PyMVG Documentation

Release 1.0

Andrew Straw

August 02, 2015

Contents

PyMVG is a Python implementation of various computational camera geometry operations.

Features:

- triangulate 2D features from multiple calibrated cameras into a single 3D point (using algorithm from the classic textbook by Hartley & Zisserman). [example]
- load/save camera calibrations from ROS (which uses OpenCV)
- load/save camera system calibrations from MultiCamSelfCal
- complete implementation of OpenCV camera model in pure Python in a single file for easy understanding
- complete implementation of DLT camera calibration procedure
- completely vectorized code for rapid operation on many points using numpy
- completely written in Python
- plotting utilities [example 1] [example 2]

It contains a complete re-implementation of the OpenCV camera model and can thus use calibrations made by or for OpenCV. PyMVG is entirely written in Python, and thus – depending on your preferences – it may be significantly easier to understand than the equivalent OpenCV implementation. PyMVG makes extensive use of numpy, and thus when called on large batches of points, is no slower than native code.

CHAPTER 1

Ecosystem

PyMVG is designed to interoperate with OpenCV, ROS, and MultiCamSelfCal. Unit tests ensure exact compatibility with the relevant parts of these packages.

See also opengl-hz.

CHAPTER 2

Development

All development is done on our github repository.

2.1 PyMVG file format

The PyMVG file format specifies a camera system completely. The file is valid JSON. Here is an example that specifies a system of 3 cameras:

```
{ "__pymvg_file_version__": "1.0",
  "camera_system": [
    {"name": "cam1",
     "width": 640,
     "height": 480,
     "P": [[ 320.0, 0, 319.99999999999994, 0 ],
           [ 0, 320.00000000000006, 240.0, 0 ],
           [ 0, 0, 1.0, 0 ]],
     "K": [[ 320.0, 0, 319.99999999999994 ],
           [ 0, 320.00000000000006, 240.0 ],
           [ 0, 0, 1.0 ]],
     "D": [ 0.2, 0.3, 0.1, 0.1, 0.1 ],
     "R": [[ 1.0, 0, 0 ],
           [ 0, 1.0, 0 ],
           [ 0, 0, 1.0 ]],
     "Q": [[ -1.0000000000000004, 0, 0 ],
           [ 0, 1.0, 0 ],
           [ 0, 0, -1.0000000000000004 ]],
     "translation": [ 0, 0, 0.9000000000000005 ]
    },
    {"name": "cam2",
     "width": 640,
     "height": 480,
     "P": [[ 320.0, 0, 319.99999999999994, 0 ],
           [ 0, 320.00000000000006, 240.0, 0 ],
           [ 0, 0, 1.0, 0 ]],
     "K": [[ 320.0, 0, 319.99999999999994 ],
           [ 0, 320.00000000000006, 240.0 ],
           [ 0, 0, 1.0 ]],
     "D": [ 0, 0, 0, 0, 0 ],
     "R": [[ 1.0, 0, 0 ],
           [ 0, 1.0, 0 ],
           [ 0, 0, 1.0 ]],
     "Q": [[ 0, 0, 0.9999999999999999 ],
            [ 0.847998304005088, 0.5299989400031799, 0 ],
            [ -0.5299989400031798, 0.847998304005088, 0 ]],
      "translation": [ 0, 0, 0.9433981132056602 ]
     },
     {"name": "cam3",
      "width": 640,
      "height": 480,
      "P": [[ 320.0, 0, 319.99999999999994, 0 ],
            [ 0, 320.00000000000006, 240.0, 0 ],
            [ 0, 0, 1.0, 0 ]],
      "K": [[ 320.0, 0, 319.99999999999994 ],
            [ 0, 320.00000000000006, 240.0 ],
            [ 0, 0, 1.0 ]],
      "D": [ 0, 0, 0, 0, 0 ],
      "R": [[ 1.0, 0, 0 ],
            [ 0, 1.0, 0 ],
            [ 0, 0, 1.0 ]],
      "Q": [[ 0, 0, 1.0000000000000002 ],
            [ -0.7071067811865475, 0.7071067811865477, 0 ],
            [ -0.7071067811865478, -0.7071067811865475, 0 ]],
      "translation": [ 0, 0, 0.7071067811865475 ]
     }
  ]
}
```

2.2 Plotting utilities

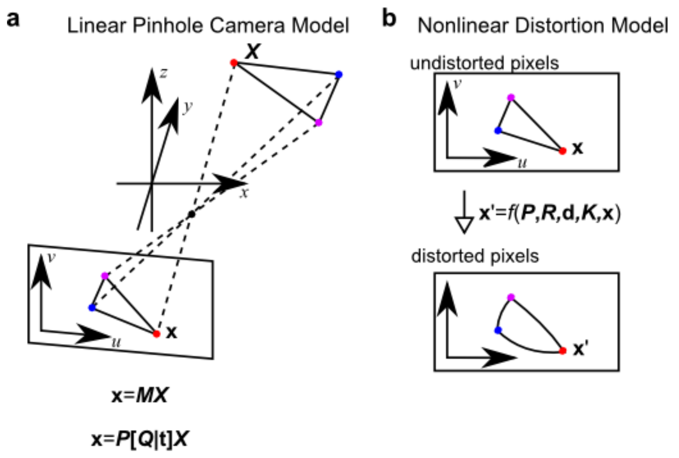
Given the above example, we can plot the camera system.

```python
from pymvg import CameraModel, MultiCameraSystem
from pymvg.plot_utils import plot_system

import os

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fname = os.path.join('..','pymvg_camsystem_example.json')
system = MultiCameraSystem.from_pymvg_file( fname )

fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')
plot_system( ax, system )
ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('z')
ax.set_xlim(-0.8,0.8); ax.set_ylim(-0.8,0.8); ax.set_zlim(-0.8,0.8)
plt.show()
```

