Discorpy Documentation

NSLS-II, Brookhaven National Laboratory, US; Diamond Light S

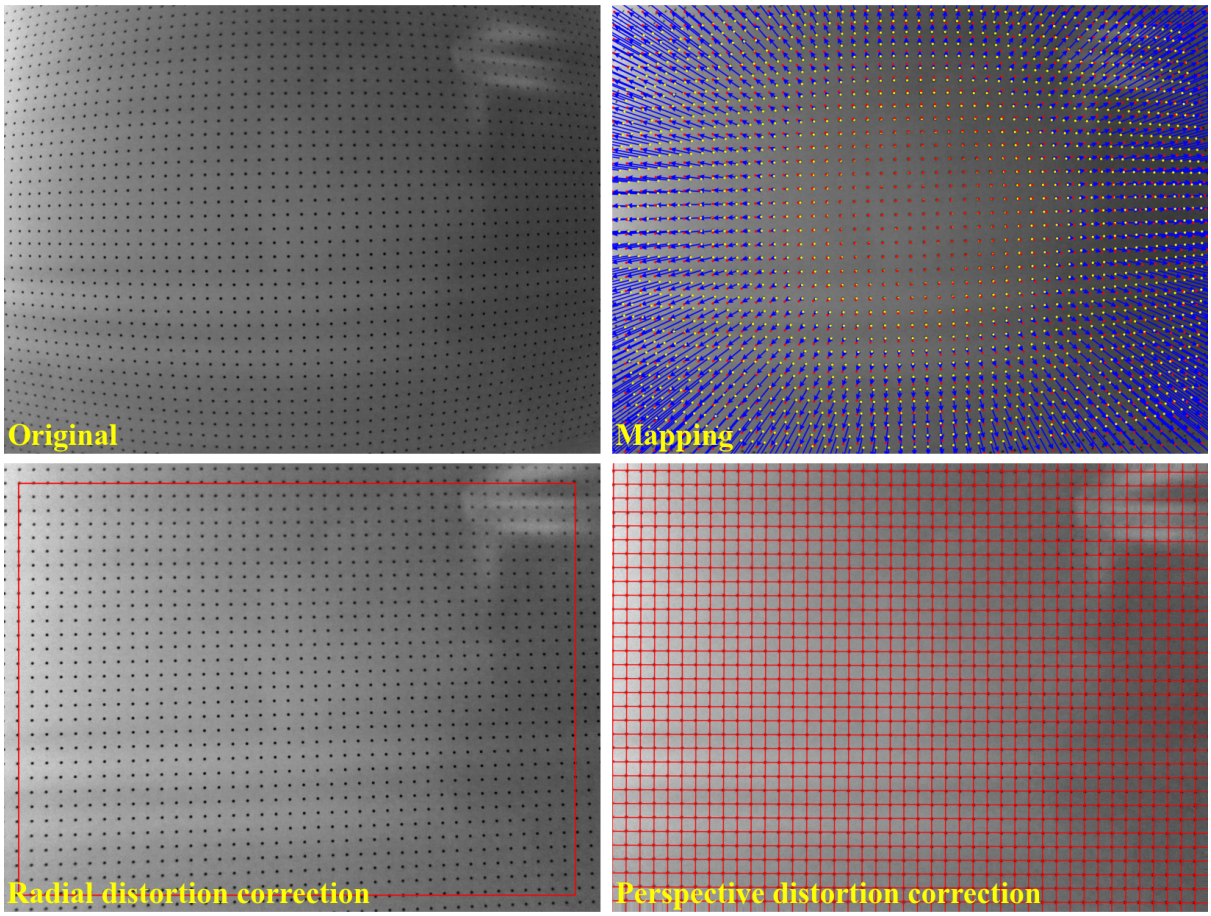
Nov 21, 2023

CONTENTS

Discorpy is an open-source Python package for correcting radial distortion with sub-pixel accuracy as required by tomography detector systems [C1]. It is used to calculate parameters of a correction model, which are the center of distortion and the polynomial coefficients, using a grid pattern image. From version 1.4, perspective distortion correction and methods for processing line-pattern and chessboard (checkerboard) images were added to the package.

A key feature of Discorpy is that radial distortion, the center of distortion, and perspective distortion are determined and corrected independently using a single calibration image. Discorpy was developed for calibrating lens-coupled detectors of tomography systems but it also can be used for commercial cameras.

Showcases: https://discorpy.readthedocs.io/en/latest/usage.html#demonstrations



Source code: https://github.com/DiamondLightSource/discorpy

Author: Nghia T. Vo - NSLS-II, Brookhaven National Laboratory, US; Diamond Light Source, UK.

Keywords: Camera calibration, radial lens distortion, perspective distortion, distortion correction, tomography.

CHAPTER

ONE

CONTENTS

1.1 Installation

Discorpy is a Python library not an app. Users have to write Python codes to process their data. For beginners, a quick way to get started with Python programming is to install Anaconda, then follow instructions here. There are many IDE software can be used to write and run Python codes e.g Spyder, Pydev, Pycharm, or Visual Studio Code. After installing these software, users need to configure Python interpreter by pointing to the installed location of Anaconda. Each software has instructions of how to do that. There is a list of standard Python libraries shipped with Anaconda, known as the *base* environment. To install a Python package out of the list, it's a good practice that users should create a separate environment from the base. This tutorial gives an overview about Python environment. Instructions of how to create a new environment and how to install new packages are here and here. Note that the IDE software needs to be reconfigured to point to the new environment. If users don't want to install Anaconda which is quite heavy due to the base environment shipped with it, Miniconda is enough to customize Python environment.

1.1.1 Using conda

- Install Miniconda as instructed above.
- Open terminal or command prompt and run the following commands:
  - If install to an existing environment:

```
conda install -c conda-forge discorpy
```

  - If install to a new environment:

```
conda create -n discorpy
conda activate discorpy
conda install python
conda install -c conda-forge discorpy
```

1.1.2 Using pip

- Install Miniconda as instructed above.
- Open terminal or command prompt and run the following commands:
  - If install to an existing environment:

```
pip install discorpy
```

  - If install to a new environment:

```
conda create -n discorpy
conda activate discorpy
conda install python
pip install discorpy
```

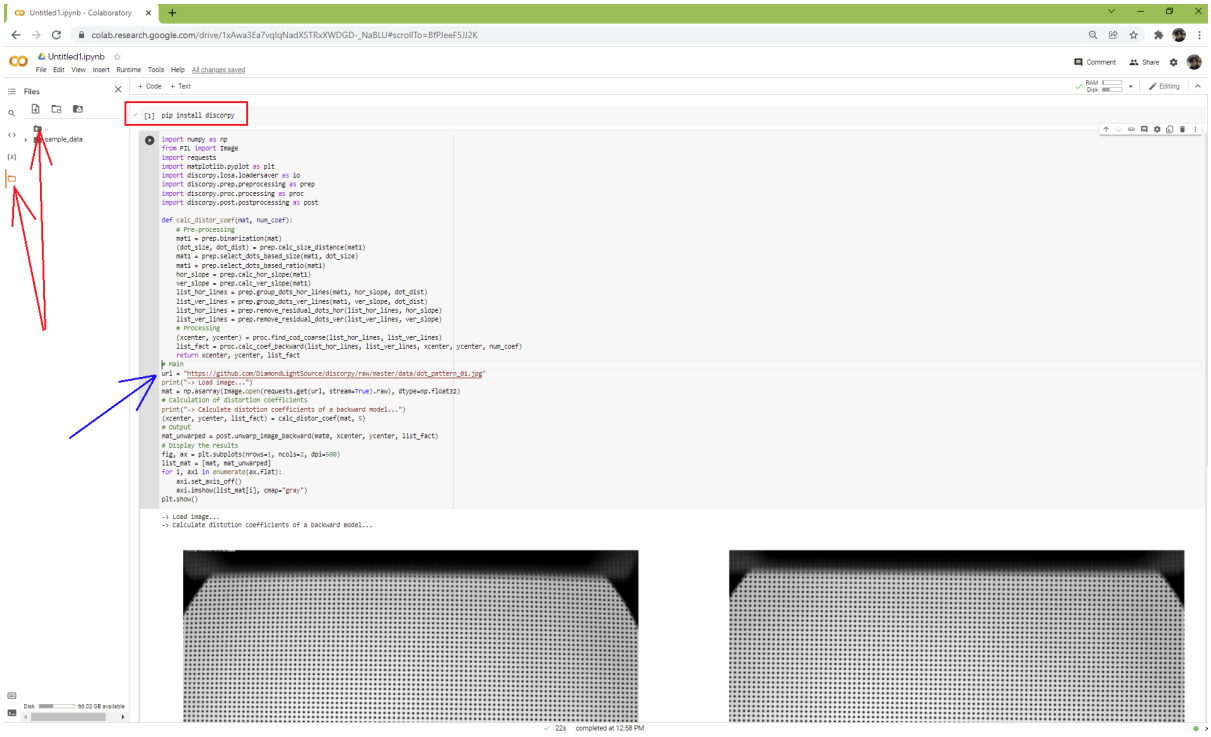
1.1.3 Installing from source

- Download the source from github (click-> Code -> Download ZIP). Unzip to a local folder.
- Install Miniconda or Anaconda as shown above.
- Open command prompt, navigate to the source folder, run the following commands:

```
conda create -n discorpy
conda activate discorpy
conda install python
python setup.py install
```

1.1.4 Using Google Colab

Above instructions are for installing Discorpy locally. Users can also run Discorpy remotely on Google Colab. This requires a Google account. As shown in the screenshot below, Discorpy is installed by running "pip install discorpy" at the first line of the notebook. Images can be upload/download to/from a Google Drive or a url address.



1.2 Tutorials

1.2.1 Causes of distortion

In a lens-coupled camera or detector, there are two major types of distortion which may often be seen in an acquired image: radial distortion and perspective distortion (Fig. 1).

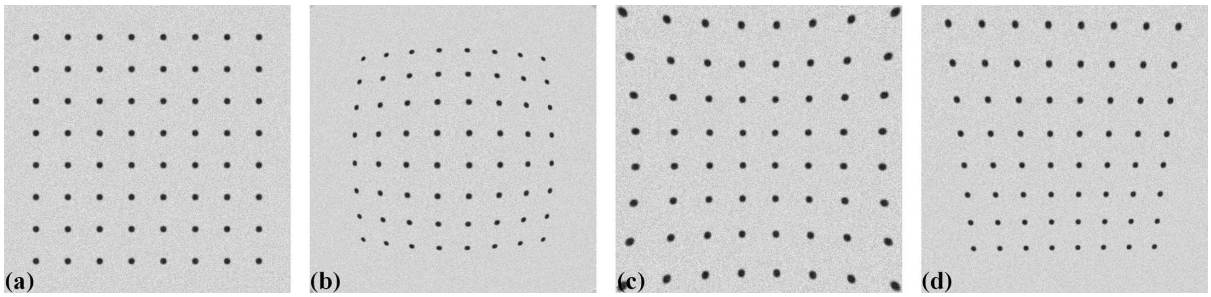


Fig. 1: (a) Non-distorted image. (b) Barrel type of radial distortion. (c) Pincushion type of radial distortion. (d) Perspective distortion.

Radial distortion is caused by the increase or decrease of the magnification of a lens with respect to the radius from the optical axis as shown in Fig. 2.

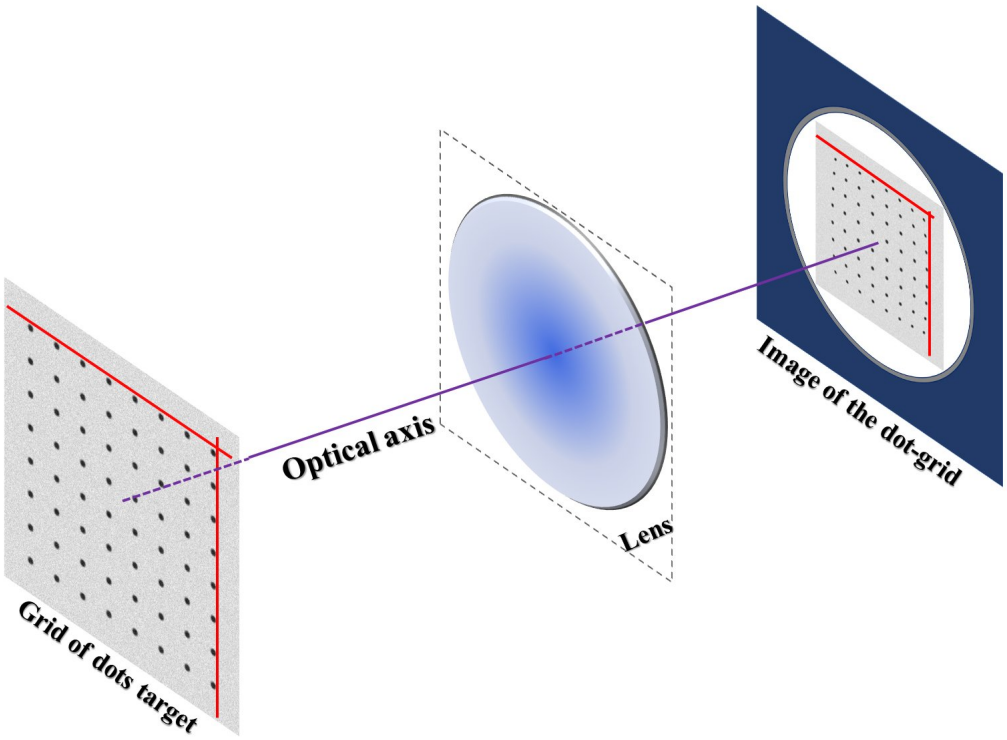


Fig. 2: Cause of radial distortion.

Perspective distortion can be caused by two sources: a lens-plane is not parallel to a camera-sensor plane, known as tangential distortion, (Fig. 3) and/or an object-plane is not parallel to a camera-sensor plane (Fig. 4).

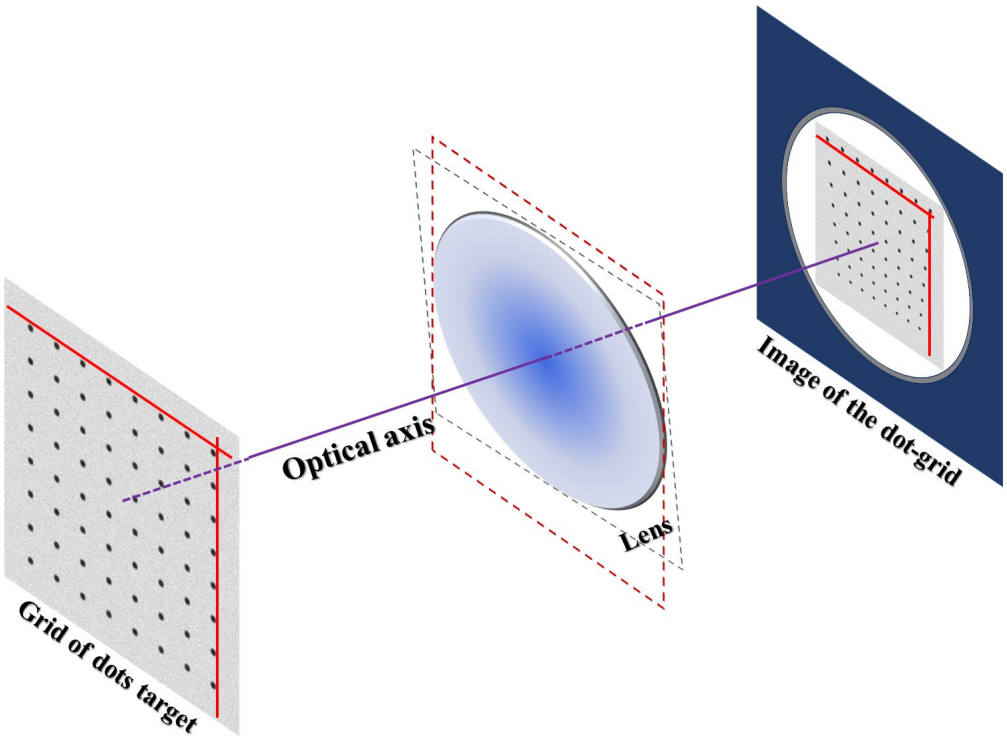


Fig. 3: Lens-plane and sensor-plane are not parallel.

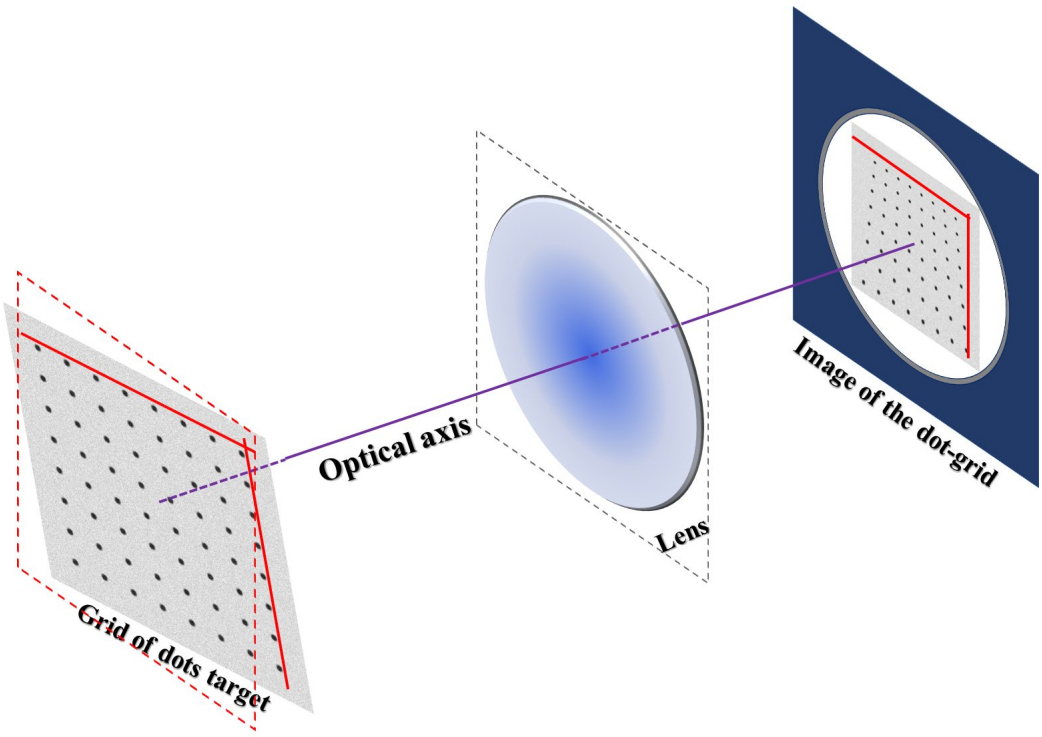


Fig. 4: Object-plane and sensor-plane are not parallel.

1.2.2 Methods for correcting distortions

Introduction

For correcting radial and/or perspective distortion, we need to know a model to map between distorted space and undistorted space. Mapping from the undistorted space to the distorted space is the forward mapping (Fig. 5). The reverse process is the backward mapping or inverse mapping (Fig. 6).

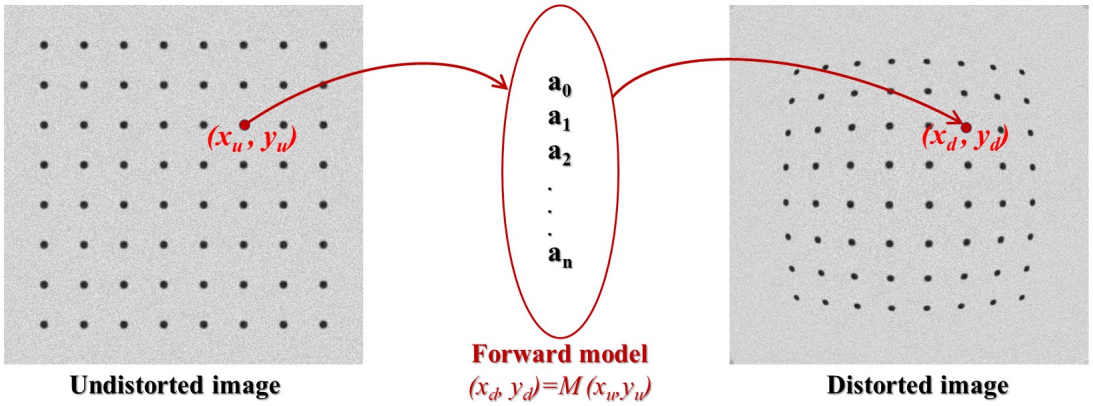


Fig. 5: Forward mapping.

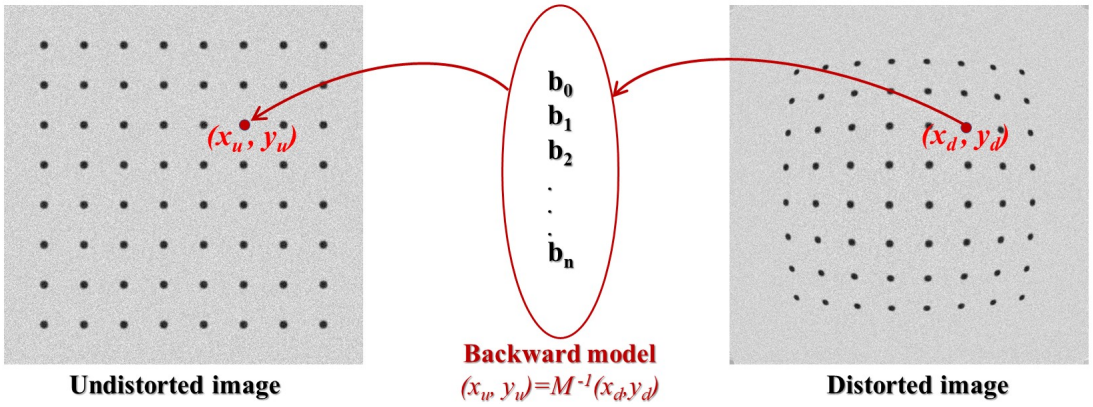


Fig. 6: Backward mapping.

There are many models which can be chosen from literature [R1, R2, R3] such as polynomial, logarithmic, field-of-view, or matrix-based models to describe the relationship between the undistorted space and distorted space. Some models were proposed for only one type of distortion while others are for both distortion types including the location of the optical center. From a selected model, we can find a practical approach to calculate the parameters of this model.

To calculate parameters of a distortion model, we have to determine the coordinates of reference points in the distorted space and their positions in the undistorted space, correspondingly. Reference points can be extracted using an image of a calibration object giving a line or dot-pattern image (Fig. 5), which is distorted. Using conditions that lines of these points must be straight, equidistant, parallel, or perpendicular we can estimate the locations of these reference-points in the undistorted space with high-accuracy.

Discorpy is the Python implementation of radial distortion correction methods presented in [C1]. These methods employ polynomial models and use a calibration image for calculating coefficients of the models where the optical center is determined independently. The reason of using these models and a calibration image is to achieve sub-pixel accuracy as strictly required by parallel-beam tomography systems. The methods were developed and used internally at the beamline I12, Diamond Light Source-UK, as Mathematica codes. In 2018, they were converted to Python codes and packaged as open-source software [R4] under the name Vounwarp. The name was changed to Discorpy in 2021. From version 1.4, methods for correcting perspective distortion [R3] and extracting reference

points from a line-pattern image were added to the software. A key feature of methods developed and implemented in Discorpy is that radial distortion, center-of-distortion (optical center), and perspective distortion are determined independently using a single calibration image. The following sections explain methods implemented in Discorpy.

Extracting reference-points from a calibration image

The purpose of a calibration-image (Fig. 7 (a,b,c)) is to provide reference-points (Fig. 7 (d)) which can be extracted from the image using some image processing techniques. As shown in Fig. 7, there are a few calibration-images can be used in practice. A dot-pattern image (Fig. 7 (a)) is the easiest one to process because we just need to segment the dots and calculate the center-of-mass of each dot. For a line-pattern image (Fig. 7 (b)), a line-detection technique is needed. Points on the detected lines or the crossing points between these lines can be used as reference-points. For a chessboard image (Fig. 7 (c)), one can employ some corner-detection techniques or apply a gradient filter to the image and use a line-detection technique.

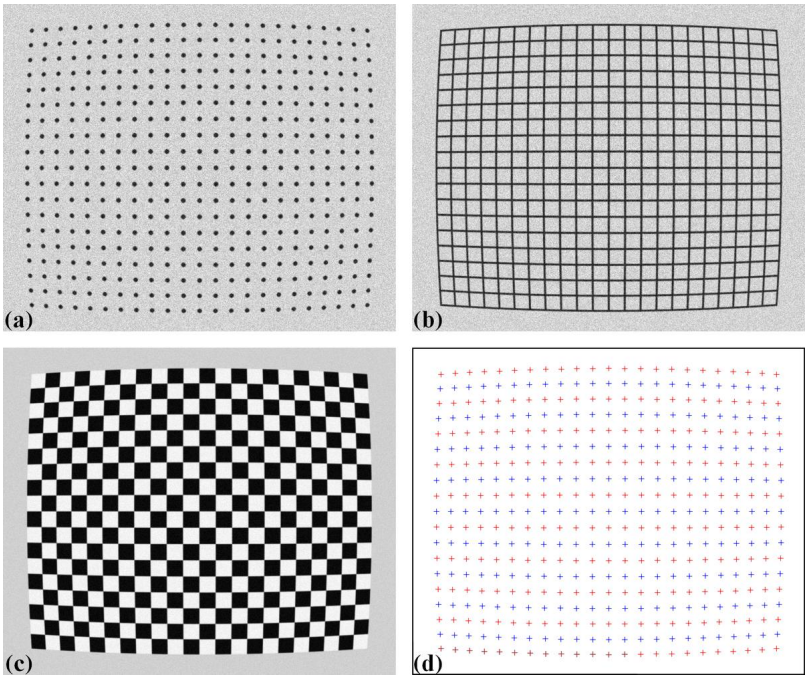


Fig. 7: (a) Dot-pattern image. (b) Line-pattern image. (c) Chessboard image. (d) Extracted reference-points from the image (a),(b), and (c).

In practice, acquired calibration images do not always look nice as shown in Fig. 7. Some are very challenging to get reference-points. The following sub-sections present practical approaches to process calibration images in such cases:

Pre-processing techniques for a dot-pattern image

Binarization

Dots are extracted from a calibration image by binarization (Fig. 8). Then the center-of-mass of each segmented dot is calculated and used as a reference-point.

Normalizing background

The binarization uses a global thresholding method. In some cases, the background of an image is non-uniform which affects the performance of the thresholding method. *Discorpy* provides two ways of normalizing the background of an image: using a strong low-pass filter (Fig. 9) or using a median filter with a large-size window.

Removing non-dot objects

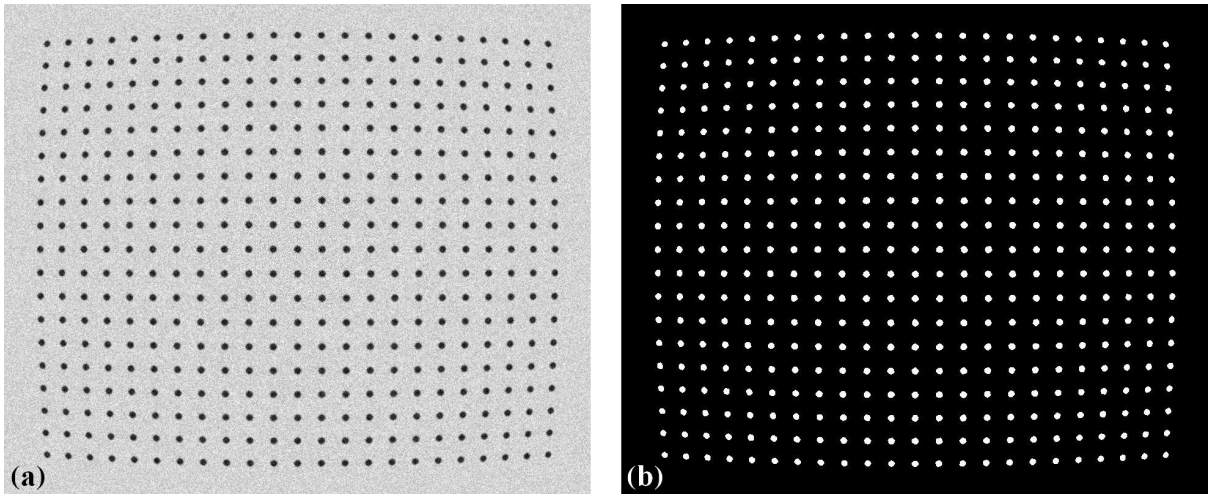


Fig. 8: Demonstration of the image binarization. (a) Dot-pattern image. (b) Segmented dots.

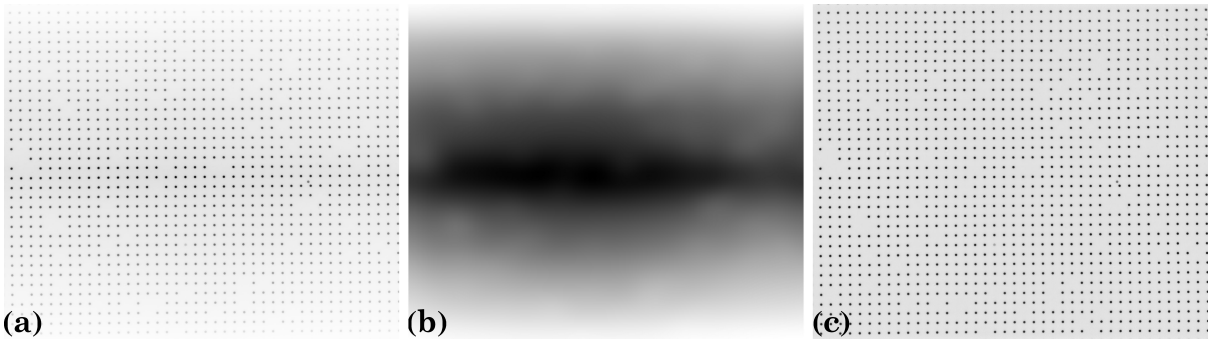


Fig. 9: Demonstration of background normalization. (a) Dot-pattern image (X-ray target). (b) Extracted background. (c) Corrected image.

Discorpy provides two methods for removing non-dot objects after a binarization step. In the first approach, the median size of dots (MS) is determined, then only objects with sizes in the range of (MS–R*MS; MS+R*MS) are kept where R (ratio) is a parameter. In the second approach, the ratio between the largest axis and the smallest axis of the best-fit ellipse is used. Objects with the ratios out of the range of (1.0; 1.0+R) are removed.

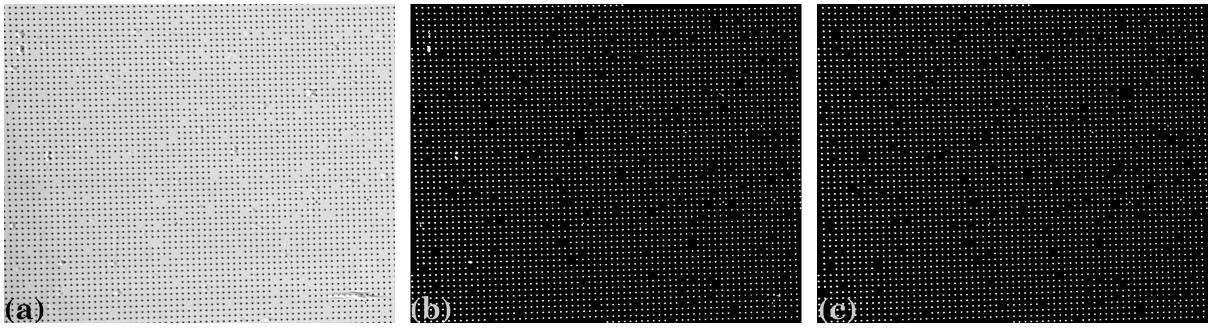


Fig. 10: Demonstration of removing non-dot objects.(a) Dot-pattern image (X-ray target). (b) Binary image. (c) Image with non-dot objects removed.

Removing misplaced dots

Custom-made dot-patterns may have dots placed in wrong positions as shown in Fig. 11. In *Discorpy*, a misplaced dot is identified by using its distances to four nearest dots. If none of the distances is in the range of (MD–R*MD; MD+R*MD), where MD is the median distance of two nearest dots and R is a parameter, the dot is removed. This method, however, should not be used for an image with strong distortion where the distance of two nearest dots changes significantly against their distances from the optical center. A more generic approach to tackle the problem is shown in section 2.2.3.

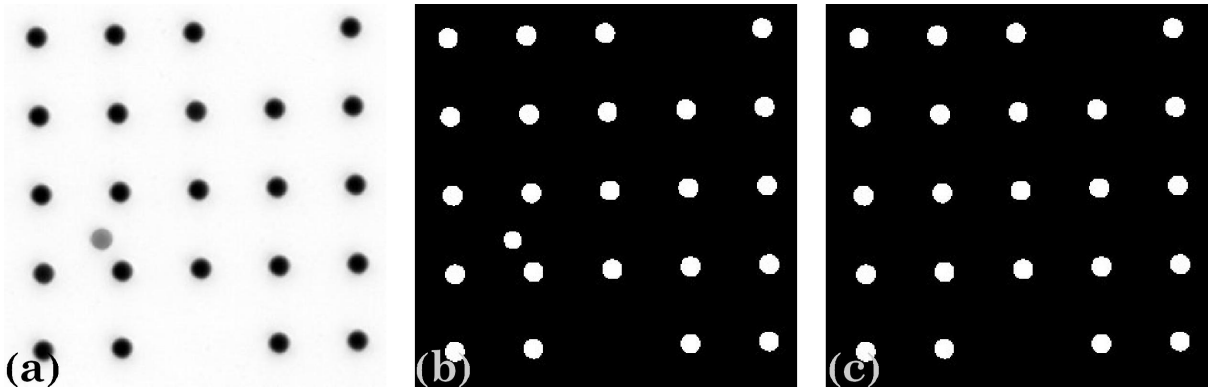


Fig. 11: (a) Image with a misplaced dot. (b) Binary image. (c) Image with misplaced dot removed.

Pre-processing techniques for a line-pattern image

As lines in a calibration image are curved under radial distortion, using well-known line-detection methods such as Hough transform-based approaches are not always feasible. A simple method for detecting curved lines has been developed for Discorpy 1.4 where points on each line are detected by locating local extrema points on an intensity-profile generated by plotting across the image (Fig. 12). Many intensity-profiles are generated to detect points on lines at different locations, then they are grouped line-by-line (Fig. 13).

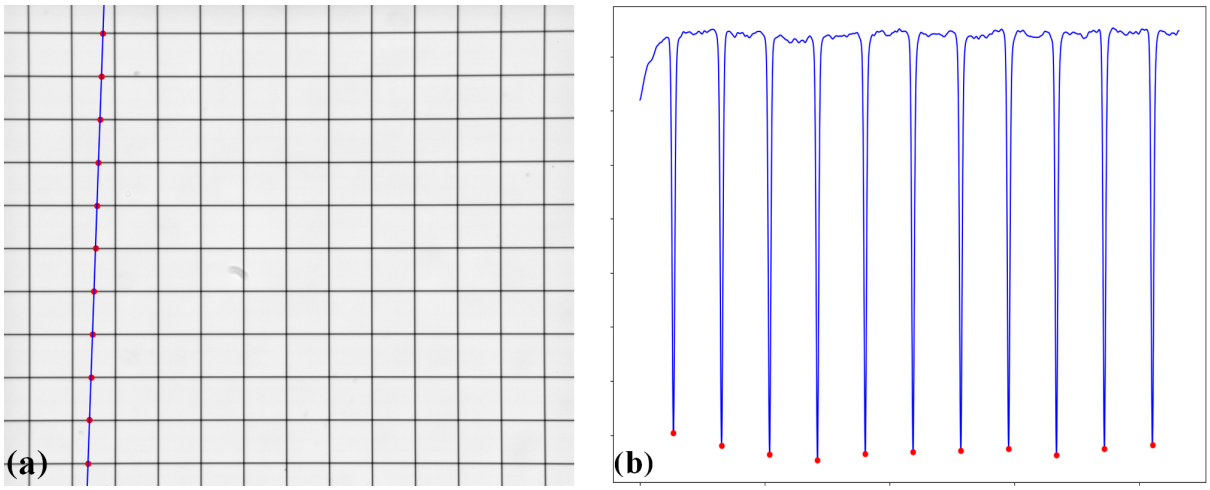


Fig. 12: Process of locating points belong to a line. (a) Intensity-profile extracted along the blue line. (b) Local minimum points located on this profile.

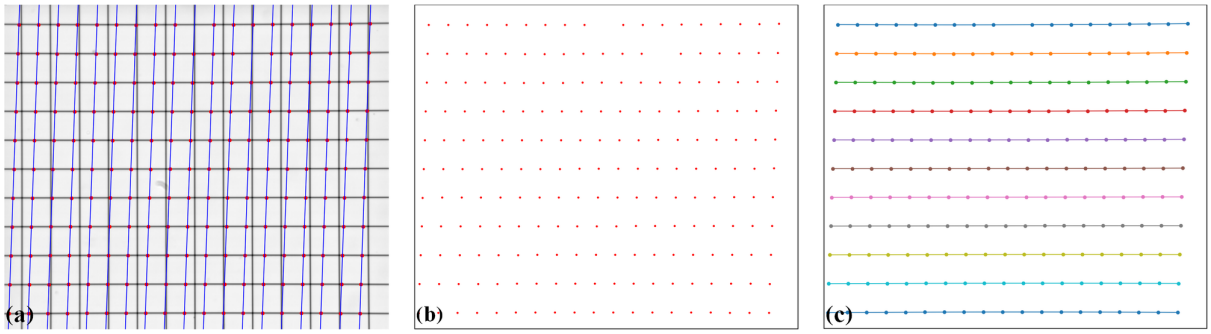


Fig. 13: Full routine of detecting lines. (a) Multiple crossing-lines are used to locate extrema points. (b) Points extracted. (c) Points after grouped into lines.

Pre-processing techniques for a chessboard image

Chessboard images are often used to calibrate commercial cameras rather than scientific cameras. To get reference-points, users can apply some corner-detection methods available in Scikit-image (Fig. 14 (a)(b)), or convert the image to a line-pattern image (Fig. 1(c)) and use the similar routine as described in *section 2.2.2.2*.

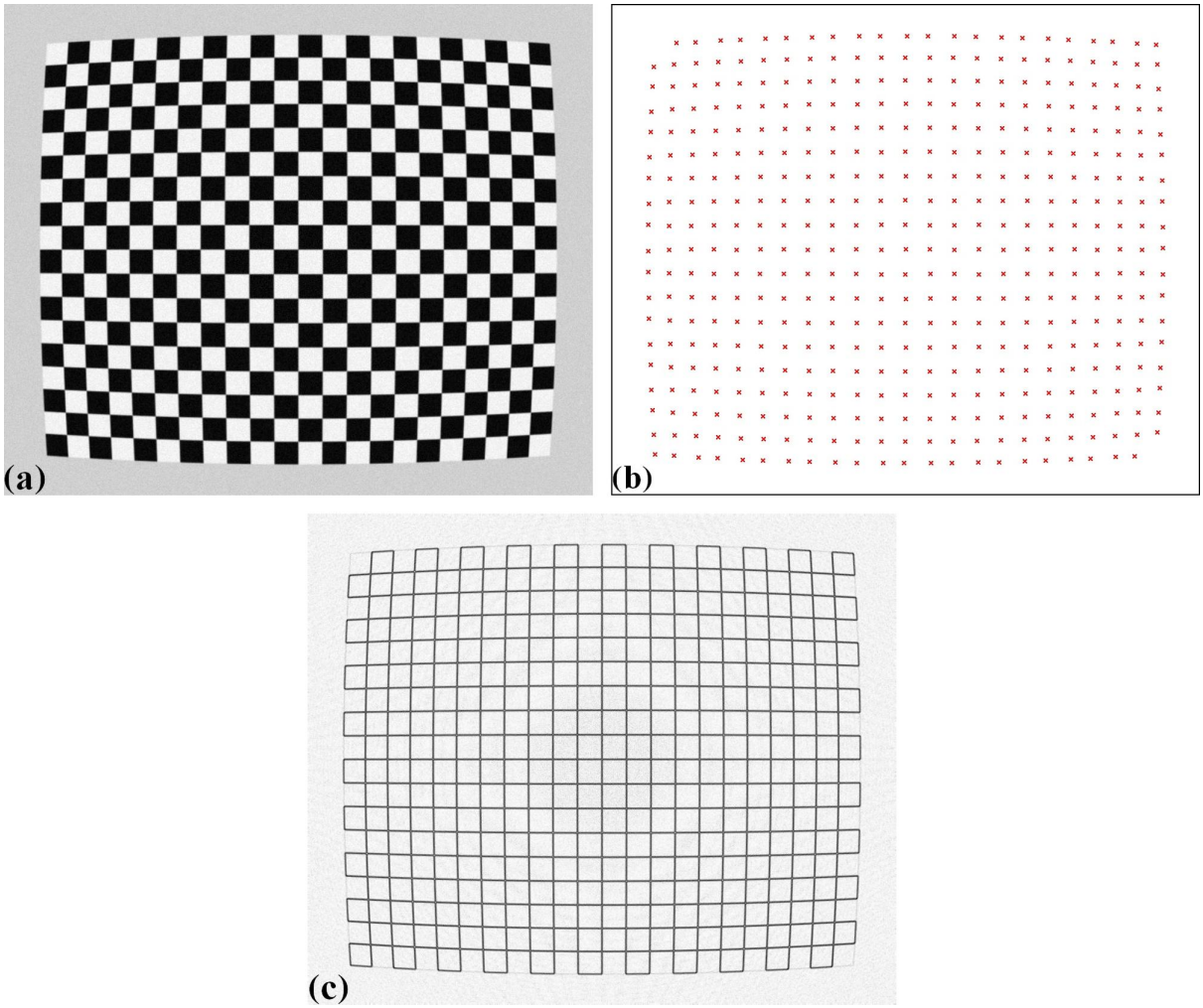


Fig. 14: (a) Chessboard image. (b) Points extracted using a corner-detection method. (c) Converted image from image (a).

