Published in Image Processing On Line on 2016–12–18.
Submitted on 2015–01–29, accepted on 2016–11–21.
ISSN 2105–1232 © 2016 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2016.130

An Iterative Optimization Algorithm for Lens Distortion Correction Using Two-Parameter Models

Daniel Santana-Cedrés[1], Luis Gomez[2], Miguel Alemán-Flores[1], Agustín Salgado[1], Julio Esclarín[1], Luis Mazorra[1], Luis Alvarez[1]

[1] CTIM (Centro de Tecnologías de la Imagen),
Departamento de Informática y Sistemas
Universidad de Las Palmas de Gran Canaria, Spain
({dsantana,maleman,asalgado,jesclarin,lmazorra,lalvarez}@ctim.es)
[2] CTIM (Centro de Tecnologías de la Imagen),
Departamento de Ingeniería Electrónica y Automática
Universidad de Las Palmas de Gran Canaria, Spain
(lgomez@ctim.es)

This IPOL article is related to a companion publication in the SIAM Journal on Imaging Sciences:
Santana-Cedrés D., Gomez L., Alemán-Flores M., Salgado A., Esclarin J., Mazorra L. and Alvarez, L. "Invertibility and Estimation of Two-Parameter Polynomial and Division Lens Distortion Models" *SIAM Journal on Imaging Sciences*, vol. 8, no. 3, pp. 1574–1606, 2015. http://dx.doi.org/10.1137/151006044

Abstract

We present a method for the automatic estimation of two-parameter radial distortion models, considering polynomial as well as division models. The method first detects the longest distorted lines within the image by applying the Hough transform enriched with a radial distortion parameter. From these lines, the first distortion parameter is estimated, then we initialize the second distortion parameter to zero and the two-parameter model is embedded into an iterative nonlinear optimization process to improve the estimation. This optimization aims at reducing the distance from the edge points to the lines, adjusting two distortion parameters as well as the coordinates of the center of distortion. Furthermore, this allows detecting more points belonging to the distorted lines, so that the Hough transform is iteratively repeated to extract a better set of lines until no improvement is achieved. We present some experiments on real images with significant distortion to show the ability of the proposed approach to automatically correct this type of distortion as well as a comparison between the polynomial and division models.

Source Code

The source code, the code documentation, and the online demo are accessible at the IPOL web page of this article[1] In this page, an implementation is available for download. Compilation and usage instructions are included in the README.txt file of the archive.

Keywords: lens distortion; wide-angle lens; Hough transform; line detection; iterative optimization

[1] https://doi.org/10.5201/ipol.2016.130

1 Introduction

Radial distortion is considered as the most important type of distortion for cameras. It causes barrel distortion at short focal lengths as well as pincushion distortion at longer focal lengths. This is mainly due to the imperfection of the lens and the misalignment of the optical system. This type of distortion can be embedded in the well-known pinhole camera model [12] by means of a distortion model.

The general equation of a lens distortion model is given by the equation

$$\begin{pmatrix} \hat{x} - x_c \\ \hat{y} - y_c \end{pmatrix} = L(r) \begin{pmatrix} x - x_c \\ y - y_c \end{pmatrix}, \tag{1}$$

where (x_c, y_c) represents the distortion center, (x, y) is a point in the image domain, $(\hat{x}, \hat{y})$ is the transformed point, $r = \|(x, y) - (x_c, y_c)\|$, and $L(r)$ represents the shape of the distortion model. Two radial lens distortion models are the most frequently applied in computer vision due to their excellent trade-off between accuracy and easy calculation: the polynomial model and the division model. The polynomial model, or simple radial distortion model [16], is formulated for two parameters as

$$L(r) = 1 + k_1 r^2 + k_2 r^4. \tag{2}$$

Two-parameter models have been used in the literature due to their simplicity and accuracy ([3], [18]). In [3], an algebraic method is proposed which is suitable for correcting significant radial distortion. Moreover, it is highly efficient in terms of computational cost. An on-line demo of the implementation of this algebraic method can be found in [4].

The division model was initially proposed in [20], but has received special attention after the research by Fitzgibbon [13]. For the case of two parameters, it is formulated as

$$L(r) = \frac{1}{1 + k_1 r^2 + k_2 r^4}. \tag{3}$$

The main advantage of the division model is the requirement of fewer terms than the polynomial model to cope with images showing severe distortion. Therefore, the division model seems to be more adequate for wide-angle lenses (see a recent review on distortion models for wide-angle lenses in [17]). The work we present in this paper can be considered as an extension of the IPOL paper in [1], where we studied the case of the one-parameter division model. The main novelties with respect to [1] is that in this paper we use a two-parameter lens distortion model, the distortion center location is optimized and we study the problem in the case of polynomial as well as division models. To deal with two-parameter models requires a different approach concerning parameter estimation.

For both models, $L(r)$ can be estimated by considering that 3D lines must be projected onto 2D straight lines, and minimizing the distortion error, which is given by the sum of the squares of the distances from the points to the lines [11]. In [5], [7] and [22], the user must identify some known straight lines, so that no calibration pattern is required and the methods are robust and independent of the camera parameters. However, these methods are slow when we deal with a large set of images. Different strategies to find out aligned structures in an image are possible. In [10], a parameterless method to detect segments which guarantees that no more than one segment on the average will be wrongly detected (*false positive*) is discussed. This work is based on the Helmholtz principle which states that any observed (*in an image*) geometric structure is perceptually meaningful if the expectation of its occurrences (*false alarms*) is small in a random image. This approach has been extended in [14, 15], where the authors introduce the LSD (Line Segment Detector) method. This method detects segments in an image in a very effective and fast way. In our case, as we deal with distorted lines, we use an extension of the Hough transform which allows extracting aligned structures

distorted by a one-parameter distortion model. This method extends the standard Hough space by including a distortion parameter in the voting matrix, which results in a better recognition of the existing straight lines and has become a usual approach in automatic lens distortion estimation.

A fully automatic distortion correction method, which also embeds the radial distortion parameter into the Hough transform for a better detection of straight lines, is presented in Lee et al. [19]. It is applied to endoscopic images captured with a wide-angle *zoomable* lens. In this case, a one-parameter division model is used to deal with the strong distortion caused by the wide-angle lens. The method is adapted to include the effects related to the variation of the focal length due to zooming operations. It is intended for real time applications, once mapped to a GPU computing platform, and the distortion parameter is estimated by optimizing the Hough entropy in the image gradient space.

In [2], an efficient method to detect lines in images showing significant lens distortion is discussed. Once a set of straight lines has been detected, an algebraic lens distortion model with two distortion parameters is applied to estimate and correct the distortion. The method is inspired by Cucchiara et al. [9] and states the problem in a new three-dimensional space, which includes orientation, distance to the origin and also radial distortion. It is tested on a set of real images and, for all the cases, the lens distortion is effectively removed. Furthermore, it outperforms the results obtained using the standard Hough transform. However, since the polynomial distortion model is used, the method fails in the case of a strong distortion produced by wide-angle lenses, where the division model would provide better performances.

In this work, we deal with division and polynomial models to correct the radial distortion in images. Based on the Hough voting scheme, we define a way to automatically compare the accuracy of both models. One of the main differences with the models mentioned above is that we use two-parameter models which, as discussed below, perform better when dealing with strong distortions caused by wide-angle lenses. In order to increase the number of straight lines which are extracted, a nonlinear iterative optimization method is applied to *refine* the candidate values for the distortion parameters. Not only is the proposed method useful to detect lines in single images, but it may also be applied to a video sequence to correct the frames through a tracking process. Algorithm 1 and Figure 1 summarize the main stages of the proposed method.

Algorithm 1: Summary of the lens distortion method described in Figure 1

```
check_input_params( input_params );

// STAGE 1:  Detect edges with Canny
contours ← Canny( input_image, ... ) ;                                  // Algorithm 2
edge_cleaned ← Clean_contours( contours ) ;          // Cleaning process, Algorithm 3

// STAGE 2:  Detect lines with the improved Hough transform
i_primitive ← Hough( edge_cleaned, current_model ) ;                    // Algorithm 4

// STAGE 3:  Apply the iterative optimization process
final_model ← iterative_optimization( edge_cleaned ) ;                  // Algorithm 5

// STAGE 4:  Correct the image distortion using the estimated model,
   Algorithm 8
undistorted ← undistort_image_inverse( input_image, final_model ) ;  // Algorithms 22, 23
undistorted.write;
```

In the first stage of the algorithm, we extract the edges using the Canny edge detector. The thresholds for the Canny algorithm are provided in terms of percentiles of the gradient norm of the

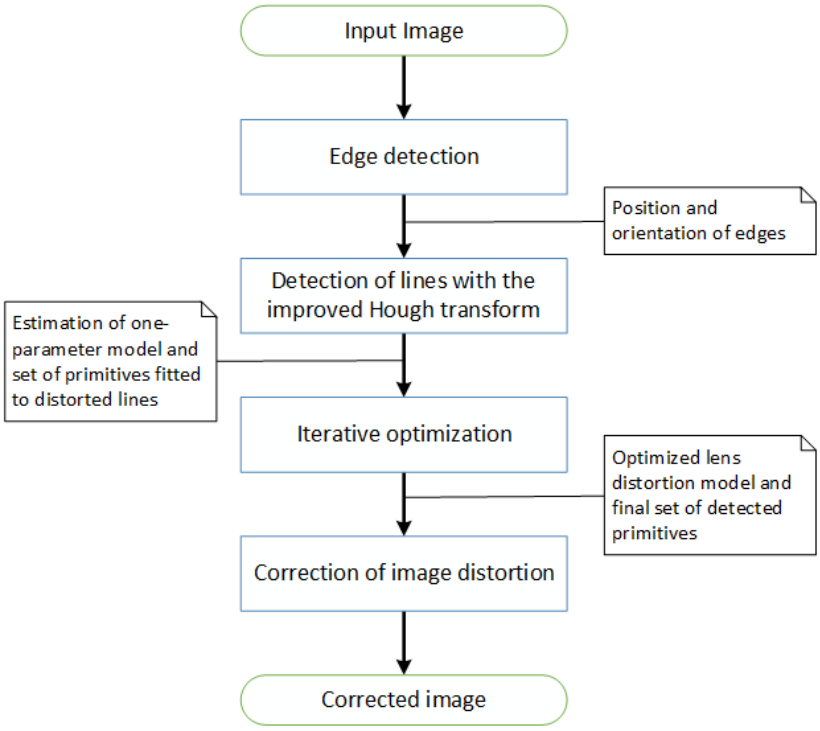


Figure 1: Summary of the lens distortion method.

image. Moreover, for each edge point, we estimate the edge orientation using the image gradient. As we are looking for aligned structures, we introduce an edge cleaning procedure to reduce the number of "locally non-aligned" edge points. To do that, we remove isolated edge points, edge points which present a high variation of edge orientation in a neighborhood and, optionally, to speed up the algorithm (especially for large images), we can reduce the number of edge points in each neighborhood by keeping in each neighborhood the point with lowest edge orientation variation.

In the second stage, we extract a one-parameter distortion model and the most voted lines using the extended Hough transform and the method proposed in [2].

In the third stage, a two-parameter model is embedded into an iterative nonlinear optimization process to improve the one-parameter solution obtained in the previous step. This minimization aims at reducing the distance from the points to the lines, adjusting two distortion parameters as well as the coordinates of the center of distortion. Furthermore, this allows detecting more points belonging to the distorted lines, so that the Hough transform is iteratively repeated to extract a better set of lines until no improvement is achieved. Finally, in the fourth stage of the algorithm, we correct the lens distortion of the image using the estimated model.

This paper is organized as follows: in Section 2, we present the details of the algorithm we propose. In Section 3, we study the complexity of the algorithm. In Section 4, we show some experiments to illustrate the performance of the proposed method, including a comparison between division and polynomial models. Finally, in Section 5, we present our main conclusions.

2 The Algorithm

2.1 Stage 1: Edge Detection using the Canny Method

In the first stage, we use the Canny edge detector to extract the collection of edge points used to estimate the distorted lines. The Canny edge detector, described in [8], tries to fulfill three conditions:

good detection, good location and only one response to a single edge.

