Calib Camera Calibrator

by Calib.io I/S

August 18, 2020

Aim and purpose

This manual covers the Calib Camera Calibrator graphical software for camera calibration. It provides both general information on sofware functionality as well as some background information on camera calibration. This information should enable users to efficiently navigate the software and interpret calibration results.

For more information on calibration and best practices, visit our Knowledge Base at calib.io. For technical support (if covered by your license), email info@calib.io.

Contents

1 Overview

The Calib Camera Calibrator software is a complete graphical software package for camera calibration. It includes unique features for robust detection, subpixel refinement, and bundle block adjustment for N-camera, M-target camera calibration setups.

Figure 1 shows a complete stereo calibration scenario in which uses a large ChAruCo board.

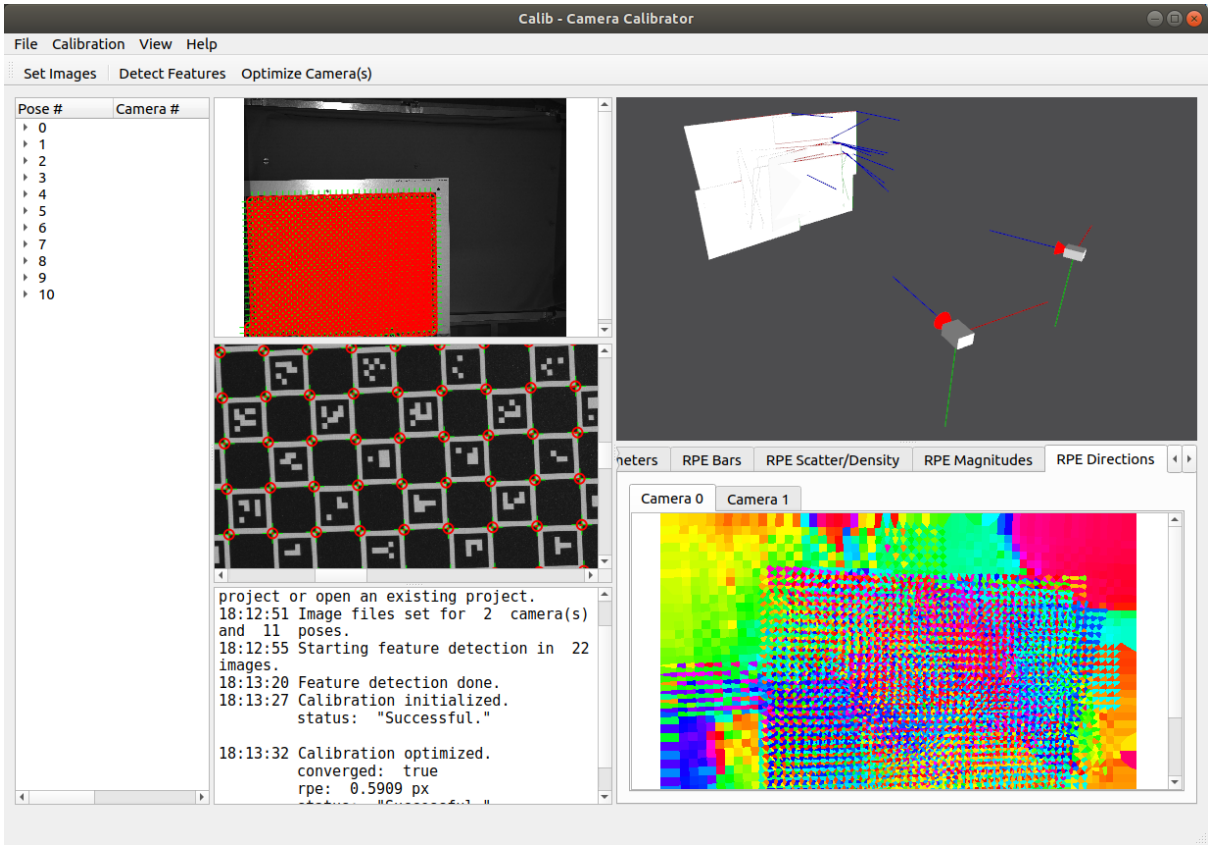

Figure 1: Overview of the Calibrator software GUI.