Grouping reference-points into horizontal lines and vertical lines

Different techniques of calculating parameters of a distortion-model use reference-points differently. The techniques [C1, R5] implemented in Discorpy group reference-points into horizontal lines and vertical lines (Fig. 15), represent them by the coefficients of parabolic fits, and use these coefficients for calculating distortion-parameters.

The grouping step is critical in data processing workflow. It dictates the performance of other methods down the line. In Discorpy, the grouping method works by searching the neighbours of a point to decide if they belong to the same group or not. The search window is defined by the distance between two nearest reference-points, the slope of the grid, the parameter R, and the acceptable number of missing points. Depending on the quality of a calibration image, users may need to tweak parameters of pre-processing methods and/or the grouping method to get the best results (Fig. 16).

The coordinates of points on each group are fitted to parabolas in which horizontal lines are represented by

$$y = a_i x^2 + b_i x + c_i \tag{1}$$

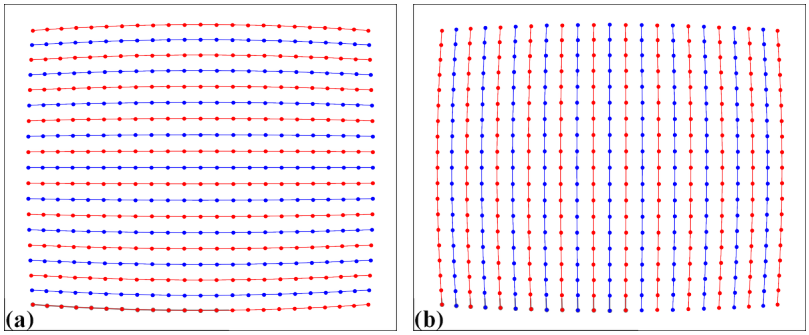


Fig. 15: (a) Points are grouped into horizontal lines. (b) Points are grouped into vertical lines.

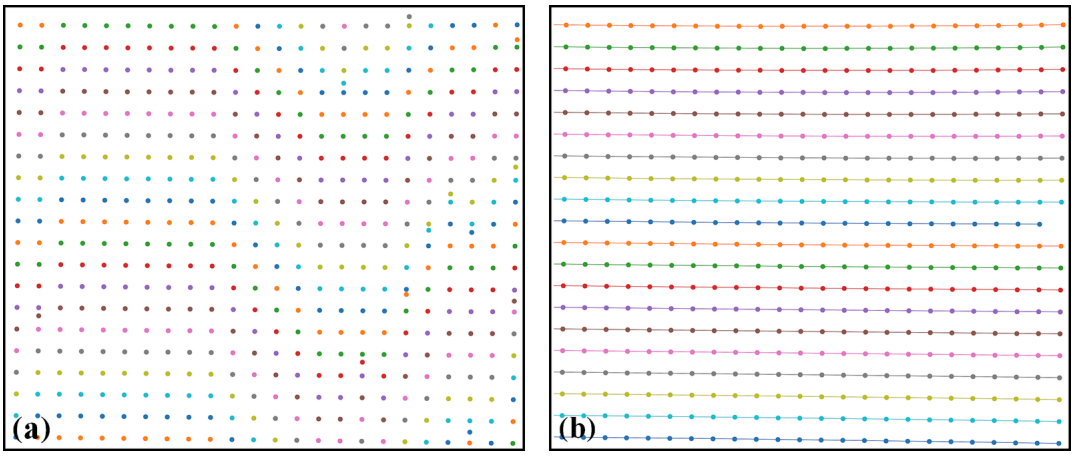


Fig. 16: (a) Points extracted from a calibration image including unwanted points. (b) Results of applying the grouping method to points in (a).

and vertical lines by

$$x = a_j y^2 + b_j y + c_j \tag{2}$$

where i, j are the index of the horizontal lines and vertical lines respectively.

Calculating the optical center of radial distortion

The coarse estimate of the center of distortion (COD) is explained in Fig. 17 where (x_0, y_0) is the average of the axis intercepts c of two parabolas between which the coefficient a changes sign. The slopes of the red and green line are the average of the b coefficients of these parabolas.

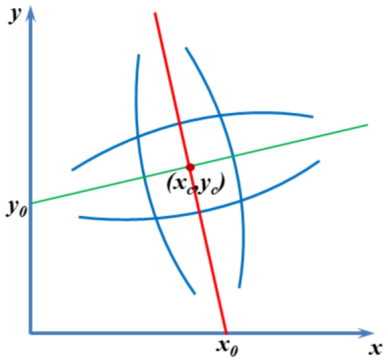


Fig. 17: Intersection between the red and the green line is the CoD.

For calculating the COD with high accuracy, Discorpy implements two methods. One approach is described in details in [R5] where the linear fit is applied to a list of (a, c) coefficients in each direction to find x-center and y-center of the distortion. Another approach, which is slower but more accurate, is shown in [C1]. The technique varies the COD around the coarse-estimated COD and calculate a corresponding metric (Fig. 18). The best COD is the one having the minimum metric. This approach, however, is sensitive to perspective distortion. In practice, it is found that the coarse COD is accurate enough.

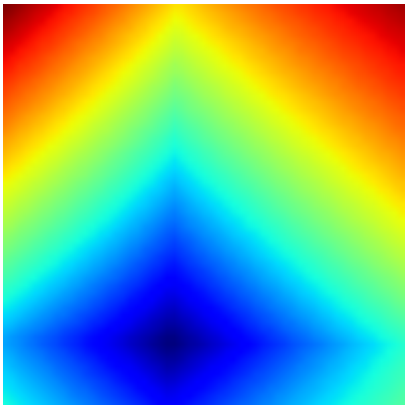


Fig. 18: Metric map of the CoD search.

Correcting perspective effect

In practice, a target sample may not be mounted in parallel to *a sensor-plane*, particularly for a high-resolution detector. This causes perspective distortion in the acquired image which affects the accuracy of a calculated model for radial distortion. Perspective distortion can be detected by making use of parabolic coefficients of lines [R5] where the origin of the coordinate system is shifted to the COD, calculated by the approach in [R5], before the parabola fitting. Fig. 19 (a) shows the plot of a-coefficients against c-coefficients for horizontal lines (Eq. (1)) and vertical lines (Eq. (2)). If there is perspective distortion, the slopes of straight lines fitted to the plotted data are different. The other consequence is that b-coefficients vary against c-coefficients instead of staying the same (Fig. 19 (b)). For comparison, corresponding plots of parabolic coefficients for the case of no perspective-distortion are shown in Fig. 20.

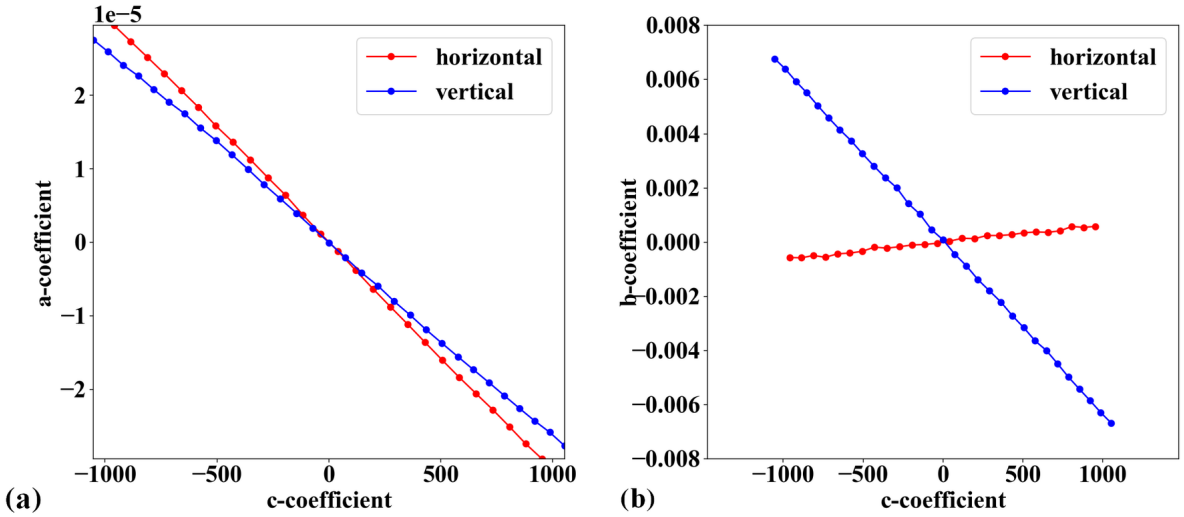


Fig. 19: Effects of perspective distortion to parabolic coefficients. (a) Between a and c-coefficients. (b) Between b and c-coefficients.

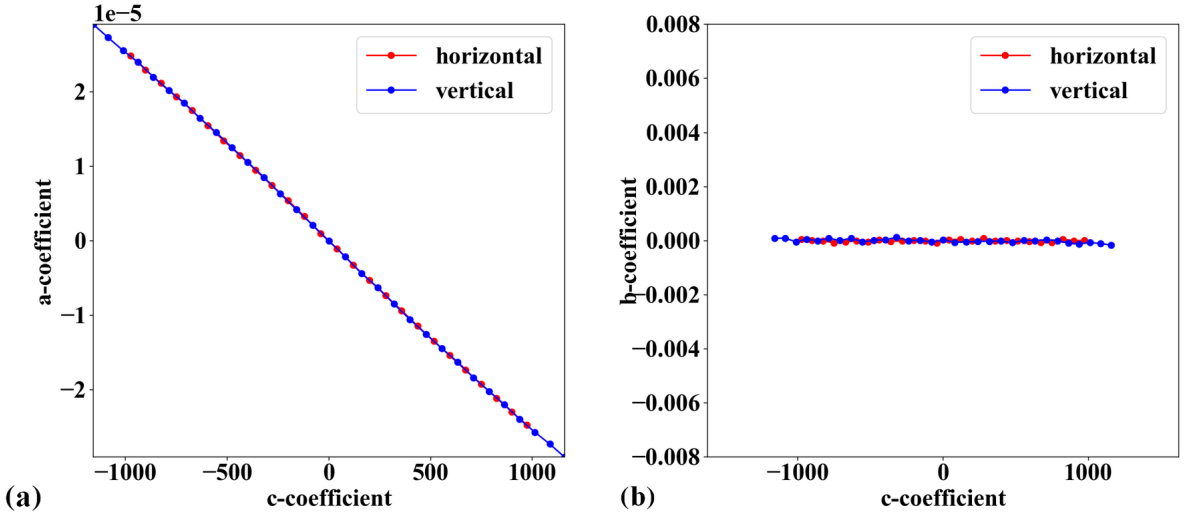


Fig. 20: (a) Corresponding to Fig. 19 (a) without perspective distortion. (b) Corresponding to Fig. 19 (b) without perspective distortion.

In Discorpy 1.4, a method for correcting perspective effect has been developed and added to the list of functionalities. This is a novel feature and has not yet been published in a journal. The method works by correcting coefficients of parabolas using the fact that the resulting coefficients have to satisfy the conditions as shown in Fig. 20. From the corrected coefficients, a grid of points are regenerated by finding cross points between parabolas. Details of the method can be found here.

Calculating coefficients of a polynomial model for radial-distortion correction

For sub-pixel accuracy, the models chosen in [C1] are as follows; for the forward mapping:

$$\frac{r_u}{r_d} = \frac{x_u}{x_d} = \frac{y_u}{y_d} = k_0^f + k_1^f r_d + k_2^f r_d^2 + k_3^f r_d^3 + .. + k_n^f r_d^n \equiv F(r_d) \tag{3}$$

for the backward mapping:

$$\frac{r_d}{r_u} = \frac{x_d}{x_u} = \frac{y_d}{y_u} = k_0^b + k_1^b r_u + k_2^b r_u^2 + k_3^b r_u^3 + .. + k_n^b r_u^n \equiv B(r_u) \tag{3}$$

(x_u, y_u) are the coordinate of a point in the undistorted space and r_u is its distance from the COD. (x_d, y_d, r_d) are for a point in the distorted space. The subscript d is used for clarification. It can be omitted as in Eq. (1) and (2).

To calculate coefficients of two models, we need to determine the coordinates of reference-points in both the distorted-space and in the undistorted-space, correspondingly; and solve a system of linear equations. In [C1] this task is simplified by finding the intercepts of undistorted lines, (c_i^u, c_j^u), instead. A system of linear equations for finding coefficients of the forward mapping is derived as

$$\begin{pmatrix} \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & r_d & r_d^2 & \cdots & r_d^n \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & r_d & r_d^2 & \cdots & r_d^n \\ \vdots & \vdots & \ddots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} k_0^f \\ k_1^f \\ k_2^f \\ \vdots \\ k_n^f \end{pmatrix} = \begin{pmatrix} \vdots \\ c_i^u/(a_i x_d^2 + c_i) \\ \vdots \\ c_j^u/(a_j y_d^2 + c_j) \\ \vdots \end{pmatrix} \tag{3}$$

where each reference-point provides two equations: one associated with a horizontal line (Eq. (1)) and one with a vertical line (Eq. (2)). For the backward mapping, the equation system is

$$\begin{pmatrix} \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & r_d/F_i & r_d^2/F_i^2 & \cdots & r_d^n/F_i^n \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & r_d/F_j & r_d^2/F_j^2 & \cdots & r_d^n/F_j^n \\ \vdots & \vdots & \ddots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} k_0^b \\ k_1^b \\ k_2^b \\ \vdots \\ k_n^b \end{pmatrix} = \begin{pmatrix} \vdots \\ F_i \\ \vdots \\ F_j \\ \vdots \end{pmatrix} \tag{3}$$

where $F_i = (a_i x_d^2 + c_i)/c_i^u$ and $F_j = (a_j y_d^2 + c_j)/c_j^u$. In practice, using distortion coefficients up to the fifth order is accurate enough, as there is no significant gain in accuracy with higher order. As can be seen, the number of linear equations, given by the number of reference-points, is much higher than the number of coefficients. This is crucial to achieve high accuracy in radial-distortion correction. Because the strength of distortion varies across an image, providing many reference-points with high-density improves the robustness of a calculated model.

To solve these above equations we need to determine c_i^u and c_j^u. Using the assumption that lines are equidistant, c_i^u and c_j^u are calculated by extrapolating from a few lines around the COD as

$$c_i^u = sgn(c_i) \times |(i - i_0)\overline{\Delta c}| + c_{i_0} \tag{3}$$

and

$$c_j^u = sgn(c_j) \times |(j - j_0)\overline{\Delta c}| + c_{j_0} \tag{4}$$

where the $sgn()$ function returns the value of -1, 0, or 1 corresponding to its input of negative, zero, or positive value. i_0 is the index of the line closest to the COD. $\overline{\Delta c}$ is the average of the difference of c_i near the COD. $\overline{\Delta c}$ can be refined further by varying it around an initial guess and find the minimum of $\sum_i (c_i - c_i^u)^2$, which also is provided in the package.

Sometime we need to calculate coefficients of a backward model given that coefficients of the corresponding forward-model are known, or vice versa. This is straightforward as one can generate a list of reference-points and calculate their positions in the opposite space using the known model. From the data-points of two spaces and using Eq. (3) or Eq. (3) directly, a system of linear equations can be formulated and solved to find the coefficients of the opposite model. This functionality is available in Discorpy.

Calculating coefficients of a correction model for perspective distortion

The forward mapping between a distorted point and an undistorted point are given by [R3]

$$x_u = \frac{k_1^f x_d + k_2^f y_d + k_3^f}{k_7^f x_d + k_8^f y_d + 1} \tag{5}$$

$$y_u = \frac{k_4^f x_d + k_5^f y_d + k_6^f}{k_7^f x_d + k_8^f y_d + 1} \tag{6}$$

These equations can be rewritten as

$$x_u = k_1^f x_d + k_2^f y_d + k_3^f + 0 \times k_4^f + 0 \times k_5^f + 0 \times k_6^f - k_7^f x_d x_u - k_8^f y_d x_u \tag{7}$$

$$y_u = 0 \times k_1^f + 0 \times k_2^f + 0 \times k_3^f + k_4^f x_d + k_5^f y_d + k_6^f - k_7^f x_d y_u - k_8^f y_d y_u \tag{7}$$

which can be formulated as a system of linear equations for n couple-of-points (1 distorted point and its corresponding point in the undistorted space).

$$\begin{pmatrix} x_{d1} & y_{d1} & 1 & 0 & 0 & 0 & -x_{d1}x_{u1} & -y_{d1}x_{u1} \\ 0 & 0 & 0 & x_{d1} & y_{d1} & 1 & -x_{d1}y_{u1} & -y_{d1}y_{u1} \\ x_{d2} & y_{d2} & 1 & 0 & 0 & 0 & -x_{d2}x_{u2} & -y_{d2}x_{u2} \\ 0 & 0 & 0 & x_{d2} & y_{d2} & 1 & -x_{d2}y_{u2} & -y_{d2}y_{u2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{dn} & y_{dn} & 1 & 0 & 0 & 0 & -x_{dn}x_{un} & -y_{dn}x_{un} \\ 0 & 0 & 0 & x_{dn} & y_{dn} & 1 & -x_{dn}y_{un} & -y_{dn}y_{un} \end{pmatrix} \begin{pmatrix} k_1^f \\ k_2^f \\ k_3^f \\ k_4^f \\ k_5^f \\ k_6^f \\ k_7^f \\ k_8^f \end{pmatrix} = \begin{pmatrix} x_{u1} \\ y_{u1} \\ x_{u2} \\ y_{u2} \\ \vdots \\ x_{un} \\ y_{un} \end{pmatrix} \tag{7}$$

For the backward mapping, the coordinates of corresponding points in Eq. (9-13) are simply swapped which results in

$$x_d = \frac{k_1^b x_u + k_2^b y_u + k_3^b}{k_7^b x_u + k_8^b y_u + 1} \tag{7}$$

$$y_d = \frac{k_4^b x_u + k_5^b y_u + k_6^b}{k_7^b x_u + k_8^b y_u + 1} \tag{8}$$

$$\begin{pmatrix} x_{u1} & y_{u1} & 1 & 0 & 0 & 0 & -x_{u1}x_{d1} & -y_{u1}x_{d1} \\ 0 & 0 & 0 & x_{u1} & y_{u1} & 1 & -x_{u1}y_{d1} & -y_{u1}y_{d1} \\ x_{u2} & y_{u2} & 1 & 0 & 0 & 0 & -x_{u2}x_{d2} & -y_{u2}x_{d2} \\ 0 & 0 & 0 & x_{u2} & y_{u2} & 1 & -x_{u2}y_{d2} & -y_{u2}y_{d2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{un} & y_{un} & 1 & 0 & 0 & 0 & -x_{un}x_{dn} & -y_{un}x_{dn} \\ 0 & 0 & 0 & x_{un} & y_{un} & 1 & -x_{un}y_{dn} & -y_{un}y_{dn} \end{pmatrix} \begin{pmatrix} k_1^b \\ k_2^b \\ k_3^b \\ k_4^b \\ k_5^b \\ k_6^b \\ k_7^b \\ k_8^b \end{pmatrix} = \begin{pmatrix} x_{d1} \\ y_{d1} \\ x_{d2} \\ y_{d2} \\ \vdots \\ x_{dn} \\ y_{dn} \end{pmatrix} \tag{9}$$

To find 8 coefficients in Eq. (7) or Eq. (9), the coordinates of at least 4 couple-of-points are needed where 1 couple-of-points provides 2 equations. If there are more than 4 couple-of-points, a least square method is used to solve the equation. Given the coordinates of distorted points on grid lines, using conditions that lines connecting these points must be parallel, equidistant, or perpendicular we can calculate the coordinates of undistorted points (Fig. 21) correspondingly. Details of this implementation can be found in Discorpy's API.

Correcting a distorted image

To correct distorted images, backward models are used because values of pixels adjacent to a mapped point are known (Fig. 22). This makes it easy to perform interpolation.

For radial distortion; given (x_u, y_u) , (x_{COD}, y_{COD}), and $(k_0^b, k_1^b, ..., k_n^b)$ of a backward model; the correction routine is as follows:

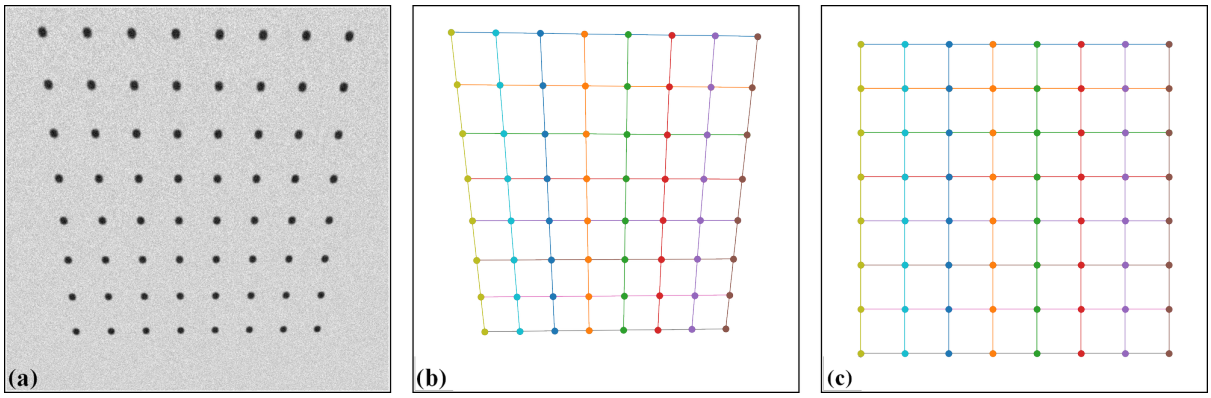


Fig. 21: Demonstration of generating undistorted points from perspective points. (a) Calibration image. (b) Extracted reference-points. (c) Undistorted points generated by using the conditions that lines are parallel in each direction, perpendicular between direction, and equidistant. As the scale between the distorted space and undistorted space are unknown, the distance between lines in the undistorted space can be arbitrarily chosen. Here the mean of distances in the distorted space is used.

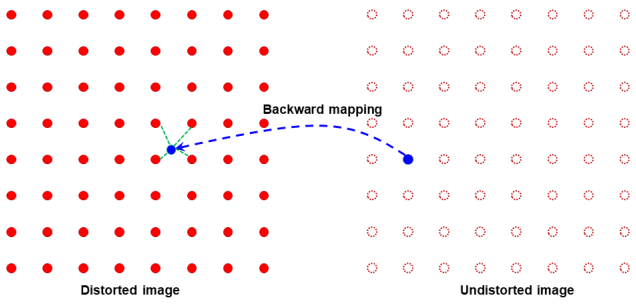


Fig. 22: Demonstration of the backward mapping.

- -> Translate the coordinates: $x_u = x_u - x_{COD}$; $y_u = y_u - y_{COD}$.
- -> Calculate: $r_u = \sqrt{x_u^2 + y_u^2}$; $r_d = r_u(k_0^b + k_1^b r_u + k_2^b r_u^2 + ... + k_n^b r_u^n)$.
- -> Calculate: $x_d = x_u r_d / r_u$; $y_d = y_u r_d / r_u$.
- -> Translate the coordinates: $x_d = x_d + x_{COD}$; $y_d = y_d + y_{COD}$.
- -> Find 4 nearest grid points of the distorted image by combing two sets of [floor(x_d), ceil(x_d)] and [floor(y_d), ceil(y_d)]. Clip values out of the range of the grid.
- -> Interpolate the value at (x_d, y_d) using the values of 4 nearest points. Assign the result to the point (x_u, y_u) in the undistorted image.

Correcting perspective distortion is straightforward. Given (x_u, y_u) and coefficients $(k_1^b, k_2^b, ..., k_8^b)$, Eq. (7) (8) are used to calculate x_d, y_d. Then, the image value at this location is calculated by interpolation as explained above.

Summary

The above sections present a complete workflow of calibrating a lens-coupled detector in a concise way. It can be divided into three stages: pre-processing stage is for extracting and grouping reference-points from a calibration image; processing stage is for calculating coefficients of correction models; and post-processing stage is for correcting images. Discorpy's API is structured following this workflow including an input-output module.

As shown above, parameters of correction models for radial distortion and perspective distortion can be determined independently because reference-points in the undistorted space can be generated easily using methods available in Discorpy. Details of how to use Discorpy to process real data are shown in *section 3*.

1.3 Usage

1.3.1 Resources

- A technical report explaining step-by-step on how to calculate distortion coefficients from a dot pattern image is available at Zenodo.
- Examples of how to use the package are in the "/examples" folder at the github page.
- Coefficients determined by the package can be used by other tomographic software such as Tomopy, Savu, or Algotom for correction.

1.3.2 Notes related to Python programming

In the *workflow* of calibrating a distortion target, some functions work with Python lists, some with Numpy arrays. They are different objects and can cause confusion or give rise to errors if users don't use these objects properly. There are many tutorials online to explain the difference between them. Here we only show some examples related to how they may be used with Discorpy.

- Initialization:

```
import numpy as np

# Declare a Python list
points_n1 = [[0.0, 0.0], [0.0, 0.0], [0.0, 0.0]]
# Declare a Numpy array
points_n2 = np.asarray(points_n1, dtype=np.float32)
# or
points_n2 = np.zeros((3,2), dtype=np.float32)
```

- Indexing:

```
# For a Python list
element1 = points_n1[0][1]
# For a Numpy array
element2 = points_n2[0,1]

# For a Python list
sub_set1 = points_n1[0][0:2]
# For a Numpy array
sub_set2 = points_n2[0,0:2]
```

- Math operations:

```
# For a Python list
points_m1 = points_n1 + 1.0 # >> Raise an error
# For a Numpy array
points_m2 = points_n2 + 1.0 # >> Add 1.0 to every element of the array.
```

- Storing:

```
# A Python list can store different objects with different size.
points_n1 =([[[0.0, 1.0], [1.0, 2.0]], np.ones((3,2))])
# A Numpy array can only store Numpy objects with the same size.
points_n2 = np.asarray(([[[0.0, 1.0], [1.0, 2.0]], np.ones((2,2))]))
```

In Discorpy, Python lists are mainly used to store Numpy arrays with different size. For example, in the *grouping* step, the number of extracted reference-points on each line are not the same.

It is important to be aware that different methods may use different coordinate systems. In the image coordinate system, the origin is at the top-left corner of the image. For example, the center-of-mass of an object refers to this origin. In the plotting coordinate system, the origin is at the bottom-left. The parabolic fits of horizontal lines refers to the bottom-left origin, however the one of vertical lines refers to the top-left origin with coordinates swapped. This is necessary to avoid the numerical problem of fitting lines nearly perpendicular to the axis.

1.3.3 Demonstrations

Process a high-quality calibration image

The following workflow shows how to use Discorpy to process a high-quality calibration image acquired at Beamline I12, Diamond Light Source.

- Load the image:

```
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.prep.preprocessing as prep
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

# Initial parameters
file_path = "../../data/dot_pattern_01.jpg"
output_base = "E:/output_demo_01/"
num_coef = 5  # Number of polynomial coefficients
mat0 = io.load_image(file_path) # Load image
(height, width) = mat0.shape
```

- Extract reference-points:

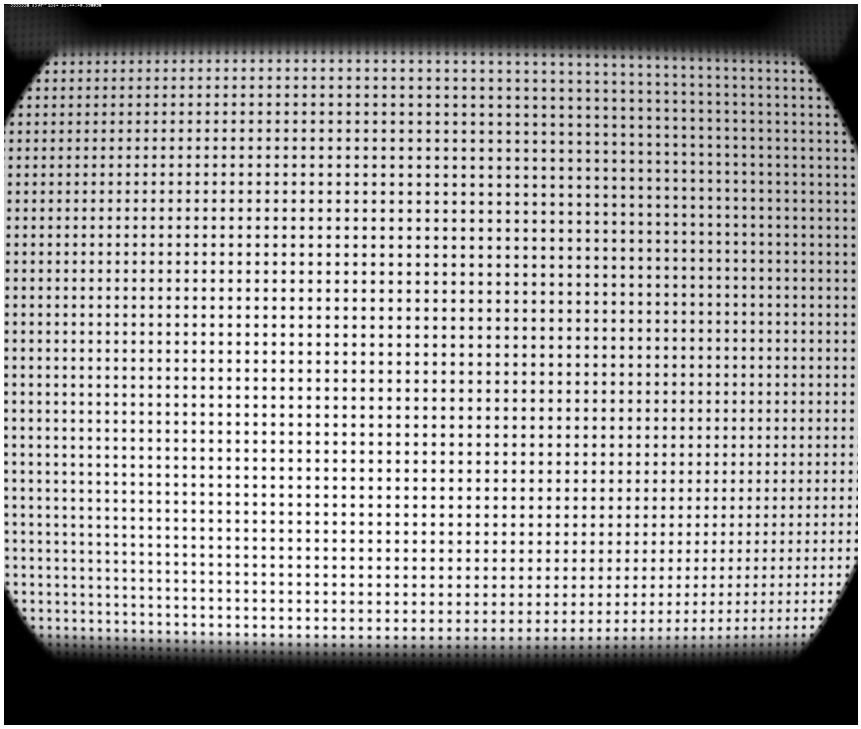


Fig. 23: Visible dot-target.

```
# Segment dots
mat1 = prep.binarization(mat0)
# Calculate the median dot size and distance between them.
(dot_size, dot_dist) = prep.calc_size_distance(mat1)
# Remove non-dot objects
mat1 = prep.select_dots_based_size(mat1, dot_size)
# Remove non-elliptical objects
mat1 = prep.select_dots_based_ratio(mat1)
io.save_image(output_base + "/segmented_dots.jpg", mat1) # Save image␣
↪for checking
# Calculate the slopes of horizontal lines and vertical lines.
hor_slope = prep.calc_hor_slope(mat1)
ver_slope = prep.calc_ver_slope(mat1)
print("Horizontal slope: {0}. Vertical slope {1}".format(hor_slope, ver_
↪slope))

#>> Horizontal slope: 0.011244738004780091. Vertical slope -0.
↪011342266682773354
```

- As can be seen from the highlighted output above, the slopes of horizontal lines and vertical lines are nearly the same (note that their signs are opposite due to the use of different origin-axis). This indicates that perspective distortion is negligible. The next step is to group points into lines. After the grouping step, perspective effect; *if there is*; can be corrected simply by adding a single line of command.

```
# Group points to horizontal lines
list_hor_lines = prep.group_dots_hor_lines(mat1, hor_slope, dot_dist)
# Group points to vertical lines
list_ver_lines = prep.group_dots_ver_lines(mat1, ver_slope, dot_dist)
# Optional: remove horizontal outliers
list_hor_lines = prep.remove_residual_dots_hor(list_hor_lines, hor_slope)
# Optional: remove vertical outliers
```

(continues on next page)

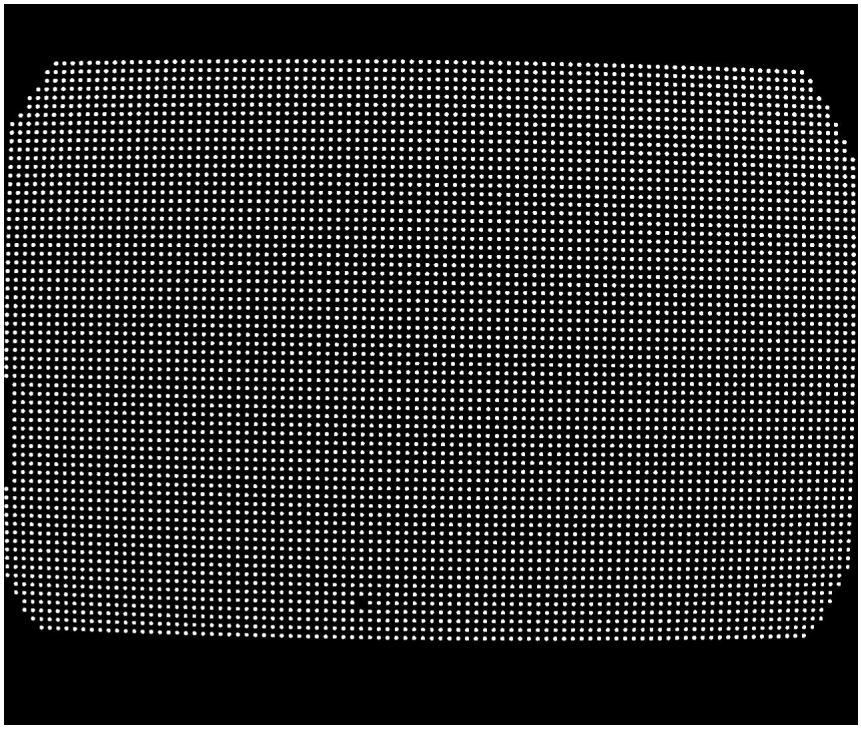


Fig. 24: Segmented dots.

(continued from previous page)

```
list_ver_lines = prep.remove_residual_dots_ver(list_ver_lines, ver_slope)
# Save output for checking
io.save_plot_image(output_base + "/horizontal_lines.png", list_hor_lines,
↪ height, width)
io.save_plot_image(output_base + "/vertical_lines.png", list_ver_lines,␣
↪height, width)

# Optional: correct perspective effect. Only available from Discorpy 1.4
# list_hor_lines, list_ver_lines = proc.regenerate_grid_points_parabola(
#                     list_hor_lines, list_ver_lines, perspective=True)
```

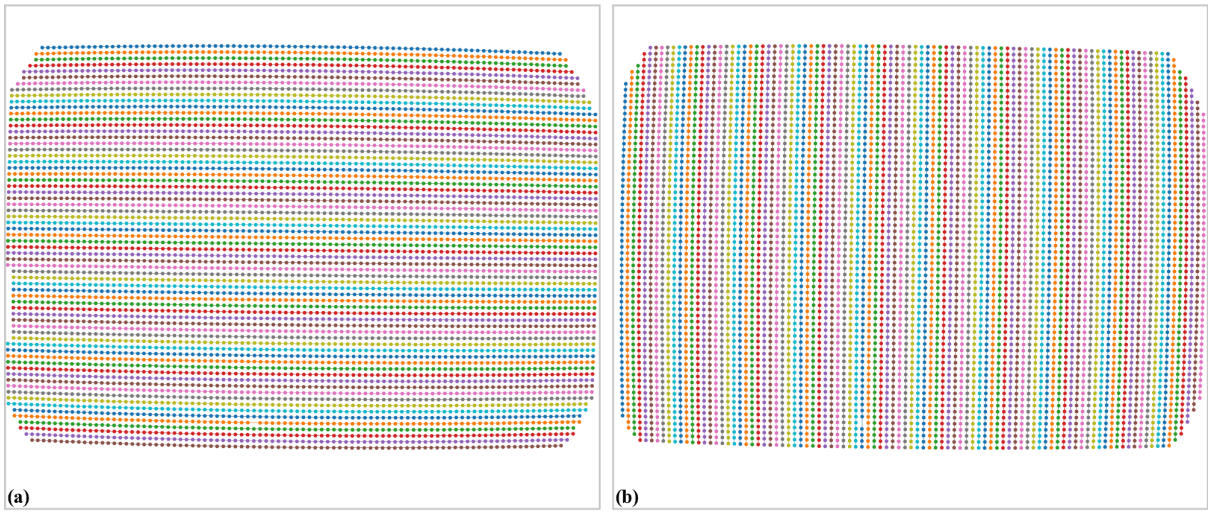


Fig. 25: Reference-points (center-of-mass of dots) after grouped into horizontal lines (a) and vertical lines (b).

- We can check the straightness of lines of points by calculating the distances of the grouped points to their fitted straight lines. This helps to assess if the distortion is significant or not. Note that the origin of the coordinate system before distortion correction is at the top-left corner of an image.

```
list_hor_data = post.calc_residual_hor(list_hor_lines, 0.0, 0.0)
list_ver_data = post.calc_residual_ver(list_ver_lines, 0.0, 0.0)
io.save_residual_plot(output_base + "/hor_residual_before_correction.png
↪",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_before_correction.png
↪",
                      list_ver_data, height, width)
```

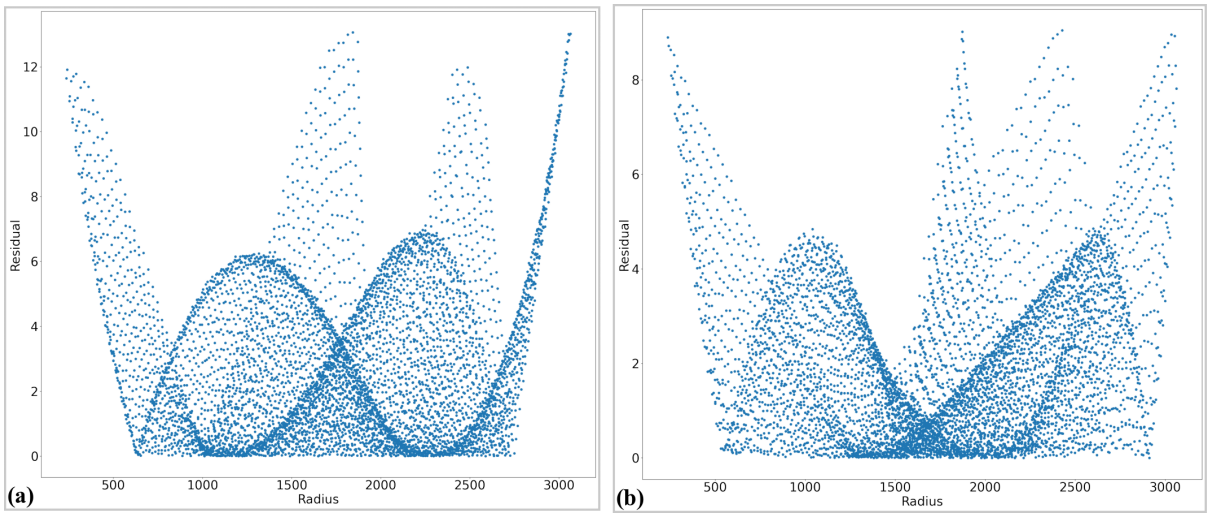


Fig. 26: Plot of the distances of the dot-centroids from their fitted straight line against their distances from the axes origin. (a) For horizontal lines. (b) For vertical lines.