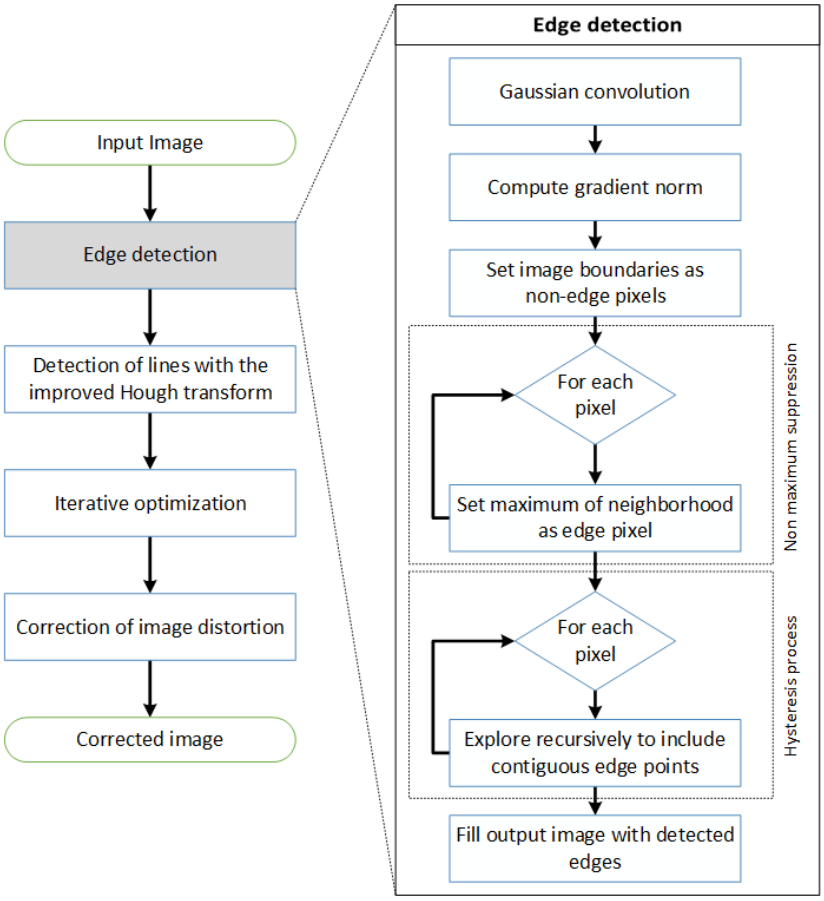


Figure 2: Edge detection using the Canny method.

Our implementation of the Canny algorithm (see Figure 2) can be divided into the following steps:

1. **Gaussian convolution:** First, the input image is smoothed using an approximation of the recursive filter proposed in [6]. This is a fast approximation to the Gaussian convolution.

2. **Image Gradient:** With the smoothed image, we compute the gradient using two 3×3 convolution masks that satisfy that the gradient norm is invariant under rotations of 45 degrees

$$\frac{\partial I}{\partial x} = 0.5 \begin{bmatrix} -\frac{2-\sqrt{2}}{2} & 0 & \frac{2-\sqrt{2}}{2} \\ -(\sqrt{2}-1) & 0 & \sqrt{2}-1 \\ -\frac{2-\sqrt{2}}{2} & 0 & \frac{2-\sqrt{2}}{2} \end{bmatrix} \quad \text{and} \quad \frac{\partial I}{\partial y} = 0.5 \begin{bmatrix} -\frac{2-\sqrt{2}}{2} & -(\sqrt{2}-1) & -\frac{2-\sqrt{2}}{2} \\ 0 & 0 & 0 \\ \frac{2-\sqrt{2}}{2} & \sqrt{2}-1 & \frac{2-\sqrt{2}}{2} \end{bmatrix}.$$

3. **Setting Canny thresholds:** The low and high thresholds used by the Canny method are represented as percentiles of the gradient norm.

4. **Non-maximum suppression:** We look for local maxima of the gradient norm in the gradient direction. If the gradient norm value of a local maximum is greater than the high threshold, it is classified as an edge, otherwise, the local maximum will be marked for the subsequent hysteresis process.

5. **Hysteresis implementation:** All the pixels where the value of the gradient norm is between the thresholds will be considered as edges if they are connected to edge pixels. After that, their neighborhood is recursively explored. Finally, the edges are stored in a structure with their orientations.

Finally, to simplify the edge points collection and to eliminate potential outliers, we optionally apply a cleaning procedure to the edge point collection. Since we are interested in extracting lines from these edge points, we remove points where the edge orientation varies in a significant way in a neighborhood of the point. For each point p we use as measure of edge orientation stability, $EOSM(p)$, the following expression

$$\begin{aligned} EOSM(p) &= \sum_{q \in Edge\ points\ \cap\ Neighborhood(p)} cos(\alpha_p)cos(\alpha_q) + sin(\alpha_p)sin(\alpha_q) \\ &= \sum_{q \in Edge\ points\ \cap\ Neighborhood(p)} cos(\alpha_p - \alpha_q), \end{aligned}$$

where α_p, α_q are the edge point orientations. We point out that the lower $EOSM(p)$, the more variation of the edge orientation exists in the neighborhood of p. In particular, an image corner will have a smaller value of $EOSM(p)$ than a straight edge point. Then, we remove points that do not belong to edge line segments by only keeping edge points with a minimum value of $EOSM(p)$ (given by a threshold). Moreover, to speed up the algorithm we fix a minimum distance between edge points and we remove any edge point q such that $EOSM(q) < EOSM(p)$ and q belongs to a square neighborhood of p with a radius given by such minimum distance. This procedure depends on the order we go through the edge points but, in practice, such order has not a significant influence in the algorithm outcome. Algorithm 3 describes in more detail the steps of this edge cleaning procedure using the parameters described in Table 4. We point out that, using the cleaning procedure, we can significantly reduce the number of edge points and, therefore, the computational cost of the algorithm is also reduced.

A practical example of the cleaning process is presented in Figure 3. From the original image (Figure 3(a)), we obtain the edges by using the Canny method (Figure 3(b)) with the parameters described in Table 3. Afterward, the points p with high $EOSM(p)$ value are removed (Figure 3(c)) according to the parameters shown on Table 4 and, finally, only the points with lowest $EOSM(p)$ value within their neighborhood are kept (Figure 3(d)), setting the minimum distance between contour points to one.

2.2 Stage 2: Initial Estimation of the One-Parameter Lens Distortion Model using the Improved Hough Transform

Once the edges have been detected and cleaned, the next step consists in detecting the straight lines present in the image. To this aim, we use the technique proposed by Alemán-Flores et al. [2, 1].

In both articles, the authors propose a modified Hough transform in order to detect straight lines in the input image. These lines are distorted by the effect of the lens distortion in the projection of 3D straight lines onto a 2D image. Tackling the detection of straight lines with the traditional Hough transform provides a wrong result, because the distorted lines are split and detected as several straight lines.

The approach proposed by the authors is based on including the distortion as a new dimension in the Hough space. Therefore, a straight line in this space is represented by three parameters: distance to the origin, orientation and distortion. This new representation transforms the two-dimensional

Algorithm 2: Edge detection using Canny method, described in Figure 2

input : input ; `// gray-scale input image.`
per_low, per_high ; `// Percentage for the LOW and HIGH threshold in [0,1].`
output: output ; `// image with edges.`
sine, cosine ; `// Orientation of the edges.`
edges ; `// (x,y) coordinate of the edges.`

gauss_conv(input, blurred_image) ; `// Gaussian convolution`

grad(blurred_image, x_grad, y_grad) ; `// Compute the gradient`

foreach $p \in$ *Image domain* **do**
  gradient_norm(p) $\leftarrow \sqrt{x\_grad(p)^2 + y\_grad(p)^2}$;
end

`// Compute the thresholds using the STL::nth_element function`
low_threshold $\leftarrow$ nth_element(gradient_norm, per_low $\cdot$ Number_pixels);
high_threshold $\leftarrow$ nth_element(gradient_norm, per_high $\cdot$ Number_pixels);

`// Non-maximum suppression`
foreach $p \in$ *Image domain* **do**
  if *p is a local maximum of the gradient norm in the direction of maximum variation of the gradient norm* **then**
    if *gradient_norm(p)* $\geq$ *high_threshold* **then**
      p is classified as EDGE
    else
      p, a gradient local maximum, is classified as a potential edge point in the hysteresis process
    end
  end
end

`// Hysteresis process of the Canny method`
foreach $p \in$ *Image domain* **do**
  if *p is classified as EDGE* **then**
    `// Explore recursively p neighborhood in order to consider contiguous points. Any contiguous point which is a gradient local maximum and whose gradient norm is bigger than low_threshold is classified as EDGE`
  end
end

`// Fill the output images taking into account the EDGE classification`
foreach $p \in$ *Image domain* **do**
  if *p is classified as EDGE* **then**
    output(p) $\leftarrow$ 255;
    cosine(p) $\leftarrow \frac{x\_grad(p)}{gradient\_norm(p)}$; `// Orientation of the EDGE`
    sine(p) $\leftarrow \frac{y\_grad(p)}{gradient\_norm(p)}$;
    edges.add(p) ; `// Position of EDGE`
  else
    output(p) $\leftarrow$ 0;
  end
end

Algorithm 3: Cleaning process.

input : p, α_p ; `// edge point coordinates and orientations.`
neighborhood_radius ; `// radius of neighborhood to take into account.`
min_neighbor_points ; `// minimum number of contour points in a neighborhood.`
min_orientation_value ; `// min average scalar product of neighborhood point orientation`
min_distance_point ; `// minimum distance between contour points.`
output: `// edge points cleaned.`

foreach $p \in$ *Edge points* **do**
  `// We compute a measure of edge orientation stability in a square neighborhood of point` p.
  EOSM(p) $\leftarrow$ 0, number_neighborhood_points(p) $\leftarrow$ 0;
  for $q \in$ *neighborhood*(p) *AND* $q \in$ *Edge points* **do**
    EOSM(p) $\leftarrow$ EOSM(p) + $cos(\alpha_p) \cdot cos(\alpha_q) + sin(\alpha_p) \cdot sin(\alpha_q)$;
    number_neighborhood_points(p) $\leftarrow$ number_neighborhood_points(p) + 1;
  end
end

`// Remove points from edge collection according to the minimum neighborhood points and stability of point orientation (corners are removed)`
foreach $p \in$ *Edge points* **do**
  if $number\_neighborhood\_points(p) < min\_neighbor\_points$ *OR* $EOSM(p) < number\_neighborhood\_points(p) \cdot min\_orientation\_value$ **then**
    `remove_point` (p);
  end
end

`// Remove isolated points using an iterative procedure. To speed up the procedure we allow only 4 iterations.`
while *the number of edge points removed is bigger than 0* **do**
  `// We compute the number of points in the neighborhood`
  `// We remove isolated contour points`
end

if $min\_distance\_point > 0$ **then**
  foreach $p \in$ *Edge points* **do**
    max $\leftarrow$ EOSM(p);
    for $q \in$ *neighborhood*(p) *AND* $q \in$ *Edge points* **do**
      if $EOSM(q) > max$ **then** max $\leftarrow EOSM(q)$ **else** `remove_point` (q) ;
    end
  end
end

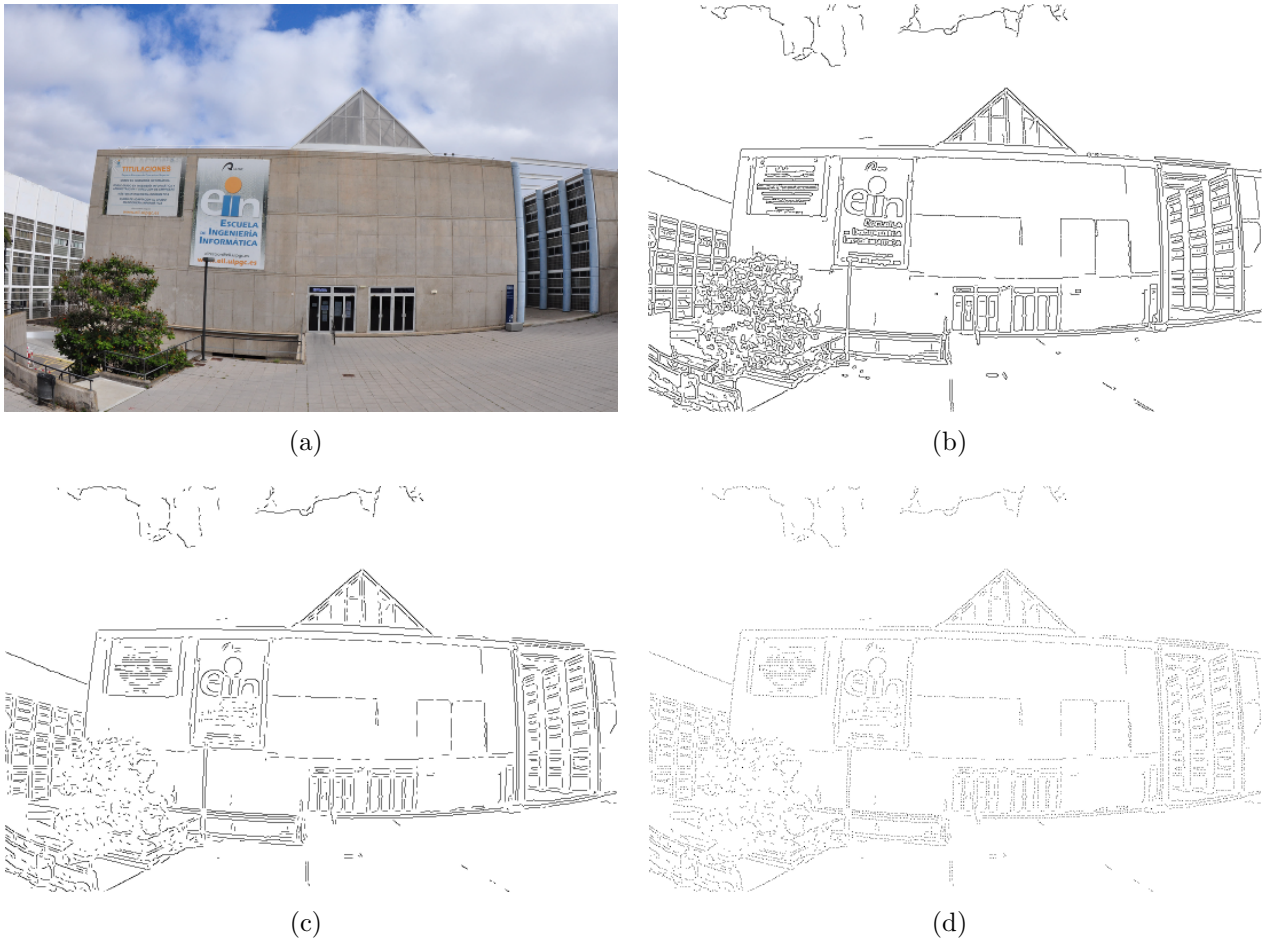


Figure 3: Example of the cleaning process after edge detection using the Canny method: (a) input image, (b) result of the edge detection, (c) edges after cleaning points with a high curvature and (d) after cleaning points in a neighborhood of 3×3 pixels.

space of the traditional Hough transform into a three-dimensional one. The new parameter is normalized in order to represent the percentage of variation of the distance between the distortion center and the furthest corner of the image. A range of values for this normalized parameter is used and, for every discretized value within this range, the position and orientation of the edge points are corrected. Each corrected point will vote for the lines which are closer and whose orientation is similar. Moreover, the votes are weighted by the distance between the edge point and the associated line. The closer the point, the higher the vote.