Two cameras were jointly optimized. The log window indicates that calibration was successful with an overall RMS of reprojection errors of 0.59px. Whether this is an acceptable or good value depends heavily on the camera/lens combination and accuracy requirements. In this case, the camera has a small, high-resolution sensor. A good setup will use a camera model that accurately and sufficiently describes the lens/sensor geometry while not overfitting to the calibration data.

Calibrator shows a number of valuable visualizations which aid in determining if the calibration was truly as accurate and un-biased as possible. This is realized by inspecting reprojection statistics, both visually and numerically to discover if any biases are left in the data.

1.1 Licensing

Calibrator must be licensed and activated via internet using a license key. A license may be transferred to another computer. This is done by de-activating on the first machine and then using the same license key on another machine. Please refer to the EULA (end user license agreement) for details regarding rights to transfer licenses and commercial use (using the software to calibrate devices for sale to third parties).

1.2 Application settings

Calibrator uses a human-readable and editable configuration file, calibrator.ini.

On most Linux variants, this file will be located in

`~/.config/Calib.io/calibrator.conf`

On Windows 10, calibrator.ini is located in

`C:\Users\<user>\AppData\Roaming\Calib.io\calibrator.ini`

, or equivalently if the system was installed on another drive.

The configuration file/registry folder can safely be deleted, which will reset Calibrator application settings to default values. Since values are read dynamically, it is also possible to edit them programmatically, enabling automation workflows.

1.3 Saving/loading projects

Calibrator allows to store the intermediate results image references and feature detections of a calibration. This can be done from the ”File” menu. These project files are written in human-readable JSON format, and can be edited/altered and combined with other software/programming work-flows.

1.4 Exporting calibration parameters

In order to use the results of a camera calibration, the final parameter estimates will usually be exported for further processing or integration into a software pipeline. Calibrator can export the ”calibration configuration” and the parameter estimates. This can be done from the ”Calibration” menu item. These files are written in human-readable JSON format, which are easily loaded into many programming environments (e.g. C++, Matlab, Python).

2 Method

Camera calibration with Calibrator is based on the principle of bundle block adjustment. The first step in this process is to find a suitable initialization (initial guess), since the following optimization will only properly converge (find the best solution) if it is started in the vicinity of it. This is then followed by an automatic optimization over all parameters.

2.1 Initialization

The initialization step is crucial, as the following optimization is dependent on a good starting guess. In Calibrator, initialization is allways performed right before bundle adjustment, and this step is usually completed after only a few seconds.

A unique feature of Calibrator is that it is able to initialize multi-camera and multi-target setups. That is, there is no theoretical limit to the number of cameras or calibration targets. This is realized by formulating all connections/links between cameras, poses and calibration targets. If all are connected by rigid transformations, Calibrator can determine suitable initial guesses for the orientation of each target, at each pose in each camera.

2.2 Optimization

The optimization step performs numerical optimization, which aims to reduce the value of an error function. Calibrator uses the sum of squared reprojection errors:

$$E = \sum_i \sum_j ||\vec{p}_{ij} - \breve{\vec{p}}(\vec{P}_j, \boldsymbol{A}, \vec{k}, \boldsymbol{R}_i, \vec{T}_i)||^2 \quad ,$$

where $\breve{\vec{p}}$ is the projection operator determining 2-D point coordinates given 3-D coordinates and the camera parameters. i sums over the positions of the calibration board and j over the points in a single position. $\vec{P}_j$ are 3-D point coordinates in the local calibration object coordinate system, (for flat calibration targets, $\vec{P}_j = [x, y, 0]^\top$), and $\vec{p}_{ij}$ the observed 2-D coordinates in the camera. The per-position extrinsics, $\boldsymbol{R}_i, \vec{T}_i$, can be understood as the position of the camera relative to the coordinate system defined by the calibration object.

From the value of the sum of squared reprojection errors, we can easily obtain the RMS of the reprojection errors:

$$\mathrm{RMS} = \sqrt{E/N} \quad ,$$

where N is the total number of observed feature points in all observations. N can be equal to "$i \cdot j$" when all features are observed in all poses for all cameras, but this is not necessarily the case.

3 Set Images Dialog

The ”Set Images” dialog allow the user to import camera images to perform a new calibration (see Figure 2). This stage also informs the software about which individual images of a multi-camera setup belong together (were taken with the same pose).

