Camera grid extrinsic self-calibration

Tim Lenertz

July 28, 2017

Contents

1 Introduction

This report describes the method used to calibrate the extrinsic camera parameters of the 3DLicorneA data sets. If is applicable for dense 2D data sets, where cameras are placed on a more or less regular grid on a plane parallel to the scene. The optical flow of tracked features, and aggregated values from the depth maps, are used to compute the camera positions and orientations. No calibration pattern needs to be present in the scene.

1.1 Requirements

To use the method, there are the following requirements:

- The intrinsic camera parameters are known. This is the camera matrix $\mathbf{K}$ and optionally the distortion coefficients. The method was used for the case where there is no distortion (Kinect v2), but with additional steps it can be used with distorted images.
- The intrinsic camera parameters are the same for each image.
- There can be a depth map for all or some views. If not, the depth values for some features (at least one) needs to be specified manually.
- The camera centers are arranged on an approximately regular 2D grid on a frontal plane P. The distance between adjacent camera positions (in x and y direction) is sufficiently small that feature points can be tracked using optical flow.
- It is assumed that camera centers lie always exactly on the plane. That is, they never more towards or back from the scene.
- The camera is facing approximately perpendicular to the plane P, towards the scene. There can be a small rotation $\mathbf{R}$ of the camera relative to P. It is estimated as part of the calibration. The yaw, pitch and roll angles should be smaller than 5°. It is assumed that this rotation remains constant for all views.
- There can be some missing images and depth maps in the data set.
- It is not necessary that the tracked feature points remain visible across the whole range of views. Calibration can be done on subsets of the camera positions, and then stitched together.

2 Method

The calibration is done in four steps: (1) Compute *image correspondences* using feature tracking. (2) Estimate the *camera rotation* $\mathbf{R}$. (3) Estimate *straight depths* of the tracked features, i.e. their distance to P. (4) Deduce camera positions.

The method calculates one global rotation matrix $\mathbf{R}$, and for each view v, a 2D vector $\vec{c_v}$ of the camera center position on the plane P. From this it then obtains the extrinsic matrices $\mathbf{Rt}$.

2.1 Overview

First the algorithm selects several *feature points* on one on multiple *reference views*. Using optical flow, it then tracks the position of the those features on the other views.

With the pin-hole camera model, it is be possible to calculate the camera position on P directly from a feature point's positions on different views - when the camera is pointing perpendicular to P, and the feature's distance to P is known. So the algorithm first needs to estimate $\mathbf{R}$, and the distances of the features.

To calculate $\mathbf{R}$, two methods are can be used. One uses a non-linear model which estimates $\mathbf{R}$ only from the slope of the lines that the tracked features make when the camera moves horizontally and vertically, without knowledge of the features' depths. The other method uses the depths of the features on the different views. Both estimate a full 3D rotation matrix.

The orthogonal distance of a feature to P will be called its *straight depth*. Knowing $\mathbf{R}$ (and $\mathbf{K}$), it can now be calculated from the feature points' depths in each view's depth map. If depth maps are not available, it is also possible to fix only the depth of one or more features, and deduce the rest from the relative scales of the different feature's disparities. These two methods are available.

Using $\mathbf{R}$ and the *straight depth* of each feature, the algorithm now estimates the set of camera positions on P, one for each feature. They are averaged to get one camera position for each view.

2.2 Preliminaries

The 2D dataset consists of several *views*. A view v consists of an image, and optionally a depth map, taken from one camera position. The views are enumerated with two integer indices $v(x, y)$. Views with the same x index are (approximately) aligned vertically, views with the same y index horizontally.

The goal is to estimate the position and orientation of the camera for each view, i.e. to find the extrinsic camera matrices $\mathbf{Rt}_v$.

If depth maps are used, they need to be in the same coordinate systems as the images. For each image pixel (i_x, i_y), the value d of the same pixel in the depth map needs to indicate the distance from the camera center to that object point, perpendicular to the camera image plane. The camera matrices $\mathbf{Rt}_v$ will be expressed in the same unit as these depths.

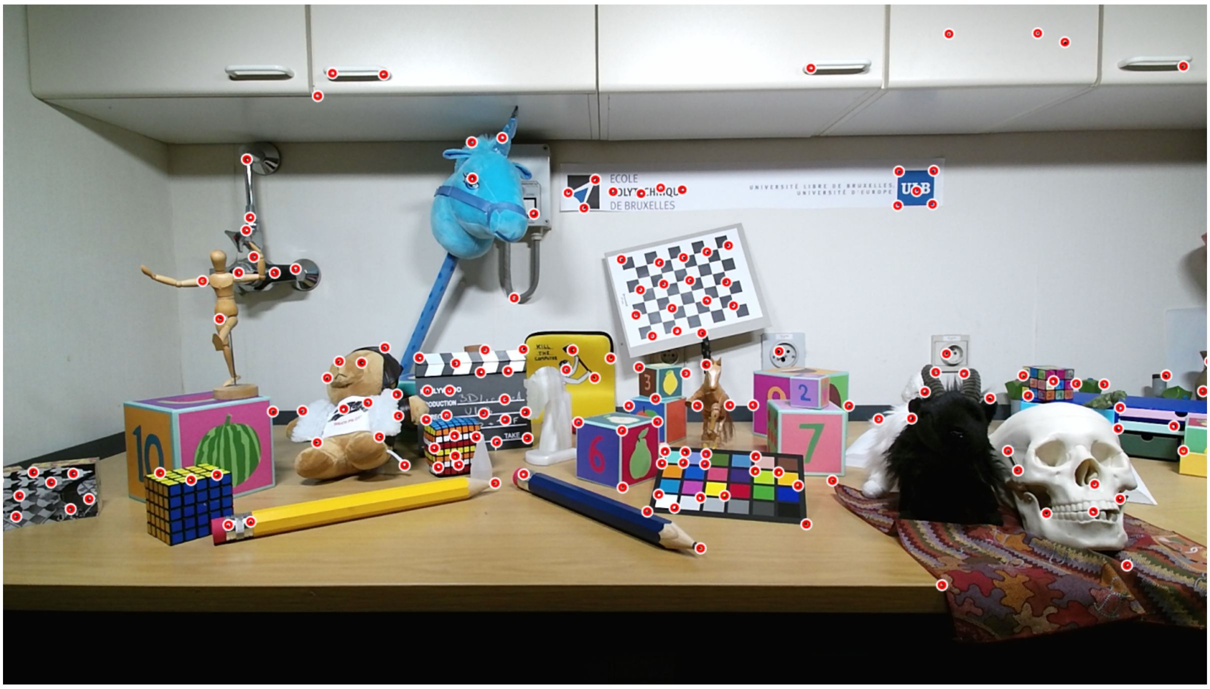

Figure 2.1: Chosen feature points

2.3 Image correspondences

First some features f are selected. They correspond to 3D points in the scene. This step aims to find for each feature f, the set of *feature points* $p(f, v) = (x, y, d, w)$, that is the pixel coordinates x, y where the feature f is visible in each view v. This data is called the *image correspondences*.

A feature point optionally also contains its orthogonal depth d, and a weight w. When $\mathbf{R} \neq \mathbf{I}$, the depths $p_{f,*}$ of the same feature for different views will be different.

2.3.1 Choosing feature points

Features are obtained by choosing feature points on a *reference view*. By default the center view in the dataset is used as reference view.

But if the range of motion of the camera is large, or the field of view is small, many feature points that are visible in the center view, will not be visible on the extremity views. For this reason, it is also possible to select features on multiple *reference views*. The entire procedure will then be done for each reference view independently, and in the end the obtained camera positions are stitched together.

The chosen feature points need to be such that they are likely to remain *stable* when doing feature tracking. It means that when one looks for a similar-looking nearby point on an adjacent view, it is likely to be the same scene point. An example is shown on figure 2.1.

The *OpenCV* function `goodFeaturesToTrack` is used. Also, the image is first subdivided into 4 or more rectangular regions, and the best chosen features from each region are taken.

The chosen features should be well distributed across the image, and have different depths. There should be about 300 or more features, considering that many will be filtered out because their optical flow is unstable.

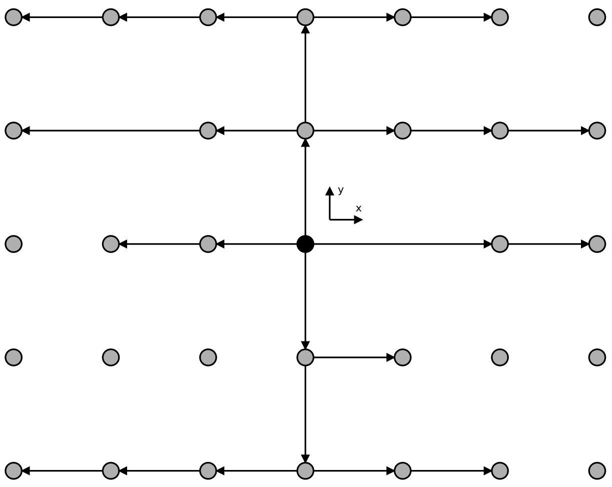

Figure 2.2: Optical flow paths

2.3.2 Optical flow tracking

Optical flow feature tracking is always done on adjacent views, for example $v(x, y)$ and $v(x+1, y)$. Then sequentially, it uses the corresponding feature points on $v(x + 1, y)$ to estimate those for $v(x + 2, y)$, and so on. So there is an error accumulation, which gets worse the longer the path that the view indices take.

The acquisition system moves line-by-line. So it is physically guaranteed that for any $v(x, y)$ and $v(x + 1, y)$, the camera only moves by a small amount, whereas for $v(x, y)$ and $v(x, y + 1)$, there can be a larger deviation. So it is better to take most optical flow correspondences in x direction.

The optical flow algorithm moves over the (x, y) view indices as shown on figure 2.2. The center view (black circle) is the reference view. As each step moving from (x, y) to (x', y'), for each feature f, all the feature points $p(f, v(x', y'))$ are computed from those of $p(f, v(x, y))$. If no feature point $p(f, v(x', y'))$ could be computed anymore, the algorithm stops for that line.

If the image for a view is missing, that view is skipped, and instead the correspondences are taken from the second-previous view, as shown. It is important that no view is missing in the column of the reference view, because then that entire line will be skipped.

The reference view is not one of the edges, but instead in the center, and the optical flow steps are done in all four directions. This minimizes the total path taken, and reduces the accumulated error.