In order to select the value for the distortion parameter, we search for the value which best corrects the lines, i.e. which provides the longest lines after correction. To compare the possible values, we consider the n longest lines for each value and we add the scores of these lines (as explained above, this score depends on the number of points associated to that line and the distance from points to the line). This is a measure of the reliability of each value of the distortion parameter which favors the values for which the longest lines have been detected. The reliability measure v_i for i^{th} value of the first distortion parameter is

$$v_i = \sum_{j=1}^{n} s_j^i, \tag{4}$$

where s_j^i is the total score of the j^{th} line of the i^{th} value. Such score is computed according to the

collection of edge points which votes to the j^{th} line in the Hough space. Roughly speaking, if $line_j$ is the line equation associated to the j^{th} line, then s_j^i is given by :

$$s_j^i = \sum_{p \in Edge \ : \ distance(line_j, C_i(p)) < 1} \frac{1}{1 + distance(line_j, C_i(p))},$$

where $C_i(p)$ is the transformation of point p given by the lens distortion model associated to the i^{th} value of the distortion parameter. In Equation (4), for each i^{th} value, the n lines in the Hough space with highest values of s_j^i are used to compute v_i.

In Figure 4 we show the flowchart of this stage and we describe the method in Algorithm 4. We remark that, due to the size of this part of the method, we have divided the procedure in several subprograms (see Appendix A).

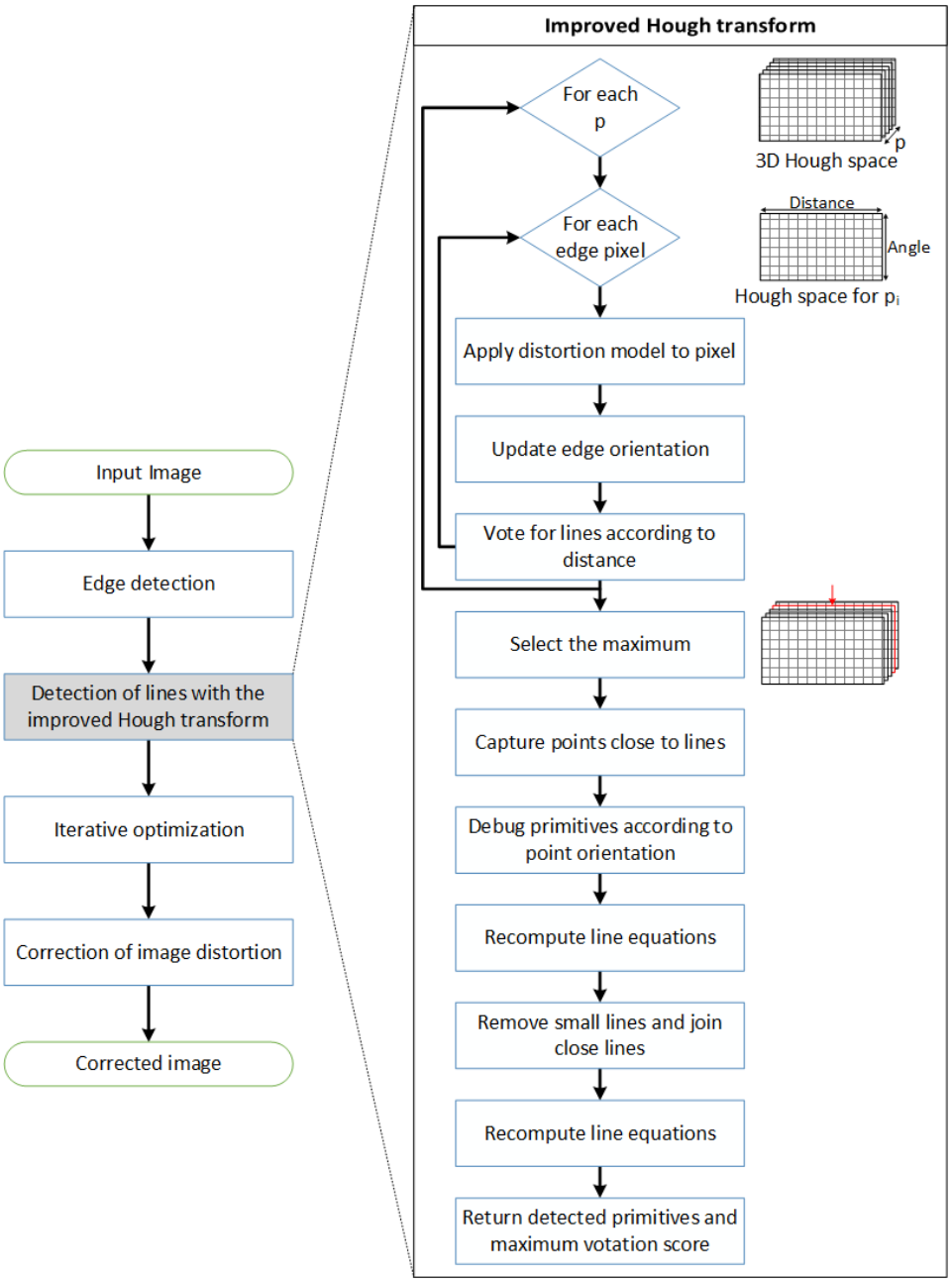


Figure 4: Initial estimation of lens distortion model using the improved Hough transform.

Algorithm 4: Stage 2. Detecting lines with improved Hough transform, described in Figure 4

input : contours ; // Contour information.
ldm ; // Initial distortion model.
output: i_primitive ; // Image primitives where lines and distortion model is defined.
$v_{i_{max}}$; // Maximum of voting score.

// Define score volume size and orientation discretization vector

// Update the contours object with the initial lens distortion model, Algorithm 10
update_contours($\cdots$);

// Fill the 3D score volume
for $i \leftarrow 0$ **to** *depth_score* **do**
  // Compute the lens distortion parameter value for each iteration, Algorithms 11, 12
  voting($\cdots$);
  // Computation of reliable measure v_i given in (4)
  // Select the distortion level i_{max} which maximizes v_i
end

// Image_primitives objects for corrected points and original points
i_primitive_corrected.lines ← i_primitive.lines;
i_primitive_original.lines ← i_primitive.lines;

// Fill the image primitives using the lens distortion model given by i_{max}

// Compute the points of the lines following their distance, Algorithm 13
capturing_points($\cdots$);

// Debugging of the primitives through the orientation of the points, Algorithm 15
ensure_consistent_line_orientations($\cdots$);

// Recompute the line equations, Algorithm 19
recompute_line_equations($\cdots$);

// Remove lines with a small number of points and merge lines which are too close, Algorithms 17, 18
remove_and_join($\cdots$);

// After removing the lines, we recompute the equations, Algorithm 19
recompute_line_equations($\cdots$);

// Update the image primitives and return the maximum voting score $v_{i_{max}}$
i_primitive ← i_primitive_original;

return $v_{i_{max}}$;

2.3 Stage 3: Iterative optimization

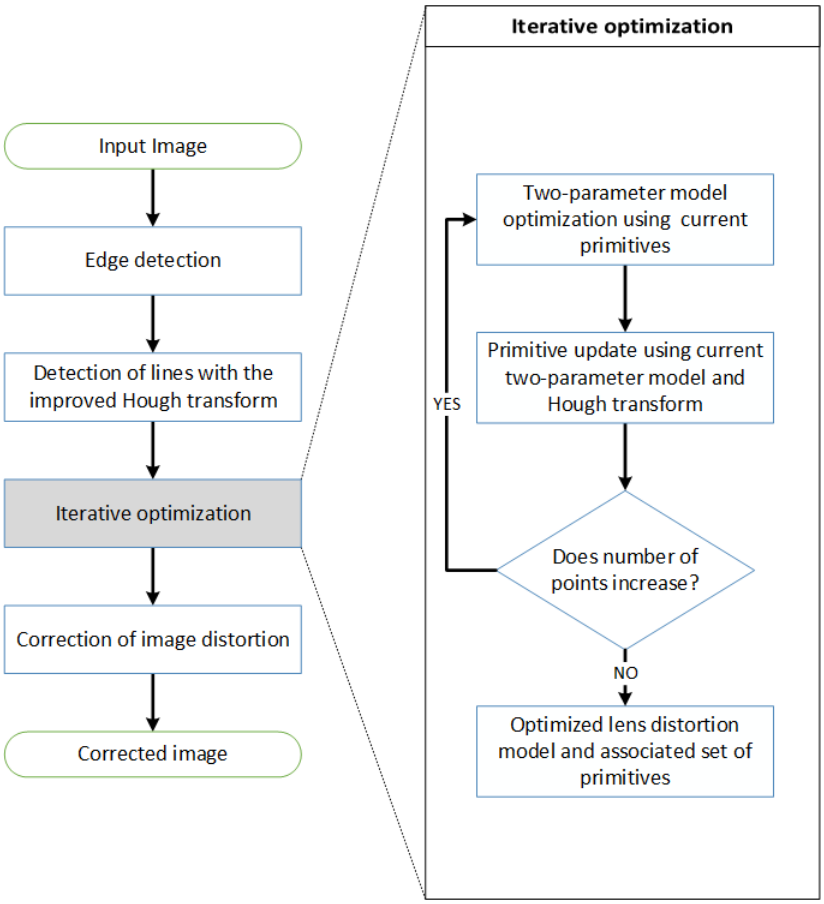


Figure 5: Iterative optimization.

Once the one-parameter model and the set of initial primitives have been obtained, two tasks are iteratively carried out in this stage. Firstly, we optimize the lens distortion model including the second distortion parameter and the distortion center. Secondly, with the optimized two-parameter lens distortion model, we apply the Hough transform with the aim of detecting points that were not considered before.

The optimization of the lens distortion model is performed by minimizing the distance from the corrected primitive points to the lines. If we denote by $\mathbf{d}$ the tuple (k_1, k_2, x_c, y_c), which defines the distortion model by its parameters and distortion center, this minimization consists in reducing the energy

$$E(\mathbf{d}) = \sum_{j}^{Nl} \sum_{i}^{N(j)} distance\left(C_d(\overline{x}_{ji}), line_j\right)^2, \tag{5}$$

where Nl is the number of lines, $N(j)$ is the number of points of the j^{th} line, and $C_d(\overline{x}_{ji})$ represents the corrected points associated to $line_j$, using the model given by $\mathbf{d}$. The initial value for k_1 is given by the final value obtained in the previous step, whereas the initial value for k_2 is 0. The center of distortion is initialized at the geometric center of the image. The Taylor expansion of $E(\mathbf{d})$ around the initial approximation $\mathbf{d_0}$ is given by

$$E(\mathbf{d}) = E(\mathbf{d_0}) + (\nabla E(\mathbf{d_0}))^T(\mathbf{d} - \mathbf{d_0}) + \frac{1}{2}(\mathbf{d} - \mathbf{d_0})^T \nabla^2 E(\mathbf{d_0})(\mathbf{d} - \mathbf{d_0}) + \dots \tag{6}$$

Algorithm 5: Stage 3. Iterative optimization, described in Figure 5

input : contours ; `// The edges of the input image.`
i_primitives ; `// The previous collection of lines.`
opt_center ; `// Option for the center optimization.`
output: i_primitives ; `// The collection of lines optimized.`

best_model ← previous_model ← i_primitives.ldm;
previous_ip ← i_primitives;
TOL ← 10^{-2};
fails ← 0;
next_num_points ← best_num_points ← num_points ← `count_points` (i_primitives);

while $next\_num\_points \geq (num\_points \cdot (1 + TOL))$ OR $fails < 3$ **do**
  error ← energy_minimization(previous_model, i_primitives, opt_center) ; `// Algorithm 7`
  `// Call Hough with the model computed before`
  i_primitives ← Hough(contours, previous_model) ; `// Algorithm 4`
  local_num_points ← `count_points` (i_primitives);
  if *local_num_points* > *next_num_points* **then**
    `// We update the primitives only if Hough provides a better result`
    if *local_num_points* > *best_num_points* **then**
      previous_ip ← i_primitives;
      best_num_points ← local_num_points;
      best_model ← previous_model;
      final_error ← error;
    end
  else
    fails ← fails+1;
  end
  num_points ← next_num_points;
  next_num_points ← local_num_points;
end
`// We take the last and best image primitives object and model`
i_primitives ← previous_ip;
i_primitives.distortion ← best_model;
`// We return the average error`
return $\frac{final\_error}{best\_num\_points}$;

