Undistortion

NCS Lab

November 23, 2021

Correcting for lens distortion in iniVation's event-based cameras, requires following two procedures: (1) Camera Calibration and (2) Pixel Mapping. A github repository is available (see here) to perform such.

1 Camera / Lens Calibration

The following procedure applies for 'mono' configurations (single camera), and requires iniVation's DV-GUI, which can be downloaded here. For a more generic procedure, please consult here. For information on stereo calibration please consult here.

In this tutorial a 6 x 9 chessboard with a square size of 30 mm is used (available here). In the DV software, the width should be set to 9, the height to 6, and the square size to 30; these are already the default values. It should be noted that the square size can be expressed in any length scale. For alternatives, please consult here.

Step 1: Open calibration project in DV-GUI and go to 'Structure' Tab. The modules do the following:

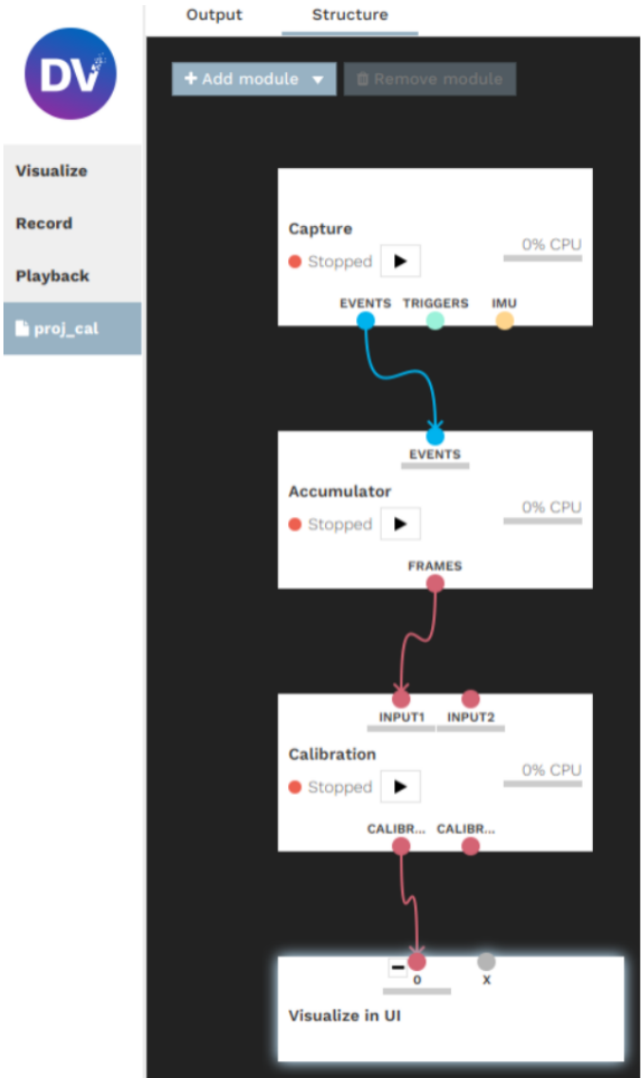

Figure 1: Structure Tab

1. **Capture:** streams events from camera to computer
2. **Accumulator:** adds up events that happen within a certain interval, to generate a pseudo-frame
3. **Calibration:** uses OpenCV to obtain, among other parameters, the:
    - Camera matrix (center of distortion and focal length)
    - Distortion coefficients (radial and tangential)
4. **Visualize in UI:** shows artificial frames in a window

Step 2: Go to the 'Output' tab once all the modules are started (by clicking on their corresponding 'play' button) and start moving (presenting) the calibration target to the camera. The following behavior is expected:

