which is artificial and presents an angular point in the instant the joint starts rotating. Because of the joint velocity limits, the robot arm cannot replicate this unnatural behaviour and performs a smoother trajectory; however, the error soon converges to 0 again.
In figure 16, reference trajectory and robot trajectory are both shown, and it can be seen that they appear as overlapped most of the time.

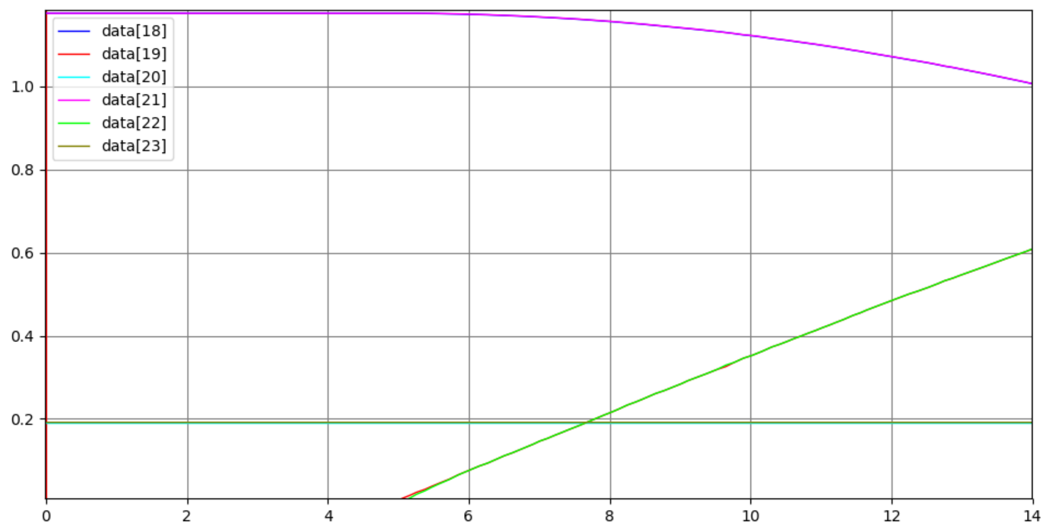


Figure 16: Reference signal (data 18 to 20) and robot position (data 21 to 23) for CLIK2 on halfcircle test.

We can consider a more difficult test, where the reference trajectory has the arm retracting on itself along the x direction. In this test, the mock trajectory is not very precise because of the approximations in the calculations, so such test is useful to check the robustness of the algorithms with respect to far from perfect reference signals. The results are in the figures 17 to 22.