The maximal number of steps to be taken in x and in y direction, called *outreach*, can be set to a maximal limit. Using a smaller *outreach*, and instead doing the calibration from multiple reference views, and combining the results in the end, can produce better final results.

To compute the optical flow, the *OpenCV* function `calcOpticalFlowPyrLK` is used. The parameters can be adjusted. It can also be set to automatically generate multi-scale image pyramids. Thay way larger images can be used.

Figures 2.3 are examples of good feature correspondences. The background image is a close-up of a view image. The red dot is the feature point on this view. The other dots are the corresponding feature points on the other views.

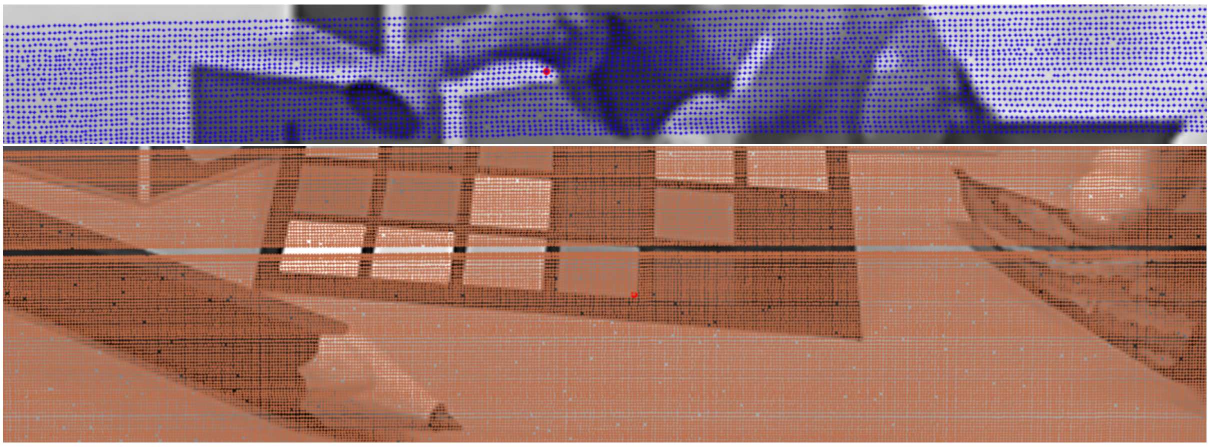

Figure 2.3: Good feature correspondences

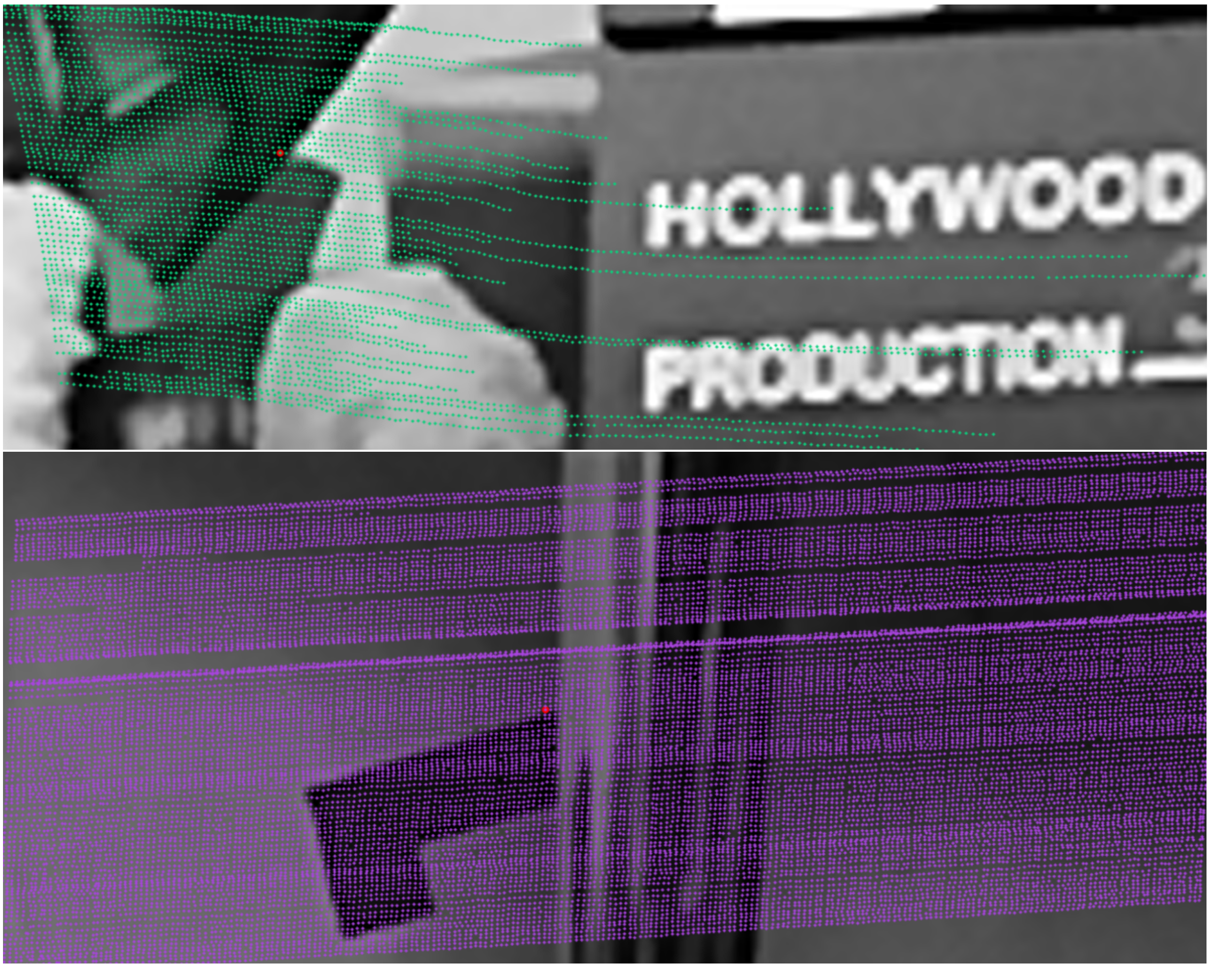

Figure 2.4: Bad feature correspondences

2.3.3 Filtering features

The next step is to filter the generated *image correspondences*. It is important, because incorrect feature points can have a large impact on the final results. The optical flow procedure tends to generate a large number of bad correspondences. There is an algorithm to automatically filter out bad correspondences, but they should also be verified by hand.

Figures 2.4 are examples of good feature correspondences. In the first example, the deviation occurred because a foreground object with a curved border moved in front of the tracked feature. In the second example, the pattern appears regular but the correspondence is still incorrect. Deviations can also occur because of specular reflections (for example on the metal sink), because of badly chosen feature (such as on the furry objects), or because of the limited pixel resolution.

The filtering algorithm removes all feature points for one a feature f entirely, if there are too little feature points, or if the pattern deviates too much from a regular lattice.

Properly filtering the correspondences is important: Having incorrect correspondences, and having too little correspondences, both have a large impact on the final result. In practice, for each view, there should remain about 100 features.

2.3.4 Reference views

As explained, it is possible to use multiple *reference views*. They must be chosen on a grid called the the *reference grid* (or *reference views grid*).

Figure 2.5 shows how the reference views can be selected. The four colored dots are the reference views. The *x key* and *y key* is the distance between reference view indices, in x and y direction.

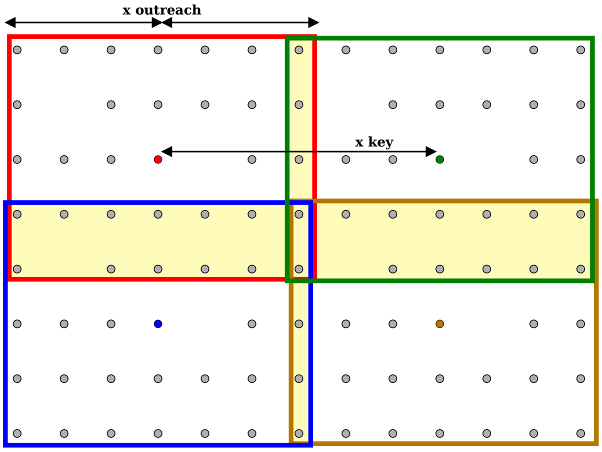

Figure 2.5: Reference views arrangement

The *outreach* indicates the maximal range of view indices around the *reference view*, for which correspondences will be searched. In the figure these regions are shown as the colored rectangles. For the stitching to work, there must be some overlap in these rectangles, as shown by the yellow regions in the figure.

The overlap should be large enough so that there will remain many views with feature points from two references, after the filtering. The *outreach* should be kept relatively small, for two reasons: (1) The maximal propagated error from the optical flow feature tracking is larger if the outreach is large. This also makes it more likely for entire features to be filtered out, even though a smaller portion of its feature points are good. (2) When camera positions are computed at the end, the impact of the error in the camera rotation and in the feature depths, becomes much larger for views that are further away from the reference view. (see later, in section 2.7)

For example an *outreach* of 80, and a *key* of 100 can be a good choice. It would leave an overlap range of 60. Generally, the *outreach* should be small and the *overlap* large.

Given a horizontal/vertical *outreach* and *key*, the algorithm chooses the *reference grid* so that there are no missing images on the columns of the reference views' x indices.

2.3.5 Feature point depths

If depth maps are available, they are used to attribute a depth value d to each feature point $p(f, v) = (x, y, d, w)$. The depth is read from the view's depth map, at pixel position (x, y).

However, features are often located on the border of foreground objects. Taking a single pixel value in the depth map could incorrectly take the depth of the background, or an intermediary value. Therefore the algorithm takes a small pixel window around (x, y), and retains the minimal (i.e. closest to camera) value in it.

2.3.6 Feature point weights

Feature points $p(f, v)$ can have a weight value. If many feature points are clustered together on a small region of the image (for example a checkerboard), it is reasonable to given them a lower weight, and to give a higher weight to more isolated feature points.

Especially for the rotation estimation from optical flow slopes (see later), it is important that all regions in the image are uniformly represented.

This is not implemented.

2.4 Observations

Looking at the image correspondences on figures 2.3, it can be seen that the arrangement of the feature points roughly corresponds to the (inverted) camera positions on P. (The vertical gap on the second figure is a result of the acquisition system: It did not take the y-step properly at that height.)