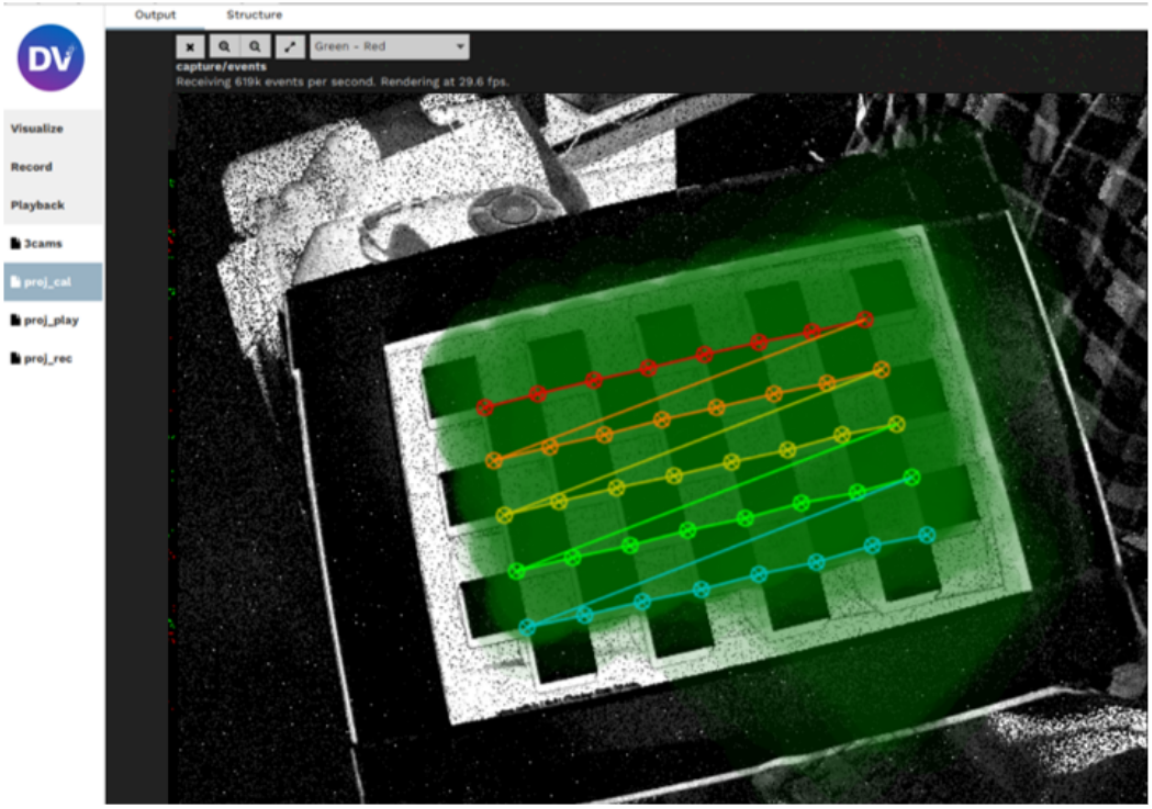

Figure 2: Output Tab

It is important that the calibration target is presented to the entire field of view. The green 'shadow' in the previous image indicates the region of the field-of-view for which data has been generated. Since lens distortion has more impact around the edges, it is key to present the calibration target close to them.

Step 3: Repeat step 2, until DV-GUI has generated enough frames ('enough' is determined by the parameter 'Min Detection' from the 'Calibration' module).

Step 4: Once DV-GUI has enough images, it will require visual inspection to keep/discard good/bad images ('good' is when the colorful circles/lines match with the corners of the squares in the chessboard).

Step 5: If during step 4 one or more images we're discarded, then it is necessary to create replacement images. This means jump back to step 3. If, on the other hand, all images are kept, then DV-GUI will internally use OpenCV to generate an *.xml file with the camera calibration parameters.