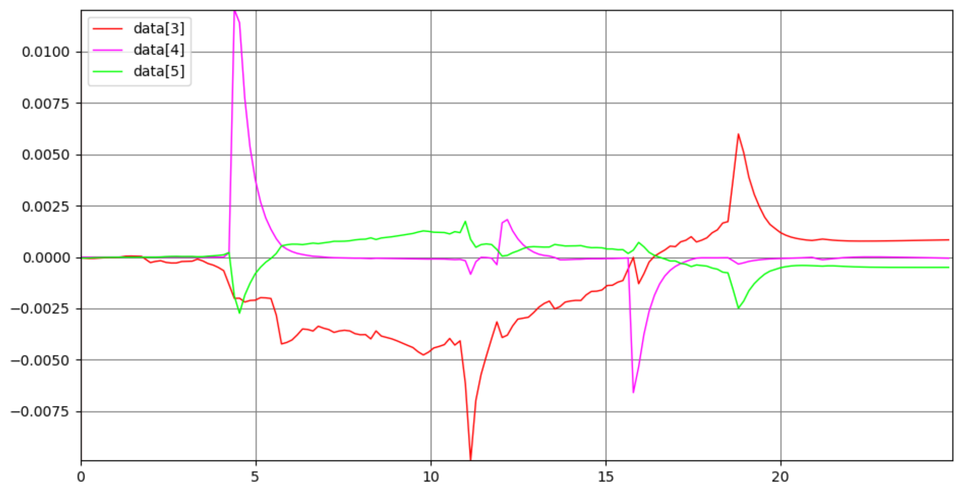


Figure 17: Positional error for CLIK2 on the second test

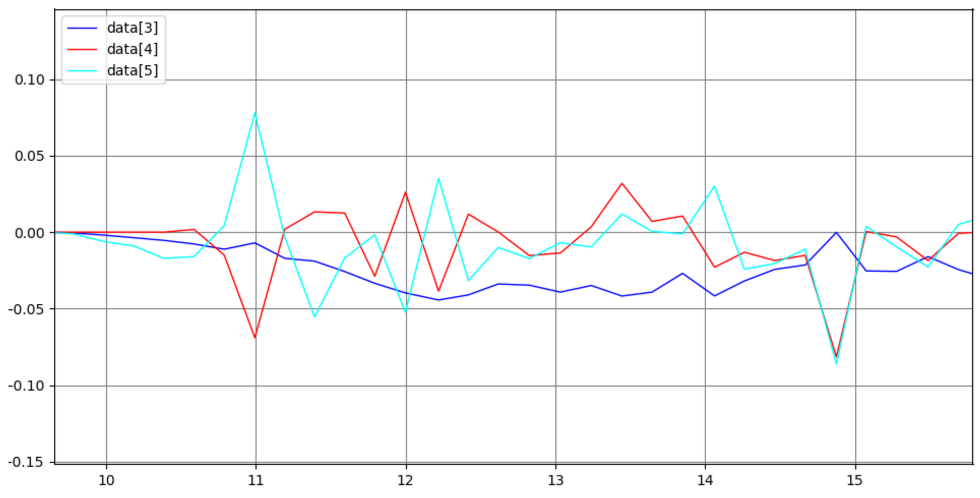


Figure 18: Positional error for transpose algorithm on the second test

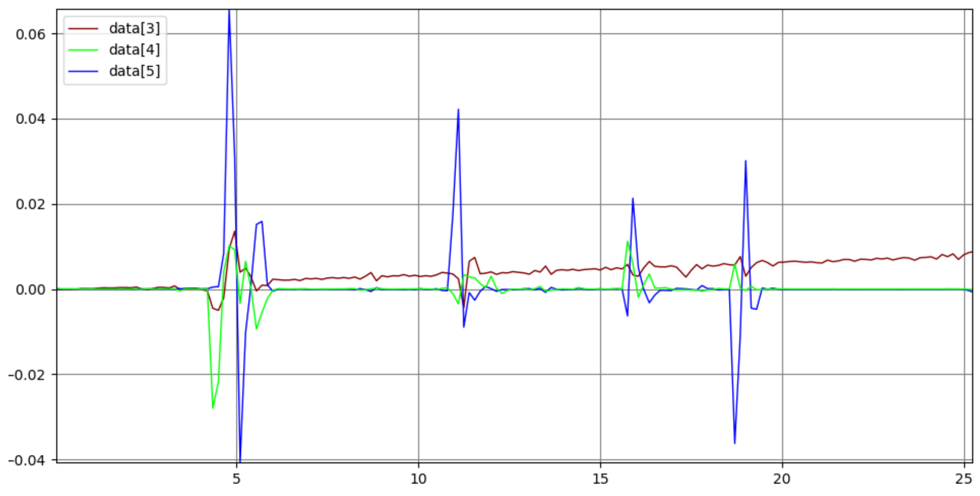


Figure 19: Positional error for analytic algorithm on the second test

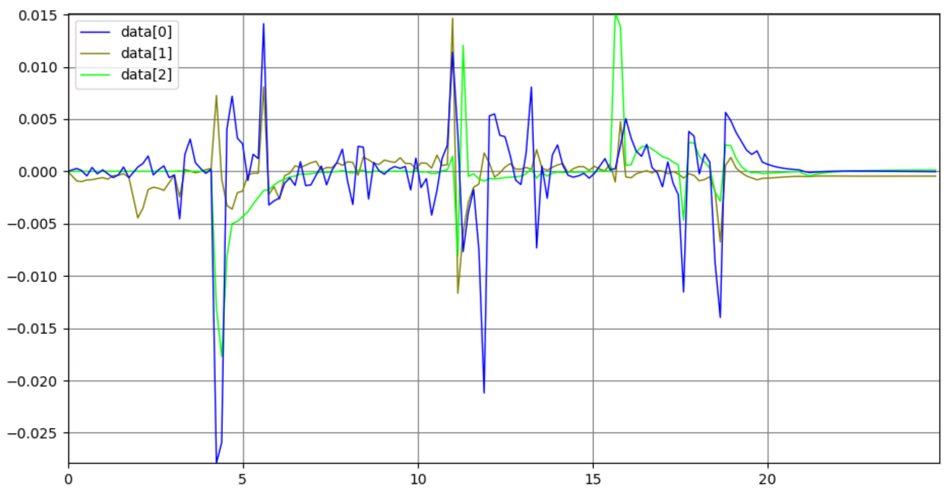


Figure 20: Angular error for CLIK2 on the second test

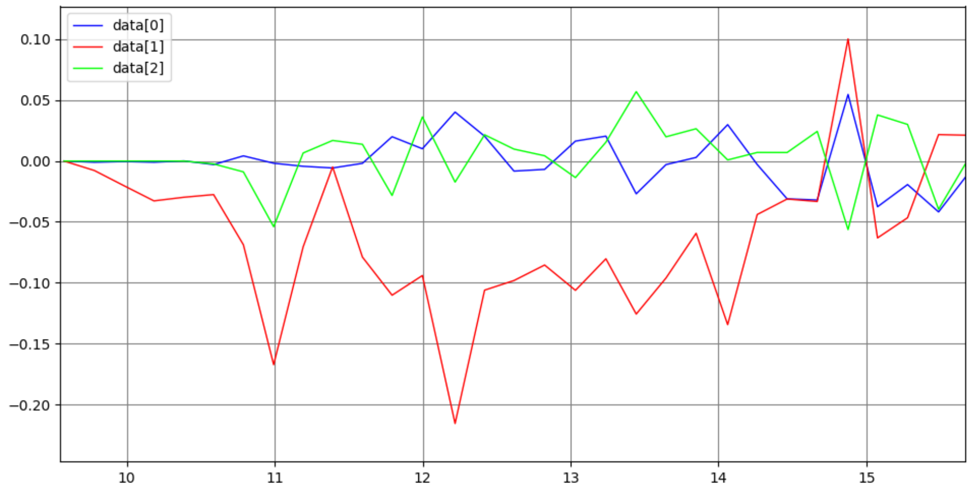


Figure 21: Angular error for transpose algorithm on the second test

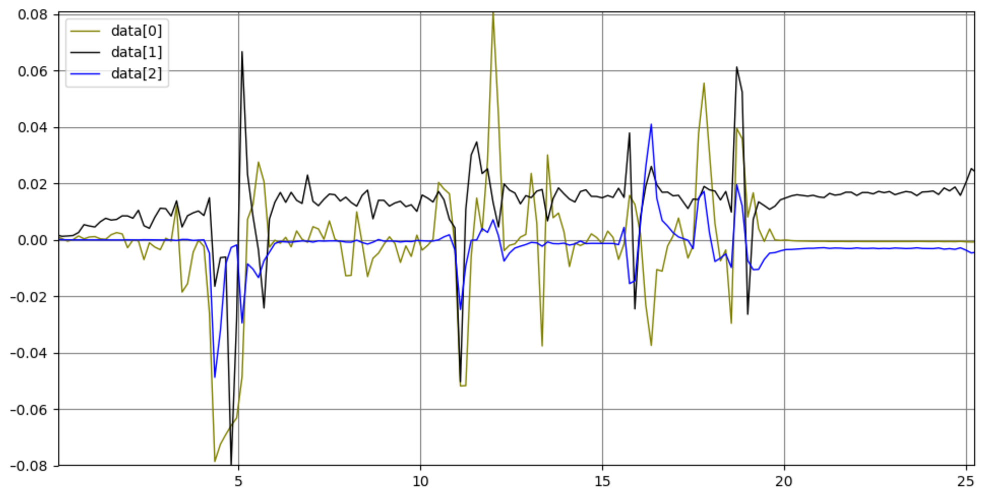


Figure 22: Angular error for analytic algorithm on the second test

As it can be seen, the pseudoinverse algorithm is the one that keeps the error closest to zero; for example, if we consider the position error, we see that in such algorithm the error is typically one degree of magnitude smaller than the one given by the other ones.
Running other simulations with mock data yields similar results.
Among the pseudoinverse algorithms, the most stable one appeared to be the first order one optimized for joint velocities; the second order one with the same optimization function has a similar performance, but the first order one was preferred because it presents less error oscillation (however, note that such oscillations are usually in the order of millimiters).
On the other hand, the pseudoinverse algorithm optimized with the favorite configuration technique presents worse matrix conditioning, and is therefore less stable, so it was quickly judged as not ideal.