The feature points for every feature f will be arranged in the same pattern, just at different places in the image, with different scales (i.e. disparities), and with a distortion due to the camera orientation $\mathbf{R}$. The basic idea of this calibration method is to overlay the feature points for several features, make their scale uniform, remove the rotation distortion, and take the averages. From this the camera position is then derived.

2.4.1 Rotation

The rotation $\mathbf{R}$ (orientation of the camera relative to the plane P) distorts the feature points. For figure 2.6, a 1D optical flow was taken, with the camera moving on a horizontal axis only. The feature points were then overlaid, centered on one feature point, and given uniform scale. The figure shows these transformed feature points, one color for each feature. It can be seen that they form lines with different slopes. The scaling does not affect the slope. The different slopes are caused by the camera's rotation $\mathbf{R}$. In the 2D case, it is possible to estimate $\mathbf{R}$, from these slopes alone. This is done in section 2.5.

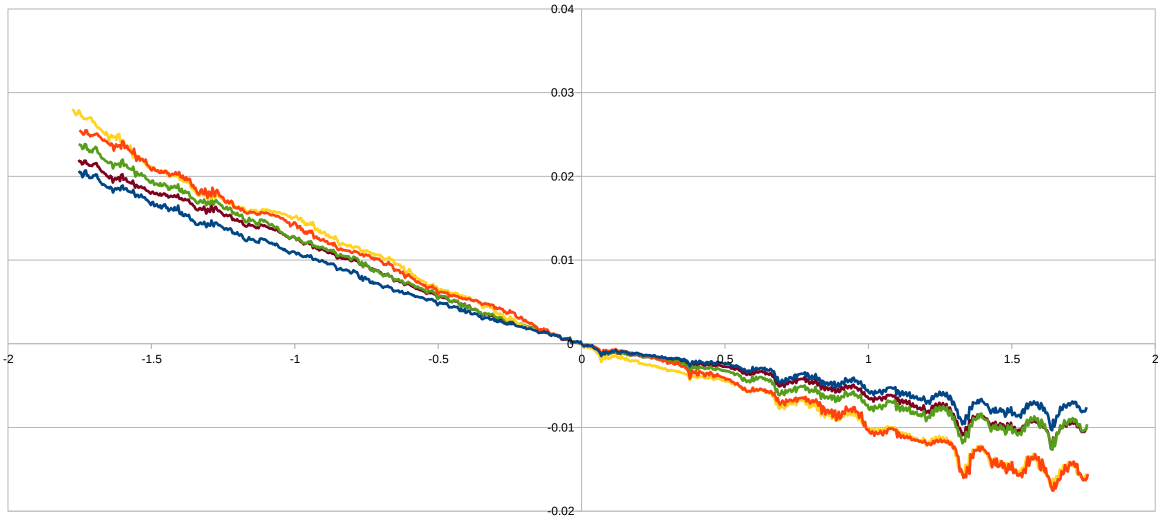

Figure 2.6: Overlayed feature points in 1D case

2.4.2 Depth

Another problem is that if more than two sets of feature points should be given a common scale, one set of feature points would need to be chosen as reference. But this would amplify the error in the correspondences of that particular feature. So it is better to calculate scaling factors in a more global way. This is done with the *straight depths*, in section 2.6.

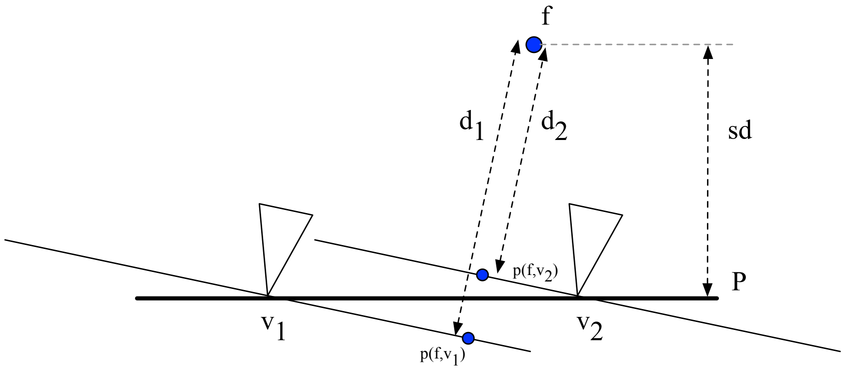

Figure 2.7: Differing feature point depths

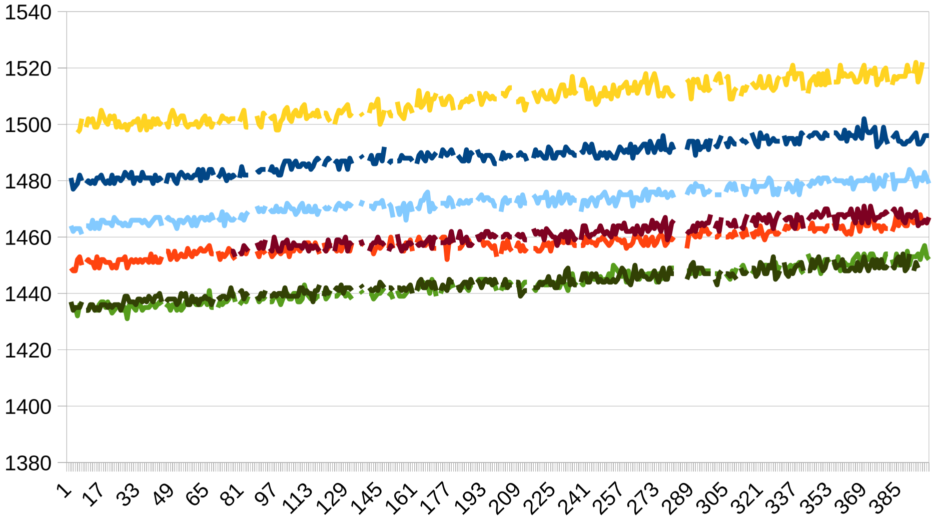

Figure 2.8: Feature point depths as x index of view varies

Figure 2.7 shows how because of the rotation $\mathbf{R}$, the feature points depths d_1, d_2 for one feature f are not all equal. Figure 2.8 shows the feature point depths from one data set, of 7 different features, as the x index of the view varies. Despite the noise, a clear linear increase can be seen, with the same slope for each feature.

In the 2D case, these instead be parallel planes. In section 2.5.2, the yaw and pitch components of the rotation $\mathbf{R}$ are estimated from this.

In section 2.6 all of these feature points depths are aggregated together, to calculate the straight depth sd_f of each feature, using $\mathbf{R}$ and the camera intrinsic matrix $\mathbf{K}$.

2.5 Camera rotation

The next step is to determine the camera rotation $\mathbf{R}$. This is the rotation of the cameras relative to the plane P on which the camera centers are placed. It is a 3D rotation matrix, with 3 degrees of freedom.

There are two methods to estimate $\mathbf{R}$.

One is based only on the *slopes* of the image correspondences, and does not need depth maps. It needs the cameras to be aligned on a regular grid. It seems to produce an accuracy of around 0.5° for the three rotation angles. It is described in the next section 2.5.1.

The other uses the differing depths of the feature points to estimate the rotation, by doing a least squares plane fitting operation, followed by an adjustment of the roll rotation. It is described in section 2.5.2. It seems to give better results.

2.5.1 Optical flow slopes

2.5.1.1 Flow equation

The camera intrinsic matrix $\mathbf{K}$ projects points in the camera view space (v_x, v_y, v_z), to pixel positions on the image (i_x, i_y) in homogeneous coordinates, according to

$$w \begin{bmatrix} i_x \\ i_y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{2.1}$$

$$i_x = f_x \frac{v_x}{v_z} + c_x \qquad i_y = f_y \frac{v_y}{v_z} + c_y \tag{2.2}$$

$$v_x = \frac{v_z}{f_x}(i_x - c_x) \qquad v_y = \frac{v_z}{f_y}(i_y - c_y) \tag{2.3}$$

As shown on figure 2.7, the camera moves such that the camera center is on a plane P, and the camera has a constant rotation $\mathbf{R}$ relative to P.

The *world space* coordinate system is set such that its $z = 0$ plane is P, and its origin is any point $\vec{O}(0,0,0) \in P$.

Let $\vec{i}(\vec{O})$ be some point in the image of the camera when it is placed at O. Let $\vec{v}(\vec{O})$ be the same point in the camera's view space, calculated with formula 2.3, with a given value $z = v_z(\vec{O})$. Let $\vec{w}$ be the same point in *world space*. For any camera center position $\vec{Q} \in P$, the relation is

$$\vec{v}(\vec{Q}) = \mathbf{R}(\vec{w} + \vec{Q}) \tag{2.4}$$

To get from world space to view space, the coordinate system is first translated by $\vec{Q}$ (where $Q_z = 0$), then rotated by $\mathbf{R}$. In particular,

$$\vec{v}(\vec{O}) = \mathbf{R}\vec{w} \tag{2.5}$$

hence

$$\vec{v}(\vec{Q}) = \vec{v}(\vec{O}) + \mathbf{R}\vec{Q} \tag{2.6}$$

Using formula 2.2, $\vec{i}(\vec{Q})$ can now be calculated from this. So one has the function

$$\text{flow}_{\mathbf{R}} : \langle \vec{i}(\vec{O}), \vec{Q}, z \rangle \mapsto \vec{i}(\vec{Q}) \tag{2.7}$$

2.5.1.2 Horizontal and vertical camera movement

The following section will analyze how $\vec{i}(\vec{Q})$ evolves when $\vec{Q}$ moves on P. Most importantly, it is shown that when $\vec{Q}$ moves horizontally or vertically on P, then the **slope** at which $\vec{i}(\vec{Q})$ moves on the image does not depend on z.

Let $\vec{H} = (\epsilon, 0, 0)$ and $\vec{V} = (0, \epsilon, 0)$. They represent a horizontal and vertical displacement of the camera on P, by some magnitude ϵ.

Because of the transformation between cartesian and homogeneous coordinates in formulae 2.2 and 2.3, the function flow$_{\mathbf{R}}$ cannot be expressed directly as a matrix equation. **R** is decomposed:

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.8}$$

To simplify the expressions, the coordinates of the chosen image point $\vec{i}(\vec{O})$ are simply denoted (i_x, i_y). Also, $z = v_z(\vec{O})$.

