

# Visual Servoing for Power Grid Interconnection

Guilherme Aramizo Ribeiro

December 26, 2015

## Todo list

something . . . . . 1

## Abstract

...

## 1 Introduction

### 1.1 Objective

Project's goal

something

Requirements and Constraints

### 1.2 Hardware Setup

- USB camera (ELP 2.1mm 120 def Field of View USB camera);
- Trossen WidowX Robotic Arm;
- Male connector compatible to gripper;
- Female connector with visual markers.

## 2 Background

This section provides a brief overview of subjects related to visual servoing for the power grid project. Initially it describes the topic of robotic manipulation, which involves inverse kinematics solution and the command

interface to Trossen WidowX manipulator. Later it explores computer vision, like camera calibration, image acquisition parameters and tracking algorithms. Finally the visual servoing theory is reviewed, combining the two fields.

## 2.1 Robotic Manipulator

Serial robotic manipulators are systems designed to move a tool of interest (or end-effector) arbitrarily in space. And this system is a chain of segments connected through actuated joints, being generally revolute joints. The present section will define the kinematic description of a serial manipulator and explain a method to solve the inverse kinematics problem.



Figure 1: Coordinate frame definition on a serial manipulator

When selecting a robot to a certain task, three parameters are generally observed: the number of degrees of freedom (DOF), its workspace and the maximum payload. The DOF is the number of active joints there is in the kinematic chain. If all joints act the end-effector motion independently, the DOF is simply the number of end-effector mobility. The workspace, which is the space through which the tool can translate to. Finally, the payload is the maximum weight is can manipulate with the end-effector. It is usually represented as the weight at worst case scenario position or given as a curve with weight vs distance to the base.

The Inverse Kinematics (IK) Problem is the calculation of the joint angles to move the end-effector in a certain position and orientation. The basic process for this solution is

1. Define a coordinate frame  $F_i$  in each robot segment;
2. Calculate the rotation matrix and translation from frame  $F_i$  to frame  $F_{i+1}$  as a function of joint angle  $q_i$ ;

3. Represent the end-effector position and orientation in the base frame  $F_0$ ;
4. Solve the nonlinear system of equations for the joint angles, given the end-effector coordinates.

For the kinematic description of the robot, this paper defines a vector defined in a coordinate frame  $F_i$  as  ${}^i\mathbf{r}$ , the orientation and translation of frame  $F_i$  in respect to frame  $F_j$  as  ${}^j\mathbf{R}_i$  and  ${}^j\mathbf{t}_i$ , respectively. And the composition of both the translation and rotation as the 4D transformation matrix  ${}^j\mathbf{T}_i$  such that

$${}^j\mathbf{r} = {}^j\mathbf{R}_i {}^i\mathbf{r} + {}^j\mathbf{t}_i \quad (1)$$

Equation 1 can be represented as a single matrix multiplication if the vector is augmented by a unitary element.

$$\begin{pmatrix} {}^j\mathbf{r} \\ 1 \end{pmatrix} = \left( \begin{array}{c|c} {}^j\mathbf{R}_i & {}^j\mathbf{t}_i \\ \hline 0 & 1 \end{array} \right) \begin{pmatrix} {}^i\mathbf{r} \\ 1 \end{pmatrix} \quad (2)$$

$${}^j\bar{\mathbf{r}} = {}^j\mathbf{T}_i {}^i\bar{\mathbf{r}} \quad (3)$$

For the kinematic representation of a  $n$  DOF robot, it's convenient to define a coordinate frame  $F_i$  to the axis of rotation of each of the  $i^{th}$  joint, from the base to the end-effector. Therefore, a vector in the end-effector frame  $F_n$  is represented on the base frame  $F_0$  as

$${}^j\bar{\mathbf{r}} = ({}^0\mathbf{T}_1 {}^1\mathbf{T}_2 \dots {}^{n-1}\mathbf{T}_n) {}^n\bar{\mathbf{r}} \quad (4)$$