Because of all the considerations explained until now, in the end the best algorithm, that is the algorithm to be employed in the **online** phase, was judged to be the first order pseudoinverse algorithm with joint velocity optimization.

6 Recommendations

6.1 Module 1: Smartphone

This module is the one actually responsible for the physical interface with the user, carried out by using the smartphone throughout the whole experiment. This necessarily implied the need for special attention and assumptions both during the development of the code and the testing phase, and also results in a series of recommendations that can improve the functioning of the whole module:

- The calibration phase has to be performed with particular care: an improper placing of the smartphone in different positions always leads to inaccurate calibration parameters, which introduce an error at the very first step of the complete data flow.
- The motion of the human arm is naturally characterised by undesired vibrations, which result in noisy sensor data. Again, an appropriate support would definitely improve the situation.

Besides these, many limitations arose as a direct consequence of the forced change of the development environment and available tools during the pandemic:

- The smartphone is here replacing a smartwatch, which would of course have been easier and more practical to handle during the experiment.
- Since the development of the project had to be carried out remotely, the code was thought as totally independent of any specific smartphone. Moreover, different smartphones showed different behaviors (sometimes even incompatibility) when using the same code and performing the same motion patterns. This can be due both to hardware (e.g. sensors) and software (e.g. Android O.S. version) differences. Therefore, all the involved parameters had to be found dynamically, whereas the use of a single device may have led to a more precise and optimised tuning.
- A whole module had to be removed from the project: the Motion Capture could not be performed due to the inability of physically going to the laboratory. This resulted in the absence of complete data about the configuration of the human arm of the user, hence the necessity of compromises in the smartphone module (e.g. clipping in the gravity removal), as well as in the other ones. The presence of this module would have played a key role in the complete success of the experiment. Thus, a future development in this direction is strongly advised.

6.2 Module 2: Mathematics and Inverse Kinematics

6.2.1 Jacobian Transpose IK

This module presents many limitations due to the algorithm's own nature, more adequate to simpler manipulators and tasks.

First of all, even considering the best case for it ($\dot{x}_d$ =0), the configurations chosen to reach to the desired reference are unpredictable: they do not respect a particular criterium and they can appear as counter-intuitive, not being the shortest path or moving also unnecessary joints. But, being in the context of a tracking task, this aspect is not a problem, since the path is showed point by point by the user and the algorithm does not need to formulate it, unlike it happens for far targets.

