Multiple-camera System Calibration Toolbox for Matlab

This is a toolbox for calibrating multiple-camera systems. The requirement of this toolbox is that two neighbor cameras in your system should be able to see some part of a calibration board at the same time. In general if the angle between the orientations of two neighbor cameras is no more than 90°, this toolbox can work well for your system.

The toolbox is related to the paper:

A Multiple-Camera System Calibration Toolbox Using A Feature Descriptor-Based Calibration Pattern, submitted to IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2013.

This toolbox also exploits some util codes from Bouguet's calibration toolbox

---

Contents

System requirements

The toolbox requires Matlab 2012b or higher version and can be used on Windows, Unix and Linux systems.

Quick start

Get your calibration pattern and take images

The toolbox detect SURF features as correspondences for calibration. You can generate a random pattern which is full of SURF features by using this provided Matlab function:

```
pattern = generatePattern(N, M);
```

where `N` and `M` are the height and width of the pattern in pixels.

After printing the pattern and pasting it on a flat board, you can take images of the pattern by your camera system. Your camera system should be well synchronized. Next move the pattern in front of each pair of neighbor cameras and take synchronized images. Make sure that each time both of neighbor cameras can see a sufficient large part of the pattern. For better calibration result, it is also recommended to take some images with the pattern in front of each one camera so that we have images with pattern covering every region of the camera view.

Images should be named in the form of `camera_index-time_stamp.extension`, e.g. `1-1234.png` or `3-2983.bmp`. The camera index is a 1-based integer and the time stamp is a

positive integer.

Calibrate your system

Run the `main` script to launch a simple command-line interface for the calibration. Follow each step of the interface.

1. Select the pattern image file you are using.
2. Resize the pattern if needed. If you use a large calibration pattern with very high resolution but the camera resolution is much lower, it is better to input a resized smaller pattern into the interface image. This can enhance the feature detection and matching performance. The input scale should be less or equal to 1.
3. Input number of the cameras.
4. Select camera models. You can select pinhole model for normal cameras, or catadioptric model for catadioptric, fisheye or wide-angle cameras.
5. Load images. Images should be named in the form of `camera_index-time_stamp.extension`. You can use `Shift` or `Ctrl` keys for multiple selection in the file selection dialog.
6. Calibration will then automatically launch.

An example

This is an example of calibrating a 5-camera rig. Download the images here and follow the following steps:

Run the calibration interface by calling `main` in Matlab. You will see:

```
>> main
------------------------------------------------------------------------
Multiple-Camera Calibration Toolbox
------------------------------------------------------------------------
### Load Pattern
Input the path of the pattern
```

Select the pattern image `pattern.png` in the file selection dialog. You will see something like this:

```
/home/li/Workspace/calibration-toolbox/image/pattern.png successfully
### Resize Pattern
Do you need to resize the pattern?
If the pattern resolution is very high,
suitable shrinking can help speed up and enhance the feature detectior
Input the scale ([] = no resize):
```

Since `pattern.png` has very high resolution, input `0.5` to resize it. Next you will see:

```
Resized to 50%
------------------------------------------------------------------------
### Camera Numbers
Input the number of cameras in the system:
```

Input 5 since we have 5 cameras in the system.

```
### Camera Type
Use pinhole (1) or catadioptric (2) model for Camera #1 (1/2)?
```

Select 2 since this data is from wide angle cameras.

```
Use the same model for the rest of the cameras (Y/N, []=Y)?
```

Press ENTER to use the same model for the rest 4 cameras.

```
------------------------------------------------------------------------
### Load Images
### Select images all together (should be named in form "cameraIndex-t
```

Multi-select all images in the file dialog. Then images will be automatically loaded. You will see the image loading info as follows:

```
Camera #1: Adding photo #129...
....Matches: 62
....Matches after Fundam. Check: 56
....Matches after smoothness Check: 53
....53 features kept
Camera #1: Adding photo #132...
....Matches: 61
....Matches after Fundam. Check: 47
....Matches after smoothness Check: 40
....40 features kept
Camera #1: Adding photo #140...
....Matches: 86
....Matches after Fundam. Check: 66
....Matches after smoothness Check: 54
....54 features kept
Camera #1: Adding photo #141...
....Matches: 93
....Matches after Fundam. Check: 70
....Matches after smoothness Check: 57
....57 features kept
...
...
```