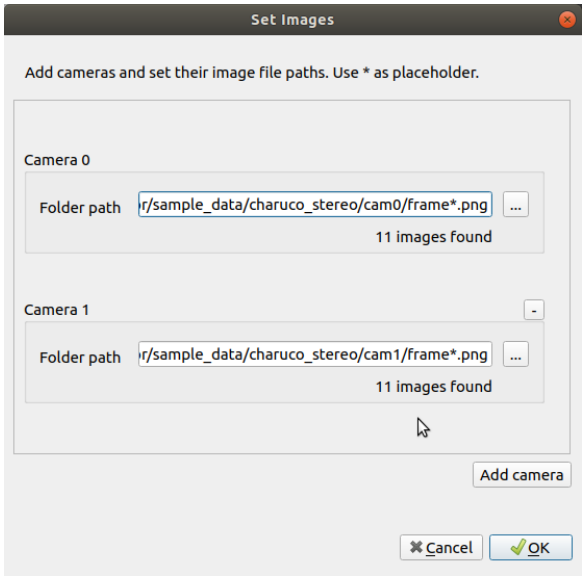

Figure 2: The ”Set Images” dialog.

This stage supports various organizations of files, e.g. images of multiple cameras in the same folder, etc. An image folder can be browsed using the ”...” button, but the resulting path can be adjusted to user’s needs by using the wildcard ”*” character. For instance, to associate only PNG filenames containting the word ”camera_Left”, the path can be altered to ”path/*camera_Left*.png”.

It is important to note that multi-camera setups always need an equal amount of images, and that the indexed order for each camera must be coherent. Only then will corresponding images be imported into the software and assigned the same Pose # after feature detection.

4 Detect Features Dialog

The detection dialog lets a user configure the feature detection step of a camera calibration.

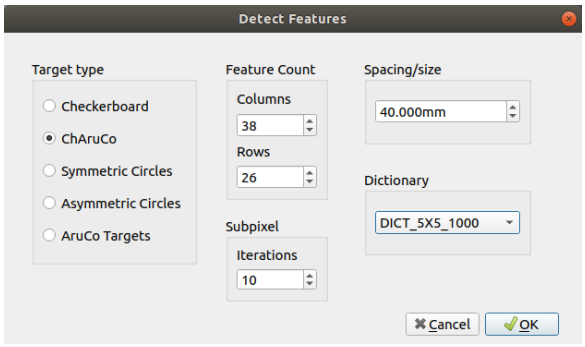

Figure 3: The feature detection dialog.

4.1 Target Type

The type of pattern which was used during calibration.

4.2 Feature Count

For chessboard or ChArUco targets, this is the number of saddle points (not checker fields). For circle targets, this is the number of rows/colums of circles.

4.3 Subpixel

Calibrator performs subpixel feature optimization using optimized, proprietary algorithms. These typically converge after 5-6 iterations. It is only necessary to use a lower setting if computation time is a concern.

4.4 Spacing/size

This should be the spacing between dots on symmetrical circles targets, diagonal spacing on asymmetrical circles targets, and the width of a single square on checker and ChArUco targets. For Calib.io's AruCo targets, it is the spacing between saddle points.

4.5 Dictionary

The dictionary setting is relevant for ChArUco targets and should correspond to that used on the calibration target. Note that the last number indicates the dictionary size. This should be chosen as low as possible, while still containing all unique marker ids used. E.g. for a 40x20 field ChArUco board, there are 400 unique markers, and a dictionary of size 500 or 1000 should be chosen.

5 Optimization Dialog

The detection dialog lets a user configure the feature detection step of a camera calibration. See Figure 4 for an example.