An example of how the image generation/checking/keeping/discarding works:

DV-GUI will show a real-time visualization of the scene (see figure above) while generating the frames. The visualization will freeze when enough frames have been stored. At that point, one can go over all the stored frames, click on 'Check Images' and decide either to 'Keep' them or 'Discard' them (see calibration menu on the right, shown in Figure below). Once a decision is made on one stored frame, the next one will be shown until there are no more images to make decisions about. If 'Min Detection' is set to 30, the visualization will freeze when 30 images are stored. If 5 of such images are not representative of the pattern or the camera distortion

(it is up to the user to judge this based on, for instance, how far from the camera or how close the screen edges the pattern is), one can 'Discard' them. Once all the stored images have been reviewed, the visualization will unfreeze so 5 more images (replacing the discarded ones) can be generated/stored. The user must, again, make a decision for each new image. This until 30 images have been kept. Once the calibration algorithm has enough images to work with, it will automatically start and the outcome, an *.xml file, will be stored

2 Pixel Mapping

In the Figure below, the yellow square, the 'real' 2D projection of space, consists of pixels of type (x,y) whereas the green square, the 'chip' space, consists of pixels of type (x',y'). The barrel distortion generated by the lens will make the camera 'think' that some events happening at (x, y) in the 'real' 2D projection of space occur at (x', y') in the 'chip' space. Moreover, some pixels of type (x,y) outside the yellow square will be assigned to some pixels of type (x',y'). This alteration of reality can be described by radial and tangential components of distortion, which are dependent on the intrinsic characteristics of the camera/lens setup (see equations and trivial software implementation below). These characteristics are given in a camera matrix and an array of distortion coefficients. Both, the matrix and the array, are outputs of the calibration procedure and are shown in the *.xml file produced by the DV-GUI.

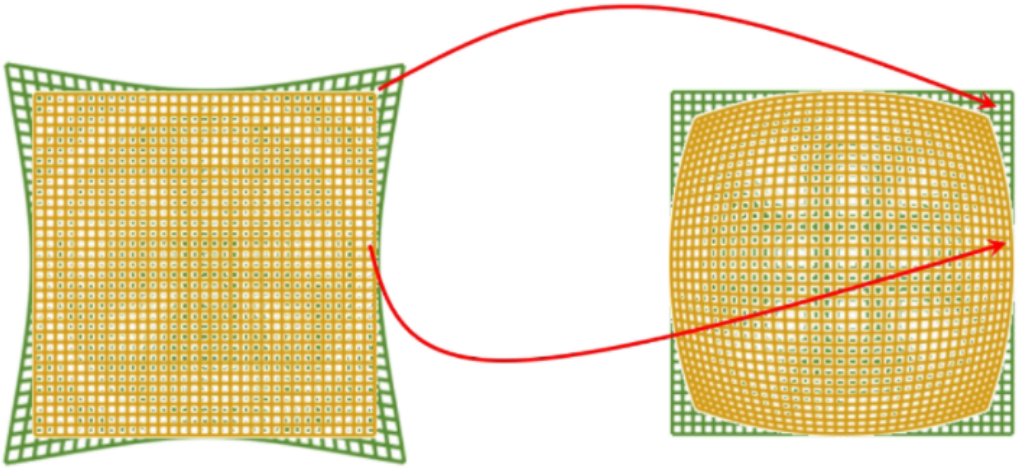

Figure 3

2.1 Equations

$$C_{Mat} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$D_{coeff} = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix}$$

$$r^2 = x^2 + y^2$$

$$x_d = [x \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 + r^6)] + [2p_1 xy + p_2 \cdot (r^2 + 2x^2)]$$

$$y_d = [y \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)] + [p_1 \cdot (r^2 + 2y^2) + 2p_2 xy]$$