For the horizontal camera movement by $\vec{H}$, one gets:

$$i_x(\vec{H}) = \frac{i_x z + f_x r_{11}\epsilon + c_x r_{31}\epsilon}{z + r_{31}\epsilon} \quad \text{and} \quad i_y(\vec{H}) = \frac{i_y z + f_y r_{21}\epsilon + c_y r_{31}\epsilon}{z + r_{31}\epsilon} \tag{2.9}$$

And for the vertical camera movement by $\vec{V}$, one gets:

$$i_x(\vec{V}) = \frac{i_x z + f_x r_{12}\epsilon + c_x r_{32}\epsilon}{z + r_{32}\epsilon} \quad \text{and} \quad i_y(\vec{V}) = \frac{i_y z + f_y r_{22}\epsilon + c_y r_{32}\epsilon}{z + r_{32}\epsilon} \tag{2.10}$$

2.5.1.3 Slopes

It can be shown that as ϵ varies, $\vec{i}(\vec{H})$ moves on a straight line. Its slope is

$$s_H = \frac{i_x - i_x(\vec{H})}{i_y - i_y(\vec{H})} \tag{2.11}$$

This expression simplifies to

$$s_H = \frac{i_x - i_x(\vec{H})}{i_y - i_y(\vec{H})} = \frac{f_y r_{21} + c_y r_{31} - i_y r_{31}}{f_x r_{11} + c_x r_{31} - i_x r_{31}} \tag{2.12}$$

The variables ϵ and z both vanish. s_H depends only on **R**, **K** and $\vec{i}(\vec{O})$.

For the vertical camera movement by $\vec{V}$, one gets the similar expression

$$s_V = \frac{i_y - i_y(\vec{V})}{i_x - i_x(\vec{V})} = \frac{f_x r_{12} + c_x r_{32} - i_x r_{32}}{f_y r_{22} + c_y r_{32} - i_y r_{32}} \tag{2.13}$$

Note that s_H is a slope x/y, whereas s_V is a slope y/x. This is because **R** is expected to be near **I**, and in that case $\vec{i}(\vec{H})$ and $\vec{i}(\vec{V})$ move almost horizontally and vertically on the images respectively, and so both slopes approach zero (and not infinity).

2.5.1.4 Samples from image correspondences

In order to apply this to estimate $\mathbf{R}$, the dataset must be such that the camera centers of views $v(x-1,y), v(x,y), v(x+1,y), \dots$ must be in an approximately straight line ("horizontal"). For views $v(x,y-1), v(x,y), v(x,y+1), \dots$ they must also also be an approximately straight line ("vertical"), which is perpendicular.

After the *image correspondences* were computed, for each *feature* f, a horizontal and a vertical slope $s_H(f), s_V(f)$ are estimated using line fitting on the feature point correspondences for those view indices.

This gives for each feature f a sample

$$S_f = \langle p(f,v), s_H(f), s_V(f) \rangle \tag{2.14}$$

$p(f,v)$ is the feature point position of f for the reference view v. This corresponds to the (i_x, i_y) from the previous formulae. If multiple reference views were used, the samples from different reference views can be put together here.

2.5.1.5 Estimating camera rotation

It is possible to estimate $\{r_{11}, r_{21}, r_{31}, r_{12}, r_{22}, r_{32}\}$ from these samples by solving two linear homogeneous least squares systems. From this $\mathbf{R}$ could probably be completed knowing it is an orthogonal matrix with $\det(\mathbf{R}) = 1$.

But this is not done in the algorithm. Instead a parametrization of $\mathbf{R}$ with three Euler angles (X, Y, Z) is optimized with an iterative method. The error to minimize is the mean squares sum of the predicted slopes for a given $\mathbf{R}$, minus the measured slopes.

The parametrization (X, Y, Z) of $\mathbf{R}$ is $\mathbf{R}^\intercal = \mathbf{R_z}(Z)\mathbf{R_y}(Y)\mathbf{R_x}(X)$. $\mathbf{R}^\intercal$ is the orientation of the camera in world space. So the *roll* rotation Z (around the optical axis) is performed last.

The three angles are interdependent: Say X is adjusted to minimize the error. Then Y is adjusted to reduce the error even more. Now X is no longer at the optimal setting, and needs to be readjusted.

The roll rotation has the most impact. Three golden-section searches are performed sequentially which optimize Z, X and Y, in that order. The entire process is repeated iteratively until a certain error threshold. With each (outer) iteration, the tolerance and search interval of the (inner) golden-section searches are reduced.

2.5.1.6 Accuracy

On an artificially generated test dataset with a known camera rotation of $(10°, 20°, 5°)$, the estimated rotation was $(10.5289°, 20.6345°, 5.36933°)$. This artificial dataset has some random noise and outliers, 200 features, and 30×30 views. It is hard to estimate the accuracy for real datasets because the real rotation is unknown and typically very small.

2.5.2 Feature point depths

As shown on figure 2.7, feature points of the same feature get different depths on different views because of the rotation $\mathbf{R}$. The figure shows the situation in world space. There is one global position of the feature f, and different positions of the camera centers v_1, v_2.

Figure 2.9 shows the same situation, but instead in the overlaid view spaces of the two cameras. The plane formed by the feature points for f in the different view spaces, has the same orientation $\mathbf{R}$.

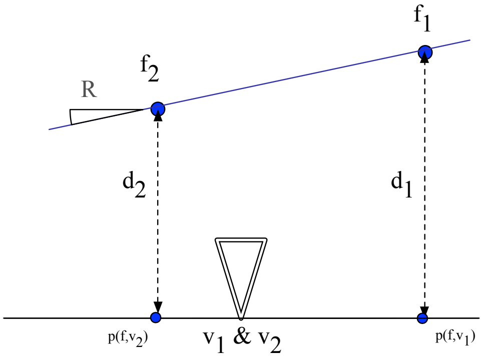

Figure 2.9: View of figure 2.7 from overlaid view space coordinate systems

This algorithm calculates for each feature f, the view space coordinates of the feature points $p(f, v_i) = \langle x, y, d, w \rangle$, using $\vec{v}_{f,v_i} = \mathbf{K}^{-1}(dx, dy, d)$. Then, using linear least squares, a plane it fitted to the points $\vec{v}_{f,v_i}$, and its normal vector $\vec{n}_f$ is computed. This plane fitting should average out the noise in the depth measurements.

Then the computed normal vectors $\vec{n}_f$ for each feature f are averaged to get a global estimate $\vec{n}$. A preliminary rotation matrix $\mathbf{R}_{xy}$ is derived from $\vec{n}$. It contains the correct pitch and yaw rotation of the camera relative to P, but not the correct roll.

Now again for each feature f, the view space points $\vec{v}_{f,v_i}$ are taken, and premultiplied by $\mathbf{R}_{xy}$. Using only the x and y components of the resulting vectors, a 2D rotation angle α_f is estimated such that those 2D points become horizontally/vertically aligned.

Again an average α is computed and transformed into a 3D roll rotation matrix $\mathbf{R}_z$.

Finally $\mathbf{R}_{xy}$ and $\mathbf{R}_z$ are combined into $\mathbf{R}$.

There are some problems left with this implementation. On the artificially generated test dataset (same as previously), a rotation of $(9.93871°, 19.8223°, 5.22812°)$ was estimated. The correct values are $(10°, 20°, 5°)$. So the method appears to be more accurate.

2.6 Straight depths

Knowing $\mathbf{R}$, the *straight depth* of each feature f can now be calculated. It is the orthogonal distance of the feature point in the scene, to the plane P. This is the distance sd in figure 2.7. In can be samples from each feature point distance d_i, using the rotation $\mathbf{R}$ and the camera intrinsic matrix $\mathbf{K}$.

Two methods are available for estimating the *straight depth* sd_f of each feature. One aggregates the measured feature point distances d_i. Another estimates the straight depths from the relative scales of the feature points, whereby one or more straight depths are fixed manually.

In the case where no depth maps are available, the second method can be used. Some (at least one) depth then need to be determined manually. If depth maps are available, the second method can still be used to complete or correct estimations from the first method.

2.6.1 Aggregating feature point depths

For each feature point $p(f,v) = \langle x, y, d, w \rangle$ which has a feature point depth d, the straight depth of that feature sd_f can be estimated.

The feature point is first back-projected into the camera's view space with $\vec{v} = \mathbf{K}^{-1}(dx, dy, d)$. Then it is transformed into the view space of a camera with the same optical center, but perpendicular to P. This is $\vec{v'} = \mathbf{R}^{\intercal}\vec{v}$. The straight depth is the third component of $\vec{v'}$.

For each feature f, samples $sd_{f,v}$ are calculated in this way. They should theoretically be all the same, but due to noise, outliers, and the error in $\mathbf{R}$, they will differ. At least the noise part of the error can be removed by averaging the samples.

To also remove the influence of outliers, the following procedure is used: First the median of the samples $sd_{f,v}$ is computed. It is not affected by outliers, but by the noise. Then the $sd_{f,v}$ whose absolute difference to the median is above a given threshold, are removed. The remaining samples are averaged.

This average is used as the final sd_f. t is set to 10 mm. As a metric of accuracy, the standard deviation of these samples is also taken.

2.6.2 Depth from disparity

This alternate method computes straight depths using only the relative scales of the different features' disparities, and some fixed feature depth given as input. It proceeds in three steps: (1) Calculate the relative scale for each pair of features. (2) Derive a global scale for each feature. (3) Using at least one *known depth*, calculate the depth of each feature.

First the feature points $p(f,v)$ of each feature are undistorted (if any distortion) and unrotated. For this the image coordinates are premultiplied by $\mathbf{K}\mathbf{R}^{\intercal}\mathbf{K}^{-1}$. This is invariant of the feature depths, and d is fixed to 1.

2.6.2.1 Pairwise scale ratios

The feature points of each feature f_i now all have the same pattern, except for a different scale, a different position in the image, and a different subset of covered views.

For each pair of features $(f_{\mathrm{i}}, f_{\mathrm{j}})$, a relative scale $r_{j \rightarrow i}$, and a translation $\vec{t} \in \mathbb{R}^2$ are computed that make the two *feature point* sets overlap, by optimizing

$$\forall v : p(f_i, v) = r_{j \rightarrow i} p(f_j, v) + \vec{t} \tag{2.15}$$

If the two features were taken on different *reference views*, then this linear least squares problem is solved for $r_{j\to i}$ and $\vec{t}$.