$$= {}^0\mathbf{T}_n {}^n\bar{\mathbf{r}} \quad (5)$$

The matrix  ${}^0\mathbf{T}_n$  represents the pose of the end-effector in respect to the base frame as a function of the joint angles. If an Euler angle rotation of Z-Y-X ( $\psi, \theta, \phi$ ) is considered, the Cartesian position and orientation are extracted from the transformation matrix as

$$x = {}^0\mathbf{T}_n(1, 4) \quad (6)$$

$$y = {}^0\mathbf{T}_n(2, 4) \quad (7)$$

$$z = {}^0\mathbf{T}_n(3, 4) \quad (8)$$

$$\psi = \text{atan2}({}^0\mathbf{T}_n(2, 1), {}^0\mathbf{T}_n(1, 1)) \quad (9)$$

$$\theta = \text{asin}(-{}^0\mathbf{T}_n(3, 1)) \quad (10)$$

$$\phi = \text{atan2}({}^0\mathbf{T}_n(3, 2), {}^0\mathbf{T}_n(3, 3)) \quad (11)$$

Once the Forward Kinematics Equations are defined, the Inverse Kinematics Problem is solved by numerically computing the joint angles for a desired end-effector coordinates. Numerical solvers like Python Scipy's **Optimization** package or MATLAB's **fsolve** function can be used in this stage.

If the robot has a mobility less than 6, only a subset of the equations can be solved. The selection of equations depends on the particularities of the robot construction.

Another interesting derivation of the Forward Kinematics Equations is the Jacobian matrix, that relates the rate of change of the end-effector coordinates and the joint angles. If  $f_i$  is the  $i_{th}$  end-effector coordinate and  $q_j$  is the  $j_{th}$  joint angle, the Jacobian is calculated as

$$\mathbf{J} = \frac{d\mathbf{f}}{d\mathbf{q}} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial q_1} & \dots & \frac{\partial \mathbf{f}}{\partial q_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \dots & \frac{\partial f_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial q_1} & \dots & \frac{\partial f_m}{\partial q_n} \end{bmatrix} \quad (12)$$

## 2.2 Computer Vision

Computer vision uses image processing to reproduce the human ability of vision. The performance of these algorithms depends on a good camera-lenses selection, camera settings and calibration procedures. This section will give an overview of these subjects.

Optimizing Image Quality: frame rate, auto-exposition time, bright/-contrast etc

Calibration

Lens distortion on wide FOV cameras

## 2.3 Visual Servoing

Visual Servoing is the use of computer vision as a feedback to a manipulator action. This section describes the most common control schemes, the camera placing architecture and options of tracked image features.

Control schemes:

Camera

Trajectory planning. Keeping target in field of view (FOV)

Camera placement, perspective error (Gripper to camera transformation estimation algorithm) ———- Extrinsic camera calibration (hand-eye calibration)

## 3 Methodology

This section describes the necessary steps to deploy the visual servoing system. It includes the initial setup procedures, like source dependencies installation, creation of configuration files and camera calibration. Following, it describes the control algorithm and a performance test.



Figure 2: Diagram of program architecture, including the data flow and configuration files' dependencies

### 3.0.1 Dependencies Installation

The visual servoing system relies on a ROS Indigo Installation and several ROS packages and libraries.

For the robotic control and interface it uses the Python numerical library Scipy for the IK calculations, a Python binding for the Kinematics and Dynamics Library (KDL) to perform the Forward Kinematics, the U-Robot Description Format ROS package to import a robot description file into KDL and the Arbotix ROS package to control the Dynamixel servomotors of WidowX.

The computer vision component requires the USB CAM ROS package for camera interface and a ROS binding for the Alvar Augmented Reality Library.