The loading info displays how many valid features are detected from each image. Then the calibration procedure will be launched automatically. You will see a large bunch of process info. When the calibration is finished, you should see results like this:

```
### Calibration finished
------------------------------------------------------------------------
### Intrinsics:
Camera #1 :
....xi: [0.63694]
....Focal length: [665.1664, 664.6834]
....Aspect ratio: 0
....Principle Point: [417.1035, 226.2096]
....Distortion Coeff: [-0.28498, 0.090382, -0.00015087, -0.00051561]
```

```
Camera #2 :
....xi: [0.67678]
....Focal length: [691.8022, 687.3821]
....Aspect ratio: 0
....Principle Point: [424.1059, 249.1433]
....Distortion Coeff: [-0.30643, 0.10574, -0.0032633, -0.00061987]
Camera #3 :
....xi: [0.58413]
....Focal length: [648.1884, 646.17]
....Aspect ratio: 0
....Principle Point: [438.2655, 227.507]
....Distortion Coeff: [-0.29618, 0.088587, -0.0011176, -0.00028494]
Camera #4 :
....xi: [1.0159]
....Focal length: [811.819, 810.734]
....Aspect ratio: 0
....Principle Point: [438.5078, 231.2533]
....Distortion Coeff: [-0.21193, 0.09778, -0.0021793, -0.0031256]
Camera #5 :
....xi: [0.57423]
....Focal length: [640.0202, 639.4601]
....Aspect ratio: 0
....Principle Point: [423.131, 239.4429]
....Distortion Coeff: [-0.29568, 0.10104, -0.0013975, -0.0002087]
### Extrinsics:
Extrinsics:
Camera #1 :
1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1
Camera #2 :
  0.5046575  -0.004224369    -0.8633093     -316.6129
0.008001342      0.999968 -0.0002157935      37.75129
  0.8632825  -0.006798731     0.5046752     -166.9321
          0             0             0             1
Camera #3 :
-0.7012307     0.1420181     0.6986461      393.3241
0.04431023     0.9867457    -0.1561077     -24.16492
-0.7115561   -0.07851037    -0.6982292     -429.6906
         0             0             0             1
Camera #4 :
 0.3794783  -0.008424554     0.9251623      311.6048
-0.1016615     0.9935239    0.05074605       2.98553
-0.9195984    -0.1133104     0.3761643     -100.3085
         0             0             0             1
Camera #5 :
-0.7059149   -0.03452746    -0.7074546     -154.0953
 0.2324513     0.9321977    -0.2774415     -159.0963
 0.6690669    -0.3602989    -0.6500263     -516.6838
         0             0             0             1
Press enter to visualize camera poses plot and pose graph
```

Press ENTER key and you will see the 3D plot of camera poses and visualized pose graph. NOTE your result for this demo maybe different from the above result. This is due to the

sample images are not well synchronized yet.

APIs descriptions

The main part of the toolbox are encapsulated in several Matlab objects. You can easily use the object API to build your own calibration program or camera model in your computer vision applications.

`CameraBase < handle & matlab.mixin.Heterogeneous`

Base class for camera model.

- **`nParams`** Number of intrinsic parameters.
- **`obj = CameraBase(width, height)`** Constructor.
- **`p = toParamVector(obj)`** Convert intrinsic parameters to a vector.
- **`fromParamVector(obj, p)`** Load intrinsic parameters from a vector.
- **`[x, jacobPose, jacobCamera] = projectPoints(obj, points, rvec, tvec)`** Project 3D points to image points. `points` should be `3×N`. `jacobPose` is `2N×6`. `jacobCamera` is `2N×nParams`.
- **`undist = undistort(obj, raw, focal, width, height)`** Undistort a given raw image. The undistorted image is `height×width`, with `focal` as the focal length.
- **`outputIntrinsics(obj, fileName)`** Output intrinsics parameters to a XML file or to the console if the `fileName` is omitted.
- **`loadIntrinsics(obj, fileName)`** Load intrinsics parameters from a XML file.

`PinholeCamera < CameraBase`

Camera class for pinhole camera model.

`CataCamera < CameraBase`

Camera class for generic camera model, using the catadioptric model.

`CameraCalibrationBase < handle & matlab.mixin.Heterogeneous`

Base class for single camera calibration.

- **`obj = CameraCalibrationBase(width, height, pattern)`** Constructor. `width` and `height` are the camera resolution. `pattern` is the pattern image.
- **`[photoIndex, valid] = addPhoto(obj, photo)`** Add a photo of the pattern. return a index and a boolean if the photo is good.
- **`p = getPoseVector(obj, photoIndex)`** Put the pattern pose for photo `photoIndex` in a `6×1` vector.
- **`setPoseVector(obj, p, photoIndex)`** Load the pattern pose for photo `photoIndex` from a vector.
- **`calibrate(obj)`** Calibrate the camera.
- **`[error, jacobPose, jacobCamera] = computeReprojError(obj, photoIndex)`** Compute the reprojection error of one photo.
- **`render = reprojectPattern(obj, photoIndex)`** Reproject the pattern from a photo to the pattern plane.
- **`showPoints(obj)`** Show all detected features from different photo together. This can be used to check if the features are uniform distributed.

`PinholeCameraCalibration < CameraCalibrationBase`

Class implementation for calibrating a pinhole camera.

`CataCameraCalibration < CameraCalibrationBase`

Class implementation for calibrating a generic camera.

`CameraSystemCalibration < handle`

Class implementation for calibrating a camera system.

- **`obj = CameraSystemCalibration(nCameras, pattern)`** Constructor. `pattern` is the pattern image.
- **`setCameraType(obj, cameraIndices, type, width, height)`** Set camera type. Camera indices are 1-based index. You can set several cameras as the same type together.
- **`addPhoto(obj, cameraIndex, photo, key)`** Add a photo for a camera. `key` is the string timestamp.
- **`calibrate(obj)`** Calibrate the system after adding all photos.
- **`visualizeObjects(obj)`** Visualize the camera poses and pattern poses.
- **`visualizeGraph(obj)`** Visualize the pose graph.

Reference

Please see our paper for references: A Multiple-Camera System Calibration Toolbox Using A Feature Descriptor-Based Calibration Pattern, submitted to IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2013.