Master thesis
Master in Automatic, Control and Robotics

Object recognition and grasping using bimanual robot

Annex

Author: Aleix Ripoll Ruiz

Director: Jan Rosell

Call: October 2016

ETSEIB
Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

UPC
UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Glossary

DLS	Damped Least Squares
DoF	Degrees of Freedom
FK	Forward Kinematics
GA2H	Grasping with two Allegro Hands library
GIKL	General Inverse Kinematics Library
GUI	Graphical User Interface
IK	Inerse Kinematics
IOC	Institut d'Organització i Control de sistemes industrials
KDL	orocos Kinematics and Dynamics Library
PCL	Point Cloud Library
ROS	Robot Operating System
SVD	Singular Value Decomposition
UPC	Universitat Politècnica de Catalunya
URDF	Unified Robot Description Format
XML	eXtensible Markup Language

Contents

ANNEXES 1

A. General Inverse Kinematics Library 1

1. Robot kinematics theory 1
2. Description of the GIKL library 8
3. Evaluation 11
4. Further work 15
5. Short tutorial 15

B. Object pose estimation extra figures 18

1. Figures of scenario 1: Kinect One with aerial front view 18
2. Figures of scenario 2: Kinect 360 with aerial front view 20
3. Figures of scenario 3: two Kinects 360 with lateral view 22
4. Figures of scenario 4: Kinect One and two Kinects 360 24

C. User Guide of Object Recognition and Grasping 26

1. Dependencies 26
2. Launching the application 26

BIBLIOGRAPHY 28

Annexes

A. General Inverse Kinematics Library

1. Robot kinematics theory

Kinematics is a branch of physics that studies the motion of systems without the consideration of forces, mass or moments that cause motion. Thus, robot kinematics refers the analytical study of the motion of a robot manipulator analysing the relationship between the dimensions and connectivity of kinematic chains and the pose (position and orientation) for each link of the manipulator[1].

Therefore, it is crucial to define suitable kinematic models for a robot mechanism in order to analyze the behaviour of manipulators. A manipulator is characterized by an arm that provides mobility, a wrist that confers dexterity and an end-effector that performs the robot's task. The mechanical structure of a robot manipulator consists of a sequence of rigid bodies, called links, interconnected by means of joints which provide pure rotation or translation between two consecutive links. So, a kinematic model is built with a hierarchic structure with a parent-child relationship. This means that, if a joint is rotated around an arbitrary axis, all its children will also rotate around the same axis because they derive all of its parent's transformations[1].

In fact, the robot kinematics describes the analytical relationship between the joint positions and the pose of the end-effector. Hence, the formulation of the kinematic relationships allow the study of the forward kinematic problem and the inverse kinematic problem. Forward kinematics uses the kinematic equations of a manipulator to compute the position of the end-effector from specified values for the joint parameters. In contrast, for serial manipulators inverse kinematics is a much more difficult problem because it involves the transformation of cartesian coordinates, as positions and orientations of the end-effector, into joint coordinates of a robot manipulator. In fact, the existence of revolute joints causes the problem more difficult to solve because of the presence of nonlinear equations[1]. The following figure shows the relationship between forward and inverse kinematics:

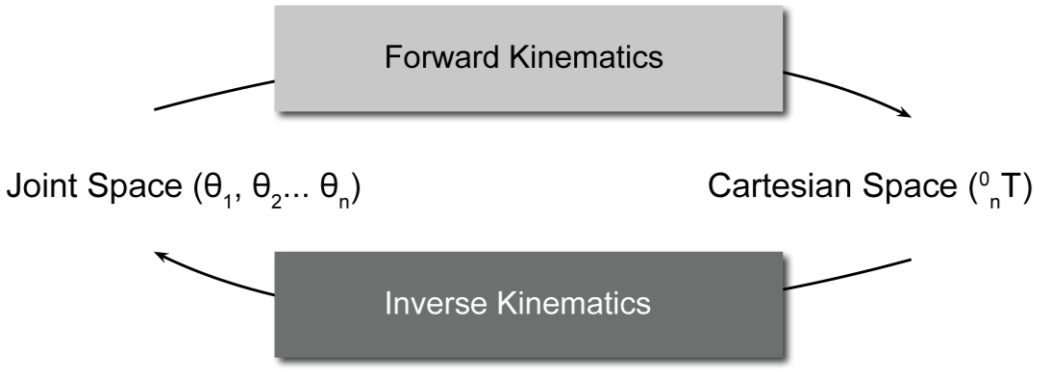

Figure A.1. The schematic representation of forward and inverse kinematics.

In addition, there are two main techniques that are used in order to solve an inverse kinematic problem: the analytical and numerical one [2]. Through the first method, the joint variables are solved analytically according to a given end-effectors' pose, while through the second one, the joint variables are obtained based on an iterative technique. This chapter has focused on explaining the numerical solutions rather than the analytical.

There are mainly two different spaces used in kinematics modelling of manipulators called cartesian space and quaternion space [3]. The transformation between two cartesian coordinate systems can be decomposed into a rotation and a translation. Homogeneous transformations based on 4x4 real matrices have been used most often within the robotics community. The homogeneous matrix is described below in a block-partitioned form [1]:

$$T_A^B = \begin{bmatrix} \boldsymbol{R}_A^B & \boldsymbol{P_A^B} \\ \boldsymbol{0^T} & 1 \end{bmatrix}$$

where $\boldsymbol{R}_A^B$ is the relative orientation and $\boldsymbol{P_A^B}$ is the relative position of frame A relative to frame B.

In the end, the forward kinematics of the end-effector with respect to the base frame is determined by multiplying all the homogeneous transformation matrices described in this section, where **n** is the number of robot manipulator joints:

$$^{base}_{EE}T = {}^0_1T\,{}^1_2T \ldots {}^{n-1}_nT$$

Denavit and Hartenberg presented in 1955 a general transformation between two joints, which requires four parameters known as DH parameters [4]. The parameters a_{i-1} , α_{i-1} , d_i and θ_i are the link length, link twist, link offset and joint angle, respectively and they have become the standard for describing robot kinematics. The matrix transformation of the modified DH parameters for a generic link is described below [1]:

$$^{i-1}T = R_X(\alpha_{i-1})D(a_{i-1})R_z(\theta_i)D_Z(d_i) =$$
$$= \begin{bmatrix} cos\ \theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \cos\alpha_{i-1}\sin\theta_i & \cos\alpha_{i-1}\cos\theta_i & -\sin\alpha_{i-1} & -d_i\sin\alpha_{i-1} \\ \sin\alpha_{i-1}\sin\theta_i & \sin\alpha_{i-1}\cos\theta_i & cos\ \alpha_{i-1} & d_i\cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Denavit-Hartenberg convention allows the construction of the forward kinematics function by the composition of the individual coordinate transformations into one homogeneous transformation matrix. The procedure can be applied to any open kinematic chain.

According to Craig [5] and Siciliano [1], the degree of freedom (DoF) of a mechanical system is defined as the number of independent variables or coordinates required to completely specify the configuration of the mechanical system. In other words, the degree of freedom explains how a joint can move in its space.

Articulations provide relative rotation and translation motions between two links and, as an articulation can rotate around an axis, it is said to have one degree of freedom. The most used joints in robotics are: prismatic joint, that translates a point along an axis and revolute joint, which rotates a point around an axis.

There are two main classes of robot manipulators: serial or open manipulators and parallel or closed manipulators. A closed kinematic manipulator is defined when a sequence of links forms a loop [1]. An open-loop mechanism is defined as a robot tree which could contain different branches. An unbranched tree is known as robot chain. Other types of systems with specialized kinematics equations are air, land, and submersible mobile robots, hyper-redundant or snake robots and humanoid robots.

The dimensions of the manipulator and its kinematics equations define the volume of space reachable by the manipulator, known as its workspace. The following shows the working space of cartesian manipulator and cylindrical manipulator.

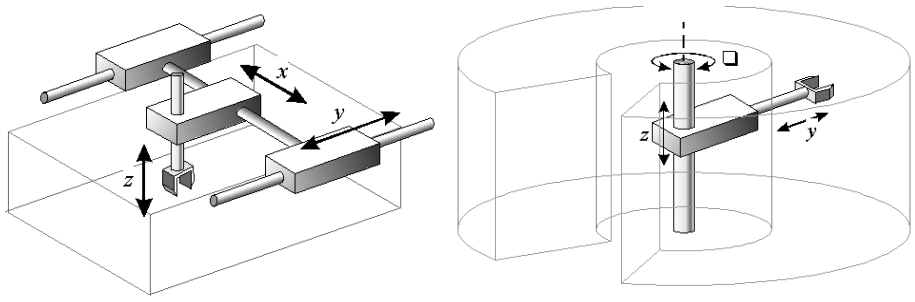

Figure A.2. Working space of cartesian and cylindrical manipulators.

Sometimes, it is useful to consider two definitions of workspace: Dextrous workspace and the reachable workspace. The first type is that volume of space that the robot end-effector

can reach with all orientations while the second type is that volume of space that the robot can reach in at least one orientation [5].

Inverse Kinematics

The inverse kinematics (IK) problem consists of the calculation of the joint variables given an end-effector pose. Given the equation $p = f(\theta)$ for forward kinematics, the inverse kinematics formulation can be derived as the following, where *f* is a highly non-linear operator which it is difficult to invert:

$$\theta = f^{-1}(p)$$

There are multiple solutions for the inverse kinematics problem. Figure A.3 shows a three-link planar arm with its end-effector at a certain position and orientation. The dashed lines indicate a second possible configuration with the same end-effector pose [5].