The following script carries out the installation of all the dependencies. It should be executed in a desktop running Ubuntu 14.04 LTS with a ROS Indigo installation.

```
# installation

sudo apt-get install ros-indigo-ar-track-alvar
sudo apt-get install python-numpy python-scipy python-
    ↪ matplotlib ipython ipython-notebook python-
    ↪ pandas python-sympy python-nose
cd ~/catkin_ws/src
git clone https://github.com/bosch-ros-pkg/usb_cam.git
git clone -b kdl_parser_py https://github.com/ros/
    ↪ robot_model.git
git clone https://github.com/vanadiumlabs/arbotix_ros.
    ↪ git
git clone https://github.com/ros/urdfdom.git
wget https://raw.githubusercontent.com/ros-gbp/urdfdom-release/
    ↪ debian/indigo/trusty/urdfdom/package.xml
devel_prefix=$(cd $(catkin_find --first-only)/.. &&
    ↪ pwd)
cd ../urdf_parser_py
python setup.py install --install-layout deb --prefix
    ↪ $devel_prefix

# intrinsic calibration

roslaunch camera_calibration cameracalibrator.py --size
    ↪ 11x7 --square 0.02 image:=/usb_cam/image_raw
    ↪ camera:=/usb_cam
mv ost.txt ost.ini
roslaunch camera_calibration_parsers convert ost.ini cal.
    ↪ yml
```

### 3.1 Configuring the Robotic Arm Description and Interface

The manipulator description and interface are defined by two different configuration files, the Unified Robot Description Format (URDF) and YAML file defining the Dynamixel servomotor network.

The arm description file ...

The manipulator interface file ...

### 3.2 Configuring the Vision System

The vision system requires the intrinsic and extrinsic calibration files, and the visual target description. The intrinsic calibration file defines the internal properties of the camera-lenses combination, while the extrinsic calibration file defines the position and orientation of the camera in respect to the robotic arm. Also, the target is described by visual cues using a XML file.

The camera calibration ...

The AR Alvar library ...

### 3.3 Deployment

### 3.4 Algorithm

State1: Look around until target is found.  $Cam^W(yaw) = (ang * sin(alf * time))$

State2: centralize target.  $C(x, y) = (- > 0, - > 0)$ .

State3: approximate to target.  $Target - Cam^C(x, y, z) = (0, 0, - > z0)$ .

State3: orbit around target.  $Target^T(pitch, yaw, dist) = (- > 0, - > 0, d0)$

State5: connect  $^T(pitch, yaw, dist) = (0, 0, - > 0)$

Pseudo-code of main ROS node (visser.py)

### 3.5 Performance Test

Describe reachable workspace (attitude included) and speed

## 4 Conclusion

Verify if initial workspace requirements are met

### 4.1 Development Timeline

Table with dated problems encountered and proposed solutions

## References

- [1] *How to Calibrate a Monocular Camera*. [http://wiki.ros.org/camera\\_calibration/Tutorials/MonocularCalibration](http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration). [Online; accessed 7-December-2015]. 2015.
- [2] *Hand to Eye Calibration*. [http://wiki.ros.org/visp\\_hand2eye\\_calibration/Electric](http://wiki.ros.org/visp_hand2eye_calibration/Electric). [Online; accessed 7-December-2015]. 2015.



# Appendices

## A Intrinsic Camera Calibration

Latest version available at [1].

**Note:** This tutorial assumes that you have completed the previous tutorials: ROS Tutorials (/ROS/Tutorials).

💡 It is appreciated that problems/questions regarding this tutorial are asked on [answers.ros.org](http://answers.ros.org) (<http://answers.ros.org>). Don't forget to include in your question the link to this page, versions of your OS & ROS, and also add appropriate tags.

# 1. How to Calibrate a Monocular Camera

**Description:** This tutorial cover using the camera\_calibration (/camera\_calibration)'s cameracalibrator.py node to calibrate a monocular camera with a raw image over ROS.

