Sapienza University of Rome

Robotics 2 and Medical Robotics courses

ROS/Gazebo simulations of an eye-to-hand visual servoing control scheme

Students
Marco Bellaccini
Matteo Gagliardi
Manuel Ruiz

December 2012

Contents

1 Project topic

The goal of our project is to perform eye-to-hand image-based visual servoing without violating RCM constraint. This kind of problem is typical of minimally invasive surgical robotics context: in laparoscopic surgery instruments are inserted and commanded through trocars positioned at incisions point on patient body. Usually two robot arms are used in order to accomplish the desired task: one equipped with the surgical tool and the other one with endoscopic camera. Optical markers are mounted on surgical tool in order to get information about surgical tool pose w.r.t. camera frame and perform visual servoing. Assuming that the instrument is already in the camera field of view (in the real case it must be manually positioned in such configuration) features can be extracted from the image via filtering processes; in our case we use only one marker recognized in the image basing on his color and area. The robot link has to be constrained to move through the incision point, can only translate along its axis and rotate about the incision point: this is our RCM constraint.

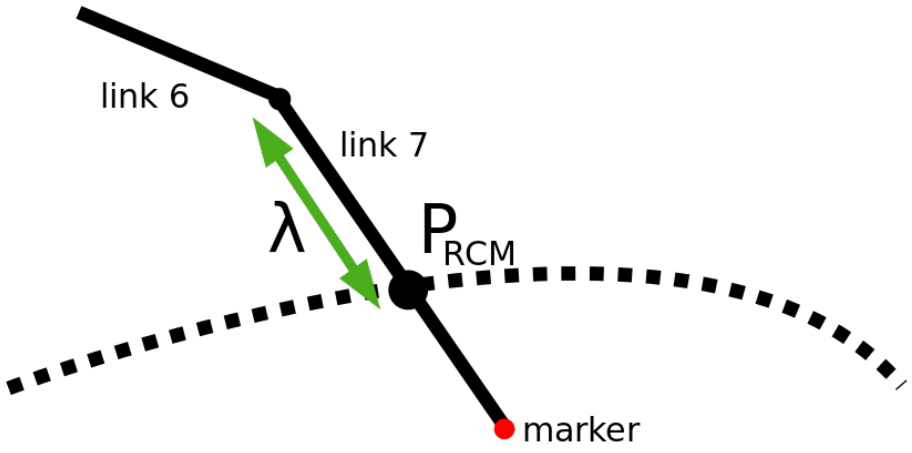

Figure 1: Schematic representation of RCM constraint and marker position

We aim to perform a simulation of a similar (but simplified) setup in ROS+Gazebo environment using a KUKA LWR robot, and a fixed camera. We will try to accomplish two main tasks and a secondary one: RCM control, marker position regulation in the image plane and, if possible, end effector penetration control.

1.1 Control law

In order to satisfy visual task, a vision based control-scheme was implemented whose aim is to minimize the error defined as:

$$e_{vis}(t) = g(t) - g \tag{1}$$

in which $g(t)$ is the vector of actual values of marker feature (that are immediately available since we are using an image-based approach) and g is the vector of desired ones.
In our case the features consist of the pixel marker's coordinates on image plane u,v obtained from perspective equations:

$$u = f\frac{X}{Z}\ v = f\frac{Y}{Z} \tag{2}$$

with $P = (X, Y, Z)$ being marker's coordinates in camera frame and f camera focal length . The

relation between features and robot's end-effector motions is modeled by the following formula:

$$\dot{g} = J_{cam}\dot{r} \tag{3}$$

with $\dot{r} = \begin{bmatrix} V_x & V_y & V_z & \Omega_x & \Omega_y & \Omega_z \end{bmatrix}^T$ end-effector velocities (with respect to camera frame), J_v is the 2x6 interaction matrix and $\dot{g} = \begin{bmatrix} \dot{u} & \dot{v} \end{bmatrix}^T$ features' velocities. Expanding the formula and applying it to our case:

$$\dot{g} = \begin{bmatrix} \frac{f}{Z} & 0 & -\frac{u}{Z} & -\frac{uv}{f} & f+\frac{u^2}{f} & -v \\ 0 & \frac{f}{Z} & -\frac{v}{Z} & -f-\frac{v^2}{f} & \frac{uv}{f} & u \end{bmatrix} \begin{bmatrix} V \\ \Omega \end{bmatrix} \tag{4}$$

Now we can compute the following formula that relates features motion and joints velocities:

$$\dot{g} = J_{cam}J\dot{q} = J_{vis}\dot{q}$$

with J being the geometric jacobian of the robot.
In order to satisfy RCM constraint we want to minimize the error defined as:

$$e(t)_{RCM} = P_{RCM}(t) - P_{TRC} \tag{5}$$

more specifically $P_{RCM}(t) = P_6(t) + \lambda(t)(P_7(t) - P_6(t))$ with P_6 and P_7 position of 6th and 7th robot frames origins and λ the value linked to penetration as shown in the previous image and P_{TRC} the trocar position. Specifically, from now on, we're considering normalized λ (i.e. λ divided by the EE length).
Differentiating eq. (6) and writing it in matrix notation:

$$\dot{P}_{RCM} = \begin{pmatrix} J_6 + \lambda(J_7 - J_6) \\ P_7 - P_6 \end{pmatrix}^T \begin{pmatrix} \dot{q} \\ \dot{\lambda} \end{pmatrix} = J_{RCM} \begin{pmatrix} \dot{q} \\ \dot{\lambda} \end{pmatrix} \tag{6}$$

Finally, using both jacobian matrix, we can write an unique control law that uses an augmented jacobian:

$$\begin{pmatrix} \dot{q} \\ \dot{\lambda} \end{pmatrix} = K \begin{pmatrix} J_{vis} \\ J_{RCM} \end{pmatrix}^{\#} \begin{pmatrix} e_{vis} \\ e_{RCM} \end{pmatrix} = K J_t^{\#} e_t \tag{7}$$

In order to control the penetration of the last link of the robot we added a secondary task to the previous control law (7):

$$u = K J_t^{\#} e_t + (I - J^{\#}J)w \tag{8}$$

with $w = -[\nabla(\frac{1}{2}(\lambda - \lambda_{des})^2)]^T = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda_{des} - \lambda \end{pmatrix}^T$.

2 Introduction to ROS

Robot Operating System (ROS) is an open-source software framework for robot software development written in C++ language that provides operating-system-like functionalities.
It was created in 2007 in support of the Stanford AI Robot project, following developments were carried on by Willow Garage[1], an american private robotics research institute; the last stable release, that we used to implement our work, is called "Fuerte"[2].
The ROS environment is designed to run on Linux systems, Windows and Mac OSX versions exists too but lack of some important functionalities.
ROS provides standard operating-system services such as:

- hardware abstraction;
- low-level device control;
- communication between processes;
- package management.

