1. Introduction

1.1. Background

Camera calibration is a basic problem in computer vision, consisting of finding the relative position and pose of all the cameras to one another. This is important for inferring 3D structure [1], aligning camera arrays in computational photography [2], and more. Due to the recent release of the consumer-grade Microsoft Kinect™ depth sensor, there are also many vision algorithms that aim to take advantage of the fusion of RGB and point cloud scene structure. For example, in the problem of remote vital sign monitoring, careful aim has to be maintained at the subject’s chest [3]. Computer vision along with depth information have been used for this purpose [4].

The basic technique for conventional camera calibration is to image a 2-D target or a 3-D target in distinct orientations where image-to-image point correspondences can be easily obtained either manually or semi-automatically [5]. This target is commonly the checkerboard pattern, for which numerous corner extraction algorithms exist, given the rough manually selected location of the target region [6]. Recent improvements have also made checkerboard extraction automatic, e.g. [6,7].

For point cloud to color image calibration, the pinhole camera model gives the mapping from a homogeneous point in the world \( p = [x, y, z, 1]^T \) to homogeneous image coordinates \( p' = [u', v', 1]^T \) as

\[
wp = KR \begin{bmatrix} f & 0 & c_x \\ 0 & -f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_x' \\ r_y' \\ r_z' \\ t_x \\ t_y \\ t_z \end{bmatrix} P,
\]

where \( K \) is the intrinsic camera matrix, \( R \) is the camera rotation matrix, and \( t = -Rc \) with \( c \) being the pinhole location in world coordinates. To model the radial and tangential distortion function introduced by the lens, this linear model is usually concatenated with a truncated Taylor expansion.

Since depth cameras, like the Kinect™, can directly give values for world coordinates \( P \), it seems at first glance that it should be easier to obtain the calibration parameters of the model, but depth cameras present their own problems. First of all, it is not easy – even manually – to provide point cloud to RGB correspondences due to lack of visual information in the point cloud. These correspondences must come from structural corners that are present in the point cloud that are also visible in RGB. However, the second major problem with depth-camera-generated point clouds is that they provide extremely noisy and incomplete information at such corners and depth edges due to the measurement method (structured light). This creates the paradox that points which should be visible in both the color and depth images are exactly those whose information is missing in the point cloud.

Many algorithms have been developed to solve these problems and to calibrate a depth and color camera pair. In the field of time-of-flight (ToF) cameras, which measure distance by pulsing a light source and measuring the reflection delay, recent work includes \[8,9\]. In the field of structured light cameras, such as the Kinect™, recent work includes \[10–13\]. All of these methods require user
interaction in some form, usually marking points in the color or depth image manually.

1.2. Motivation for our work

The main drawback with all of the methods that we have found is that it is a tedious process. In our experience, calibration, especially extrinsic calibration, has to be performed often due to the sensitivity of camera systems to even the slightest nudge. This is exacerbated when the setup is on a mobile platform. In addition, recalibration has to be performed if the setup of the cameras is changed altogether. In our applications with remote vital sign monitoring, the setup is moved around a lot since it is on a mobile platform which is transported to various environments for testing, creating small changes in the orientation of the cameras which is problematic.

We recognize that once the intrinsic parameters have been found, there is no need to recalibrate them. Therefore, we present a calibration technique where the extrinsic parameters can be found very quickly, usually in under a minute, with no tedious work on the part of the user.

Using our method, not only can we mitigate the aforementioned paradox, wherein we find direct point correspondences between the point cloud and the color image to perform calibration, but this happens very rapidly, online, and without requiring the user to manually specify any point correspondences.

