Mutual Localization Documentation

Release 0.1

Vikas Dhiman, Julian Ryde and Jason J. Corso

Nov 09, 2019

Contents

Contents:

CHAPTER 1

Introduction

This library includes the experiments used for the development of the algorithm as described in the paper

V. Dhiman, J. Ryde, and J. J. Corso. Mutual localization: Two camera relative 6-dof pose estimation from reciprocal fiducial observation. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), November 2013.

A copy of paper is supplied with this release: download paper.

This document intends to relate code with the paper.

CHAPTER 2

Installation

To run the experiments, first we need to install the python dependencies that are required for the code. Some dependencies are essential for the core of the algorithm other are required for some peripheral code.

2.1 Dependencies

- Opencv : cv2
- Numpy : numpy
- matplotlib
- mayavi2
- scipy.linalg
- scipy.ndimage
- scipy.optimize
- logging
- pyexiv2
- pygame
- sympy
- yaml
- unittest
- texlive

Installing core dependencies:

```
sudo apt-get install libopencv python-opencv python-numpy python-sympy python-
↪matplotlib python-scipy python-yaml mayavi2 texlive
```

Installing other dependencies:

```
sudo apt-get install python-unittest python-pyexiv2 python-pygame
```

CHAPTER 3

Download

Download from here.:

```
git clone git@github.com:wecacuee/mutual_localization.git
```

Get data by using the script in data directory:

    cd data/ python wgetdata.py

Setting python path:

```
cd mutual_localization/
export PYTHONPATH=$PYTHONPATH:`pwd`/lib:`pwd`/src
```

For the rest of the documentation we will assume that we will assume that we are in the `mutual_localization` directory.

CHAPTER 4

Example

Here we describe how to use the important functions of the library. If you just want to repeat the experiments, please jump to experiments section *Experiments*.

4.1 Terminology

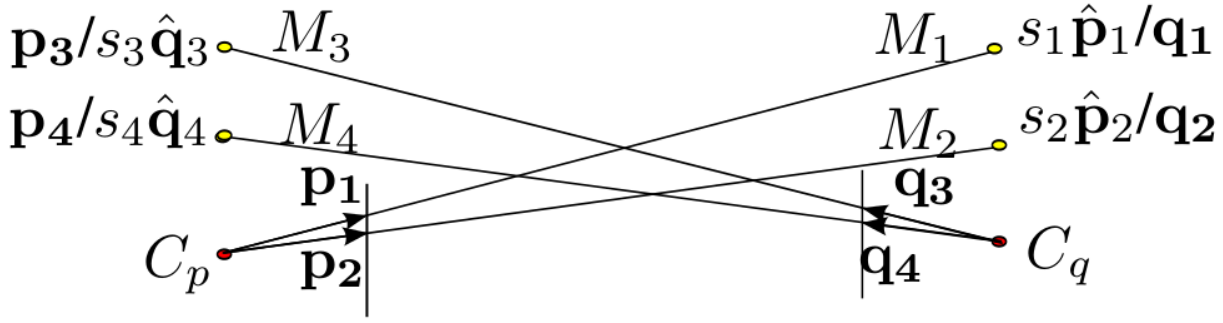

Fig. 1: Simplified diagram for the two-camera problem. Assuming the length of respective rays to be s_1, s_2, s_3, s_4 respectively, each marker coordinates can be written in both coordinate frames $\{p\}$ and $\{q\}$. For example M_1 is $s_1 p_1$ in frame $\{p\}$ and q_1 in $\{q\}$, where p_1 unit vector parallel to p_1.

We have used the variable names *frame1scaled* and *frame2scaled* repeatedly in the code. In the above configuration frame1scaled will be a list of pairs of marker positions that have "scaled" positions in frame "1". If $\{p\}$ is assumed to be the frame "1" than frame1scaled will be [($s_1 p_1$, q_1), ($s_1 p_2$, q_2)]

Note that the markers M_1, M_2 are "scaled" in frame 1.

Similarly the *frame2scaled* represents the rest of the marker position pairs. [(p_3, s_3q_3), (p_4, s_4q_4)]

4.2 From coordinates with scale ambiguity to rotation translation.

This example will walk through computing R, t from $[(p_1, q_1), (p_2, q_2), (p_3, q_3), (p_4, q_4)]$

Since the polynomials to be solved are too complicated to be solved by hand and we have different polynomials for different distributions of markers over the robots. This is why we use symbolic mathematics package `sympy` to generate the multi-variate polynomials, eleminating scale varibles to get single variable polynomials.

Assuming that you have reached the point where you have normalized vectors as shown in the diagram above. The following code will help you compute the scale factors (s_1, s_2, s_3, s_4).

