Object Pose Estimation and Camera Localization Code Documentation

September 5, 2016

This document explains the high level concepts and setup of our pose estimation code. The code can be used to estimate 6DoF poses of potentially multiple objects from RGB or RGB-D images. It also supports camera localization (e.g. pose estimation of scenes). You find a explanation of the method and theory in our CVPR 16 paper. Please cite this paper, if you use this code in your own work:

E. Brachmann, F. Michel, A. Krull, M. Y. Yang, S. Gumhold, C. Rother, "Uncertainty-Driven 6D Pose Estimation of Objects and Scenes from a Single RGB Image", CVPR 2016

Compiling the code creates two programs:

- `train_trees`: Reads in training data and trains an auto-context random forest that predicts per pixel object label distributions and conditional object coordinate distributions.
- `test_pose_estimation`: Processes test frames with a previously trained random forest and estimates object poses.

Compiling the code will require: OpenCV 2.4, PNG++, NLOpt, and the EDISON source code. Details can be found at the end if this document. We compiled the code using Ubuntu 14.04 and GCC 4.8.4.

The code is published under the BSD License.

In the following, we will first explain which data has to be available and in which structure when running our code. Then, we describe the two programs `train_trees` and `test_pose_estimation` in more detail. Next, we list all parameters which are used throughout the code. Finally, we give a dummy example of how to compile and execute the code.

In case you have questions regarding the code, feel free to contact the first author of the associated paper.

1 Data Structure

Both tools expect training and test data to be of a specific format and be structured in a specific way.

```
training
    object1
        depth_noseg
        rgb_noseg
        seg
        obj
        info
    object2
        ...
    object3
        ...
    ...
test
   ...
background
    backgroundset
        depth_noseg
        rgb_noseg
info1.txt
info2.txt
info3.txt
...
pc1.xyz
pc2.xyz
pc3.xyz
...
default.config
```

Per object, there is one sub-folder in `training` and `test`, respectively. Both programs will determine the total number of objects from the number of sub-folders. The object sub-folders contain the actual training resp. test data: `depth_noseg` contains depth images (16 bit PNG file, 1 channel, unsigned short, depth in mm), `rgb_noseg` contains RGB images (8 bit PNG file, 3 channels, unsigned char), `seg` contains object segmentation masks (8 bit PNG file, 1 channel, unsigned char), `obj` contains object coordinate ground truth (16 bit PNG file, 3 channel, unsigned short), `info` contains ground truth object poses (ASCII txt files, details below). The `test` folder should contain the exact same number of object folders as `training` with the same sub-folder structure.

There is also an optional `background` folder which can contain RGB and depth images of background scenes (e.g. random rooms). They can serve as negative class when training the forest, and will improve performance when available.

There are also some files in the root folder which have to follow specific naming conventions. Info files `infoX.txt` and point cloud files `pcX.xyz` have to be provided for each object, where `X` is the object ID. The ID of an object

is its index (starting with 1) after sorting the object sub-folders in `training` alphabetically. The info files are needed to read in the object extents at the beginning of `test_pose_estimation`. The poses in these specific info files will be ignored.
Info file format:

```
image size
<image width> <image height>
<object name>
occlusion: [optional]
<fraction occluded> [optional]
rotation:
<rot11> <rot12> <rot13>
<rot21> <rot22> <rot23>
<rot31> <rot32> <rot33>
center:
<transX> <transY> <transZ>
extent:
<extentX> <extentY> <extentY>
```

All entries marked with `<>` have to be filled with the appropriate values. Entries marked with [optional] can be omitted. Image size is given in pixels. The object pose is given as a 3x3 rotation matrix and a translation vector in meters (`center`). The object extent is also given in meters.

Point cloud files are needed to calculate certain pose error metrics during the execution of `test_pose_estimation`.
Point cloud file format:

```
<vertex1X> <vertex1Y> <vertex1Z>
<vertex2X> <vertex2Y> <vertex2Z>
...
```

Vertex positions should be given in meters. Other point cloud formats are also supported. Please see `pointCloudTools.h` for details.

There should also be a `default.config` configuration file listing training and testing parameters. Details on setting parameters including configuration files can be found later in this document.