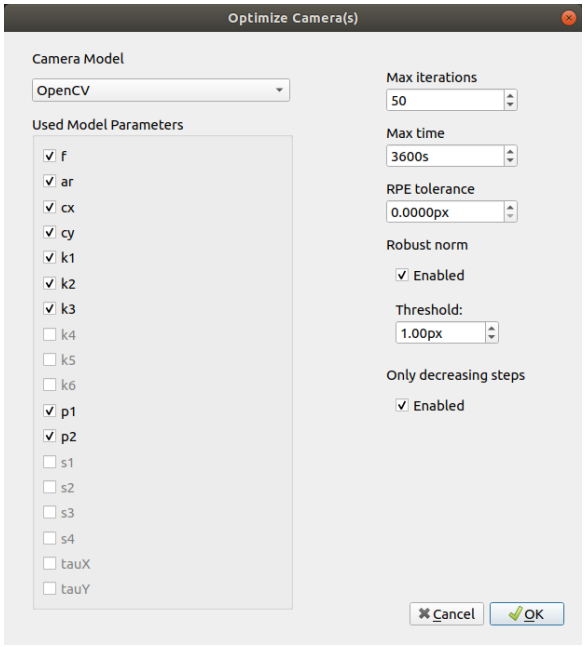

Figure 4: The optimization dialog.

In this dialog, the following settings can be altered.

- Camera model with a choice of popular camera models. Individual parameters can then be included/excluded from the model.
- Max iterations: the optimizer will terminate if this number is reached, or if the RPE changes less than RPE tolerance.
- Max time: the optimizer will terminate if this amount of seconds passes since beginning.
- RPE tolerance: the optimizer will terminate if the RPE changes less than this value between two consecutive iterations.
- Robust norm: uses a Huber norm during optimization, which makes the process much less sensitive to outliers in the data. For individual errors above the ”Threshold” parameter, a linear loss is applied instead of a quadratic one. This setting is recommended for difficult cases only (i.e. low resolution thermal camera calibration).
- Only decreasing steps: by unchecking this setting, the optimizer is allowed to explore the optimization landscape a little more, potentially escaping weak local minima. Should be disabled in difficult calibration setups.

6 Analysis

Calibrator gives you strong insights into the results of a calibration. After running optimization, all results are presented in the right-most part of the GUI.

6.1 Convergence

Shows the RMS of the norms of all reprojection errors (across all cameras, poses and targets) as a function of iteration number. By hovering the mouse cursor over individual data points, the individual values are shown.

6.2 Initialization

Shows the results of the linear initialization of parameters. Initialization is based on Zhang's method with extensions and improvements. Note that standard errors cannot be estimated at the initialization stage, which is why the "+/-" fields are zero.

6.3 Parameters

Similar to the data in "Initialization", the "Parameters" tab shows the final, optimized camera parameters along with the standard deviations of these estimates. Inspecting these standard deviations is a valuable tool for camera model selection. In general, standard deviations in the order of more than a few percent of the parameter value indicate that the camera model is not well-constrained. More images need to be taken, or a simpler camera-model used.

6.4 RPE Bars

Graphically shows the per-pose RPE. By hovering the mouse cursor over individual data points, the individual values are shown.

6.5 RPE Scatter/Density

The plots in this tab are organized according to camera (if more than one camera is part of the calibration). The raw scatter-plot shows all individual error vectors. With the assumption of normally distributed noise (with mean zero and zero co-variance) the distribution of points should resemble a Gaussian bell-curve.

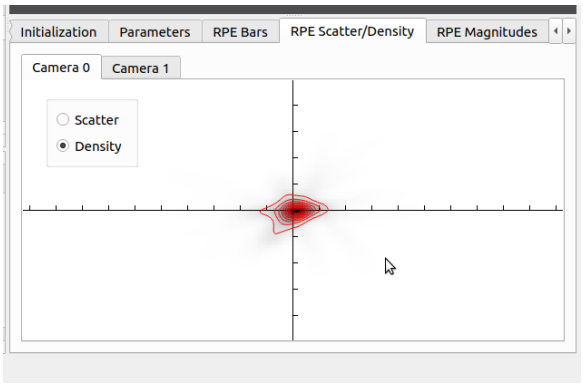

Figure 5: Scatter plot and kernel density estimate plot. In this case, the error distribution is quite skewed. Alternative (richer) camera models should be investigated.

By switching to the "Density" view, a kernel-density-estimate (with Gaussian kernel) is shown (see Figure 5), which makes it very easy to judge if the distribution of errors is truly close to normal. Note that the "Density" view also shows iso-contours for the distribution at iso-probability-levels 0.1, 0.2, 0.3 ... 0.9.