2. Methods

Our goal is to project points from the point cloud created by any depth camera into the color camera frame of reference. In order to do this, we need to estimate the 14 parameters of the pinhole projection model with lens distortion (up to 4th order lens distortion approximation):

\[ \Pi = \begin{bmatrix} f_x^D & s^D & c_x^D & t_x^D \\ f_y^D & s^D & c_y^D & t_y^D \\ 0 & 0 & 1 & 0 \end{bmatrix}, \]

where \( f_x^D, f_y^D, c_x^D, c_y^D, t_x^D, t_y^D, s^D, \) and \( s^D \) are the intrinsic depth camera parameters, \( x, y, \) and \( z \) are the real-world coordinates, \( t_x^D, \) and \( t_y^D, \) and \( s^D \) are the radial distortion coefficients, and \( v_1 \) and \( v_2 \) are the tangential distortion coefficients. The superscript \( D \) denotes a parameter of the depth camera, a superscript \( C \) denotes a parameter of the color camera, and a superscript \( D \rightarrow C \) denotes a transformation from the depth camera frame to the color camera frame.

In order to perform calibration, we use a color image and a point cloud obtained simultaneously. The point cloud can come from any source, e.g., a ToF camera, a laser range finder, a Kinect™. The target we use is a checkered, planar (as opposed to textured) square. We use such a target because it is simple to create and does not require any special materials. A flowchart of the automatic calibration process can be seen in Fig. 1.

2.1. Intrinsic parameters

2.1.1. Color camera intrinsic parameter estimation

Since color camera calibration is well-studied topic, we use the Bouguet toolbox [14] for MATLAB [15] with an automatic checkerboard-finding algorithm [7]. This allows us to keep the whole algorithm automatic while obtaining very precise intrinsic parameters and distortion coefficients for the color camera. This step eliminates the need to find \( f_x^C, f_y^C, c_x^C, c_y^C, t_x^C, t_y^C, s^C, \) and \( s^C \), since these are returned by the toolbox.

2.1.2. Depth camera intrinsic parameter estimation

For our work, we use OpenNI™ drivers, which create a point cloud from the Kinect’s™ disparity map. The point cloud we retrieve is created from a depth image using the Bouguet™ intrinsic parameters:

\[ x = \left( \frac{(u^d - c_x^C)}{f_x^C} \right) z, \]  \hspace{1cm} (2)

\[ y = \left( \frac{(v^d - c_y^C)}{f_y^C} \right) z, \]  \hspace{1cm} (3)

\[ z = z, \]  \hspace{1cm} (4)

where \([x, y, z] = P\) is a real-world coordinate, \([u^d, v^d] = p^d\) is a pixel coordinate in the depth frame of reference, \( c_x^C \) and \( c_y^C \) define the principal point in the depth frame of reference, and \( f_x^C \) and \( f_y^C \) are the focal length of the depth camera.

The OpenNI™ drivers return a 3-dimensional image as a 3-channel matrix, where each \((u, v)\) point in the matrix has a corresponding \((x, y, z)\) value in the 3 channels. Since the Kinect™ is only able to directly acquire the \(z\)-channel, this means that the \(x\)- and \(y\)-channels are calculated according to Eqs. (2) and (3).

Since we get an image of \(x, y, z, u^d,\) and \(v^d\) values, we estimate the intrinsic depth camera parameters \((c_x^D, c_y^D, f_x^D, f_y^D)\) that were used to create the point cloud. We do this by minimizing the reprojection error between points in the point cloud that we retrieve and points in a point cloud that we create using only the \(z\) values of the retrieved point cloud and Eqs. (2) and (3):

\[ \arg\min_{c_x^D, c_y^D, f_x^D, f_y^D} \sum_{i=1}^{n} \left[ \left( \frac{(u_{ij}^d - c_x^D)}{f_x^D} \right) z_i - x_i \right] + \left[ \left( \frac{(v_{ij}^d - c_y^D)}{f_y^D} \right) z_i - y_i \right]. \]  \hspace{1cm} (5)

where \(i\) and \(j\) index the image.