**Keywords:** monocular, camera, calibrate


**Tutorial Level:** BEGINNER

## Conteúdo

1. Before Starting
2. Compiling
3. Camera Publishing
4. Running the Calibration Node
  1. Dual Checkerboards
5. Moving the Checkerboard
6. Calibration Results
7. Creating a yml file
8. Rectifying an image

## 1.1 Before Starting

Make sure that you have the following:

- a large  checkerboard ([/camera\\_calibration/Tutorials/MonocularCalibration?action=AttachFile&do=view&target=check-108.pdf](/camera_calibration/Tutorials/MonocularCalibration?action=AttachFile&do=view&target=check-108.pdf)) with known dimensions. This tutorial uses a 8x6 checkerboard with 108mm squares. Calibration uses the interior vertex points of the checkerboard, so an "8x6" board is nine squares wide and seven high, like the example below.
- a well lit 5m x 5m area clear of obstructions and check board patterns
- a monocular camera publishing images over ROS

## 1.2 Compiling

Start by getting the dependencies and compiling the driver.

```
$ rosdep install camera_calibration  
$ rosmake camera_calibration
```

## 1.3 Camera Publishing

Make sure that your monocular camera is publishing images over ROS. Let's list the topics to check that the images are published:

```
$ rostopic list
```

This will show you all the topics published, check to see that there is an `image_raw` topic. The default topics provided by most ROS camera drivers are:

```
/camera/camera_info  
/camera/image_raw
```

If you have multiple cameras or are running the driver in its own namespace, your topic names may differ.

## 1.4 Running the Calibration Node

To start the calibration you will need to load the image topics that will be calibrated:

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 i  
mage:=/camera/image_raw camera:=/camera
```

This will open up the calibration window which will highlight the checkerboard:



If it does not open up the window try the following parameter:

```
--no-service-check
```

If you can't see any colored dots make sure you count the interior vertex points, not the squares!

### 1.4.1 Dual Checkerboards

#### New in D

Starting in Diamondback, you will be able to use multiple size checkerboards to calibrate a camera.

To use multiple checkerboards, give multiple `--size` and `--square` options for additional boards. Make sure the boards have different dimensions, so the calibration system can tell them apart.

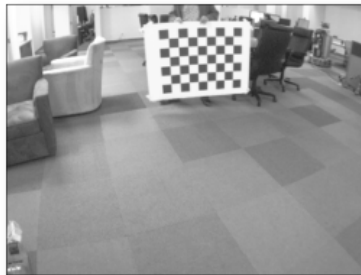
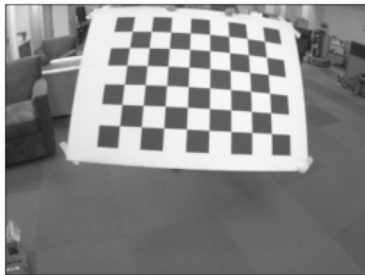
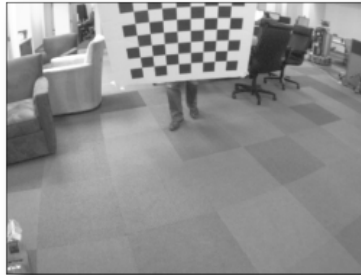
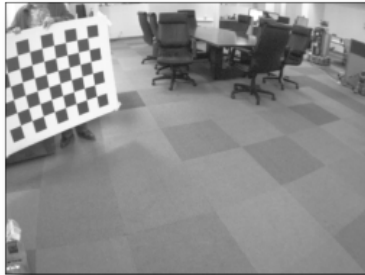
## 1.5 Moving the Checkerboard

In order to get a good calibration you will need to move the checkerboard around in the camera frame such that:

- checkerboard on the camera's left, right, top and bottom of field of view
  - X bar - left/right in field of view

- Y bar - top/bottom in field of view
- Size bar - toward/away and tilt from the camera
- checkerboard filling the whole field of view
- checkerboard tilted to the left, right, top and bottom (Skew)

At each step, hold the checkerboard still until the image is highlighted in the calibration window.



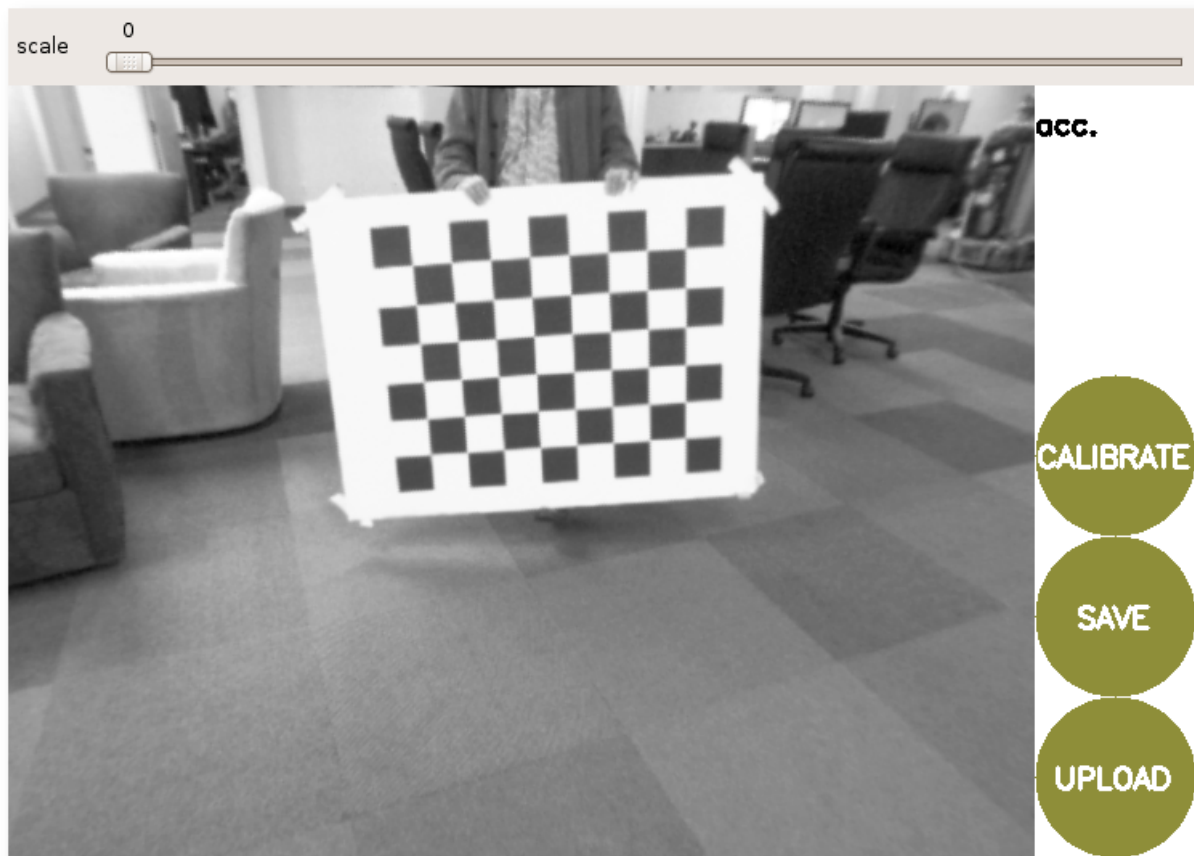
As you move the checkerboard around you will see three bars on the calibration sidebar increase in length. When the **CALIBRATE** button lights, you have enough data for calibration and can click **CALIBRATE** to see the results.

Calibration can take about a minute. The windows might be greyed out but just wait, it is working.



## 1.6 Calibration Results

After the calibration is complete you will see the calibration results in the terminal and the calibrated image in the calibration window:



A successful calibration will result in real-world straight edges appearing straight in the corrected image.

A failed calibration usually results in blank or unrecognizable images, or images that do not preserve straight edges.

After a successful calibration, you can use the slider at the top of the calibration window to change the size of the rectified image. A scale of 0.0 means that the image is sized so that all pixels in the rectified image are valid. The rectified image has no border, but some pixels from the original image are discarded. A scale of 1.0 means that all pixels in the original image are visible, but the rectified image has black borders where there are no input pixels in the original image.

