argus$_{d}ocsDocumentation$

Release latest

March 09, 2016

Contents

Contents:

CHAPTER 1

Introduction

Thank you for using Argus. Argus is a free, opensource suite of 3D camera calibration and reconstruction tools written in python. Argus was originally created with goal of accomplishing 6 major tasks, each of which is encapsulated in a GUI included with Argus.

- Removing lens distortion from video and images (DWarp)
- Finding the offset between videos started at slightly different times using audio signals (Sync)
- Finding dot grids and checkerboard patterns for the purpose of solving for distortion coefficients (Patterns)
- Solving for distortion coefficients and other camera intrinsics (focal length, optical center) (Calibrate)
- Marking pixel coordinates for the purpose of triangulation and multi-camera calibration (Clicker)
- Solving for DLT coefficients using SBA (Wand)

More detailed documentation of the GUIs and their layouts can be found at http://argus.web.unc.edu/documentation. In addition to these GUIs, Argus offers many other functions and classes that can aid in 3D camera calibration tasks including:

- Removing and adding lens distortion from points in an array
- Projecting back and forth between the image plane and 3D space, given DLT coefficients
- Directly solve for DLT coefficients using least-squares method
- Adding homemade camera intrinsics and distortion profiles to the Argus library
- Triangulation, given camera intrinsics and marked pixel coordinates
- Bootstrapping for confidence intervals in 3D coordinates

Argus was created by Dylan Ray and Dennis Evangelista at the University of North Carolina at Chapel Hill. If you have any questions, comments, or concerns please email: ddray@email.unc.edu

Thanks to: My mom Dr. Tyson Hedrick Manolis I. A. Lourakis (Sparse Bundle Adjustment)

CHAPTER 2

Examples

GUIs

GUI objects can be instantiated in multiple ways. You may simply run the Argus launcher from terminal:

```
$ Argus
```

which will bring up a Tkinter window with 6 buttons that each launches a GUI.

Or you may instantiate any of the GUI objects within a python console or script:

```
import argus_gui as ag

# bring up DWarp
ag.dwarpGUI()

# bring up Sync
ag.syncGUI()

# bring up Patterns
ag.patternsGUI()

# bring up Calibrate
ag.calibrateGUI()

# bring up Wand
ag.wandGUI()

# bring up Clicker
ag.clickerGUI()
```

Finally, Argus comes with executable scripts that can accomplish the same tasks as these GUIs in command line:

```
# undistort a video
$ argus-dwarp movie.mp4 --model "GoPro Hero4 Black" --mode 1080p-120fps-wide --write --ofile undisto

# find the offset between three videos
$ argus-sync "a.mp4,b.mp4,c.mp4"

# find patterns in a video and write to a pickle file
$ argus-patterns patterns.mp4 patterns.pkl

# solve for camera intrinsics and pinhole distortion coefficients, write to a CSV
$ argus-calibrate patterns.pkl intrinsics.csv --inverted
```

```
# use SBA to solve for DLT coefficients and output 3D coordinates
$ argus-wand cams.txt project_name --paired_points wand-xypts.csv --scale 1
```

For more information on the function and use of the GUIs please consult the GUI section of the documenation found at http://argus.web.unc.edu.