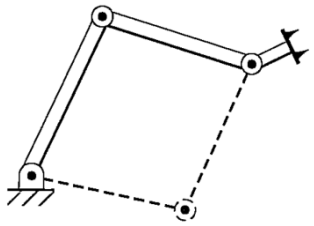

Figure A.3. Three-link manipulator with two solutions [5].

The fact that a manipulator has multiple solutions causes that the system has to be able to choose one. The criteria vary but a reasonable choice would be the closest solution from the initial starting point minimizing the amount of energy to move each joint. However, the presence of obstacle would determine another admissible solution which could not be the closest one. Furthermore, the existence of mechanical joint limits may eventually reduce the number of admissible multiple solutions for the real structure. Moreover, any IK solution exists if the specified goal point lies within the manipulator workspace.

When there is no analytical solution or it is difficult to find, it might be appropriate to use numerical solution techniques [6]. These clearly have the advantage of being applicable to any kinematic structure, but in general they do not allow computation of all admissible solutions [5].

Some iterative methods use the Jacobian matrix which is a linear approximation of kinematics. The Jacobian constitutes one of the most important tools for manipulator characterization. It is useful for finding singularities, analyzing redundancy, solving the inverse kinematic problem, etc. The analytical Jacobian matrix *J* is a function of the θ values and is defined by:

$$J(\theta) = \frac{\partial f}{\partial \theta}$$

Each column of *J* describes an approximated change of the end effectors position when changing the corresponding joint positions [7]. Geometric Jacobian matrix can be used for its simplicity instead of the analytical one. The dimensions of the geometric Jacobian matrix depends on the *n*-DOF manipulator and (6x1) column vectors J_{P_1} and J_{O_1}:

$$J_{6\,x\,n}(\theta) = \begin{bmatrix} \boldsymbol{J_{P_1}} & \cdots & \boldsymbol{J_{P_n}} \\ \boldsymbol{J_{O_1}} & \cdots & \boldsymbol{J_{O_n}} \end{bmatrix}$$

where

$$\begin{bmatrix} \boldsymbol{J_{P_i}} \\ \boldsymbol{J_{O_i}} \end{bmatrix} = \begin{cases} \begin{bmatrix} \boldsymbol{z_{i-1}} \\ \boldsymbol{0} \end{bmatrix} & for\ a\ prismatic\ joint \\ \begin{bmatrix} \boldsymbol{z_{i-1}} \times (\boldsymbol{p_e} - \boldsymbol{p_{i-1}}) \\ \boldsymbol{z_{i-1}} \end{bmatrix} & for\ a\ revolute\ joint \end{cases} \quad [1]$$

$$\boldsymbol{z_{i-1}} = \boldsymbol{R_1^0}(q_1) \ldots \boldsymbol{R_{i-1}^{i-2}}(q_{i-1})\boldsymbol{z_0}$$

$$\boldsymbol{p_e} = \boldsymbol{T_1^0}(q_1) \ldots \boldsymbol{T_n^{n-1}}(q_n)\boldsymbol{p_0}$$

$$\boldsymbol{p_{i-1}} = \boldsymbol{T_1^0}(q_1) \ldots \boldsymbol{T_{i-1}^{i-2}}(q_{i-1})\boldsymbol{p_0}$$

Then, the forward kinematics problem is formulated as follows:

$$p = f(\theta) \rightarrow \Delta P = J(\theta)\Delta\theta$$

Now, the forward kinematics equation ΔP is a linear approximation and easier to resolve. By inverting $J(\theta)$, the inverse kinematics equation can be written as the following, where $\Delta \mathrm{P} = \vec{\mathbf{e}} = \mathbf{t} - \mathbf{P}$:

$$\Delta\theta = J(\theta)^{-1}\Delta \mathrm{P}$$

$\vec{\mathbf{e}}$ is the desirable change of the end effector, ***t*** is the target pose and ***P*** is the current end effector pose [7].

Given a manipulator with a set of **n** articulations and an end effector *P*, a Jacobian matrix can be created and inverted to solve the inverse kinematic problem. However, the solutions are linear approximations of θ and the equation needs to be done repeatedly until is sufficiently close to a solution [7].

$$\theta := \theta + \Delta\theta$$

Then, the forward kinematics is computed to obtain the new current pose of the end effector with the new θ and check if $\vec{\mathbf{e}}$ is converging to zero. In most cases, the Jacobian

J may not be square or invertible and even if it is invertible, *J* may work poorly as it may be nearly singular [7]. Several approaches have been proposed to overcome these problems.

Solvers of inverse kinematics

There are several methods for solving inverse kinematic problems numerically. These include cyclic coordinate descent methods, pseudoinverse methods, Jacobian transpose methods, the Levenberg-Marquardt damped least squares methods, quasi-Newton and conjugate gradient methods and neural net and artificial intelligence methods [7]. In this section, three Jacobian IK solvers have been presented: Transpose method, Pseudoinverse method and Damped Least Squares method. All methods mentioned try to choose an appropriate $\boldsymbol{\Delta\theta}$ to converge to a solution.

a) Jacobian Transpose method

The Jacobian transpose method is introduced by Balestrino [8] and the basic idea is to use the transpose of *J* instead of the inverse of *J*:

$$\Delta\theta = \alpha J^T \vec{\mathbf{e}}$$

One way to compute an adequate α is:

$$\alpha = \frac{\langle \vec{\mathbf{e}}, \mathrm{J}\,\mathrm{J}^{\mathrm{T}}\,\vec{\mathbf{e}} \rangle}{\langle \mathrm{J}\,\mathrm{J}^{\mathrm{T}}\,\vec{\mathbf{e}},\ \mathrm{J}\,\mathrm{J}^{\mathrm{T}}\,\vec{\mathbf{e}} \rangle}$$

b) Jacobian Pseudoinverse method

The pseudoinverse method sets the value $\Delta\theta$ equal to:

$$\Delta\theta = J^{\dagger}\,\vec{\mathbf{e}}\,,$$

where matrix ($J^{\dagger}$) it the Moore-Penrose inverse. A problem of the pseudoinverse method is that it has stability problems near singularities. The matrix $J^{\dagger}$ is computed by the singular value decomposition (SVD) of *J* which can be expressed in the form [7]:

$$J = UDV^T$$

U and V are orthogonal matrices and D is diagonal with σ_i elements. Then, *J* can be rewritten in the form:

$$J = \sum_{i=1}^{m} \sigma_i u_i v_i^T$$

The pseudoinverse of *J* is equal to:

$$J^{\dagger} = VD^{\dagger}U^T$$

Thus:

$$J^{\dagger} = \sum_{i=1}^{r} {\sigma_i}^{-1} v_i u_i^T$$

c) Jacobian Damped Least Squares method

The damped least squares (DLS) method deals with many of the pseudoinverse method's problems with singularities. It was first used for inverse kinematics by Wampler [9] and Nakamura and Hanafusa [10].

This method attempts to find the value of $\Delta\theta$ that minimises the quantity:

$$\min \{ \| J\, \Delta\theta \; - \vec{\mathbf{e}} \|^2 + \lambda^2 \, \| \Delta\theta \|^2 \} \, ,$$

where $\lambda \in \mathbb{R}$ is a non-zero damping constant. There are many methods to choose correctly the value of λ [11]. The previous statement can be rewritten as:

$$(J^T J + \; \lambda^2 I) \, \Delta\theta = J^T \, \vec{\mathbf{e}}$$

Then, the damped least squares solution can be expressed in the form:

$$\Delta\theta = (J^T J + \; \lambda^2 I)^{-1} \, J^T \, \vec{\mathbf{e}}$$

with

$$J^T \, (J \, J^T + \; \lambda^2 I)^{-1} = \sum_{i=1}^{r} \frac{\sigma_i}{{\sigma_i}^2 + \lambda^2} v_i u_i^T$$

The damped least squares behaves similarly to the pseudoinverse method since for large σ_i, then $\frac{\sigma_i}{{\sigma_i}^2+\lambda^2} \approx {\sigma_i}^{-1}$. However, when σ_i is of the same order of magnitude as λ or smaller, both methods differ. Therefore, for any $\lambda > 0$, the DLS tends to act similarly to the pseudoinverse away from singularities but improves the pseudoinverse in the neighbourhood of singularities.

The damping constant λ t has to be chosen correctly in order to obtain good results near singularities. There are many methods for selecting damping constants based on the configuration of the articulated manipulator [10], [11], [12]. One of the most used λ is:

$$\lambda \; = \begin{cases} \lambda_0 \left(1 - \dfrac{\mathrm{w}}{\mathrm{w_t}}\right)^2, & \text{if } \mathrm{w} < \mathrm{w_t} \\ 0, & \text{if } \mathrm{w} \geq \mathrm{w_t} \end{cases}$$

with

$$w = \sqrt{\det(J \, J^T)}$$

2. Description of the GIKL library

General Inverse Kinematics Library is designed to be able to solve as many robot topologies as possible from tree-like structures to simple chain manipulators with proper performance. Close-loop manipulators, mobile manipulators and collision avoidance obstacles strategy have been not included yet. C++ programming language has been chosen to develop the GIKL library. The solver will be an initial core of a general inverse kinematic library currently nonexistent. A diagram of the code structure is shown in Figure A.4:

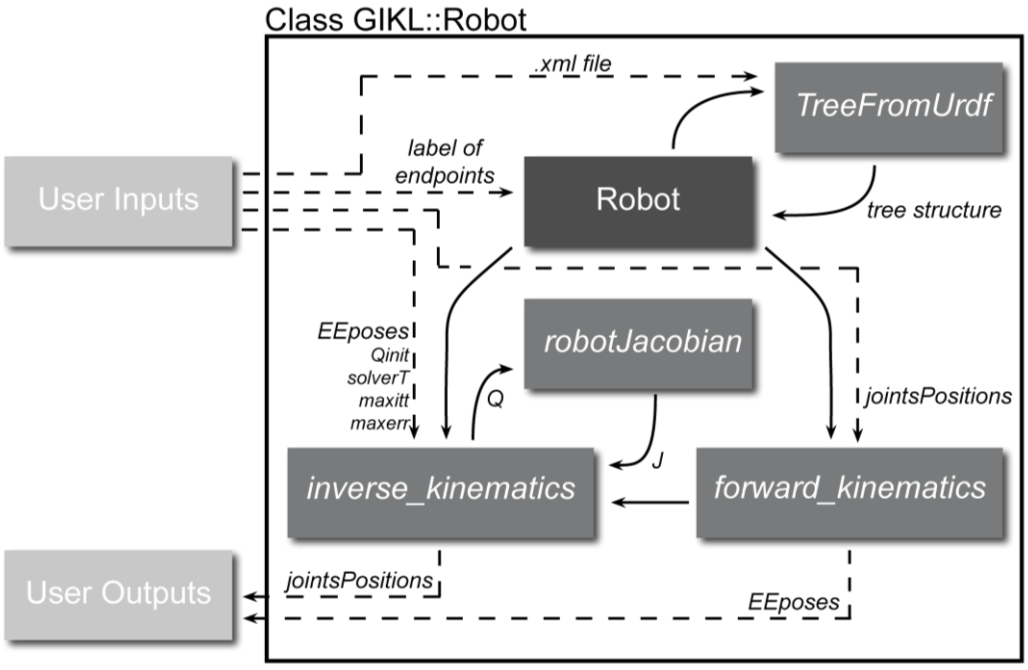

Figure A.4. Diagram showing the different structural blocks and flow of the developed library.

The library starts by reading two input user parameters: URDF file with the robot structure description and a vector containing the labels of **n** end effectors of the manipulator. A robot tree or chain structure is created. At this point, the user can determine other inputs parameters according to his/her necessities. Forward kinematic solver needs the joint positions as an input and returns a vector of the end effectors poses. Inverse kinematic solver computes the joint positions with initial joint positions, a method (pseudoinverse, transpose or DLS), a maximum iteration and error allowed and the vector of the desired end-effectors poses.

TreeFromUrdf function

This block consists of constructing a tree or chain structure manipulator reading from URDF file. This allows the system to know the number of links and joints of the robot, the parent-child link relationships and joint limits.

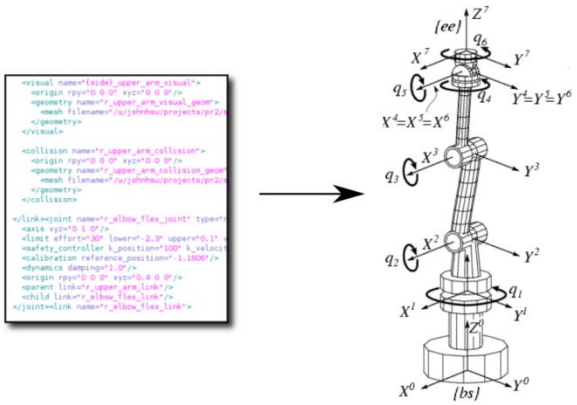

Figure A.5. Construct a KDL tree from an XML robot representation in URDF.

forward_kinematics function

KDL's forward kinematics solver has been introduced in this module because of its simplicity and high performance. Forward kinematics theory explained in section 1 is considered in this part.

robotJacobian function

How a geometric Jacobian can be constructed, has been explained in section 1. KDL contains a method which provides the Jacobian of a manipulator. However, it is convenient to use it, exclusively, for chain structure manipulators because of accumulative numeric errors for constructing a Jacobian of tree robot structure. If the Jacobian matrix has some numeric errors, inverse kinematic solution will contain error and will increase the time to converge to a valid solution.

This function has two options: uses the Jacobian computed by the KDL solver (recommended for chain manipulators) or uses a specific Jacobian that was introduced off-line in the code (recommended to tree structures).

inverse_kinematics function

This function implements three Jacobian inverse kinematics algorithms which have been explained in section 1. First, a pseudocode about common IK algorithm has been presented. Common algorithm is based on theory analyzed in section 1.

Algorithm 1 Common algorithm

```
procedure INVERSE_KINEMATICS(Q_init, EE_goal, max_iter, max_err)
    while error > max_err or itt < max_iter do
        Δe ← EE_goal − EE_curr
        error ← ||Δe||
        J ← robotJacobian(Q_itt)                    ▷ Geometric Jacobian
        Compute ΔQ ← solverType                     ▷ Non common part
        Q_itt+1 := Q_itt + ΔQ_itt
        if not tree_structure then
            if not jacobian_transpose then
                if inLocalMinima(Q_itt+1, Q_itt) then
                    Q_itt+1 := Q_rand               ▷ random Q to unstick
                end if
            end if
        end if
        EE_curr ← forward_kinematics(Q_itt+1)       ▷ Forward Kinematics
    end while
    return (Q, error)
end procedure
```

Figure A.6. Pseudocode 1: Common algorithm.

Second, three different methods to compute $\boldsymbol{\Delta\theta}$ have been presented: Jacobian Transpose method, Pseudoinverse method and DLS method.

Algorithm 2 Jacobian transpose algorithm

```
procedure JACOBIAN TRANSPOSE(J, Δe)
    Compute J^T
    Compute α ← ⟨Δe, JJ^TΔe⟩ / ⟨JJ^TΔe, JJ^TΔe⟩
    ΔQ = αJ^TΔe
    return ΔQ
end procedure
```

Figure A.7. Pseudocode 2: Jacobian Transpose algorithm.

Algorithm 3 Pseudoinverse algorithm

```
procedure PSEUDOINVERSE(J, Δe)
    Compute U · D · V^T ← J                         ▷ singular value decomposition
    pD_ii = Σ_{i=1}^{r} σ_i^{-1}
    Compute J† ← V · pD · U^T
    ΔQ = J†Δe
    return ΔQ
end procedure
```

Figure A.8. Pseudocode 3: Jacobian Pseudoinverse algorithm.

Algorithm 4 DLS algorithm

1: **procedure** DLS($J, \Delta e$)
2: Compute $U \cdot D \cdot V^T \leftarrow J$ ▷ singular value decomposition
3: Compute J^T
4: $w = \sqrt{det(JJ^T))}$ ▷ manipulability measure
5: **if** $w < w_t$ **then**
6: $\lambda = \lambda_0(1 - \frac{w}{w_t})^2$
7: **else**
8: $\lambda = 0$
9: **end if**
10: $pD_{ii} = \sum_{i=1}^{r} \frac{\sigma}{\sigma^2+\lambda^2}$
11: Compute $J^\dagger \leftarrow V \cdot pD \cdot U^T$
12: $\Delta Q = J^\dagger \Delta e$
13: **return** ΔQ
14: **end procedure**

Figure A.9. Pseudocode 4: Jacobian DLS algorithm.

3. Evaluation

This section presents the methodology followed to evaluate the GIKL library, shows a comparison between Orocos' KDL IK solver, presents a brief ROS node description to execute all tests and which models and parameters have been taken into account.

A ROS package has been built to manage the input parameters of the user, load the desired manipulator from XML file, visualize in real time the manipulator actions and to analyse the performance of the GIKL library algorithms.

Two purposes were set for this ROS package: first, to visualize the manipulator while user enters an action and see if the solutions have coherence and second, to analyse in deep the performance of the GIKL library. For this reason, two independent nodes have been created:

- *robot_ik_node*: computes the inverse kinematic solution using GIKL library and publish the */joint_states* topic to visualize later using *rviz* package.
- *algorithm_tests*: computes the solve rate and average time to find a solution for the three Jacobian methods implemented and compares with the Orocos' KDL results.

The *robotfik* package has the following structure:

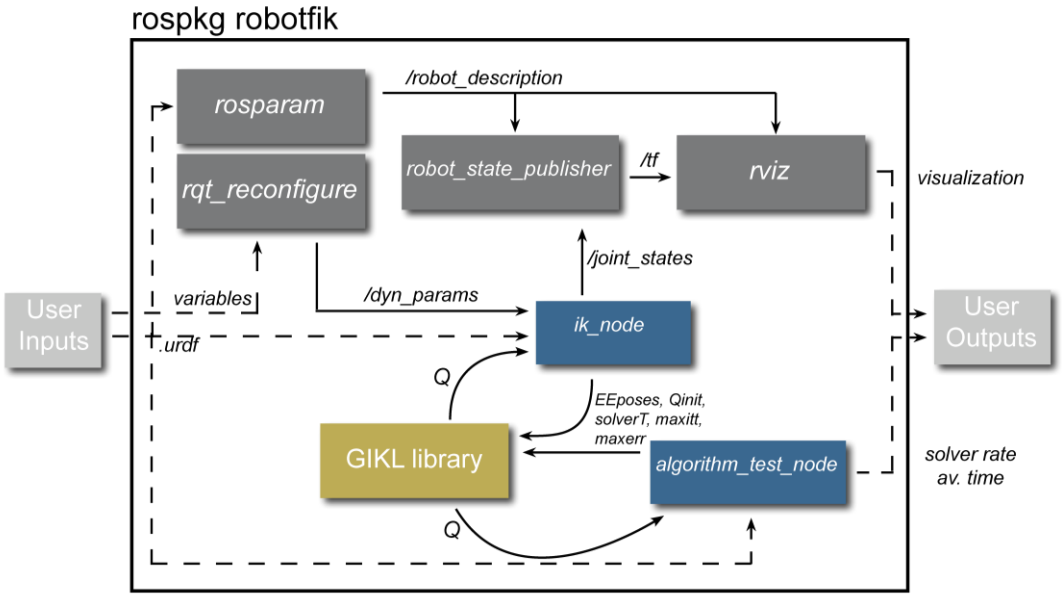

Figure A.10. Diagram showing the robotfik package structure and parameters flow.