Besides the meta-operating-system side, this software offers the possibility to provide packages by which, through an architecture composed of nodes, implement functionalities.
"Packages" are the lowest level of ROS software organization and can contain anything: libraries, tools, executables ecc. The description of a package is provided by a "manifest" by which even dependencies between packages are defined. Two or more of them may compose an higher-level structure called "stack".
ROS packages executables are compiled from C++ (as in our case) or Python code using "roscpp" or "rospy" Client Libraries.
As previously mentioned, ROS implements functions through executables called "nodes" which can communicate with each other using ROS client libraries and "messages" that are simple data structures, used to subscribe or publish to a "topic", comprising primitive C types, simple or nested. Topics are like buses over which nodes exchange messages; nodes, in fact, are not aware of who they are communicating with but only which topic they have to communicate on[3]. ROS currently supports even TCP/IP-based communications.
Another feature of ROS nodes is to provide "services": the publish/subscribe communication model implemented by messages is not appropriate for many-to-many request/reply interactions so, for this purpose, a ROS node offers a service under a string name, and a client calls the service by sending the request message and waiting the reply.
Packages can be built with a specific command (rosmake) that not only allows the building of our package but also of all those it depends on.
An important ROS package is "roslaunch" that contains a series of tool which assist you in the process of launching multiple ROS nodes and setting parameters on the Parameter Server as described in special ".launch" files; "Gazebo", a physical-simulator software, is also included in the meta-operating-system as a package.

[1] http://www.willowgarage.com
[2] http://www.ros.org
[3] Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type.

3 Introduction to Gazebo simulator

Gazebo[4] is an open source realistic multi-robot simulator for both indoor and outdoor environments. From 2004 through 2011 it has been a part of the "Player/Stage Project" (a project that aims to create free software for research in the robotics field). In 2011 Gazebo became an independent project supported by Willow Garage. The core of Gazebo is the "Open Dynamics Engine"[5] (ODE), a free-software physics engine written in C++. Through ODE, Gazebo is able to simulate rigid body dynamics and collisions. Apart from this, Gazebo integrates OpenGL 3D rendering via the OGRE engine[6] (Object-Oriented Graphics Rendering Engine) and code for sensors simulation and actuators control. From the year 2009, modified versions of Gazebo have also been integrated in ROS, improving Gazebo diffusion among the robotics research community. Actually[7] the last stable stand-alone version of Gazebo is the 1.3.0 release, however the last version integrated in the stable ROS release (actually ROS Fuerte) is 1.0.2.
In our work we've been using ROS Fuerte, so physics simulations were made with the 1.0.2 version. From a practical point of view Gazebo is divided in two parts: the server (gzserver) that includes the physics engine and the client (gzclient) that implements the GUI. Both these components in the ROS version are started by ROS when requested.

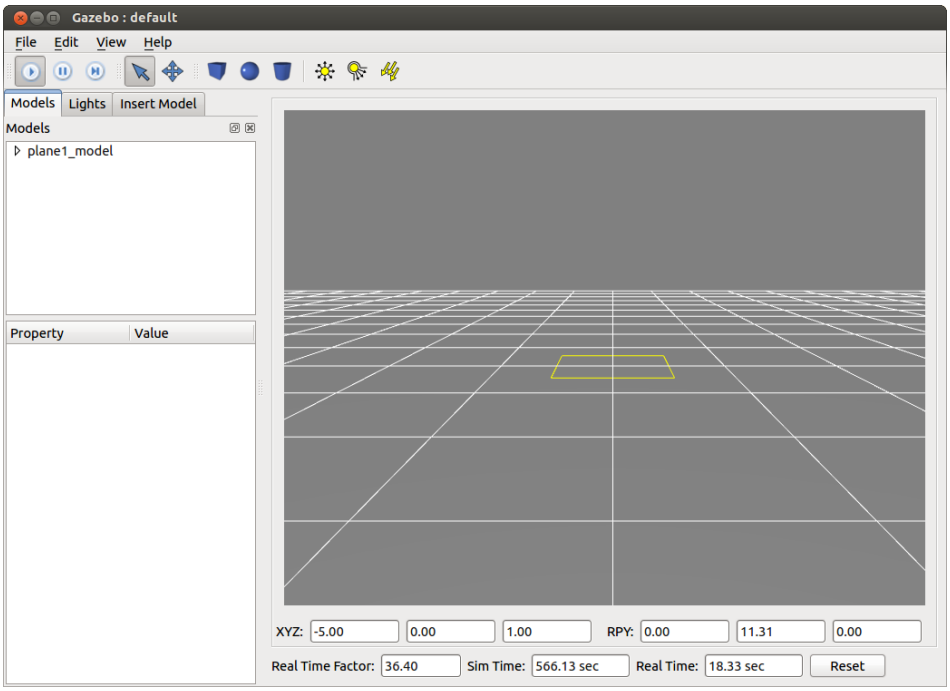

Figure 2: An empty Gazebo world running in version 1.0.2

To describe the environment of the simulation a "world" file must be written. This file is formatted in SDF[8] format and may contain references to any kind of element of the simulation (robots, lights, sensors, and static objects etc.). However objects may be "spawned" into the simulation environment even after the gzserver's boot-up via command line instructions and so, usually, only "static" elements (like ground, obstacles and lights) are described in that file. Another important thing described in the world file is the ODE configuration: this includes gravity constant definition and other parameters

[4] http://gazebosim.org
[5] http://www.ode.org
[6] http://www.ogre3d.org
[7] December 2012
[8] Simulation Description Format (a markup format based on XML 1.0)

like the step used by the numerical methods implemented in ODE to solve differential equations. The "objects" themselves are usually described in "model files". These are formatted in SDF or URDF (Unified Robot Description Format) format and they contain explicit informations about the described object: pose w.r.t the Gazebo reference frame, mass, inertia tensor as well as color and geometry of the object. In the ROS version of Gazebo, model files may also be integrated with XACRO macros to make them shorter and slightly more readable. The object's geometry may be described explicitly for simple shapes (i.e. we may describe a cube shape by writing its side's length or a sphere shape by writing its radius) or with a reference to a mesh file for complex shapes. In our work almost all the object's meshes are described in separate STL (STereoLithography) files, editable with 3D graphics programs like Blender[9].
Gazebo supports plugins: you can interact with any active component of the simulation through plugins. There're four types of plugins: "world" plugins, "model" plugins, "sensor" plugins ans "system" plugins. Plugins interact with Gazebo through APIs (Application programming interfaces), whose documentation can be found on Gazebo's website[10]. Plugins are compiled as separate ".so" files (the Linux equivalent of Microsoft Windows dll files - i.e.: dynamic link libraries).
In our work we've used a model plugin to interact with the robot and a sensor plugin to interface with the camera.