6.6 RPE Magnitudes

This is a graphical representation based on the following. For each reprojected point, the image area for which this point is the closest is determined (a so-called Voronoi diagram). The coloring is based on the magnitude of the RPE for that particular point (the 95% percentile of RPEs is completely white). This plot both shows if there is any spatial correlation between error and image location (cell coloring should be diffuse and chaotic), and also shows wether image features were evenly distributed across the image sensor plane (cells should ideally have uniform sizes).

6.7 RPE Directions

The visualization is similar to ”RPE Magnitudes”, but shows the color-coded directions of errors instead of their magnitudes. This coloring should also be diffuse and chaotic, indicating no residual tendencies in the errors. An example of this is shown in Figure 6.

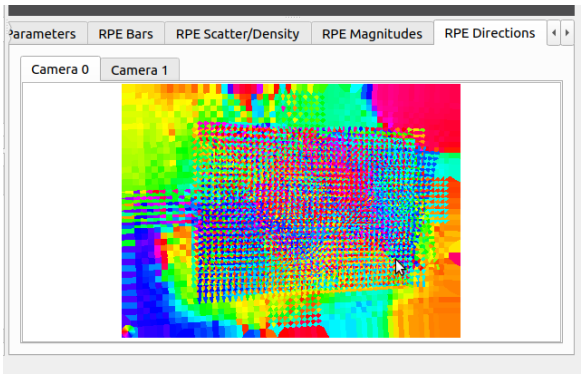

Figure 6: RPE directions plot.

6.8 Statistics

The statistics tab shows both raw numerical statistics (mean, median and rms RPE errors) for the ensemble of errors and also for the individual cameras, poses and features. These numbers allow the user to detect issues with one of the cameras, data recording (if one pose has higher RPE) or with the calibration board (if a particual feature has higher RPE).

In addition, the tab shows the Anderson-Darling statistic for zero mean normality of the error components (x and y). A value close to zero indicates a high probability that data is indeed normally distributed.

7 Using parameters in OpenCV

In order to use the estimated camera parameters in OpenCV, the following function can be used to load parameter estimates (with camera model 'OpenCV'). The function below can also be easily adapted to load other camera models into C++ programs. The code uses a single-header, permissive license JSON library by Github user nlohmann for file import.

```
// Read Calib Camera Calibrator parameters into OpenCV datastructures, suitable
// for further use with OpenCV function. Uses nlohmann's single-header "JSON for
// Modern C++" for JSON import. See https://github.com/nlohmann/json

#include <opencv2/opencv.hpp>
#include "json.hpp"

bool readCalibParameters(const std::string &filePath,
                         std::vector<cv::Matx33d> &K,
                         std::vector<cv::Vec<double, 14>> &k,
                         std::vector<cv::Vec3d> &rvec,
                         std::vector<cv::Vec3d> &tvec) {

  std::ifstream fileStream(filePath);
  nlohmann::json jsonStruct;
  try {
    fileStream >> jsonStruct;
  } catch (...) {
    return false;
  }

  assert(jsonStruct["Calibration"]["cameras"][0]["model"]["polymorphic_name"] ==
         "libCalib::CameraModelOpenCV");

  int nCameras = jsonStruct["Calibration"]["cameras"].size();

  if (nCameras < 1) {
    return false;
  }

  K.resize(nCameras);
  k.resize(nCameras);

  rvec.resize(nCameras);
  tvec.resize(nCameras);

  for (int i = 0; i < nCameras; ++i) {
    auto intrinsics = jsonStruct["Calibration"]["cameras"][i]["model"]
                                ["ptr_wrapper"]["data"]["parameters"];

    double f = intrinsics["f"]["val"];
    double ar = intrinsics["ar"]["val"];
    double cx = intrinsics["cx"]["val"];
    double cy = intrinsics["cy"]["val"];
    double k1 = intrinsics["k1"]["val"];
    double k2 = intrinsics["k2"]["val"];
    double k3 = intrinsics["k3"]["val"];
    double k4 = intrinsics["k4"]["val"];
    double k5 = intrinsics["k5"]["val"];
    double k6 = intrinsics["k6"]["val"];
    double p1 = intrinsics["p1"]["val"];
    double p2 = intrinsics["p2"]["val"];
    double s1 = intrinsics["s1"]["val"];
    double s2 = intrinsics["s2"]["val"];
    double s3 = intrinsics["s3"]["val"];
    double s4 = intrinsics["s4"]["val"];
    double tauX = intrinsics["tauX"]["val"];
    double tauY = intrinsics["tauY"]["val"];

    K[i] = cv::Matx33d(f, 0.0, cx, 0.0, f * ar, cy, 0.0, 0.0, 1.0);
    k[i] = cv::Vec<double, 14>(k1, k2, p1, p2, k3, k4, k5, k6, s1, s2, s3, s4,
                               tauX, tauY);

    auto transform = jsonStruct["Calibration"]["cameras"][i]["transform"];
```

```
    auto q = transform["rotation"];
    double qw = q["w"];
    double qx = q["x"];
    double qy = q["y"];
    double qz = q["z"];

    double angle = 2 * acos(qw);
    double x = qx / sqrt(1 - qw * qw);
    double y = qy / sqrt(1 - qw * qw);
    double z = qz / sqrt(1 - qw * qw);

    if (std::abs(angle) < std::numeric_limits<double>::epsilon()) {
      rvec[i] = cv::Vec3d(0.0, 0.0, 0.0);
    } else {
      rvec[i] = angle * cv::Vec3d(x, y, z);
    }

    auto t = transform["translation"];
    tvec[i] = cv::Vec3d(t["x"], t["y"], t["z"]);
  }

  return true;
}

int main() {
  std::string jsonFilePath =
      std::string(SRCDIR) + "/../data/opencv_parameters.json";

  std::vector<cv::Matx33d> K;
  std::vector<cv::Vec<double, 14>> k;
  std::vector<cv::Vec3d> rvec;
  std::vector<cv::Vec3d> tvec;

  readCalibParameters(jsonFilePath, K, k, rvec, tvec);

  return 0;
}
```