Since we want to minimize this energy, we derive the previous expression and make it equal to 0

$$\nabla E(\mathbf{d}) \approx \nabla E(\mathbf{d_0}) + \nabla^2 E(\mathbf{d_0})(\mathbf{d} - \mathbf{d_0}) = 0. \tag{7}$$

Therefore,

$$\nabla^2 E(\mathbf{d_0})(\mathbf{d} - \mathbf{d_0}) = -\nabla E(\mathbf{d_0}), \tag{8}$$

where

$$\nabla E(\mathbf{d_0}) = \begin{pmatrix} \frac{\partial E(\mathbf{d_0})}{\partial k_1} \\ \frac{\partial E(\mathbf{d_0})}{\partial k_2} \\ \frac{\partial E(\mathbf{d_0})}{\partial x_c} \\ \frac{\partial E(\mathbf{d_0})}{\partial y_c} \end{pmatrix}, \tag{9}$$

$$\nabla^2 E(\mathbf{d_0}) = \begin{pmatrix} \frac{\partial^2 E(\mathbf{d_0})}{\partial k_1^2} & \frac{\partial^2 E(\mathbf{d_0})}{\partial k_1 \partial k_2} & \frac{\partial^2 E(\mathbf{d_0})}{\partial k_1 \partial x_c} & \frac{\partial^2 E(\mathbf{d_0})}{\partial k_1 \partial y_c} \\ \frac{\partial^2 E(\mathbf{d_0})}{\partial k_2 \partial k_1} & \frac{\partial^2 E(\mathbf{d_0})}{\partial k_2^2} & \frac{\partial^2 E(\mathbf{d_0})}{\partial k_2 \partial x_c} & \frac{\partial^2 E(\mathbf{d_0})}{\partial k_2 \partial y_c} \\ \frac{\partial^2 E(\mathbf{d_0})}{\partial x_c \partial k_1} & \frac{\partial^2 E(\mathbf{d_0})}{\partial x_c \partial k_2} & \frac{\partial^2 E(\mathbf{d_0})}{\partial x_c^2} & \frac{\partial^2 E(\mathbf{d_0})}{\partial x_c \partial y_c} \\ \frac{\partial^2 E(\mathbf{d_0})}{\partial y_c \partial k_1} & \frac{\partial^2 E(\mathbf{d_0})}{\partial y_c \partial k_2} & \frac{\partial^2 E(\mathbf{d_0})}{\partial y_c \partial x_c} & \frac{\partial^2 E(\mathbf{d_0})}{\partial y_c^2} \end{pmatrix}. \tag{10}$$

We optimize the values of the tuple $\mathbf{d}$ using the following iterative scheme

$$\mathbf{d_{n+1}} = \mathbf{d_n} + (\nabla^2 E(\mathbf{d_n}) + \gamma Id)^{-1}(-\nabla E(\mathbf{d_n})), \tag{11}$$

where γ is used to control the convergence of the minimization as follows: γ is updated in each iteration to ensure that $E(\mathbf{d_{n+1}}) < E(\mathbf{d_n})$. Usually, its value is higher when we are far from the solution and decreases when we approach it. We observe that we look for local minima of the energy $E(\mathbf{d})$ because, in general, the global minima of this energy can be attained at singular configurations (especially when a very small number of primitive points are used). To avoid such singular solutions, we check, using the results of theorems 1 and 2 showed in the companion paper [21], if the lens distortion transformation is one-to-one and, otherwise, we reject the proposed solution (the details of the algorithm for checking the invertibility of the lens distortion models are described in Algorithm 6).

In order to normalize the parameters k_1 and k_2 so that the method is independent of the image resolution, and avoid working with too small values, we use the parameter normalization introduced in the companion paper [21]. Therefore, the minimization is performed in the normalized parameters p_1 and p_2 instead of k_1 and k_2. The value of p_1 represents the percentage of correction of the furthest point in the image from the center of distortion, and p_2 represents the same percentage of correction, but for the midpoint between the center of distortion and the furthest point. This way, the parameters are easier to interpret and do not depend on the image resolution. In what follows, we will denote as r_1 the distance from the center of distortion to the furthest point in the image domain and r_2 half of this distance. In the case of the polynomial model, we obtain the relation for p_1 and p_2 showed in Equation (12), as well as the correspondence for the distortion parameters k_1 and k_2 in (13).

$$p_1 = k_1 4r_2^2 + k_2 16r_2^4, \qquad p_2 = k_1 r_2^2 + k_2 r_2^4. \tag{12}$$

$$k_1 = \frac{p_1 - 16p_2}{-12r_2^2}, \qquad k_2 = \frac{4p_2 - p_1}{-12r_2^4}. \tag{13}$$

When we deal with the division model, we obtain a similar relation between the distortion parameters and the normalized ones. In (14), we show the correspondence obtained for p_1 and p_2, whereas in (15) we do the same related to k_1 and k_2.

$$p_1 = \frac{1}{1 + k_1 r_1^2 + k_2 r_1^4} - 1, \qquad p_2 = \frac{1}{1 + k_1 r_2^2 + k_2 r_2^4} - 1. \tag{14}$$

$$k_1 = \frac{\frac{-p_1}{1+p_1} + \frac{16p_2}{1+p_2}}{-12r_2^2}, \qquad k_2 = \frac{\frac{-4p_2}{1+p_2} + \frac{p_1}{1+p_1}}{-12r_2^4}. \tag{15}$$

In Algorithm 7 we describe the main ideas of this minimization process.

During this minimization procedure, we consider the same points which were originally detected as line points. Only the parameters vary in order to obtain a better fitting between the points and the line equations. However, once the center of distortion and the two distortion parameters have been optimized, new points could be detected by the Hough transform using the optimized lens distortion model to correct the distortion. Some edge points which were originally rejected because they did not match the line equations (especially near the image boundaries, where the effect of the distortion is more noticeable) can now be added to the set of line points. Therefore, we correct the distortion of the edge points and apply the Hough transform once again, trying to detect a larger collection of line points. The introduction of the iterative scheme to recalculate the line points using the optimized parameters allows detecting new lines and significantly increases the number of points of the detected lines, especially when the distortion is strong. We point out that this iterative procedure always converges because we stop the iterations when the global amount of points associated to the distorted lines does not increase across the iterations. In practice, only a few iterations are required to attain convergence. In Figure 5 we show the flowchart of the iterative optimization process, and in Algorithm 5 the method is described. In order to simplify the reading, some of the functions used in the algorithm are included in Appendix B.

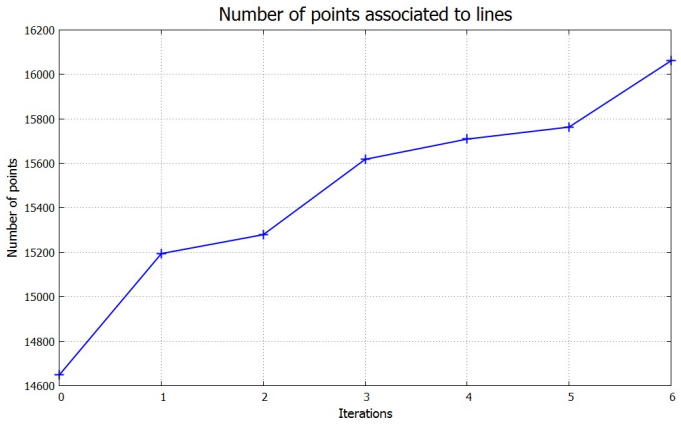


Figure 6: Variation of the number of points throughout the iterative optimization process applied to the image in Figure 10.

Figure 6 shows the variation of the number of detected points throughout the iterations. As observed, even from the first iteration, the number of points increases. This is due to the use of the lens distortion model obtained in the previous iteration for correcting the edge points. As the model has been optimized, by means of reducing the distance from the corrected points to the associated straight lines, the improved Hough transform obtains a greater number of points.

2.4 Stage 4: Image Distortion Correction

The correction of the distortion is carried out using the inverse of the lens distortion model. As explained in the companion paper [21], we have to invert the radial function:

$$\hat{r}(r, k_1, k_2) = r \cdot L(r),$$

Algorithm 6: Checking the invertibility of the lens distortion model

input : ldm ; `// Lens distortion model to check.`
(width, height) ; `// Image size.`
output: ; `// True if the lens distortion model is invertible or false otherwise`

`// We obtain the distance from the distortion center to the furthest corner of the image`
r $\leftarrow$ `get_radius`(`get_center`(ldm), $width$, $height$);
$hr \leftarrow \frac{r}{2}$;
if `get_type`(ldm) $= POLYNOMIAL$ **then**
  $k_1 \leftarrow \frac{p_1 - 16p_2}{-12hr^2}$;
  $k_2 \leftarrow \frac{4p_2 - p_1}{-12hr^4}$;
  if
  $((r^2k_1 < \frac{-2}{3})\ AND\ (9r^4k_1^2 - 20r^4k_2 < 0))\ OR\ ((r^2k_1 \geq \frac{-2}{3})\ AND\ (5r^4k_2 + 3r^2k_1 + 1 > 0))$
  then return *true* ;
else
  $k_1 \leftarrow \frac{\frac{-p_1}{1+p_1} + \frac{16p_2}{1+p_2}}{-12hr^2}$;
  $k_2 \leftarrow \frac{\frac{-4p_2}{1+p_2} + \frac{p_1}{1+p_1}}{-12hr^4}$;
  if $(-2 < r^2k_1 < 2)\ AND\ (-1 - r^2k_1 < r^4k_2 < \frac{1-r^2k_1}{3})$ **then return** *true* ;
  if $(r^2k_1 \geq 2)\ AND\ (-1 - r^2k_1 < r^4k_2 < -\frac{r^4k_1^2}{12})$ **then return** *true* ;
end
return *false*;

where r is the distance from an image point (x, y) to the distortion center, which we solve by a Newton algorithm. To speed up the procedure, instead of computing the inverse of $\hat{r}(r, k_1, k_2)$ for each point (x, y) independently, we first calculate the inverse in the discrete set $r = 1, 2, 3, \ldots, \hat{r}(r_1, k_1, k_2)$ (where r_1 is the maximum distance from an image point to the distortion center) and then, for any point (x, y), we compute the inverse of $\hat{r}(r, k_1, k_2)$ using an interpolation procedure. Figure 7 shows the flowchart of the distortion correction process, whereas Algorithm 8 describes the method. More details about the correction of the distortion are given in Appendix C.

Algorithm 7: The optimization of p_1, p_2, x_c, y_c by minimizing $E(p_1, p_2, x_c, y_c)$

input : lines ; `// Primitives detected in the image`
ldm ; `// Lens distortion model given by the parameters` p_1, p_2, x_c, y_c.
output: ldm ; `// The optimized lens distortion model.`

$h_1 \leftarrow 10^{-4}$; $h_2 \leftarrow 10^{-2}$;
$\gamma \leftarrow 10.0$;
$Id(4,4) \leftarrow$ Identity matrix;

convergence_it $\leftarrow 0$;
TOL $\leftarrow 10^{-4}$;

`// We compute` v_{res} `values using the relation` $r_1 = 2 \cdot r_2$
$v_{res}(0) \leftarrow TOL(|p_1| + 2)$;
$v_{res}(1) \leftarrow TOL(|p_2| + 2)$;
$v_{res}(2) \leftarrow TOL(|x_c| + 2)$;
$v_{res}(3) \leftarrow TOL(|y_c| + 2)$;

while $(|v_{res}(0)| > TOL(|p_1| + 1)\ OR\ |v_{res}(1)| > TOL(|p_2| + 1)$
$OR\ |v_{res}(2)| > TOL(|x_c| + 1)\ OR\ |v_{res}(3)| > TOL(|y_c| + 1))$
$AND\ convergence\_it \leq 100$ **do**

`// We compute the gradient vector and the Hessian matrix`
$grad_E \leftarrow$ Compute_gradient$(p_1, p_2, x_c, y_c, h_1, h_2)$; `// Algorithm 20`
$hess_E \leftarrow$ Compute_hessian$(p_1, p_2, x_c, y_c, h_1, h_2)$; `// Algorithm 21`

`// We solve the equations system by means of Gauss method`
$v_{res}(4) \leftarrow$ Gauss$(grad_E, hess_E + \gamma \cdot Id)$;
$p_{1_{new}} \leftarrow p_1 + v_{res}(0)$;
$p_{2_{new}} \leftarrow p_2 + v_{res}(1)$;
$x_{c_{new}} \leftarrow x_c + v_{res}(2)$;
$y_{c_{new}} \leftarrow y_c + v_{res}(3)$;