[9] http://www.blender.org
[10] http://gazebosim.org/api

4 Image processing

4.1 Visual sensor

One or more sensors can be defined for each link inside a model, using either Gazebo Simulation Description Format (SDF) or ROS Universal Robot Description Format (URDF). In particular, the camera sensor allows to obtain a stream of image frames at simulation rate, from an arbitrary viewpoint and viewport of the environment or world and arbitrary focal length, defined intrinsically with the horizontal field of view of the camera[11]. Using the SDF, the camera is modeled using the simple *pinhole camera model* and the image data is available inside the simulator node via the Gazebo *CameraSensor* Plugin API facilities. For real applications this limitation can be easily removed using a ROS camera driver together with the image pre-processing node *image_proc* via the URDF definition, which handles all the underlaying steps and abstraction layers between the simulated camera and the image data stream, available as a ROS topic, allowing the implementation of the image processing as a separate node. In our early work we used the URDF alternative, but changed during the develop because the multiple stages of image processing and data formating conversion, in particular because the conversion between ROS *Image Messages* and the computer vision library image type, and to avoid the exchange of tons of irrelevant data: the marker detection is done inside the sensor handler, so the data published is limited to the image features.
It is important to note that the developed sensor plugin not only hold the features extraction process but also the interactive target positioning interface, thats because the window showing the captured data is also handled by the computer vision library, given more flexibility and portability of the code.

4.2 Features extraction

The visual servoing is based on the visual error between the target coordinates and the estimated coordinates of the robot end-effector inside the image plane. To this end an optical marked is placed at the end of the end-effector link, consisting on a little red colored portion of the end-effector tool. The marker detection is realized in two image processing stages: the first segments the whole image frame based on a color matching technique, while the second extracts from the resulting masked[12] image the marker position by a boundary-based segmentation. All image processing is done with the computer vision library OpenCV[13].
First stage takes the advantage of the color space HSV of the RGB color model, in which the image intensity is taken apart from the color information, to apply robust color based segmentation to lighting changes. The masked image is then morphologically filtered to join nearly disconnected areas and finally a noise filter is applied to reduce the probability of false positives in the second and final stage, in which the marker contour is computed after an edge detection procedure. From the contour information the optical marker coordinates can be easily computed as the *centroid*[14] of the image region inside the contour, based on the region moments.

[11] Remeber that field of view α and focal length ρ are related by the equation $\rho = \frac{d}{2}\tan\left(\frac{\alpha}{2}\right)$ where d represents the size of the image plane in the direction measured.

[12] Color based segmentation gives a binary image, called *mask*, in which colored matched regions are highlighted respect to the rest of the image, usually using a white color against a black background.

[13] For more information about OpenCV consult the web site: http://www.opencv.org

[14] Suppose u and v are the image coordinates of a pixel inside an image region $\mathcal{R}$ of a binary image, then the moment $m_{i,j}$ with $i, j = 1, 2, ...$ is:

$$m_{i,j} = \sum_{u,v \in \mathcal{R}} u^i v^j$$

The centroid of the region is thus given by coordinates $\bar{u} = \frac{m_{1,0}}{m_{0,0}}$ $\bar{v} = \frac{m_{0,1}}{m_{0,0}}$

5 Migration to the new Gazebo APIs

The starting point of our work was a model of the KUKA LWR robot written for an old Gazebo release that used Gazebo APIs in pre-1.0 version. This model was incompatible with Gazebo 1.0.2 (based on APIs 1.0) and so a conversion of that model was needed. This was done basing on some plugins examples and on the guidelines found on Gazebo's website and reported in the table below:

Item	Old plugin API	New plugin API
Includes	<gazebo/Controller.hh>	All replaced by "gazebo.h"
	<gazebo/ControllerFactory.hh>	
	<gazebo/Param.hh>	
Class Inheritance	All plugins derives from gazebo::Controller	Sensor plugins derive from gazebo::SensorPlugin
		Model plugins derives from gazebo::ModelPlugin
		World plugins derive from gazebo::WorldPlugin
		System plugins derive from gazebo::SystemPlugin
Constructor	Constructor is called with an Entity* parent pointer	No arguments for the constructor. Pointer to the parent object is provided during loading.
Loading	LoadChild(XMLConfigNode *node)	Load(gazebo::SensorPtr _parent, sdf::ElementPtr _sdf)
	Plugin parameters are accessed through the XMLConfigNode pointer.	Pointer to parent entity and to the plugin parameters.
Initialization	InitChild()	Init()
Unloading	FiniChild()	N/A: Move your code into the destructor.
Parsing Parameters	gazebo/Param.hh	sdf/interface/Param.hh
		sdf/interface/SDF.hh
Plugin Registration Macros	GZ_REGISTER_DYNAMIC_ CONTROLLER(std::string plugin_name, [Class name])	GZ_REGISTER_SENSOR_ PLUGIN([class name])
		GZ_REGISTER_MODEL_ PLUGIN([class name])
		GZ_REGISTER_WORLD_ PLUGIN([class name])
		GZ_REGISTER_SYSTEM_ PLUGIN([class name])

Table 1: Summary of the main differences between Gazebo's pre-1.0 APIs and 1.0 APIs

6 Package structure

In this section we are going to describe the structure of our ROS package. In the root directory of package's tree we can find some important files:

- Manifest.xml: lists package's dependencies and contains library paths indications.
- CMakeLists.txt: provides instructions to "cmake", a cross-platform Makefile generator, utilized by "rosmake" in order to build the package.
- run.sh, Plot.sh: shell scripts to automatically run simulation and generate dynamic plots.

Apart from files there are also some directories:

- bin: controller executable files will be put here.
- build: contains compilation auto-generated files.
- include: contains header files for both model plugin and camera plugin.
- launch: contains .launch files.
- lib: contains compiled libraries.
- Media: contains materials description (mainly the colors) of the model and of other objects.
- models: contains the "world" file, the urdf/xacro descriptions of the models and a sub-directory containing the "stl" meshes.
- msg: contains package-specific message types.
- msg_gen: contains message compilation auto-generated files.
- src: contains source files of the model Gazebo plugin, the optical-marker plugin and the controller.