Otherwise, if the two features were taken on the same *reference view* rv, then is is known that $p(f_i, rv)$ and $p(f_j, rv)$ should coincide perfectly: They represent the same view, and they were initially chosen as feature points, so they have no error. So the feature point positions for fv are fixed, and for each view $v \neq rv$, a sample of $r_{j\to i}$ is calculated using

$$v : r_{v,j\to i} = \frac{p(f_i, rv) - p(f_i, v)}{p(f_j, rv) - p(f_j, v)}$$

Samples closer to the rv are given a lower weight. They have more error because of the limited pixel resolution. The samples are then averaged to obtain $r_{j\to i}$. The translation is set to $\vec{t} = p(f_i, rv) - s \times p(f_j, rv)$.

The resulting $r_{j\to i}$ is discarded or assigned a lower weight if the error of the solution is too large. The resulting $\vec{t}$ is not further used (only to calculate the error).

Calculating $r_{j\to i}$ for each feature pair $(f_{\mathrm{i}}, f_{\mathrm{j}})$ gives a *pairwise scales matrix*, as seen shown in figures 2.10 and 2.11. The axis are the feature indices i and j. Because each unordered pair is considered only once, the matrix is triangular. When there are multiple reference views, features from two different reference views have none (or little) feature points in common. This causes the black areas in the lower-left part in figure 2.11.

2.6.2.2 Global scale ratios

The next step is to deduce global scales $\{r_{f_0}, r_{f_1}, r_{f_2}, ...\}$ from these samples. The *global scale* of one (arbitrarily chosen) feature f_0 is set to $r_{f_0} = 1$.

Then all the global scales are calculated such that

$$\forall i, j : \frac{r_{f_j}}{r_{f_i}} = r_{j\to i}$$

This is done by solving a sparse linear least-squares system $\mathbf{A}\vec{x} = \vec{b}$. The system consists of equations

$$\begin{cases} \forall i, j : r_{f_j} \times r_{j\to i} - r_{f_i} = 0 \\ r_{f_0} = 1 \end{cases}$$

So the $\mathbf{A}$ matrix is very large and sparse. It has one column for each relative scale ratio, and one row for each feature. A sparse QR decomposition is done to compute it, using the *Eigen* linear algebra library.

2.6.2.3 Depths

The scale ratios relate directly to the depths: Using the pin-hole camera model, one gets

$$\forall i, j : \frac{sd_i}{sd_j} = \frac{r_{f_j}}{r_{f_i}}$$

where sd_i, sd_j are the straight depths of the features f_i, f_j. As a consequence, there is one global s such that

$$\forall i : sd_i = s \times r_{f_j}$$

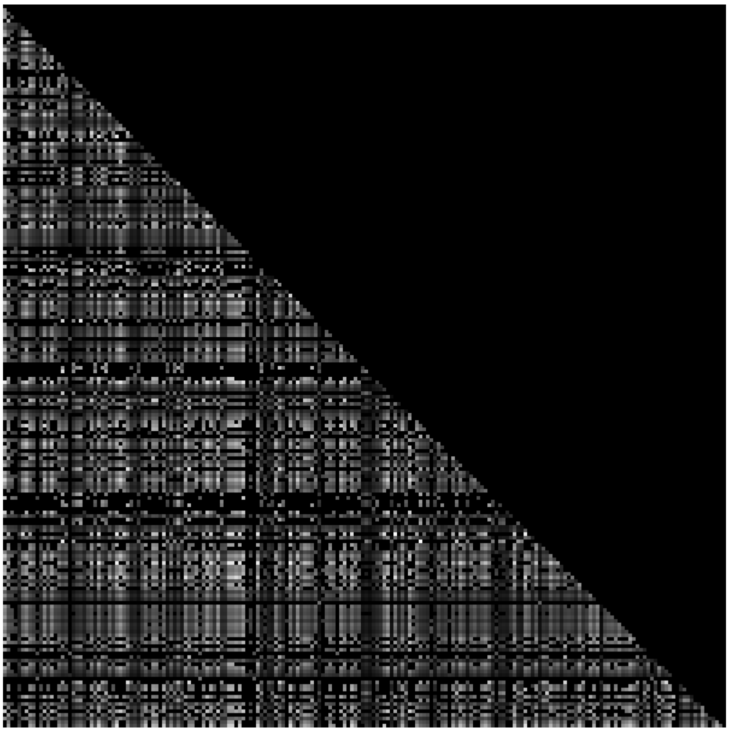

Figure 2.10: Pairwise feature scales matrix

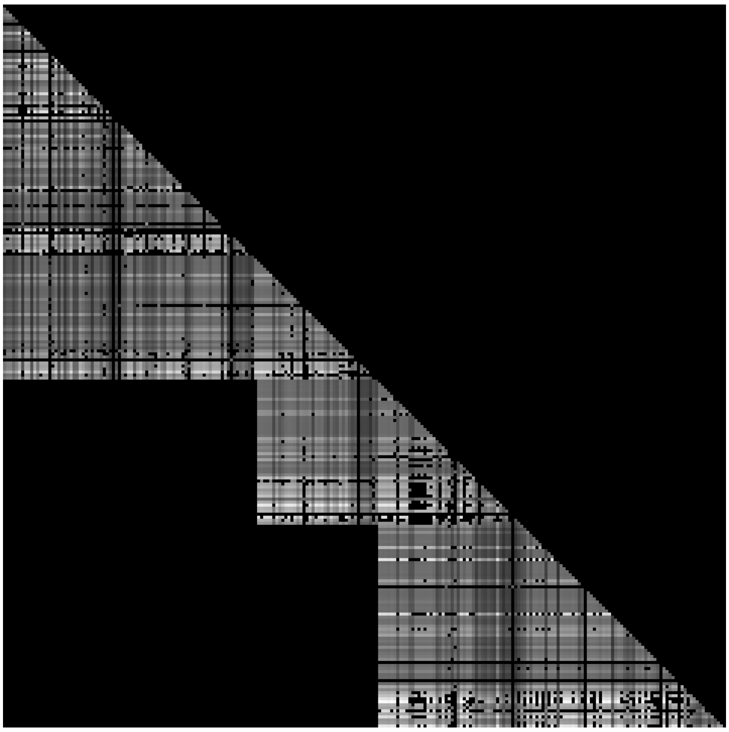

Figure 2.11: Pairwise feature scales matrix (multiple reference views)

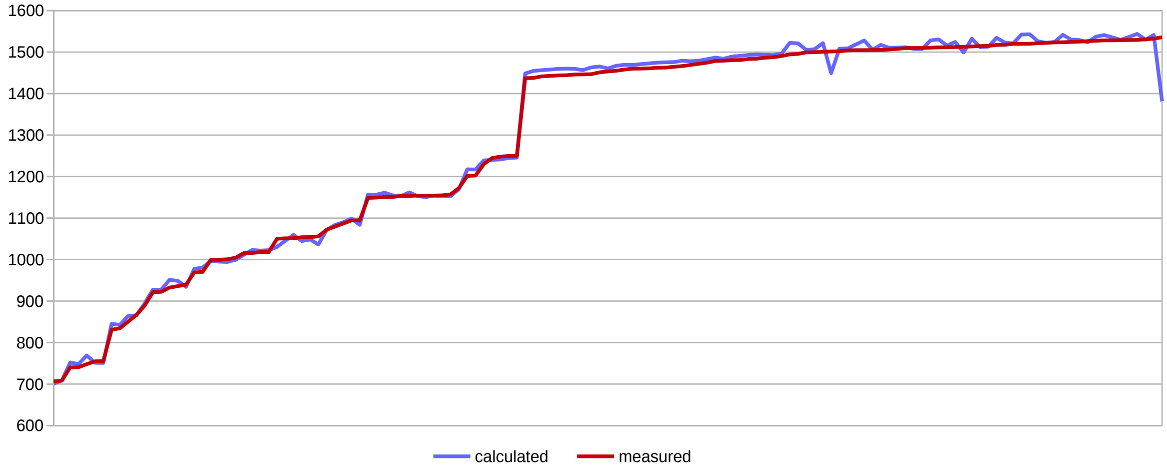

Figure 2.12: Calculated v. measured straight depths

One or multiple *known depths* $\{sd'_0, sd'_1, ...\}$ need to be given as input. They must also be *straight depths*. From these, s is calculated using

$$s = \frac{\sum_i sd'_i}{\sum_i r_{f_i}}$$

Then, the remaining sd_i are calculated using s and the global scales r_{f_i}.

This parameter s will be the scale of the final camera positions. If it is at a wrong value, then the coordinate system of the final camera positions will globally be at the wrong scale (not corresponding to the actual size of the scene, given by the depth maps). This will lead to an error when view synthesis is done, that gets larger linearily with the baseline. It may be needed to manually readjust the scale after the calibration is done.

2.6.2.4 Results

Figure 2.12 shows the straight depth of 135 features, sorted in ascending order. The “measured” (red) line are the depth values taken from the depth maps, using the previous method in section 2.6.1. The “calculated” (blue) line are the same depth values, calculated only using the relative scales of the feature points. Only 3 *known depth* values were taken to compute these values.

The root mean square error in this example is about 17.60 mm. This error can be both in the measured and in the calculated depth values. This graph does not indicate whether the measured or the calculated values are wrong: The “measured” line is smooth only because the samples have been sorted by the “measured” values.

2.7 Camera positions

Finally, having *image correspondences*, the *camera rotation*, and the *straight depths*, the camera center positions can be computed.

2.7.1 Relative camera positions

For each *reference view*, a set of *relative camera positions* c_v will be computed. It is the position of the camera center for view v, relative to that of the *reference view.* If there are multiple reference views, the process is repeated for each reference view. The different sets of *relative camera positions* will then be *stitched* together in the next step.

If there is distortion in the camera intrinsics, all feature points $p(f, v)$ are first undistorted. Then they are unrotated, by premultiplication with $\mathbf{K}\mathbf{R}^{\intercal}\mathbf{K}^{-1}$.

The camera position for the reference view rv itself is always set to $c_{rv} = (0, 0)$.