- As shown in Fig. 26, the residual is more than 12 pixels which means that distortion is significant and needs to be corrected. The next step is to calculate the center of distortion (COD) and the coefficients of the backward mapping for *a radial distortion model*.

```
# Calculate the center of distortion
(xcenter, ycenter) = proc.find_cod_coarse(list_hor_lines, list_ver_lines)
# Calculate coefficients of the correction model
list_fact = proc.calc_coef_backward(list_hor_lines, list_ver_lines,
                                    xcenter, ycenter, num_coef)
# Save the results for later use.
io.save_metadata_txt(output_base + "/coefficients_radial_distortion.txt",
                     xcenter, ycenter, list_fact)
print("X-center: {0}. Y-center: {1}".format(xcenter, ycenter))
print("Coefficients: {0}".format(list_fact))
"""
>> X-center: 1252.1528590042283. Y-center: 1008.9088499595639
>> Coefficients: [1.00027631e+00, -1.25730878e-06, -1.43170401e-08,
                  -1.65727563e-12, 7.89109870e-16]
"""
```

- Using the determined parameters of the correction model, we can unwarp the lines of points and check the correction results.

```
# Apply correction to the lines of points
list_uhor_lines = post.unwarp_line_backward(list_hor_lines, xcenter,␣
```

(continues on next page)

(continued from previous page)

```
↪ycenter,
                                                list_fact)
list_uver_lines = post.unwarp_line_backward(list_ver_lines, xcenter,␣
↪ycenter,
                                                list_fact)
# Save the results for checking
io.save_plot_image(output_base + "/unwarpped_horizontal_lines.png", list_
↪uhor_lines,
                   height, width)
io.save_plot_image(output_base + "/unwarpped_vertical_lines.png", list_
↪uver_lines,
                   height, width)
# Calculate the residual of the unwarpped points.
list_hor_data = post.calc_residual_hor(list_uhor_lines, xcenter, ycenter)
list_ver_data = post.calc_residual_ver(list_uver_lines, xcenter, ycenter)
# Save the results for checking
io.save_residual_plot(output_base + "/hor_residual_after_correction.png",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_after_correction.png",
                      list_ver_data, height, width)
```

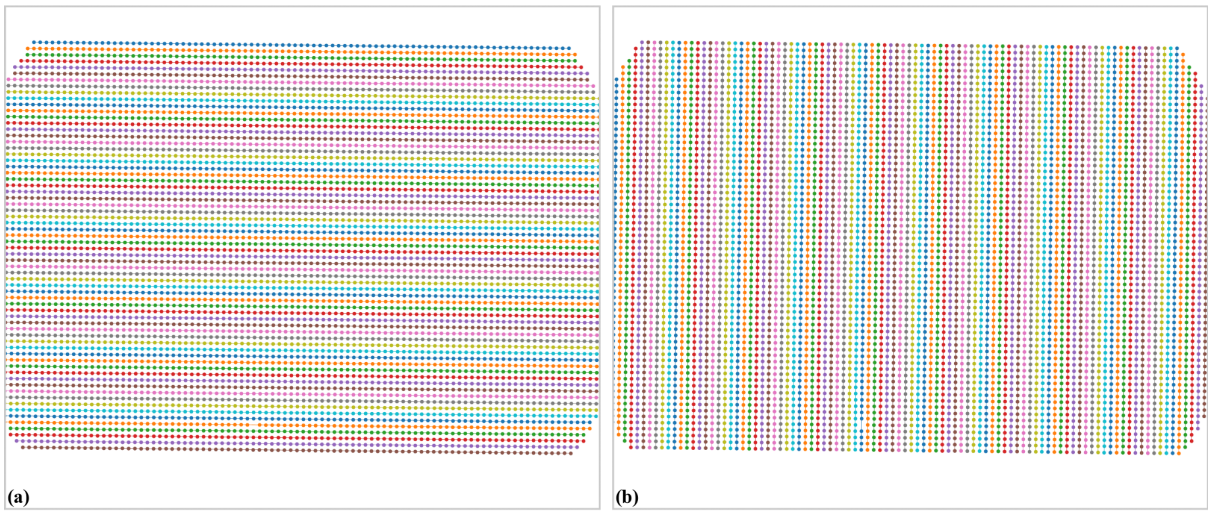


Fig. 27: . (a) Unwarpped horizontal lines. (b) Unwarpped vertical lines.

- As can be seen from Fig. 28 the accuracy of the correction results is sub-pixel. The last step of the workflow is to use the determined model for correcting images.

```
# Load coefficients from previous calculation if need to
# (xcenter, ycenter, list_fact) = io.load_metadata_txt(
#     output_base + "/coefficients_radial_distortion.txt")
# Correct the image
corrected_mat = post.unwarp_image_backward(mat0, xcenter, ycenter, list_
↪fact)
# Save results. Note that the output is 32-bit numpy array. Convert to␣
↪lower-bit if need to.
io.save_image(output_base + "/corrected_image.tif", corrected_mat)
io.save_image(output_base + "/difference.tif", corrected_mat - mat0)
```

Click here to download the Python codes.

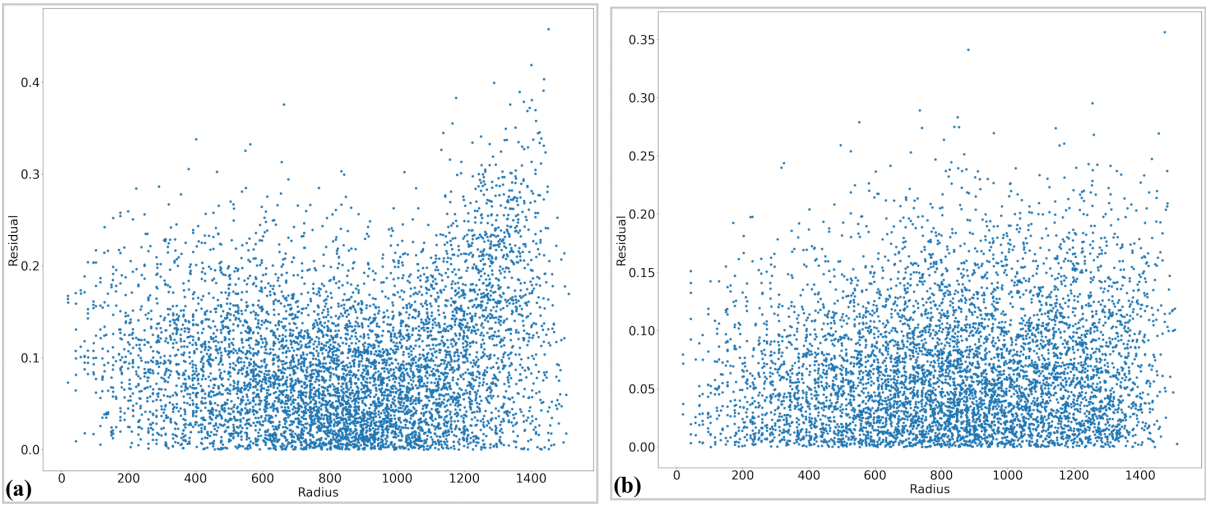


Fig. 28: Residual of the unwarpped points. Note that the origin of the coordinate system is at the center of distortion. (a) For horizontal lines. (b) For vertical lines.

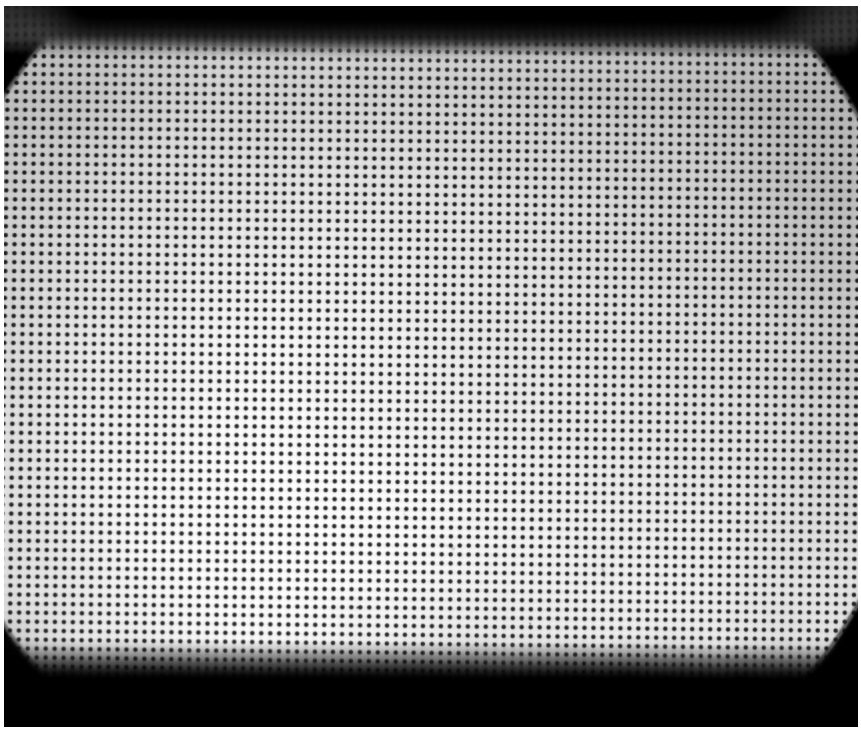


Fig. 29: Corrected image.

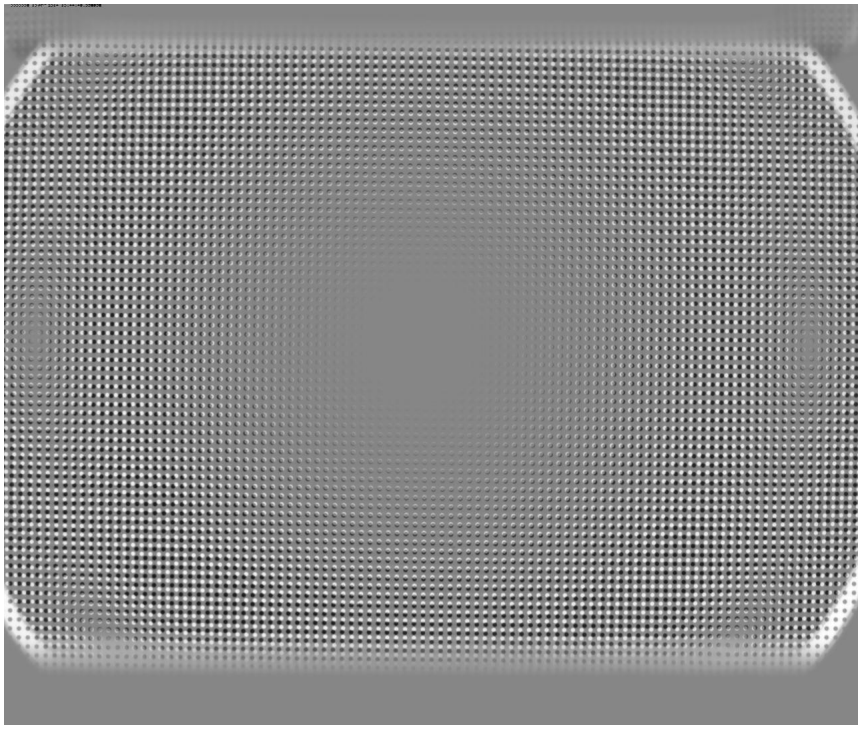


Fig. 30: Difference between images before (Fig. 23) and after (Fig. 29) the correction.

Process an X-ray target image having perspective effect

The following workflow shows how to use Discorpy to process an X-ray target image, acquired at Beamline I13 Diamond Light Source, which has a small perspective effect.

- The following codes load the image, extract reference-points, and group them into lines. Note that the *accepted_ratio* parameter is adjusted to remove lines have a small number of points. Lines with small number of points can affect the parabolic fit leading to a cascading effect to the accuracy of the correction model.

```
import numpy as np
import matplotlib.pyplot as plt
import discorpy.losa.loadersaver as io
import discorpy.prep.preprocessing as prep
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

# Initial parameters
file_path = "../../data/dot_pattern_02.jpg"
output_base = "E:/output_demo_02/"
num_coef = 5  # Number of polynomial coefficients
mat0 = io.load_image(file_path) # Load image
(height, width) = mat0.shape
# Segment dots
mat1 = prep.binarization(mat0)
# Calculate the median dot size and distance between them.
(dot_size, dot_dist) = prep.calc_size_distance(mat1)
# Remove non-dot objects
mat1 = prep.select_dots_based_size(mat1, dot_size)
# Remove non-elliptical objects
mat1 = prep.select_dots_based_ratio(mat1)
io.save_image(output_base + "/segmented_dots.jpg", mat1)
# Calculate the slopes of horizontal lines and vertical lines.
```

(continues on next page)

(continued from previous page)

```
hor_slope = prep.calc_hor_slope(mat1)
ver_slope = prep.calc_ver_slope(mat1)
# Group points into lines
list_hor_lines = prep.group_dots_hor_lines(mat1, hor_slope, dot_dist,␣
↪accepted_ratio=0.8)
list_ver_lines = prep.group_dots_ver_lines(mat1, ver_slope, dot_dist,␣
↪accepted_ratio=0.8)
# Optional: remove outliers
list_hor_lines = prep.remove_residual_dots_hor(list_hor_lines, hor_slope)
list_ver_lines = prep.remove_residual_dots_ver(list_ver_lines, ver_slope)
# Save output for checking
io.save_plot_image(output_base + "/horizontal_lines.png", list_hor_lines,
↪ height, width)
io.save_plot_image(output_base + "/vertical_lines.png", list_ver_lines,␣
↪height, width)
list_hor_data = post.calc_residual_hor(list_hor_lines, 0.0, 0.0)
list_ver_data = post.calc_residual_ver(list_ver_lines, 0.0, 0.0)
io.save_residual_plot(output_base + "/hor_residual_before_correction.png
↪",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_before_correction.png
↪",
                      list_ver_data, height, width)

print("Horizontal slope: {0}. Vertical slope: {1}".format(hor_slope, ver_
↪slope))
#>> Horizontal slope: -0.03194770332102831. Vertical slope: 0.
↪03625649318792672
```

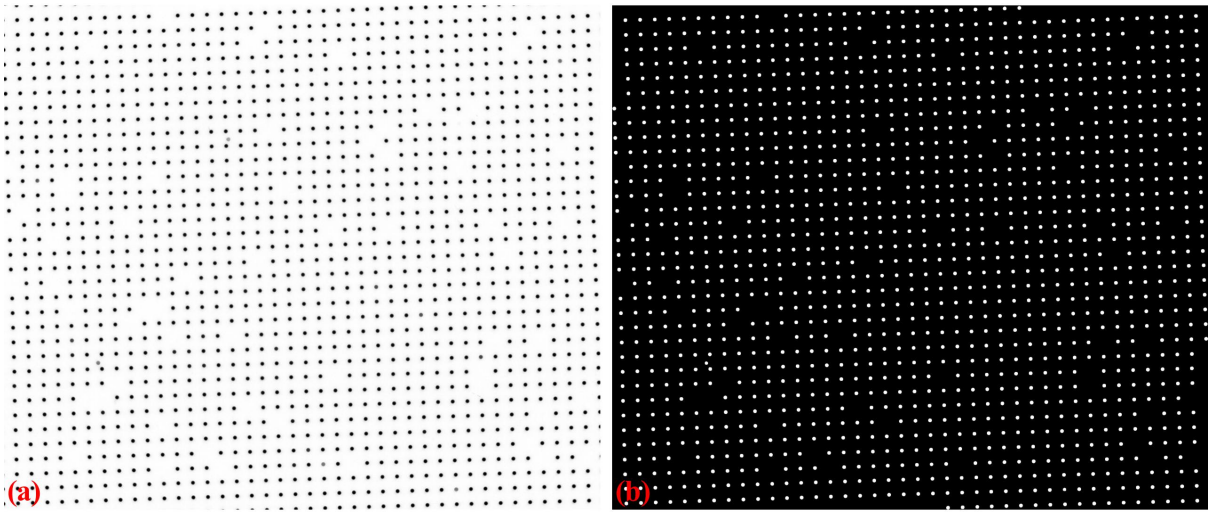


Fig. 31: . (a) X-ray target image. (b) Segmented image.

- As can be seen from the highlighted output above, the slopes of horizontal lines and vertical lines are quite different, compared to the results in *demo 1*. This indicates that there is a perspective effect. We can confirm that by plotting coefficients of parabolic fits of lines. Note that the signs of b-coefficients of parabolic fits for horizontal lines are reversed because they use a different *axes origin* compared to vertical lines.

```
# Calculate the center of distortion, but just for parabolic fit.
(xcen_tmp, ycen_tmp) = proc.find_cod_bailey(list_hor_lines, list_ver_
↪lines)
```

(continues on next page)

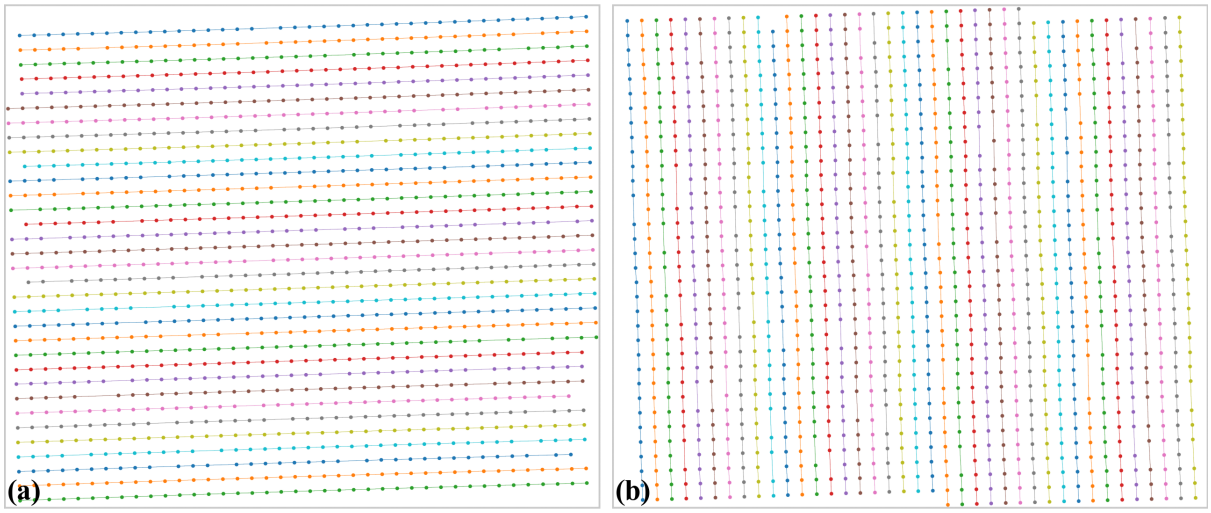


Fig. 32: . (a) Horizontal lines. (b) Vertical lines.

(continued from previous page)

```
# Apply the parabolic fit to lines
list_hor_coef = proc._para_fit_hor(list_hor_lines, xcen_tmp, ycen_tmp)[0]
list_ver_coef = proc._para_fit_ver(list_ver_lines, xcen_tmp, ycen_tmp)[0]
# Plot the results
plt.figure(0)
plt.plot(list_hor_coef[:, 2], list_hor_coef[:, 0], "-o")
plt.plot(list_ver_coef[:, 2], list_ver_coef[:, 0], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("a-coefficient")

plt.figure(1)
plt.plot(list_hor_coef[:, 2], -list_hor_coef[:, 1], "-o")
plt.plot(list_ver_coef[:, 2], list_ver_coef[:, 1], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("b-coefficient")
plt.show()
```

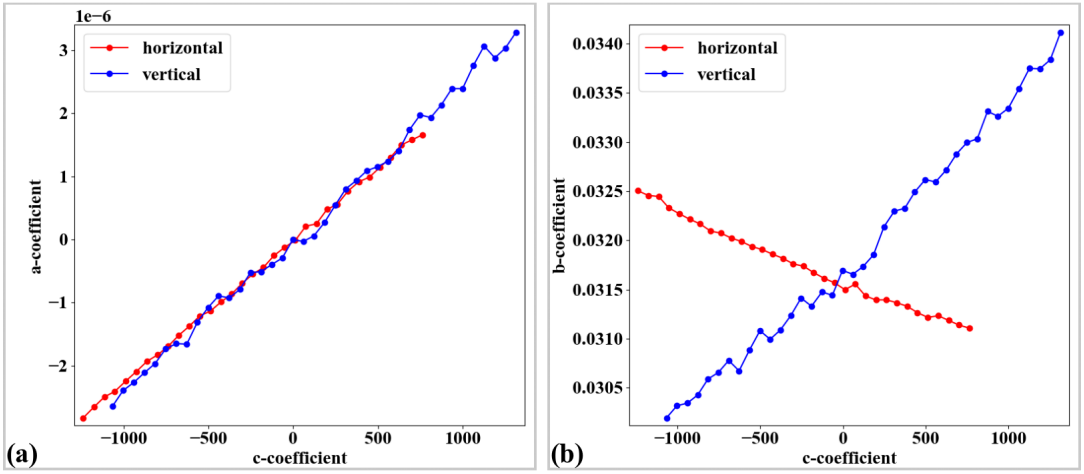


Fig. 33: . (a) Plot of a-coefficients vs c-coefficients of parabolic fits. (b) Plot of b-coefficients vs c-coefficients.

- As can be seen in Fig. 33 (b), the slopes are significantly different between two groups of lines. To correct this perspective effect, the coefficients of parabolas are adjusted (Fig. 34) to satisfy the conditions as explained

in *section 2.2*. After that, grid of points are regenerated using these updated coefficients (Fig. 35).

```
# Correct parabola coefficients
hor_coef_corr, ver_coef_corr = proc._generate_non_perspective_parabola_
↪coef(
                                    list_hor_lines, list_ver_lines)[0:2]
# Plot to check the results
plt.figure(0)
plt.plot(hor_coef_corr[:, 2], hor_coef_corr[:, 0], "-o")
plt.plot(ver_coef_corr[:, 2], ver_coef_corr[:, 0], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("a-coefficient")
plt.figure(1)
plt.plot(hor_coef_corr[:, 2], -hor_coef_corr[:, 1], "-o")
plt.plot(ver_coef_corr[:, 2], ver_coef_corr[:, 1], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("b-coefficient")
plt.ylim((0.03, 0.034))
plt.show()
# Regenerate grid points with the correction of perspective effect.
list_hor_lines, list_ver_lines = proc.regenerate_grid_points_parabola(
    list_hor_lines, list_ver_lines, perspective=True)
# Save output for checking
io.save_plot_image(output_base + "/horizontal_lines_regenerated.png",␣
↪list_hor_lines,
                   height, width)
io.save_plot_image(output_base + "/vertical_lines_regenerated.png", list_
↪ver_lines,
                   height, width)
```

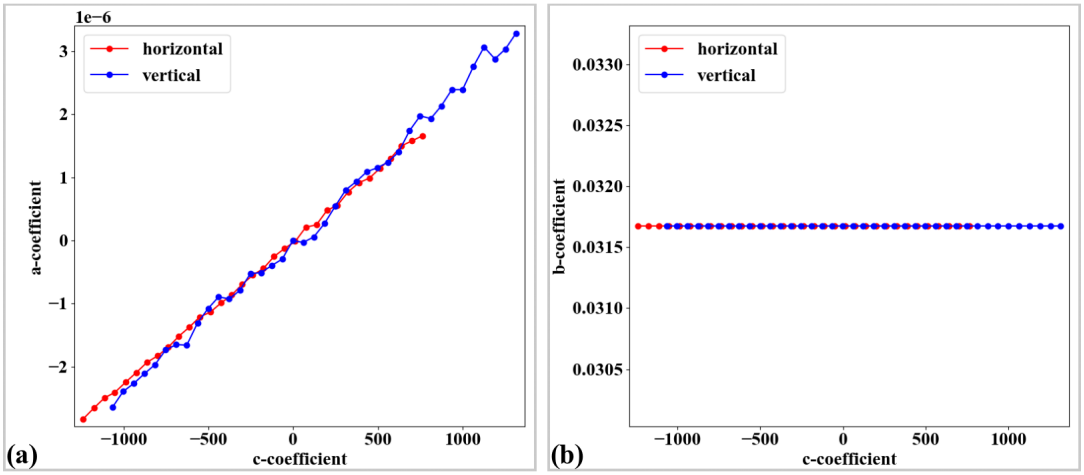


Fig. 34: Parabola coefficients after correction. (a) Plot of a-coefficients vs c-coefficients. (b) Plot of b-coefficients vs c-coefficients.

- The rest of the workflow is to calculate the center of distortion and coefficients of the backward model, then unwarp the image. As can be seen in Fig. 37 and Fig. 38, the improvement of the accuracy after correcting the perspective effect is clear.

```
(xcenter, ycenter) = proc.find_cod_coarse(list_hor_lines, list_ver_lines)
list_fact = proc.calc_coef_backward(list_hor_lines, list_ver_lines,
                                    xcenter, ycenter, num_coef)
io.save_metadata_txt(output_base + "/coefficients_radial_distortion.txt",
                     xcenter, ycenter, list_fact)
```

(continues on next page)

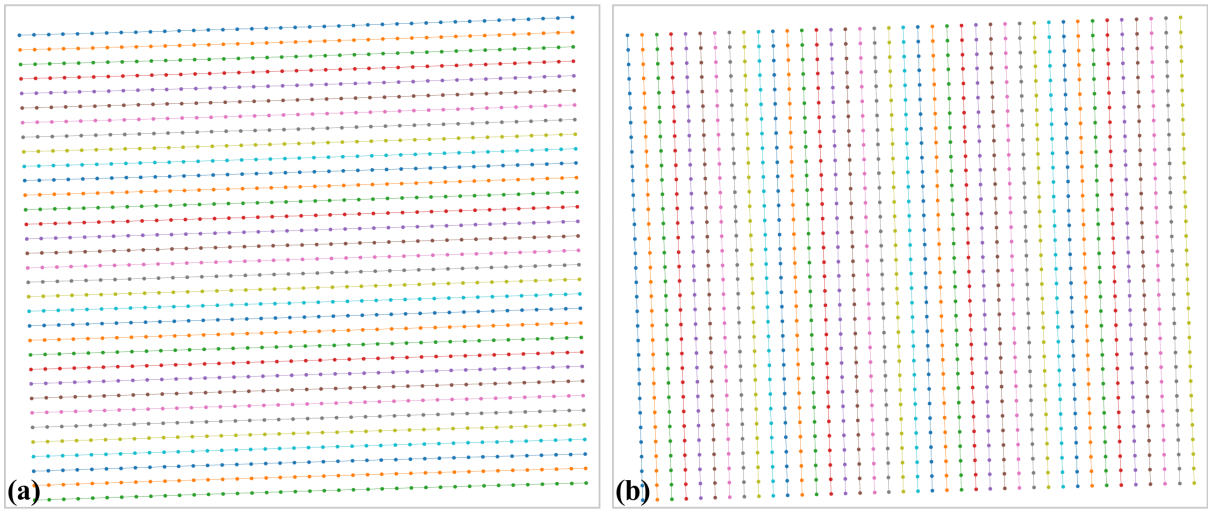


Fig. 35: Grid points regenerated using the updated parabola-coefficients. Note that there are no missing points as compared to Fig. 32. (a) Horizontal lines. (b) Vertical lines.

(continued from previous page)

```
print("X-center: {0}. Y-center: {1}".format(xcenter, ycenter))
print("Coefficients: {0}".format(list_fact))
# Check the correction results:
# Apply correction to the lines of points
list_uhor_lines = post.unwarp_line_backward(list_hor_lines, xcenter,␣
↪ycenter, list_fact)
list_uver_lines = post.unwarp_line_backward(list_ver_lines, xcenter,␣
↪ycenter, list_fact)
list_hor_data = post.calc_residual_hor(list_uhor_lines, xcenter, ycenter)
list_ver_data = post.calc_residual_ver(list_uver_lines, xcenter, ycenter)
io.save_residual_plot(output_base + "/hor_residual_after_correction.png",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_after_correction.png",
                      list_ver_data, height, width)
# Load coefficients from previous calculation if need to
# (xcenter, ycenter, list_fact) = io.load_metadata_txt(
#     output_base + "/coefficients_radial_distortion.txt")
# Correct the image
corrected_mat = post.unwarp_image_backward(mat0, xcenter, ycenter, list_
↪fact)
# Save results. Note that the output is 32-bit numpy array. Convert to␣
↪lower-bit if need to.
io.save_image(output_base + "/corrected_image.tif", corrected_mat)
io.save_image(output_base + "/difference.tif", corrected_mat - mat0)
```

Click here to download the Python codes.

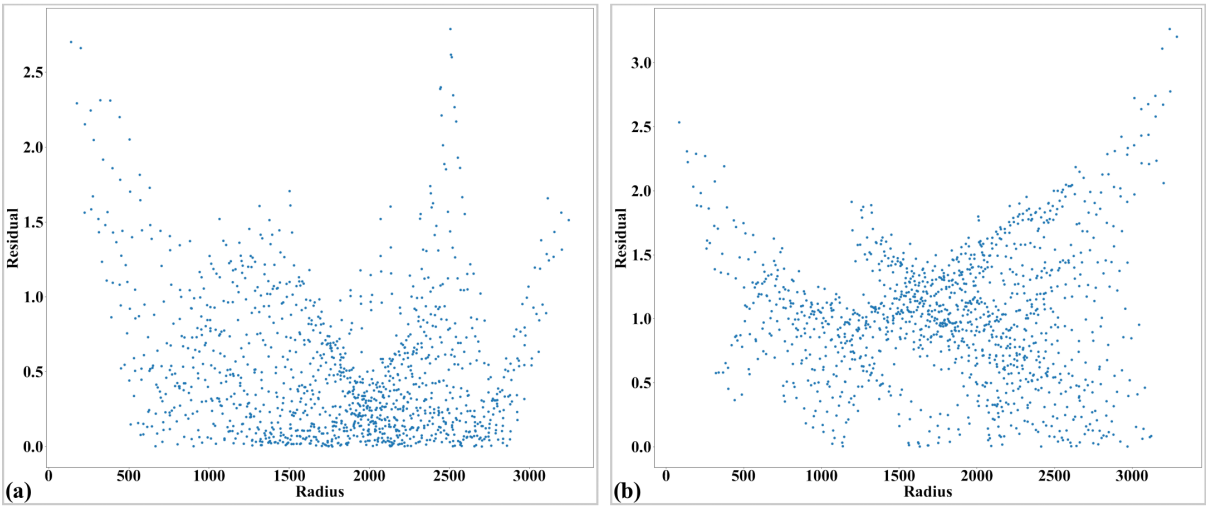


Fig. 36: Residual of the distorted points. The origin of the coordinate system is at the top-left of an image. (a) For horizontal lines. (b) For vertical lines.

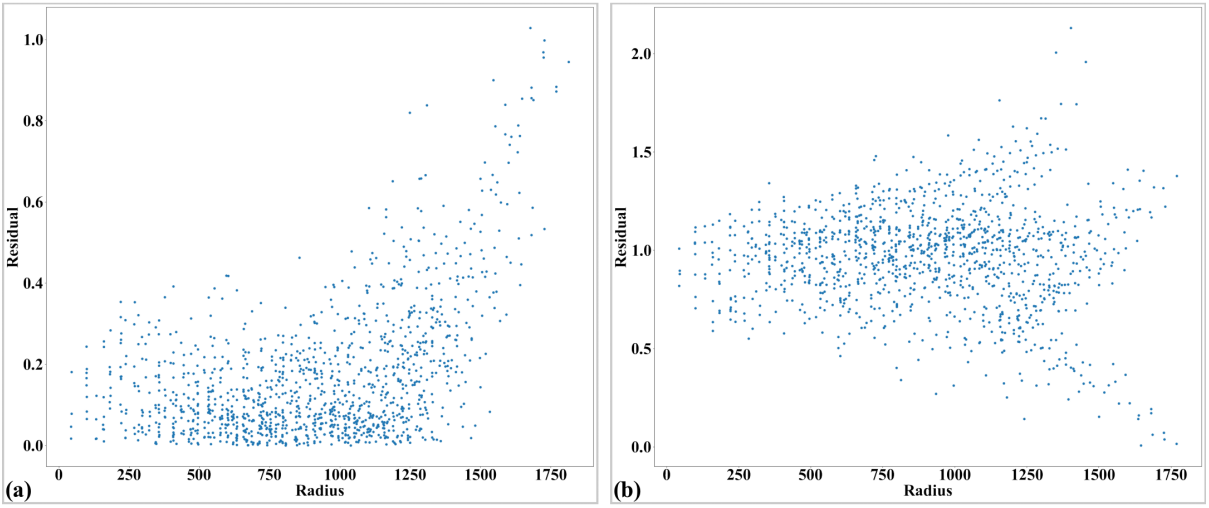


Fig. 37: Residual of the unwarpped points with perspective effect. The origin of the coordinate system is at the center of distortion. (a) For horizontal lines. (b) For vertical lines.

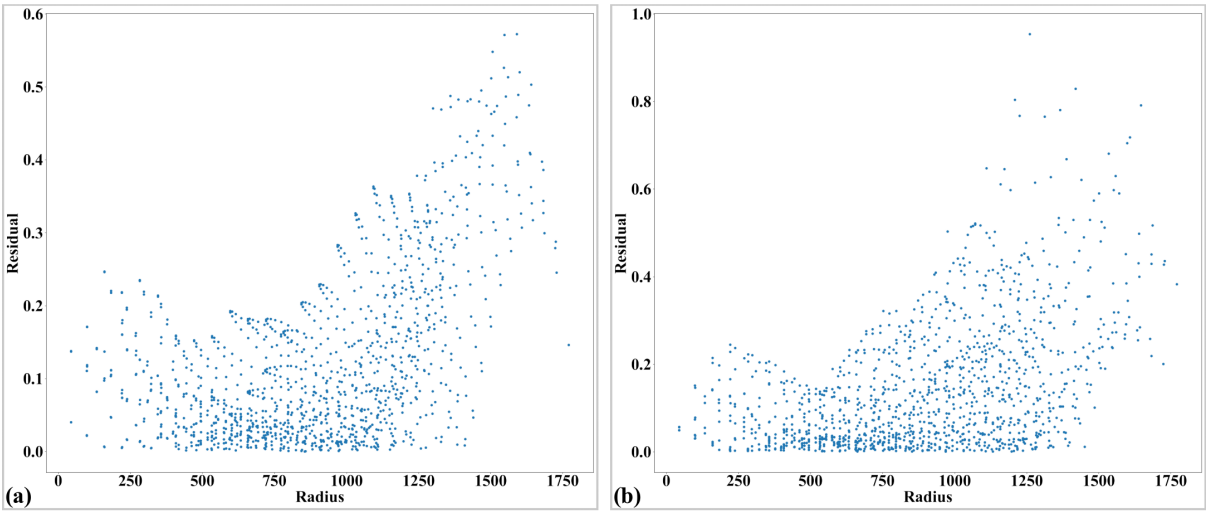


Fig. 38: Residual of the unwarpped points after correcting the perspective effect. (a) For horizontal lines. (b) For vertical lines.

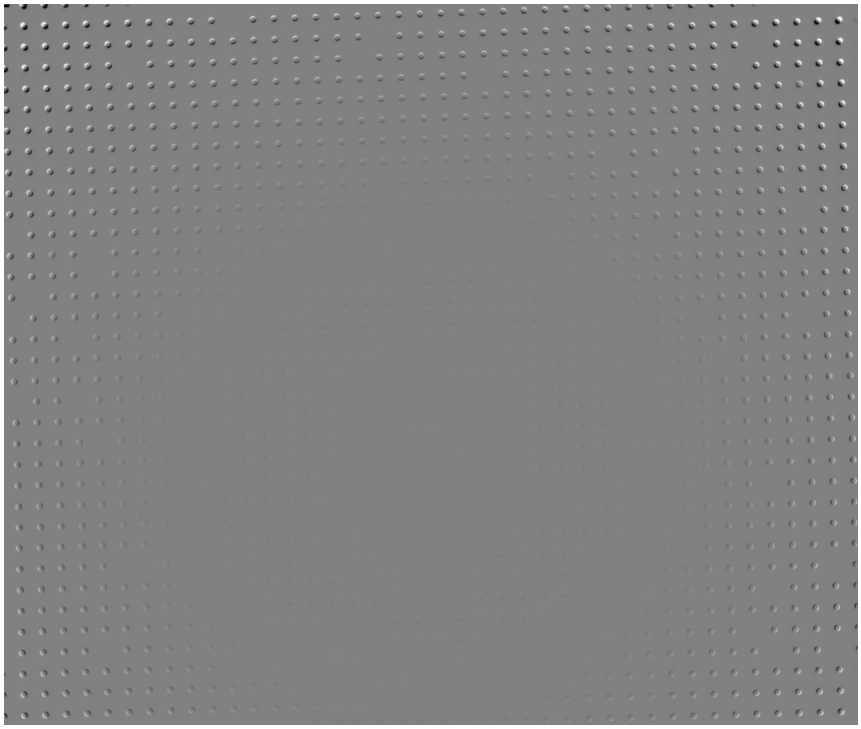


Fig. 39: Difference between images before and after distortion correction.

Process a challenging X-ray target image

Extracting reference-points from an image and grouping them into lines are the most challenging steps in the processing workflow. Calculating coefficients of distortion-correction models is straightforward using Discorpy's API. The following demo shows how to tweak parameters of pre-processing methods to process a challenging calibration-image which was acquired at Beamline I13, Diamond Light Source.

- First of all, the background of the image is corrected to support the step of binarizing the image.

```
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.prep.preprocessing as prep
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

# Initial parameters
file_path = "../../data/dot_pattern_04.jpg"
output_base = "E:/output_demo_03/"
num_coef = 5  # Number of polynomial coefficients
mat0 = io.load_image(file_path)  # Load image
(height, width) = mat0.shape

# Correct non-uniform background.
mat1 = prep.normalization_fft(mat0, sigma=20)
io.save_image(output_base + "/image_normed.tif", mat1)
```

- The binarization method uses the Otsu's method for calculating the threshold by default. In a case that the calculated value may not work, users can pass a threshold value manually or calculate it by another method.

```
# Segment dots.
threshold = prep.calculate_threshold(mat1, bgr="bright", snr=3.0)
mat1 = prep.binarization(mat1, ratio=0.5, thres=threshold)
```

(continues on next page)

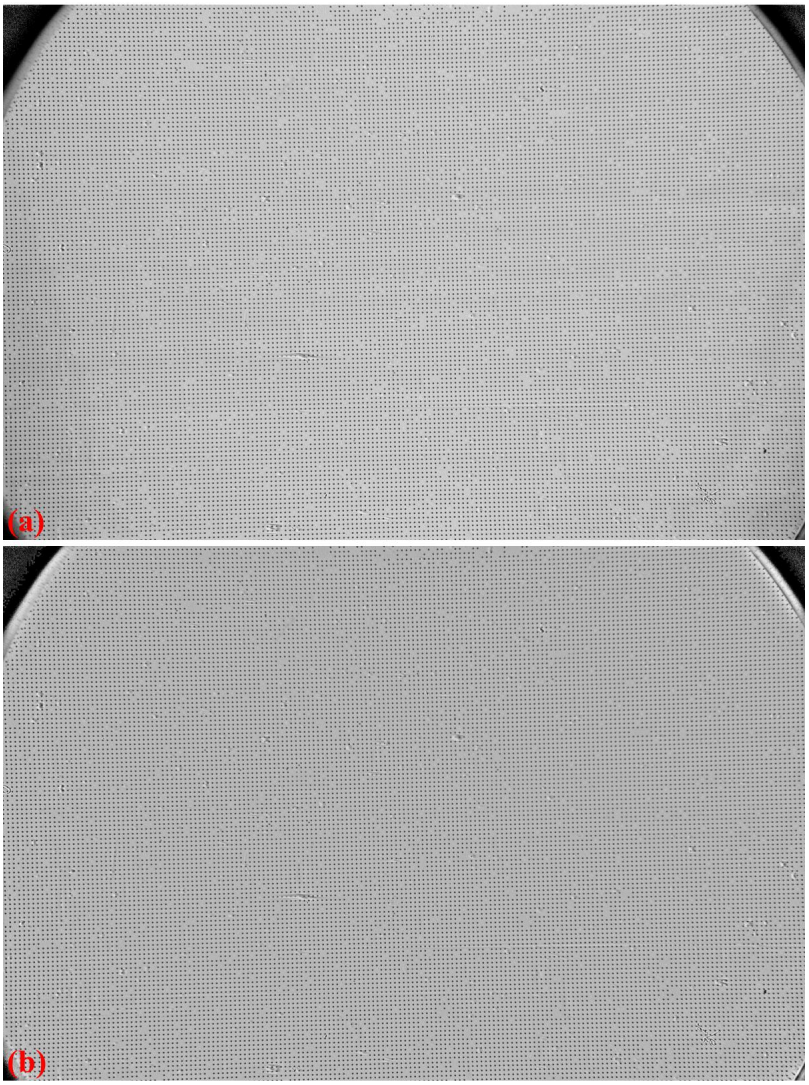


Fig. 40: . (a) X-ray target image. (b) Normalized image.