```

D = [-0.33758562758914146, 0.11161239414304096, -0.00021819272592442094,
-3.029195446330518e-05]
K = [430.21554970319971, 0.0, 306.6913434743704, 0.0, 430.53169252696676,
227.22480030078816, 0.0, 0.0, 1.0]
R = [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P = [1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
# oST version 5.0 parameters

[image]

width
640

height
480

[narrow_stereo/left]

camera matrix
430.215550 0.000000 306.691343
0.000000 430.531693 227.224800
0.000000 0.000000 1.000000

distortion
-0.337586 0.111612 -0.000218 -0.000030 0.0000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000

```

If you are satisfied with the calibration, click **COMMIT** to send the calibration parameters to the camera for permanent storage. The GUI exits and you should see "writing calibration data to ..." in the console.

## 1.7 Creating a yml file

The Camera Calibration Parser (/camera\_calibration\_parsers) helps you to create a yml file, which you can load with nearly all ros camera driver using the *camera\_info\_url* parameter.



## 1.8 Rectifying an image

Simply loading a calibration file does not rectify the image. For rectification, use the `image_proc` package (`/image_proc`).

Except

where

Wiki: [camera\\_calibration/Tutorials/MonocularCalibration](http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration) (última edição 2015-03-23 21:04:57 efectuada por bradknox (/bradknox))

otherwise

noted, the ROS wiki is licensed under the

Creative Commons Attribution 3.0 (<http://creativecommons.org/licenses/by/3.0/>) | Find us on

Google+ (<https://plus.google.com/113789706402978299308>)

Brought to you by:  Open Source Robotics Foundation

(<http://www.osrfoundation.org>)

## **B Extrinsic Camera Calibration**

Latest version available at [2].

electric	fuerte	groovy	hydro	indigo	jade	Documentation Status
----------	--------	--------	-------	--------	------	----------------------

**vision\_visp** (/vision\_visp?distro=jade): *visp\_auto\_tracker* (/visp\_auto\_tracker?distro=jade) | *visp\_bridge* (/visp\_bridge?distro=jade) | *visp\_camera\_calibration* (/visp\_camera\_calibration?distro=jade) | *visp\_hand2eye\_calibration* (/visp\_hand2eye\_calibration?distro=jade) | *visp\_tracker* (/visp\_tracker?distro=jade)

#### Package Links

- **Code API** ([http://docs.ros.org/jade/api/visp\\_hand2eye\\_calibration/html](http://docs.ros.org/jade/api/visp_hand2eye_calibration/html))
- **Msg/Srv API** ([http://docs.ros.org/jade/api/visp\\_hand2eye\\_calibration/html/index-msg.html](http://docs.ros.org/jade/api/visp_hand2eye_calibration/html/index-msg.html))