For each other *target* view v, for each of its feature points $p(f, v)$, a sample $c_{f,v} = (x, y)$ of its camera position is calculated using

$$c_{f,v} = \left(\frac{[p_x(f,v) - p_x(f,rv)] \times sd_f}{f_x}, \frac{[p_y(f,v) - p_y(f,rv)] \times sd_f}{f_y} \right)$$

where f_x, f_y are the focal lengths from the camera intrinsic matrix $\mathbf{K}$. This is derived directly from the pin-hole camera model.

In theory all $c_{*,v}$ should have the same value (for the same v), but there can be a large deviation because of errors in the image correspondences, rotation, or straight depth. The deviations necessarily get larger the further the target view v is from the reference view rv. This is why in section 2.3.4, the *outreach* of the optical flow needed to be kept small.

Figures 2.13 and 2.14 show the relative camera positions computed for an artificial data set. This is an artificially generated scene with random features, for which the accurate image correspondences, rotation, camera parameters and depths are known. The figures show all the *samples* $c_{f,v}$, with a different color for each f. At the center is the reference view rv. On figure 2.13, the rotation is set to a slightly incorrect value. On figure 2.14, the straight depths for some features are set an an incorrect value. When they are both set to the correct value, a perfect overlap is obtained.

The rotation $\mathbf{R}$ is a global value, and an incorrect value will affect all samples equally. But the straight depth sd_f is different for each each feature f. So, in a first pass, the algorithm finds *bad features*, whose straight depth is likely to be incorrect.

For each view v, it takes the mean of the feature points $p(f, v)$, and then takes each feature point's distance $e(f, v)$ to the mean. For features with an incorrect straight depth, this value will likely be larger. (Even though the mean is also affected by the outliers.) After this is done for each view, the average $e(f)$ is calculated for each feature. The features whose $e(f)$ are largest are marked as *bad features* and removed. During this procedure, features from all *reference views* are considered.

Figures 2.15, 2.16 and 2.17 show camera position samples $c_{f,v}$ for a real dataset, both before and after this filtering. The distance between adjacent camera positions is about 1 mm. Figure 2.17 is a close-up view of feature points, far from the center (i.e. the reference view rv). It can be seen that after the filtering, the outlier samples are removed and the camera positions become recognizable.

The camera positions c_v are computed by averaging the filtered camera position samples $c_{f,v}$. They are shown on figure 2.17 also (the $\bullet$ symbols). If the variance is too large, or there are not enough samples, the camera position c_v is rejected. If outliers are removed, then this averaging should remove the noise in the image correspondences.

2.7.2 Stitching

If only one *reference view* was used, this completes the calibration process.

If multiple *reference views* were used, the previous step is repeated for each reference view, giving each time *relative camera positions* with that reference view at origin. To stitch these together into *absolute camera positions* in a common coordinate system, the absolute camera positions of the reference views need to be determined.

As shown in section 2.3.4, the reference views need to be in a grid. (Allowing for arbitrarily chosen reference views would render this algorithm more complex.) The center reference view is chosen as origin, its *absolute camera position* is set to $(0, 0)$. The distances between horizontally and vertically adjacent reference view camera positions, are computed by comparing *relative camera positions* that exist for both reference views.

For this to work the reference views grid in section 2.3.4 needed to be set so that the overlap is big enough. Note that for the optical flow, the *outreach* is an upper bound, not all features are tracked that far, and many are filtered out.

This algorithm only compares horizontally and vertically adjacent reference views, no others. An algorithm that takes all overlaps into account and does not require the reference views to be on a grid might be better, but would be more complicated.

Then, knowing the *absolute camera positions* of each reference view C_{rv}, and *relative camera positions* $c_v^{(rv)}$, the final *absolute camera positions* for each view v are computed as

$$C_v = C_{rv} + c_v^{(rv)}$$

For views v where relative camera positions exists for multiple reference views rv, only the one where $\|v - rv\|$ is smallest is taken (the euclidian distance between view indices).

The distances between these absolute camera position samples is used as metric of accuracy. It should ideally be zero.

Figures 2.18 and 2.19 show final absolute camera positions, for a real dataset. The irregularity in the bottom area of figure 2.18 was caused by the vertical camera movement in the acquisition system. The distance between adjacent camera positions is about 1 mm. The close-up figure 2.19 shows how the best sample for C_v was chosen during the stitching. Rejected samples are in gray. This figure is greatly exaggerated in y direction.

2.7.3 Redistributing image correspondences

The *reference views* and *outreach* were initially set for the optical flow tracking. However, for the camera position computation, it may be better to have more reference views, with even smaller outreach.

There is a program to *redistribute* the image correspondences. For each *reference view* rv, some views are selected that will become *pseudo-reference views* rv', in the new image correspondences. Overlapping subsets of the feature points with reference view rv, with a limited radial *outreach*, are selected and copied for rv'.

They are *pseudo-reference views* unlike with the real *reference view*, they have an error from the optical flow feature tracking.

But no better results were obtained from this.

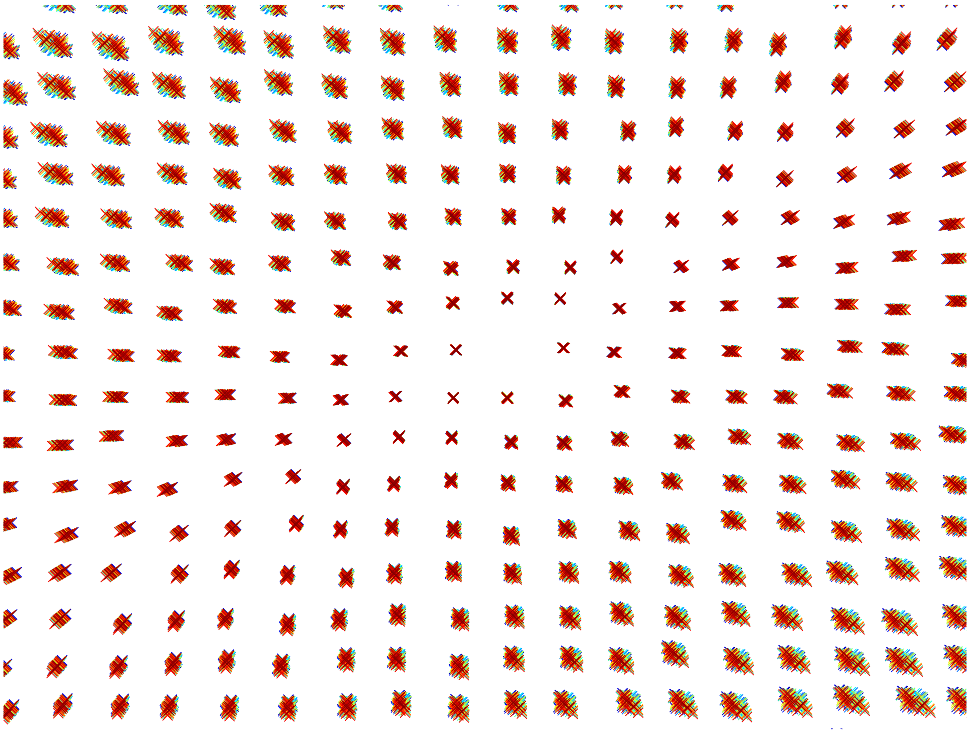

Figure 2.13: Camera positions for artificial dataset (erroneous **R**)

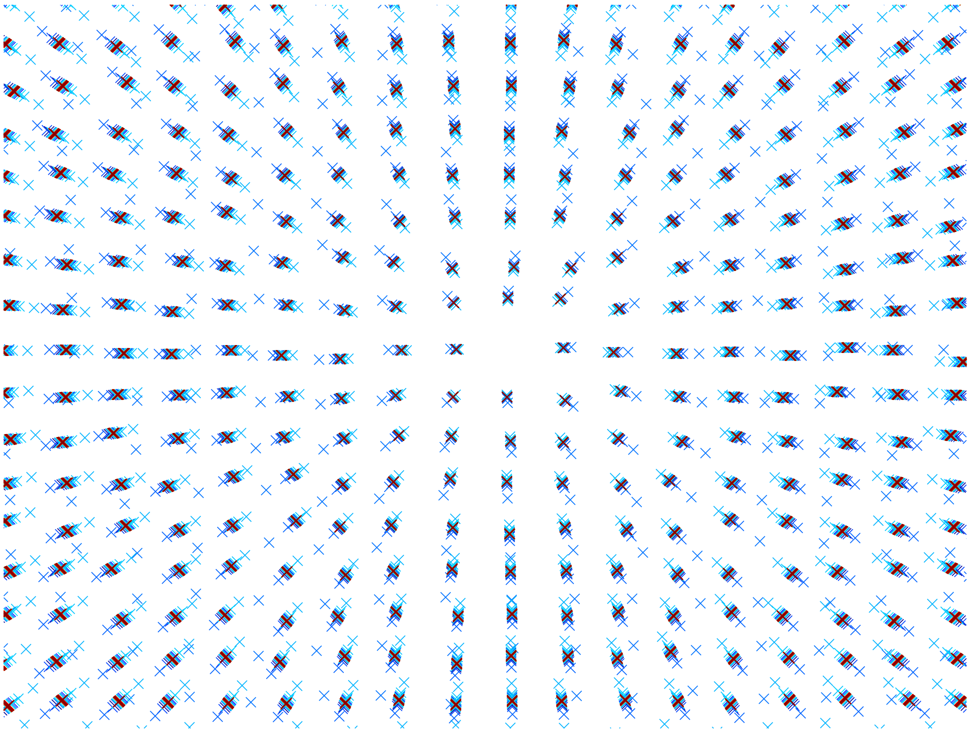

Figure 2.14: Camera positions for artificial dataset (erroneous straight depths)