(continued from previous page)

```
io.save_image(output_base + "/image_binarized.tif", mat1)
```

- There are lots of non-dot objects left after the binarization step (Fig. 41 (a)). They can be removed further (Fig. 41 (b)) by tweaking parameters of the selection methods. The *ratio* parameter is increased because there are small binary-dots around the edges of the image.

```
# Calculate the median dot size and distance between them.
(dot_size, dot_dist) = prep.calc_size_distance(mat1)
# Remove non-dot objects
mat1 = prep.select_dots_based_size(mat1, dot_size, ratio=0.8)
# Remove non-elliptical objects
mat1 = prep.select_dots_based_ratio(mat1, ratio=0.8)
io.save_image(output_base + "/image_cleaned.tif", mat1)
```

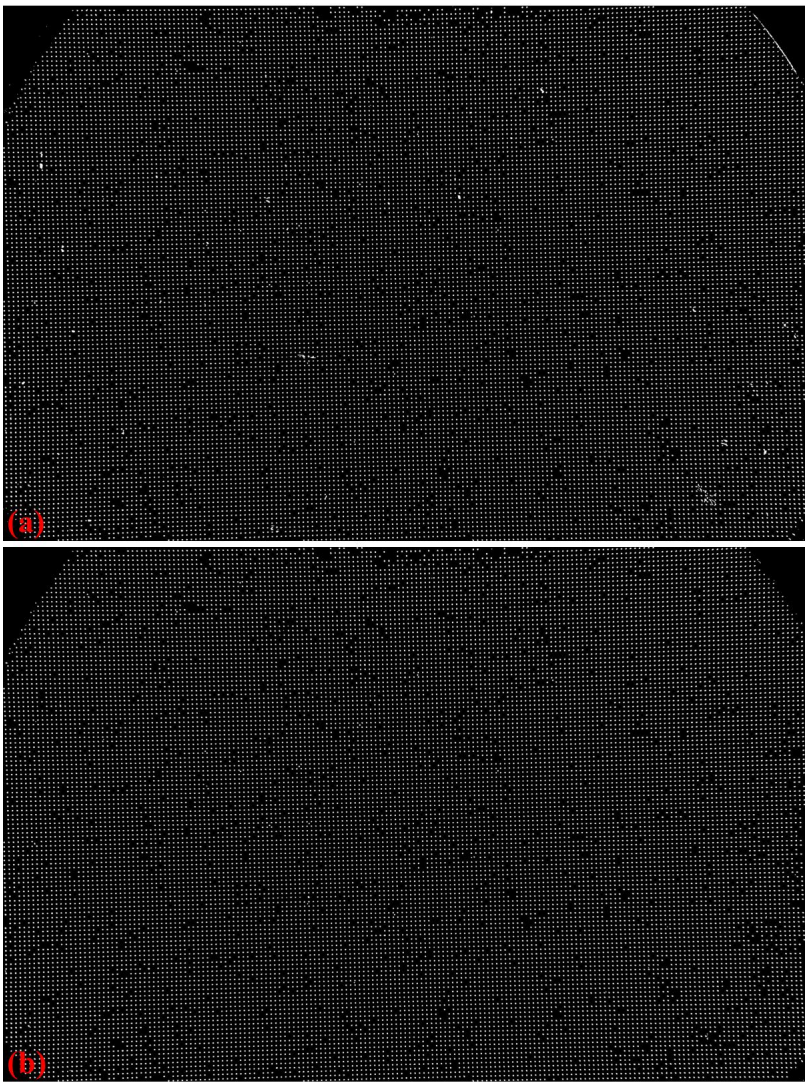


Fig. 41: . (a) Segmented binary objects. (b) Large-size objects removed.

- Many of these small binary-objects are not the dots (reference-objects). They can be removed in the grouping step as shown in the codes below and can be seen in Fig. 42. Distortion is strong (Fig. 43).

```
# Calculate the slopes of horizontal lines and vertical lines.
hor_slope = prep.calc_hor_slope(mat1)
```

(continues on next page)

(continued from previous page)

```
ver_slope = prep.calc_ver_slope(mat1)
print("Horizontal slope: {0}. Vertical slope: {1}".format(hor_slope, ver_
↪slope))

# Group points into lines
list_hor_lines = prep.group_dots_hor_lines(mat1, hor_slope, dot_dist,␣
↪ratio=0.3,
                                           num_dot_miss=10, accepted_
↪ratio=0.65)
list_ver_lines = prep.group_dots_ver_lines(mat1, ver_slope, dot_dist,␣
↪ratio=0.3,
                                           num_dot_miss=10, accepted_
↪ratio=0.65)
# Remove outliers
list_hor_lines = prep.remove_residual_dots_hor(list_hor_lines, hor_slope,
                                               residual=2.0)
list_ver_lines = prep.remove_residual_dots_ver(list_ver_lines, ver_slope,
                                               residual=2.0)

# Save output for checking
io.save_plot_image(output_base + "/horizontal_lines.png", list_hor_lines,
                   height, width)
io.save_plot_image(output_base + "/vertical_lines.png", list_ver_lines,
                   height, width)
list_hor_data = post.calc_residual_hor(list_hor_lines, 0.0, 0.0)
list_ver_data = post.calc_residual_ver(list_ver_lines, 0.0, 0.0)
io.save_residual_plot(output_base + "/hor_residual_before_correction.png
↪",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_before_correction.png
↪",
                      list_ver_data, height, width)
```

- There is perspective effect (Fig. 44) caused by the X-ray target was mounted not in parallel to the CCD chip. This can be corrected by a single line of code.

```
# Regenerate grid points after correcting the perspective effect.
list_hor_lines, list_ver_lines = proc.regenerate_grid_points_parabola(
    list_hor_lines, list_ver_lines, perspective=True)
```

- The next steps of calculating the center of rotation and coefficients of a polynomial model are straightforward.

```
# Calculate parameters of the radial correction model
(xcenter, ycenter) = proc.find_cod_coarse(list_hor_lines, list_ver_lines)
list_fact = proc.calc_coef_backward(list_hor_lines, list_ver_lines,
                                    xcenter, ycenter, num_coef)
io.save_metadata_txt(output_base + "/coefficients_radial_distortion.txt",
                     xcenter, ycenter, list_fact)
print("X-center: {0}. Y-center: {1}".format(xcenter, ycenter))
print("Coefficients: {0}".format(list_fact))
```

- The accuracy of the model is checked by unwarping the lines of points and the image. There are some points with residuals more than 1 pixel near the edges of the image. It can be caused by blurry dots.

```
# Apply correction to the lines of points
list_uhor_lines = post.unwarp_line_backward(list_hor_lines, xcenter,␣
↪ycenter,
```

(continues on next page)

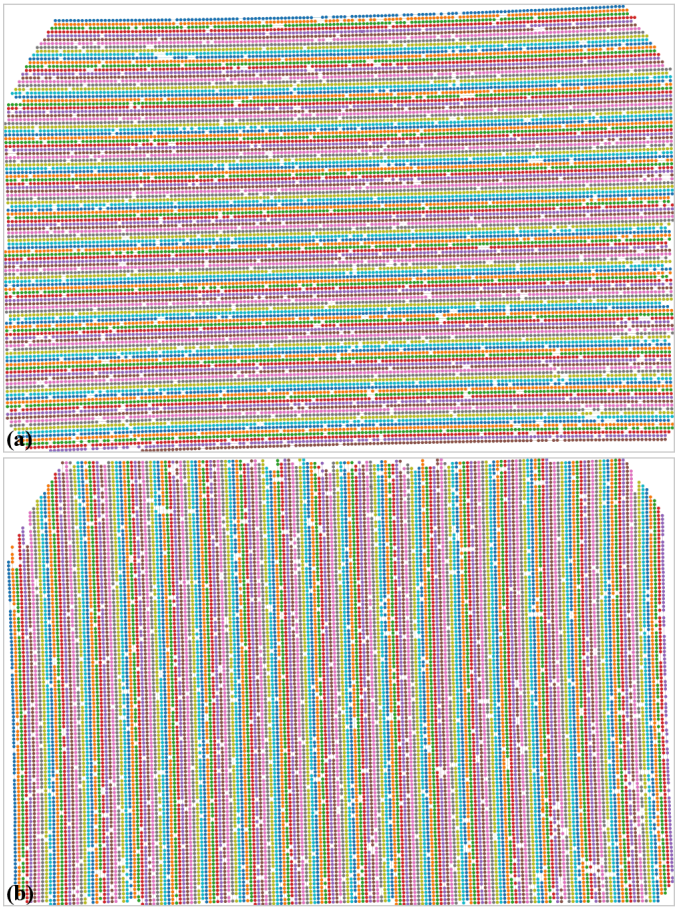


Fig. 42: . (a) Grouped horizontal points. (b) Grouped vertical points.

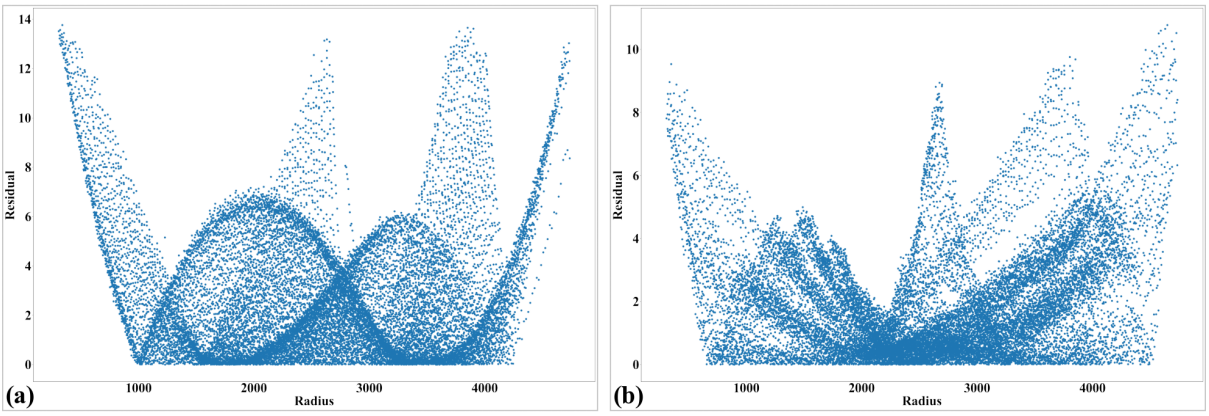


Fig. 43: Residual of the distorted points. The origin of the coordinate system is at the top-left of an image. (a) For horizontal lines. (b) For vertical lines.

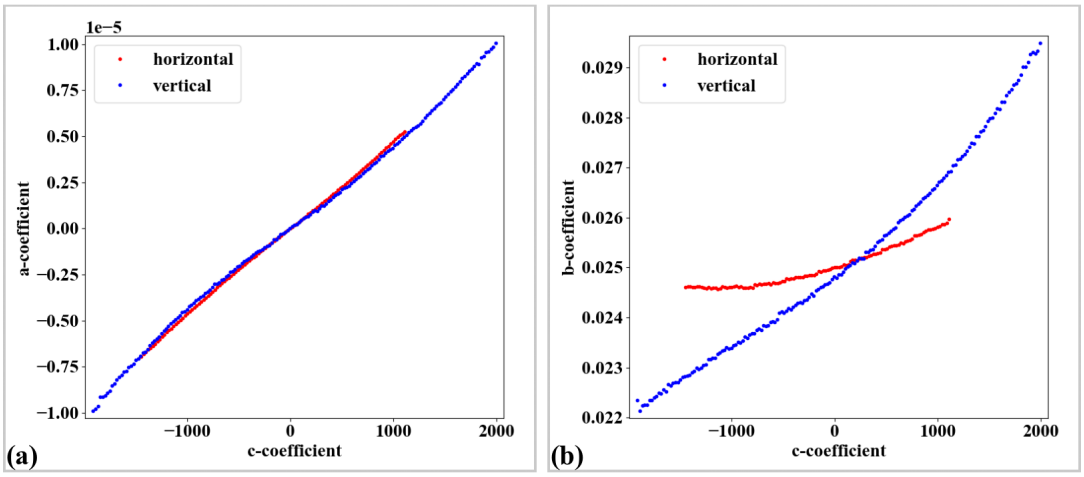


Fig. 44: Impact of the perspective distortion. (a) Plot of a-coefficients vs c-coefficients of parabolic fits. (b) Plot of b-coefficients vs c-coefficients.

(continued from previous page)

```
                                                list_fact)
list_uver_lines = post.unwarp_line_backward(list_ver_lines, xcenter,␣
↪ycenter,
                                                list_fact)
# Calculate the residual of the unwarpped points.
list_hor_data = post.calc_residual_hor(list_uhor_lines, xcenter, ycenter)
list_ver_data = post.calc_residual_ver(list_uver_lines, xcenter, ycenter)
# Save the results for checking
io.save_plot_image(output_base + "/unwarpped_horizontal_lines.png",
                   list_uhor_lines, height, width)
io.save_plot_image(output_base + "/unwarpped_vertical_lines.png",
                   list_uver_lines, height, width)
io.save_residual_plot(output_base + "/hor_residual_after_correction.png",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_after_correction.png",
                      list_ver_data, height, width)

# Correct the image
corrected_mat = post.unwarp_image_backward(mat0, xcenter, ycenter, list_
↪fact)
# Save results. Note that the output is 32-bit-tif.
io.save_image(output_base + "/corrected_image.tif", corrected_mat)
io.save_image(output_base + "/difference.tif", corrected_mat - mat0)
```

Click here to download the Python codes.

Process a line-pattern image

The following workflow shows how to use Discorpy to process a line-pattern image, collected at Beamline I13 Diamond Light Source.

- Load the image, calculate the slopes of lines at the middle of the image and distances between. These values will be used by the grouping step.

```
import discorpy.losa.loadersaver as io
import discorpy.prep.preprocessing as prep
import discorpy.prep.linepattern as lprep
```

(continues on next page)

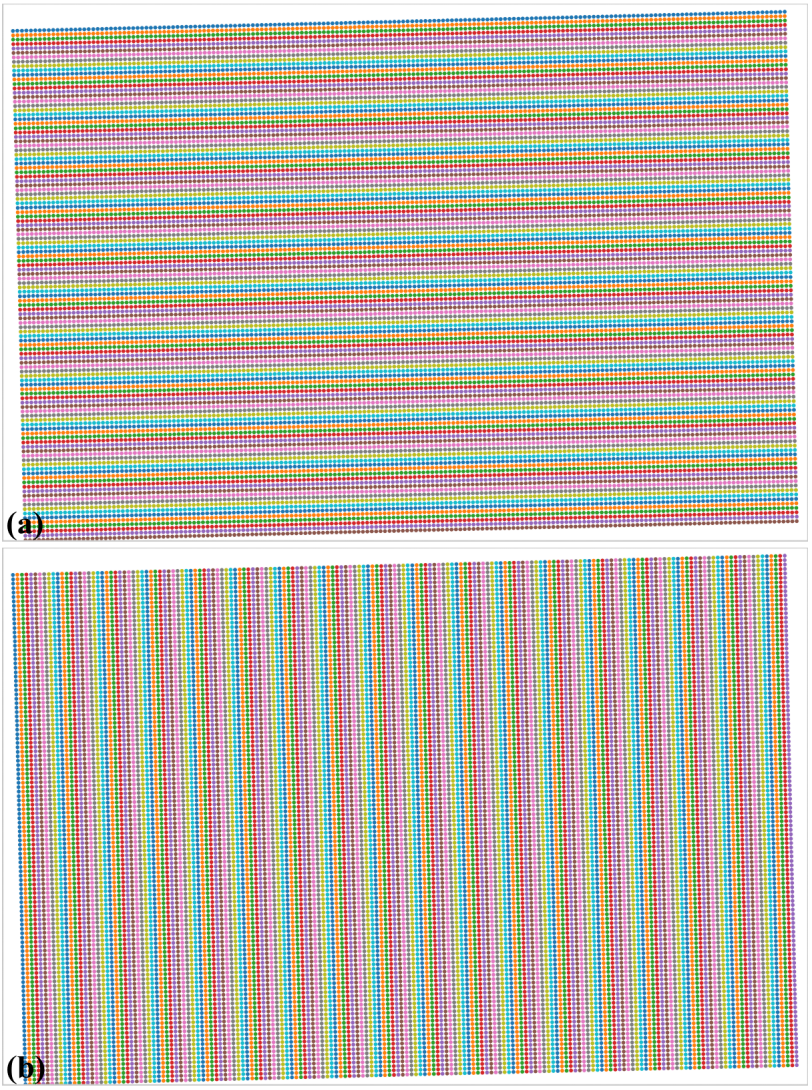


Fig. 45: Unwarped lines of points. Note that these lines are regenerated after the step of correcting perspective effect. (a) Horizontal lines. (b) Vertical lines.

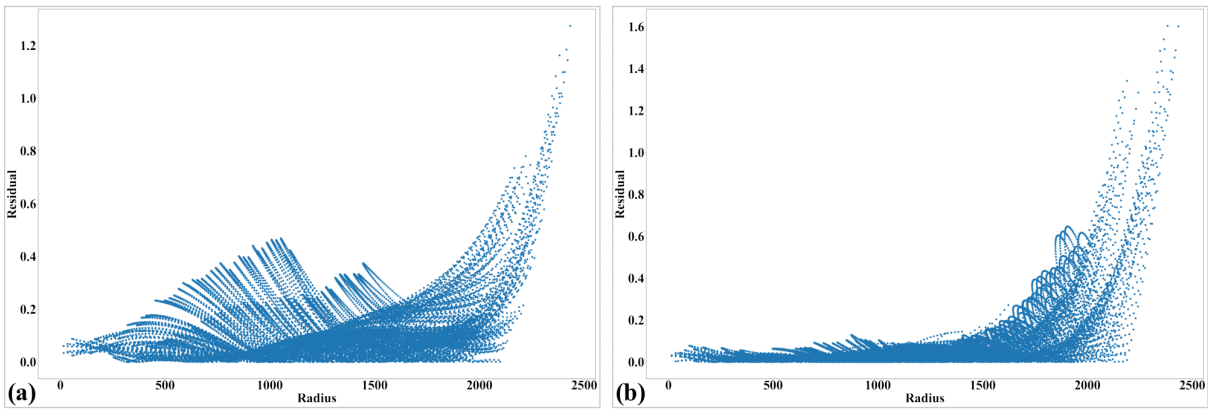


Fig. 46: Residual of the unwarpped points after correcting the perspective effect. (a) For horizontal lines. (b) For vertical lines.

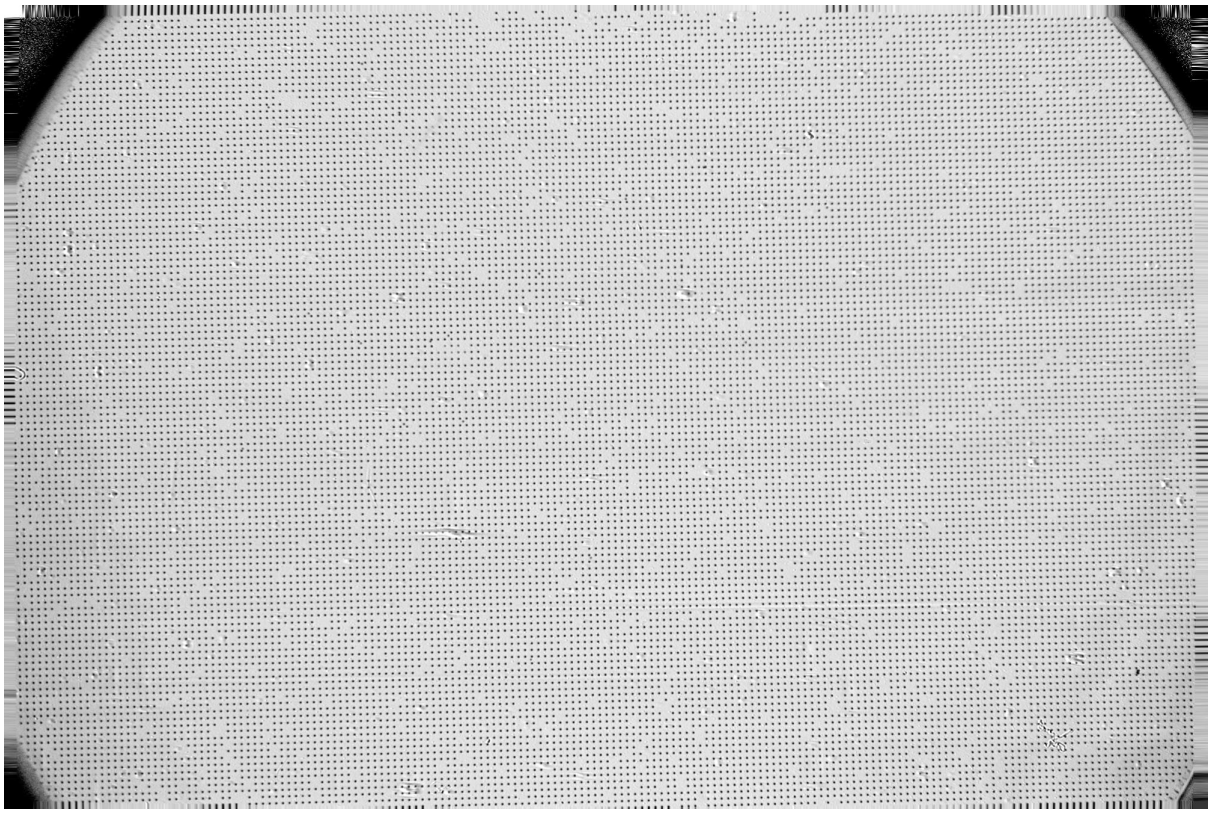


Fig. 47: Unwarped image

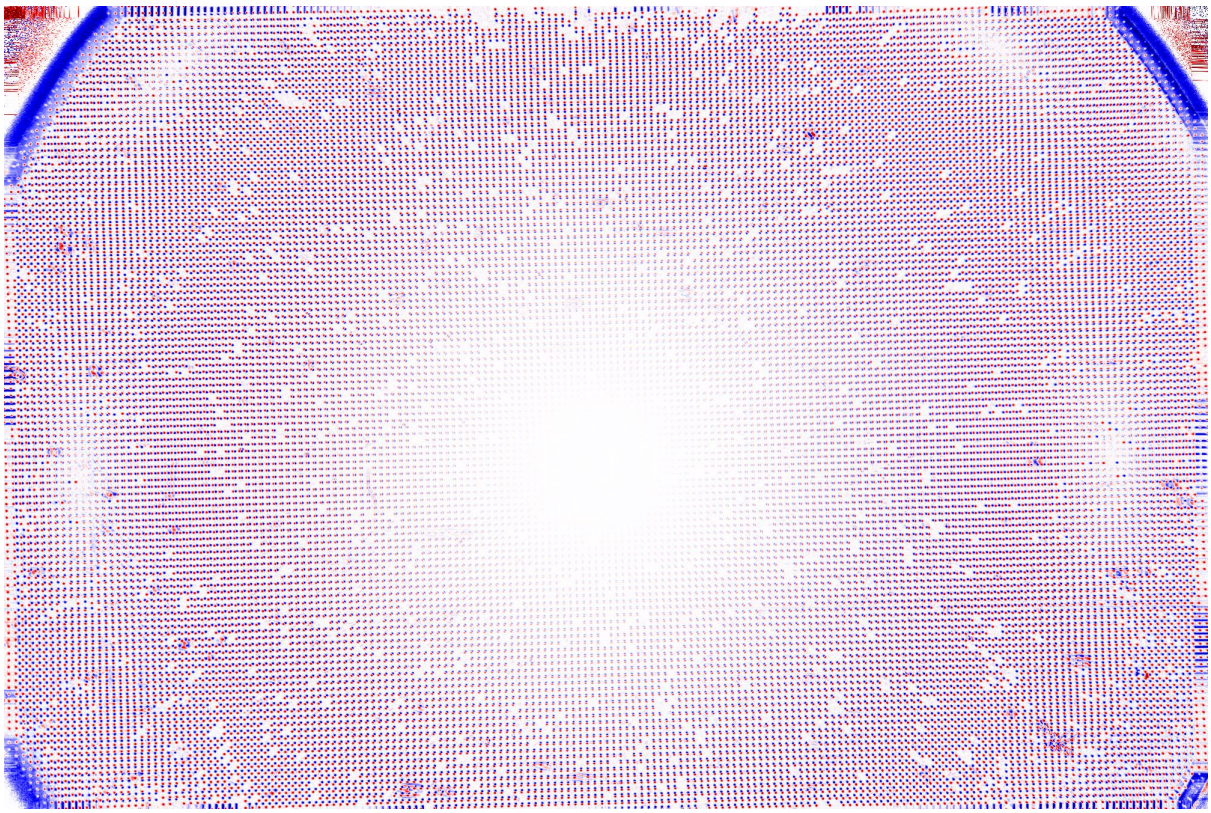


Fig. 48: Difference between images before (Fig. 40) and after (Fig. 47) unwarping.

(continued from previous page)

```
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post


# Initial parameters
file_path = "../../data/line_pattern_01.jpg"
output_base = "E:/output_demo_04/"
num_coef = 5  # Number of polynomial coefficients

print("1-> Load image: {}".format(file_path))
mat0 = io.load_image(file_path)
(height, width) = mat0.shape

print("2-> Calculate slope and distance between lines!!!")
slope_hor, dist_hor = lprep.calc_slope_distance_hor_lines(mat0)
slope_ver, dist_ver = lprep.calc_slope_distance_ver_lines(mat0)
print("    Horizontal slope: ", slope_hor, " Distance: ", dist_hor)
print("    Vertical slope: ", slope_ver, " Distance: ", dist_ver)
# Horizontal slope:  9.921048172113442e-16  Distance:  62.22050730757496
# Vertical slope:  1.5501637768927253e-17  Distance:  62.258521480417485
```

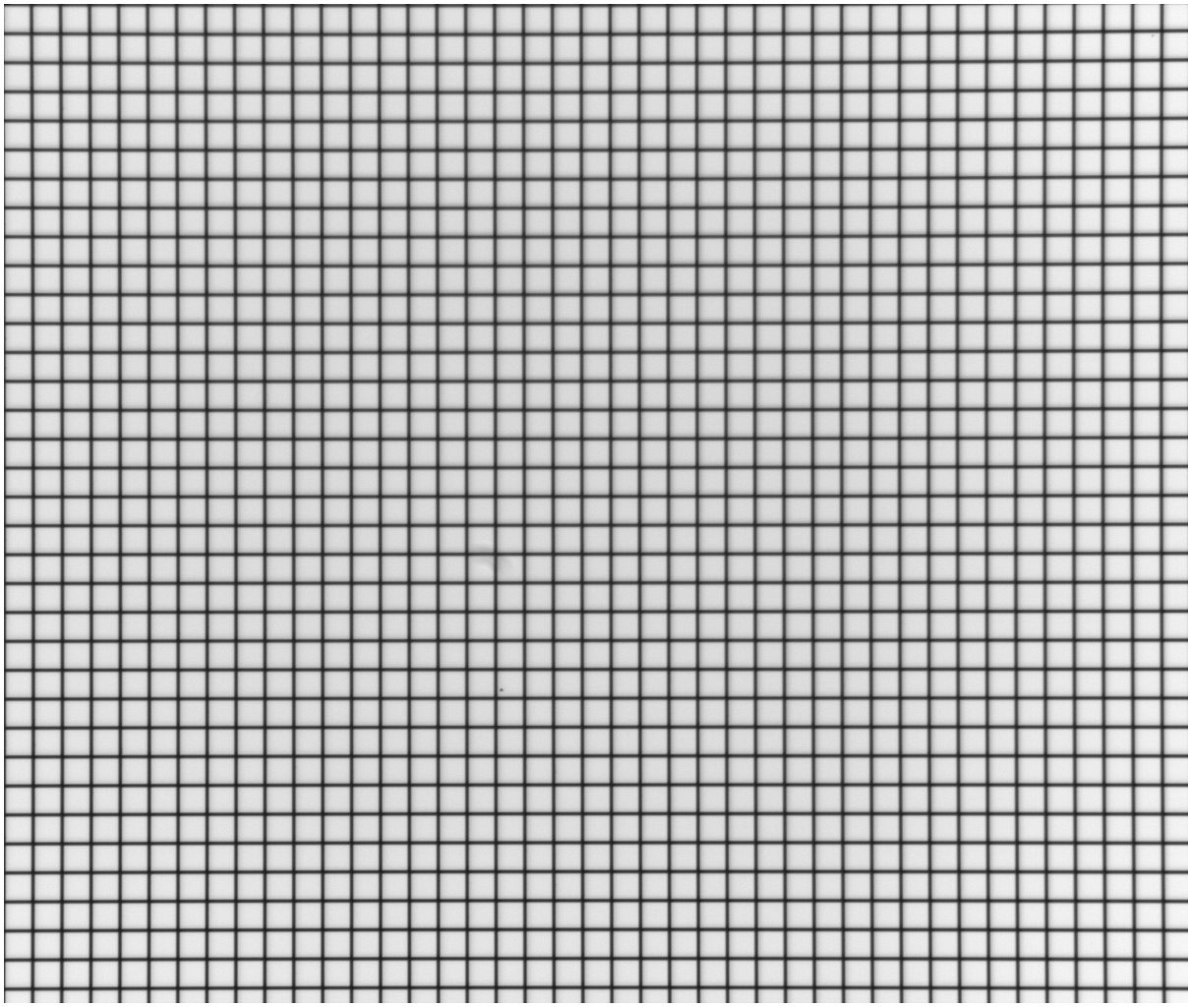


Fig. 49: Line-pattern image.

- Points belong to lines are detected by locating local *minimum points* of intensity-profiles across the image.

The number of intensity-profiles generated is controlled by the *ratio* parameter. The *sensitive* parameter controls the sensitivity of the detection method.

```
print("3-> Extract reference-points !!!!")
list_points_hor_lines = lprep.get_cross_points_hor_lines(mat0, slope_ver,
↪ dist_ver, ratio=0.5, sensitive=0.1)
list_points_ver_lines = lprep.get_cross_points_ver_lines(mat0, slope_hor,
↪ dist_hor, ratio=0.5, sensitive=0.1)
io.save_plot_points(output_base + "/extracted_hor_points.png", list_
↪points_hor_lines, height, width)
io.save_plot_points(output_base + "/extracted_ver_points.png", list_
↪points_ver_lines, height, width)
```

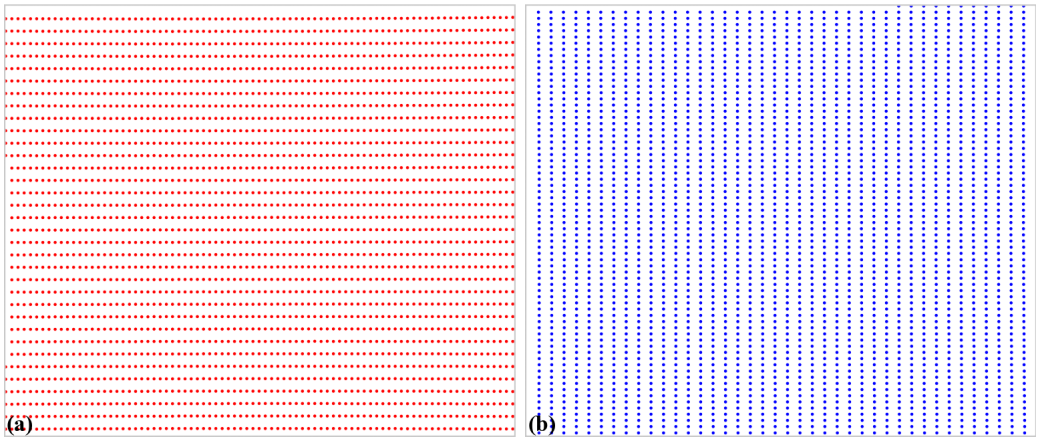


Fig. 50: Detected points. (a) Horizontal lines. (b) Vertical lines.

- Group detected-points into lines (Fig. 51). This step can remove unwanted points.

```
print("4-> Group points into lines !!!!")
list_hor_lines = prep.group_dots_hor_lines(list_points_hor_lines, slope_
↪hor, dist_hor)
list_ver_lines = prep.group_dots_ver_lines(list_points_ver_lines, slope_
↪ver, dist_ver)
# Optional: remove residual dots
list_hor_lines = prep.remove_residual_dots_hor(list_hor_lines, slope_hor,
↪ 2.0)
list_ver_lines = prep.remove_residual_dots_ver(list_ver_lines, slope_ver,
↪ 2.0)
io.save_plot_image(output_base + "/grouped_hor_lines.png", list_hor_
↪lines, height, width)
io.save_plot_image(output_base + "/grouped_ver_lines.png", list_ver_
↪lines, height, width)
```

- The rest of the workflow is similar to other *demos*.

```
print("5-> Correct perspective effect !!!!")
# Optional: correct perspective effect.
list_hor_lines, list_ver_lines = proc.regenerate_grid_points_parabola(
    list_hor_lines, list_ver_lines, perspective=True)

# Check if the distortion is significant.
list_hor_data = post.calc_residual_hor(list_hor_lines, 0.0, 0.0)
io.save_residual_plot(output_base + "/residual_horizontal_points_before.
↪png",
```

(continues on next page)

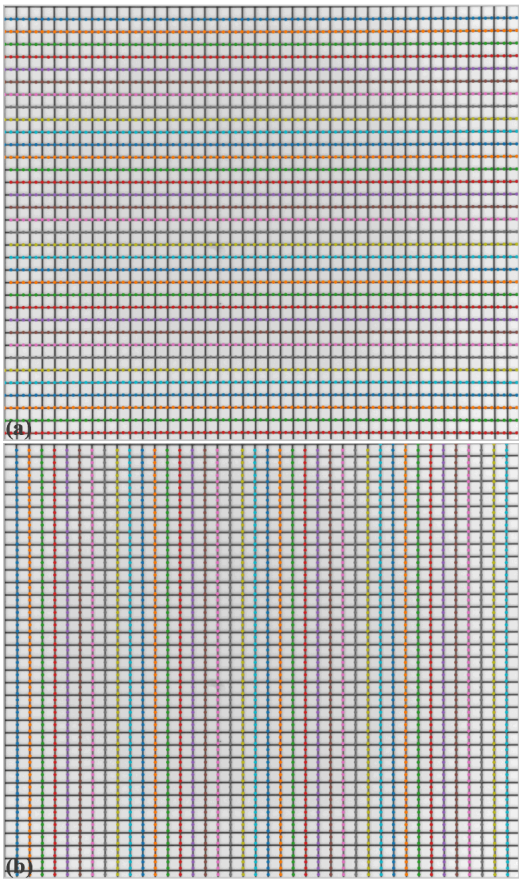


Fig. 51: Grouped points. (a) Horizontal lines. (b) Vertical lines.

(continued from previous page)

```
                        list_hor_data, height, width)
list_ver_data = post.calc_residual_ver(list_ver_lines, 0.0, 0.0)
io.save_residual_plot(output_base + "/residual_vertical_points_before.png
↪",
                        list_ver_data, height, width)

print("6-> Calculate the centre of distortion !!!!")
(xcenter, ycenter) = proc.find_cod_coarse(list_hor_lines, list_ver_lines)
print("   X-center: {0}, Y-center: {1}".format(xcenter, ycenter))

print("7-> Calculate radial distortion coefficients !!!!")
list_fact = proc.calc_coef_backward(list_hor_lines, list_ver_lines,␣
↪xcenter,
                                    ycenter, num_coef)

# Check the correction results
list_uhor_lines = post.unwarp_line_backward(list_hor_lines, xcenter,␣
↪ycenter, list_fact)
list_uver_lines = post.unwarp_line_backward(list_ver_lines, xcenter,␣
↪ycenter, list_fact)
list_hor_data = post.calc_residual_hor(list_uhor_lines, xcenter, ycenter)
list_ver_data = post.calc_residual_ver(list_uver_lines, xcenter, ycenter)
io.save_residual_plot(output_base + "/residual_horizontal_points_after.
↪png",
                        list_hor_data, height, width)
io.save_residual_plot(output_base + "/residual_vertical_points_after.png
↪",
                        list_ver_data, height, width)
# Output
print("8-> Apply correction to image !!!!")
corrected_mat = post.unwarp_image_backward(mat0, xcenter, ycenter, list_
↪fact)
io.save_image(output_base + "/corrected_image.tif", corrected_mat)
io.save_metadata_txt(output_base + "/coefficients.txt", xcenter, ycenter,
↪ list_fact)
io.save_image(output_base + "/difference.tif", mat0 - corrected_mat)
print("!!! Done !!!!")
```

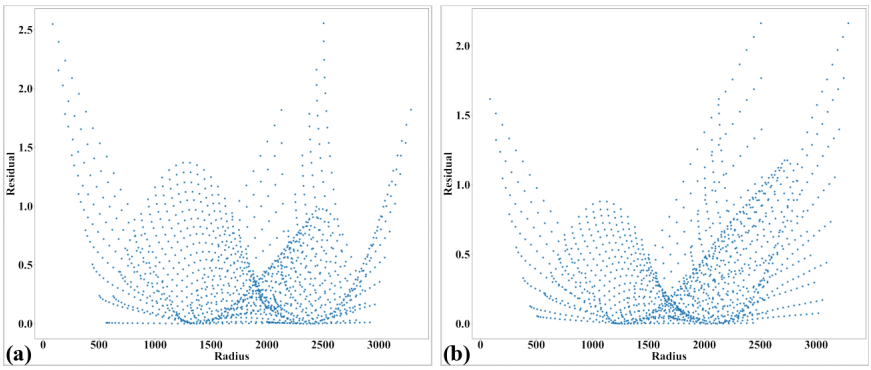


Fig. 52: Residual of distorted points. (a) Horizontal lines. (b) Vertical lines.

Click here to download the Python codes.

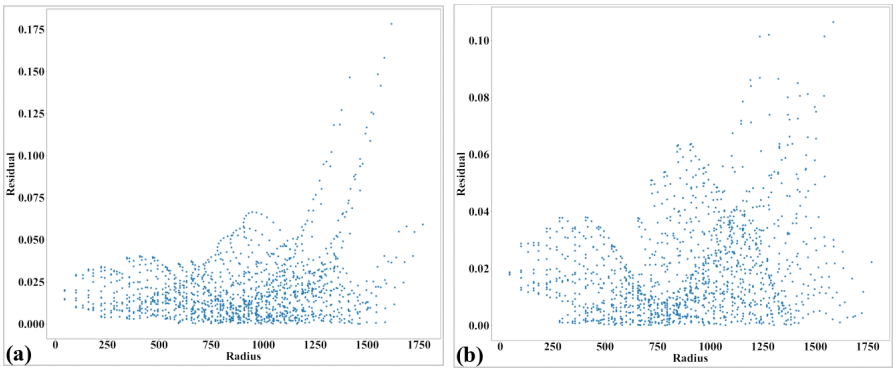


Fig. 53: Residual of unwarped points. (a) Horizontal lines. (b) Vertical lines.