Briefly, all packages descriptions have been presented below:

- *robotfik*: manages the whole system allowing the correctly communication between nodes and other packages.
- *Rosparam* [13]: contains the *rosparam* command-line tool for getting and setting ROS Parameters on the Parameter Server. Here, *robot_description* is set by reading URDF file.
- *robot_state_publisher* [14]: takes the joint angles of the manipulator as input and publishes the 3D poses of the robot links using a kinematic tree model of the robot. It uses the URDF specified by the parameter *robot_description* and the joint positions form topic */joint_states* to calculate the forward kinematics and publish the results via */tf*.
- *rqt_reconfigure* [15]: provides a mechanism to view and to change node parameters at any time without having to restart the node.
- *rviz* [16]: this package provides 3D visualization which can navigate with the mouse, adds robot model visualization and its joints *tf* positions.

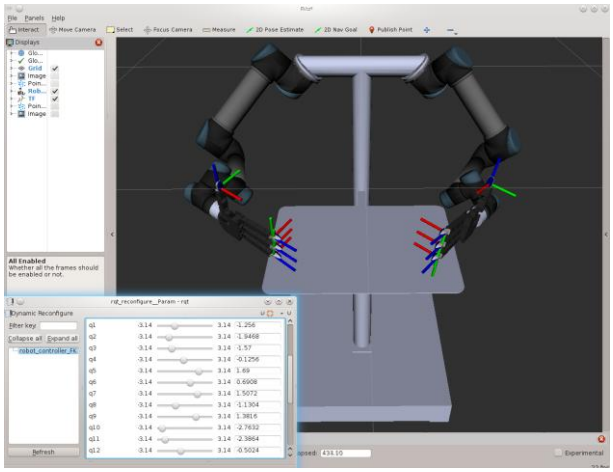

Figure A.11. Rviz platform and rqt_reconfigure tool.

In order to quantify the performance of the different IK methods, some tests have been executed by running *algorithm_tests* node which has the following actions:

1. Selecting a kinematic tree manipulator.
2. Generating a random joint value for each joint in the tree.
3. Performing FK to determine the target cartesian pose from the random joints.
4. Calling each IK implementation for the target pose.
5. Repeating the steps 2 to 4 while summing the number of successful IK solvers and summing the time in milliseconds for each simulation.
6. Averaging the success rate and solve time for each IK implementation after 1000 random samples.

A solve was considered successful if the difference in each dimension between the pose and the target pose is no more than error =1e-6. A maximum number of iteration has been considered to converge to a solution but there is no maximum timeout for testing. The robot models that were used for comparison are:

a) 3R planar
b) UR5
c) ABB irb5400
d) Kuka LWR 4+
e) AllegroHand
f) UR5+ AllegroHandL + UR5+ AllegroHandR

The results presented in Table A.1 and Table A.2 are temporary. The library is still under development and the results obtained are not quite good. It has been decided to implement analytical Jacobian for each known manipulators instead of using the Jacobian given by KDL.

Here, Table A.1 and Table A.2 show the summary of the test completed by using *algorithm_tests* node:

Table A.1. Results for reachable and random pose for Chain structures.

Chain	DOFs	Qinit	Total tests	max_{itt}	max_{err}	Orocos' KDL pinv solve rate	Orocos's KDL pinv avg time	Orocos' KDL wdls solve rate	Orocos's KDL wdls avg time	GIKL pinv RR solve rate	GIKL pinv RR avg time	GIKL tr solve rate	GIKL tr avg time	GIKL DLS RR solve rate	GIKL DLS RR avg time
3R planar	3	0	1000	100	1,00E-06	95,90%	0,21ms	0,00%	5,35ms	95,90%	1,75ms	77,60%	80,87ms	95,40%	3,17ms
UR5	6	0	1000	100	1,00E-06	87,70%	0,722ms	0,10%	1,54ms	87,50%	4,22ms	49,00%	183,04ms	88,90%	7,79ms
irb5400	7	0	1000	100	1,00E-06	93,70%	0,708ms	0,00%	26,76ms	93,90%	3,36ms	74,40%	118,46ms	93,40%	6,46ms
kukaLWR	7	0	1000	100	1,00E-06	100,00 %	0,54ms	0,00%	25,96ms	100,00 %	2,35ms	63,30%	158,63ms	100,00 %	4,88ms

Table A.2. Results for reachable and random pose for Tree structures.

Tree	DOFs	Qinit	Total tests	max_{itt}	max_{err}	Orocos' KDL pinv solve rate	Orocos' KDL pinv avg time	Orocos' KDL wdls solve rate	Orocos' KDL wdls avg time	GIKL pinv solve rate	GIKL pinv avg time	GIKL tr solve rate	GIKL tr avg time	GIKL DLS solve rate	GIKL DLS avg time
UR5+AHL+UR5+AHR	44	0	1000	100	1,00E-06	X	X	0,00%	67,57ms	-	-	-	-	96.55%	1555ms

4. Further work

Improvements:

I. Kinematic tree structure can be optimized.
II. Use some kind of files to load a robot such as .dh files.
III. Joint limits and collisions.
IV. Introduction of geometric algebra.
V. Closed-form solutions:
   - Identification of local and global singularities
   - Study of the area and volume of the workspace
   - Identification (for redundant manipulators) of redundant joints.

5. Short tutorial

- Create your robot model in URDF format. Simple example is shown below:

```
<?xml version="1.0"?>
<robot name="test_robot">
          <link name="base_link"/>
          <link name="link1">
            <visual>
              <origin xyz="0.175 0 0" rpy="0 0 0" />
              <geometry>
                 <box size="0.35 0.01 0.01"/>
              </geometry>
            </visual>
          </link>
          <link name="link2">
             <visual>
              <origin xyz="0.175 0 0" rpy="0 0 0" />
              <geometry>
                 <box size="0.35 0.01 0.01"/>
              </geometry>
            </visual>
          </link>
         <joint name="joint1" type="continuous">
           <parent link="base_link"/>
           <child link="link1"/>
           <origin xyz="0 0 0" rpy="1.57 0 0"/>
           <axis xyz="0 0 1"/>
          </joint>
          <joint name="joint2" type="continuous">
           <parent link="link1"/>
           <child link="link2"/>
           <origin xyz="0.35 0 0" rpy="0 0 0"/>
           <axis xyz="0 0 1"/>
          </joint>
</robot>
```

- Apply simple forward kinematic problem to your robot model. Short code example is shown below:

```
#include “gikl_functions.h”

Eigen::VectorXd Q(2); //input: joint array with size 2
std::vector<Eigen::VectorXd> EEpose; //output: end effector’s pose
int main ( int argc, char** argv)
{
        //Definition of a kinematic chain
        GIKL::Robot robot_example(“/home/../../urdf/robot_example.urdf”);
        //Assign some values to the joint positions
        Q(0) = 0.78539;
        Q(1) = 1.5708;
        //Create solver based on kinematic chain
        bool sol = robot_example.forward_kinematics(Q, EEpose);
        std::cout<<”EEpose: “<< Epose.at(0)<<std::endl;
{
```

- Apply simple inverse kinematic problem to your robot model. Short code example is shown below:

```
#include “gikl_functions.h”

Eigen::VectorXd q_init(2);
const int maxiter = 100;
const double err = 1e-3;
std::vector<Eigen::Vector3d> position(1); //input: end effector’s position
std::vector<Eigen::Vector3d> orientation(1); //input: end effector’s orientation
GIKL::SolverType solverName = GIKL::JACOBIAN_DAMPING_LEAST_SQUARES;
const double error_weight = 0.25; //0.75 position error– 0.25 orientation error
Eigen::VectorXd Q(2); //output: joint array with size 2
std::vector<double> final_error;//output: error
int main ( int argc, char** argv)
{
        //Definition of a kinematic chain
        GIKL::Robot robot_example(“/home/../../urdf/robot_example.urdf”);
        //Assign the position and orientation of the end-effector
        position.at(0) (0) = 0.247; //x
        position.at(0) (1) = 0.0; //y
        position.at(0) (2) = -0.247; //z
        orientation.at(0) (0) = 1.5708; //roll
        orientation.at(0) (1) = -0.78539; //pitch
        orientation.at(0) (2) = 0.0; //yaw
        //Assign the initial joint positions of the robot
        q_init (0) = 0.0;
        q_init (1) = 0.0;
        //Create solver based on kinematic chain
        bool sol = robot_example.inverse_kinematics(q_init, Q, position, orientation,
solverName, maxiter, err, error_weight, final_error);
        std::cout<<”Joints positions: “<< Q<<std::endl;
        std::cout<<”Final error: “<<final_error.at(0) <<std::endl;
{
```

- The results of both problems should be as follows:

Table A.3. GIKL tutorial: Final results of the two problems.

	x	y	z	r	p	y	Q1	Q2	final_error
FK problem	0.247	0.0	0.247	0.0	-π/2	π/2	π/4	π/4	-
IK problem	0.247	0.0	-0.247	π/2	-π/4	0.0	-π/4	π/2	7.4e-4

B. Object pose estimation extra figures

The solutions obtained are presented in the following figures and a table summarising the results. The estimated position error for all simulations is represented in red plus sign and the average of all in blue. The blue circle indicates the desired error.