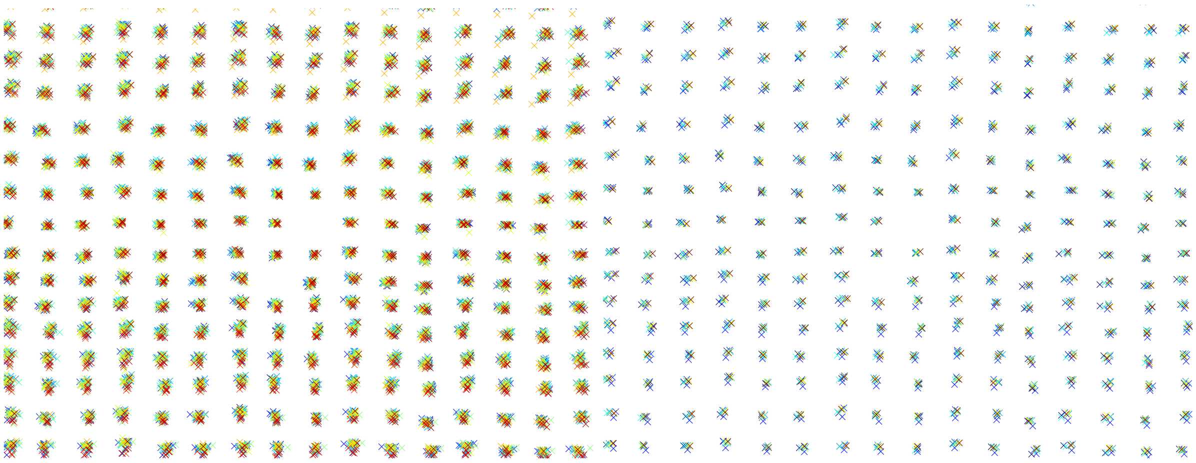

Figure 2.15: Camera positions for real dataset, unfiltered and filtered (close to center)

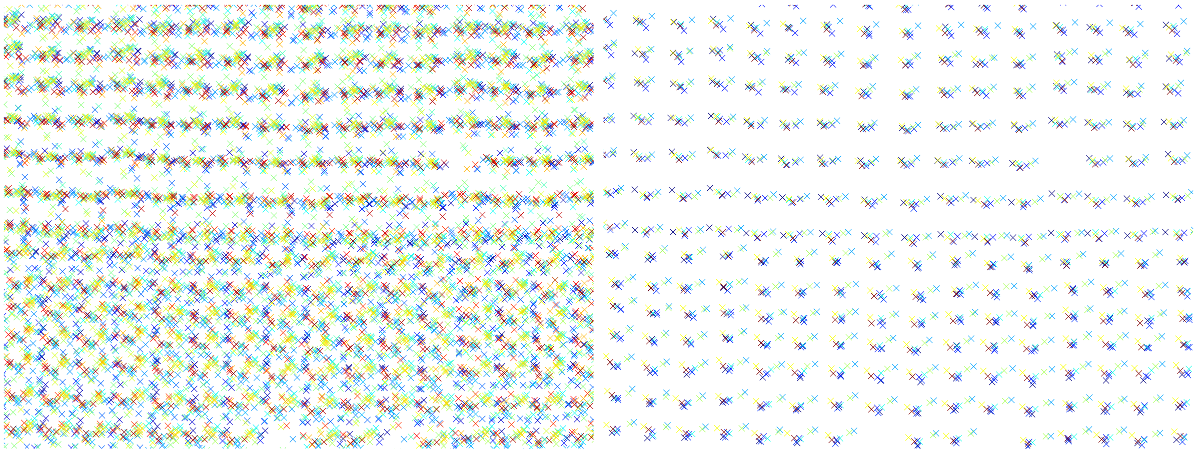

Figure 2.16: Camera positions for real dataset, unfiltered and filtered (far from center)

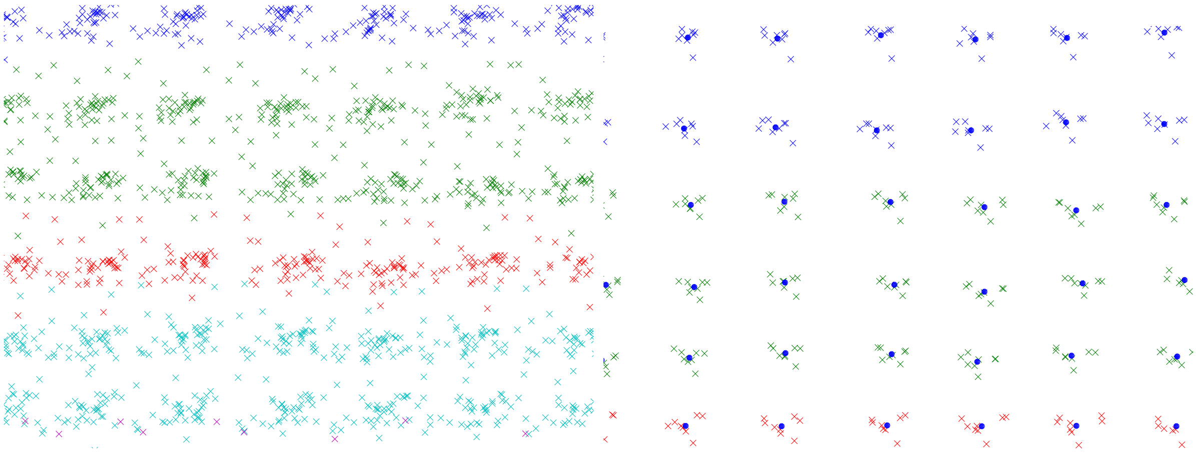

Figure 2.17: Camera positions for real dataset, unfiltered and filtered (close-up)

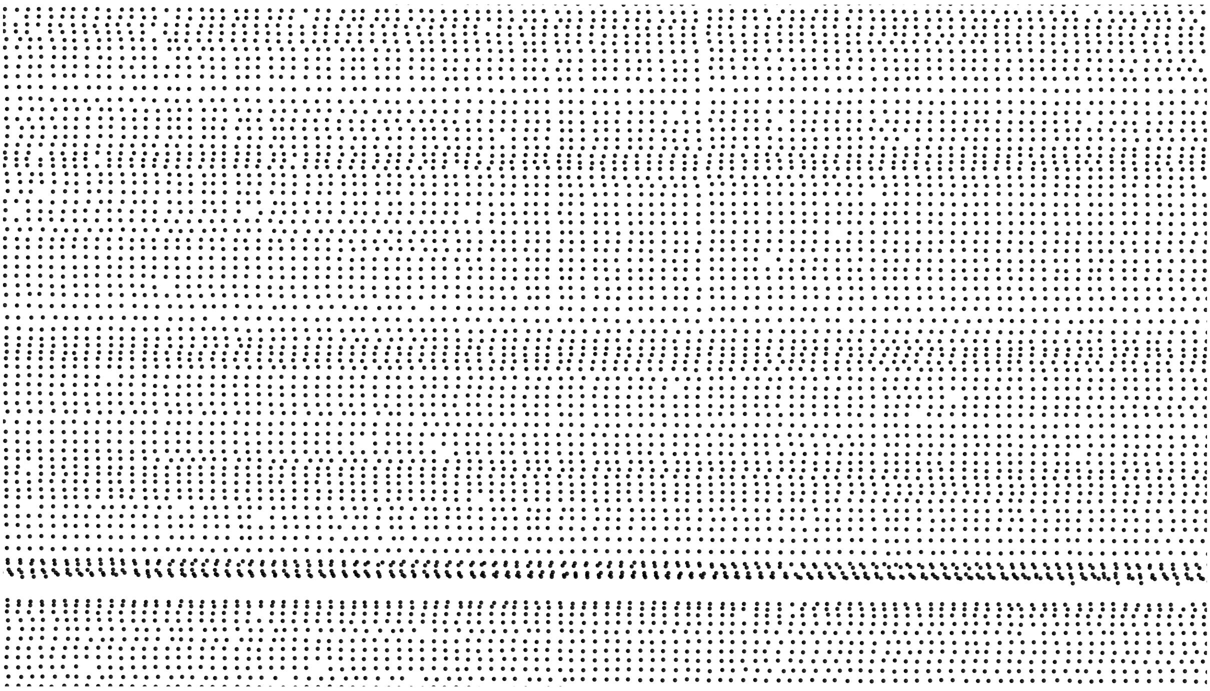

Figure 2.18: Stitched absolute camera positions

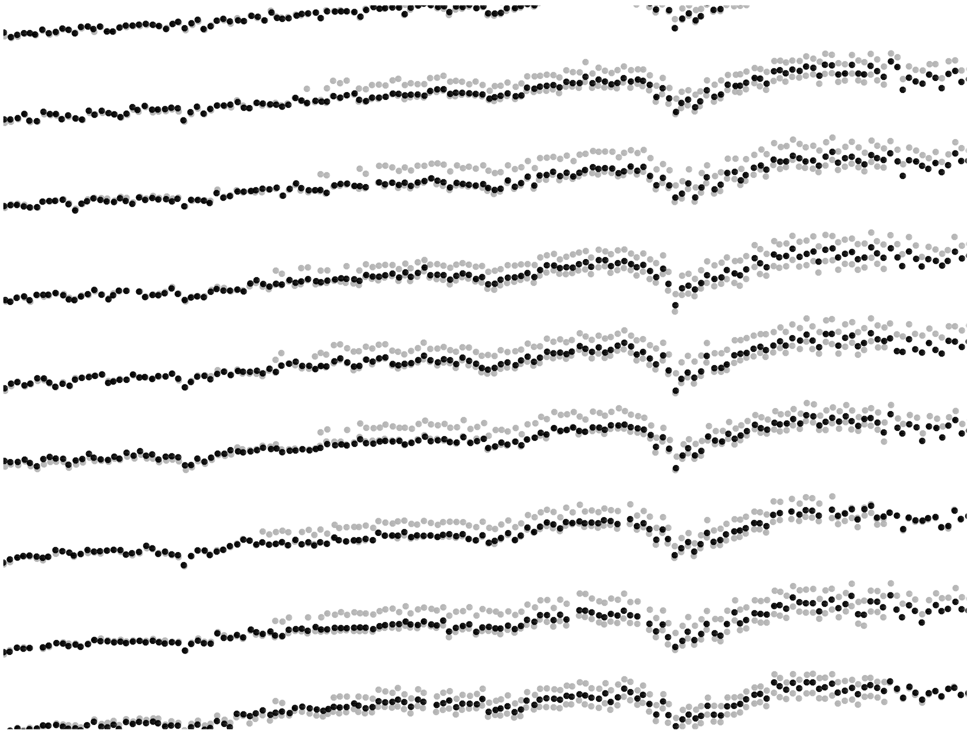

Figure 2.19: Stitched absolute camera positions, close-up with rejected samples

2.7.4 Final camera extrinsics

The final **Rt** extrinsic camera matrix to compute is a transformation from *world space* to *view space*, such that

$$v = \mathbf{R}w + \vec{t}$$

The world space can be any coordinate system, but must be the same for every view.