We assume for our work that the distortion coefficients for the depth camera are all zero. In the work of Smisek et al. [10] and Herrera et al. [12], it was shown that there is indeed a distortion in the Kinect's™ disparity map. The point cloud we retrieve is created using the Bouguet™ intrinsic parameters:

\[ x = \left( \frac{(u^d - c_x^C)}{f_x^C} \right) z, \]  \hspace{1cm} (2)

\[ y = \left( \frac{(v^d - c_y^C)}{f_y^C} \right) z, \]  \hspace{1cm} (3)

\[ z = z, \]  \hspace{1cm} (4)

where \([x, y, z] = P\) is a real-world coordinate, \([u^d, v^d] = p^d\) is a pixel coordinate in the depth frame of reference, \( c_x^C \) and \( c_y^C \) define the principal point in the depth frame of reference, and \( f_x^C \) and \( f_y^C \) are the focal length of the depth camera.

The OpenNI™ drivers return a 3-dimensional image as a 3-channel matrix, where each \((u, v)\) point in the matrix has a corresponding \((x, y, z)\) value in the 3 channels. Since the Kinect™ is only able to directly acquire the \(z\)-channel, this means that the \(x\)- and \(y\)-channels are calculated according to Eqs. (2) and (3).

Since we get an image of \(x, y, z, u^d,\) and \(v^d\) values, we estimate the intrinsic depth camera parameters \((c_x^D, c_y^D, f_x^D, f_y^D)\) that were used to create the point cloud. We do this by minimizing the reprojection error between points in the point cloud that we retrieve and points in a point cloud that we create using only the \(z\) values of the retrieved point cloud and Eqs. (2) and (3):

\[ \arg\min_{c_x^D, c_y^D, f_x^D, f_y^D} \sum_{i=1}^{n} \left[ \left( \frac{(u_{ij}^d - c_x^D)}{f_x^D} \right) z_i - x_i \right] + \left[ \left( \frac{(v_{ij}^d - c_y^D)}{f_y^D} \right) z_i - y_i \right]. \]  \hspace{1cm} (5)

where \(i\) and \(j\) index the image.

We assume for our work that the distortion coefficients for the depth camera are all zero. In the work of Smisek et al. [10] and Herrera et al. [12], it was shown that there is indeed a distortion in the depth image. However, we did not find significant improvement

Fig. 1. Flowchart of algorithm.
for our camera using Smisek’s method, and Herrera’s method is particular to the Kinect™ and is performed on the disparity image, which we did not use. Herrera’s method can potentially improve our results when used with the Kinect™ and can be easily added into the algorithm, but again, we aimed to keep everything as general as possible.

2.2. Extrinsic parameters

The main goal of our work is to accurately and automatically find the extrinsic parameters between a color camera and a depth camera, i.e., \( t_{x}^{D-C}, t_{y}^{D-C}, t_{z}^{D-C}, \psi_{R}^{D-C}, \theta_{R}^{D-C}, \) and \( \phi_{R}^{D-C}. \)

2.2.1. Target corner estimation in color image

We start our algorithm by finding the target in a color image. To do that, we use the checkerboard pattern on our target. Using an automatic checkerboard corner finding algorithm [7], we find all the internal corners on our pattern. We then keep only the four outermost corners and number them counterclockwise, starting with the top left, creating the set \( \{ P_{C} = (u_{1}, v_{1}), \ldots, u_{s}, v_{s} \} \). The result of this automatic corner detection can be seen in Fig. 2.

2.2.2. Target corner estimation in point cloud

To find the target corners in the point cloud (\( P \)), we have to compensate for the fact that data at edges and corners do not exist in \( P \). Since we know that the target is on a plane, \( (x, y, z) \) coordinates can be extrapolated at any \( (u, v) \) value, given knowledge of the plane.