1. Figures of scenario 1: Kinect One with aerial front view

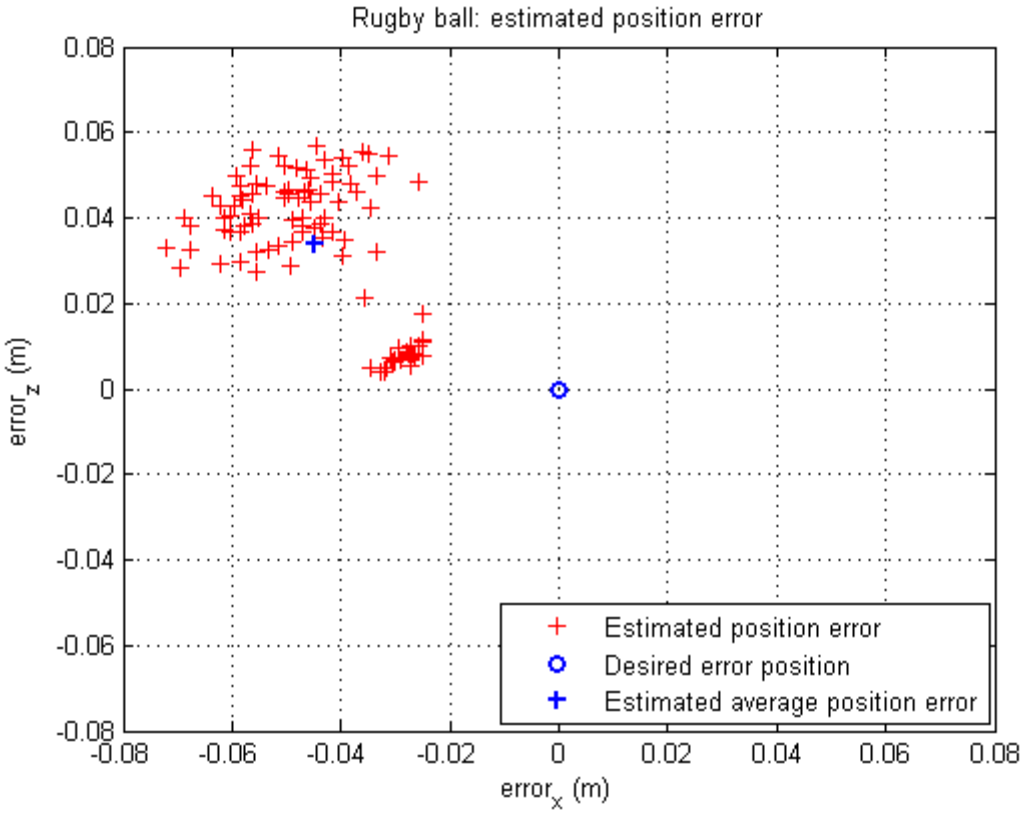

Figure B.1. Scenario 1: plane XY of the estimated position error of the Rugby ball.

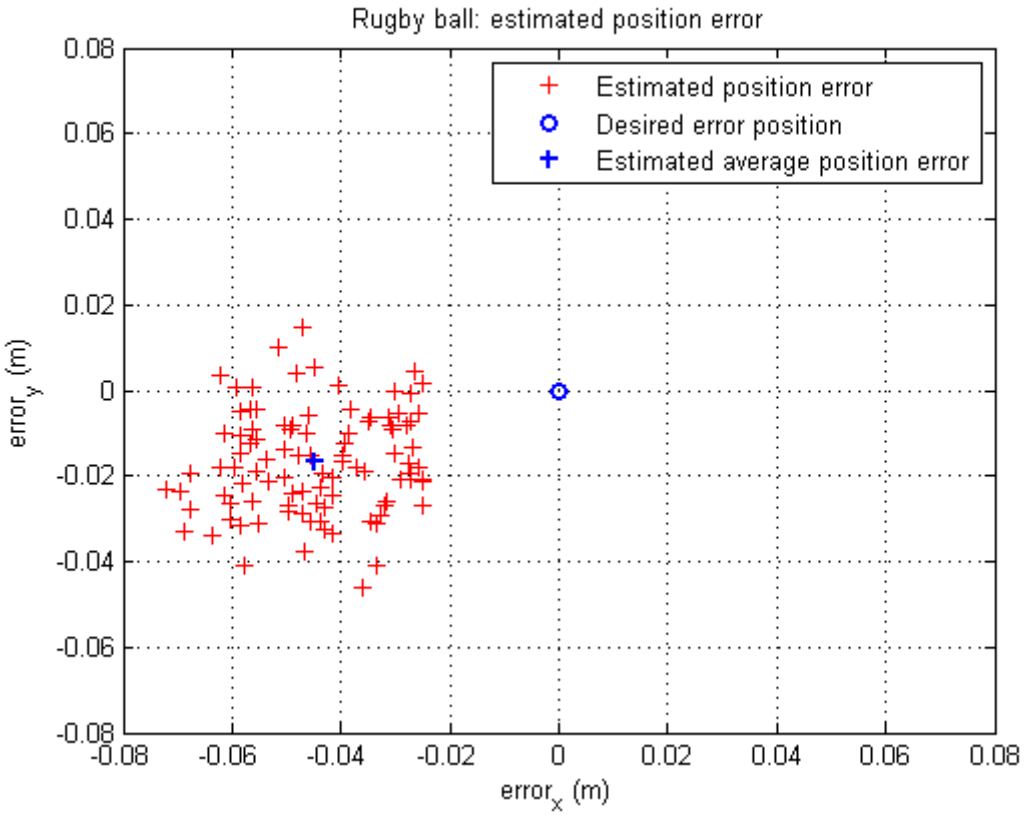

Figure B.2. Scenario 1: plane XZ of the estimated position error of the Rugby ball.

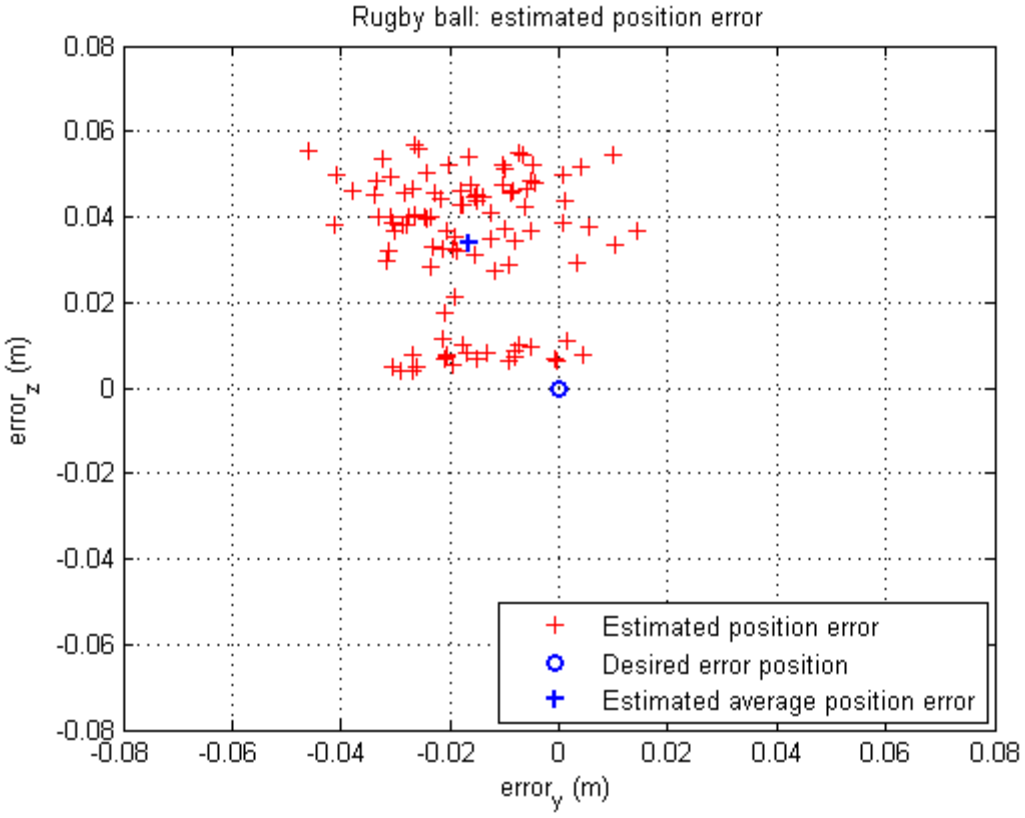

Figure B.3. Scenario 1: plane YZ of the estimated position error of the Rugby ball.