2 Random Forest Trainer

`train_trees` should be executed in the same folder which contains `training` and `test`. It reads the training data of *all* objects in the `training` folder, and trains one joint (auto-context) random forest for all these objects. In the forest, objects are associated with ID numbers. These numbers correspond to the alphabetical order of sub-folders in `training` starting with 1. In general, it is not recommended to train a random forest for less than 3 objects, since its segmentation performance might suffer. Additionally, the use of a background image set (see above) is recommended. The executable generates multiple binary `*.rf` files which contain the auto-context random forests. One file contains

one forest of the auto-context stack. The file names encode some of the main parameters used for training.

3 Pose Estimation

`test_pose_estimation` should be executed in the same folder which contains `training` and `test`. It loads a forest previously trained, reads the test data of a *specfic* object in the `test` folder and processes it frame by frame. It either estimates the poses of all known objects in each frame, or just the pose of a specific one based on testing parameters. By default the program will stop after each frame and show segmentation results, object coordinate estimations and estimated poses together with the respective ground truth. Pressing any button (while the image windows are active) will continue to the next frame. Evaluation results are printed to the console. After each frame it prints for all objects processed:

0 object name

1 percentage of occlusion

2 angular distance to the closest training image

3 average probability predicted for that object within the ground truth segmentation

4 average probability predicted for that object outside the ground truth segmentation

5 percentage of object coordinate predictions within the ground truth segmentation that where correct within a certain tolerance

6 whether the object was recognized to be in the image

7 whether the object was correctly localized based on 2D bounding box overlap (50% IoU)

8 whether the pose of the object was correctly estimated according to the metric proposed by Hinterstoisser et al.

9 whether the pose error was below 5 deg and 5 cm

10 whether the pose error was below 10 deg and 10 cm

11 the angular error of the pose estimate in deg

12 the translational error of the pose estimate in cm

After processing all images, the program will print average statistics to the console. These are:

0 object name

1 number of frames where the object appeared

2 average percentage of occlusion

3 average angular distance in deg to the closest training image

4 average predicted probability for that object within the ground truth segmentation

5 average predicted probability for that object outside the ground truth segmentation

6 average percentage of object coordinate predictions within the ground truth segmentation that were correct within a certain tolerance

7 percentage of frames where the object was correctly recognized to be in the frame

8 percentage of frames where the object was correctly localized based on 2D bounding box overlap (50% IoU)

9 percentage of frames where the estimated pose was correct according to the metric proposed by Hinterstoisser et al.

10 percentage of frames where the pose error was below 5 deg and 5 cm

11 percentage of frames where the pose error was below 10 deg and 10 cm

12 the median angular error in deg

13 the median translational error in deg

There is also a batch mode available (see parameter section below). In this mode, the program will display no result images but instead work through all available frames without stopping. In this mode, multiple output files per object will be generated (`X` is the object ID):
`objX.txt` - Contains statistics averaged over frames. One line per run of `test_pose_estimation`:

0 session ID (see parameter section, might be empty)

1 config file used

2 object ID

3 frames processed

4 average percentage occluded

5 average angular distance in deg to the closest training image

6 *"inliers"*

7 average percentage of object coordinate predictions within the ground truth segmentation that were correct within a certain tolerance

8 *"probs"*

9 average predicted probability for that object within the ground truth segmentation

10 average predicted probability for that object outside the ground truth segmentation

11 *"detection"*

12 percentage of frames where the object was correctly recognized to be in the frame

13 percentage of frames where the object was correctly localized based on 2D bounding box overlap (50% IoU)

14 *"poses"*

15 percentage of frames where the estimated pose was correct according to the metric proposed by Hinterstoisser et al.

16 percentage of frames where the pose error was below 5 deg and 5 cm

17 percentage of frames where the pose error was below 10 deg and 10 cm

18 *"times"*

19 average time for random forest evaluation in ms

20 average time for RANSAC stage in ms

`objX_details.txt` - Contains statistics per frame. One line per frame. Multiple runs of `test_pose_estimation` will append results:

0 session ID (see parameter section, might be empty)

1 frame number

2 is the object in the frame?