Other functions

Argus can be used to accomplish a variety of other camera calibration tasks. Here are a few examples:

```
"""

Undistorting and redistorting points from an array

"""


from argus_gui import DistortionProfile

# getting a profile, Model: GoPro Hero4 Black, Mode: 1440p-80fps-wide'
dis = DistortionProfile()
prof = dis.get_coefficients('GoPro Hero4 Black', '1440p-80fps-wide')

# pixel coordinate array
pts = np.array([[100., 250.],[15., 525.]])

print 'Original pixel coordinates: ', pts

# undistort
pts = undistort_pts(pts, prof)

print 'Undistorted pixel coordinates: ', pts

# redistort back
pts = redistort_pts(pts, prof)

print 'Redistorted pixel coordinates: ', pts


"""

Undistorting a frame from a movie, saving as a png

"""


# filname
movie = '1080p_120_test.mp4'

# make a distortion profile
dis = DistortionProfile()
prof = dis.get_coefficients('GoPro Hero4 Black', '1080p-120fps-wide')

# get an undistorter object
undistorter = dis.get_undistorter()
undistorter.set_movie(movie)

# undistort frame number 10
array = undistorter.undistort_frame(10, '10.png')
```

```
"""

Undistorting an array

"""

# undistort the image array returned (double undistortion)
array = undistorter.undistort_array(array, '10_2.png')


"""

Load some DLT coefficients, and uv coordinates, make a camera profile, then project
back and forth

"""

# set our distortion profile to another camera
dis.set_coefficients(model = 'GoPro Hero4 Black', mode = '1080p-120fps-narrow')
prof = dis.get_coefficients()

# get a camera profile (list of distortion profiles)
# here each distortion profile is the same, so we use this method
cam_pro = dis.get_cam_profile(ncams = 3)

# load som DLT coefficients
dlt = np.loadtxt('test-dlt-coefficients.csv', delimiter = ',').T

# load some points
points = pd.DataFrame.from_csv('pts.csv', index_col = False).as_matrix()

# undistort them
points[:,:2] = undistort_pts(points[:,:2], prof)
points[:,2:4] = undistort_pts(points[:,2:4], prof)
points[:,4:] = undistort_pts(points[:,4:], prof)

# project into 3D space
xyz = uv_to_xyz(points, cam_pro, dlt)


# project back into the image plane for camera 1
uv = dlt_inverse(dlt[0], xyz)

# plot the reprojection to see how good the mapping is
plt.scatter(uv[:,0], uv[:,1], c= 'b')
plt.scatter(points[:,0], points[:,1], c = 'r')
plt.show()

"""

Solve for our own DLT coefficients

"""

L1, er = solve_dlt(xyz, points[:,:2], prof)
L2, er = solve_dlt(xyz, points[:,2:4], prof)
L3, er = solve_dlt(xyz, points[:,4:], prof)
```

```
"""

Triangulate pixel coordinates with camera intrinsics

"""
# get the distorted points
points = pd.DataFrame.from_csv('pts.csv', index_col = False).as_matrix()

# triangulate
xyz = multiTriangulator(points, cam_pro)
xyz = xyz.triangulate()
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(xyz[:,0], xyz[:,1], xyz[:,2])

plt.show()

"""

Get reprojection errors

"""

errors = get_repo_errors(xyz, points, cam_pro, dlt)

fig = plt.figure()
ax = fig.add_subplot(111)

plt.plot(range(len(errors[0])), errors[0])
plt.show()


"""

Bootstrap to get confidence intervals, spline weights and tolerances

"""

CIs, weights, tols = bootstrapXYZs(points, errors, cam_pro, dlt)
```

CHAPTER 3

Classes

Argus includes some classes that are meant to be used by others and some that are only used by the GUI objects. This page is here to document the classes we intended to be used, how to instantiate them, and use their methods.

3.1 argus_gui.undistort.Undistorter

argus_gui.undistort.Undistorter (fnam = None, omni = None, coefficients = None)

Parameters:

- **fnam: str**
  - the filename of a movie to undistort (any format that OpenCV supports is ok)
- **omni: iterable**
  - Omnidirectional distortion coefficients (> 8 coefficients)
- **coefficients: iterable**
  - Pinhole distortion coefficients (8 in all)

Returns:

- Undistorter object

Methods:

Undistort a movie according to the supplied undistortion coefficients

- ***undistort_movie*** **(self, ofnam = None, frameint = 25, crf = 12, write = True, display = True, tmp_dir = None, crop = False):**
  - **ofnam: str**
    - the filename of a movie to write to
  - **framint: int**
    - FFMPEG parameter, gives the maximum number of frames between key frames.
  - **crf: int**
    - FFMPEG parameter, describes the level of compression. See FFMPEG documentation.
  - **write: bool**
    - whether or not to write when calling undistort_movie