`// We iterate until the energy is reduced and the model is invertible`
ldm $\leftarrow$ `lens_distortion_model` $(p_{1_{new}}, p_{2_{new}}, x_{c_{new}}, y_{c_{new}})$;
if $E(p_{1_{new}}, p_{2_{new}}, x_{c_{new}}, y_{c_{new}}) < E(p_1, p_2, x_c, y_c)\ AND$ `check_invertibility`(ldm) **then**
$p_1 \leftarrow p_{1_{new}} + v_{res}(0)$;
$p_2 \leftarrow p_{2_{new}} + v_{res}(1)$;
$x_c \leftarrow x_{c_{new}} + v_{res}(2)$;
$y_c \leftarrow y_{c_{new}} + v_{res}(3)$;
$\gamma \leftarrow \frac{\gamma}{10}$;
else
$\gamma \leftarrow \gamma \cdot 10$;
end
convergence_it $\leftarrow$ convergence_it + 1;
end

`// We update the final model`
ldm $\leftarrow$ `lens_distortion_model` $(p_{1_{new}}, p_{2_{new}}, x_{c_{new}}, y_{c_{new}})$;
return $min(E(p_{1_{new}}, p_{2_{new}}, x_{c_{new}}, y_{c_{new}}), E(p_1, p_2, x_c, y_c))$;

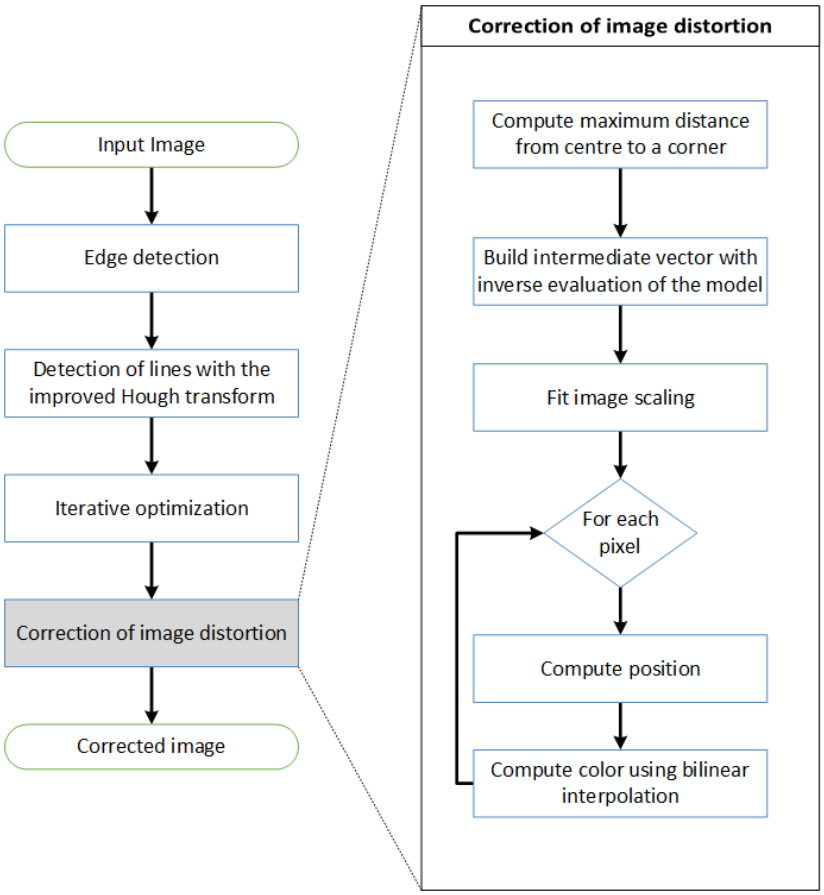


Figure 7: Image distortion correction.

Algorithm 8: Distortion-free image computation, described in Figure 7

input : ; // Input image.
; // The estimated lens distortion model.
output: ; // A distortion free output image.

// We compute the $InverseVector$ in $1, 2, \ldots, \hat{r}(r_1, k_1, k_2)$ (as explained in the text).

for $(x', y') \in$ *Output image pixels* **do**

$r' \leftarrow \sqrt{(x' - x_c)^2 + (y' - y_c)^2}$;
$index \leftarrow Floor(r')$;
$weight \leftarrow r' - index$;
$r \leftarrow (1 - weight) \cdot InverseVector(index) + weight \cdot InverseVector(index + 1)$;
$x \leftarrow x_c + r\,(x' - x_c)$;
$y \leftarrow y_c + r\,(y' - y_c)$;

// We use bilinear interpolation to estimate $InputImage(x, y)$
$OutputImage(x', y') \leftarrow InputImage(x, y)$;

end

3 Complexity Analysis

In this section we present the complexity analysis of the described algorithms. First, we start by defining some variables used in the description:

- N_{pixels}: Number of image pixels.
- N_{edges}: Number of edge pixels.
- N_{lines}: Number of lines to extract from the Hough space.
- $S_{distortion}$: Size of the discretized interval of distortion parameters used in the Hough space.
- S_{Hough}^{matrix}: Size of the Hough voting matrix.
- S_{Hough}^{voting}: Size of the neighborhood used for each point to vote in the Hough score matrix (this neighborhood includes the lines passing near the edge point and with an orientation similar to the edge point orientation)

Canny edge detector: In the Canny edge detection algorithm, the complexity is determined by the algorithm we use to implement the Gaussian convolution. As we use the fast recursive algorithm proposed in [6] as an approximation of the Gaussian convolution, which has a linear complexity with respect to the image size, we can conclude that the complexity of the Canny edge detector we use is

$$O(N_{pixels}).$$

Improved Hough transform: The traditional Hough transform has a complexity related to the number of provided edge points and the dimension of the voting matrix. In our case, we divide the complexity analysis in the following stages:

1. We consider the complexity of the voting process. The computational cost of this stage is based on the number of detected edge points provided to our improved Hough Transform by the Canny edge detector. Moreover, we have to take into account the dimension of the voting matrix, determined by the size of the interval for the discretized lens distortion parameter and the size of the intervals for the discretized distance and angle variation for each edge point to vote. In this way, we can write the complexity of this section as

$$O(N_{edges} \times S_{distortion} \times S_{Hough}^{voting}).$$

2. As well as in the above description, in the selection of the maximum, we consider the size of the Hough voting matrix. However, as we provide a maximum number of lines to be considered, we need to include this number in the complexity estimation. Therefore, the cost of the selection of the maximum will be

$$O(N_{lines} \times S_{Hough}^{matrix}).$$

3. The complexity of the association of the edge points with the lines depends on the number of edge points and the number of lines to compute

$$O(N_{lines} \times N_{edges}).$$

Distortion parameter optimization: The modified Newton-Raphson method proposed for the optimization of the distortion parameter converges in a few iterations for all the experiments performed. Therefore, time complexity comes from the evaluation of the function to optimize (5). In our case, such complexity is given by

$$O(N_{edges}).$$

Image distortion correction: For the correction of the distortion, we have to go through the image pixels and correct the lens distortion, so the complexity is

$$O(N_{pixels}).$$

Iterative optimization: The complexity of the iterative optimization process is similar to the first point of the improved Hough transform, described above. However, in this stage the method is applied without a range for the normalized distortion parameter. We can rewrite the complexity as

$$O(N_{edges} \times S_{Hough}^{voting}),$$

where this complexity is multiplied by the number of iterations. In practice, only a few iterations are needed to reach the optimized model. Therefore, the complexity of the method is given by

$$O(N_{pixels}) + O(N_{pixels}) + O(N_{edges} \times S_{Hough}^{voting}).$$

Even though providing a precise estimation for the relation between the different variables is difficult because it depends on the image, in general we have that $N_{lines} << N_{edges} << N_{pixels} \approx S_{Hough}^{matrix}$. Hence, considering the above expressions, the total complexity will be around $O(N_{pixels})$.

Therefore, the time complexity is expected to be approximately linear with respect to the number of pixels. In order to illustrate this, in Figure 8, we show the time required for 8 images of different sizes (red crosses), which correspond to subimages of the image of the calibration pattern. The size of the k^{th} image (for $k = 1, \ldots, 8$) is $\frac{k}{8}$ the size of the original image. Using this collection of images and their corresponding CPU times, we compute the linear regression (green line). We obtain a correlation coefficient of 0.9884, which shows a computational complexity close to $O(N_{pixels})$.

4 Experimental Results

In this section we show some experiments carried out on an image of a synthetic calibration pattern and an image of a building. Figures 9 and 10 illustrate the results of applying the proposed technique to detect the primitives. We remark that all the experiments have been carried out using the same set of parameters, described in tables 3, 4 and 5.

Figure 9 shows the different steps of the method applied to an image of a calibration pattern. From the input image (Figure 9(a)), we detect the edges using the Canny method and the cleaning process, described in Section 2.1 (see Figure 9(b)). After this step, the iterative optimization process is applied, providing the primitives present in the image (Figure 9(c)). As observed, all the lines of the calibration pattern are properly detected. Furthermore, some lines which do not belong to the pattern are also detected (see lines on the bottom left corner). On the other hand, even when the lines are not straight due to the distortion, they are detected as single lines (see the brown line on the bottom or the green on the top of Figure 9(c)). Finally, the image is corrected using the information obtained with the iterative procedure (Figure 9(d)).

Figure 10 illustrates the result of applying the method to an image of a building. In Figure 10(b), we show the detected edges, and in Figure 10(c) the lines obtained with the iterative optimization

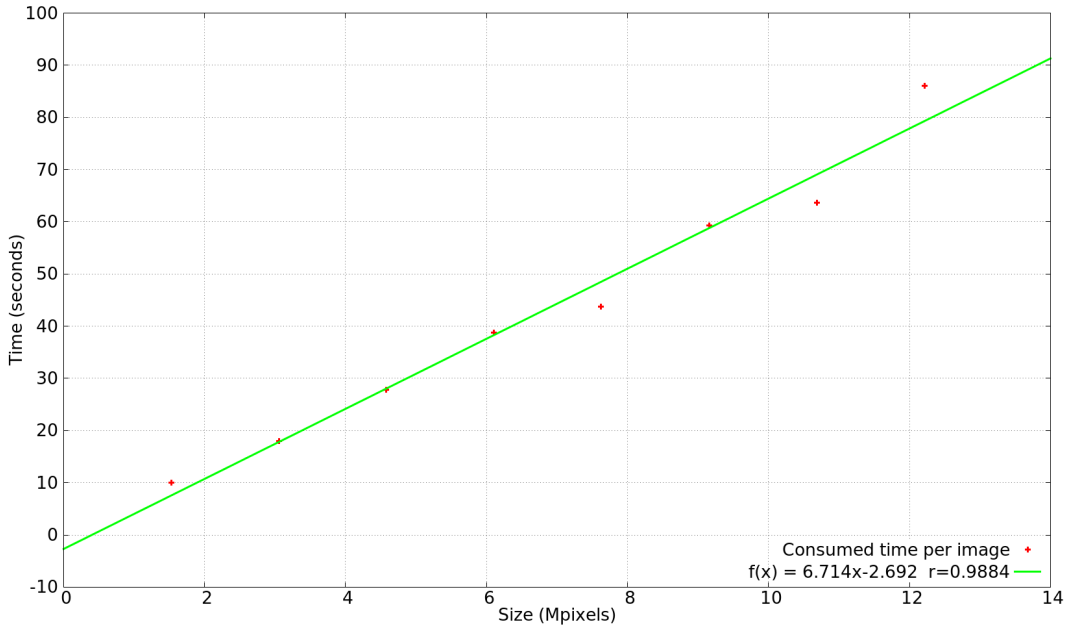


Figure 8: CPU time for images of different sizes obtained from scaled versions of the calibration pattern (red crosses). Linear regression (green line).