Something that doesn't fit the situation is the redundancy of the robot: it implies the existence of a kernel of the J^T and, if $Ke \in Ker(J^T)$, the algorithm will remain struck at $\dot{q} = 0$, with $e \neq 0$.
Another issue concerns the singularities: the full rank assumption cannot always hold and the algorithm has no way to deal with them. Moreover, it's not possible to accomplish additional tasks or perform solution's optimization, as it would be working on the kernel space defined by the pseudoinverse: among them, for example, joint limits. If they are ignored, results can make the robot move in a unfeasible way, breaking it or blocking it (limits applied) in a situation in which the Inverse Kinematics solution is not able to converge.

Possible Improvements

After explaining all the limitations of this algorithm, it is not difficult to find ways to improve it. To make the algorithm more reactive and adequate to the tracking task, [Ali] has been taken into account. It suggests using additional terms like the velocity error ($K_d\ \dot{e}$) and another one ($h(t)$) able to realize an approximation of the Feedback Linearization. The latter was not fitting the case of a redundant robot, while the first was usable: as the following test show, this addition was not able to improve the results as expected.

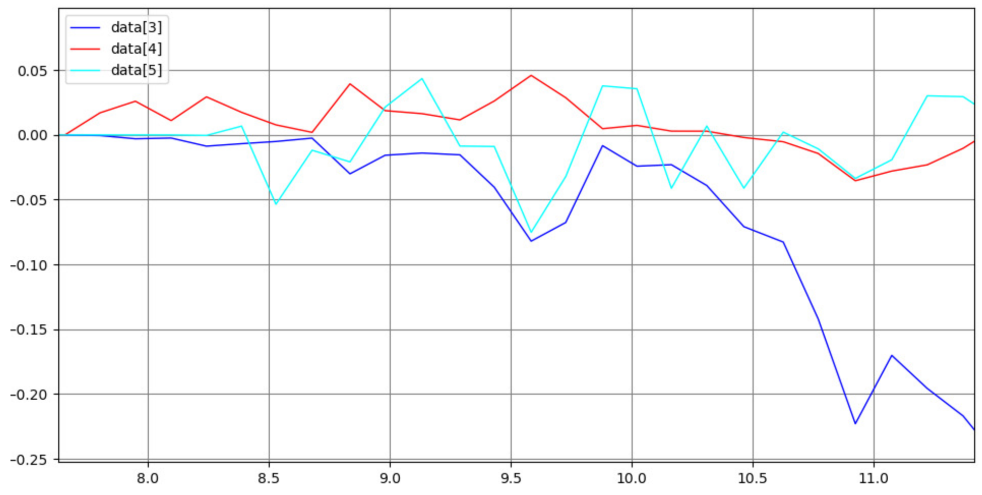


Figure 23: Position Error for JT modified version

The position error (e_x, e_y, e_z) in Figure 23 displays bigger peaks and more oscillations w.r.t the original case (Figure 18), even using very small gain. This can be due to a velocity sensitivity that reveals to be harmful, instead of improving the results where needed. For this reason, this approach has been discarded.

Since it is not able to avoid singularities, another possible improvement could be, as described in [Husky-Lenarcic], to add a correction term that makes the manipulator able to escape from them. But given the kinematics structure of the Baxter's arm, it's not so simple to explicitly derive them, so a technique like the previous paper's one cannot be easy implemented on it, differently from other seven dofs arm manipulators.

These two possible improvements have been tried by implementing them as python scripts in the GitHub folder '*Jacobian_Transpose_Enhanced*' for possible future applications related to this algorithm.

The last possible way to further develop this algorithm consists in making it aware of the Joint limits. In this way it would be able to overcome the issues related to some trajectories, as the test section showed (Figure 18).