8 Using parameters in Matlab

In order to use the estimated camera parameters in Matlab, the following function can be used to load parameter estimates (with camera model 'Matlab') into Matlab. This camera model corresponds directly to the one used in Matlab's Computer Vision Toolbox, and the parameters also match those used in the 'Camera Calibration Toolbox for Matlab' by Caltech. The source code for this function can be found in the installation directory.

```
%readCalibParameters(fileName) reads a json parameter file generated with
% Calib Camera Calibrator (v. 1.2.7).
%
% The resulting structs can either be used directly or
% cameraIntrinsics()/cameraParameters()/stereoParameters()
% objects can be created using them if CV Toolbox is available.
%
% (c) Calib.io I/S
%
% Note: coordinates are adjusted to Matlab's 1-based index convention

function [intrinsics, extrinsics] = readCalibParameters(fileName)

jsonStruct = jsondecode(fileread(fileName));

nCameras = length(jsonStruct.Calibration.cameras);

assert(nCameras > 0)
assert(strcmp(jsonStruct.Calibration.cameras(1).model.polymorphic_name, 'libCalib::
    CameraModelMatlab'));

for i=1:nCameras

   h = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.CameraModelCRT.CameraModelBase
       .imageHeight;
   w = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.CameraModelCRT.CameraModelBase
       .imageWidth;

   f = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.f.val;
   ar = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.ar.val;
   cx = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.cx.val;
   cy = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.cy.val;
   k1 = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.k1.val;
   k2 = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.k2.val;
   k3 = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.k3.val;
   p1 = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.p1.val;
   p2 = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.p2.val;
   skew = jsonStruct.Calibration.cameras(i).model.ptr_wrapper.data.parameters.skew.val;

   intrinsics(i).RadialDistortion = [k1 k2 k3];
   intrinsics(i).TangentialDistortion =  [p1 p2];
   intrinsics(i).ImageSize = [w h];
   intrinsics(i).IntrinsicMatrix = [f, 0, 0; skew, f*ar, 0; cx+1, cy+1, 1.0];

   extrinsics(i).RotationVector = quaternionToAxisAngle(jsonStruct.Calibration.cameras(i).
       transform.rotation);
   extrinsics(i).TranslationVector = cell2mat(struct2cell(jsonStruct.Calibration.cameras(i).
       transform.translation));

end

end


function aa = quaternionToAxisAngle(q)

  angle = 2 * acos(q.w);
  x = q.x / sqrt(1-q.w*q.w);
  y = q.y / sqrt(1-q.w*q.w);
  z = q.z / sqrt(1-q.w*q.w);

  aa = angle * [x,y,z];

end
```