2.2 Implementation

```python
for i in range(w): #640
    for j in range(h): #480

        # To relative coordinates
        x = (i-c_x)/f_x
        y = (j-c_y)/f_y
        r_2 = x*x + y*y

        # Radial Distortion
        x_d = x*(1 + k_1 * r_2 + k_2 * r_2 * r_2 + k_3 * r_2 * r_2 * r_2)
        y_d = y*(1 + k_1 * r_2 + k_2 * r_2 * r_2 + k_3 * r_2 * r_2 * r_2)

        # Tangential Distortion
        x_d = x_d + (2 * p_1 * x * y + p_2 * (r_2 + 2 * x * x))
        y_d = y_d + (p_1 * (r_2 + 2 * y * y) + 2 * p_2 * x * y)

        # To absolute coordinate
        x_d = x_d*f_x + c_x
        y_d = y_d*f_y + c_y

        if ((x_d>=0) and (x_d<w)) and ((y_d>=0) and (y_d<h)):
            dst[j, i, :] = img[int(y_d), int(x_d), :]
```

Figure 4: Snippet of software implementation; for more click here

2.3 Look-Up Tables (LUTs)

OpenCV uses these equations to find for each x, y the corresponding x_d,y_d to then extract the content of pixel (x_d, y_d) from the distorted image (the source) and put it in pixel (x,y) in the undistorted one (the destination). This works well for RGB images since at any instant the algorithm has access to all the pixels in the distorted image. However, when it comes to events, this no longer applies. In fact, to be able to map events from distorted to undistorted spaces, the opposite procedure is required: for each x_d, y_d the corresponding x, y needs to be calculated so the event happening at (x_d, y_d) in the chip can get mapped to the appropriate (x,y) in the accurate (non-distorted) representation of the 2D projection of space. An analytical solution to this would require inverting the problem described in the equations above, i.e. finding x,y as a function of x_d, y_d. This computational complexity can't be afforded. For this reason, an alternative method using a look-up table is used. The CAMagic repository can be used to create the LUT for any camera once the calibration procedure has been done. The arborescence should look like this (for a setup with 3 cameras):

```
CAMagic/
├── calibration
│   ├── dv_gui
│   │   ├── proj_cal.xml
│   │   ├── proj_play.xml
│   │   └── proj_rec.xml
│   └── parameters
│       ├── cam1.xml
│       ├── cam2.xml
│       └── cam3.xml
├── images
│   ├── distorted_1.png
│   ├── distorted_2.png
│   └── distorted_3.png
├── recordings
│   ├── rec_cam1.aedat4
│   ├── rec_cam2.aedat4
│   └── rec_cam3.aedat4
├── undistortion
│   ├── cam_1.json
│   ├── cam_2.json
│   └── cam_3.json
├── README.md
├── Undistorter.ipynb
└── Visualizer.ipynb
```

Figure 5

The whole process, from camera to *.json is shown in the figure below:

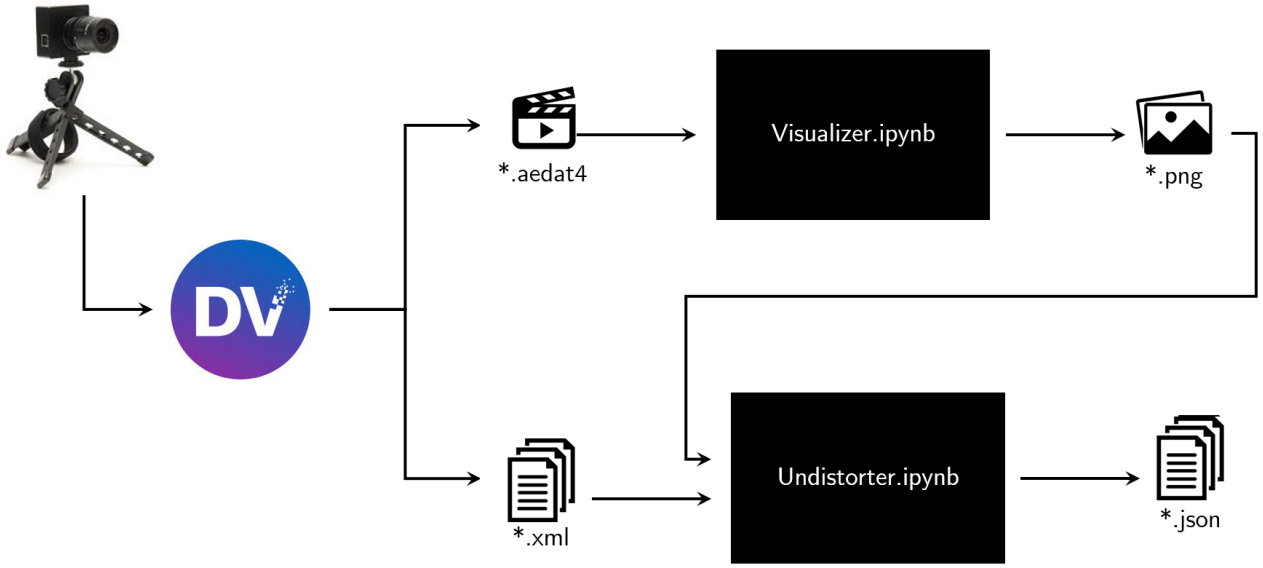

Figure 6

The *.xml generated by the DV-GUI should be stored in *CAMagic/calibration/parameters* using the following name convention: **cam<N>.xml**, where 'N' is the camera ID used in DV-GUI. Distorted recordings in *.aedat4 format should be stored in *CAMagic/recordings* using the following name convention: **rec*cam<N>.aedat4**. These recordings are events, however, for 'fast/easy' visual inspection it's worth generating one representative/meaningful distorted image out of the corresponding recording. That can be done by running CAMagic/Visualizer.ipynb, which will store the output in *CAMagic/images* using the following name convention: **distorted*<N>.png**. With *.xml and .*png ready, the CAMagic/Undistorter.ipynb can be run to generate, for each of the cameras, the 'optimal' LUT and store it in *CAMagic/undistortion* using the following name convention: **cam*<N>.json**.

2.4 Visualizer.ipynb

The Visualizer.ipynb accumulates events within a preset time frame and stores an the k-th image it finds in which a predefined percentage of the pixels shows activity. The value for 'k' is selected upon visual inspection, since the idea is to select a 'frame' where barrel distortion is clear/evident so, when performing undistortion, the results are also clear/evident. By default, the activity threshold is 1

2.5 Undistorter.ipynb

The Undistorter.ipynb finds the 'optimal' LUT for a camera so the coordinates of the events reported in 'chip' space are properly mapped, in real-time, to the accurate 2D representation of 'real' space. Here, 'optimal' means that the number of inactive pixels (in the undistorted image) as well as the number of unused pixels (from the distorted image) are minimized. The following images explain the problem; green indicates 'chip' space; yellow indicates 'real' space, blue indicates undistorted space. When there is no distortion, all these spaces should map one-to-one. However, that's not the general case, as shown in the figures below. **Case №1** shows how the 'real' space is distorted in the 'chip' space and how, when mapping to undistorted space, a non-negligible amount of pixels on that space don't receive information from 'chip' pixels. **Case №2** shows how when trying to map 'real' distorted pixels one-to-one with pixels in the undistorted space, some 'chip' pixels are mapped outside that undistorted space leading to information loss, i.e. pixels in the 'chip' space are unused. Finally, **Case №3** shows a trade-off between unused pixels, from the 'chip' space, and inactive pixels, in the undistorted space. The latter case is the one that Undistorter.ipynb tries to find.