6.2.2 CLIK algorithms

Some enhancements could be used for the four CLIK algorithms to make them more effective, such as:

- Adaptive poles: in the current implementation, the poles for the safety tasks are fixed. Such poles define the rate at which the safety task is performed. A better way to assign them could be to use some run-time calculation and assign them dynamically. For example, the joint limit correction formula is $\dot{q} = -K(q - q_{safe}), K > 0.$, where K is the absolute value of the pole. To decide K, we could use the inequalities

$$|\dot{q}_i| < \dot{q}_{i,max}, i \in [1, 7]$$

to obtain

$$K|q_i - q_{i,safe}| < \dot{q}_{i,max},$$

i.e. the maximum gain that does not break the speed limit constraint is given by

$$K_{max} = \frac{\dot{q}_{i,max}}{|q_i - q_{i,safe}|}.$$

In practice, it would be wise to use a smaller gain, say $K(1 - \varepsilon)$, $\varepsilon > 0$ to be decided (either by more analytical reasoning or by testing).
A similar reasoning could be used for the speed limits, too.

- Adding the obstacle avoidance: in a future implementation it could be useful to implement obstacle avoidance as an high priority task, the obstacles being Baxter's body.

6.3 Module 3: Simulation

Before achieving the final structure of the *Simulator* component we developed another version with some differences: in particular the *Simulation* node received data directly from the *IK* component without any integration, so V-REP input was a joint velocity vector ($\dot{q}$) instead of a joint position one (q). This choice allowed us to have more coherent data to send to the simulation environment, since they were not further manipulated by an integration. Then V-REP scripts were such that the simulation was performed by applying input joints velocity to the seven Baxter arm joints by using a proper function. Then the resulting pose was saved and sent to the logger node which stored in the text file all the joint configurations Baxter reached during the test period. So the Simulation node was both a subscriber of joints velocity and a publisher of joints position.
We needed to change this original structure because we were not able to respect the simulation frequency of 100 Hz defined by the project specifications, once all the modules were merged together.
By adding the *Integrator* node the information passed to the V-REP environment is a bit compromised since the dynamic model of the motors is not

considered by the integrator, so the mechanism inside the simulator changes because the Baxter arm moves in order to reach each specific input configuration without considering any velocity the joints servos need to produce.
However this is not a big deal, since the joint velocities generated from the inverse kinematics module satisfy positional and speed constraints, so they constitute a feasible velocity vector, and its integration gives a reachable configuration. Moreover we have lighter lua scripts, that means speedy computations and graphics updates such that the simulation rate is really close to the desired one.

In any case the solution of inserting an *Integrator* node could be not the only possible one, for example it could be reasonable to find a way to make the overall simulation less heavy by performing more code optimisation, therefore avoiding the need of an integration. Another possibility would be replacing the *Integrator* node with a component performing the dynamic model in order to send as V-REP input a joints velocity vector which takes into account also the torques limits.

In this section we would also like to show some proposals for a future improvement of the *Simulation* node development.

In particular in the whole system we decided to focus on the motion of only one Baxter arm, the left one, but we can go a step further and make both the arms move. To do that we can simply add a new child script in the V-REP environment, attach it to the right arm branch and use the very same commands shown for the left side.
Another improvement would be exploiting the humanoid item inside V-REP to study which would be the joints configuration due to specific velocity inputs. That could be a good way to understand how a human really behaves if certain positions are imposed to be reached to his arm and thanks to these results we can try to estimate more accurate inputs.

References

[Kok] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. 2017. Using Inertial Sensors for Position and Orientation Estimation. Found. Trends Signal. Process. 20−21.