7 Controller code

Our controller is implemented in file "vis_controller.cpp": in that file we initialize a new ROS node called "controller" .

ros::init(argc, argv, "controller");

At the beginning of the file we include some header files: ROS libraries, Eigen libraries, "math" libraries, messages definitions and Boost multithread libraries[15].

```
#include "ros/ros.h"
#include <eigen3/Eigen/Dense>
#include <math.h>
#include "lwrSimTest/lwr_vel_cmd.h"
#include "sensor_msgs/JointState.h"
#include "lwrSimTest/ImageCoordinates.h"
#include "lwrSimTest/Objective.h"
#include "lwrSimTest/ControlErrors.h"
#include <boost/thread.hpp>
```

Linear algebra calculations are made using Eigen library that simplify matrix manipulation in C++ environment.
This node reads joint angles from "/joint_states" topic, published by "gazebo" node, marker coordinates from "/marker" topic and target coordinates from "/target" topic, both published by the "optical" node.

```
subq = n.subscribe("/joint_states", 1, ctrlcallback);
subc = n.subscribe("/fixed_camera/marker", 1, cam_callback);
subt = n.subscribe("/fixed_camera/target", 1, tgt_callback);
```

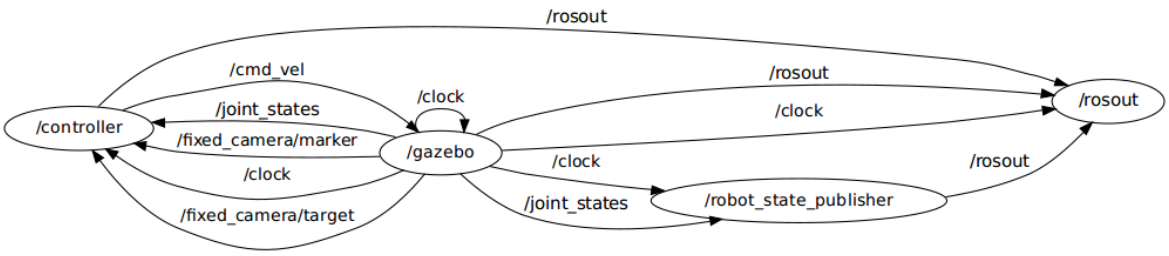

Figure 3: ROS graph: control node communicating with other nodes.

The core of the program is a "while" loop:

```
ros::Rate loop_rate(30);

while (ros::ok()) {
    ros::spinOnce();
    loop_rate.sleep();
}
    return 0;
}
```

[15] http://www.boost.org

that waits for messages and, when they arrive, execute the callbacks. This ROS loop is executed at the frequency specified by the "loop rate" statement. Our node also send messages on "/cmd_vel" topic that provides velocity commands to the robot plugin.

pub = n.advertise<lwrSimTest::lwr_vel_cmd>("/cmd_vel", 1);

The controller is implemented in the "joint states" callback, so that, when a new joint states vector arrives, a new command is generated and sent to the robot.
Even direct kinematics, homogeneous transformations and jacobian computations are performed in the same file. Jacobian pseudo-inverse is computed via SVD: singular values are also used in order to check for singularity states.
Penetration (lambda) control may be enabled via a graphic switch whose position is sent by "optical" node to the controller; even desired lambda is set by a graphic cursor. If and only if $\lambda \in [0, 1]$ (remember that λ is normalized) computed commands are sent to the robot, otherwise robot stops.

8 Simulation results

Two experiments, with the same experiment task, where performed to test the controller response with and without penetration parameter λ control. In the first experiment such control is disabled while enable in the second with the target penetration factor λ_0 set to 0.3. The experiment task consist in the positioning of the end-effector of the robot from a fixed initial position to a fixed target position in the image plane. Here follows some screenshots of Gazebo's environment taken while the simulation was running.