- **FAQ** ([http://answers.ros.org/questions/scope:all/sort:activity-desc/tags:visp\\_hand2eye\\_calibration/page:1/](http://answers.ros.org/questions/scope:all/sort:activity-desc/tags:visp_hand2eye_calibration/page:1/))
- **Changelog**  
([http://docs.ros.org/jade/changelogs/visp\\_hand2eye\\_calibration/changelog.html](http://docs.ros.org/jade/changelogs/visp_hand2eye_calibration/changelog.html))
- **Change List** (/vision\_visp/ChangeList)
- **Reviews** (/visp\_hand2eye\_calibration/Reviews)

#### Dependencies (10)

#### Used by (1)

#### Jenkins jobs (9)

## Package Summary

✓ Released   ✓ Continuous integration   ✓ Documented

*visp\_hand2eye\_calibration* estimates the camera position with respect to its effector using the ViSP library.

- Maintainer status: maintained
- Maintainer: Fabien Spindler <Fabien.Spindler AT inria DOT fr>
- Author: Filip Novotny
- License: GPLv2
- Source: git [https://github.com/lagadic/vision\\_visp.git](https://github.com/lagadic/vision_visp.git) ([https://github.com/lagadic/vision\\_visp](https://github.com/lagadic/vision_visp)) (branch: jade)

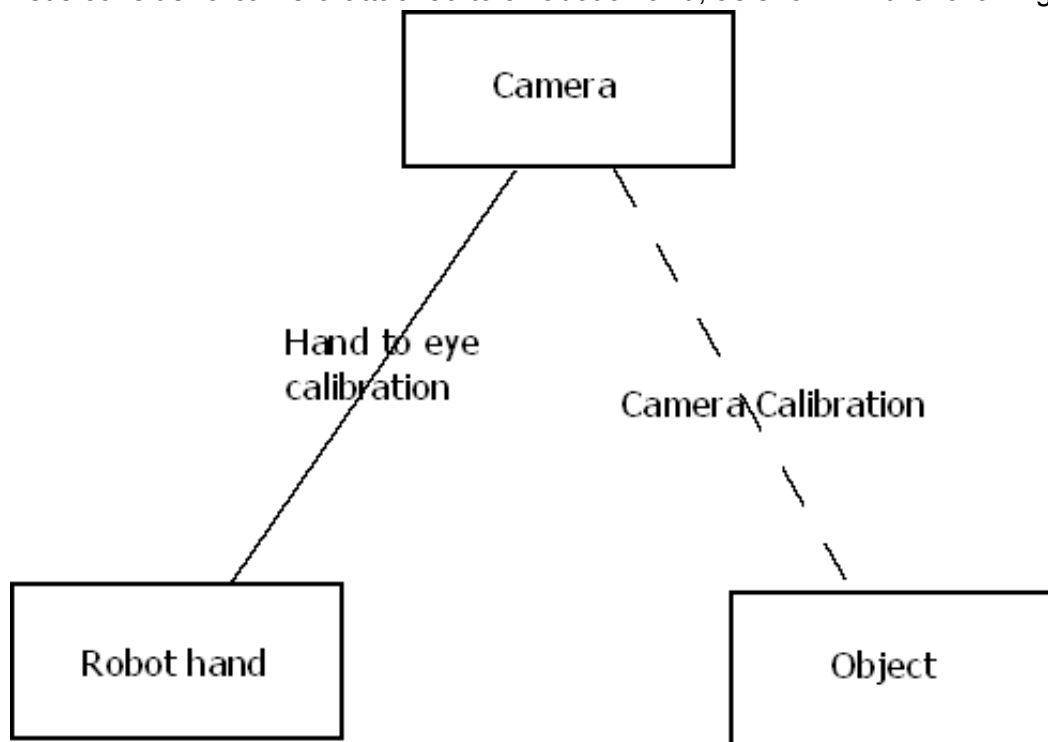
## Conteúdo

1. Overview
2. Usage
3. Example
4. Nodes
  1. calibrator
    1. Subscribed Topics
    2. Services Called

# 1. Overview

visp\_hand2eye\_calibration is a ROS package that computes extrinsic camera parameters : the constant transformation from the hand to the camera coordinates.

Let's consider a camera attached to a robotic hand, as shown in the following diagram:



The interest of this package is to perform the hand to eye calibration part.

To do so, two sets of transformations are fed to the `calibrator` node.

- a list of camera-to-object transformations
- a list of world to hand transformations

Once the node has enough transformations, it can estimate the camera-to-hand transformation which is the goal of the calibration.

The package consists of a `calibrator` node and an experimental client doing a sample calibration from a few poses.

The experimental client is merely there for a quick demonstration of the package abilities.

## 2. Usage

To run the calibrator node:

```
roslaunch visp_hand2eye_calibration visp_hand2eye_calibration_calibrator
```

## 3. Example

An example client is shipped with this package. It feeds different transformations (camera->object and world->hand) to the calibrator which computes the relative transformation between the camera and the hand.

To try this example, run roscore first.