Therefore, we start by finding the strongest plane in \( P \) using an adaptation of the Hough line-finding algorithm [16]. Since the dominant plane in most scenes is a wall, we first eliminate it by performing a depth segmentation, where we discard all points with \( z \) value greater than \( \zeta \), determined by the geometry of the room. To create a 2D depth mask \( (l_{d}) \), we project the resulting point cloud using \( \mathbf{R} = I, \mathbf{t} = \mathbf{0}, \) and \( (t_{x}^{D}, t_{y}^{D}, t_{z}^{D}, \psi_{R}^{D}) \) are the intrinsic parameters of the depth camera (see Fig. 3(b)). If a point is on the plane, it satisfies the following equation:

\[
\rho = x \cos(\theta) \sin(\phi) + y \sin(\theta) \sin(\phi) + z \cos(\phi),
\]

where \( \rho \) is the perpendicular distance from the origin to the plane, \( \theta \) is the azimuth angle, and \( \phi \) is the zenith angle, and these together are the Hough voting space.

To perform voting, we segment the parameter space into a number of orthotopical bins. For each \( (x, y, z) \) in \( P \), we increment the bin count for every bin that satisfies eq. (6). For speed and accuracy, we employ a coarse-to-fine approach for the bin size in practice, where the initial full parameter space is pared down at each level of granularity based on the fullest bins in the previous level.

Once we have the best-fit plane (Fig. 3(c)), we segment \( P \) based on this plane, keeping only those points which have perpendicular projection distance of less than \( \epsilon \), as seen in Fig. 3(d). This creates a new point cloud, \( P_{d} \), and likewise a new depth mask, \( l_{d} \). After this segmentation, there might still be points that do not belong to the target but are coplanar with it, so we find the largest connected component and eliminate the rest. This leaves only the target in \( P_{t} \).

Ultimately, we are looking for the corners of the target in the point cloud to create point correspondences with the color image. However, the corners are not visible in \( P_{t} \) and \( l_{d} \) because of the aforementioned paradox in Section 1.1. Therefore, we implement a nonlinear optimization scheme to find the corners.

The optimization problem is

\[
\text{arg min } f(p_{1}, p_{2}, p_{3}, p_{4}) = \sum_{i=1}^{4} D(u_{i}, v_{i})
\]

s.t. \( \|P_{2} - P_{1}\| = l, \|P_{3} - P_{2}\| = l, \|P_{4} - P_{3}\| = l, \|P_{3} - P_{1}\| = l\sqrt{2}, \|P_{4} - P_{2}\| = l\sqrt{2}, \)

where \( p_{1}, p_{2}, p_{3}, \) and \( p_{4} \) are points in \( l_{d} \) ordered counterclockwise, \( P_{t} \) are points in \( P_{t} \) corresponding to \( P_{c} \) as determined by Eqs. (1) and (6), \( l \) is the target edge length, and \( D(u_{i}, v_{i}) \) is the Euclidean distance from pixel \( (u_{i}, v_{i}) \) to the edge of an object (in this case the target in Fig. 3(d)) [17]. We allow the sum of the edges to have up to a length \( \kappa \) of slack (implemented as Lagrange multipliers with a slack variable for Eq. (8)). In plain English, Eq. (7) says that we are minimizing the distance from each corner to the edge of the target in \( l_{d} \), while at the same time we are constrained by the true size of the target, such that the edges and diagonals must be very close to their true lengths [Eq. (8)]. The minimization [Eq. (7)] returns \( \theta = (p_{1}, p_{2}, p_{3}, p_{4}) \).

To solve Eqs. (7) and (8), we used the Sequential Quadratic Programming (SQP) algorithm [18], implemented in MATLAB [15]. Fig. 4(a) shows the cost function (using the same image as in Fig. 3), with the optimized corner locations shown by green circles. Fig. 4(b) shows \( l_{d} \) with the corners denoted by green circles. It is important to note that the corner locations are not actually on the “target” in \( l_{d} \). This is because the “target” in \( l_{d} \) is smaller than the true target due to missing data at edges and corners. Our method finds the true corners despite this problem.

After the optimization, we project the set \( P_{t} \) onto the best-fit plane in order to get the real-world coordinates of the corners of the target. Then, we translate the corners towards the center of the plane by \( i \sqrt{2} m \), where \( i \) is the length of a side of each square on the checkerboard (in meters), so that the translated corners are in the same location as that marked by \( P_{c} \) in the color image.