2.5.1 Case №1

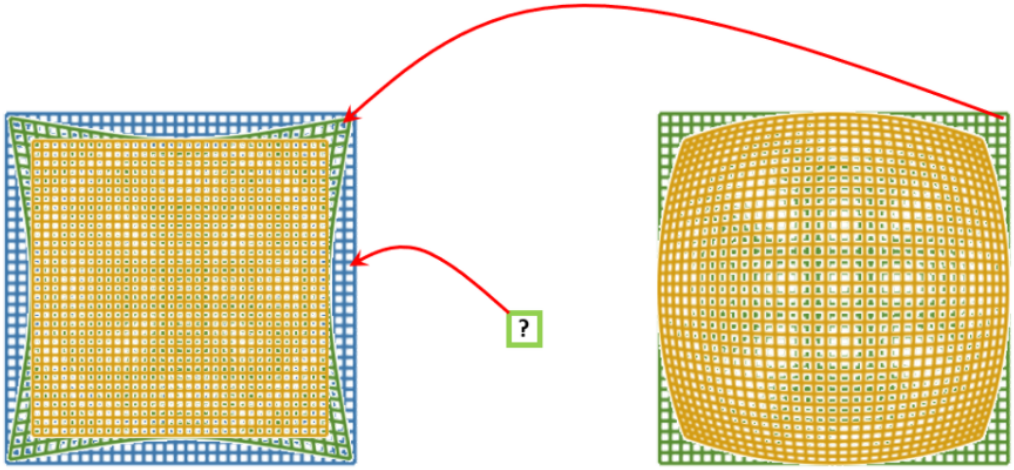

Figure 7

2.5.2 Case №2

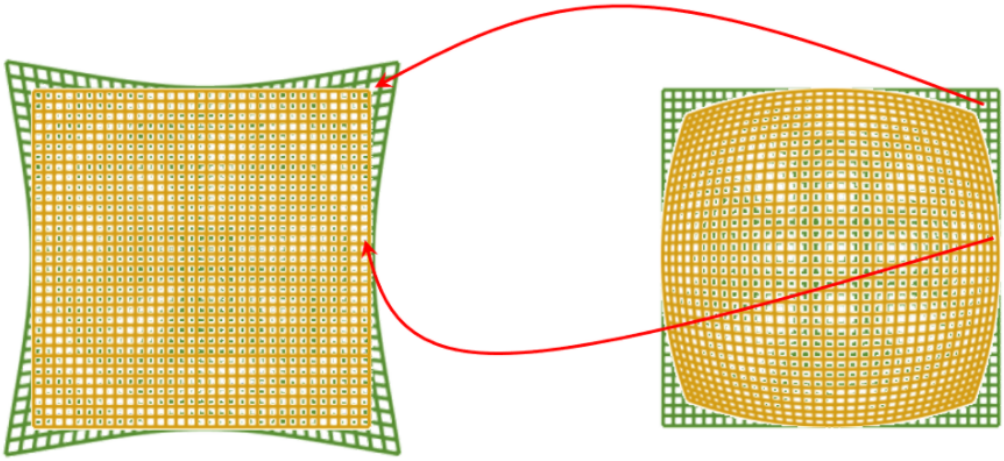

Figure 8

2.5.3 Case №3

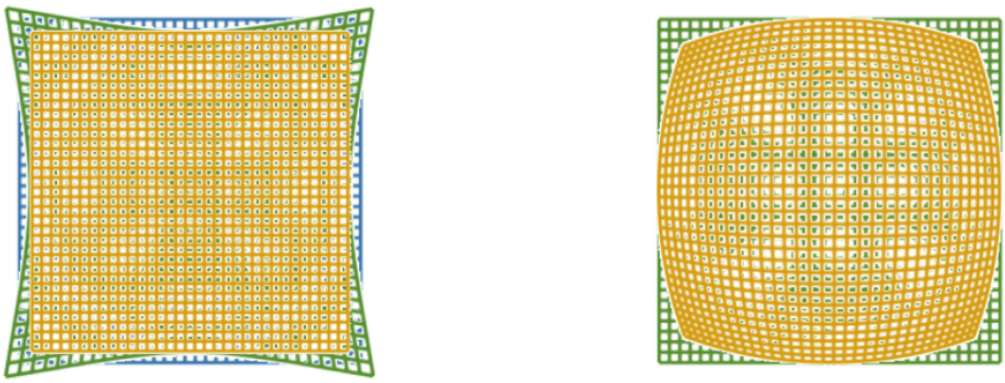

Figure 9