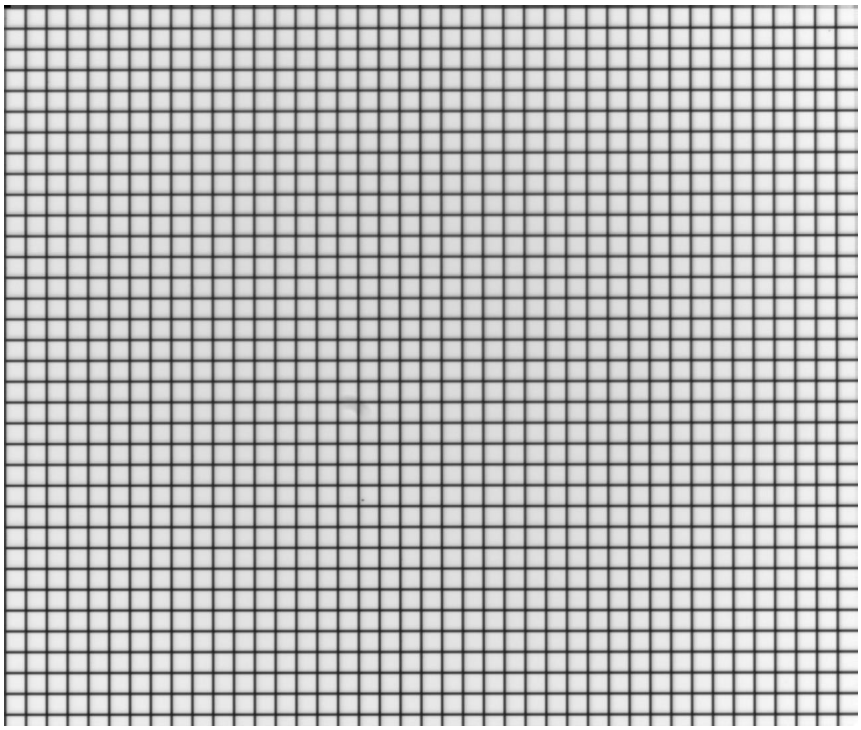


Fig. 54: Unwarped image.

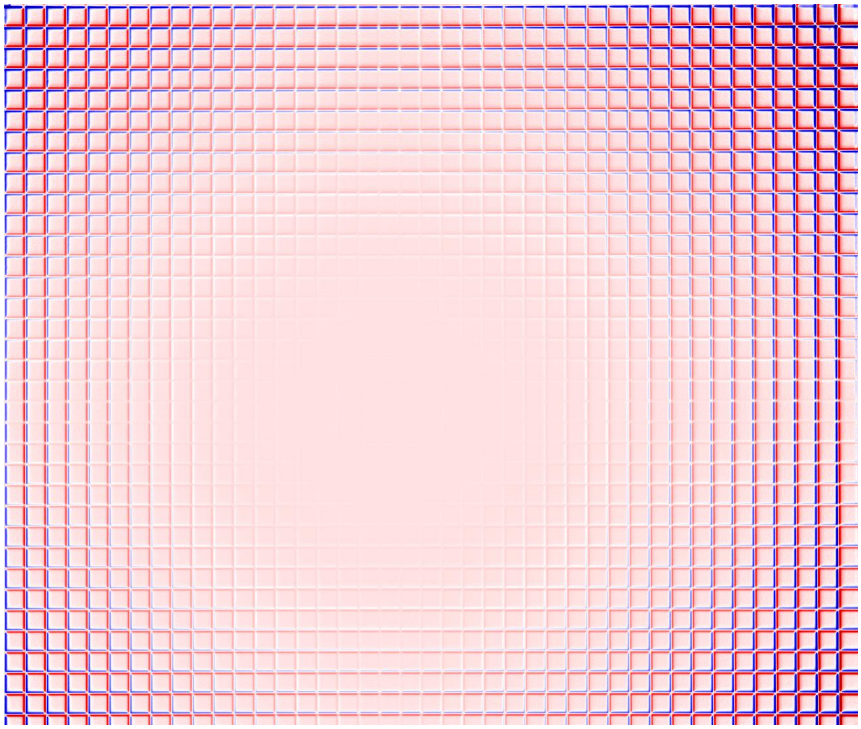


Fig. 55: Difference between images before (Fig. 49) and after (Fig. 54) unwarping.

Correct both radial distortion and perspective distortion

The following workflow shows how to use Discorpy to correct both radial distortion and perspective distortion from a dot-pattern image (Fig. 56 (a)) shared on Jerome Kieffer's github page. The image suffers both types of distortions.

- Load the image, segment the dots (Fig. 56 (b)), group them into lines (Fig. 57), and check the residual of the distorted points (Fig. 58).

```
import numpy as np
import matplotlib.pyplot as plt
import discorpy.losa.loadersaver as io
import discorpy.prep.preprocessing as prep
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

# Initial parameters
file_path = "../../data/dot_pattern_06.jpg"
output_base = "E:/output_demo_05/"
num_coef = 4  # Number of polynomial coefficients
mat0 = io.load_image(file_path) # Load image
(height, width) = mat0.shape

# Normalize background
mat1 = prep.normalization_fft(mat0, sigma=20)
# Segment dots
threshold = prep.calculate_threshold(mat1, bgr="bright", snr=1.5)
mat1 = prep.binarization(mat1, thres=threshold)
io.save_image(output_base + "/segmented_dots.jpg", mat1)

# Calculate the median dot size and distance between them.
(dot_size, dot_dist) = prep.calc_size_distance(mat1)
```

(continues on next page)

(continued from previous page)

```
# Calculate the slopes of horizontal lines and vertical lines.
hor_slope = prep.calc_hor_slope(mat1)
ver_slope = prep.calc_ver_slope(mat1)
print("Horizontal slope: {0}. Vertical slope: {1}".format(hor_slope, ver_
↪slope))

# Group dots into lines.
list_hor_lines0 = prep.group_dots_hor_lines(mat1, hor_slope, dot_dist,
                                            ratio=0.3, num_dot_miss=2,
                                            accepted_ratio=0.6)
list_ver_lines0 = prep.group_dots_ver_lines(mat1, ver_slope, dot_dist,
                                            ratio=0.3, num_dot_miss=2,
                                            accepted_ratio=0.6)

# Save output for checking
io.save_plot_image(output_base + "/horizontal_lines.png", list_hor_
↪lines0, height, width)
io.save_plot_image(output_base + "/vertical_lines.png", list_ver_lines0,␣
↪height, width)
list_hor_data = post.calc_residual_hor(list_hor_lines0, 0.0, 0.0)
list_ver_data = post.calc_residual_ver(list_ver_lines0, 0.0, 0.0)
io.save_residual_plot(output_base + "/hor_residual_before_correction.png
↪", list_hor_data,
                      height, width)
io.save_residual_plot(output_base + "/ver_residual_before_correction.png
↪", list_ver_data,
                      height, width)
```

- Perspective distortion can be detected by using coefficients of parabolic fits to the horizontal lines and vertical lines (Fig. 59) as presented in the *method section*. The effect of perspective distortion is corrected by adjusting these parabolic coefficients (Fig. 60). Then, they will be used for the step of determining parameters of the radial distortion model.

```
# Optional: for checking perspective distortion
(xcen_tmp, ycen_tmp) = proc.find_cod_bailey(list_hor_lines0, list_ver_
↪lines0)
list_hor_coef = proc._para_fit_hor(list_hor_lines0, xcen_tmp, ycen_
↪tmp)[0]
list_ver_coef = proc._para_fit_ver(list_ver_lines0, xcen_tmp, ycen_
↪tmp)[0]
# Optional: plot the results
plt.figure(0)
plt.plot(list_hor_coef[:, 2], list_hor_coef[:, 0], "-o")
plt.plot(list_ver_coef[:, 2], list_ver_coef[:, 0], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("a-coefficient")

plt.figure(1)
plt.plot(list_hor_coef[:, 2], -list_hor_coef[:, 1], "-o")
plt.plot(list_ver_coef[:, 2], list_ver_coef[:, 1], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("b-coefficient")
plt.show()
# Optional: correct parabola coefficients
hor_coef_corr, ver_coef_corr = proc._generate_non_perspective_parabola_
↪coef(
```

(continues on next page)

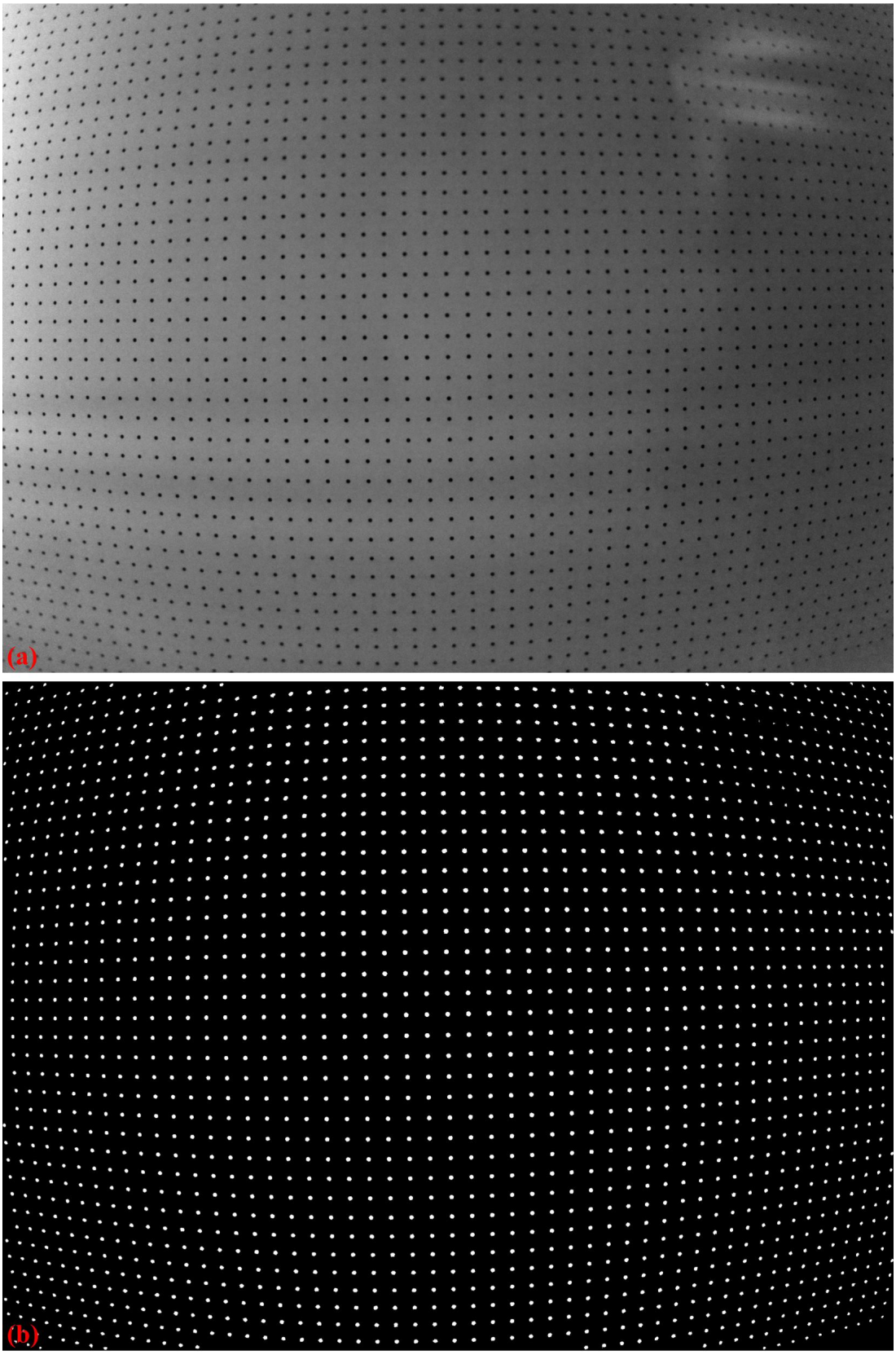


Fig. 56: . (a) Calibration image. (b) Binarized image.

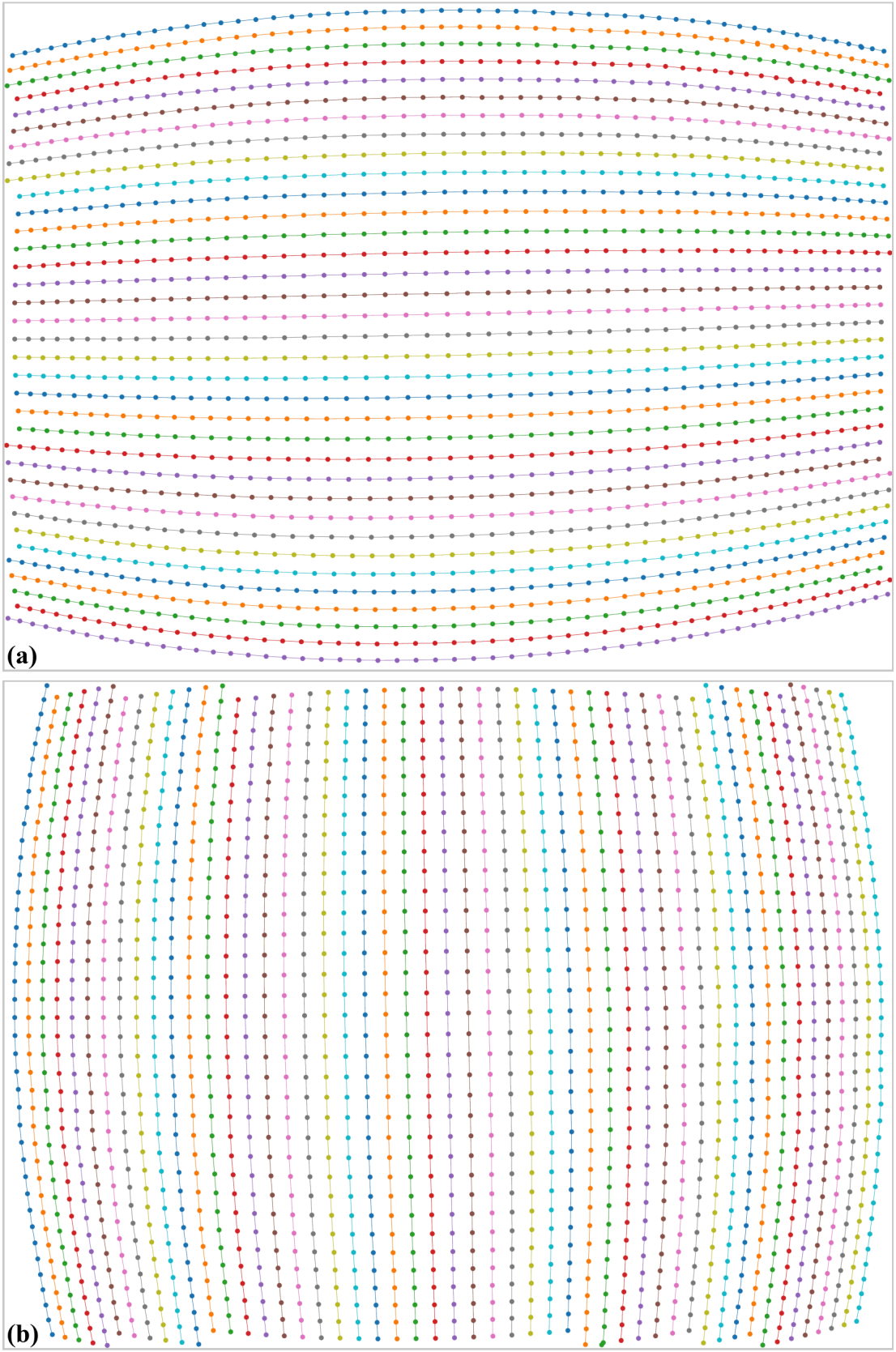


Fig. 57: Grouped points. (a) Horizontal lines. (b) Vertical lines.

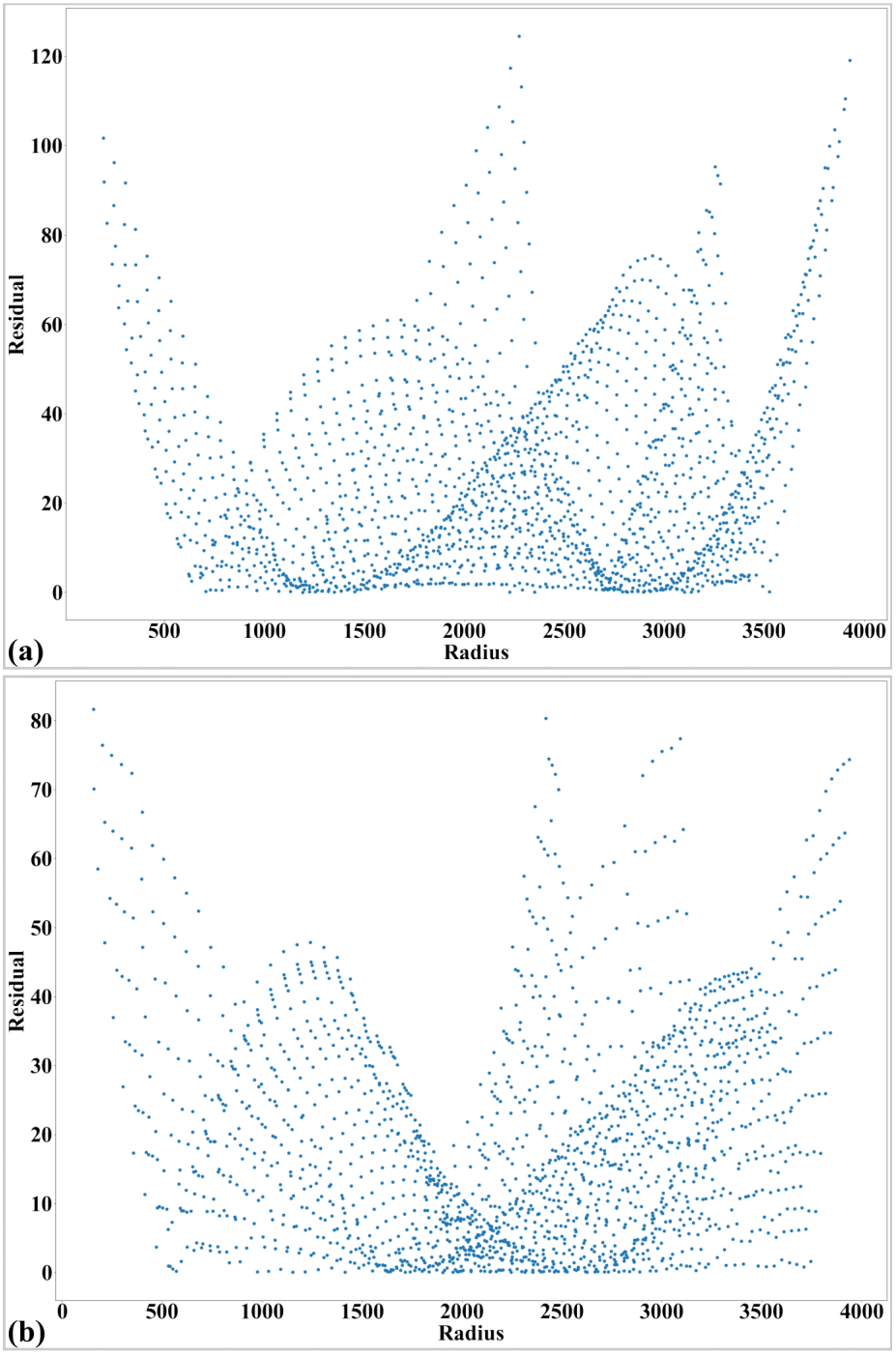


Fig. 58: Residual of distorted points. (a) Horizontal lines. (b) Vertical lines.

(continued from previous page)

```
    list_hor_lines0, list_ver_lines0)[0:2]
# Optional: plot to check the results
plt.figure(0)
plt.plot(hor_coef_corr[:, 2], hor_coef_corr[:, 0], "-o")
plt.plot(ver_coef_corr[:, 2], ver_coef_corr[:, 0], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("a-coefficient")
plt.figure(1)
plt.plot(hor_coef_corr[:, 2], -hor_coef_corr[:, 1], "-o")
plt.plot(ver_coef_corr[:, 2], ver_coef_corr[:, 1], "-o")
plt.xlabel("c-coefficient")
plt.ylabel("b-coefficient")
plt.show()

# Regenerate grid points with the correction of perspective effect.
list_hor_lines1, list_ver_lines1 = proc.regenerate_grid_points_parabola(
    list_hor_lines0, list_ver_lines0, perspective=True)
```

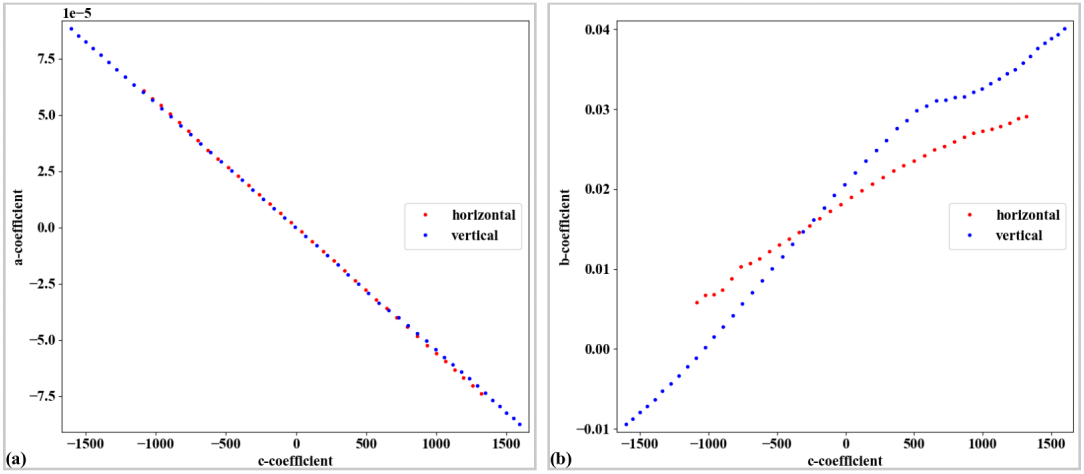


Fig. 59: . (a) Plot of a-coefficients vs c-coefficients of parabolic fits. (b) Plot of b-coefficients vs c-coefficients.

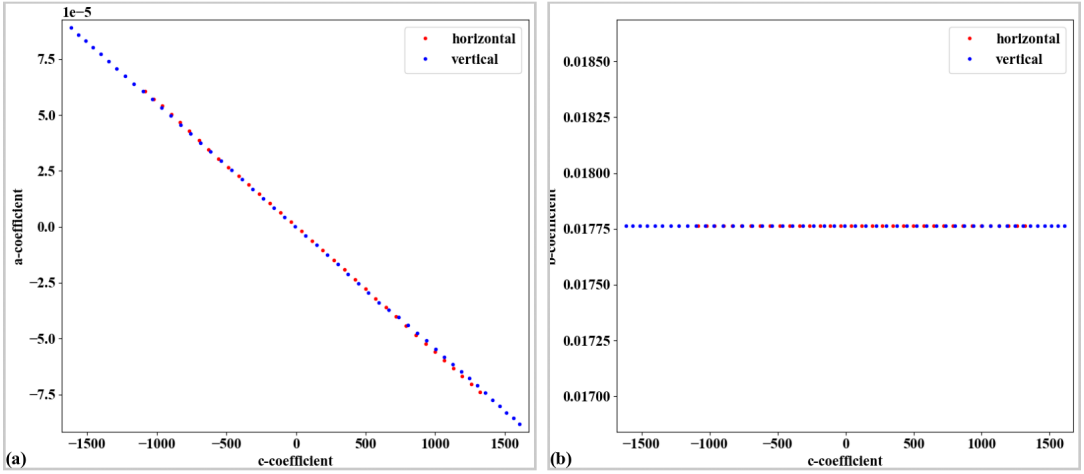


Fig. 60: Parabola coefficients after correction. (a) Plot of a-coefficients vs c-coefficients. (b) Plot of b-coefficients vs c-coefficients.

- Parameters of the radial distortion correction are determined and the image is corrected. At this step only radial distortion correction is applied (Fig. 62 (a)). For checking the accuracy of the model, noting that un-

warping lines using the backward model is based on optimization [R4] which may result in strong fluctuation if lines are strongly curved. In such case, using the forward model is more reliable.

```
# Calculate parameters of the radial correction model
(xcenter, ycenter) = proc.find_cod_coarse(list_hor_lines1, list_ver_
↪lines1)
list_fact = proc.calc_coef_backward(list_hor_lines1, list_ver_lines1,
                                    xcenter, ycenter, num_coef)
io.save_metadata_txt(output_base + "/coefficients_radial_distortion.txt",
                     xcenter, ycenter, list_fact)
print("X-center: {0}. Y-center: {1}".format(xcenter, ycenter))
print("Coefficients: {0}".format(list_fact))

# Regenerate the lines without perspective correction for later use.
list_hor_lines2, list_ver_lines2 = proc.regenerate_grid_points_parabola(
    list_hor_lines0, list_ver_lines0, perspective=False)

# Unwarp lines using the backward model:
list_uhor_lines = post.unwarp_line_backward(list_hor_lines2, xcenter,␣
↪ycenter, list_fact)
list_uver_lines = post.unwarp_line_backward(list_ver_lines2, xcenter,␣
↪ycenter, list_fact)

# Optional: unwarp lines using the forward model.
# list_ffact = proc.calc_coef_forward(list_hor_lines1, list_ver_lines1,
#                                     xcenter, ycenter, num_coef)
# list_uhor_lines = post.unwarp_line_forward(list_hor_lines2, xcenter,␣
↪ycenter,
#                                            list_ffact)
# list_uver_lines = post.unwarp_line_forward(list_ver_lines2, xcenter,␣
↪ycenter,
#                                            list_ffact)

# Check the residual of unwarped lines:
list_hor_data = post.calc_residual_hor(list_uhor_lines, xcenter, ycenter)
list_ver_data = post.calc_residual_ver(list_uver_lines, xcenter, ycenter)
io.save_residual_plot(output_base + "/hor_residual_after_correction.png",
↪ list_hor_data,
                      height, width)
io.save_residual_plot(output_base + "/ver_residual_after_correction.png",
↪ list_ver_data,
                      height, width)

# Unwarp the image
mat_rad_corr = post.unwarp_image_backward(mat0, xcenter, ycenter, list_
↪fact)
# Save results
io.save_image(output_base + "/image_radial_corrected.jpg", mat_rad_corr)
io.save_image(output_base + "/radial_difference.jpg", mat_rad_corr -␣
↪mat0)
```

- Perspective distortion can be caused by the calibration-object-plane not parallel to the *camera-sensor plane*. In such case, determining the radial distortion coefficients is enough. However, if the distortion is caused by the lens-plane (tangential distortion), we need to correct for this type of distortion, too. This is straightforward using Discorpy's API.

```
# Generate source points and target points to calculate coefficients of␣
↪a perspective model
```

(continues on next page)

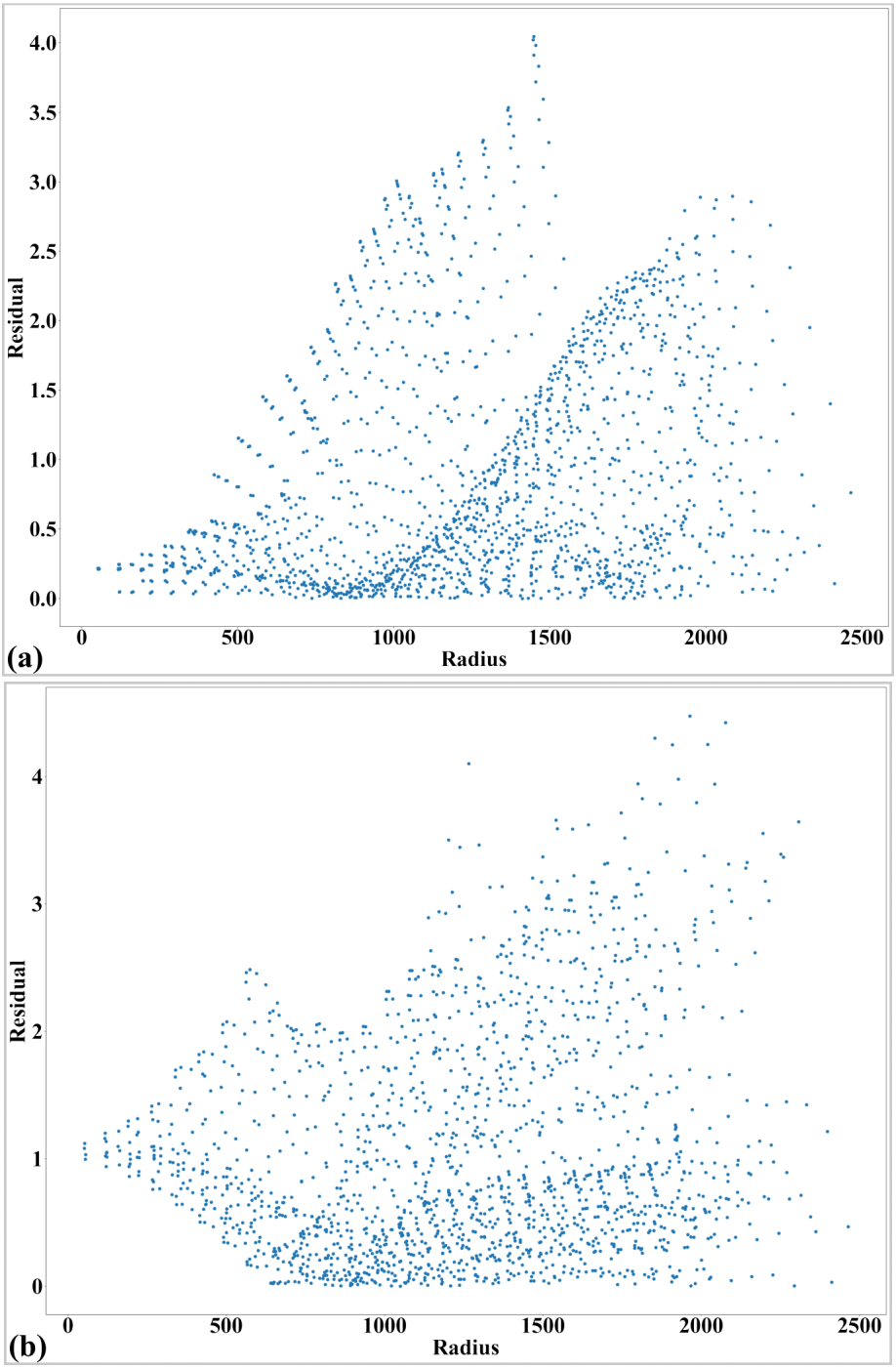


Fig. 61: Residual of unwarped points in the horizontal lines (a) and vertical lines (b). Note that this is from a commercial camera, the accuracy may not be at sub-pixel as compared to scientific cameras used in *demo 1-4*.

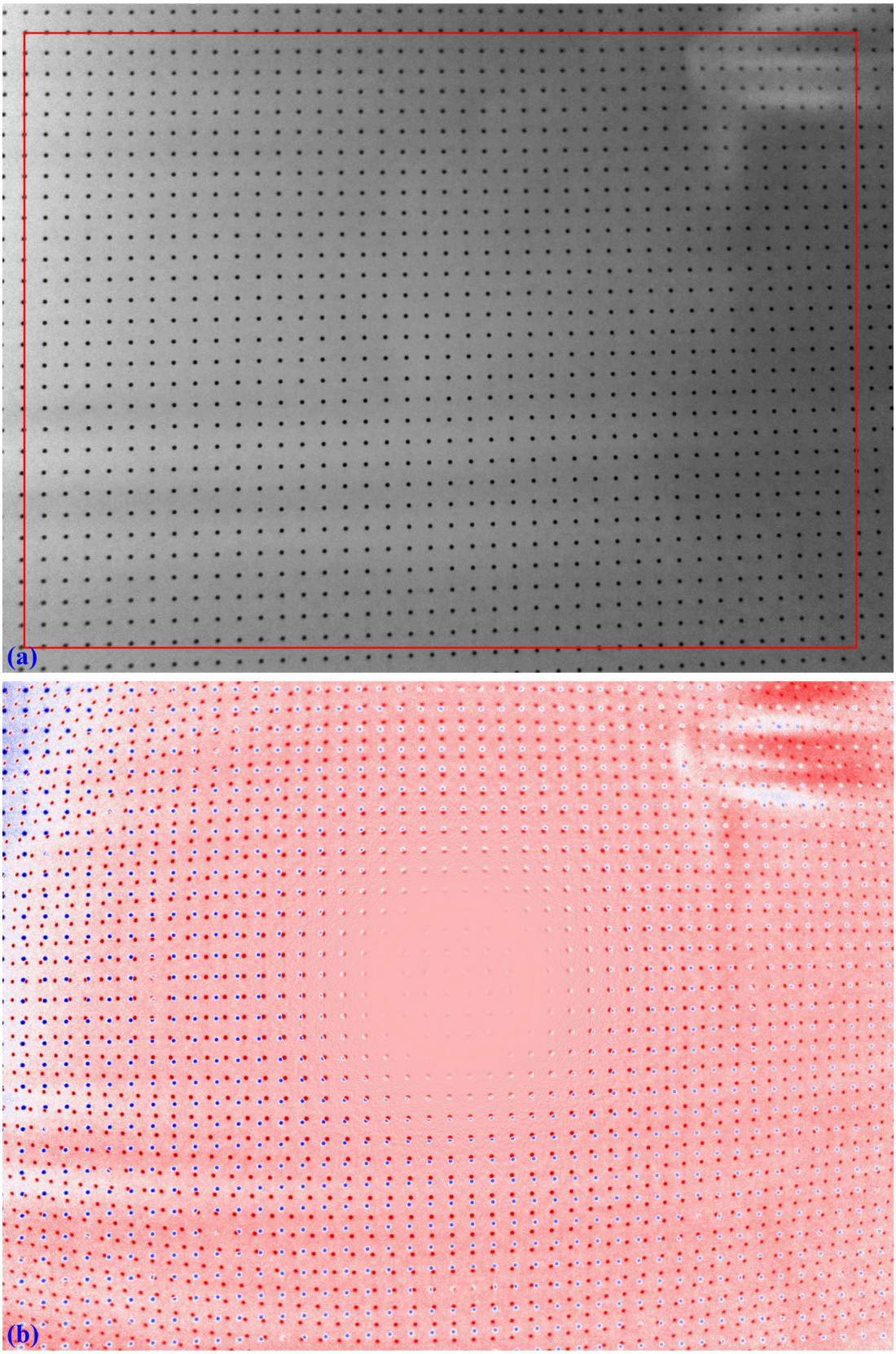


Fig. 62: . (a) Unwarped image where perspective distortion can be seen. (b) Difference between image in Fig. 62 (a) and image in Fig. 56 (a).

(continued from previous page)

```
source_points, target_points = proc.generate_source_target_perspective_
↪points(list_uhor_lines, list_uver_lines,
                                                                        ␣
↪      equal_dist=True, scale="mean",
                                                                        ␣
↪      optimizing=False)
# Calculate perspective coefficients:
pers_coef = proc.calc_perspective_coefficients(source_points, target_
↪points, mapping="backward")
image_pers_corr = post.correct_perspective_image(mat_rad_corr, pers_coef)
# Save results
np.savetxt(output_base + "/perspective_coefficients.txt", np.
↪transpose([pers_coef]))
io.save_image(output_base + "/image_radial_perspective_corrected.jpg",␣
↪image_pers_corr)
io.save_image(output_base + "/perspective_difference.jpg", image_pers_
↪corr - mat_rad_corr)
```

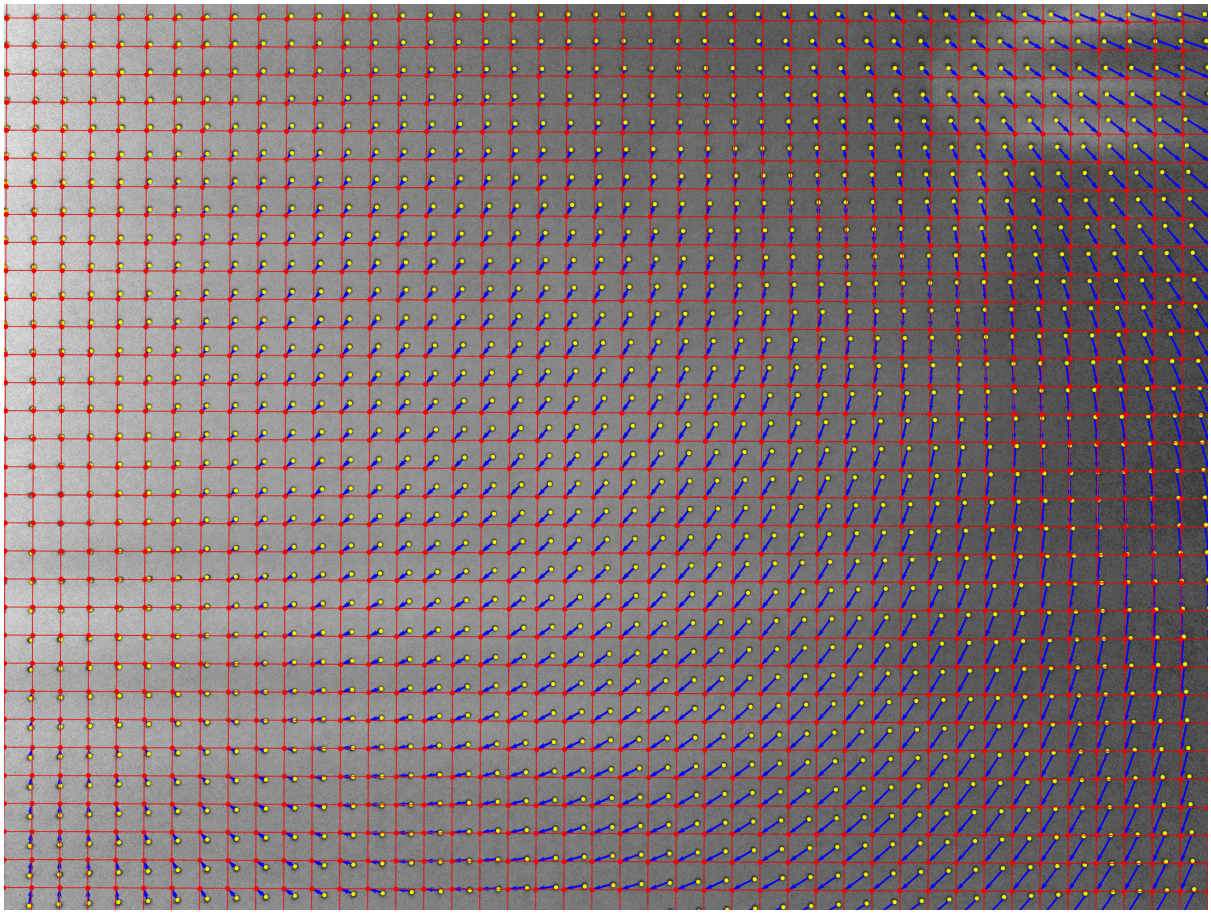


Fig. 63: Image showing source-points and targets-points used to calculate perspective-distortion coefficients

Click here to download the Python codes.

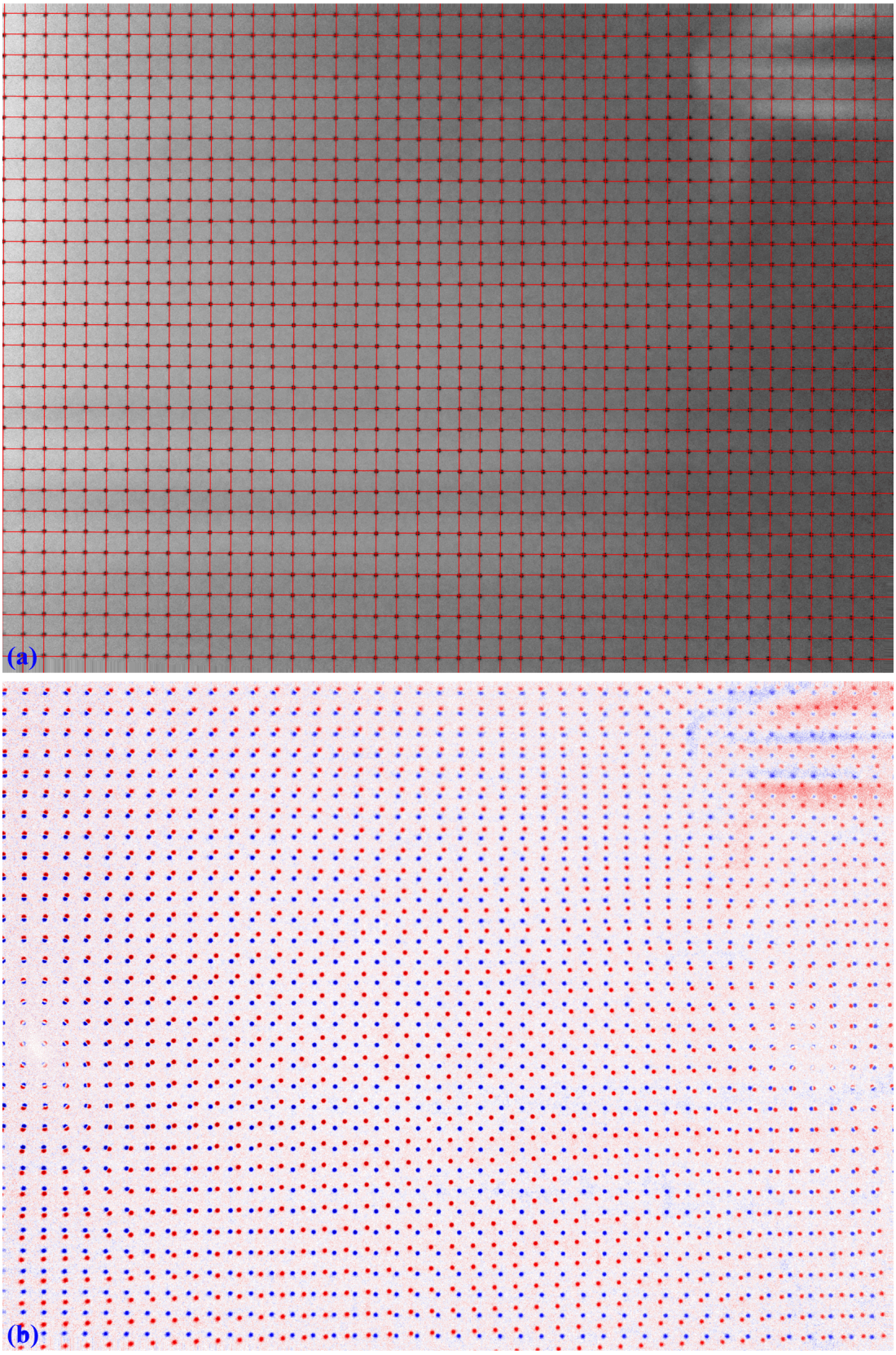


Fig. 64: . (a) Perspective corrected image. (b) Difference between image in Fig. 64 (a) and image in Fig. 62 (a).