2. Figures of scenario 2: Kinect 360 with aerial front view

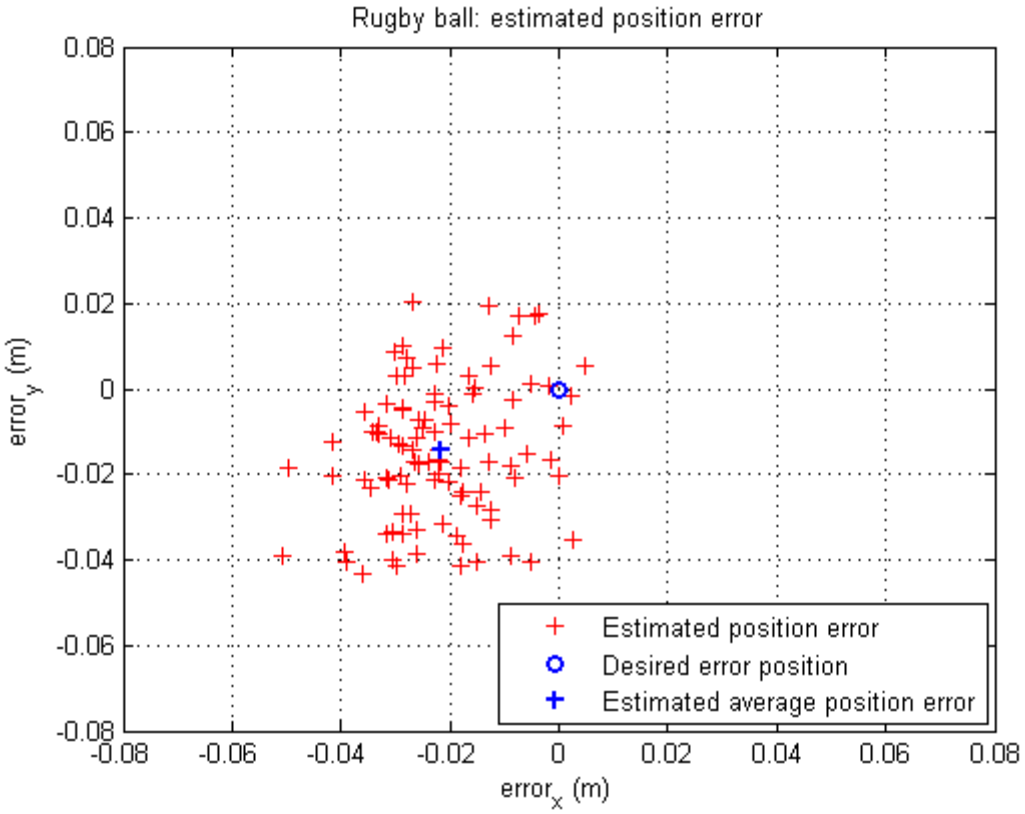

Figure B.4. Scenario 2: plane XY of the estimated position error of the Rugby ball.

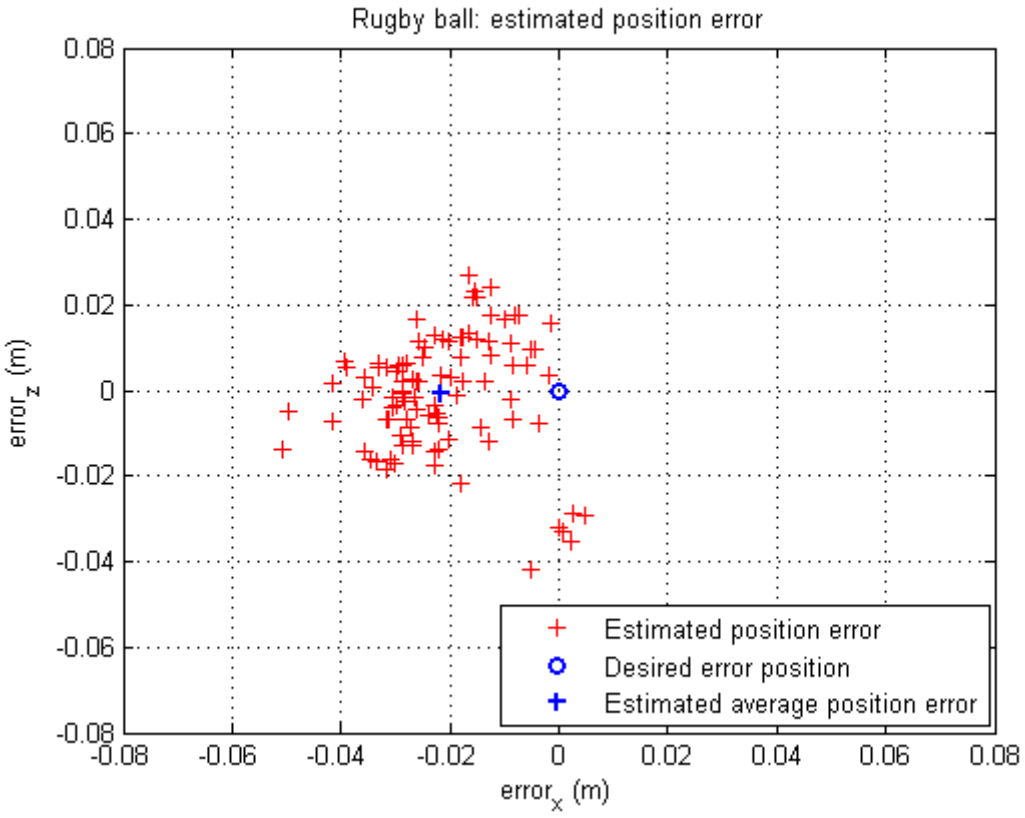

Figure B.5. Scenario 2: plane XZ of the estimated position error of the Rugby ball.

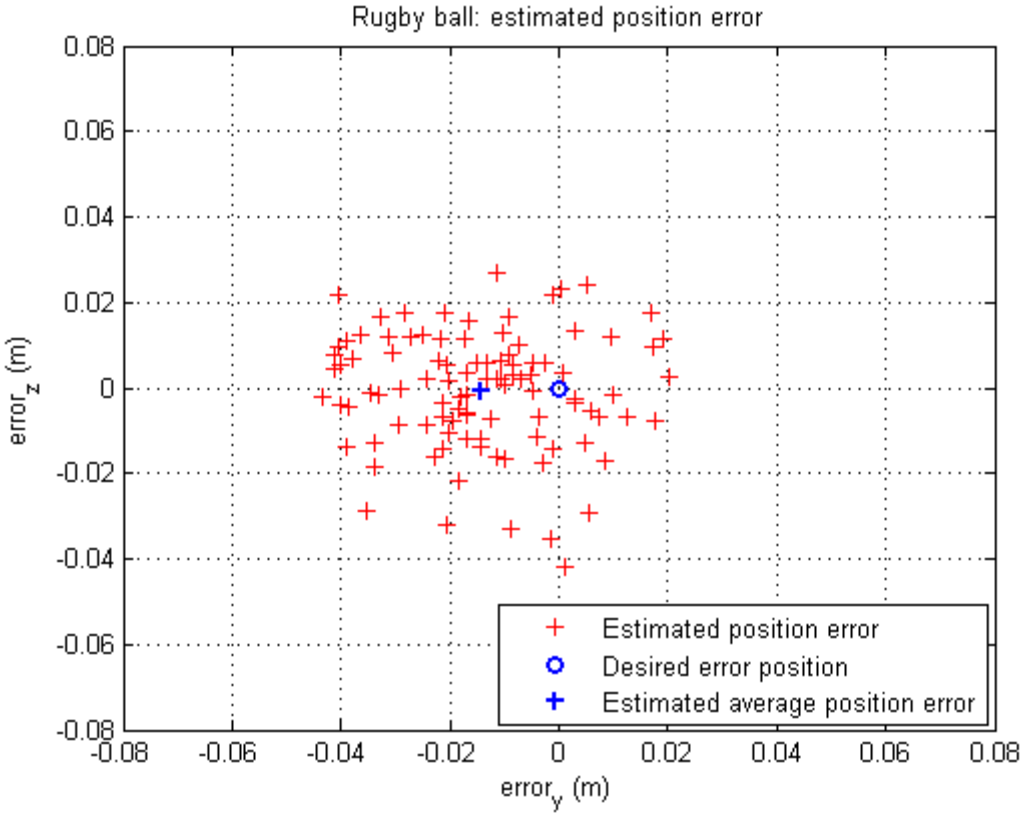

Figure B.6. Scenario 2: plane YZ of the estimated position error of the Rugby ball.

3. Figures of scenario 3: two Kinects 360 with lateral view

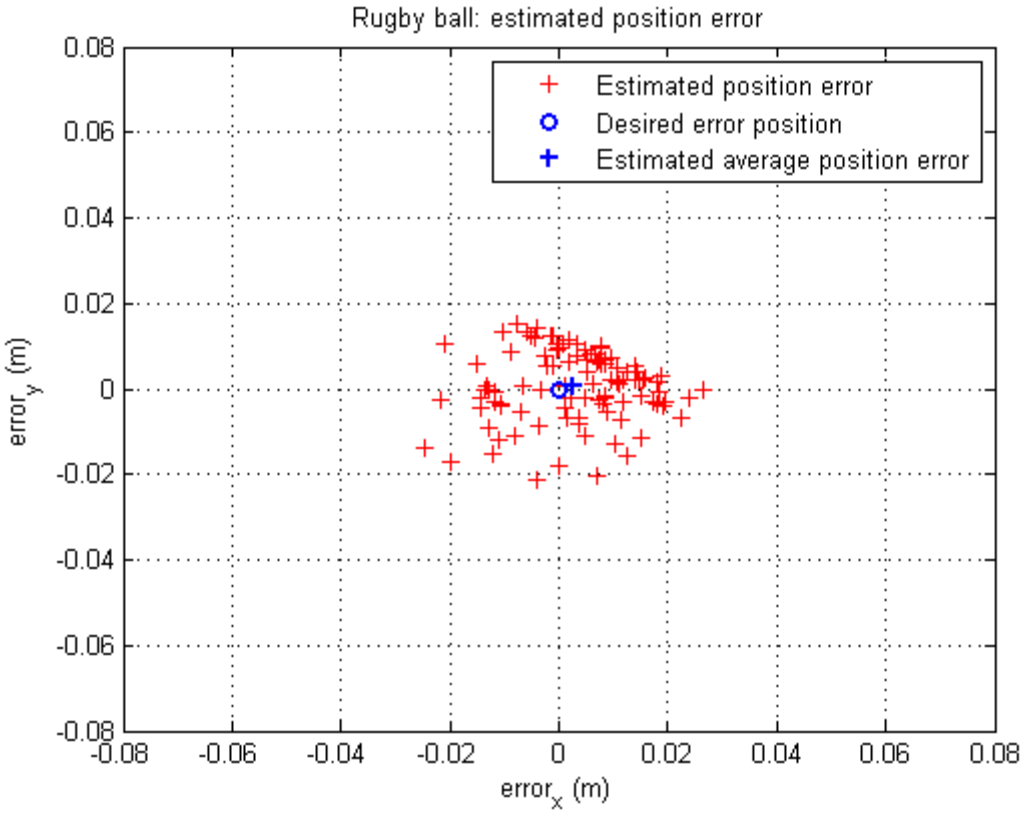

Figure B.7. Scenario 3: plane XY of the estimated position error of the Rugby ball.