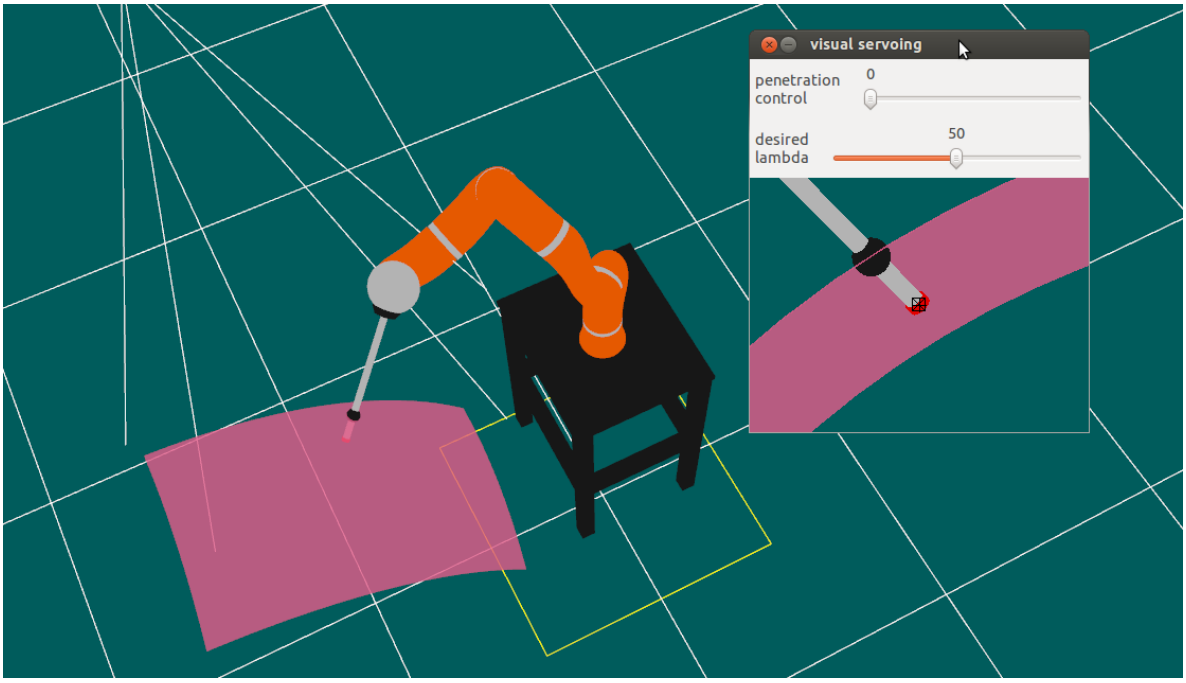

Figure 4: Robot in the starting configuration

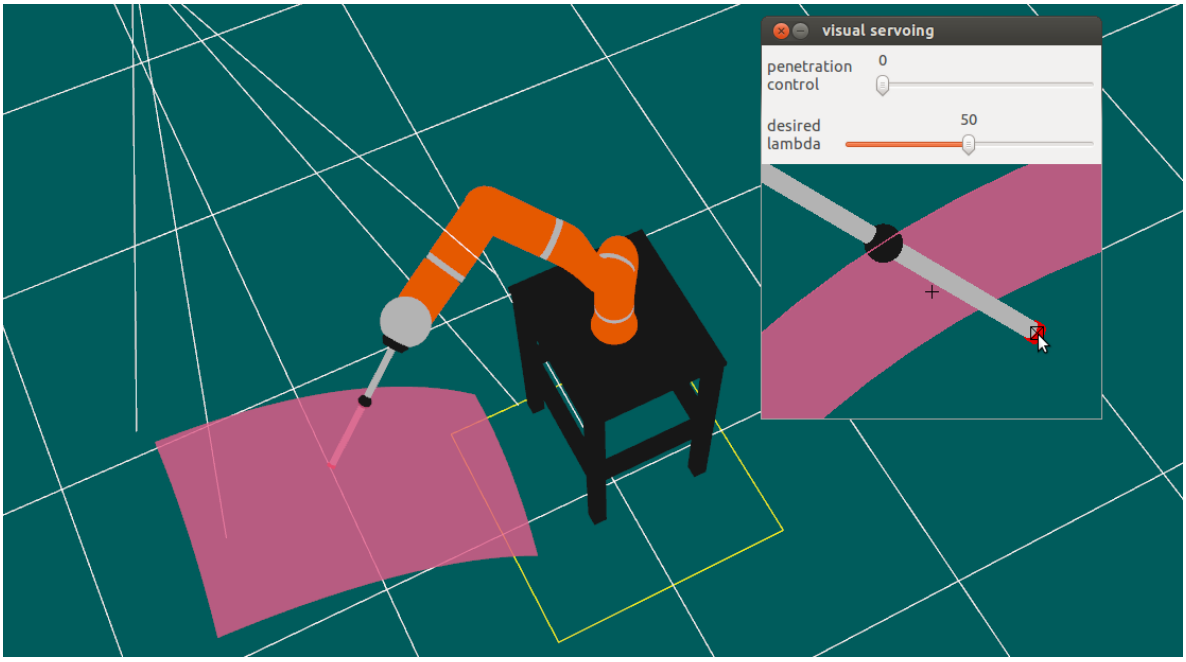

Figure 5: Task accomplished with penetration control disabled

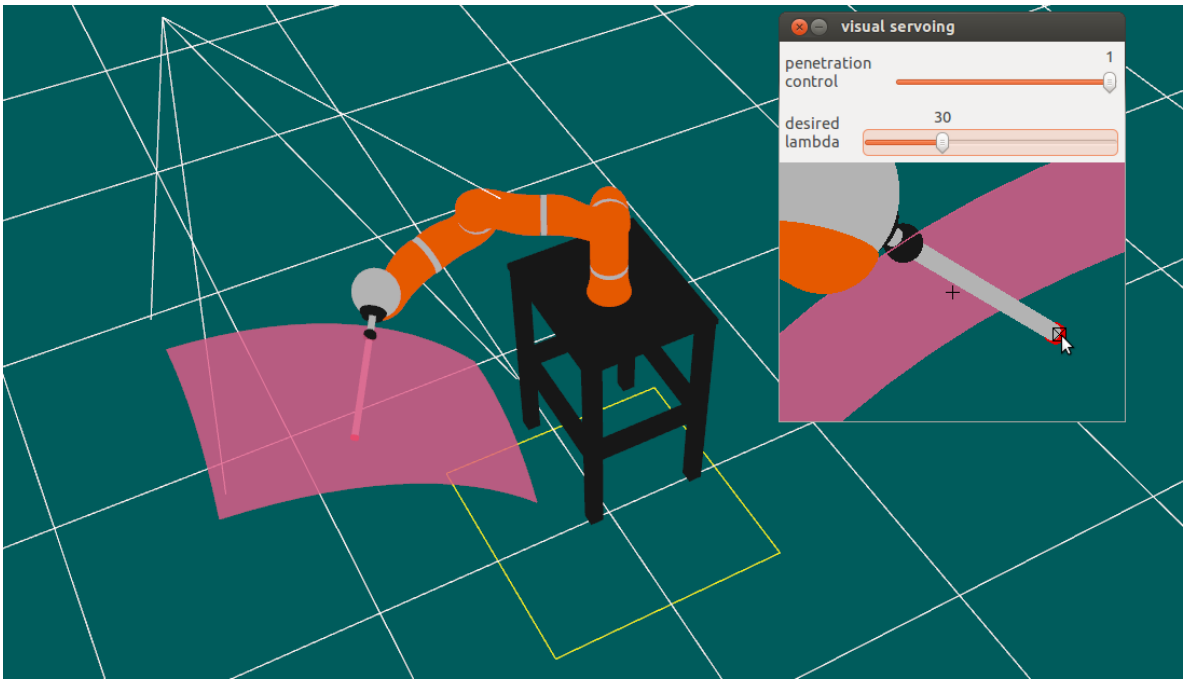

Figure 6: Task accomplished with penetration control enabled

As seen on Figures 7, 8 and 9 the RCM error remains zero in both cases but has some little vibrations (order of magnitude 10^{-3}) when the new target is defined. These variations can be made smaller by reducing ODE's integration step and by choosing a higher operating frequency for the controller. The visual error, as seen on Figures 10 and 11 remains almost unchanged, with the same exponential decay to zero. The penetration factor as seen in 12 and 13 change its behavior to follow the control constraints.
A second test was performed to obtain a better characterization of the penetration controller respect to the penetration reference. In this case the experiment conditions are the same of the previous second experiment but is repeated with increasing values of λ_0 from 0.0 to 1.0 with increment of 0.1. Figure 14 shows the regime value of the λ error as function of the reference λ_0. The behavior shows that, in the chosen configuration, for $\lambda > \lambda_{lim}(u_{target}, v_{target}) \approx 0.6$, the lower priority secondary task cannot be satisfied because its incompatibility with the primary tasks, as expected for a projected gradient control scheme.