For this method, world space is set to lie on the plane P. The *absolute camera positions* are translations in world space, where the third component is set to $z = 0$. The computed $\mathbf{R}$ is the orientation of the camera in world space. The transformation is

$$v = \mathbf{R}(w + \vec{c_v})$$

To obtain the final **Rt** matrices, $\mathbf{R}$ is unchanged, and the translation is set to $\vec{t} = \mathbf{R}\vec{c_v}$.

This concludes the calibration.

3 Usage

The above method is implemented as part of the `licornea_tools` package. Tools in `calibration` that are prefixed with `cg` are specific to it.

3.1 Preliminaries

Before doing the *camera grid* calibration, the following needs to be done:

- Prepare `parameters.json` dataset parameters file for the dataset. It indicates the index ranges (and steps), and the locations and file name formats of the images and depths.
- Prepare `intr.json` file with intrinsic parameters of the camera. It can contain distortion coefficients, if the images (and depths) are distorted. But this was not tested.
- Do the reprojection of the depth maps. The reprojected depth maps will be at the location indicated by `depth_filename_format` in the root group of the dataset parameters.

3.2 Image correspondences

3.2.1 Reference grid

Firstly, the *references view grid* can be chosen using

```
calibration/cg_choose_refgrid parameters.json 200 100 refgrid.json
```

This computes a *reference grid* (indices of the *reference views*), and puts it into `refgrid.json`. Here, the *horizontal key* is 200, and the *vertical key* is 100. If the keys are larger than the horizontal and vertical ranges, the *reference grid* will consist only of one reference view. The program chooses the *reference views* such that there are no missing views on each vertical axis.

3.2.2 Undistort images (if applicable)

If there is distortion in the images (defined in the intrinsic parameters file), then an image can be undistorted using

```
calibration/undistort_image in_image.png out_image.png intr.json texture
```

For the depth maps, use `depth` instead of `texture`. It will then use nearest neighbor interpolation. This needs to be done for each image and each depth map in the dataset.

Alternately, it is also possible to compute the optical flow on the distorted images, and undistort the *image correspondences* later.

3.2.3 Optical flow features

Now the feature points on the reference views to track are selected using

```
calibration/cg_optical_flow_features parameters.json refgrid.json of/
```

It displays a graphical user interface, and the parameters can be adjusted such that good feature points get selected for all reference views. Hitting *Enter* puts a file `of/fpoints_*.json` into the `of/` directory, each containing the feature points for one reference view.

All of the features will get globally unique names of the form `feat_RFFF`, where `R` is the number of the reference view, and `FFF` the number of the feature.

3.2.4 Optical flow correspondences

Now the optical flow correspondences can be computed, using

```
calibration/cg_optical_flow_cors parameters.json of/fpoints_100,100.json
250 150 cors_100,100.bin
```

This computes the *image correspondences* from the optical flow for the reference view feature points `of/fpoints_100,100.json`. They get written into `cors_100,100.bin`. If there are multiple reference views, this process needs to be repeated for each one, by calling the program once for each file in `of/`.

In this example, the *horizontal outreach* is 250, and the *vertical outreach* is 150.

This process takes by far the most time in the calibration process. It will open each image file once. The *image correspondences* output file can have a `.bin` or a `.json` filename extension. If `.bin` is used, they are stored in a binary format that takes up less disk space, and can be read faster. This is useful for large datasets.
To copy the *image correspondences* from/to binary format, use

```
calibration/copy_cors cors_100,100.json cors_100,100.bin
```

Information about the *image correspondences* can be obtained using

```
calibration/cors_info parameters.json cors_100,100.bin
```

3.2.5 Merge image correspondences

The *image correspondences* computed for the different *reference views* should now be merged into one file, using

```
calibration/merge_cors cors_100,100_f2.bin cors_200,100_f2.bin cors_all.bin
```

If there are more than two reference views it should be called multiple times. The resulting file will still contain the information about the different reference views. This makes the following steps easier. All of the programs are aware that there can be features with different reference views in the image correspondences file.

3.2.6 Visualizing image correspondences

The resulting *image correspondences* can be visualized in two ways:

To see the all the feature points on one view, use

```
calibration/cg_cors_viewer_v cors_all.bin
```

It displays a graphical user interface where the view to show can be selected.

To see all the feature points of one feature, use

```
calibration/cg_cors_viewer_f cors_all.bin
```

It displays a graphical user interface where the feature to show can be selected. It displays a dot for each position of this feature (on a different view). The (backdrop) view to show can also be selected. With

```
calibration/cg_cors_viewer_f cors_100,100.bin closeup
```

it instead displays a closeup view of the image, with only the area where the feature points are placed. It can also show the corresponding depth map as overlay. Figures 2.3 and 2.4 were generated with this.

3.2.7 Feature point depths (if applicable)

The feature point depths can be added to the *image correspondences* using

```
calibration/read_feature_depths cors_all.bin 1
cors_all_with_depths.bin
```

It will open each (reprojected) depth map file once, which can also take a lot of time. The image correspondences with depth are stored into `cors_all_with_depth.bin` in this example. The output file can also be the same as the input file (then it will replace it).

An optional parameter (here `1`) indicates the margin of the window to look for a depth. With the value 1, it looks at the pixel obtained by rounding down the feature point position (floating point value), plus a margin of 1 pixels, forming a 3 × 3 pixel square window. It selects the minimal value in this window as the feature point depth.

3.2.8 Filtering image correspondences

The feature points should be filtered both automatically and manually. First use

```
calibration/cg_filter_features parameters.json cors_all.bin
cors_all_f.bin 125 75 use_depth
```

to filter out most obvious bad image correspondences. In this example 125 and 75 are the number of *expected feature points* in horizontal and vertical directions. It should be the *outreach* divided by two. (Because if the reference view is close to the border, only half of the outreach can be done). If `use_depth` is provided, it also checks the constancy of the feature depths. (There will be a slight linear increase in the depths, but there must be no jumps) It should be set, unless the calibration is done without depth maps. Some hardcoded parameters in `src/calibration/cg_filter_features.cc` probably need to be adjusted to get good results for a particular data set.

To filter out the remaining bad features, `cg_cors_viewer_f` should be used in `closeup` mode. The names of the features to remove should be noted. Then, use

```
calibration/remove_cors cors_all_f.bin cors_all_f2.bin
feat1003,feat1010,feat2110,feat3001
```

to filter out those features, and save the remaining image correspondences into the file `cors_all_f2.bin`.

3.2.9 Undistort image correspondences (if applicable)

If there is distortion (defined in the intrinsic parameters file), and the images were not undistorted before, the *image correspondences* can be undistorted now, with

```
calibration/undistort_cors cors_all_f2.bin
cors_all_f2_undist.bin intr.json
```

It puts each feature point to the position where it would be if the image had been undistorted before the optical flow computation. The *undistorted* image correspondences must be used for the subsequent steps.

3.3 Rotation estimation

As described before, the rotation estimation can be done using the slopes of the optical flow (when no depth maps are available), or using the feature point depths.

3.3.1 Measuring optical flow slopes

The optical flow slopes on the image correspondences can be measured using

```
calibration/cg_measure_optical_flow_slopes parameters.json cors_all_f2.bin
intr.json slopes.json
```

It will measure a horizontal and a vertical slope for each feature point from each reference view, and write it into `slopes.json`.

3.3.2 Visualizing optical flow slopes

To visualize optical flow slopes (actual and model), use

```
calibration/cg_slopes_viewer parameters.json intr.json slopes.json
```

where `slopes.json` are measured optical flow slopes. If there are no measured optical flow slopes, a feature points file can also be given as input (instead of `slopes.json`), such as `of/fpoints_100,100.json`.

It displays a graphical user interface where the model Euler angles can be adjusted, and the modelled slopes are displayed, along with the measured slopes. This can be used to manually estimate Euler angles that correspond to the measured optical flow slopes.

3.3.3 Optimizing optical flow slopes

To estimate a camera rotation $\mathbf{R}$ using the measured optical flow slopes, use

```
calibration/cg_rotation_from_fslopes intr.json slopes.json R.json
```

It will save the estimated rotation matrix into `R.json`.

3.3.4 Rotation from depths

To instead estimate the rotation using the feature points depths, use

```
calibration/cg_rotation_from_depths cors_all_f2.bin intr.json R.json
```

3.4 Straight depths

3.4.1 Aggregate feature point depths

To calculate the feature point *straight depths* using the feature points depths, use

```
calibration/cg_straight_depths_from_depths cors_all_f2.bin R.json
depths.json
```

3.4.2 Depth from disparity

To estimate the straight depths using only the relative scales of the feature points, use

```
calibration/cg_straight_depths_from_disparity cors_all_f.bin intr.json
R.json some_depths.json depths.json
```

The file some_depths.json needs to contain at least one measured *straight depth*, or more to get a better fit. If no depth maps are available, it can for example be obtained manually using laser distance measurement on one of the chosen feature points of a reference view.

This can also be used to complete the straight depths obtained from aggregating feature point depths: Two image correspondences files cors_all_f.bin and cors_all_fd.bin are maintained. The latter has been filtered with the use_depth option when using calibration/cg_filter_features, the former without it.

Then cg_straight_depths_from_depths is executed on cors_all_fd.bin. To also obtain straight depths for the additional image correspondences that remain in the file cors_all_f.bin, cg_straight_depths_from_disparity is now used, whereby the previously obtained straight depths are given as some_depths.json.

3.5 Camera positions

3.5.1 Relative camera positions

To compute the *relative camera positions*, use

```
calibration/cg_rcpos_from_cors parameters.json cors_all_f2.bin intr.json
R.json depths.json rcpos.json
```

The resulting rcpos.json file will contain the *relative camera positions* for all the reference views. There is no need to run the program multiple times.

3.5.2 Stitching

To stitch the relative camera positions together and obtain the final camera parameters, use

```
calibration/cg_stitch_cameras refgrid.json rcpos.json intr.json R.json
cams.json
```

Here refgrid.json is the *reference grid* file chosen as the first step. This needs to be done even if there is only one reference view. cams.json will contain the final camera parameters.

3.5.3 Visualization and export

To visualize the camera parameters, use

```
camera/visualize cams.json cams.ply world 0.3
```

It will generate the PLY file cams.ply containing a 3D visualization of the cameras in world space. 0.3 is the size of the cameras in this visualization.

To convert the camera parameters into the format and convention used by VSRS, use

```
camera/export_mpeg cams.json cams.txt
```