Calibrate a camera using a chessboard image

The following workflow shows how to calibrate a commercial camera using a chessboard image. First of all, the chessboard pattern was created using this website. The generated pattern was printed and stuck to a flat wall (Fig. 65). Then, its image was taken using the front-facing camera of a Microsoft Surface Pro Laptop. The pincushion distortion is visible in the image.

- Load the chessboard image, convert to a line-pattern image.

```
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.prep.preprocessing as prep
import discorpy.prep.linepattern as lprep
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

# Initial parameters
file_path = "../../data/laptop_camera/chessboard.jpg"
output_base = "E:/output_demo_06/"
num_coef = 5  # Number of polynomial coefficients
mat0 = io.load_image(file_path) # Load image
(height, width) = mat0.shape

# Convert the chessboard image to a line-pattern image
mat1 = lprep.convert_chessboard_to_linepattern(mat0)
io.save_image(output_base + "/line_pattern_converted.jpg", mat1)

# Calculate slope and distance between lines
slope_hor, dist_hor = lprep.calc_slope_distance_hor_lines(mat1,␣
↪radius=15, sensitive=0.5)
slope_ver, dist_ver = lprep.calc_slope_distance_ver_lines(mat1,␣
↪radius=15, sensitive=0.5)
print("Horizontal slope: ", slope_hor, " Distance: ", dist_hor)
print("Vertical slope: ", slope_ver, " Distance: ", dist_ver)
```

- Extract reference-points (adjust parameters: *radius* and/or *sensitive* if need to). Note that users can use other methods available in Scikit-image to perform this step directly on the chessboard image. There are many unwanted points (Fig. 66) but they can be removed by the grouping method (Fig. 67) in Discorpy.

```
# Extract reference-points
list_points_hor_lines = lprep.get_cross_points_hor_lines(mat1, slope_ver,
↪ dist_ver,
                                                         ratio=0.3,␣
↪norm=True, offset=450,
                                                         bgr="bright",␣
↪radius=15,
                                                         sensitive=0.5,␣
↪denoise=True,
                                                         subpixel=True)
list_points_ver_lines = lprep.get_cross_points_ver_lines(mat1, slope_hor,
↪ dist_hor,
                                                         ratio=0.3,␣
↪norm=True, offset=150,
                                                         bgr="bright",␣
↪radius=15,
                                                         sensitive=0.5,␣
↪denoise=True,
                                                         subpixel=True)
```

(continues on next page)

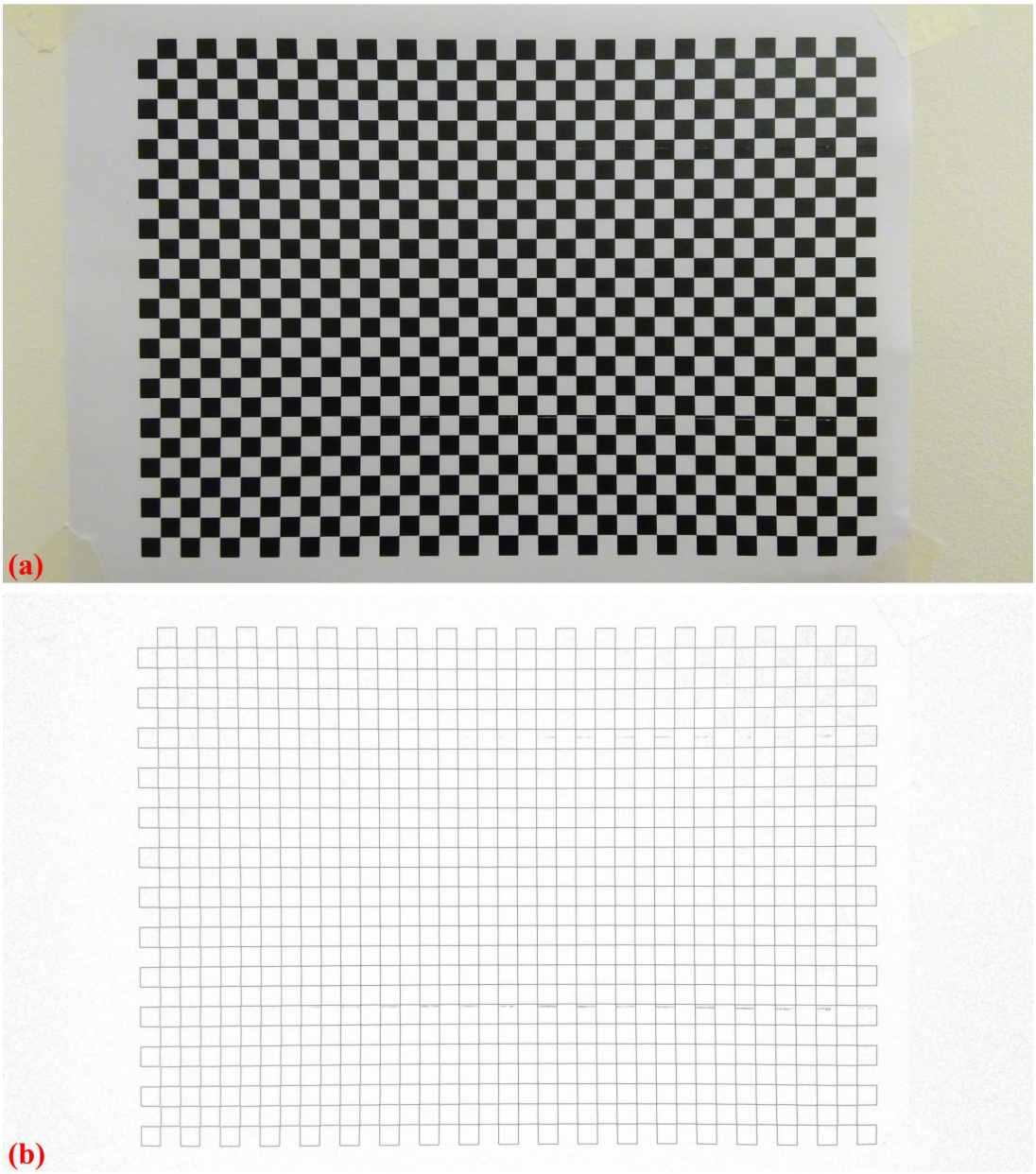


Fig. 65: . (a) Chessboard image. (b) Line-pattern image generated from image (a).

(continued from previous page)

```
if len(list_points_hor_lines) == 0 or len(list_points_ver_lines) == 0:
    raise ValueError("No reference-points detected !!! Please adjust␣
↪parameters !!!")
io.save_plot_points(output_base + "/ref_points_horizontal.png", list_
↪points_hor_lines,
                    height, width, color="red")
io.save_plot_points(output_base + "/ref_points_vertical.png", list_
↪points_ver_lines,
                    height, width, color="blue")

# Group points into lines
list_hor_lines = prep.group_dots_hor_lines(list_points_hor_lines, slope_
↪hor, dist_hor,
                                           ratio=0.1, num_dot_miss=2,␣
↪accepted_ratio=0.8)
list_ver_lines = prep.group_dots_ver_lines(list_points_ver_lines, slope_
↪ver, dist_ver,
                                           ratio=0.1, num_dot_miss=2,␣
↪accepted_ratio=0.8)

# Remove residual dots
list_hor_lines = prep.remove_residual_dots_hor(list_hor_lines, slope_hor,
↪ 2.0)
list_ver_lines = prep.remove_residual_dots_ver(list_ver_lines, slope_ver,
↪ 2.0)

# Save output for checking
io.save_plot_image(output_base + "/horizontal_lines.png", list_hor_lines,
↪ height, width)
io.save_plot_image(output_base + "/vertical_lines.png", list_ver_lines,␣
↪height, width)
list_hor_data = post.calc_residual_hor(list_hor_lines, 0.0, 0.0)
list_ver_data = post.calc_residual_ver(list_ver_lines, 0.0, 0.0)
io.save_residual_plot(output_base + "/hor_residual_before_correction.png
↪",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_before_correction.png
↪",
                      list_ver_data, height, width)
```

- Next steps are straightforward. Coefficients of the radial-distortion model are calculated where the perspective effect is *corrected* before that. The results of applying the model for unwarping lines and images can be seen in Fig. 69 and Fig. 70

```
# Regenerate grid points after correcting the perspective effect.
list_hor_lines, list_ver_lines = proc.regenerate_grid_points_parabola(
    list_hor_lines, list_ver_lines, perspective=True)

# Calculate parameters of the radial correction model
(xcenter, ycenter) = proc.find_cod_coarse(list_hor_lines, list_ver_lines)
list_fact = proc.calc_coef_backward(list_hor_lines, list_ver_lines,
                                    xcenter, ycenter, num_coef)
io.save_metadata_txt(output_base + "/coefficients_radial_distortion.txt",
                     xcenter, ycenter, list_fact)
print("X-center: {0}. Y-center: {1}".format(xcenter, ycenter))
print("Coefficients: {0}".format(list_fact))
```

(continues on next page)

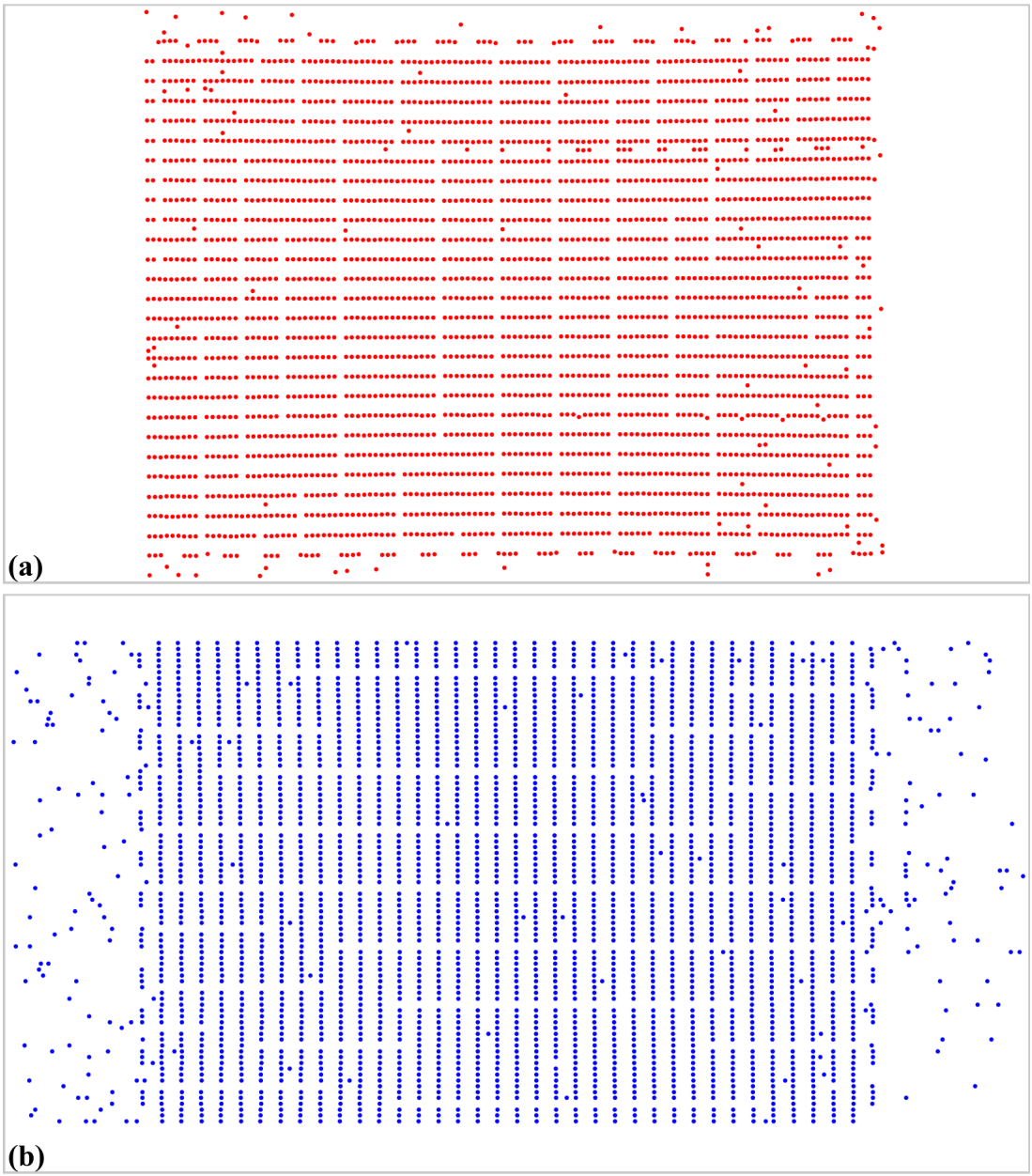


Fig. 66: Extracted reference points from Fig. 65 (b). (a) For horizontal lines. (b) For vertical lines.

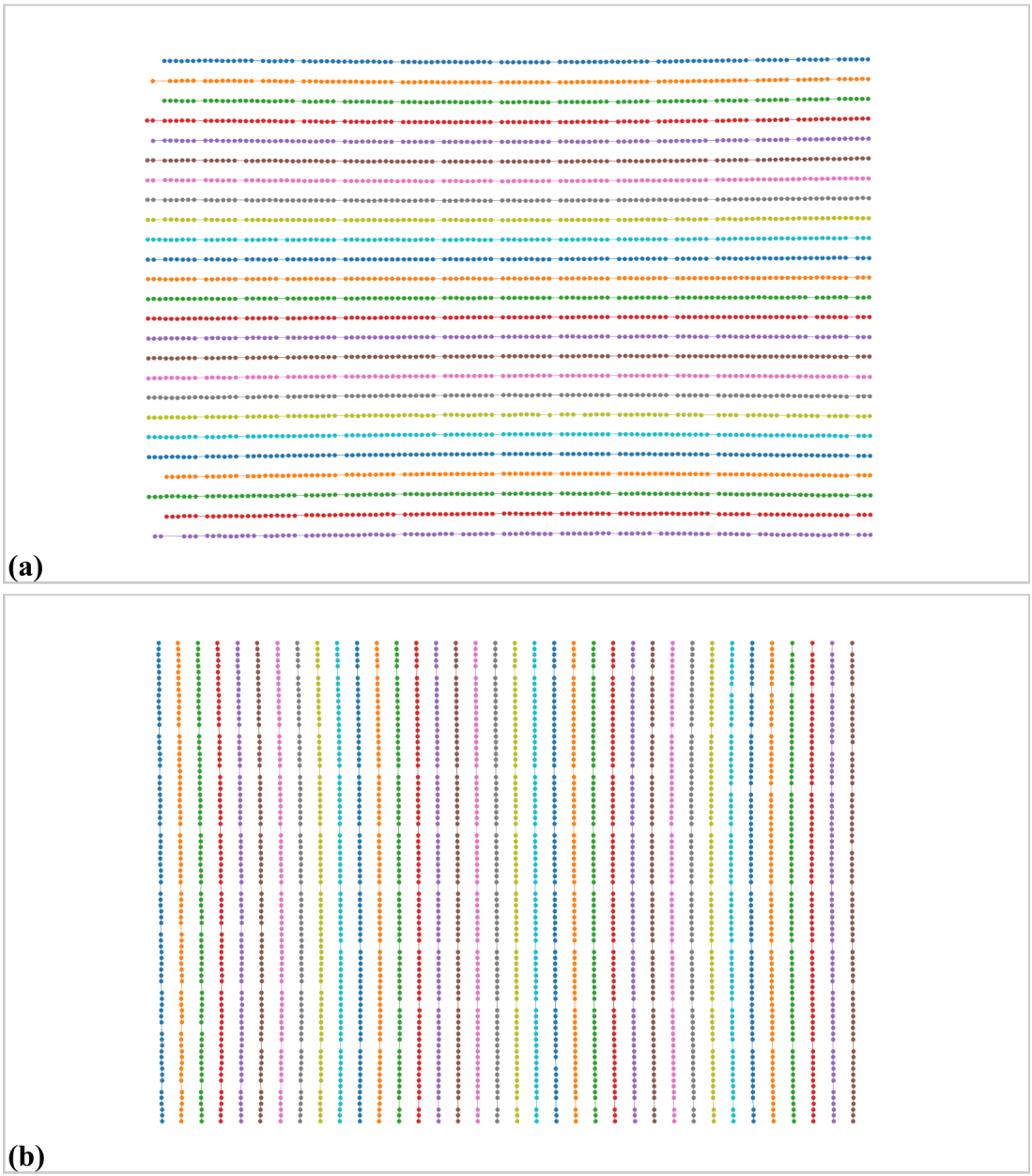


Fig. 67: Grouped points. (a) Horizontal lines. (b) Vertical lines.

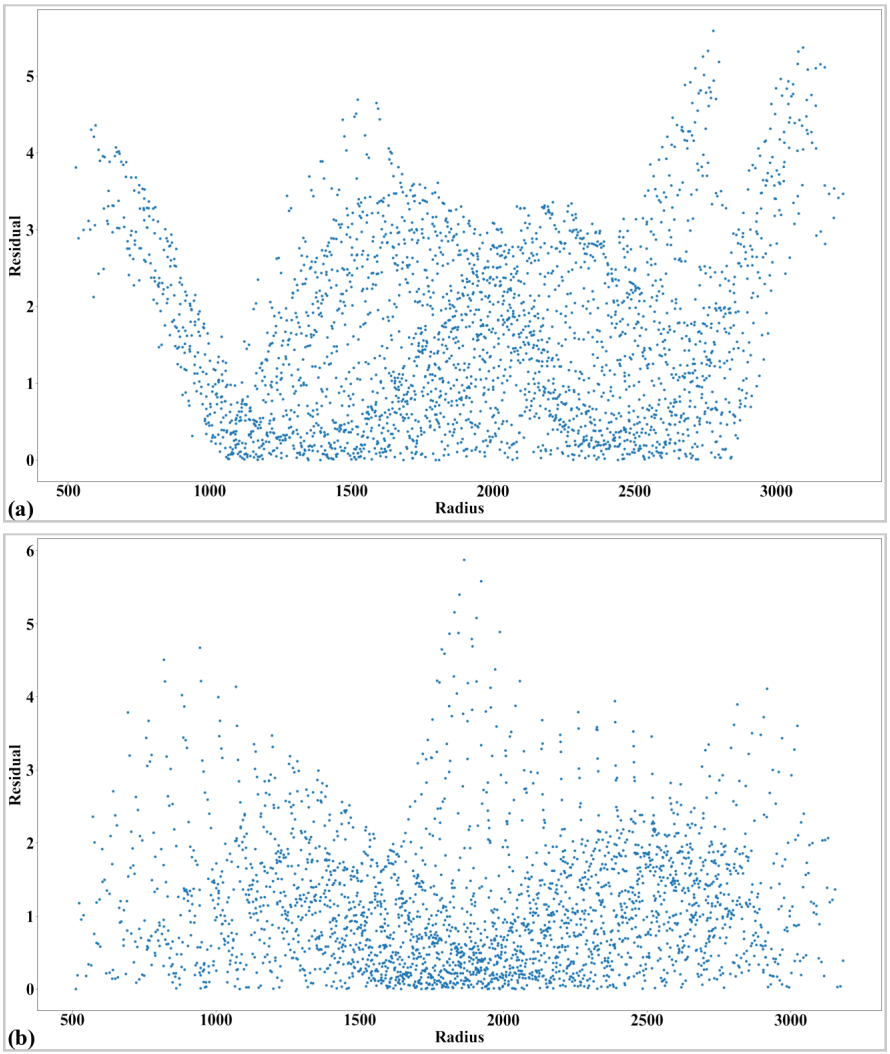


Fig. 68: Residual of distorted points. (a) Horizontal lines. (b) Vertical lines.

(continued from previous page)

```
# Check the correction results:
# Apply correction to the lines of points
list_uhor_lines = post.unwarp_line_backward(list_hor_lines, xcenter,␣
↪ycenter,
                                            list_fact)
list_uver_lines = post.unwarp_line_backward(list_ver_lines, xcenter,␣
↪ycenter,
                                            list_fact)
# Calculate the residual of the unwarpped points.
list_hor_data = post.calc_residual_hor(list_uhor_lines, xcenter, ycenter)
list_ver_data = post.calc_residual_ver(list_uver_lines, xcenter, ycenter)
# Save the results for checking
io.save_plot_image(output_base + "/unwarpped_horizontal_lines.png",
                   list_uhor_lines, height, width)
io.save_plot_image(output_base + "/unwarpped_vertical_lines.png",
                   list_uver_lines, height, width)
io.save_residual_plot(output_base + "/hor_residual_after_correction.png",
                      list_hor_data, height, width)
io.save_residual_plot(output_base + "/ver_residual_after_correction.png",
                      list_ver_data, height, width)
```

- Calculated coefficients of the correction model can be used to unwarp another image taken by the same camera as demonstrated in Fig. 71. For a color image, we have to correct each channel of the image.

```
# Load coefficients from previous calculation
(xcenter, ycenter, list_fact) = io.load_metadata_txt(
    output_base + "/coefficients_radial_distortion.txt")

# Load an image and correct it.
img = io.load_image("../../data/laptop_camera/test_image.jpg",␣
↪average=False)
img_corrected = np.copy(img)
for i in range(img.shape[-1]):
    img_corrected[:, :, i] = post.unwarp_image_backward(img[:, :, i],␣
↪xcenter,
                                                        ycenter, list_
↪fact)
io.save_image(output_base + "/test_image_corrected.jpg", img_corrected)
```

Click `here` to download the Python codes.

Correct perspective distortion

Discorpy can be used to correct perspective distortion of an image, *e.g.* to read information from the image or to make a document captured by a camera look like a scanned one.

- Firstly, we need to locate 4 points from the image knowing that they should be at 4 corners of a rectangular shape. This can be done using ImageJ software as shown in Fig. 72 below. Or, it can be done in a more fancy way using line-detection methods and finding cross-points between these lines using Scikit-image and Discorpy's API.

```
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post
```

(continues on next page)

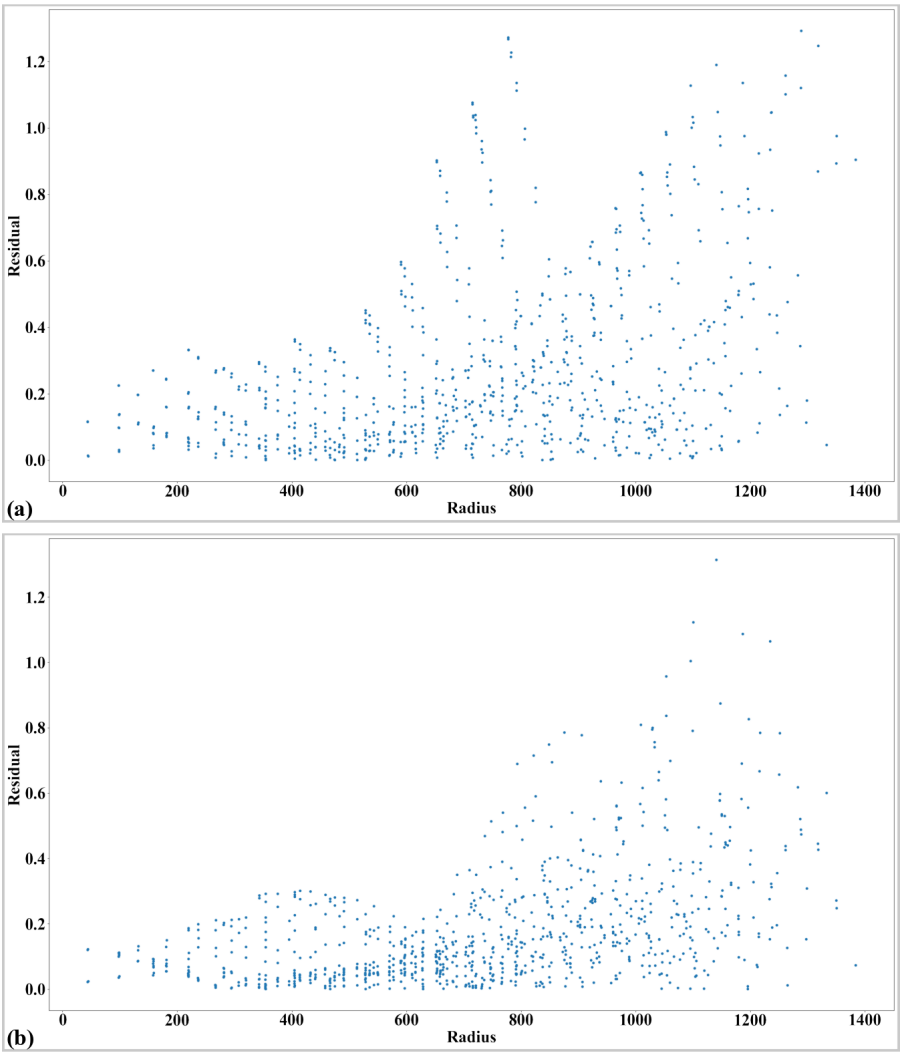


Fig. 69: Residual of unwarped points. (a) Horizontal lines. (b) Vertical lines.

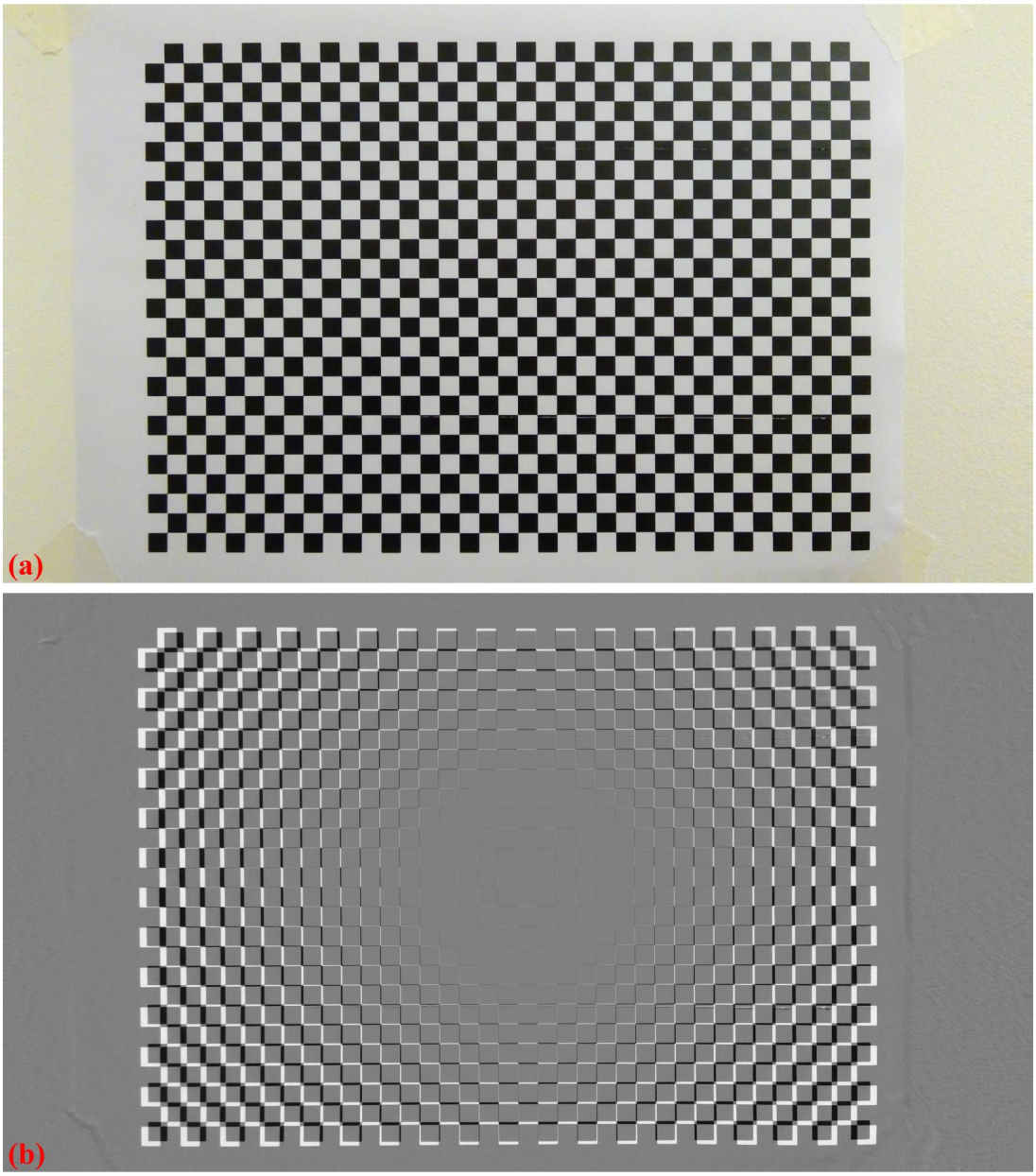


Fig. 70: . (a) Unwarped image of Fig. 65 (a). (b) Difference between the images before and after unwarping.

Fig. 71: . (a) Test image taken from the same camera. (b) Unwarped image. The red straight line is added for reference.

(continued from previous page)

```
# Load image
file_path = "../../data/demo/perspective_demo.jpg"
output_base = "./output_demo_07/"
mat = io.load_image(file_path, average=False)

# Provide the coordinates of 4-points. They can be in xy-order or yx-
↪order, this info
# needs to be consistent with other functions. In this example, it's in␣
↪the xy-order.
list_points = [[180, 1920], [1500, 1602], [2754, 2430], [942, 3246]]
```

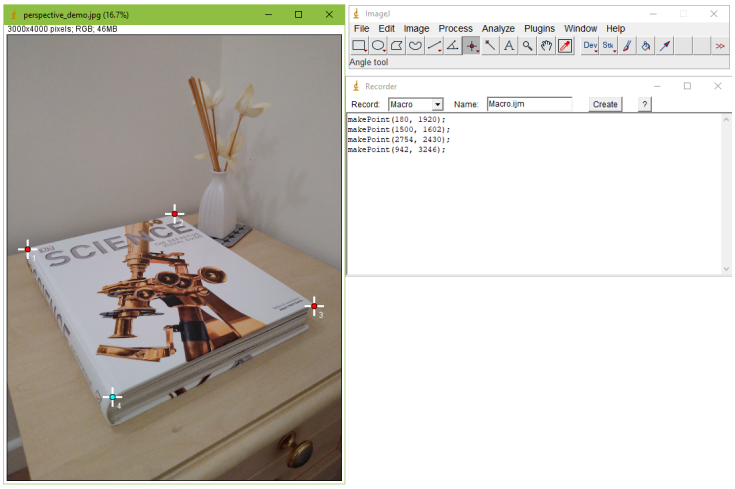


Fig. 72: Reference-points selected using ImageJ software.

- Target points are generated using Discorpy's API, then perspective coefficients are calculated and correcting to the image is applied.

```
# Generate undistorted points. Note that the output coordinate is in the␣
↪yx-order.
s_points, t_points = proc.generate_4_source_target_perspective_points(
    list_points, input_order="xy", scale="mean", equal_dist=False)

# Calculate distortion coefficients
list_coef = proc.calc_perspective_coefficients(s_points, t_points,
                                               mapping="backward")
# Apply correction.
mat_cor = np.zeros_like(mat)
for i in range(mat_cor.shape[-1]):
    mat_cor[:, :, i] = post.correct_perspective_image(mat[:, :, i], list_
↪coef)
io.save_image(output_base + "/corrected_image.jpg", mat_cor)
```

- As can be seen in Fig. 73, the region of interest is out of the field of view and rotated. We can rotate the output image, offset, and scale it by changing the target points as follows.

```
rotate_angle = 35.0
x_offset = 2600
y_offset = -1000
scale = 1.5
```

(continues on next page)

Fig. 73: Perspective corrected image.

(continued from previous page)

```
# Apply rotating
x = t_points[:, 1]
y = t_points[:, 0]
a = np.deg2rad(rotate_angle)
x_rot = x * np.cos(a) - y * np.sin(a)
y_rot = x * np.sin(a) + y * np.cos(a)

# Apply scaling
x_rot = x_rot * scale
y_rot = y_rot * scale

# Apply translating
x_rot = x_rot + x_offset
y_rot = y_rot + y_offset

# Update target points
t_points2 = np.asarray(list(zip(y_rot, x_rot)))

# Calculate coefficients
list_coef2 = proc.calc_perspective_coefficients(s_points, t_points2,␣
↪mapping="backward")
# Correct the image.
mat_cor = np.zeros_like(mat)
for i in range(mat_cor.shape[-1]):
    mat_cor[:, :, i] = post.correct_perspective_image(mat[:, :, i], list_
↪coef2, order=3)
io.save_image(output_base + "/adjusted_image.jpg", mat_cor)
```

Click here to download the Python codes.

Correct radial distortion of an image without using a calibration target

In the previous demos, distortion coefficients are determined by using a calibration target. In practice, we may have to correct radial distortion of an image but don't have the calibration image taken from the same camera. The following workflow shows how to do that on images acquired by the front-right hazard-camera of the Percy Rover

- Load the image, create a line-pattern for visual inspection (Fig. 75). The idea is that we apply an estimated forward model to the line-pattern and overlay the result on top of the image.

```
import os
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post
import scipy.ndimage as ndi


file_path = "../../data/percy_cam/F_R_hazcam.png"
output_base = "E:/output_demo_08/"
mat0 = io.load_image(file_path, average=True)
mat0 = mat0 / np.max(mat0)
(height, width) = mat0.shape

# Create a line-pattern image
line_pattern = np.zeros((height, width), dtype=np.float32)
for i in range(50, height - 50, 40):
```

(continues on next page)

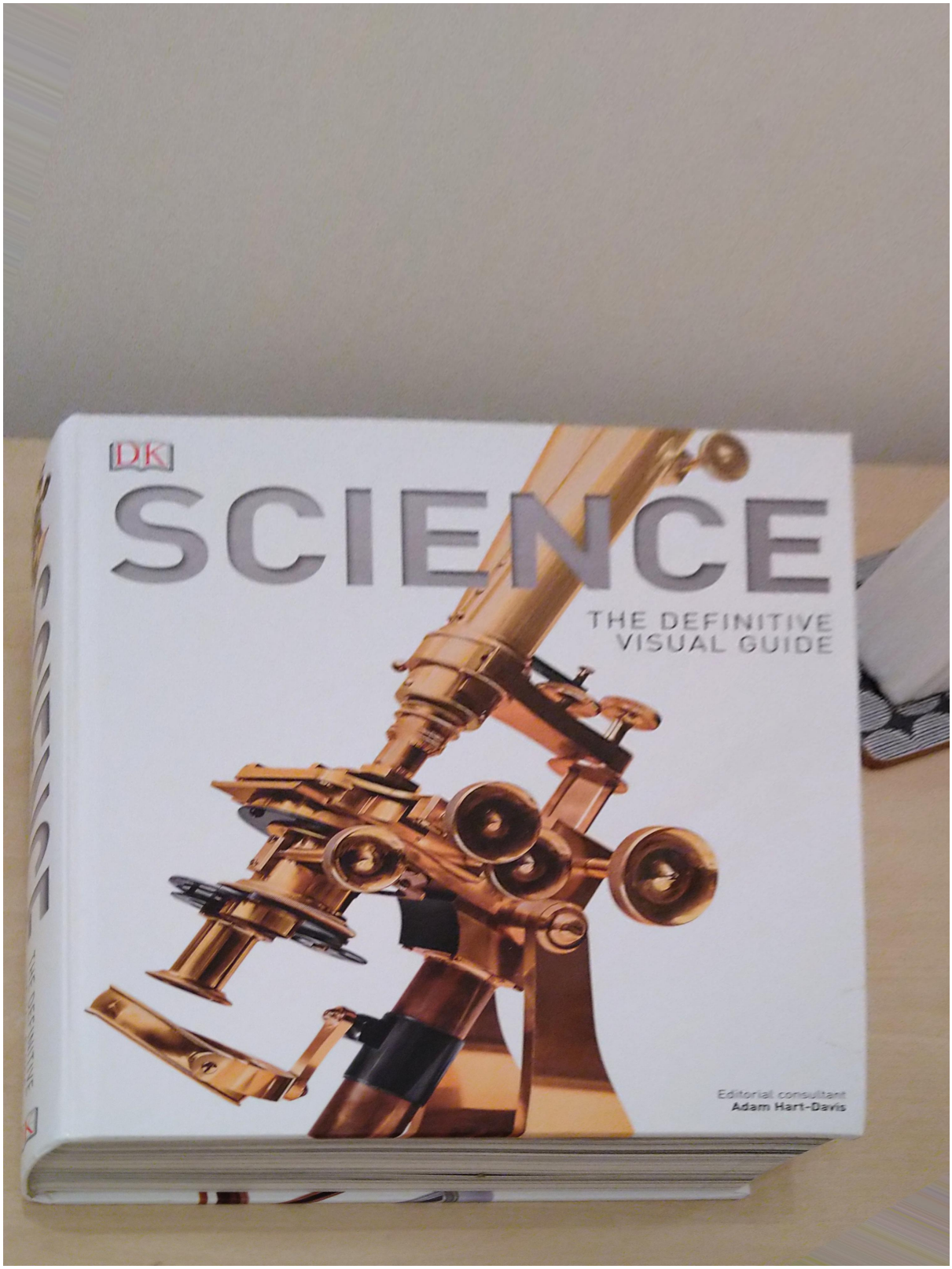


Fig. 74: Image after applying the scaling, rotating, and translating operation.

(continued from previous page)

```
    line_pattern[i - 2:i + 3] = 1.0

# Estimate parameters by visual inspection:
# Coarse estimation
xcenter = width // 2
ycenter = height // 2
list_power = np.asarray([1.0, 10**(-4), 10**(-7), 10**(-10), 10**(-13)])
list_coef = np.asarray([1.0, 1.0, 1.0, 1.0, 1.0])

# Rotate the line-pattern image if need to
angle = 2.0 # Degree
pad = width // 2 # Need padding as lines are shrunk after warping.
mat_pad = np.pad(line_pattern, pad, mode='edge')
mat_pad = ndi.rotate(mat_pad, angle, reshape=False)
```

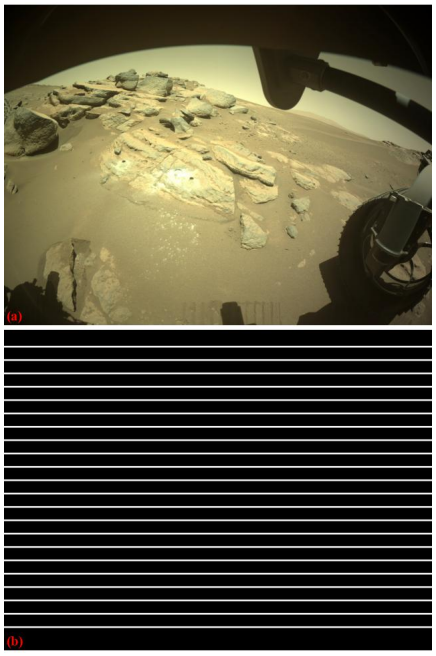


Fig. 75: . (a) Image from the Percy Rover. (b) Generated line-pattern.