```
roslaunch visp_hand2eye_calibration visp_hand2eye_calibration_client
```

Then, run the client:

```
roslaunch visp_hand2eye_calibration visp_hand2eye_calibration_client
```

Finally, run the calibrator:

```
roslaunch visp_hand2eye_calibration visp_hand2eye_calibration_calibrator
```

The output displayed in the terminal running your client should look like this:

```
[ INFO] [1329226083.184531090]: Waiting for topics...
[ INFO] [1329226084.186233704]: 1) GROUND TRUTH:
[ INFO] [1329226084.186327570]: hand to eye transformation:
translation:
  x: 0.1
  y: 0.2
  z: 0.3
rotation:
  x: 0.96875
  y: 0.0863555
  z: -0.0863555
  w: 0.215889

[ INFO] [1329226085.186682976]: 2) QUICK SERVICE:
[ INFO] [1329226085.188853282]: hand_camera:
translation:
  x: 0.1
  y: 0.2
  z: 0.3
rotation:
  x: 0.96875
  y: 0.0863555
  z: -0.0863555
  w: 0.215889

[ INFO] [1329226085.188915366]: 3) TOPIC STREAM:
[ INFO] [1329226085.190303537]: hand_camera:
translation:
  x: 0.1
  y: 0.2
  z: 0.3
rotation:
  x: 0.96875
  y: 0.0863555
  z: -0.0863555
  w: 0.215889
```

## 4. Nodes

### 4.1 calibrator

#### 4.1.1 Subscribed Topics

world\_effector (visp\_hand2eye\_calibration/TransformArray

([http://docs.ros.org/api/visp\\_hand2eye\\_calibration/html/msg/TransformArray.html](http://docs.ros.org/api/visp_hand2eye_calibration/html/msg/TransformArray.html)))  
transformation between the world and the hand frame. The node expects to receive several of those transformations.

camera\_object (visp\_hand2eye\_calibration/TransformArray  
([http://docs.ros.org/api/visp\\_hand2eye\\_calibration/html/msg/TransformArray.html](http://docs.ros.org/api/visp_hand2eye_calibration/html/msg/TransformArray.html)))  
transformation between the camera and the object frame. The node expects to receive several of those transformations.

## 4.1.2 Services Called

compute\_effector\_camera (visp\_hand2eye\_calibration/compute\_effector\_camera  
([http://docs.ros.org/api/visp\\_hand2eye\\_calibration/html/srv/compute\\_effector\\_camera.html](http://docs.ros.org/api/visp_hand2eye_calibration/html/srv/compute_effector_camera.html)))  
Returns the hand to camera transformation result after calibration based on the data published on the subscribed topics.

compute\_effector\_camera\_quick (visp\_hand2eye\_calibration/compute\_effector\_camera\_quick  
([http://docs.ros.org/api/visp\\_hand2eye\\_calibration/html/srv/compute\\_effector\\_camera\\_quick.html](http://docs.ros.org/api/visp_hand2eye_calibration/html/srv/compute_effector_camera_quick.html)))  
Returns the hand to camera transformation result after calibration. This service is not based on the data published on the subscribed topics. Instead, the transformation are passed as service parameters.

reset (visp\_hand2eye\_calibration/reset  
([http://docs.ros.org/api/visp\\_hand2eye\\_calibration/html/srv/reset.html](http://docs.ros.org/api/visp_hand2eye_calibration/html/srv/reset.html)))  
Reset all the data published on the subscribed topics. Only new publications (transformations) will be taken into account.

Except where

otherwise noted, the

ROS wiki is licensed under the

Creative Commons Attribution 3.0 (<http://creativecommons.org/licenses/by/3.0/>) | Find us on  
Google+ (<https://plus.google.com/113789706402978299308>)

Brought to you by:  Open Source Robotics Foundation

(<http://www.osrfoundation.org>)