[Williams01] R.L. Williams II, 'Baxter Humanoid Robot Kinematics', Internet Publication, https://www.ohio.edu/mechanical-faculty/williams/html/pdf/BaxterKinematics.pdf, April 2017

[Control01] Simetti E., Casalino G. A Novel Practical Technique to Integrate Inequality Control Objectives and Task Transitions in Priority Based Control. *J Intell Robot Syst* 84, 877–902 (2016)

[Control02] Simetti E., Casalino G., Wanderlingh F., Aicardi M., Task priority control of underwater intervention systems: Theory and applications. *Ocean Engineering* 164, 40–54 (2018)

[Control03] F. Caccavale, S. Chiaverini and B. Siciliano, Second-order kinematics control of robot manipulators with Jacobian damped least-squares inverse: theory and experiments, in *IEEE/ASME Transactions on Mechatronics*, vol. 2, no. 3, pp. 188-194, Sept. 1997.

[Siciliano02] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo, 'Robotics: Modelling, Planning, Control', pp. 135-137, 2009. Springer

[Buss] S.R. Buss, 'Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods', Internet Publication, https://www.math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf, October 2009

[Ali] S. Ali, A. Moosavian, E. Papadopoulos, 'Modified transpose Jacobian control of robotic systems', Internet Publication, https://www.researchgate.net/publication/237507230_Brief_paper_Modified_transpose_Jacobian_control_of_robotic_systems, July 2007

[Husky-Lenarcic] J. Lenarcic, M. Husky, 'Advances in Inverse Kinematics: Analysis and Control', 1998, pp. 478-482. Springer Science+Business Media

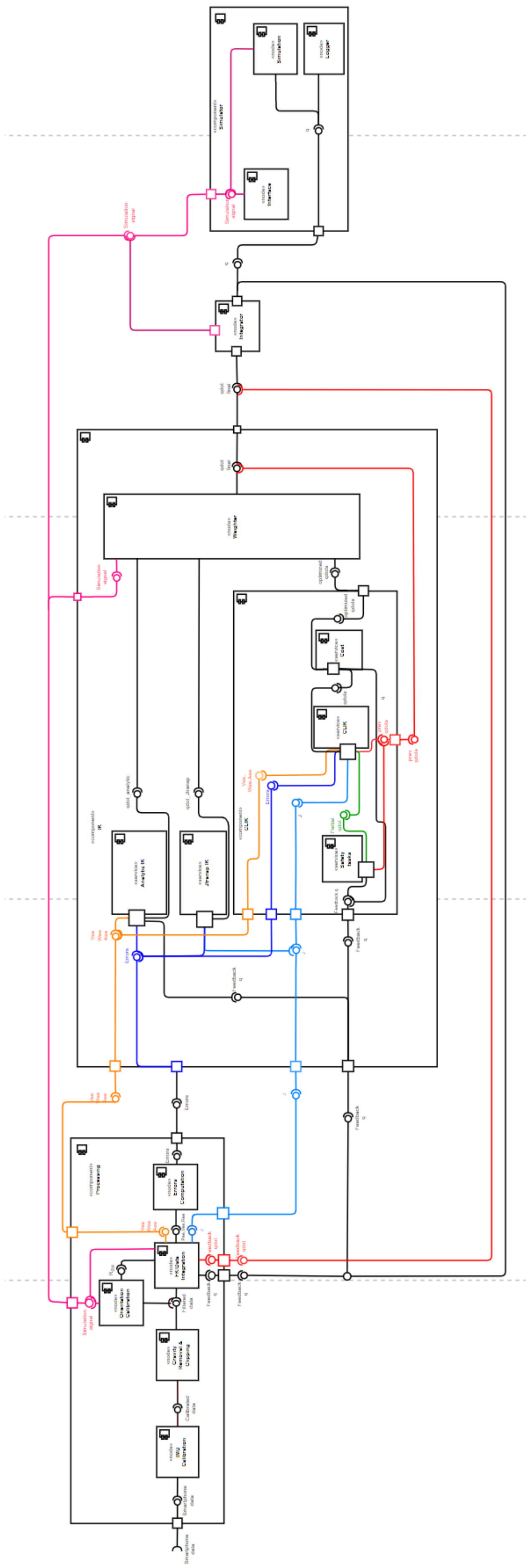


Figure 24: UML