Let us create an test case. The test case is implemented in `mutloc.test.test_pose_computation`. We will try to walk you through the test case. Suppose there are two marker per camera and the arrangement of markers is same.:

```
# Set the absolute position of markers
# M_1, M_2 are fixed on robot L
markersL = [np.array(p) for p in [(-.1, -.1, .3), (.1, -.1, .3)]]
# M_3, M_4 are fixed on robot R
markersR = [np.array(p) for p in [(-.1, -.1, .3), (.1, -.1, .3)]]
```

Next we choose a random transform between the two robots. We choose a random rotation without any constraints and set translation between 0.7 units and 10 units. Note that minimum translation is set to twice the euclidean distance of he markers considering the arguments given in section III C of the paper; otherwise in a certain rotation configuration the marker of the other robot can be nearer robot's own markers.:

```
# Choose a random rotation
quaternion = [random.random() for i in range(4)]
mag = math.sqrt(sum([x**2 for x in quaternion]))
quaternion = [x/mag for x in quaternion]

# Random translation with minimum more than twise the euclidean
# distance of the markers.
translation = [random.uniform(.7, 10) for i in range(3)]
```

Compute the coordinates of M_1, M_2 in frame 2 and M_3, M_4 in frame 1. These computed coordinates will be normalized to unit vector to simulate perspective projection.

```
# Christoph Gohlke's transformations.py (included)
import transformations as tf
import mutloc.utils as utils

# Convert to a transform matrix
import transformations as tf
T = tf.quaternion_matrix(quaternion)
T[:3, 3] = translation

# compute the marker positions in the other coordinate frame
# M_3 and M_4 in coordinate frame L
Tinv = utils.transform_inv(T)
frame1scaled = [(utils.apply_transform(Tinv, pR), pR)
                for pR in markersR]

# M_1 and M_2 in coordinate frame R
```

(continues on next page)

(continued from previous page)

```
frame2scaled = [(pL, utils.apply_transform(T, pL))
                for pL in markersL]

# Normalize computed coordinates to unit vector in order to
# simulate perspective projection
frame1scaled = [(p1 / np.linalg.norm(p1), p2) for p1, p2 in frame1scaled]
frame2scaled = [(p1, p2 / np.linalg.norm(p2)) for p1, p2 in frame2scaled]
```

Now we feed these arguments to our method `mutloc.core.solve_mutual_localization`:

```
Tgot_roots = corr.solve_mutual_localization(frame1scaled, frame2scaled, tol=TOL)
```

Their may be multiple roots, but most of low noise situations have only one root as the roots have already been filtered and sorted by error. The first root is the most likely transformation. We can check that this transformation is what we expected.:

```
self.assertTrue(
    np.allclose(Tgot_roots[0], T, atol=TOL),
    "Got {0} expected {1}".format(Tgot_roots[0], T))
```

CHAPTER 5

Experiments

Repeatability of experiments is basic requirement for scientific research. We take the repeatability of our experiments very seriously and hence we release our source code and data to make our experiment repeatable.

5.1 Localization Experiment (Section V-A)

Setting up and running Bundler and ARToolkit is not part of this documentation so far. We have included the intermediate bundler results for comparison.:

```
python scripts/artk_vs_mutloc_on_tiles.py data/tiledexperiment_results/corrected_out.
↪txt
```

The raw file generated from all the data collected is `data/tiledexperiment_results/out.txt` but here the 0:sup:*th* entry is repeated as 1:sup:*st* entry. So in principal you can use `out.txt` in place of `corrected_out.txt` but all you will have to remove some other entry because the number of experiments in `data/tiledexperiment_results/bundler-tmp/bundle/bundle.out` should be same as `out.txt`.

This script apart from throwing out the plots used in the paper also shows up a nice mayavi2 visualization in 3D of estimated positions by each of the algorithm.

The raw data was generated by ROS nodes `ros/mutloc_ros/nodes/tilesexperiment.py` and corresspoding nodes of ROS ARToolkit package. The node allows you to caputure raw images along with computed position by ARToolkit and Mutual localization. Although we ran the experiments on incoming ROS topics on the fly, and saved the corresponding images to disk. Since we provide all these saved images (in `data/tiledexperiment_results`), it should be possible to repeat the experiments using ARToolkit and mutual localization. The logic to compute detections for mutual localizatoin is given in *ros/mutual_ros/src/muutloc_ros/detector.py*'.

5.2 Simulation experiments with noise (Section V-B)

This experiment can be run by simply running the following script. The path to data directory is hard coded in the script.

```
python scripts/noise_vs_error_on_blender.py
```

5.3 Reconstruction experiment (Section V-C)

Once camera localizations are generated by mutual localization method or bundler, we can feed them to the PMVS-2 library for semi-dense 3D reconstruction.:

```
python scripts/print_target_localization.py data/bnwmarker20120831/calib.yml
```

5.4 Additional Experiments

There are few experiments that are not mentioned in the paper, but can be of interest to the audience (And yes, I like to show off.)

5.4.1 Target Localization in Blender simulation

```
python scripts/print_target_localization_blender.py
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search