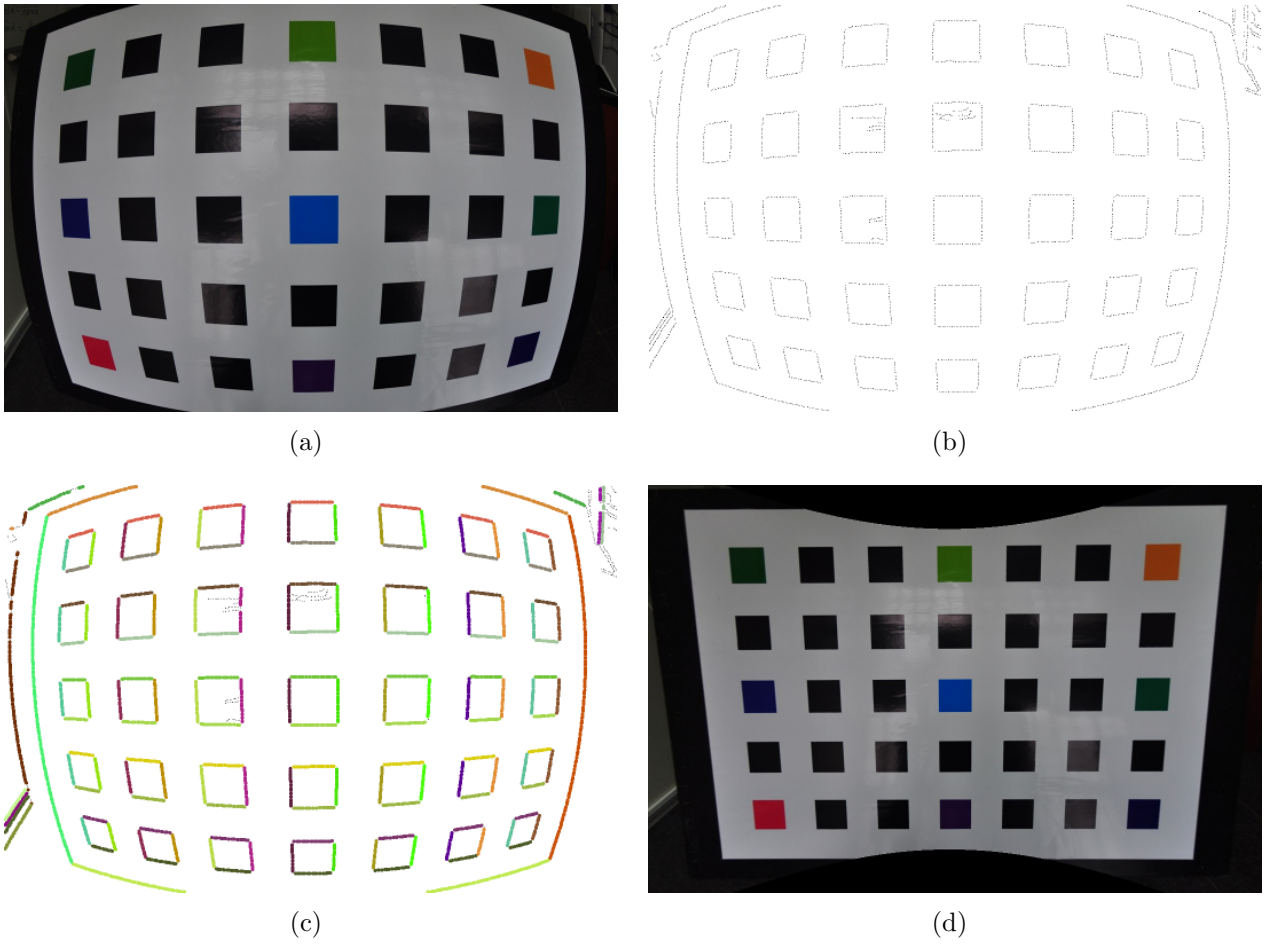
process. Even when the lines are affected by a significant distortion, they are detected (see the lines on the stairs at the bottom of the image). Moreover, the method is able to distinguish lines that are very close (e.g. the lines of the stairs at the bottom of the image). As we show in Figure 10(d), the distortion is almost perfectly corrected.

In order to show a wide variety of experiments carried out with real images under different conditions of illumination and perspective, we include Figure 11. In this figure we show the results of applying the iterative optimization method to a database of distorted images.

Tables 1 and 2 show some comparative results between polynomial and division models, using one-parameter models and two-parameter models with iterative optimization. Although the number of detected lines decreases when we use the iterative optimization instead of the one-parameter models, the number of points increases significantly. This means that the new proposal provides longer lines, which gives more information for the estimation of the distortion. On the other hand, it is also remarkable that the value of E (the error obtained measuring the distance from the corrected points to the corrected straight lines) is lower when we use the iterative optimization.

Model	N. Lines	N. Points	p_1	p_2	center	E
Pol1p	43	10159	63.48%		(536,356)	2.12279
Pol2pIO	38	11410	95.50%	11.42%	(524.3,362.7)	0.682901
Div1p	41	11229	98.40%		(536,356)	1.83278
Div2pIO	37	11481	118.60%	13.11%	(525.9,362.4)	0.321207

Table 1: Number of lines and points, distortion parameters, distortion center and energy for the calibration pattern in Figure 9 using the one-parameter polynomial and division models (Pol1p and Div1p) and the two-parameter models with iterative optimization (Pol2pIO and Div2pIO).



(a) (b) (c) (d)

Figure 9: Results of applying the method on an image of a calibration pattern: (a) input distorted image, (b) detected edges, (c) lines detected through the iterative optimization process and (d) the corrected image.

Model	N. Lines	N. Points	p_1	p_2	center	E
Pol1p	49	10647	69.13%		(536,356)	3.1627
Pol2pIO	57	11422	86.44%	15.75%	(562.3,344.2)	3.18326
Div1p	51	14649	306.65%		(536,356)	3.90514
Div2pIO	45	16061	1398.2%	22.05%	(510.1,356.1)	4.64709

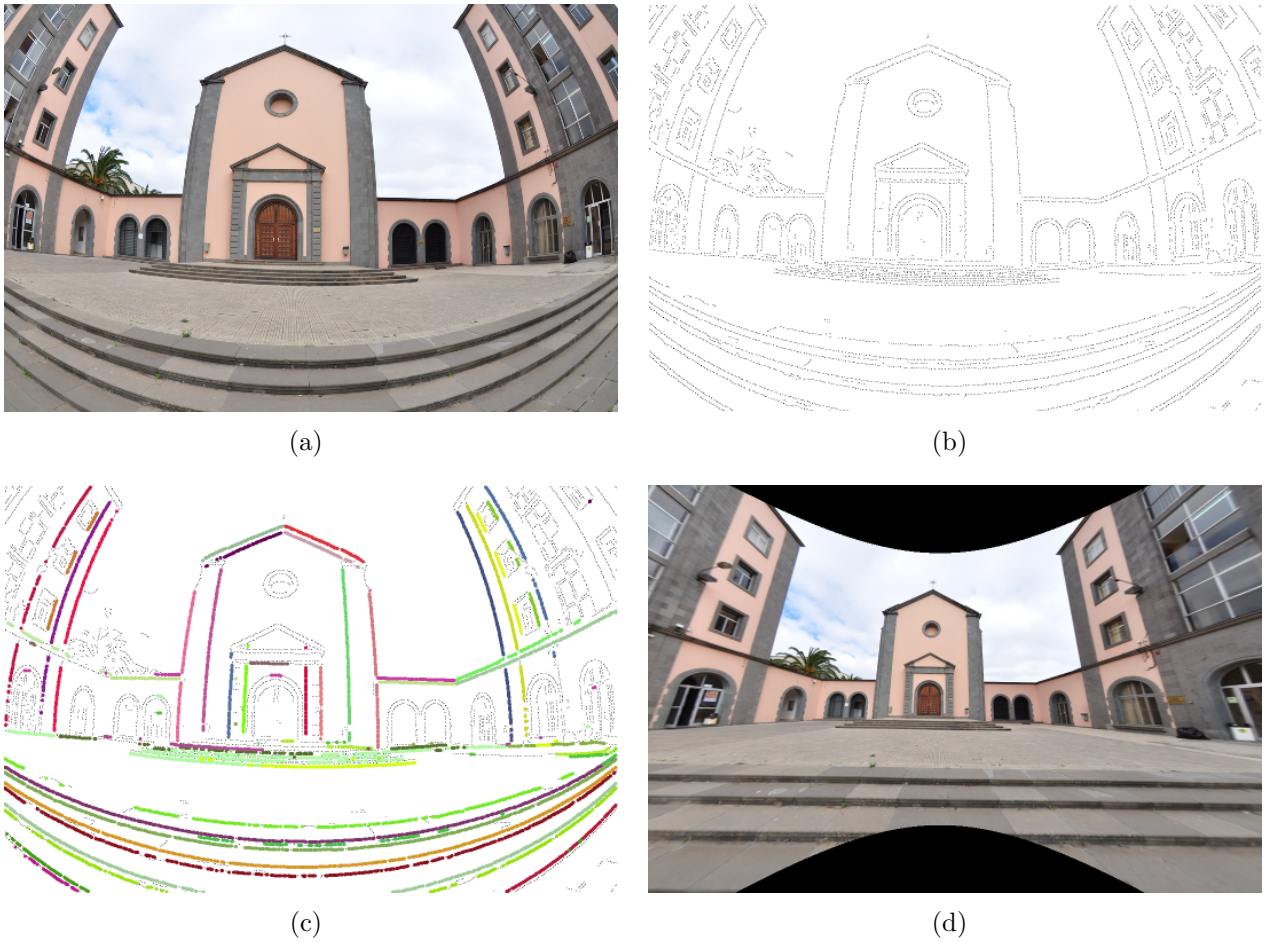
Table 2: Number of lines and points, distortion parameters, distortion center and energy for the image in Figure 10 using the one-parameter polynomial and division models (Pol1p and Div1p) and the two-parameter models with iterative optimization (Pol2pIO and Div2pIO).

Summary of algorithm parameters and default values

In this section, we describe the main parameters of the different functions which have been explained above. In each table, we show a description of the parameters, their default values, and whether they are provided in the on-line interface or not.

Table 3 shows a brief description of the parameters used in the Canny method. The first parameter is the standard deviation for the Gaussian convolution. Moreover, it is necessary to indicate two other parameters for the low and high thresholds of the Canny method. As described above, these values represent a percentile of the gradient norm, by using a number between zero and one.

Table 4 summarizes the main parameters of the cleaning process. The first one is the number of



(a) (b) (c) (d)

Figure 10: Results of applying the method on an image of a building: (a) input distorted image, (b) edges detected, (c) lines detected through the iterative optimization process and (d) the corrected image.

Canny function

Parameters	Default Value	Interface
Standard deviation of the Gaussian	2.0	-
Low threshold for the Canny method	0.7	-
High threshold for the Canny method	0.8	✓

Table 3: Parameters of the Canny function.

neighbors to consider, i.e. the size of the neighborhood in which the cleaning process analyzes what points must be removed. The next parameter is the minimum number of contour points inside this neighborhood. If the number of neighbors of a point is less than this value, it will be removed. Inside the neighborhood, we also take into account the orientation of the edge points. For this reason, we indicate the orientation correspondence. The last parameter is the minimum distance between edge points. Setting this parameter to a value greater than zero, we only keep the point with the lowest curvature inside each neighborhood.

Table 5 describes the main parameters of the improved Hough function. The first one is an input parameter, in which the set of detected edges and their orientations are provided from the Canny method. The next one is the maximum distance allowed between a point and its associated straight line. Together with the distance, it is also necessary to set the maximum difference between the

Figure 11: Some examples of image distortion corrections using the proposed method. All the results have been obtained using the default values of the parameters. Additional results and access to the original images are provided in http://www.ctim.es/WideLensImageDatabase/

Cleaning process

Parameters	Default Value	Interface
Radius of the neighborhoods to take into account	2.0	-
Minimum number of contour points in a neighborhood	2	-
Minimum average scalar product of the points orientation	0.95	-
Minimum distance between contour points	1	-

Table 4: Parameters of the cleaning process.

orientation of the point and the line angle. Another input parameter is the maximum number of lines to detect. The next two parameters set the resolution of the angle and distance, described before. Finally, the initial and final value of the interval for the normalized distortion parameter are provided as well as its resolution.

Improved Hough function

Parameters	Default Value	Interface
Set of detected edges and their orientations	-	-
Maximum distance allowed between points and associated lines (pixels)	3.0	✓
Maximum difference of the point orientation angle and the line angle (degrees)	10.0	✓
Maximum number of lines to detect	100	-
Angle resolution for the angle parameter in the Hough space (degrees)	0.1	-
Distance resolution for the distance parameter in the Hough space (pixels)	1.0	-
Minimum value for the normalized distortion parameter	0.0	✓
Maximum value for the normalized distortion parameter	3.0	✓
Distortion parameter discretization step in our improved Hough space	0.1	-

Table 5: Parameters of the improved Hough function.

5 Conclusions

In this paper we present an iterative optimization algorithm to automatically correct wide-angle lens distortion. Using an improved Hough transform, we obtain a first value for the distortion parameter and a collection of lines. In this transform, the distortion is included as a new dimension. Afterward, an iterative optimization process is applied to obtain a two-parameter model and a better set of primitives. This scheme is applied until the number of points does not increase. Finally, with the improved lens distortion model, the input image is corrected. As we show in the experimental results, the introduction of an iterative optimization process improves the collection of lines which are obtained, providing longer lines. This allows a better correction of the distortion present in the input image, especially when it is significant. The experimental results lead to three main conclusions. First, the use of two distortion parameters instead of a single one improves the quality of the models. Second, the division model performs better than the polynomial model when the distortion is significant, as observed in the most challenging cases. Finally, the introduction of an iterative optimization allows a significant increase in the number of points associated to each distorted straight line. This translates into a better estimation of the distortion parameters and, therefore, into a better correction of the images.