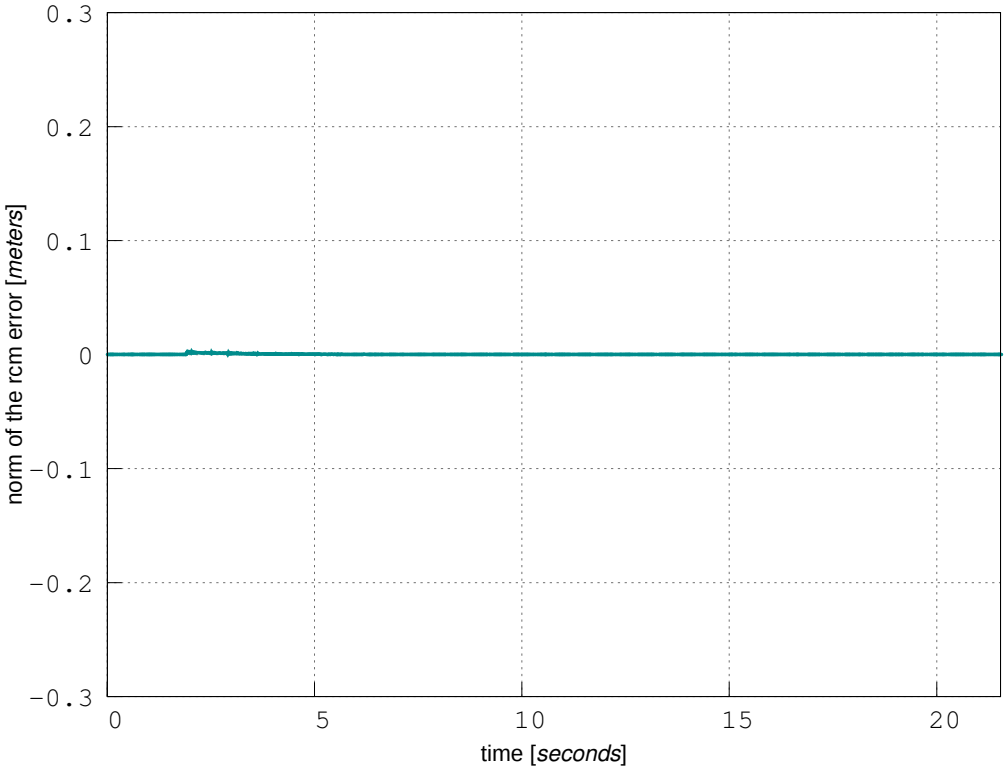

Figure 7: RCM error

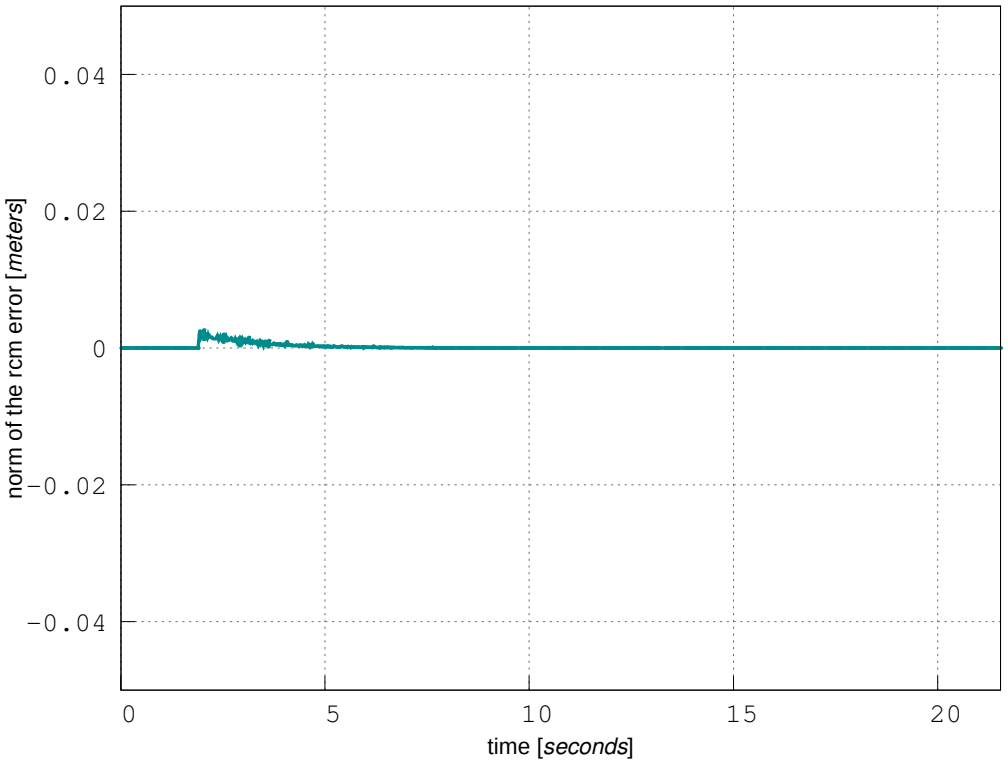

Figure 8: RCM error - detail with expanded scale

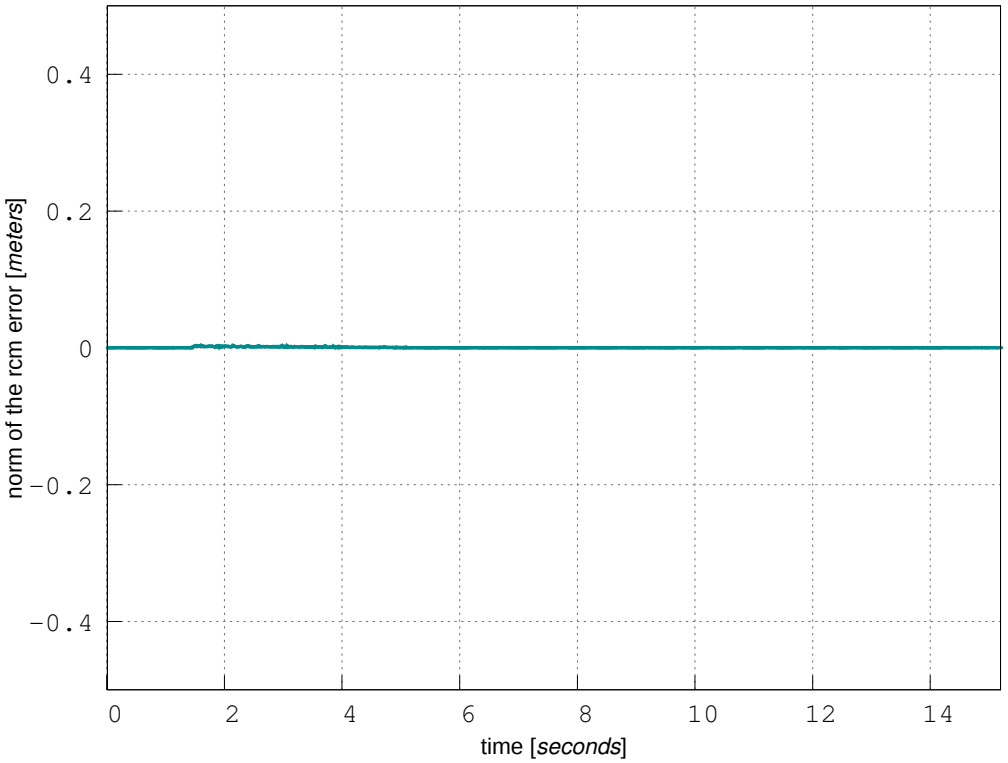

Figure 9: RCM error - penetration controller enabled

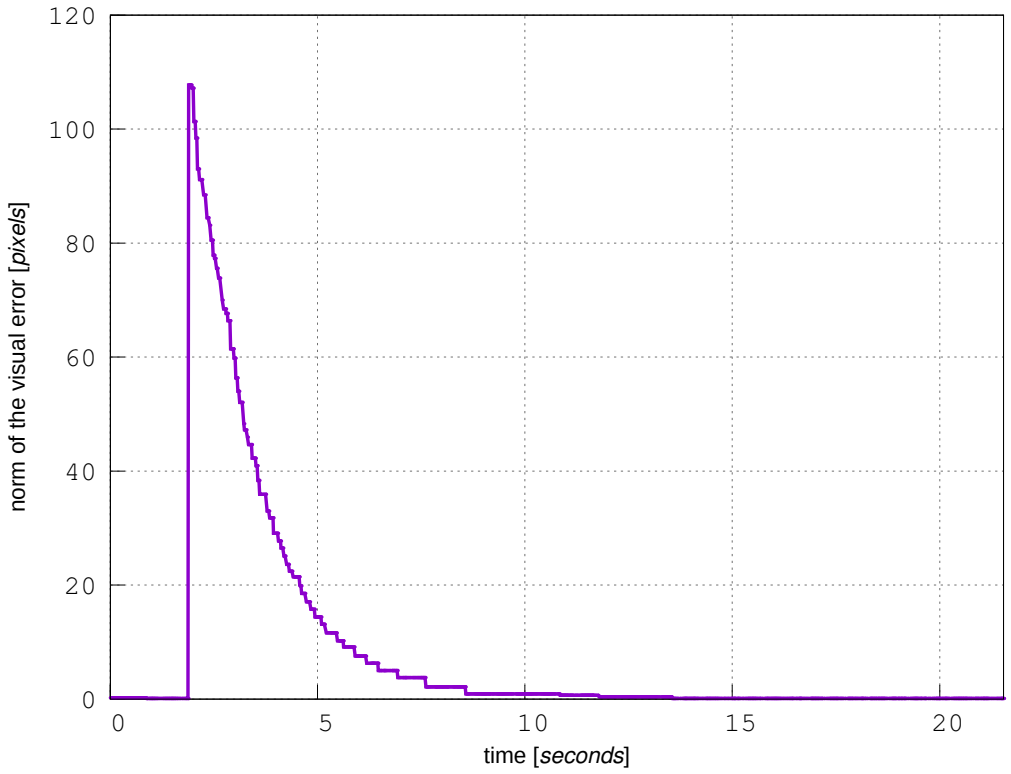

Figure 10: Visual error

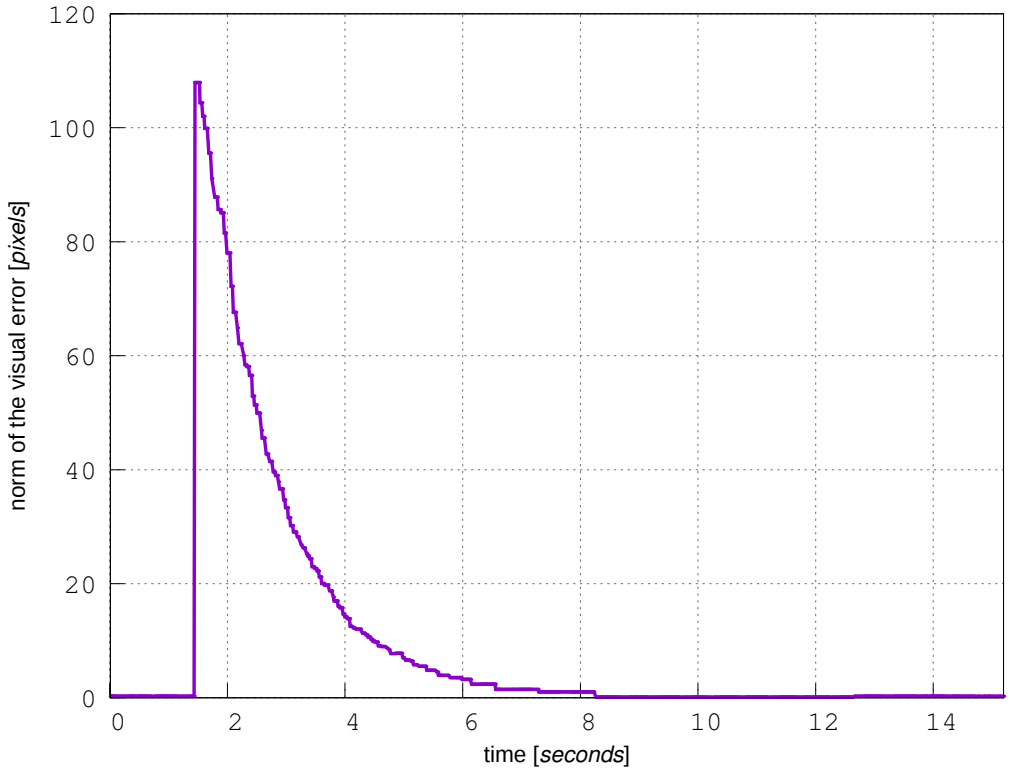

Figure 11: Visual error - penetration controller enabled

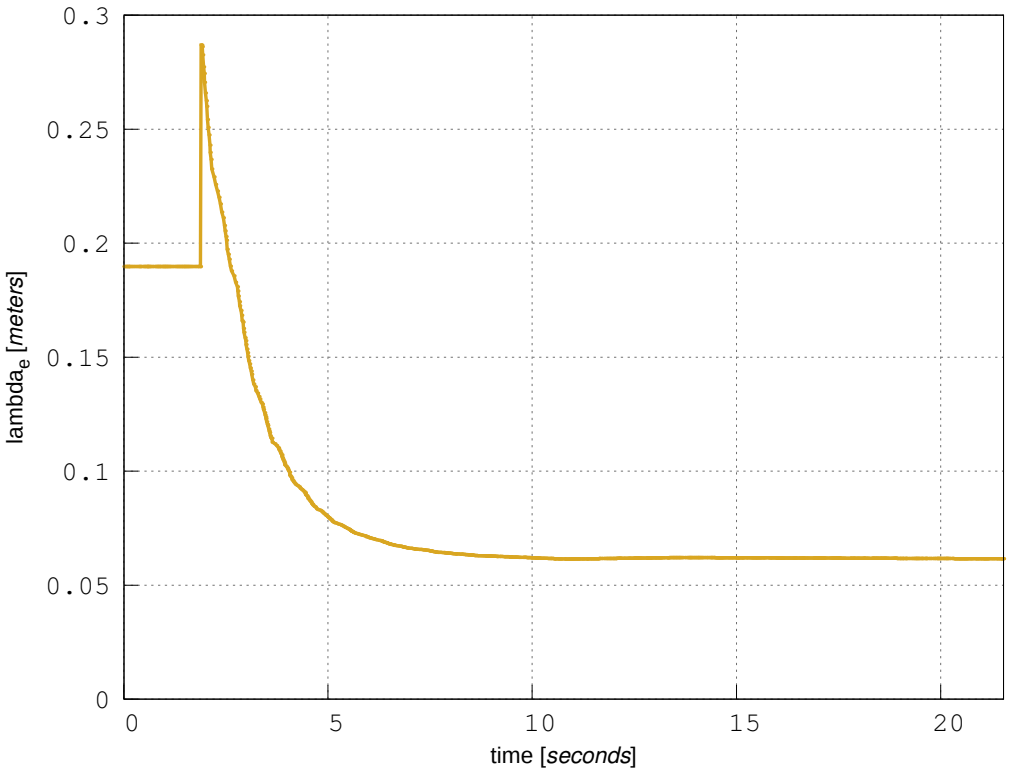

Figure 12: λ error without controller

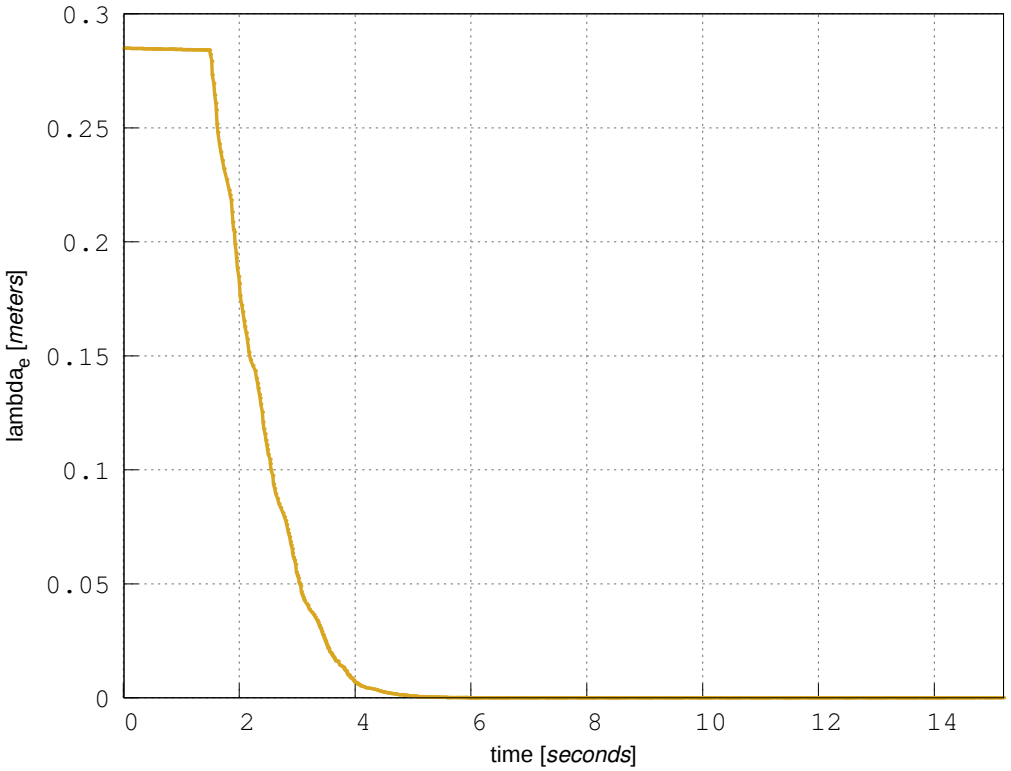