- The following codes show how to adjust parameters of a forward model, apply to the line-pattern, overlay to the Percy's image, and make sure the warped lines are matching with the curve feature in the Percy's image. If there are lines in the image which we can be certain that they are straight, an automated method can be developed to extract these lines and match with warped lines from the line-pattern. For the currently used image, we simply use visual inspection (Fig. 76).

```
# Define scanning routines
def scan_coef(idx, start, stop, step, list_coef, list_power, output_
↪base0, mat0,
              mat_pad, pad, ntime=1, backward=True):
    output_base = output_base0 + "/coef_" + str(idx) + "_ntime_" +␣
↪str(ntime)
    while os.path.isdir(output_base):
        ntime = ntime + 1
        output_base = output_base0 + "/coef_" + str(idx) + "_ntime_" +␣
↪str(ntime)
    (height, width) = mat0.shape
    for num in np.arange(start, stop + step, step):
```

(continues on next page)

(continued from previous page)

```
        list_coef_est = np.copy(list_coef)
        list_coef_est[idx] = list_coef_est[idx] + num
        list_ffact = list_power * list_coef_est
        line_img_warped = post.unwarp_image_backward(mat_pad, xcenter +␣
↪pad,
                                                     ycenter + pad, list_
↪ffact)
        line_img_warped = line_img_warped[pad:pad + height, pad:pad +␣
↪width]
        name = "coef_{0}_val_{1:4.2f}".format(idx, list_coef_est[idx])
        io.save_image(output_base + "/forward/img_" + name + ".jpg",␣
↪mat0 + 0.5 * line_img_warped)
        if backward is True:
            # Transform to the backward model for correction
            hlines = np.int16(np.linspace(0, height, 40))
            vlines = np.int16(np.linspace(0, width, 50))
            ref_points = [[i - ycenter, j - xcenter] for i in hlines for␣
↪j in vlines]
            list_bfact = proc.transform_coef_backward_and_forward(list_
↪ffact, ref_points=ref_points)
            img_unwarped = post.unwarp_image_backward(mat0, xcenter,␣
↪ycenter, list_bfact)
            io.save_image(output_base + "/backward/img_" + name + ".jpg",
↪ img_unwarped)

def scan_center(xcenter, ycenter, start, stop, step, list_coef, list_
↪power,
                output_base0, mat0, mat_pad, pad, axis="x", ntime=1,␣
↪backward=True):
    output_base = output_base0 + "/" + axis + "_center" + "_ntime_" +␣
↪str(ntime)
    while os.path.isdir(output_base):
        ntime = ntime + 1
        output_base = output_base0 + "/" + axis + "_center" + "_ntime_"␣
↪+ str(ntime)
    (height, width) = mat0.shape
    list_ffact = list_power * list_coef
    if axis == "x":
        for num in np.arange(start, stop + step, step):
            line_img_warped = post.unwarp_image_backward(mat_pad,
                                                         xcenter + num +␣
↪pad,
                                                         ycenter + pad,
                                                         list_ffact)
            line_img_warped = line_img_warped[pad:pad + height, pad:pad␣
↪+ width]
            name = "xcenter_{0:7.2f}".format(xcenter + num)
            io.save_image(output_base + "/forward/img_" + name + ".jpg",␣
↪mat0 + 0.5 * line_img_warped)
            if backward is True:
                # Transform to the backward model for correction
                hlines = np.int16(np.linspace(0, height, 40))
                vlines = np.int16(np.linspace(0, width, 50))
                ref_points = [[i - ycenter, j - xcenter] for i in hlines␣
↪for j in vlines]
                list_bfact = proc.transform_coef_backward_and_
```

(continues on next page)

(continued from previous page)

```
↪forward(list_ffact, ref_points=ref_points)
                img_unwarped = post.unwarp_image_backward(mat0,␣
↪xcenter+num, ycenter, list_bfact)
                io.save_image(output_base + "/backward/img_" + name + ".
↪jpg",  img_unwarped)
    else:
        for num in np.arange(start, stop + step, step):
            line_img_warped = post.unwarp_image_backward(mat_pad,␣
↪xcenter + pad,
                                                         ycenter + num +␣
↪pad,
                                                         list_ffact)
            line_img_warped = line_img_warped[pad:pad + height, pad:pad␣
↪+ width]
            name = "ycenter_{0:7.2f}".format(ycenter + num)
            io.save_image(output_base + "/forward/img_" + name + ".jpg",␣
↪mat0 + 0.5 * line_img_warped)
            if backward is True:
                # Transform to the backward model for correction
                hlines = np.int16(np.linspace(0, height, 40))
                vlines = np.int16(np.linspace(0, width, 50))
                ref_points = [[i - ycenter, j - xcenter] for i in hlines␣
↪for j in vlines]
                list_bfact = proc.transform_coef_backward_and_
↪forward(list_ffact, ref_points=ref_points)
                img_unwarped = post.unwarp_image_backward(mat0, xcenter,␣
↪ycenter+num, list_bfact)
                io.save_image(output_base + "/backward/img_" + name + ".
↪jpg",  img_unwarped)


## Scan the 4th coefficient
scan_coef(4, 0, 30, 1, list_coef, list_power, output_base, mat0, mat_pad,
↪ pad)
## The value of 24.0 is good, update the 4th coefficient.
list_coef[4] = 24.0

## Scan the 3rd coefficient
scan_coef(3, 0, 10, 1, list_coef, list_power, output_base, mat0, mat_pad,
↪ pad)
## The value of 2.0 is good, update the 3rd coefficient.
list_coef[3] = 2.0

## Scan the 2nd coefficient
scan_coef(2, 0, 10, 1, list_coef, list_power, output_base, mat0, mat_pad,
↪ pad)
## The value of 5.0 is good, update the 2nd coefficient.
list_coef[2] = 5.0

## Scan the x-center
scan_center(xcenter, ycenter, -50, 50, 2, list_coef, list_power, output_
↪base,
            mat0, mat_pad, pad, axis="x")
## Found x=648 looks good.
xcenter = 646
```

(continues on next page)

(continued from previous page)

```
## Scan the y-center
scan_center(xcenter, ycenter, -50, 50, 2, list_coef, list_power, output_
↪base,
            mat0, mat_pad, pad, axis="y")
## Found y=480 looks good.
ycenter = 480
```

- The 0-order and 1st-order of *polynomial coefficients* control the scaling and the shearing of a warping image, we can adjust them to get the most of image staying inside the field-of-view. From the estimated coefficients of the forward model, it's straightforward to calculate the coefficients of the backward model which is used to unwarp the Percy's image.

```
# Adjust the 1st-order and 0-order coefficients manually if need to.
list_coef[1] = 1.0
list_coef[0] = 1.0

# Get a good estimation of the forward model
list_ffact = list_coef * list_power
# Transform to the backward model for correction
ref_points = [[i - ycenter, j - xcenter] for i in range(0, height, 50)␣
↪for j in
              range(0, width, 50)]
list_bfact = proc.transform_coef_backward_and_forward(list_ffact, ref_
↪points=ref_points)

# Load the color image
img = io.load_image(file_path, average=False)
img_corrected = np.copy(img)

# Unwarped each color channel of the image
for i in range(img.shape[-1]):
    img_corrected[:, :, i] = post.unwarp_image_backward(img[:, :, i],␣
↪xcenter,
                                                        ycenter, list_
↪bfact)

# Save the unwarped image.
io.save_image(output_base + "/F_R_hazcam_unwarped.png", img_corrected)
```

- From the determined coefficients, we can correct other images of the same camera (Fig. 77, Fig. 78).

```
# Correct other images from the same camera:
img = io.load_image("../../data/percy_cam/rock_core1.png", average=False)
for i in range(img.shape[-1]):
    img_corrected[:, :, i] = post.unwarp_image_backward(img[:, :, i],␣
↪xcenter,
                                                        ycenter, list_
↪bfact)
io.save_image(output_base + "/rock_core1_unwarped.png", img_corrected)

img = io.load_image("../../data/percy_cam/rock_core2.png", average=False)
for i in range(img.shape[-1]):
    img_corrected[:, :, i] = post.unwarp_image_backward(img[:, :, i],␣
↪xcenter,
                                                        ycenter, list_
↪bfact)
io.save_image(output_base + "/rock_core2_unwarped.png", img_corrected)
```

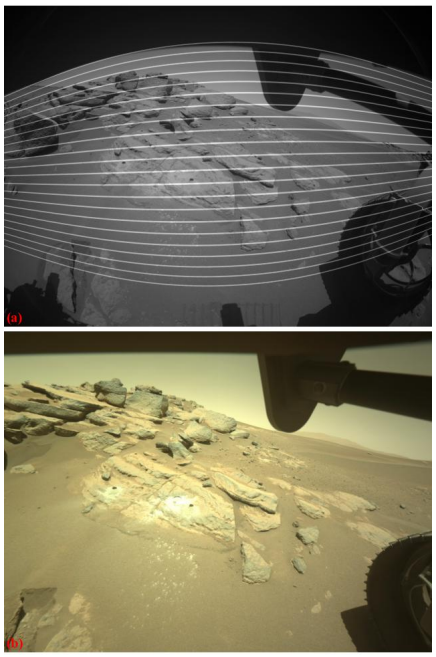


Fig. 76: . (a) Overlay between the warped line-pattern and the Percy's image. (b) Unwarped image of Fig. 75 (a).

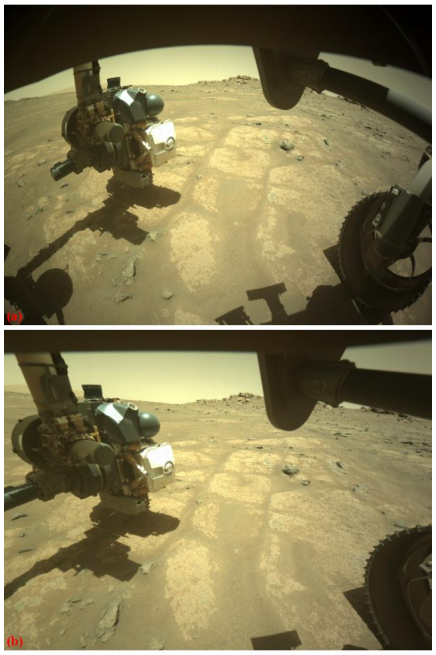


Fig. 77: . (a) Image of Percy trying to get the first rock-core. (b) Unwarped image of (a).

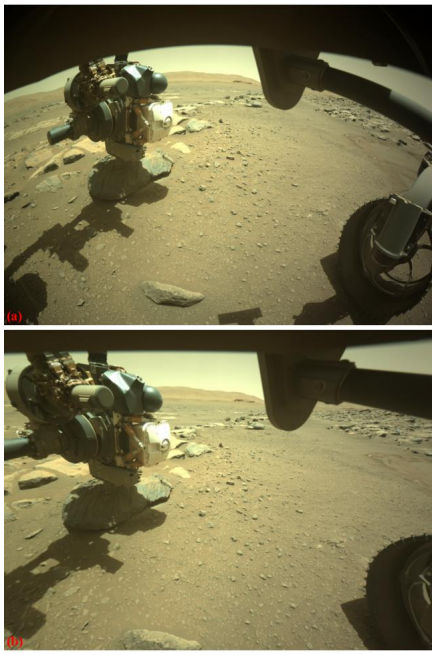


Fig. 78: . (a) Image of Percy trying to get the second rock-core. (b) Unwarped image of (a).

Click here to download the Python codes.

Useful tips

1) Apply distortion correction to a color image. Given distortion coefficients from the calibration process shown in previous demos, we have to apply unwarping to each channel of the image as follows:

```
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

file_path = "../../data/percy_cam/F_R_hazcam.png"
output_base = "E:/output/unwarping/"

img = io.load_image(file_path, average=False)
# Define a function for convenient use.
def unwarp_color_image(img, xcenter, ycenter, list_bfact, mode='reflect'):
    if len(img.shape) != 3:
        raise ValueError("Check if input is a color image!!!")
    img_corrected = []
    for i in range(img.shape[-1]):
        img_corrected.append(post.unwarp_image_backward(img[:, :, i], xcenter,
                                                        ycenter, list_bfact,␣
↪mode=mode))
    img_corrected = np.moveaxis(np.asarray(img_corrected), 0, 2)
    return img_corrected

# Suppose that we get xcenter, ycenter, list_bfact (backward mode) to this point
img_corrected = unwarp_color_image(img, xcenter, ycenter, list_bfact)
io.save_image(output_base + "/corrected_F_R_hazcam.png", img_corrected)
```

2) Find the coordinates of a point in one space given its corresponding coordinates in another space. Suppose that we get distortion coefficients of the backward model; given a point in a distorted image, we want to find

its position in the undistorted image. This can be done as the following:

```
import numpy as np
import discorpy.losa.loadersaver as io
import discorpy.proc.processing as proc
import discorpy.post.postprocessing as post

img = io.load_image("E:/data/image.jpg", average=True)
# Suppose that we get coefficients of the backward model (xcenter, yxenter, list_
↪bfact).
# We need to find the forward transformation using the given backward model.
(height, width) = img.shape
ref_points = [[i - ycenter, j - xcenter] for i in np.linspace(0, height, 40) for␣
↪j in
              np.linspace(0, width, 40)]
list_ffact = proc.transform_coef_backward_and_forward(list_bfact, ref_points=ref_
↪points)

# Define the function to calculate corresponding points between distorted and␣
↪undistorted space
# This function can be used both ways:
# In: distorted point, forward model :-> Out: undistorted point
# In: undistorted point, backward model :-> Out: distorted point
def find_point_to_point(points, xcenter, ycenter, list_fact):
    """
    points : (row_index, column_index) of the point.
    """
    xi, yi = points[1] - xcenter, points[0] - ycenter
    ri = np.sqrt(xi * xi + yi * yi)
    factor = np.float64(np.sum(list_fact * np.power(ri, np.arange(len(list_
↪fact)))))
    xo = xcenter + factor * xi
    yo = ycenter + factor * yi
    return xo, yo

# Find top-left point in the undistorted space given top-left point in the␣
↪distorted space.
xu_top_left, yu_top_left = find_point_to_point((0, 0), xcenter, ycenter, list_
↪ffact)
# Find bottom-right point in the undistorted space given bottom-right point in␣
↪the distorted space.
xu_bot_right, yu_bot_right = find_point_to_point((height - 1, width - 1),␣
↪xcenter, ycenter,
                                                 list_ffact)
```

3) Maintain the original field of view of an image. In correction for the barrel distortion, some parts of the image edges will be cropped. However, there are two possible methods to incorporate these areas back into the undistorted image:

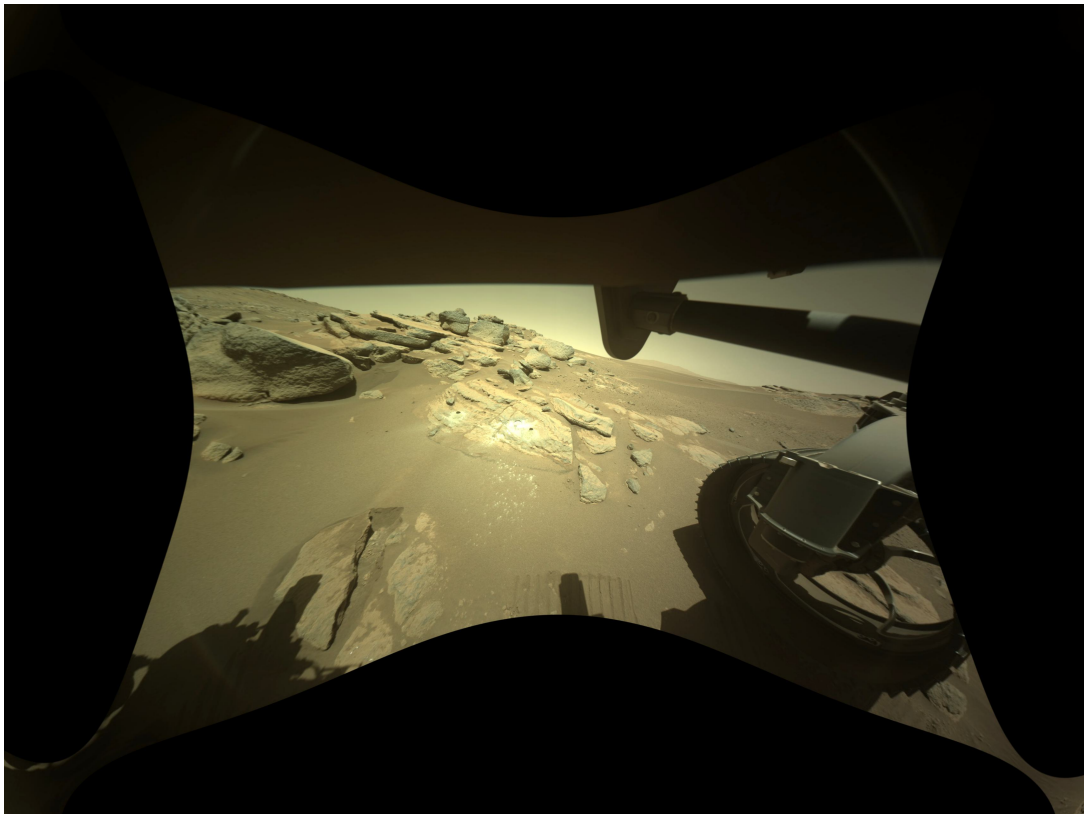
   - Apply padding to the original image before unwarping. The appropriate amount of padding can be determined using the results from the example mentioned above. Note that we have to update the center of distortion corresponding to the padding.

```
# Calculate padding width for each side.
pad_top = int(np.abs(yu_top_left))
pad_bot = int(yu_bot_right - height)
pad_left = int(np.abs(xu_top_left))
pad_right = int(xu_bot_right - width)
```

(continues on next page)

(continued from previous page)

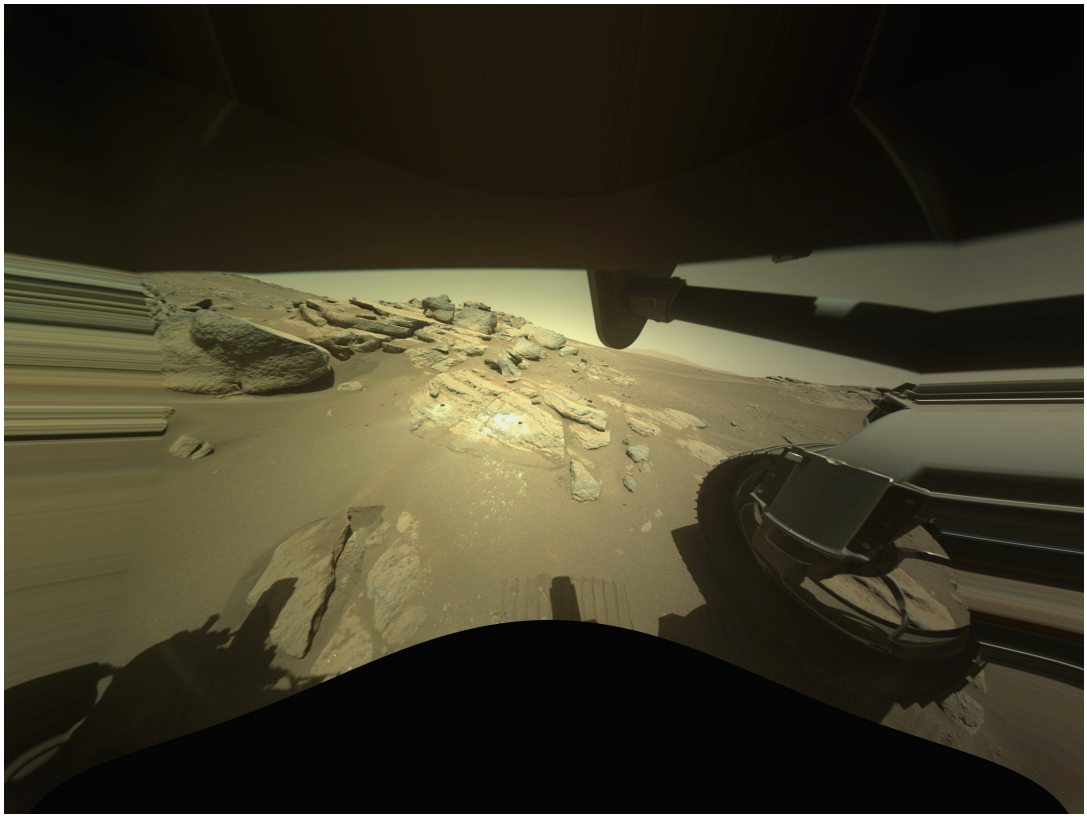
```
img_pad = np.pad(img, ((pad_top, pad_bot), (pad_left, pad_right), (0, 0)),␣
↪mode="constant")
img_corrected = unwarp_color_image(img_pad, xcenter + pad_left, ycenter +␣
↪pad_top,
                                   list_bfact, mode='constant')
io.save_image(output_base + "/F_R_hazcam_unwarped_padding.jpg", img_
↪corrected)
```



- Rescale the distortion coefficients and resize the original image correspondingly.

```
import scipy.ndimage as ndi

zoom = 2.0
list_bfact1 = zoom * list_bfact
xcenter1 = xcenter * zoom
ycenter1 = ycenter * zoom
img_corrected = []
for i in range(img.shape[-1]):
    img_tmp = ndi.zoom(img[:, :, i], zoom)
    img_tmp = post.unwarp_image_backward(img_tmp, xcenter1, ycenter1, list_
↪bfact1)
    img_corrected.append(img_tmp)
img_corrected = np.moveaxis(np.asarray(img_corrected), 0, 2)
io.save_image(output_base + "/F_R_hazcam_unwarped_zoom.jpg", img_corrected)
```



4. To unwarp live images from a webcam or camera, it is recommended to use the remap function provided by OpenCV library due to its high performance. The following is a demonstration of its usage. However, it should be noted that for packaging purpose, Discorpy does not include this function in its API. Instead, map_coordinate function in Scipy is utilized.

```
import os
import numpy as np
import cv2


def mapping_cv2(mat, xmat, ymat, method=cv2.INTER_LINEAR,
                border=cv2.BORDER_CONSTANT):
    """
    Apply a geometric transformation to a 2D array using Opencv.

    Parameters
    ----------
    mat : array_like.
        Input image. Can be a color image.
    xmat : array_like
        2D array of the x-coordinates. Origin is at the left of the image.
    ymat : array_like
        2D array of the y-coordinates. Origin is at the top of the image.
    method : obj
        To select interpolation method. Note to use the prefix: cv2.<method>\\
        https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html
    border : obj
        To select method for boundary handling. Note to use the prefix: cv2.
↪<method> \\
        https://docs.opencv.org/3.4/d2/de8/group__core__array.html
↪#ga209f2f4869e304c82d07739337eae7c5
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    array_like
    """
    mat = cv2.remap(mat, xmat, ymat, interpolation=method, borderMode=border)
    return mat


def unwarp_image_backward_cv2(mat, xcenter, ycenter, list_fact,
                              method=cv2.INTER_LINEAR,
                              border=cv2.BORDER_CONSTANT):
    """
    Unwarp an image using the backward model with the Opencv remap method for
    fast performance.

    Parameters
    ----------
    mat : array_like
        Input image. Can be a color image.
    xcenter : float
        Center of distortion in x-direction.
    ycenter : float
        Center of distortion in y-direction.
    list_fact : list of float
        Polynomial coefficients of the backward model.
    method : obj
        To select interpolation method. Note to use the prefix: cv2.<method>\\
        https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html
    border : obj
        To select method for boundary handling. Note to use the prefix: cv2.
↪<method> \\
        https://docs.opencv.org/3.4/d2/de8/group__core__array.html
↪#ga209f2f4869e304c82d07739337eae7c5

    Returns
    -------
    array_like
        2D array. Distortion-corrected image.
    """
    (height, width) = mat.shape[:2]
    xu_list = np.arange(width) - xcenter
    yu_list = np.arange(height) - ycenter
    xu_mat, yu_mat = np.meshgrid(xu_list, yu_list)
    ru_mat = np.sqrt(xu_mat**2 + yu_mat**2)
    fact_mat = np.sum(
        np.asarray([factor * ru_mat**i for i,
                    factor in enumerate(list_fact)]), axis=0)
    xd_mat = np.float32(np.clip(xcenter + fact_mat * xu_mat, 0, width - 1))
    yd_mat = np.float32(np.clip(ycenter + fact_mat * yu_mat, 0, height - 1))
    mat = mapping_cv2(mat, xd_mat, yd_mat, method=method, border=border)
    return mat


def unwarp_video_cv2(cam_obj, xcenter, ycenter, list_fact,
                     method=cv2.INTER_LINEAR, border=cv2.BORDER_CONSTANT):
```

(continues on next page)

(continued from previous page)

```
    """
    Unwarp frames from Opencv video object using the backward model.

    Parameters
    ----------
    cam_obj : obj
        Opencv camera object. e.g. cv2.VideoCapture(0)
    xcenter : float
        Center of distortion in x-direction.
    ycenter : float
        Center of distortion in y-direction.
    list_fact : list of float
        Polynomial coefficients of the backward model.
    method : obj
        To select interpolation method. Note to use the prefix: cv2.<method>\\
        https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html
    border : obj
        To select method for boundary handling. Note to use the prefix: cv2.
↪<method> \\
        https://docs.opencv.org/3.4/d2/de8/group__core__array.html
↪#ga209f2f4869e304c82d07739337eae7c5

    Returns
    -------
    Generator
    """
    width = int(cam_obj.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cam_obj.get(cv2.CAP_PROP_FRAME_HEIGHT))
    xu_list = np.arange(width) - xcenter
    yu_list = np.arange(height) - ycenter
    xu_mat, yu_mat = np.meshgrid(xu_list, yu_list)
    ru_mat = np.sqrt(xu_mat**2 + yu_mat**2)
    fact_mat = np.sum(
        np.asarray([factor * ru_mat**i for i,
                    factor in enumerate(list_fact)]), axis=0)
    xd_mat = np.float32(np.clip(xcenter + fact_mat * xu_mat, 0, width - 1))
    yd_mat = np.float32(np.clip(ycenter + fact_mat * yu_mat, 0, height - 1))
    while True:
        check, frame = cam_obj.read()
        if check:
            uframe = mapping_cv2(frame, xd_mat, yd_mat, method=method,
                                 border=border)
        cv2.imshow('Transformed webcam video. Press ESC to stop!!!', uframe)
        c = cv2.waitKey(1)
        if c == 27:
            break
    cam_obj.release()
    cv2.destroyAllWindows()


def save_color_image_cv2(file_path, image):
    """
    Convenient method for saving color image which creates a folder if it
    doesn't exist.
    """
    file_base = os.path.dirname(file_path)
```

(continues on next page)

(continued from previous page)

```
    if not os.path.exists(file_base):
        try:
            os.makedirs(file_base)
        except OSError:
            raise ValueError("Can't create the folder: {}".format(file_path))
    cv2.imwrite(file_path, image)


#--------------------------------------------------------------------------
# Demonstration of using above functions for unwarping images from a webcam.
#--------------------------------------------------------------------------

# Open the webcam
cam_obj = cv2.VideoCapture(0)
# Get height and width
width = int(cam_obj.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cam_obj.get(cv2.CAP_PROP_FRAME_HEIGHT))

# For demonstration, assuming that we get the following coefficients.
xcenter = width / 2.0
ycenter = height / 2.0
list_power = np.asarray([1.0, 10**(-3), 10**(-7)])
list_coef = np.asarray([1.0, 1.0, 1.0])
list_fact = list_power * list_coef

# Get a single image from the camera, apply correction, and save the result.
check, frame = cam_obj.read()
if check:
    frame = unwarp_image_backward_cv2(frame, xcenter, ycenter, list_fact)
save_color_image_cv2("E:/tmp/distortion_cv2/unwarp_single_image.jpg", frame)

# Unwarp frames from the webcam and display the results. Press ESC for stopping␣
↪the streaming.
unwarp_video_cv2(cam_obj, xcenter, ycenter, list_fact)
```

1.4 API reference

1.4.1 Input-output

discorpy.losa.loadersaver

Module for I/O tasks:

- Load data from an image file (tif, png, jpg) or a hdf file.
- Save a 2D array as a tif/png/jpg image or a 2D, 3D array to a hdf file.
- Save a plot of data points to an image.
- Save/load metadata to/from a text file.

Functions:

load_image(file_path[, average])	Load data from an image.
get_hdf_information(file_path)	Get information of datasets in a hdf/nxs file.
find_hdf_key(file_path, pattern)	Find datasets matching the pattern in a hdf/nxs file.
load_hdf_file(file_path[, key_path, index, axis])	Load data from a hdf5/nxs file.
load_hdf_object(file_path, key_path)	Load a hdf/nexus dataset as an object.
save_image(file_path, mat[, overwrite])	Save 2D data to an image.
save_plot_image(file_path, list_lines, ...)	Save the plot of dot-centroids to an image.
save_residual_plot(file_path, list_data, ...)	Save the plot of residual against radius to an image.
save_hdf_file(file_path, idata[, key_path, ...])	Write data to a hdf5 file.
open_hdf_stream(file_path, data_shape[, ...])	Open stream to write data to a hdf/nxs file with options to add metadata.
save_metadata_txt(file_path, xcenter, ...[, ...])	Write metadata to a text file.
load_metadata_txt(file_path)	Load distortion coefficients from a text file.
save_plot_points(file_path, list_points, ...)	Save the plot of dot-centroids to an image.

discorpy.losa.loadersaver.**load_image**(*file_path*, *average=True*)

Load data from an image.

Parameters

- **file_path** (*str*) – Path to a file.
- **average** (*bool, optional*) – Average a multichannel image if True.

Returns

array_like

discorpy.losa.loadersaver.**get_hdf_information**(*file_path*)

Get information of datasets in a hdf/nxs file.

Parameters

file_path (*str*) – Path to the file.

Returns

- **list_key** (*str*) – Keys to the datasets.
- **list_shape** (*tuple of int*) – Shapes of the datasets.
- **list_type** (*str*) – Types of the datasets.

discorpy.losa.loadersaver.**find_hdf_key**(*file_path*, *pattern*)

Find datasets matching the pattern in a hdf/nxs file.

Parameters

- **file_path** (*str*) – Path to the file.
- **pattern** (*str*) – Pattern to find the full names of the datasets.

Returns

- **list_key** (*str*) – Keys to the datasets.
- **list_shape** (*tuple of int*) – Shapes of the datasets.
- **list_type** (*str*) – Types of the datasets.

discorpy.losa.loadersaver.**load_hdf_file**(*file_path*, *key_path=None*, *index=None*, *axis=0*)

Load data from a hdf5/nxs file.

Parameters

- **file_path** (*str*) – Path to a hdf/nxs file.

- **key_path** (*str*) – Key path to a dataset.
- **index** (*int or tuple of int*) – Values for slicing data. Can be integer, tuple or list, e.g index=(start,stop,step) or index=(slice1, slice2, slice3,slice4).
- **axis** (*int*) – Slice direction

Returns
array_like – 2D array or 3D array.

`discorpy.losa.loadersaver.load_hdf_object(file_path, key_path)`

Load a hdf/nexus dataset as an object.

Parameters

- **file_path** (*str*) – Path to a hdf/nxs file.
- **key_path** (*str*) – Key path to a dataset.

Returns
object – hdf/nxs object.

`discorpy.losa.loadersaver.save_image(file_path, mat, overwrite=True)`

Save 2D data to an image.

Parameters

- **file_path** (*str*) – Output file path.
- **mat** (*array_like*) – 2D array.
- **overwrite** (*bool, optional*) – Overwrite an existing file if True.

Returns
str – Updated file path.

`discorpy.losa.loadersaver.save_plot_image(file_path, list_lines, height, width, overwrite=True, dpi=100)`

Save the plot of dot-centroids to an image. Useful to check if the dots are arranged properly where dots on the same line having the same color.

Parameters

- **file_path** (*str*) – Output file path.
- **list_lines** (*list of array_like*) – List of 2D arrays. Each list is the coordinates of dots on a line.
- **height** (*int*) – Height of the image.
- **width** (*int*) – Width of the image.
- **overwrite** (*bool, optional*) – Overwrite the existing file if True.
- **dpi** (*int, optional*) – The resolution in dots per inch.

Returns
str – Updated file path.

`discorpy.losa.loadersaver.save_residual_plot(file_path, list_data, height, width, overwrite=True, dpi=100)`

Save the plot of residual against radius to an image. Useful to check the accuracy of unwarping results.

Parameters

- **file_path** (*str*) – Output file path.
- **list_data** (*array_like*) – 2D array. List of [residual, radius] of each dot.
- **height** (*int*) – Height of the output image.

- **width** (*int*) – Width of the output image.
- **overwrite** (*bool, optional*) – Overwrite the existing file if True.
- **dpi** (*int, optional*) – The resolution in dots per inch.

Returns
str – Updated file path.

discorpy.losa.loadersaver.**save_hdf_file**(*file_path*, *idata*, *key_path='entry'*, *overwrite=True*)

Write data to a hdf5 file.

Parameters

- **file_path** (*str*) – Output file path.
- **idata** (*array_like*) – Data to be saved.
- **key_path** (*str*) – Key path to the dataset.
- **overwrite** (*bool, optional*) – Overwrite an existing file if True.

Returns
str – Updated file path.

discorpy.losa.loadersaver.**open_hdf_stream**(*file_path*, *data_shape*, *key_path='entry/data'*, *data_type='float32'*, *overwrite=True*, ***options*)

Open stream to write data to a hdf/nxs file with options to add metadata.

Parameters

- **file_path** (*str*) – Path to the file.
- **data_shape** (*tuple of int*) – Shape of the data.
- **key_path** (*str*) – Key path to the dataset.
- **data_type** (*str*) – Type of data.
- **overwrite** (*bool*) – Overwrite the existing file if True.
- **options** (*dict, optional*) – Add metadata. Example: options={"entry/angles": angles, "entry/energy": 53}.

Returns
object – hdf object.

discorpy.losa.loadersaver.**save_metadata_txt**(*file_path*, *xcenter*, *ycenter*, *list_fact*, *overwrite=True*)

Write metadata to a text file.

Parameters

- **file_path** (*str*) – Output file path.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*float*) – 1D array. Coefficients of a polynomial.
- **overwrite** (*bool, optional*) – Overwrite an existing file if True.

Returns
str – Updated file path.

discorpy.losa.loadersaver.**load_metadata_txt**(*file_path*)

Load distortion coefficients from a text file.

Parameters
file_path (*str*) – Path to a file.

Returns
tuple of floats and list – Tuple of (xcenter, ycenter, list_fact).

`discorpy.losa.loadersaver.`**`save_plot_points`**(*file_path*, *list_points*, *height*, *width*, *overwrite=True*, *dpi=100*, *marker='o'*, *color='blue'*)

Save the plot of dot-centroids to an image. Useful to check if the dots are arranged properly where dots on the same line having the same color.

Parameters

- **file_path** (*str*) – Output file path.
- **list_points** (*list of 1D-array*) – List of the (y-x)-coordinates of points.
- **height** (*int*) – Height of the image.
- **width** (*int*) – Width of the image.
- **overwrite** (*bool, optional*) – Overwrite the existing file if True.
- **dpi** (*int, optional*) – The resolution in dots per inch.
- **marker** (*str*) – Plot marker. Full list is at: https://matplotlib.org/stable/api/markers_api.html
- **color** (*str*) – Marker color. Full list is at: https://matplotlib.org/stable/tutorials/colors/colors.html

Returns
str – Updated file path.

1.4.2 Pre-processing

`discorpy.prep.preprocessing`

Module of pre-processing methods:

- Normalize, binarize an image.
- Determine the median dot-size, median distance between two nearest dots, and the slopes of grid-lines of a dot-pattern image.
- Remove non-dot objects or misplaced dots.
- Group dot-centroids into horizontal lines and vertical lines.
- Calculate a threshold value for binarizing.

Functions:

normalization(mat[, size])	Correct a non-uniform background of an image using the median filter.
normalization_fft(mat[, sigma, pad, mode])	Correct a non-uniform background image using a Fourier Gaussian filter.
binarization(mat[, ratio, thres, denoise])	Apply a list of operations: binarizing an 2D array; inverting the contrast of dots if needs to; removing border components; cleaning salty noise; and filling holes.
check_num_dots(mat)	Check if the number of dots is not enough for parabolic fit.
calc_size_distance(mat[, ratio])	Find the median size of dots and the median distance between two nearest dots.
select_dots_based_size(mat, dot_size[, ratio])	Select dots having a certain size.
select_dots_based_ratio(mat[, ratio])	Select dots having the ratio between the axes length of the fitted ellipse smaller than a threshold.
select_dots_based_distance(mat, dot_dist[, ...])	Select dots having a certain range of distance to theirs neighbouring dots.
calc_hor_slope(mat[, ratio])	Calculate the slope of horizontal lines against the horizontal axis.
calc_ver_slope(mat[, ratio])	Calculate the slope of vertical lines against the vertical axis.
group_dots_hor_lines(mat, slope, dot_dist[, ...])	Group dots into horizontal lines.
group_dots_ver_lines(mat, slope, dot_dist[, ...])	Group dots into vertical lines.
remove_residual_dots_hor(list_lines, slope)	Remove dots having distances larger than a certain value from fitted horizontal parabolas.
remove_residual_dots_ver(list_lines, slope)	Remove dots having distances larger than a certain value from fitted vertical parabolas.
calculate_threshold(mat[, bgr, snr])	Calculate a threshold value based on Algorithm 4 in Ref.

discorpy.prep.preprocessing.**normalization**(*mat*, *size=51*)

Correct a non-uniform background of an image using the median filter.

Parameters

- **mat** (*array_like*) – 2D array.
- **size** (*int*) – Size of the median filter.

Returns

array_like – 2D array. Corrected background.

discorpy.prep.preprocessing.**normalization_fft**(*mat*, *sigma=10*, *pad=100*, *mode='reflect'*)

Correct a non-uniform background image using a Fourier Gaussian filter.

Parameters

- **mat** (*array_like*) – 2D array.
- **sigma** (*int*) – Sigma of the Gaussian.
- **pad** (*int*) – Pad width.
- **mode** (*str*) – Padding mode.

Returns

array_like – 2D array. Corrected background image.

discorpy.prep.preprocessing.**binarization**(*mat*, *ratio=0.3*, *thres=None*, *denoise=True*)

Apply a list of operations: binarizing an 2D array; inverting the contrast of dots if needs to; removing border components; cleaning salty noise; and filling holes.

Parameters

- **mat** (*array_like*) – 2D array.
- **ratio** (*float*) – Used to select the ROI around the middle of the image for calculating threshold.
- **thres** (*float, optional*) – Threshold for binarizing. Automatically calculated if None.
- **denoise** (*bool, optional*) – Apply denoising to the image if True.

Returns

array_like – 2D binary array.

discorpy.prep.preprocessing.**check_num_dots**(*mat*)

Check if the number of dots is not enough for parabolic fit.

Parameters

mat (*array_like*) – 2D binary array.

Returns

bool – True means not enough.

discorpy.prep.preprocessing.**calc_size_distance**(*mat*, *ratio=0.3*)

Find the median size of dots and the median distance between two nearest dots.

Parameters

- **mat** (*array_like*) – 2D binary array.
- **ratio** (*float*) – Used to select the ROI around the middle of an image.

Returns

- **dot_size** (*float*) – Median size of the dots.
- **dot_dist** (*float*) – Median distance between two nearest dots.

discorpy.prep.preprocessing.**select_dots_based_size**(*mat*, *dot_size*, *ratio=0.3*)

Select dots having a certain size.

Parameters

- **mat** (*array_like*) – 2D binary array.
- **dot_size** (*float*) – Size of the standard dot.
- **ratio** (*float*) – Used to calculate the acceptable range. [dot_size - ratio*dot_size; dot_size + ratio*dot_size]

Returns

array_like – 2D array. Selected dots.

discorpy.prep.preprocessing.**select_dots_based_ratio**(*mat*, *ratio=0.3*)

Select dots having the ratio between the axes length of the fitted ellipse smaller than a threshold.