3 angular distance to the closest training image

4 percentage of occlusion

5 percentage of object coordinate predictions within the ground truth segmentation that where correct within a certain tolerance

6 average probability predicted for that object within the ground truth segmentation

7 average probability predicted for that object outside the ground truth segmentation

8 has the object been recognized by RANSAC?

9 2D bounding box overlap (intersection over union)

10 X component of translational pose error (mm)

11 Y component of translational pose error (mm)

12 Z component of translational pose error (mm)

13 rotational pose error (deg)

14 average error of projected vertices (2D)

15 average error of transformed vertices (3D, metric proposed by Hinterstoisser et al.)

16 inliers identified by RANSAC

17 pixel count looked at by RANSAC

18 likelihood of the pose determined by RANSAC

`objXposes.txt` - Contains the estimated pose for this object per frame. One line per frame. Multiple runs `test_pose_estimation` will append results:

0 image number

1 pose rotation, Rodrigues vector, 1st component

2 pose rotation, Rodrigues vector, 2nd component

3 pose rotation, Rodrigues vector, 3rd component

4 pose translation, X in mm

5 pose translation, Y in mm

6 pose translation, Z in mm

4 Parameters

There are two ways of setting the parameters when calling `train_trees` and `test_pose_estimation`. The first and direct way is to append it to the program call in the command line. Every parameter has a few letter abbreviation (case sensitive) which can be given to the command line starting with a dash and followed by a space and the value to set (some parameters are flags and are not followed by a value). Because there might be a large number of parameters to set, there is also the second way which is providing a config file. These files should contain one parameter per line. More specifically, each line should start with the parameter abbreviation (without a leading dash) followed by a space and the value to set. Config files can also include comment lines which start with the # symbol. Both programs will first look for a `default.config` file which will be parsed at the very beginning of each run. After that, command line parameters will be processed (overwriting parameters from the `default.config`). Whenever the program comes across the session ID parameter (`sid`), it will look for a `<sid>.config` and parse that file. After that it will continue parsing the command line arguments.

Here is a complete list of parameters that can be set by command line or config files (see also the file `properties.h` for definition and parsing of parameters).

Parameters related to the data:

iw width of input images (px)

ih height of input images (px)

fl focal length of the camera (depth camera if RGB-D)

xs x position of the principal point (depth camera if RGB-D)

ys y position of the principal point (depth camera if RGB-D)

fso true if the objects to estimate fill the entire image, i.e. full scenes (used for calculating the 2D bounding box during optimization)

rd set to 1 if RGB and depth channels are not registered, this will trigger a coarse registration where the RGB channel is rescaled and shifted

sfl only used if `rd` is true, focal length of the RGB camera

rxs only used if `rd` is true, x shift to be applied to the RGB channel (in px)

rys only used if `rd` is true, y shift to be applied to the RGB channel (in px)

smin minimal scale factor when simulating different image scales (data augmentation, RGB case only)

smax maximal scale factor when simulating different image scales (data augmentation, RGB case only)

srel if 1 the image scale (see above) is calculated relative to the distance in the training image, i.e. the distance in the training image is normalized to be 1m

amin minimal inplane rotation angle (deg) when simulation image rotations (data augmentation)

amax maximal inplane rotation angle (deg) when simulation image rotations (data augmentation)

ud if 1 RGB-D case instead of RGB only

Parameters related to the forest training:

sid custom name that can be used to identify a certain forest (will be appended on the file name)

tc how many trees to train for each auto-context layer

tp how many samples to draw per object (in total, will be split among available training images)

tfr more samples (`tp` · `tfr`) will be drawn when fitting leaf distributions

tfb more samples (`tp` · `tfb`) will be drawn for the background object/class

cs how often are object coordinates quantized in each dimension (`cs`·`cs`·`cs` is the number of pseudo classes during forest training)

wbgr weight of sampling RGB difference features during forest training, integer value

wacc weight of sampling auto-context object class features during forest training (only used after auto-context layer one), integer value

wacr weight of sampling auto-context object coordinate features during forest training (only used after auto-context layer one), integer value

fc how many features to try out per node during forest training

mo maximal offset of the forest features (in px for RGB or in px*m for RGB-D)

md tree branches are only grown until this depth

ms a node is not split further during forest training if less samples arrive

mlp maximum number of samples used in each leaf to fit object coordinate distributions

mb kernel size for mean shift when fitting leaf distributions, in mm

acp number of auto-context layers

acs auto-context feature channels can be kept sub-sampled in order to save memory