A Algorithms of Stage 2: One-Parameter Lens Distortion Model Estimation

Algorithm 9: orientation_update

```
// Correction of the edge orientation (sin,cos) using the provided lens
   distortion model.
input  : p ; // The edge point
         sine ; // Value of the sine component
         cosine ; // Value of the cosine component
         ldm ; // The lens distortion model
output: // A vector with the corrected orientation

// Point plus the orientation
p_ori ← (p.x + cosine, p.y + sine);

// We apply the model to the original point
p' ← ldm.evaluation(p);

// We apply the model to the point plus the orientation
p'_ori ← ldm.evaluation(p_ori);

// We compute the new orientation and the norm
a ← p'_ori.x − p'.x;
b ← p'_ori.y − p'.y;

norm ← sqrt(a^2 + b^2);

if a^2 + b^2 ≤ 0 then
    corrected_orientation ← (cosine, sine);
else
    corrected_orientation ← (a/norm, b/norm);
end

return corrected_orientation;
```

Algorithm 10: update_contours

```
// Update the contours object with the provided lens distortion model.  If the
   model is empty, just makes a copy.
input  : ldm ; // The lens distortion model.
         contours ; // The input contours object.
         index ; // Vector with the indexes of the edge points.
output: contours_modified ; // The output contours object.

if ldm is identity then
    contours_modified ← contours ;                                  // Copy the model
else
    // Update the contours object using the provided model
    foreach i ∈ index do
        // Update the contour component of contours object
        contours_modified(i).is_contour ← contours(i).is_contour;

        // Correct the position
        ori ← (contours(i).x, contours(i).y);
        contours_modified(i) ← ldm.evaluation(ori);

        // Correct the orientation
        corrected_orientation ← orientation_update( ori, contours(i).sin, contours(i).cos, ldm);

        // Update the value of the sine and cosine
        contours_modified(i).cos ← corrected_orientation.x;
        contours_modified(i).sin ← corrected_orientation.y;
        if contours_modified(i).sin<0 then
            contours_modified(i).cos ← - contours_modified(i).cos;
            contours_modified(i).sin ← - contours_modified(i).sin;
        end
    end
end
```

Algorithm 11: voting

```
// Fill the voting matrix corresponding to the k slice
input  : contours_modified ; // Input contours information.
         ldm ; // The lens distortion model of slice k.
         step_angle ; // Step for the angle dimension in the Hough voting matrix.
         height_score, width_score ; // Size of the voting matrix.
         angle_increment ; // Increment for the angle interval
         distance_resolution ; // Resolution of the distance interval (width of the
voting matrix)
         sine, cosine ; // Arrays of samples of sine and cosine functions.
         index ; // Vector with the indexes of the edge points.
output: score_k ; // The voting matrix
```

$(x_c, y_c) \leftarrow$ (ldm.distortion_center.x, ldm.distortion_center.y);

```
// Do the voting process only with the points inside the vector index
```

for $q \in index$ **do**

```
    // Compute the distortion model according to the type
```

$(x_3, y_3) \leftarrow (x_2, y_2) \leftarrow$ (contours_modified(q).x, contours_modified(q).y);

if $ldm.get\_d(1) \neq 0$ **then**

$(x_3, y_3) \leftarrow ldm.evaluation((x_2, y_2))$;

end

```
    // Orientation correction
```

corrected_orientation ← orientation_update((x_2, y_2), contours_modified(q).sin, contours_modified(q).cos, ldm);

```
    // Estimate the angle interval
```

if $corrected\_orientation.y \geq 0$ **then**

angle $\leftarrow \frac{\pi - \arctan(corrected\_orientation.y, corrected\_orientation.x)}{step\_angle}$;

else

angle $\leftarrow \frac{\pi - \arctan(-corrected\_orientation.y, -corrected\_orientation.x)}{step\_angle}$;

end

if $angle = height\_score$ **then** angle $\leftarrow height\_score - 1$;

(l_min, l_max) $\leftarrow angle \pm angle\_increment$;

id ← 2;

```
    // Algorithm 12
```

end

Algorithm 12: This algorithm is included in 11

for $l \leftarrow l\_min$ **to** l_max **do**
  angle_index $\leftarrow$ l; sine_sign $\leftarrow 1$;
  if $l < 0$ **then** $angle\_index \leftarrow height\_score + l$;
  if $l \geq height\_score$ **then**
    $angle\_index \leftarrow l - height\_score$;
    $sine\_sign \leftarrow -1$;
  end
  distance $\leftarrow -cosine(angle\_index) \cdot y_3 - sine(angle\_index) \cdot x_3$;
  for $nd \leftarrow -id$ **to** id **do**
    d $\leftarrow \frac{width\_score}{2} + \frac{distance}{distance\_resolution} + nd$;
    if $0 \leq d < width\_score$ **then**
      $rect\_distance \leftarrow |distance + cosine(angle\_index) \cdot y_3 + sine\_sign \cdot sine(angle\_index) \cdot x_3 + nd \cdot distance\_resolution|$;
      $score\_k(d, angle\_index) \leftarrow score\_k(d, angle\_index) + \frac{1}{1+rect\_distance}$;
    end
  end
end

Algorithm 13: capturing_points

```
// Associate points to lines according to their distance
input  : contours_modified, contours ; // Contours corrected with the input lens
         distortion model.  Original subpixel_contours object.
         angle_point_orientation_max_difference ; // Maximum angle difference between
the orientation of the edge point and the orientation of the line.
         i_primitive, i_primitive_corrected, i_primitive_original ; // Set of detected,
corrected and original primitives, respectively.
         nlines_plus ; // Total amount of lines to detect.
         max_distance ; // Maximum distance between a point an its associated
line.
         index ; // Indexes of the edge points.
```

dot_product_min $\leftarrow cos(\frac{\pi}{180} \cdot angle\_point\_orientation\_max\_difference)$;
ld $\leftarrow$ i_primitive.distortion;

for $k \leftarrow 0$ **to** *nlines_plus* **do** i_primitive.lines(k).points.clear() ;

foreach $q \in index$ **do**
    $p \leftarrow$ (contours_modified(q).x, contours_modified(q).y);
    $p_{ori} \leftarrow$ (contours(q).x, contours(q).y);
    $p_d \leftarrow$ ld.evaluation(p);
    corrected_orientation $\leftarrow$ orientation_update(p, contours_modified(q).sin, contours_modified(q).cos, i_primitive.distortion);
    for $k \leftarrow 0$ **to** *nlines_plus* **do**
        if $corrected\_orientation.x \cdot i\_primitive.lines(k).b - corrected\_orientation.y \cdot i\_primitive.lines(k).a < dot\_product\_min$ **then** continue;
        if $i\_primitive.lines(k).evaluation(p_d) < max\_distance$ **then**
            i_primitive.lines(k).points $\leftarrow p$;
            i_primitive_corrected.lines(k).points $\leftarrow p_d$;
            i_primitive_original.lines(k).points $\leftarrow p_{ori}$;
            break;
        end
    end
end

Algorithm 14: erase_point

```
// Erasing a point ip in the line il in the image primitives objects
input  :  i_primitive_original, i_primitive, i_primitive_corrected, il
output:  ip
```

i_primitive_original.lines(il).erase_point(ip);
i_primitive.lines(il).erase_point(ip);
i_primitive_corrected.lines(il).erase_point(ip);

ip $\leftarrow$ ip - 1;

Algorithm 15: ensure_consistent_line_orientations

```
// Ensuring the consistence of lines orientation, according to the predominant
   orientation of their points
input : i_primitive, i_primitive_corrected, i_primitive_original ; // Set of detected,
        corrected and original, primitives, respectively.
        contours ; // Original contours object.
        width ; // Image width.

for il ← 0 to i_primitive_original.num_lines do
    pos_count ← 0, neg_count ← 0;
    for ip ← 0 to i_primitive_original.lines(il).num_points do
        orientation_sign ← compute_orientation_sign( image_primitive_original.lines(il), width,
        ip, contours);
        if orientation_sign > 0 then pos_count ← pos_count + 1 ;
        if orientation_sign < 0 then neg_count ← neg_count + 1 ;
    end
    if pos_count ≠ 0 AND neg_count ≠ 0 then
        if pos_count > neg_count then s ← 1;
        else s ← -1 ;
        for ip ← 0 to i_primitive_original.lines(il).num_points do
            orientation_sign ← compute_orientation_sign( image_primitive_original.lines(il),
            width, ip, contours);
            if s · orientation_sign < 0 then
                erase_point( i_primitive_original, i_primitive, i_primitive_corrected, il, ip);
            end
        end
    end
end
```

Algorithm 16: compute_orientation_sign

```
// Computing the orientation sign of a point
input : line, width, ip, contours

current_point ← line.points(ip);
pos ← width · round(current_point.y) + round(current_point.x);

(a, b) ← (line.a, line.b);

return b · contours(pos).cos − a · contours(pos).sin;
```

Algorithm 17: remove_and_join

```
// Remove short lines and to join lines which are very close
input : i_primitive, i_primitive_corrected, i_primitive_original ; // Set of detected,
        corrected and original primitives, respectively.
        angle_point_orientation_max_difference ; // Maximum angle difference between
the orientation of the edge point and the orientation of the line.
        distance_point_line_max ; // Maximum allowed distance between the edge
point and the associated line.
        width ; // Image width.
        contours ; // Original contours object.
output: i_primitive, i_primitive_corrected, i_primitive_original ; // The set of primitives
        with the short lines removed and with the close lines joined.
```

MIN_POINTS_IN_LINE ← 20;
MAX_DISTANCE_BETWEEN_LINES ← 10;

dot_product_min ← max(0.95, $cos(\frac{\pi}{180} \cdot 2 \cdot$ angle_point_orientation_max_difference));

min_line_points ← max(20, $0.05 \cdot i\_primitive.lines(0).number\_points$);

distance_line_line_min ← min(10, $2 \cdot distance\_point\_line\_max$);

for $k \leftarrow 0$ **to** $i\_primitive.number\_lines$ **do**
  if $i\_primitive.lines(k).number\_points < min\_line\_points$ **then**
    i_primitive.erase_line(k);
    i_primitive_corrected.erase_line(k);
    i_primitive_original.erase_line(k);
    k ← k - 1;
    continue;
  end
  for $l \leftarrow k + 1$ **to** $i\_primitive.number\_lines$ **do**
    // We check the number of points
    if $NOT\ (i\_primitive.lines(l).number\_points > 0)$ **then** continue;
    $aux \leftarrow i\_primitive.lines(k).a \cdot i\_primitive.lines(l).a + i\_primitive.lines(k).b \cdot i\_primitive.lines(l).b$;
    if $aux > dot\_product\_min$ **then**
      // We check the orientation of the first point of each primitive
      a_1 ← i_primitive_corrected.lines(k).a;
      a_2 ← i_primitive_corrected.lines(l).a;
      b_1 ← i_primitive_corrected.lines(k).b;
      b_2 ← i_primitive_corrected.lines(l).b;
      point ← i_primitive_original.lines(k).points(0);
      // Algorithm 18
    end
  end
end

Algorithm 18: remove_and_join. Continuation of Algorithm 17

```
pos ← width · point.y + point.x;
sign1 ← b_1 · contours(pos).cos − a_1 · contours(pos).sin;
point ← i_primitive_original.lines(k).points(0);
pos ← width · point.y + point.x;
sign2 ← b_2 · contours(pos).cos − a_2 · contours(pos).sin;
if sign1 · sign2 ≤ 0 then continue;
// We check the average distance of the points to the line
aux ← 0;
m ← 0;
foreach p ∈ i_primitive_corrected.lines(l).points do
    dist ← i_primitive.lines(k).evaluation( p );
    aux ← aux + dist;
    if dist > 5 · distance_point_line_max then break;
    m ← m + 1;
end
if m < i_primitive.lines(l).number_points then continue;
if aux / i_primitive.lines(l).number_points < MIN_POINTS_IN_LINE then
    aux ← 0;
    foreach p ∈ i_primitive_corrected.lines(l).points do
        (a, b, c) ← i_primitive.lines(k).abc;
        d ← a · p.x + b · p.y + c;
        np2d ← (p.x − d · a, p.y − d · b);
        aux ← aux + i_primitive_corrected.lines(k).distance( np2d );
    end
    if aux / i_primitive_corrected.lines(l).number_points > MAX_DISTANCE_BETWEEN_LINES then
        // We add the points of the line to the line point structure
        i_primitive.lines(k).add_points ← i_primitive.lines(l);
        i_primitive_corrected.lines(k).add_points ← i_primitive_corrected.lines(l);
        i_primitive_original.lines(k).add_points ← i_primitive_original.lines(l);
        // We remove the line points structure
        i_primitive.erase_line( l );
        i_primitive_corrected.erase_line( l );
        i_primitive_original.erase_line( l );
        l ← l -1;
    end
end
```

Algorithm 19: recompute_line_equations

```
// Recompute the equations of the lines using their points
input  : i_primitive, i_primitive_corrected, i_primitive_original ; // Set of detected,
         corrected and original primitives, respectively.
         lens_distortion_estimation ; // Flag for considering the lens distortion.
output: i_primitive, i_primitive_corrected, i_primitive_original ; // The set of primitives
         with the recomputed equations

if lens_distortion_estimation = True then
    for i ← 0 to i_primitive.number_lines do
        if i_primitive.lines(i).number_points > 2 then
            i_primitive_corrected.lines(i).points_to_equation();
            (a, b, c) ← i_primitive_corrected.lines(i).abc;
            i_primitive.lines(i).abc ← (a, b, c);
            i_primitive_original.lines(i).abc ← (a, b, c);
        end
    end
else
    for i ← 0 to i_primitive.number_lines do
        if i_primitive.lines(i).number_points > 2 then
            i_primitive.lines(i).points_to_equation();
            (a, b, c) ← i_primitive.lines(i).abc;
            i_primitive_corrected.lines(i).abc ← (a, b, c);
            i_primitive_original.lines(i).abc ← (a, b, c);
        end
    end
end
```

B Algorithms of Stage 3: Iterative Optimization

Algorithm 20: Computing the gradient of E

input : p_1, p_2 ; // normalized value for the distortion parameter k_1 and k_2.
x_c, y_c ; // (x, y) coordinate of the distortion center.
h_1 ; // discretization step for p_1 and p_2.
h_2 ; // discretization step for the coordinates of the distortion center.
output: ; // The gradient vector

$grad_E(0) \leftarrow -\frac{(E(p_1+h_1,p_2,x_c,y_c)-E(p_1,p_2,x_c,y_c))}{h_1}$;
$grad_E(1) \leftarrow -\frac{(E(p_1,p_2+h_1,x_c,y_c)-E(p_1,p_2,x_c,y_c))}{h_1}$;
$grad_E(2) \leftarrow -\frac{(E(p_1,p_2,x_c+h_2,y_c)-E(p_1,p_2,x_c,y_c))}{h_2}$;
$grad_E(3) \leftarrow -\frac{(E(p_1,p_2,x_c,y_c+h_2)-E(p_1,p_2,x_c,y_c))}{h_2}$;

Algorithm 21: Computing the Hessian of E

input : p_1, p_2 ; // normalized value for the distortion parameter k_1 and k_2.
x_c, y_c ; // (x, y) coordinate of the distortion center.
h_1 ; // discretization step for p_1 and p_2.
h_2 ; // discretization step for the coordinates of the distortion center.
output: ; // The Hessian matrix.