Parameters

- **mat** (*array_like*) – 2D binary array.
- **ratio** (*float*) – Threshold value.

Returns

array_like – 2D array. Selected dots.

discorpy.prep.preprocessing.**select_dots_based_distance**(*mat*, *dot_dist*, *ratio=0.3*)

Select dots having a certain range of distance to theirs neighbouring dots.

Parameters

- **mat** (*array_like*) – 2D array.
- **dot_dist** (*float*) – Median distance of two nearest dots.
- **ratio** (*float*) – Used to calculate acceptable range.

Returns

array_like – 2D array. Selected dots.

discorpy.prep.preprocessing.**calc_hor_slope**(*mat*, *ratio=0.3*)

Calculate the slope of horizontal lines against the horizontal axis.

Parameters

- **mat** (*array_like*) – 2D binary array.
- **ratio** (*float*) – Used to select the ROI around the middle of an image.

Returns

float – Horizontal slope of the grid.

discorpy.prep.preprocessing.**calc_ver_slope**(*mat*, *ratio=0.3*)

Calculate the slope of vertical lines against the vertical axis.

Parameters

- **mat** (*array_like*) – 2D binary array.
- **ratio** (*float*) – Used to select the ROI around the middle of a image.

Returns

float – Vertical slope of the grid.

discorpy.prep.preprocessing.**group_dots_hor_lines**(*mat*, *slope*, *dot_dist*, *ratio=0.3*, *num_dot_miss=6*, *accepted_ratio=0.65*)

Group dots into horizontal lines.

Parameters

- **mat** (*array_like*) – A binary image or a list of (y,x)-coordinates of points.
- **slope** (*float*) – Horizontal slope of the grid.
- **dot_dist** (*float*) – Median distance of two nearest dots.
- **ratio** (*float*) – Acceptable variation.
- **num_dot_miss** (*int*) – Acceptable missing dots between dot1 and dot2.
- **accepted_ratio** (*float*) – Use to select lines having the number of dots equal to or larger than the multiplication of the *accepted_ratio* and the maximum number of dots per line.

Returns

list of array_like – List of 2D arrays. Each list is the coordinates (y, x) of dot-centroids belong to the same group. Length of each list may be different.

discorpy.prep.preprocessing.**group_dots_ver_lines**(*mat*, *slope*, *dot_dist*, *ratio=0.3*, *num_dot_miss=6*, *accepted_ratio=0.65*)

Group dots into vertical lines.

Parameters

- **mat** (*array_like*) – A binary image or a list of (y,x)-coordinates of points.
- **slope** (*float*) – Vertical slope of the grid.

- **dot_dist** (*float*) – Median distance of two nearest dots.
- **ratio** (*float*) – Acceptable variation.
- **num_dot_miss** (*int*) – Acceptable missing dots between dot1 and dot2.
- **accepted_ratio** (*float*) – Use to select lines having the number of dots equal to or larger than the multiplication of the *accepted_ratio* and the maximum number of dots per line.

Returns

list of array_like – List of 2D arrays. Each list is the coordinates (y, x) of dot-centroids belong to the same group. Length of each list may be different.

discorpy.prep.preprocessing.**remove_residual_dots_hor**(*list_lines*, *slope*, *residual=2.5*)

Remove dots having distances larger than a certain value from fitted horizontal parabolas.

Parameters

- **list_lines** (*list of array_like*) – List of the coordinates of dot-centroids on horizontal lines.
- **slope** (*float*) – Horizontal slope of the grid.
- **residual** (*float*) – Acceptable distance in pixel unit between a dot and a fitted parabola.

Returns

list of array_like – List of 2D arrays. Each list is the coordinates (y, x) of dot-centroids belong to the same group. Length of each list may be different.

discorpy.prep.preprocessing.**remove_residual_dots_ver**(*list_lines*, *slope*, *residual=2.5*)

Remove dots having distances larger than a certain value from fitted vertical parabolas.

Parameters

- **list_lines** (*list of float*) – List of the coordinates of the dot-centroids on the vertical lines.
- **slope** (*float*) – Slope of the vertical line.
- **residual** (*float*) – Acceptable distance in pixel unit between the dot and the fitted parabola.

Returns

list of float – List of 2D array. Each list is the coordinates (y, x) of dot-centroids belong to the same group. Length of each list may be different.

discorpy.prep.preprocessing.**calculate_threshold**(*mat*, *bgr='bright'*, *snr=2.0*)

Calculate a threshold value based on Algorithm 4 in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array.
- **bgr** (*{"bright", "dark"}*) – To indicate the brightness of the background against image features.
- **snr** (*float*) – Ratio (>1.0) used to separate image features against noise. Greater is less sensitive.

Returns

float – Threshold value.

References

discorpy.prep.linepattern

Module of pre-processing methods for handling a line-pattern image:

- Determine the slopes and distances between lines.
- Extract points belong to a line.
- Convert a chessboard image to a line-pattern image.

Functions:

locate_subpixel_point(list_point[, option])	Locate the extremum point of a 1D array with subpixel accuracy.
get_local_extrema_points(list_data[, ...])	Get a list of local extremum points from a 1D array.
calc_slope_distance_hor_lines(mat[, ratio, ...])	Calculate the representative distance between horizontal lines and the representative slope of these lines using the ROI around the middle of a line-pattern image.
calc_slope_distance_ver_lines(mat[, ratio, ...])	Calculate the representative distance between vertical lines and the representative slope of these lines using the ROI around the middle of a line-pattern image.
get_tilted_profile(mat, index, angle_deg, ...)	Get the intensity-profile along a tilted line across an image.
get_cross_points_hor_lines(mat, slope_ver, ...)	Get points on horizontal lines of a line-pattern image by intersecting with a list of generated vertical-lines.
get_cross_points_ver_lines(mat, slope_hor, ...)	Get points on vertical lines of a line-pattern image by intersecting with a list of generated horizontal-lines.
convert_chessboard_to_linepattern(mat[, ...])	Convert a chessboard image to a line-pattern image.

discorpy.prep.linepattern.**locate_subpixel_point**(*list_point*, *option='min'*)

Locate the extremum point of a 1D array with subpixel accuracy.

Parameters

- **list_point** (*array_like*) – 1D array.
- **option** (*{"min", "max"}*) – To locate the minimum point or the maximum point.

Returns

float – Subpixel position of the extremum point.

discorpy.prep.linepattern.**get_local_extrema_points**(*list_data*, *option='min'*, *radius=7*, *sensitive=0.1*, *denoise=True*, *norm=True*, *subpixel=True*)

Get a list of local extremum points from a 1D array.

Parameters

- **list_data** (*array_like*) – 1D array.
- **option** (*{"min", "max"}*) – To get minimum points or maximum points
- **radius** (*int*) – Search radius. Used to locate extremum points.
- **sensitive** (*float*) – To detect extremum points against random noise. Smaller is more sensitive.
- **denoise** (*bool, optional*) – Applying a smoothing filter if True.
- **norm** (*bool, optional*) – Apply background normalization to the array.

- **subpixel** (*bool, optional*) – Locate points with subpixel accuracy.

Returns

array_like – 1D array. Positions of local extremum points.

discorpy.prep.linepattern.**calc_slope_distance_hor_lines**(*mat*, *ratio=0.3*, *search_range=30.0*, *radius=9*, *sensitive=0.1*, *bgr='bright'*, *denoise=True*, *norm=True*, *subpixel=True*)

Calculate the representative distance between horizontal lines and the representative slope of these lines using the ROI around the middle of a line-pattern image.

Parameters

- **mat** (*array_like*) – 2D array.
- **ratio** (*float*) – Used to select the ROI around the middle of an image.
- **search_range** (*float*) – Search range in Degree to determine the slope of lines.
- **radius** (*int*) – Search radius. Used to locate lines.
- **sensitive** (*float*) – To detect lines against random noise. Smaller is more sensitive.
- **bgr** (*{"bright", "dark"}*) – Specify the brightness of the background against the lines.
- **denoise** (*bool, optional*) – Applying a smoothing filter if True.
- **norm** (*bool, optional*) – Apply background normalization to the array.
- **subpixel** (*bool, optional*) – Locate points with subpixel accuracy.

Returns

- **slope** (*float*) – Slope of horizontal lines in Radian.
- **distance** (*float*) – Distance between horizontal lines.

discorpy.prep.linepattern.**calc_slope_distance_ver_lines**(*mat*, *ratio=0.3*, *search_range=30.0*, *radius=9*, *sensitive=0.1*, *bgr='bright'*, *denoise=True*, *norm=True*, *subpixel=True*)

Calculate the representative distance between vertical lines and the representative slope of these lines using the ROI around the middle of a line-pattern image.

Parameters

- **mat** (*array_like*) – 2D array.
- **ratio** (*float*) – Used to select the ROI around the middle of an image.
- **search_range** (*float*) – Search range in Degree to determine the slope of lines.
- **radius** (*int*) – Search radius. Used to locate lines.
- **sensitive** (*float*) – To detect lines against random noise. Smaller is more sensitive.
- **bgr** (*{"bright", "dark"}*) – Specify the brightness of the background against the lines.
- **denoise** (*bool, optional*) – Applying a smoothing filter if True.
- **subpixel** (*bool, optional*) – Locate points with subpixel accuracy.

Returns

- **slope** (*float*) – Slope of vertical lines in Radian.
- **distance** (*float*) – Distance between vertical lines.

discorpy.prep.linepattern.**get_tilted_profile**(*mat*, *index*, *angle_deg*, *direction*)

Get the intensity-profile along a tilted line across an image. Positive angle is counterclockwise.

Parameters

- **mat** (*array_like*) – 2D array.
- **index** (*int*) – Index of the line.
- **angle_deg** (*float*) – Tilted angle in Degree.
- **direction** (*{"horizontal", "vertical"}*) – Direction of line-profile.

Returns

- **xlist** (*array_like*) – 1D array. x-positions of points on the line.
- **ylist** (*array_like*) – 1D array. y-positions of points on the line.
- **profile** (*array_like*) – 1D array. Intensities of points on the line.

discorpy.prep.linepattern.**get_cross_points_hor_lines**(*mat*, *slope_ver*, *dist_ver*, *ratio=1.0*, *norm=True*, *offset=0*, *bgr='bright'*, *radius=7*, *sensitive=0.1*, *denoise=True*, *subpixel=True*)

Get points on horizontal lines of a line-pattern image by intersecting with a list of generated vertical-lines.

Parameters

- **mat** (*array_like*) – 2D array.
- **slope_ver** (*float*) – Slope in Radian of generated vertical lines.
- **dist_ver** (*float*) – Distance between two adjacent generated lines.
- **ratio** (*float*) – To adjust the distance between generated lines to get more/less lines.
- **norm** (*bool, optional*) – Apply background normalization to the array.
- **offset** (*int*) – Starting index of generated lines.
- **bgr** (*{"bright", "dark"}*) – Specify the brightness of the background against the lines.
- **radius** (*int*) – Search radius. Used to locate extremum points.
- **sensitive** (*float*) – To detect extremum points against random noise. Smaller is more sensitive.
- **denoise** (*bool, optional*) – Applying a smoothing filter if True.
- **subpixel** (*bool, optional*) – Locate points with subpixel accuracy.

Returns

array_like – List of (y,x)-coordinates of points.

discorpy.prep.linepattern.**get_cross_points_ver_lines**(*mat*, *slope_hor*, *dist_hor*, *ratio=1.0*, *norm=True*, *offset=0*, *bgr='bright'*, *radius=7*, *sensitive=0.1*, *denoise=True*, *subpixel=True*)

Get points on vertical lines of a line-pattern image by intersecting with a list of generated horizontal-lines.

Parameters

- **mat** (*array_like*) – 2D array.
- **slope_hor** (*float*) – Slope in Radian of generated horizontal lines.
- **dist_hor** (*float*) – Distance between two adjacent generated lines.
- **ratio** (*float*) – To adjust the distance between generated lines to get more/less lines.
- **norm** (*bool, optional*) – Apply background normalization to the array.

- **offset** (*int*) – Starting index of generated lines.
- **bgr** (*{"bright", "dark"}*) – Specify the brightness of the background against the lines.
- **radius** (*int*) – Search radius. Used to locate extremum points.
- **sensitive** (*float*) – To detect extremum points against random noise. Smaller is more sensitive.
- **denoise** (*bool, optional*) – Applying a smoothing filter if True.
- **subpixel** (*bool, optional*) – Locate points with subpixel accuracy.

Returns
array_like – List of (y,x)-coordinates of points.

`discorpy.prep.linepattern.convert_chessboard_to_linepattern`(*mat*, *smooth=False*, *bgr='bright'*)

Convert a chessboard image to a line-pattern image.

Parameters

- **mat** (*array_like*) – 2D array.
- **smooth** (*bool, optional*) – Apply a gaussian smoothing filter if True.
- **bgr** (*{'bright', 'dark'}*) – Select the background of the output image.

Returns
array_like – Line-pattern image.

1.4.3 Processing

`discorpy.proc.processing`

Module of processing methods:

- Fit lines of dots to parabolas, find the center of distortion.
- Calculate undistorted intercepts of gridlines.
- Calculate distortion coefficients of the backward model, the forward model, and the backward-from-forward model.
- Correct perspective distortion affecting curve lines.
- Generate non-perspective points or lines from perspective points or lines.
- Calculate perspective coefficients.

Functions:

_para_fit_hor(list_lines, xcenter, ycenter)	Fit horizontal lines of dots to parabolas.
_para_fit_ver(list_lines, xcenter, ycenter)	Fit vertical lines of dots to parabolas.
find_cod_coarse(list_hor_lines, list_ver_lines)	Coarse estimation of the center of distortion.
find_cod_fine(list_hor_lines, ...)	Find the best center of distortion (CoD) by searching around the coarse estimation of the CoD.
_calc_undistor_intercept(list_hor_lines, ...)	Calculate the intercepts of undistorted lines.
calc_coef_backward(list_hor_lines, ...[, ...])	Calculate the distortion coefficients of a backward mode.
calc_coef_forward(list_hor_lines, ...[, ...])	Calculate the distortion coefficients of a forward mode.
calc_coef_backward_from_forward(...[, ...])	Calculate the distortion coefficients of a backward mode from a forward model.
transform_coef_backward_and_forward(list_fact	Transform polynomial coefficients of a radial distortion model between forward mapping and backward mapping.
find_cod_bailey(list_hor_lines, list_ver_lines)	Find the center of distortion (COD) using the Bailey's approach (Ref.
_generate_non_perspective_parabola_coef(...)	Correct the deviation of fitted parabola coefficients of each line caused by perspective distortion.
_find_cross_point_between_parabolas(...)	Find a cross point between two parabolas.
regenerate_grid_points_parabola(...[, ...])	Regenerating grid points by finding cross points between horizontal lines and vertical lines using their parabola coefficients.
_generate_linear_coef(list_hor_lines, ...[, ...])	Get linear coefficients of horizontal and vertical lines from linear fit.
_find_cross_point_between_lines(...)	Find a cross point between two lines.
_calc_undistor_intercept_perspective(...[, ...])	Calculate the intercepts of undistorted lines from perspective distortion.
regenerate_grid_points_linear(...)	Regenerating grid points by finding cross points between horizontal lines and vertical lines using their linear coefficients.
generate_undistorted_perspective_lines(...)	Generate undistorted lines from perspective lines.
generate_source_target_perspective_points(.	Generate source points (distorted) and target points (undistorted).
generate_4_source_target_perspective_point	Generate 4 rectangular points corresponding to 4 perspective-distorted points.
calc_perspective_coefficients(source_points, ...)	Calculate perspective coefficients of a matrix to map from source points to target points (Ref.
update_center(list_lines, xcenter, ycenter)	Update the coordinate-center of points on lines.

discorpy.proc.processing.**find_cod_coarse**(*list_hor_lines*, *list_ver_lines*)

Coarse estimation of the center of distortion.

Parameters

- **list_hor_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each horizontal line.
- **list_ver_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each vertical line.

Returns

- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.

discorpy.proc.processing.**find_cod_fine**(*list_hor_lines*, *list_ver_lines*, *xcenter*, *ycenter*, *dot_dist*)

Find the best center of distortion (CoD) by searching around the coarse estimation of the CoD.

Parameters

- **list_hor_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each horizontal line.
- **list_ver_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each vertical line.
- **xcenter** (*float*) – Coarse estimation of the CoD in x-direction.
- **ycenter** (*float*) – Coarse estimation of the CoD in y-direction.
- **dot_dist** (*float*) – Median distance of two nearest dots.

Returns

- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.

discorpy.proc.processing.**calc_coef_backward**(*list_hor_lines*, *list_ver_lines*, *xcenter*, *ycenter*, *num_fact*, *optimizing=False*, *threshold=0.3*)

Calculate the distortion coefficients of a backward mode.

Parameters

- **list_hor_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each horizontal line.
- **list_ver_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each vertical line.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **num_fact** (*int*) – Number of the factors of polynomial.
- **optimizing** (*bool, optional*) – Apply optimization if True.
- **threshold** (*float*) – To determine if there are missing lines. Larger is less sensitive.

Returns

list_fact (*list of float*) – Coefficients of the polynomial.

discorpy.proc.processing.**calc_coef_forward**(*list_hor_lines*, *list_ver_lines*, *xcenter*, *ycenter*, *num_fact*, *optimizing=False*, *threshold=0.3*)

Calculate the distortion coefficients of a forward mode.

Parameters

- **list_hor_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each horizontal line.
- **list_ver_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each vertical line.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **num_fact** (*int*) – Number of the factors of polynomial.
- **optimizing** (*bool, optional*) – Apply optimization if True.
- **threshold** (*float*) – To determine if there are missing lines. Larger is less sensitive.

Returns

list_fact (*list of float*) – Coefficients of the polynomial.

discorpy.proc.processing.**calc_coef_backward_from_forward**(*list_hor_lines*, *list_ver_lines*, *xcenter*, *ycenter*, *num_fact*, *optimizing=False*, *threshold=0.3*)

Calculate the distortion coefficients of a backward mode from a forward model.

Parameters

- **list_hor_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each horizontal line.
- **list_ver_lines** (*list of 2D arrays*) – List of the (y,x)-coordinates of dot-centroids on each vertical line.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **num_fact** (*int*) – Number of the factors of polynomial.
- **optimizing** (*bool, optional*) – Apply optimization if True.
- **threshold** (*float*) – To determine if there are missing lines. Larger is less sensitive.

Returns

- **list_ffact** (*list of floats*) – Polynomial coefficients of the forward model.
- **list_bfact** (*list of floats*) – Polynomial coefficients of the backward model.

discorpy.proc.processing.**transform_coef_backward_and_forward**(*list_fact*, *mapping='backward'*, *ref_points=None*)

Transform polynomial coefficients of a radial distortion model between forward mapping and backward mapping.

Parameters

- **list_fact** (*list of floats*) – Polynomial coefficients of the radial distortion model.
- **mapping** (*{'backward', 'forward'}*) – Transformation direction.
- **ref_points** (*list of 1D-arrays, optional*) – List of the (y,x)-coordinates of points used for the transformation. Generated if None given.

Returns

list of floats – Polynomial coefficients of the reversed model.

discorpy.proc.processing.**find_cod_bailey**(*list_hor_lines*, *list_ver_lines*, *iteration=2*)

Find the center of distortion (COD) using the Bailey's approach (Ref. [1]).

Parameters

- **list_hor_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each horizontal line.
- **list_ver_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each vertical line.

Returns

- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.

References

[1].. https://www-ist.massey.ac.nz/dbailey/sprg/pdfs/2002_IVCNZ_59.pdf

discorpy.proc.processing.**regenerate_grid_points_parabola**(*list_hor_lines*, *list_ver_lines*, *perspective=True*)

Regenerating grid points by finding cross points between horizontal lines and vertical lines using their parabola coefficients.

Parameters

- **list_hor_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each horizontal line.
- **list_ver_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each vertical line.
- **perspective** (*bool, optional*) – Apply perspective correction if True.

Returns

- **new_hor_lines** (*list of 2D-arrays*) – List of the updated (y,x)-coordinates of points on each horizontal line.
- **new_ver_lines** (*list of 2D-arrays*) – List of the updated (y,x)-coordinates of points on each vertical line.

discorpy.proc.processing.**regenerate_grid_points_linear**(*list_hor_lines*, *list_ver_lines*)

Regenerating grid points by finding cross points between horizontal lines and vertical lines using their linear coefficients.

Parameters

- **list_hor_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each horizontal line.
- **list_ver_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each vertical line.

Returns

- **new_hor_lines** (*list of 2D-arrays*) – List of the updated (y,x)-coordinates of points on each horizontal line.
- **new_ver_lines** (*list of 2D-arrays*) – List of the updated (y,x)-coordinates of points on each vertical line.

discorpy.proc.processing.**generate_undistorted_perspective_lines**(*list_hor_lines*, *list_ver_lines*, *equal_dist=True*, *scale='mean'*, *optimizing=True*)

Generate undistorted lines from perspective lines.

Parameters

- **list_hor_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each horizontal line.
- **list_ver_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each vertical line.
- **equal_dist** (*bool*) – Use the condition that lines are equidistant if True.
- **scale** (*{'mean', 'median', 'min', 'max', float}*) – Scale option for the undistorted grid.
- **optimizing** (*bool*) – Apply optimization for finding line-distance if True.

Returns

- **list_uhor_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on undistorted horizontal lines.
- **list_uver_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on undistorted vertical lines.

discorpy.proc.processing.**generate_source_target_perspective_points**(*list_hor_lines*, *list_ver_lines*, *equal_dist=True*, *scale='mean'*, *optimizing=True*)

Generate source points (distorted) and target points (undistorted).

Parameters

- **list_hor_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each horizontal line.
- **list_ver_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each vertical line.
- **equal_dist** (*bool*) – Use the condition that lines are equidistant if True.
- **scale** (*{'mean', 'median', 'min', 'max', float}*) – Scale option for the undistorted grid.
- **optimizing** (*bool*) – Apply optimization for finding line-distance if True.

Returns

- **source_points** (*list of 1D-arrays*) – List of the (y,x)-coordinates of distorted points.
- **target_points** (*list of 1D-arrays*) – List of the (y,x)-coordinates of undistorted points.

discorpy.proc.processing.**generate_4_source_target_perspective_points**(*points*, *input_order='yx'*, *equal_dist=False*, *scale='mean'*)

Generate 4 rectangular points corresponding to 4 perspective-distorted points.

Parameters

- **points** (*list of 1D-arrays*) – List of the coordinates of 4 perspective-distorted points.
- **input_order** (*{'yx', 'xy'}*) – Order of the coordinates of input-points.
- **equal_dist** (*bool*) – Use the condition that the rectangular making of 4-points is square if True.
- **scale** (*{'mean', 'min', 'max', float}*) – Scale option for the undistorted points.

Returns

- **source_points** (*list of 1D-arrays*) – List of the (y,x)-coordinates of distorted points.
- **target_points** (*list of 1D-arrays*) – List of the (y,x)-coordinates of undistorted points.

discorpy.proc.processing.**calc_perspective_coefficients**(*source_points*, *target_points*, *mapping='backward'*)

Calculate perspective coefficients of a matrix to map from source points to target points (Ref. [1]). Note that the coordinate of a point are in (y,x)-order. This is to be consistent with other functions in the module.

Parameters

- **source_points** (*array_like*) – List of the (y,x)-coordinates of distorted points.
- **target_points** (*array_like*) – List of the (y,x)-coordinates of undistorted points.
- **mapping** (*{'backward', 'forward'}*) – To select mapping direction.

Returns
array_like – 1D array of 8 coefficients.

References

[1].. https://doi.org/10.1016/S0262-8856(98)00183-8

`discorpy.proc.processing.`**`update_center`**(*list_lines*, *xcenter*, *ycenter*)

Update the coordinate-center of points on lines.

Parameters

- **list_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on lines.
- **xcenter** (*float*) – X-origin of the coordinate system.
- **ycenter** (*float*) – Y-origin of the coordinate system.

Returns
list of 2D-arrays.

1.4.4 Post-processing

discorpy.post.postprocessing

Module of post-processing methods:

- Correct distortion for a line of dots or an image.
- Generate unwarped slices of a 3D dataset.
- Calculate the residual of undistorted dots.

Functions:

unwarp_line_forward(list_lines, xcenter, ...)	Unwarp lines of dot-centroids using a forward model.
unwarp_line_backward(list_lines, xcenter, ...)	Unwarp lines of dot-centroids using a backward model.
unwarp_image_backward(mat, xcenter, ycenter, ...)	Unwarp an image using a backward model.
unwarp_image_forward(mat, xcenter, ycenter, ...)	Unwarp an image using a forward model.
unwarp_slice_backward(mat3D, xcenter, ...)	Generate an unwarped slice [:,index.:] of a 3D dataset, i.e. one unwarped sinogram of a 3D tomographic data.
unwarp_chunk_slices_backward(mat3D, xcenter, ...)	Generate a chunk of unwarped slices [:,start_index: stop_index, :] used for tomographic data.
calc_residual_hor(list_ulines, xcenter, ycenter)	Calculate the distances of unwarped dots (on each horizontal line) to each fitted straight line which is used to assess the straightness of unwarped lines.
calc_residual_ver(list_ulines, xcenter, ycenter)	Calculate the distances of unwarped dots (on each vertical line) to each fitted straight line which is used to assess the straightness of unwarped lines.
check_distortion(list_data)	Check if the distortion is significant or not.
correct_perspective_line(list_lines, list_coef)	Apply perspective correction to lines.
correct_perspective_image(mat, list_coef[, ...])	**Parameters** • **mat** (*array_like*) -- 2D array. Image for correction.

discorpy.post.postprocessing.**unwarp_line_forward**(*list_lines*, *xcenter*, *ycenter*, *list_fact*)

Unwarp lines of dot-centroids using a forward model.

Parameters

- **list_lines** (*list of 2D arrays*) – List of the coordinates of dot-centroids on each line.
- **list_fact** (*list of floats*) – Polynomial coefficients of the forward model.

Returns

list_ulines (*list of 2D arrays*) – List of the unwarped coordinates of dot-centroids on each line.

discorpy.post.postprocessing.**unwarp_line_backward**(*list_lines*, *xcenter*, *ycenter*, *list_fact*)

Unwarp lines of dot-centroids using a backward model. The method finds the coordinates of undistorted points from the coordinates of distorted points using numerical optimzation.

Parameters

- **list_lines** (*list of 2D arrays*) – List of the coordinates of dot-centroids on each line.
- **list_fact** (*list of floats*) – Polynomial coefficients of the backward model.

Returns

list_ulines (*list of 2D arrays*) – List of the unwarped coordinates of dot-centroids on each line.

discorpy.post.postprocessing.**unwarp_image_backward**(*mat*, *xcenter*, *ycenter*, *list_fact*, *order=1*, *mode='reflect'*)

Unwarp an image using a backward model.

Parameters

- **mat** (*array_like*) – 2D array.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.
- **order** (*int, optional.*) – The order of the spline interpolation.
- **mode** (*{'reflect', 'grid-mirror', 'constant', 'grid-constant', 'nearest',}* – 'mirror', 'grid-wrap', 'wrap'}, optional To determine how to handle image boundaries.

Returns

array_like – 2D array. Distortion-corrected image.

discorpy.post.postprocessing.**unwarp_image_forward**(*mat*, *xcenter*, *ycenter*, *list_fact*)

Unwarp an image using a forward model. Should be used only for assessment due to the problem of vacant pixels.

Parameters

- **mat** (*float*) – 2D array.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of floats*) – Polynomial coefficients of the forward model.

Returns

array_like – 2D array. Distortion-corrected image.

discorpy.post.postprocessing.**unwarp_slice_backward**(*mat3D*, *xcenter*, *ycenter*, *list_fact*, *index*)

Generate an unwarped slice [:,index.:] of a 3D dataset, i.e. one unwarped sinogram of a 3D tomographic data.

Parameters

- **mat3D** (*array_like*) – 3D array. Correction is applied along axis 1.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of floats*) – Polynomial coefficients of a backward model.
- **index** (*int*) – Index of the slice

Returns

array_like – 2D array. Distortion-corrected slice.

discorpy.post.postprocessing.**unwarp_chunk_slices_backward**(*mat3D*, *xcenter*, *ycenter*, *list_fact*, *start_index*, *stop_index*)

Generate a chunk of unwarped slices [:,start_index: stop_index, :] used for tomographic data.

Parameters

- **mat3D** (*array_like*) – 3D array. Correction is applied along axis 1.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of floats*) – Polynomial coefficients of a backward model.
- **start_index** (*int*) – Starting index of slices.
- **stop_index** (*int*) – Stopping index of slices.

Returns

array_like – 3D array. Distortion-corrected slices.

discorpy.post.postprocessing.**calc_residual_hor**(*list_ulines*, *xcenter*, *ycenter*)

Calculate the distances of unwarped dots (on each horizontal line) to each fitted straight line which is used to assess the straightness of unwarped lines.

Parameters

- **list_ulines** (*list of 2D arrays*) – List of the coordinates of dot-centroids on each unwarped horizontal line.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.

Returns

array_like – 2D array. Each element has two values: 1) Distance of a dot to the center of distortion; 2) Distance of this dot to the nearest fitted straight line.

discorpy.post.postprocessing.**calc_residual_ver**(*list_ulines*, *xcenter*, *ycenter*)

Calculate the distances of unwarped dots (on each vertical line) to each fitted straight line which is used to assess the straightness of unwarped lines.

Parameters

- **list_ulines** (*list of 2D arrays*) – List of the coordinates of dot-centroids on each unwarped vertical line.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.

Returns

array_like – 2D array. Each element has two values: 1) Distance of a dot to the center of distortion; 2) Distance of this dot to the nearest fitted straight line.

discorpy.post.postprocessing.**check_distortion**(*list_data*)

Check if the distortion is significant or not. If the number of dots having the residual greater than 1 pixel is greater than 15% of the total number of dots, there's distortion.

Parameters
list_data (*array_like*) – List of [radius, residual] of each dot.

Returns
bool

discorpy.post.postprocessing.**correct_perspective_line**(*list_lines*, *list_coef*)

Apply perspective correction to lines.

Parameters

- **list_lines** (*list of 2D-arrays*) – List of the (y,x)-coordinates of points on each line.
- **list_coef** (*list of floats*) – Coefficients of the forward-mapping matrix.

Returns
list_clines (*list of 2D arrays*) – List of the corrected (y,x)-coordinates of points on each line.

discorpy.post.postprocessing.**correct_perspective_image**(*mat*, *list_coef*, *order=1*, *mode='reflect'*, *map_index=None*)

Parameters

- **mat** (*array_like*) – 2D array. Image for correction.
- **list_coef** (*list of floats*) – Coefficients of the backward-mapping matrix.
- **order** (*int, optional.*) – The order of the spline interpolation.
- **mode** (*{'reflect', 'grid-mirror', 'constant', 'grid-constant', 'nearest',*) – 'mirror', 'grid-wrap', 'wrap'}, optional To determine how to handle image boundaries.
- **map_index** (*array_like*) – Indices for mapping. Generated if None is given.

Returns
array_like – Corrected image.

1.4.5 Utility methods

discorpy.util.utility

Module of utility methods:

- Generate a dot-pattern, line-pattern, and chessboard image.
- Find corresponding points between the distorted and undistorted space.
- Unwarp a color image with an option to keep the original field of view.
- Unwarping an image or video using the 'remap' function in Opencv for fast performance.

Functions:

make_dot_pattern([height, width, ...])	Generate a dot-pattern image.
make_line_pattern([height, width, ...])	Generate a dot-pattern image.
make_chessboard([height, width, size, ...])	Generate a chessboard image.
find_point_to_point(points, xcenter, ...[, ...])	Calculate corresponding points between distorted and undistorted space.
unwarp_color_image_backward(mat, xcenter, ...)	Unwarp a color image using a backward model.
mapping_cv2(mat, xmat, ymat[, method, border])	Apply a geometric transformation to a 2D array using Opencv.
unwarp_image_backward_cv2(mat, xcenter, ...)	Unwarp an image using the backward model with the Opencv 'remap' function for fast performance.
unwarp_video_cv2(cam_obj, xcenter, ycenter, ...)	Unwarp frames from Opencv video object using the backward model.

discorpy.util.utility.**make_circle_mask**(*width*, *ratio*)

Create a circle mask.

Parameters

- **width** (*int*) – Width of a square array.
- **ratio** (*float*) – Ratio between the diameter of the mask and the width of the array.

Returns

array_like – Square array.

discorpy.util.utility.**make_dot_pattern**(*height=1800*, *width=2000*, *dot_distance=90*, *dot_size=15*, *margin=150*)

Generate a dot-pattern image.

Parameters

- **height** (*int*) – Height of the image.
- **width** (*int*) – Width of the image.
- **dot_distance** (*int*) – Distance between two dots.
- **dot_size** (*int*) – Size of each dot.
- **margin** (*int*) – Blank area between the dots and the edges.

Returns

array_like – Dot-pattern image.

discorpy.util.utility.**make_line_pattern**(*height=1800*, *width=2000*, *line_distance=90*, *line_size=7*, *margin=100*)

Generate a dot-pattern image.

Parameters

- **height** (*int*) – Height of the image.
- **width** (*int*) – Width of the image.
- **line_distance** (*int*) – Distance between two dots.
- **line_size** (*int*) – Size of each dot.
- **margin** (*int*) – Blank area between the lines and the edges.

Returns

array_like – Dot-pattern image.

discorpy.util.utility.**make_chessboard**(*height=1800*, *width=2000*, *size=100*, *margin=100*, *margin_grayscale=0.95*)

Generate a chessboard image.

Parameters

- **height** (*int*) – Height of the image.
- **width** (*int*) – Width of the image.
- **size** (*int*) – Size of each cell.
- **margin** (*int*) – Blank area between the chessboard and the edges.
- **margin_grayscale** (*float*) – Gray scale of margin area (0: black 1: white).

Returns

array_like – Chessboard image.

discorpy.util.utility.**find_point_to_point**(*points*, *xcenter*, *ycenter*, *list_fact*, *output_order='xy'*)

Calculate corresponding points between distorted and undistorted space. This function can be used both ways: - Given the input is a distorted point and a forward model, the output is the undistorted point. - Given the input is an undistorted point and a backward model, the output is the distorted point.

Parameters

- **points** (*tuple of point indexes.*) – Tuple of (row_index, column_index) of the point. Note that the format is ij-index not xy-coordinate.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward/forward model.
- **output_order** (*{"xy", "yx"}*) – To select the order of the output. "yx" <-> "ij".

Returns

tuple of float – x- and y-coordinate of the point in another space.

discorpy.util.utility.**unwarp_color_image_backward**(*mat*, *xcenter*, *ycenter*, *list_fact*, *order=1*, *mode='reflect'*, *pad=False*, *pad_mode='constant'*)

Unwarp a color image using a backward model.

Parameters

- **mat** (*array_like*) – 2D/3D array.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.
- **order** (*int, optional.*) – The order of the spline interpolation.
- **mode** (*{'reflect', 'grid-mirror', 'constant', 'grid-constant', 'nearest',}* – 'mirror', 'grid-wrap', 'wrap'}, optional To determine how to handle image boundaries.
- **pad** (*bool, int, or tuple of int.*) – Use to keep the original view. If pad is True, the pad width is calculated automatically.
- **pad_mode** (*str*) – To select a method for padding: 'reflect', 'edge', 'mean', 'linear_ramp', 'symmetric',...

Returns

array_like – 2D/3D array. Distortion-corrected image.

discorpy.util.utility.**mapping_cv2**(*mat*, *xmat*, *ymat*, *method=None*, *border=None*)

Apply a geometric transformation to a 2D array using Opencv.

Parameters

- **mat** (*array_like.*) – 2D/3D array.
- **xmat** (*array_like*) – 2D array of the x-coordinates. Origin is at the left of the image.
- **ymat** (*array_like*) – 2D array of the y-coordinates. Origin is at the top of the image.
- **method** (*opencv-object*) – To select interpolation method. Note to use the prefix: cv2.<method> https://tinyurl.com/3afmv6jc
- **border** (*opencv-object*) – To select method for boundary handling. Note to use the prefix: cv2.<method> https://tinyurl.com/52xzkwt2

Returns

array_like

discorpy.util.utility.**unwarp_image_backward_cv2**(*mat*, *xcenter*, *ycenter*, *list_fact*, *method=None*, *border=None*, *pad=False*, *pad_mode='constant'*)

Unwarp an image using the backward model with the Opencv 'remap' function for fast performance.

Parameters

- **mat** (*array_like*) – Input image. Can be a color image.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.
- **method** (*opencv-object*) – To select interpolation method. Note to use the prefix: cv2.<method> https://tinyurl.com/3afmv6jc
- **border** (*opencv-object*) – To select method for boundary handling. Note to use the prefix: cv2.<method> https://tinyurl.com/52xzkwt2
- **pad** (*bool, int, or tuple of int.*) – Use to keep the original view. If pad is True, the pad width is calculated automatically.
- **pad_mode** (*str*) – To select a method for padding: 'reflect', 'edge', 'mean', 'linear_ramp', 'symmetric',...

Returns

array_like – Distortion-corrected image.

discorpy.util.utility.**unwarp_video_cv2**(*cam_obj*, *xcenter*, *ycenter*, *list_fact*, *method=None*, *border=None*, *pad=True*, *pad_mode='constant'*)

Unwarp frames from Opencv video object using the backward model.

Parameters

- **cam_obj** (*obj*) – Opencv camera object. e.g. cv2.VideoCapture(0)
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.
- **method** (*opencv-object*) – To select interpolation method. Note to use the prefix: cv2.<method> https://tinyurl.com/3afmv6jc
- **border** (*opencv-object*) – To select method for boundary handling. Note to use the prefix: cv2.<method> https://tinyurl.com/52xzkwt2
- **pad** (*bool, int, or tuple of int.*) – Use to keep the original view. If pad is True, the pad width is calculated automatically.

- **pad_mode** (*str*) – To select a method for padding: 'reflect', 'edge', 'mean', 'linear_ramp', 'symmetric',...

Returns
Generator

1.5 Credit

If Discorpy is useful for your project, citing the following article [C1] is very much appreciated.

Algorithms, methods, or techniques implemented in a scientific software package are crucial for its success. This is the same for Discorpy. Acknowledging algorithms you use through Discorpy is also very much appreciated.

REFERENCES

BIBLIOGRAPHY

[C1] Nghia T. Vo, Robert C. Atwood, and Michael Drakopoulos. Radial lens distortion correction with sub-pixel accuracy for x-ray micro-tomography. *Opt. Express*, 23(25):32859–32868, Dec 2015. URL: https://www.osapublishing.org/abstract.cfm?uri=oe-23-25-32859, doi:10.1364/OE.23.032859.

[R1] T. A. Clarke and J. G. Fryer. The development of camera calibration methods and models. *The Photogrammetric Record*, 16(91):51–66, 1998. URL: https://onlinelibrary.wiley.com/doi/epdf/10.1111/0031-868X.00113, doi:10.1111/0031-868X.00113.

[R2] Carlos Ricolfe-Viala and Antonio-Jose Sanchez-Salmeron. Lens distortion models evaluation. *Appl. Opt.*, 49(30):5914–5928, Oct 2010. URL: http://www.osapublishing.org/ao/abstract.cfm?URI=ao-49-30-5914, doi:10.1364/AO.49.005914.

[R3] A. Criminisi, I. Reid, and A. Zisserman. A plane measuring device. *Image and Vision Computing*, 17(8):625–634, 1999. URL: https://www.sciencedirect.com/science/article/pii/S0262885698001838, doi:https://doi.org/10.1016/S0262-8856(98)00183-8.

[R4] Nghia T. Vo. Python implementation of distortion correction methods for x-ray tomography. *zenodo.org*, 2018. URL: https://zenodo.org/record/1322720#.YV7_k9rTVaQ, doi:10.5281/zenodo.1322720.

[R5] Donald G. Bailey. A new approach to lens disortion correction. *Proceedings of Image and Vision Computing New Zealand Conference*, pages 59–64, 2002. URL: https://www-ist.massey.ac.nz/dbailey/sprg/pdfs/2002_IVCNZ_59.pdf.

PYTHON MODULE INDEX

d

INDEX

M

N

O

R

S

T

U