Parameters related to pose estimation using RANSAC:

nD enable batch mode (no stopping after frames), this is a flag, do not append a value

rO set to 1 if the object is rotation symmetric (affects the calculation of pose error according to Hinterstoisser et al.)

tO which test sequence to load? (selects a sub-folder of the `test` folder)

sO which object to look for? -1 for looking for all objects simultaneously

iSS only look at every n'th test image (skipping all others), for some quick testing

rI initial number of pose hypotheses drawn per frame

rMD number of times it is tried to draw a valid hypothesis (hypotheses can be rejected if certain contraints are violated)

rB how many pixels are additionally checked in each iteration of preemptive RANSAC?

rT2D inlier threshold in the RGB case (in px)

rT3D inlier threshold in the RGB-D case (in mm), also used to evaluate the quality of the intermediate object coordinate output

rCRI minimal number of times each final hypothesis must be coarsely refined (recalculating the pose based on inlier correspondences)

rMI maximal number of inlier correspondences that are used to recalculate poses in coarse refinement

rR set to 1 to refine final poses further using uncertainty

rRI number of iterations for final pose refinement with the general purpose optimizer (refinement using uncertainty)

rMinI necessary number of inlier correspondences before attempting refinement using uncertainty

Along with this code we also provide the config files for the main experiments of the paper. There are `default.config` files which cover the RGB case, and `rgbd.config` which just state those parameters different for the RGB-D case.

5 Example of Compiling and Running

First, install all required libraries. These are OpenCV 2.4, PNG++ and NLOpt. You probably find appropriate packages via your Linux package manager. Otherwise, visit the respective websites for installation instructions:

OpenCV `http://opencv.org/`

PNG++ `http://www.nongnu.org/pngpp/`

NLOpt `http://ab-initio.mit.edu/wiki/index.php/NLopt`

Furthermore, our code requires third party code from the EDISON program for running mean-shift. Go to `http://coewww.rutgers.edu/riul/research/code/EDISON/` , download and unpack the EDISON source files (`edison_src.zip`). The archive contains an `edison_source` folder which you have to copy to the same location where the `core` folder of our code lies.
ATTENTION! In order to circumvent nasty compilation prerequisites of EDISON, go to `edison_source/segm/msSys.cpp` and comment out the following two functions: `msSystem::Prompt` (lines 162-177) and `msSystem::Progress` (217-228).

To compile our code, go into the `core` folder and create a `build` directory. From there call:

```
cmake ..
make
```

Two executables should be created, `train_trees` and `test_pose_estimation`. To test the programs go to the paper website (where you downloaded the code) and download the dummy data package. This package contains some images of the Hinterstoisser et al. ACCV12 dataset in the structure our code requires. NOTE: You will not get any good results using this data, as it is just too little to train on. Also, the parameters in the accompanying `default.config` are tuned down for faster training. The purpose of this package is just to show how the code works and how the data should look like.

Unpack the data into the `build` folder such that the `dummy_data` folder is directly under `build`.

From inside `dummy_data` call:

```
../train_trees
```

This will train a 3-layer auto-context random forest. The parameters will be taken from the `default.config` file. After training, which could take a few minutes, execute:

```
../test_pose_estimation
```

The program should load the previously trained forests and display results for the first frame. By pressing a key (while the image windows are active) you can go through different frame. The following variation demonstrates the use of command line parameters:

```
../train_trees -acp 1 -sid simple
../test_pose_estimation -acp 1 -sid simple -sO 3 -nD
```

This will only train one auto-context layer (i.e. a standard random forest) because of `-acp 1`. The string `simple` will be attached to the forest file name and the program will try to parse a `simple.config` (but there is none) because of `-sid simple`. `test_pose_estimation` will load the forests you just trained (and not the one you trained before) because you also started it with `-acp 1 -sid simple`. Pose estimation will be performed for the duck only because of `-sO 3`. The program will run through all frames nonstop because of `-nD`. Note that after finishing, 3 new files containing the pose estimation results have been created (but only for object 3).