$hess_E(0,0) \leftarrow \left(\frac{(E(p_1+h_1,p_2,x_c,y_c)+E(p_1-h_1,p_2,x_c,y_c)-2E(p_1,p_2,x_c,y_c))}{h_1^2}\right)$;
$hess_E(1,1) \leftarrow \left(\frac{(E(p_1,p_2+h_1,x_c,y_c)+E(p_1,p_2-h_1,x_c,y_c)-2E(p_1,p_2,x_c,y_c))}{h_1^2}\right)$;
$hess_E(2,2) \leftarrow \left(\frac{(E(p_1,p_2,x_c+h_2,y_c)+E(p_1,p_2,x_c-h_2,y_c)-2E(p_1,p_2,x_c,y_c))}{h_2^2}\right)$;
$hess_E(3,3) \leftarrow \left(\frac{(E(p_1,p_2,x_c,y_c+h_2)+E(p_1,p_2,x_c,y_c-h_2)-2E(p_1,p_2,x_c,y_c))}{h_2^2}\right)$;

$hess_E(0,1) \leftarrow hess_E(1,0) \leftarrow \frac{(E(p_1+h_1,p_2+h_1,x_c,y_c)-E(p_1+h_1,p_2,x_c,y_c))-(E(p_1,p_2+h_1,x_c,y_c)-E(p_1,p_2,x_c,y_c))}{h_1^2}$;
$hess_E(0,2) \leftarrow hess_E(2,0) \leftarrow \frac{(E(p_1+h_1,p_2,x_c+h_2,y_c)-E(p_1+h_1,p_2,x_c,y_c))-(E(p_1,p_2,x_c+h_2,y_c)-E(p_1,p_2,x_c,y_c))}{h_1h_2}$;
$hess_E(0,3) \leftarrow hess_E(3,0) \leftarrow \frac{(E(p_1+h_1,p_2,x_c,y_c+h_2)-E(p_1+h_1,p_2,x_c,y_c))-(E(p_1,p_2,x_c,y_c+h_2)-E(p_1,p_2,x_c,y_c))}{h_1h_2}$;
$hess_E(1,2) \leftarrow hess_E(2,1) \leftarrow \frac{(E(p_1,p_2+h_1,x_c+h_2,y_c)-E(p_1,p_2+h_1,x_c,y_c))-(E(p_1,p_2,x_c+h_2,y_c)-E(p_1,p_2,x_c,y_c))}{h_1h_2}$;
$hess_E(1,3) \leftarrow hess_E(3,1) \leftarrow \frac{(E(p_1,p_2+h_1,x_c,y_c+h_2)-E(p_1,p_2+h_1,x_c,y_c))-(E(p_1,p_2,x_c,y_c+h_2)-E(p_1,p_2,x_c,y_c))}{h_1h_2}$;
$hess_E(2,3) \leftarrow hess_E(3,2) \leftarrow \frac{(E(p_1,p_2,x_c+h_2,y_c+h_2)-E(p_1,p_2,x_c+h_2,y_c))-(E(p_1,p_2,x_c,y_c+h_2)-E(p_1,p_2,x_c,y_c))}{h_2^2}$;

C Algorithms of Stage 4: Image Distortion Correction

Algorithm 22: undistort_image_inverse

input : input_image ; `// Input image.`
ldm ; `// Lens distortion model.`
image_amplification_factor ; `// image_amplification_factor.`
output: output_image ; `// The input image with the distortion correction.`

`// Calculate maximum distance from center to a corner`
ldm_center ← ldm.distortion_center;
max_distance_corner ← sqrt(update_rsqmax(ldm_center, input_image.width, input_image.height));

`// Build full polynomial (including null odd degrees)`
Na ← $2 \cdot (ldm.d.size() - 1)$;
if $ldm.d.size() < 2$ **then return** *input_image* ;
a(0) ← ldm.d(0);

for $i \leftarrow 1$ **to** $ldm.d.size()$ **do**
  $a(2\,i - 1) \leftarrow 0$;
  $a(2\,i) \leftarrow$ ldm.d(i);
end

while $Na > 0$ *AND* $a(Na) = 0$ **do** Na ← Na - 1 ;

`// We update the max_distance_corner according to lens distortion max displacement, and the type of distortion model`
if $get\_type(ldm) = POLYNOMIAL$ **then**
  step ← 0;
  power ← max_distance_corner;
  for $k \leftarrow 0$ **to** Na **do**
    step ← $step + power \cdot a(k)$;
    power ← power · max_distance_corner;
  end
  if $step > max\_distance\_corner$ **then** max_distance_corner ← step ;
else
  if $ldm.d.size() = 2$ **then** max_distance_corner ← $\frac{max\_distance\_corner}{ldm.d(0)+ldm.d(1)\cdot max\_distance\_corner^2}$;
  else max_distance_corner ← $\frac{max\_distance\_corner}{ldm.d(0)+ldm.d(1)\cdot max\_distance\_corner^2+ldm.d(2)\cdot max\_distance\_corner^4}$;
end

`// Algorithm 23`

Algorithm 23: Continuation of the Algorithm 22

```
// We build the lens distortion inverse vector
if get_type(ldm) = POLYNOMIAL then
    if Na < 2 then return input_image ;
    if build_l1r_vector( l1r, max_distance_corner, Na, a) = -1 then return input_image ;
else
    if ldm.d.size() = 2 then
        if build_l1r_quotient_vector(l1r, max_distance_corner, a, Na) < 0 then return
        input_image ;
    else
        compute_division_l1r(l1r, ldm.d, max_distance_corner);
    end
end
...
for nc ← 0 to 3 do
    n2 ← nc · size;
    for i ← 0 to height do
        for j ← 0 to width do
            temp ← ( j · scale − t.x, i · scale − t.y );
            distance_center ← (ldm_center - temp).norm();
            // Interpolation
            ind ← distance_center;
            if ind ≥ l1r.size() then continue;
            dl1r ← l1r(ind) + (distance_center - ind) · (l1r(ind+1) - l1r(ind));
            p.x ← ldm_center.x + (temp.x - ldm_center.x) · dl1r;
            p.y ← ldm_center.y + (temp.y - ldm_center.y) · dl1r;
            (n, m) ← (p.x, p.y);
            if 0 ≤ m < height0 AND 0 ≤ n < width0 then
                // Colour interpolation
                (dj, di) ← (p.x - n, p.y - m);
                w ← (1 - di) · (1 - dj);
                accum ← accum + w · input_image(m · width0 + n+n2);
                w_accum ← w_accum + w;
                ...
                if w_accum > 0 then output_image(i · width + j + n2) ← accum/w_accum ;
            end
        end
    end
end
return output_image;
```

Acknowledgments

This work has been partially supported by the MICINN project reference MTM2010-17615 (Ministry of Science and Innovation, Spain). The authors want to thank Pascal Monasse and Jose Luis Lisani for his comments. We also thank the reviewers for their very valuable corrections and comments.

Image Credits

All images by the authors (license CC-BY-SA).

References

[1] M. Alemán-Flores, L. Alvarez, L. Gomez, and D. Santana-Cedrés, *Automatic Lens Distortion Correction Using One-Parameter Division Models*, Image Processing On Line, 4 (2014), pp. 327–343. http://dx.doi.org/10.5201/ipol.2014.106.

[2] M. Alemán-Flores, L. Alvarez, L. Gomez, and D. Santana-Cedrés, *Line detection in images showing significant lens distortion and application to distortion correction*, Pattern Recognition Letters, 36 (2014), pp. 261 – 271. http://dx.doi.org/10.1016/j.patrec.2013.06.020.

[3] L. Alvarez, L. Gomez, and R. Sendra, *An algebraic approach to lens distortion by line rectification*, Journal of Mathematical Imaging and Vision, 39 (2008), pp. 36–50. http://dx.doi.org/10.1007/s10851-009-0153-2.

[4] ——, *Algebraic Lens Distortion Model Estimation*, Image Processing On Line, 1 (2010). https://doi.org/10.5201/ipol.2010.ags-alde.

[5] ——, *Accurate depth dependent lens distortion models: an application to planar view scenarios*, Journal of Mathematical Imaging and Vision, 39 (2011), pp. 75–85. http://dx.doi.org/10.1007/s10851-010-0226-2.

[6] L. Alvarez and L. Mazorra, *Signal and Image Restoration using Shock Filters and Anisotropic Diffusion*, SIAM Journal on Numerical Analysis, 31 (1994), pp. 590–605. http://dx.doi.org/10.1137/0731032.

[7] D.C. Brown, *Close-range camera calibration*, Photogrammetric Engineering, 37 (1971), pp. 855–866.

[8] J. Canny, *A Computational Approach to Edge Detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 8 (1986), pp. 679–698. http://dx.doi.org/10.1109/TPAMI.1986.4767851.

[9] R. Cucchiara, C. Grana, A. Prati, and R. Vezzani, *A Hough transform-based method for radial lens distortion correction.*, in Proceedings of 12th International Conference on Image Analysis and Processing (ICIAP), 2003, pp. 182–187. http://dx.doi.org/10.1109/ICIAP.2003.1234047.

[10] A. Desolneux, L. Moisan, and J-M. Morel, *Meaningful Alignments*, International Journal of Computer Vision, 40 (2000), pp. 7–23. http://dx.doi.org/10.1023/A%3A1026593302236.

[11] F. Devernay and O. Faugeras, *Straight lines have to be straight*, Machine Vision and Applications, 13 (2001), pp. 14–24. http://dx.doi.org/10.1007/PL00013269.

[12] O. Faugeras, *Three-dimensional computer vision*, MIT Press, 1993. ISBN 0262061589.

[13] A. W. Fitzgibbon, *Simultaneous linear estimation of multiple view geometry and lens distortion*, Proceedings of IEEE International Conference on Computer Vision and Pattern Recognition, (2001), pp. 125–132. http://dx.doi.org/10.1109/CVPR.2001.990465.

[14] R. Grompone von Gioi, J. Jakubowicz, J-M. Morel, and G. Randall, *On Straight Line Segment Detection*, Journal of Mathematical Imaging and Vision, 32 (2008), pp. 313–347. http://dx.doi.org/10.1007/s10851-008-0102-5.

[15] R. Grompone von Gioi, J. Jakubowicz, J-M. Morel, and G. Randall, *LSD: a Line Segment Detector*, Image Processing On Line, 2 (2012), pp. 35–55. http://dx.doi.org/10.5201/ipol.2012.gjmr-lsd.

[16] R. I. Hartley and A. Zisserman, *Multiple view geometry in computer vision*, Cambridge University Press, 2004. ISBN 0521540518, http://dx.doi.org/10.2277/0511188951.

[17] C. Hughes, M. Glavin, E. Jones, and P. Denny, *Review of geometric distortion compensation in fish-eye cameras*, in IET Irish Signals and Systems Conference (ISSC), 2008, pp. 162–167. http://dx.doi.org/10.1049/cp:20080656.

[18] S. Kang, *Radial distortion snakes*, Proceedings of IEICE Transactions on Information and Systems, (2000), pp. 1603–1611.

[19] T-Y. Lee, T-S. Chang, C-H. Wei, S-H. Lai, K-C. Liu, and H-S. Wu, *Automatic Distortion Correction of Endoscopic Images Captured With Wide-Angle Zoom Lens*, IEEE Transactions on Biomedical Engineering, 60 (2013), pp. 2603–2613. http://dx.doi.org/10.1109/TBME.2013.2261816.

[20] R. Lenz, *Linsenfehlerkorrigierte Eichung von Halbleiterkameras mit Standardobjektiven für hochgenaue 3D - Messungen in Echtzeit*, in Mustererkennung 1987, vol. 149 of Informatik-Fachberichte, Springer Berlin Heidelberg, 1987, pp. 212–216. ISBN 978-3-540-18375-4, http://dx.doi.org/10.1007/978-3-662-22205-8_55.

[21] D. Santana-Cedrés, L. Gomez, M. Alemán-Flores, A. Salgado, J. Esclarín, L. Mazorra, and L. Alvarez, *Invertibility and Estimation of Two-parameter Polynomial and Division Lens Distortion Models*, SIAM Journal on Imaging Sciences, 8 (2016), pp. 1574–1606. http://dx.doi.org/10.1137/151006044.

[22] A. Wang, T. Qiu, and L. Shao, *A simple method to radial distortion correction with centre of distortion estimation*, Journal of Mathematical Imaging and Vision, 35 (2009), pp. 165–172. http://dx.doi.org/10.1007/s10851-009-0162-1.