  - **display: bool**
    - whether or not to display, using OpenCV, when calling undistort_movie
  - **tmp_dir: str**
    - temporary directory to write frames to. If not specified, one will be created. If specified, note it will be destroyed afterwards
  - **crop: bool**
    - whether or not to crop to approximately the undistorted region

- ***undistort_frame*** **(self, frame, ofile = None, crop = False):**
  - **frame: int**
    - positive integer frame number to undistort
  - **ofile: str**
    - if you wish to write the the undistorted frame to disk, specify a file name
  - **crop: bool**
    - whether or not to crop to approximately the undistorted region

- ***undistort_array*** **(self, array, ofile = None, crop = False):**
  - **array: numpy.ndarray**
    - image array
  - **ofile: str**
    - filename of the image you'd like the undistorted array written to. can be left None.
  - **crop: bool**
    - whether or not to crop to approximately the undistorted region

- ***set_movie*** **(self, fnam):**
  - **fnam: str**
    - filename of a movie (any format that OpenCV supports is ok)

3.2 argus_gui.undistort.DistortionProfile

argus_gui.undistort.DistortionProfile (model = None, mode = None):

Parameters:

- **model: str**
  - string specifying a camera model supported by Argus. For a comprehensive list of modes and models supported by Argus, visit argus.web.unc.edu
- **mode: str**
  - string specifying the shooting mode of the camera

Returns:

- DistortionProfile object

Methods:

Get a list of distortion coefficients. Also sets coefficients internal to the object:

- ***get_coefficients*** **(self, model = None, mode = None):**
  - **model: str**
    - string specifying a camera model supported by Argus.
  - **mode: str**
    - string specifying the shooting mode of the camera

If distortion coefficients are specified through get_coefficients, set_coefficients, or through __init__, then return an Undistorter object for movies, movie frames, and image arrays:

- *get_undistorter* (self):

Set coefficients from scratch:

- ***set_coefficients*** **(self, coefficients):**
  - **coefficients: 1-d iterable**
    - undistortion coefficients for a Pinhole or Omnidirectional distortion model

If the distortion coefficients are for an Omnidirectional model, attempts to create a PointUndistorter object from argus.ocam package, a dependency of argus_gui. Useful when undistorting a list of points with the Omnidirectional model:

- *get_ocam_undistorter* (self):

Return a row-homogenous array of length ncams. Equivalent to np.tile(get_coefficients, ncams). Useful for triangulation or DLT solving:

- ***get_cam_profile*** **(self, ncams):**
  - **ncams: int**
    - number of cameras in the profile

3.3 argus_gui.triangulate.multiTriangulator

argus_gui.triangulate.multiTriangulator (pts, cam_profile):

Paremeters:

- **pts: numpy.ndarray**
  - Nx(2 x k) array where k is the number of cameras and consequently the length of the cam_profile object provided
- **cam_profile: iterable**
  - a list containing arrays of Pinhole distortion coefficients (Omnidirectional coefficients are not currently supported) and camera intrinsics (focal length and optical center). These are the same coefficients returned by DistortionProfile. If you wish to use this function with omnidirectional coefficients, undistort the points first (using an object returned by DistortionProfile.get_ocam_undistorter) and pass in a camera profile with all zero distortion coefficients.

Returns:

- a multiTriangulator object

Methods:

- ***triangulate*** **(self):**
  - Returns an Nx3 of 3d coordinates in a coordinate system such that the origin is located at the last camera

CHAPTER 4

Stand-alone functions

All of the following function see use in the tutorial section of this documentation. We include them here for fast reference as well.

Project xyz coordinates back into the image plane of a camera given DLT coefficients:

dlt_inverse (L, xyz):

Parameters:

- **L: iterable**
  - 1x11 string or array of DLT coefficients.
- **xyz: numpy.ndarray**
  - Nx3 of 3d coordinates

Returns:

- Nx2 numpy array of ideal, undistorted pixel coordinates

Obtain slope and intercept of an epipolar line give a pixel marked in one camera and two sets of DLT coefficients (origin marked at lower left corner):

getDLTLine (u, v, L1, L2):

Parameters:

- **u: float**
  - x coordinate in the pixel plane of camera one
- **v: float**
  - y coordinate in the pixel plane of camera one
- **L1: iterable**
  - 1x11 array or list of DLT coefficients for the first camera
- **L2: iterable**
  - 1x11 array or list of DLT coefficients for the camera which the epipole would appear

Returns:

- m, b: slope and intercept as floats

Undistort with coefficients from a Pinhole or Omnidirectional model. Uses OpenCV:

undistort_pts (pts, prof):

Parameters:

- **pts: numpy.ndarray**
  - Nx2 array of pixel coordinates to be undistorted
- **prof: iterable**
  - 1-d array with camera intrinsics and distortion info

Returns:

- Nx2 array of undistorted pixel coordinates

Distort pixel coordinates using a Pinhole or Omnidirectional model. Uses OpenCV to project into the plane:

redistort_pts (pts, prof):

Parameters:

- **pts: numpy.ndarray**
  - Nx2 array of pixel coordinates to be distorted
- **prof: iterable**
  - 1-d array with camera intrinsics and distortion info

Returns:

- Nx2 array of distorted pixel coordinates

Normalize pixel coordinates by subtracting the optical center and dividing by the focal length:

normalize (pts, prof):

Parameters:

- **pts: numpy.ndarray**
  - Nx2 array of pixel coordinates to be normalized
- **prof: iterable**
  - 1-d array with camera intrinsics and distortion info

Returns:

- Nx2 array of normalized pixel coordinates

Solve for 11 DLT coefficients using the least-squared method:

solve_dlt (xyz, uv):

Parameters:

- **xyz: numpy.ndarray**
  - Nx3 array of 3d coordinates
- **uv: numpy.ndarray**
  - Nx2 array of marked pixel coordinates corresponding to the 3d points

Returns:

- L: 1x11 array of DLT coefficients
- rmse: Root mean squared error for reprojection

Use DLT coefficients to project marked pixel coordinates from multiple cameras into 3d space. Inverse function of *dlt_inverse*.

uv_to_xyz (pts, profs, dlt)

Parameters:

- **pts: numpy.ndarray**
  - Nx(2 x k) array where N is the number of frames, k the number of cameras
- **profs: iterable**
  - list or array of camera intrinsics. len(profs) == k
- **dlt: iterable**
  - list or array of DLT coefficients. len(dlt) == k. len(dlt[0]) == 11.

Returns:

- xyz: Nx3 array of xyz coordinates

Get reprojection errors for one or multiple tracked objects:

get_repo_errors (xyzs, pts, prof, dlt):

Parameters:

- **xyzs: numpy.ndarray**
  - Nx(3 x n) array where n is the number of tracked objects
- **pts: numpy.ndarray**
  - Nx(2 x k x n) array where N is the number of frames, k the number of cameras, and n the number of tracked objects
- **profs: iterable**
  - list or array of camera intrinsics. len(profs) == k
- **dlt: iterable**
  - list or array of DLT coefficients. len(dlt) == k. len(dlt[0]) == 11.

Returns:

- errors: n x N array of reprojection errors for all 3d points over all frames

Bootstrap for CIs, spline weights, and spline tolerances:

bootstrapXYZs (pts, rmses, prof, dlt, bsIter = 250, display_progress = False):

Parameters:

- **pts: numpy.ndarray**
  - an N*(2*k*n) array where k is the number of cameras, n is the number of tracks, and N is the number of frames
- **rmses: numpy.ndarray**
  - a N*k array that contains pre-computed rmses for the N frames. (transpose of what's returned by get_repo_errors)
- **prof: iterable**
  - an iterable with coefficients of Ocam undistorter objects, len(prof) == k

- **dlt: iterable**
  - an iterable with DLT coefficients, len(dlt) == k
- **bsIter: int**
  - number of iterations to bootstrap for
- **display_progress: bool**
  - whether or not to display print a progress bar

Returns:

- CI: Nx(6 x n) array of XYZ confidence intervals
- weights: N x n array of spline weights
- tol: 1d array of spline error tolerances

CHAPTER 5

Indices and tables

- genindex
- modindex
- search