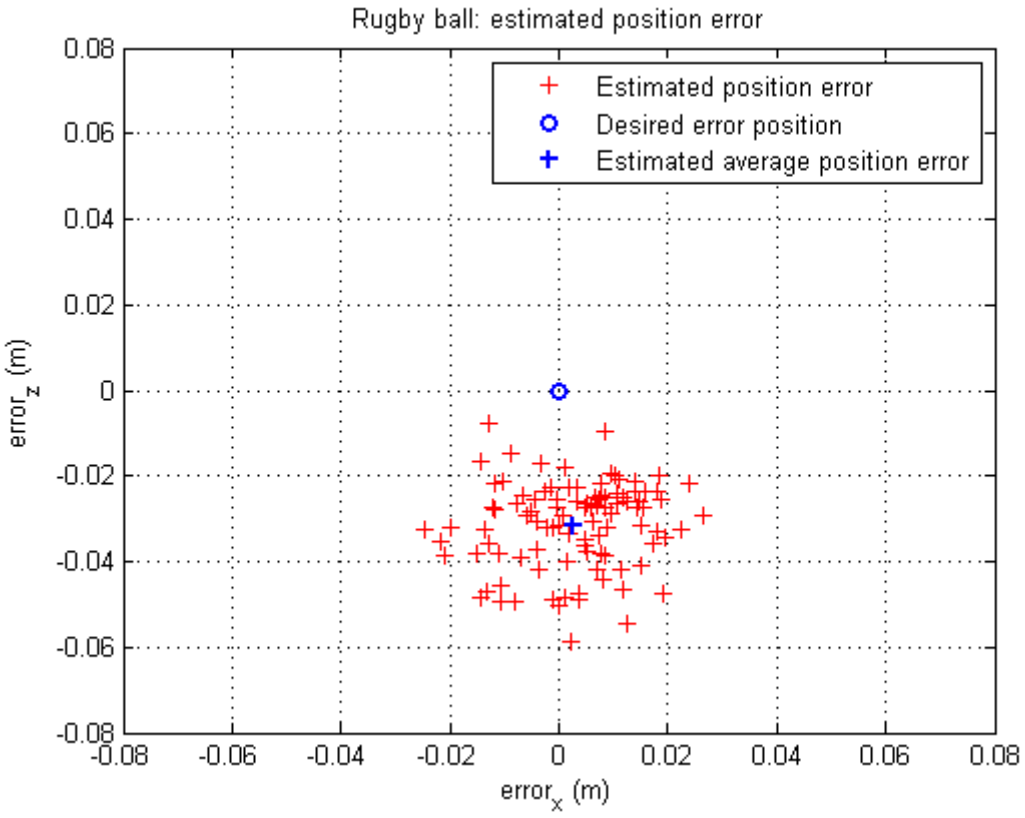

Figure B.8. Scenario 3: plane XZ of the estimated position error of the Rugby ball.

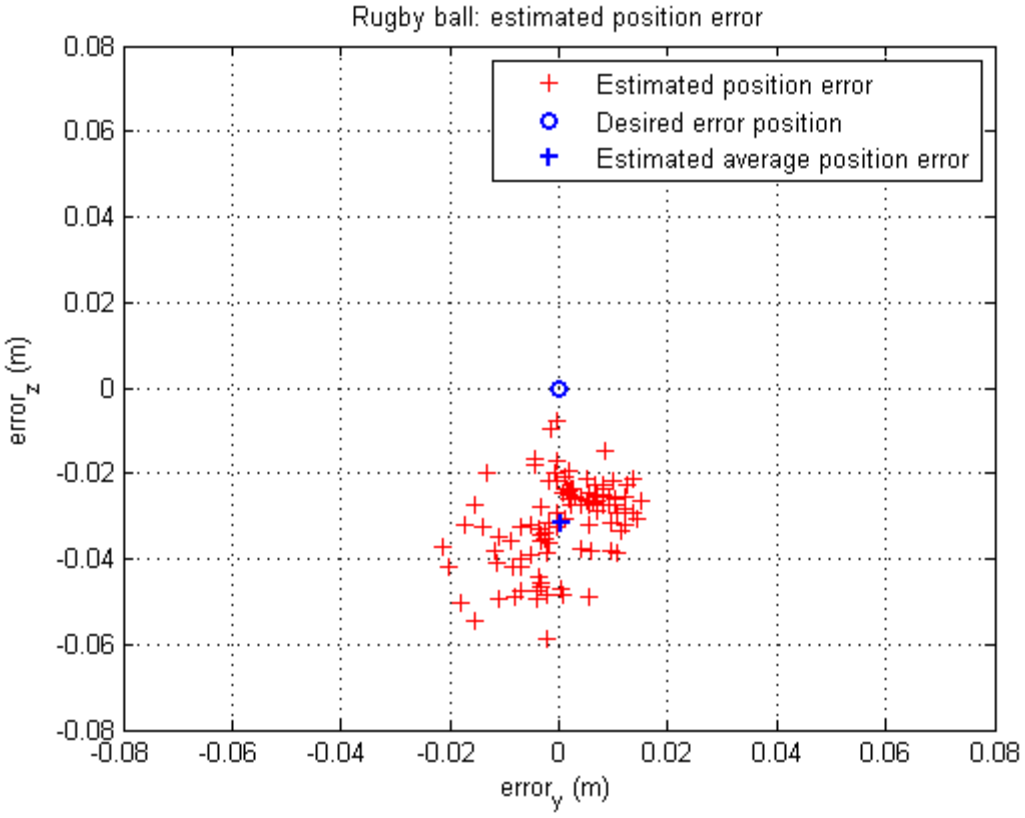

Figure B.9. Scenario 3: plane YZ of the estimated position error of the Rugby ball.

4. Figures of scenario 4: Kinect One and two Kinects 360

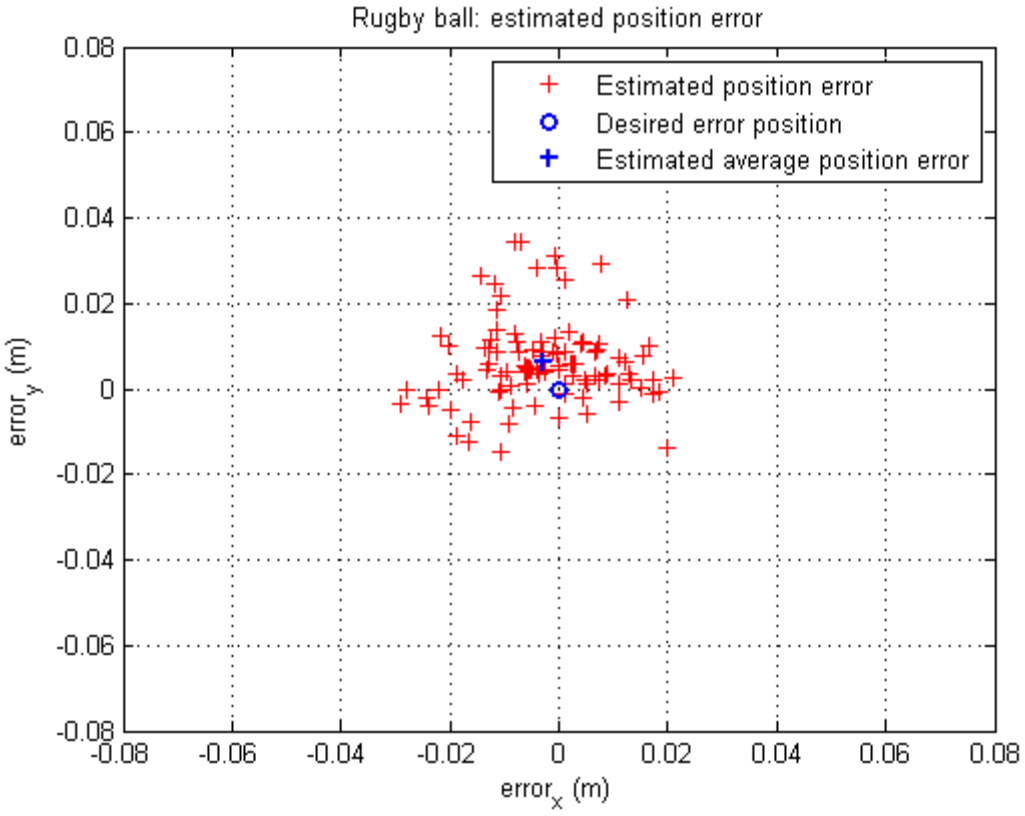

Figure B.10. Scenario 4: plane XY of the estimated position error of the Rugby ball.