---

**Fig. 2.** (a) Original RGB image with target, and (b) RGB image with detected corners shown as red circles. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
Finally, these points are projected to the RGB image frame using the current estimate of the intrinsic and extrinsic parameters. This projected set of points is denoted as $P_D$. Given perfect calibration, $P_D$ should be the same as $P_C$. The process to make them as close as possible is described next.

2.2.3. Projection parameter estimation

Given that we want to estimate $M = \{t_D^x, t_D^y, t_D^z, \theta_D^x, \theta_D^y, \theta_D^z, \psi_D^x, \psi_D^y, \psi_D^z\} \subseteq \Pi$ (since the intrinsic parameters are known), we need $N \geq |M|$ point correspondences to create at least as many equations as unknowns. For every valid image in which we obtain $P_C$ and $P_D$, we concatenate them to $A_C$ and $A_D$, respectively. Once we have obtained $|M|$ point correspondences ($|M|/4$ valid images), we are able to perform parameter estimation. We have $|M|$ parameters, but use $N > |M|$ points for noise suppression. To find the best parameters, we perform a nonlinear minimization.

The optimization problem is a minimization of the reprojection error:

$$c = \min_M \sum_{i=1}^{N} ||A_D(i) - A_C(i)||^2,$$

where, $A_D$ is a function of the parameters $M$. For the minimization, we use the Levenberg–Marquardt [19] algorithm with RANSAC [20] for outlier rejection (thus pruning $A_D$ and $A_C$).

The initial parameters are obtained as coarse estimates based on rough knowledge of the camera orientation. After one optimization is performed, the optimized parameters become the initial values for the next iteration. For each subsequent valid image, the last $N = |A_C| = |A_D|$ points are used for optimization again using eq. (9), as long as $N > |M|$. In this way, optimization occurs for every valid image after the first $|M|/4$ valid images, as long as $A_D$ and $A_C$ are not pruned too much by RANSAC.

2.3. Result validation

In order to verify our results and compare them objectively to other methods, we expanded on an idea presented in [12], wherein
we used a ground truth target composed of three perpendicular planes. The corner of this target is visible in both the point cloud (after some processing described below) and in the color image.

2.3.1. Ground truth corner in color image

To find the corner of the target in the color image, we created an automatic technique to avoid any possible bias that might occur while marking the images manually. The only action the user performs is selecting a bounding box around the corner location.

First, all of the corners in the sub-image denoted by the bounding box are found using a Harris corner detector [21]. Then, the three strongest lines in the image are found using the Hough line detection algorithm [16]. Each of these lines is parameterized by its angle from the horizontal ($\theta_i$) and its distance from the origin ($\rho_i$), where the subscript $L$ is used to distinguish between the parameters used in Section 2.2.2 and those used for finding lines here. Then, the intersection of the three lines is found as the least-squares solution of:

$\begin{bmatrix} \cos(\theta_1) & \sin(\theta_1) \\ \cos(\theta_2) & \sin(\theta_2) \\ \cos(\theta_3) & \sin(\theta_3) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \rho_{11} \\ \rho_{12} \\ \rho_{13} \end{bmatrix}$

where each line is parameterized by $\theta_i$ and $\rho_i$, for $i = 1, 2, 3$.

The corner of the target in the color image is then found as the corner that is closest to the line intersection $[u, v]^T$, denoted by $[u', v']^T$. To avoid errors, the user inspects each location $[u', v']^T$, and if $[u', v']^T$ is not close to the true corner due to an error in the Hough line intersection, the user manually selects the location he thinks is closest to the true corner, and $[u', v']^T$ is chosen as the Harris corner closest to that manual location. In this way, the actual corner location is never marked by the user explicitly, which avoids bias. This process can be seen in Fig. 5.