2.3 Camera Model

2.3.1 single camera model

The core of PyMVG is a camera model that is compatible with the calibration outputs of OpenCV and MultiCamSelf-Cal.



In the above image, you can see that this camera model consists of a linear pinhole projection model followed by a nonlinear distortion model. The pinhole model is specified completely by the 3x4 matrix M (or, equivalently, the 3x3 intrinsic matrix P, the 3x3 ortho-normal rotation matrix Q, and the translation vector t). The nonlinear distortion model is specified completely by elements of the intrinsic matrix of the pinhole model and several distortion terms.

2.3.2 camera system (multiple cameras)

PyMVG represents a camera system with the MultiCameraSystem class. You create an instance with a list of individual camera instances. The class provides additional methods for triangulation of 3D points and so on.

2.4 API Reference

class `pymvg.camera_model.`**`CameraModel`**(*name*, *width*, *height*, *_rquat*, *_camcenter*, *P*, *K*, *distortion*, *rect*)

an implementation of the Camera Model used by ROS and OpenCV

Tranformations: We can think about the overall projection to 2D in two steps. Step 1 takes 3D world coordinates and, with a simple matrix multiplication and perspective division, projects them to undistorted 2D coordinates. Step 2 takes these undistorted 2D coordinates and distorts them so they are 'distorted' and match up with a real camera with radial distortion, for example.

3D world –(step1)—-> undistorted 2D —(step2)—-> distorted 2D

Step 1 is accomplished by making the world coordinates a homogeneous vector of length 4, multiplying by a 3x4 matrix M (built from P, R and t) to get values [r,s,t] in which the undistorted 2D coordinates are [r/t, s/t]. (The implementation is vectorized so that in fact many points at once can be transformed.)

Step 2 is somewhat complicated in that it allows a separate focal length and camera center to be used for distortion. Undistorted 2D coordinates are transformed first to uncorrected normalized image coordinates using parameters from P, then corrected using a rectification matrix. These corrected normalized image coordinates are then used in conjunction with the distortion model to create distorted normalized pixels which are finally transformed to distorted image pixels by K.

Coordinate system: the camera is looking at +Z, with +X rightward and +Y down. For more information, see http://www.ros.org/wiki/image_pipeline/CameraInfo

As noted on the link above, this differs from the coordinate system of Harley and Zisserman, which has Z forward, Y up, and X to the left (looking towards +Z).'

`camcenter_like`(*nparr*)

create numpy array of camcenters like another array

`get_aligned_camera`(*scale*, *rotation*, *translation*)
: return a copy of this camera with new extrinsic coordinates

`get_flipped_camera`()
: return a copy of this camera looking in the opposite direction

  The returned camera has the same 3D->2D projection. (The 2D->3D projection results in a vector in the opposite direction.)

`get_mirror_camera`(*axis='lr'*, *hold_center=False*)
: return a copy of this camera whose x coordinate is (image_width-x)

`get_view_camera`(*eye*, *lookat*, *up=None*)
: return a copy of this camera with new extrinsic coordinates

`is_distorted_and_skewed`(*max_skew_ratio=1000000000000000.0*)
: True if pixels are skewed and distorted

`is_opencv_compatible`()
: True iff there is no skew

classmethod `load_camera_from_M`(*pmat*, *width=None*, *height=None*, *name='cam'*, *distortion_coefficients=None*, *_depth=0*, *eps=1e-15*)
: create CameraModel instance from a camera matrix M

classmethod `load_camera_from_opened_bagfile`(*bag*, *extrinsics_required=True*)
: factory function for class CameraModel

  bag - an opened rosbag.Bag instance extrinsics_required - are extrinsic parameters required

`project_3d_to_camera_frame`(*pts3d*)
: take 3D coordinates in world frame and convert to camera frame

`project_camera_frame_to_3d`(*pts3d*)
: take 3D coordinates in camera frame and convert to world frame

`save_to_bagfile`(*fname*, *roslib*)
: save CameraModel to ROS bag file

  fname - filename or file descriptor to save to roslib - the roslib module

class `pymvg.multi_camera_system.`**`MultiCameraSystem`**(*cameras*)

`find3d`(*pts*, *undistort=True*)
: Find 3D coordinate using all data given

  Implements a linear triangulation method to find a 3D point. For example, see Hartley & Zisserman section 12.2 (p.312).

  By default, this function will undistort 2D points before finding a 3D point.

classmethod `from_mcsc`(*dirname*)
: create MultiCameraSystem from output directory of MultiCamSelfCal

`get_aligned_copy`(*other*)
: return copy of self that is scaled, translated, and rotated to best match other

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index