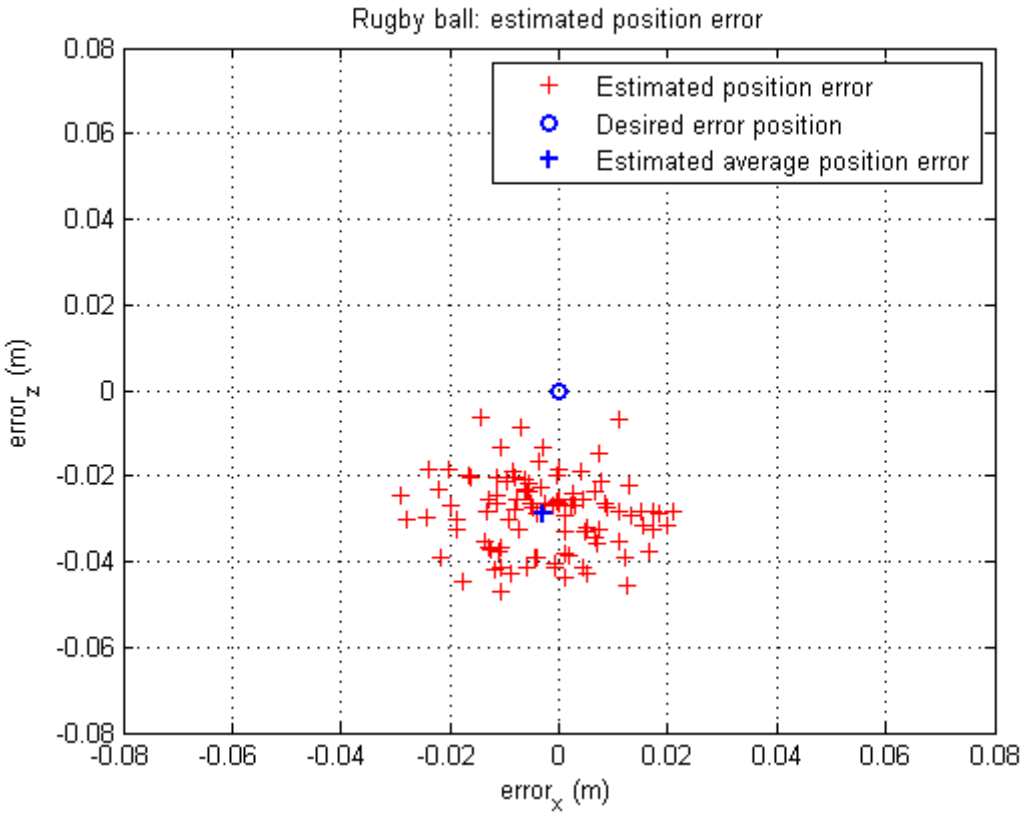

Figure B.11. Scenario 4: plane XZ of the estimated position error of the Rugby ball.

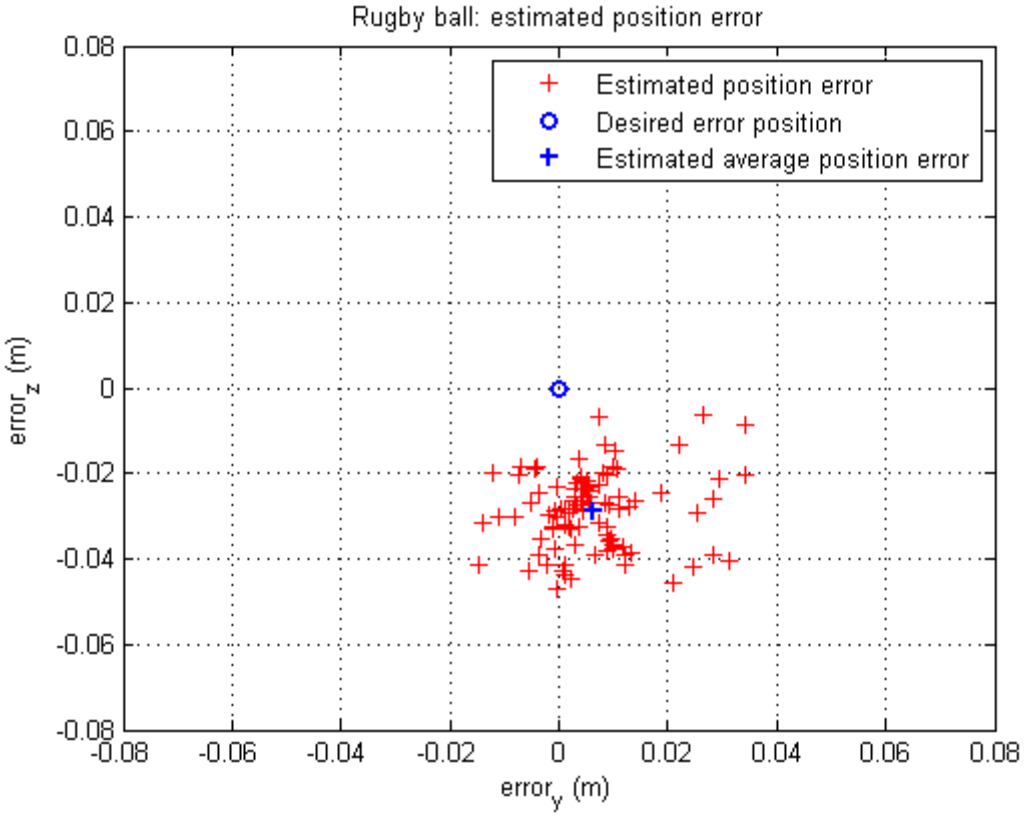

Figure B.12. Scenario 4: plane YZ of the estimated position error of the Rugby ball.

C. User Guide of Object Recognition and Grasping

1. Dependencies

- ROS
- OpenCV 2.4
- PCL 1.7
- Eigen
- OpenCL
- Libfreenect
- Libfreenect2
- OpenNI
- GIKL
- Ga2H
- Kautham
- kth_simulator

2. Launching the application

Do these steps once:

1) Install the ROS.
2) Setup your ROS environment.
3) Install drivers.
4) Clone the repository into your catkin workspace, install the dependencies and build it.
5) Clone the Kautham and kth_simulator into your catkin workspace, install dependencies and build it.
6) Connect your sensor. Check Kinect One using ./Protonect to test the device.
7) Calibrate your sensor and save the calibration files.

IMPORTANT:

- Multiple Kinect can be used in the same computer, if they are connected at different USB buses.
- Two Kinects One have to be connected at different computers. Kinect One publishes a huge amount of data and one computer cannot be handling two of them simultaneously.

Launch in a terminal the ROS nodes responsible to interpret drivers data and publish it in ROS topics.

- roslaunch camera_sens multikinn.launch

Check in other terminal if the topics have been published:

- rostopic list

Create your input parameters files:

- /../catkin_ws/src/camera_sens /txt/**pattern1_pose.txt**
  - PATTERN_TYPE: ASYMMETRIC_CIRCLES_GRID

- PATTERN_SIZE: 4 11
- SQUARE_SIZE: 0.0175
- PATTERN_POSE: 0.0 0.0 0.25 0 0 1 0

- /../catkin_ws/src/manager/txt/**task.txt**
  - OBJ_SEARCH: Single
  - MODEL: /../catkin_ws/src/camera_sens/models_dataset/object.pcd
  - COLORS: 320 100
  - WORKSPACE: 0.4 0.4 0.5

Launch the Object Recognition module in a different terminal:

- roslaunch camera_sens camera_sens.launch

Launch the Manager module in a different terminal:

- roslaunch manager manager.launch

Planer solution has been generated. In order to visualize it, next steps have to be done:

- cd ~/catkin_ws/devel/lib/Kautham
- ./Kautham_Viewer

Then, run in other terminal:

- cd ~/catkin_ws/devel/lib/kth_simulator
- ./kth_simulator_node ObjectName

Bibliography

[1] B. Siciliano, L. Sciavicco, L. Villani and G. Oriolo, Robotics: Modelling, Planning and Control (Advanced Textbooks in Control and Signal Processing), London: Springer; pp. 1- 140, 2009.

[2] J. Lenarcic, “An efficient numerical approach for calculating the inverse kinematics for robot manipulators,” Robotica, vol. 3, no 01, p. 21-26, 1984.

[3] W. R. Hamilton, “On quaternions, or on a new system of imaginaries in algebra,” Philosophical Magazine. Vol. 25, 1844.

[4] J. Denavit and R. S. Hartenberg, “A kinematic notation for lower-pair mechanisms based on matrices,” Trans ASME J. Appl. Mech, 1955.

[5] C. John J, Introduction to Robotics, Pearson Prentice Hall; pp. 1 - 134, Third Edition.

[6] D. L. Pieper, “The Kinematics of Manipulators under Computer Control,” (No. CS-116). Stanford Univ Ca Dept Of Computer Science, 1968.

[7] S. R. Buss, “Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods,” IEEE Journal of Robotics and Automation, 17(1-19), 16., 2004.

[8] A. Balestrino, G. de Maria and L. Sciavicco, “Robust control of robotic manipulators,” 9th IFAC World Congress.

[9] C. W. Wampler, “Manipulator inverse kinematics solutions based on vector formulations and damped least squares methods,” IEEE Transactions on Systems, Man, and Cybernetics, 16(1), 93-101., 1986.

[10] Y. Nakamura and H. Hanafusa, “Inverse kinematics solutions with singularity robustness for robot manipulator control,” Journal of dynamic systems, measurement, and control, 108(3), 163-171, 1986.

[11] A. S. Deo and I. D. Walker, “Overview of Damped Least-Squares Methods for Inverse Kinematics of Robot Manipulators,” Journal of Intelligent and Robotic Systems, 14(1), 43-68, 1995.

[12] A. S. Deo and I. D. Walker, “Robot subtask performance with singularity robustness

using optimal damped least squares," In Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on (pp. 434-441). IEEE., 1992.

[13] "ROS.org rosparam," [Online]. Available: http://wiki.ros.org/rosparam. [Accessed June 2016].

[14] "ROS.org robot_state_publisher," [Online]. Available: http://wiki.ros.org/robot_state_publisher. [Accessed June 2016].

[15] "ROS.org rqt_reconfigure," [Online]. Available: http://wiki.ros.org/rqt_reconfigure. [Accessed June 2016].

[16] "ROS.org rviz," [Online]. Available: http://wiki.ros.org/rviz. [Accessed June 2016].