2.3.2. Ground truth corner in point cloud

To localize the corner in the point cloud, we fit a plane to each of the three sides using the method described in Section 2.2.2. Each plane can be parameterized by its normal ($\mathbf{n} = [n_x, n_y, n_z]$) and its distance from the origin ($r$). We found the intersection of these three planes as the least-squares solution of:

$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix}$

where the planes are parameterized by $\mathbf{n}_i$ and $r_i$, for $i = 1, 2, 3$. The target corner in the point cloud is denoted as $[x', y', z']^T$. This process can be seen in Fig. 6.

2.3.3. Reprojection error

To validate our algorithm, we projected the corners found in the point cloud $([x', y', z']^T)$ to the color frame of reference and found their distance to the corners marked in the color image $([u', v']^T)$. The projection parameters $P$ were obtained as detailed above.

3. Results and discussion

We have implemented our algorithm in MATLAB [15] as the CalibRT Toolbox. The toolbox is freely and openly available at [22].

3.1. Implementation results

We present in Fig. 7 a prototypical example of our algorithm. First, we find the corner locations in the color image ($P_C$) using the algorithm in Section 2.2.1. Then, we find the corners of the target in the depth mask ($P_D$) by minimizing eq. (7). Finally, we project the color corners onto the best-fit plane in order to minimize the distance to the corners in the point cloud.

Fig. 5. (a) Color image of ground truth target, (b) Harris corner locations on user-selected sub-image, (c) Hough lines in sub-image, (d) Least-squares Hough lines intersection (green star) superimposed on Harris corners in sub-image, (e) True corner location in sub-image, and (f) True corner location on ground truth target.
Fig. 6. (a) Ground truth target in a depth image, and (b) Point cloud of ground truth target (blue) with three best-fit planes (green) and their intersection (red).

Fig. 7. Algorithm example. (a) The original color image with superimposed corners ($P_C$) shown as green circles found by the algorithm in Section 2.2.1. (b) The original depth mask $I_m$ with superimposed corners ($P_I$) shown as green circles found by minimizing Eq. (7). (c) The plane-segmented depth mask ($I_d$) with superimposed corner locations ($P_I$) shown as green circles. (d) The point cloud ($P$) with superimposed target corner locations (green circles) and the locations of the corners corresponding to the color checkerboard corners (red circles).
After five images have been captured (with the target in a different location or orientation in each image), we perform our first calibration optimization. Fig. 8 shows the result of aligning our color image with the point cloud and segmenting the color image in depth to isolate the target. Fig. 8(a) is the alignment using the initial parameters, which assumed no rotation and very coarse guesses to translation, and Fig. 8(b) is the alignment after the first optimization. It is clear that the optimized extrinsic parameters are very good after just one iteration. Depending on the application and how accurate the calibration has to be, the user can continue to refine the estimates by taking more images and thus performing more iterations of the optimization. However, subsequent iterations do not produce significant visual improvements.

3.2. Discussion

In order to further validate our algorithm objectively, we present several metrics of interest, and compare our results with a method at the forefront of depth-color camera calibration, the Kinect™ Calibration Toolbox for MATLAB [12]. To use the toolbox, we captured data using the OpenKinect libfreenect drivers in order to get a disparity image, since Herrera’s toolbox requires that for optimal performance.

For all the tests, we use a 1024 × 768 pixel color camera and a 640 × 480 pixel depth camera. For alignment and reprojection, we downsample everything to a 320 × 240 pixel image. However, all processing is first performed on the original images to avoid any errors due to aliasing and poor resolution.

We present errors in terms of pixels on a 320 × 240 pixel image, denoted as $e(p)$. While this is easy to visualize and interpret, it is not too informative. A more informative metric is the normalized error:

$$
\bar{e} = \frac{e(p)}{f_c \cdot \gamma}
$$

where $f_c$ is the focal length of the color camera and $\gamma$ is a scaling factor depending on the downsampling from the full-size color image to the smaller one. For us, $f_c = 1770$ pixels, as determined by the method in Section 2.1.1, and $\gamma = 240/768$. Using the normalized error, it is possible to get the true error in meters.