Figure 13: λ error - penetration controller enabled

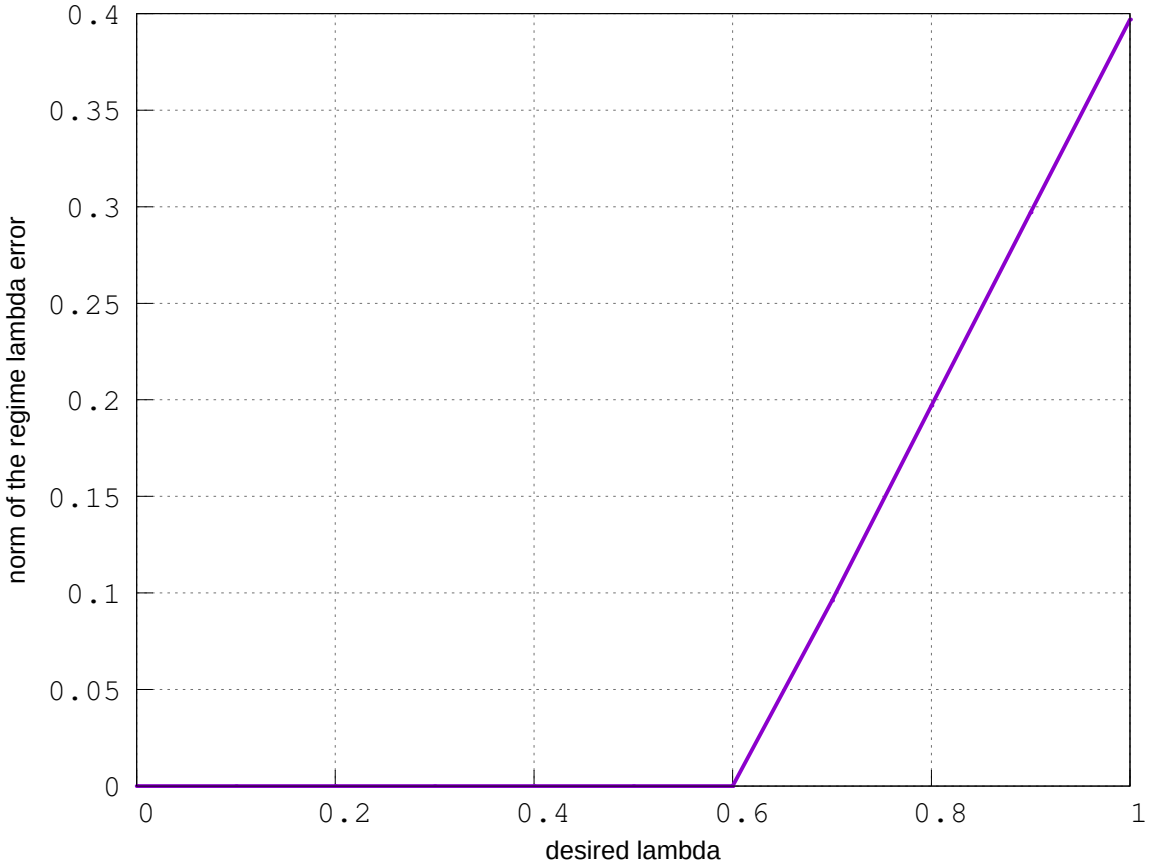

Figure 14: Regime penetration error for different desired λ

Bibliography

- Aghakhani, Geravand, Shahriari, Vendittelli, Oriolo: "Task Control with Remote Center of Motion Constraint for Minimally Invasive Robotic Surgery", 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems
- Krupa et al.: "Autonomous 3-D Positioning of Surgical Instruments in Robotized Laparoscopic Surgery Using Visual Servoing", IEEE Transactions on Robotics and Automation, vol. 19, no. 5, october 2003
- Hutchinson, Hager, Corke: "A tutorial on Visual Servo Control", IEEE Transactions on Robotics and Automation, vol. 12, no. 5, october 1996
- Chaumette, Hutchinson: "Visual Servo Control. Part I: Basic Approaches", IEEE Robotics & Automation Magazine, December 2006
- Siciliano, Sciavicco, Villani, Oriolo: "Robotics: modelling, planning and control", ed. 2009, Springer
- Gazebo website: http://gazebosim.org
- Opencv website: http://opencv.org
- ROS Wiki: http://www.ros.org/wiki