![Fig. 8. (a) Initial calibration alignment, and (b) Final calibration alignment.](image)

![Fig. 9. (a) Reprojection error during calibration process, as a function of the number of images that have been processed, and (b) Reprojection error on ground truth data as a function of the number of images that have been processed for training.](image)

<table>
<thead>
<tr>
<th>Time Statistics</th>
<th>Find color corners</th>
<th>Find XYZ corners</th>
<th>Find calibration params</th>
<th>Align color and XYZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. time (s)</td>
<td>0.31</td>
<td>0.97</td>
<td>0.0218</td>
<td>0.025</td>
</tr>
<tr>
<td>Std. dev. (s)</td>
<td>0.0084</td>
<td>0.1449</td>
<td>0.0025</td>
<td>0.0028</td>
</tr>
</tbody>
</table>
\(e(P) = \hat{e} \cdot z,\)  

(13)

where \(z\) is the distance from the camera in meters.

### 3.2.1 Accuracy

We do not judge performance by the actual values of the extrinsic parameters, because it is nearly impossible to get a ground truth on them; we would need to know the exact location of the focal centers of both cameras in order to truly measure the displacements, and the rotations are even more difficult. Instead, we use the values of the reprojection errors on the ground truth, as was described in Section 2.3, to determine accuracy.

Our algorithm converges very quickly to good parameters, as evidenced by reprojection error values (and visually in Fig. 8). Fig. 9(a) shows how quickly the reprojection error reaches a small value. After the first calibration at five images, the reprojection error varies up and down a little, but the range of the oscillation is not big, and is due to the random nature of the RANSAC algorithm. It can be seen that the reprojection error tends to decrease with more and more training images. However, it is important to note that this reprojection error does not mean that the calibration parameters are good, because the RANSAC algorithm eliminates corner correspondences that create high reprojection errors.

To truly validate our calibration, we use 20 images of a ground truth target as was described in Section 2.3, with ground truth corner positions in several locations in all three dimensions to avoid bias. Fig. 9(b) shows the reprojection error on the ground truth images as a function of the number of training images used. It is clear that the reprojection error reaches a good value very quickly (around image 20) and varies slightly (by \(\sim 0.15\) pixels) thereafter.

Using the calibration we calculated from Herrera’s toolbox for 60 images and testing on 20 ground truth images, we obtained a ground truth reprojection error of \(e(p') = 4.0738\) pixels (\(\hat{e} = 7.407e^{-3}\)). After running our algorithm for 60 images, our reprojection error on the ground truth was \(e(p') = 1.5663\) pixels (\(\hat{e} = 2.832e^{-3}\)).

### 3.2.2 Speed and automaticity

Table 1 shows the speed of computation (on an Intel® Xeon® 3.33 GHz W3580 CPU with 12 GB RAM) for each of the components of our algorithm. Most of the processing was performed in MATLAB R2012b (32-bit) with the help of MEX (MATLAB executable) functions and an OpenCV [23] wrapper. MATLAB is an interpreted language as opposed to a compiled one, which significantly slows down operation. Even so, the execution time is fast enough to make this algorithm able to be used online. It is not real-time, as not every frame is processed in 0.33 s. However, at approximately 1.5 s per frame, this is much faster than any other method that we have encountered. Fig. 10 shows two timing graphs, where Fig. 10(a) shows how consistent the run time is as more and more images are added, and Fig. 10(b) shows how quickly the ground truth reprojection error decays. The reason for the difference between the two plots in Fig. 10(a) is that the “Online” version produces plots for the user to see what is happening, which adds about 1 s per frame.

We once again compare our algorithm to Herrera’s in terms of speed for calibration using 30 images for both algorithms. For Herrera’s method, just as in ours, there are three steps: find the checkerboard in the color image, find the plane in the depth image, and calibrate the parameters. To find the checkerboard pattern, we modified Herrera’s code to use the RADDODC [7] corner finder, since the one currently in the code did not work for us. With this modification, it took about 30 s to find the checkerboards in all the images. To find the planes, the user has to segment the image by hand, and this took about 3 min for our images. Finally, the calibration took about 2 min. The whole procedure therefore took about 5.5 min and resulted in a reprojection error on the ground truth data of \(e(p') = 5.6483\) pixels (\(\hat{e} = 10.328e^{-3}\)). Our method processes 30 images in about 74 s online (45 s offline) with a ground truth reprojection error of \(e(p') = 1.6022\) pixels (\(\hat{e} = 2.897e^{-3}\)).

In order to avoid potential errors from inserting different code (for checkerboard finding) into Herrera’s algorithm, we reran the calibration using the original code. This meant marking corners in the color image by hand, which took about 6.5 min for 30 images. The whole calibration time was about 11 min, and resulted in a reprojection error on the ground truth of \(e(p') = 5.9988\) pixels (\(\hat{e} = 10.969e^{-3}\)).

It should also be mentioned that our intrinsic calibration occurs offline, since that has to happen only once for a given camera, while it occurs every time for Herrera’s toolbox. The clear advantage of our algorithm is that for a given pair of cameras, changing their setup does not have to result in a lengthy recalibration. The intrinsic parameters remain the same, and our algorithm quickly finds the extrinsic parameters. However, for a fair comparison, we state that our intrinsic color calibration took 1 min and finding the intrinsic depth parameters took 30 s.
3.3. Implementation details

For the Hough plane voting procedure (Section 2.2.2), we implemented the coarse-to-fine method by always using 6 equally divided bins for $\theta$ and $\phi$, and 1001 bins for $\rho$. After each iteration, we divide the range of the angular voting space by a factor of 2, centered around the maximum bin in the previous iteration. This procedure is performed 10 times, bringing the angular resolution to $2\pi/(6 \cdot 2^{10})$ radians and the distance resolution to 10/1001 m. The numbers of bins and the number of iterations were determined experimentally as values that simultaneously minimize the cost function and minimize the computational intensity.

For the plane segmentation, we used $\epsilon = 1$ cm and $\kappa = 1$ cm. The slack variable $\kappa$ did not impact the cost function value (Eq. (7)) very much, as long as it is between 0 and $\sim$5 cm. The segmentation length $\epsilon$ was more critical, creating bad results if it was under $\sim$1 cm. This is because the data in the point cloud is noisy and can have irregularities. Therefore, we chose the smallest value that still gave good results.

To initialize the corner locations (Section 2.2.2), $p_1, \ldots, p_n$, we use the plane-segmented depth mask $l_i$ and place the initial corners as the corners of the largest inscribed rectangle in the mask.

For our implementation, we chose $M = \left( t_{\phi}^{D-C}, t_{\theta}^{D-C}, \phi_{\phi}^{D-C}, \phi_{\theta}^{D-C}, \phi_{\phi}^{R-C}, \phi_{\theta}^{R-C} \right)$, as was mentioned in Section 2.2.3. We perform optimization (Section 2.2.3) once we have acquired 20 point correspondences (5 images).

4. Conclusion

We have presented a fully automatic, fast, and online camera calibration algorithm. We have shown that we are able to calibrate a depth camera and a color camera quickly and robustly, using minimal equipment. We demonstrated our algorithm's robustness in real conditions. In addition, we have shown our parameter estimates quickly approach physically accurate calibration values as evidenced by the reprojection error in the ground truth tests and also by visual inspection. Altogether, our algorithm has proven to be faster and more accurate than other leading algorithms, to our knowledge.

Although we avoided optimizing the intrinsic parameters of the depth camera, we plan to work on that in the future. Our goal is to again create a fully automatic algorithm that can calibrate the depth camera independently, as is being done with the color camera, so that only the extrinsic parameters need to be found.

References