CSE 264 Project 2
Joseph Adamson
June 11, 2020
Using Samsung Galaxy S7 Camera (Resolution 4032x3024)

Using the Script

The provided python script generates all deliverables and can be run by itself with the command "python main_script.py". It was developed in python3 but also tested to work with python2 as long as the cv2 and numpy modules are available. If buggy in python2, try python3. The run directory for main_script.py must also include the calibrator.py and line_getter.py scripts, as well as a chessboard images folder containing calibration images. The image to perform the homography on must be named "image_raw"

Calibrator

If the file 'params.txt' is not present in the run directory, or if the always_calibrate option is set, main calls the calibrator.py script, which reads images in the 'chessboard_images' folder and performs opencv calibration. Anytime a new chessboard dataset is used, 'params.txt' must be deleted or the calibrator script can be run by itself.

Line Getter

If the files 'lines_v.txt' and 'lines_h.txt' are not present in the run directory, or if the always_get_lines option is set, main calls the line_getter.py script, which I wrote for the projcet. It prompts the user to manually draw bundles of lines by clicking and dragging. The points are saved in the corresponding files. The following key commands work during this process:

- h: Draw horizontal bundles
- v: Draw vertical bundles
- u: Undo last-drawn line
- q: finish and resume main script execution

Options

At the top of the main script are some options. To always calibrate, set always_calibrate to True. To always prompt for lines, set always_get_lines to True. To resize images during processing (speeds up processing, but affects results) set resize_images to true. The following results were performed with this flag set to False, but doing everything initially with some resizing resulted in much faster processing, especially during calibration.

The results for all parts were generated using the files in the folder "original_run" and can be recreated by copying those files to the run directory. Otherwise, running the script will generally overwrite anything in the run directory based upon the settings.

Colab Update

The script was converted to a colab notebook, found at `https://drive.google.com/open?id=1d_tb5TtNY1958KS85C69gmMneERfujBe`. The functionality is the same as the standalone .py script except that the line_getter gui won't work. One needs only to "run all", and the process can be sped up by setting "always_calibrate" to FALSE after the first run. I've already run the calibration script and thus the params text file is already there.

Part 1

Following the provided tutorials, a calibrator script was implemented using code from both the opencv github examples and some other tutorials. A checkerboard was printed and taped to a clipboard, and several images taken from different angles, with no zoom. In my project submission, these images are in the 'chessboard_images' folder. The opencv 'findChessCorners()' and 'drawChessboardCorners()' functions were used to capture the corners in each image, and then the calibrateCamera() function generates the parameters. An example output is given below. Additionally, running the calibrator script will display each image with the found chessboard corners as it generates the data.

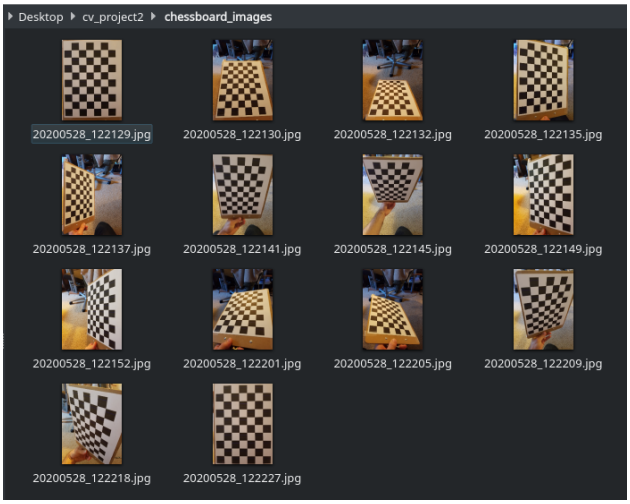
For my camera, the following parameters were found:

$$K = \begin{bmatrix} 3077.196 & 0.000 & 1460.994 \\ 0.000 & 3063.424 & 2020.021 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

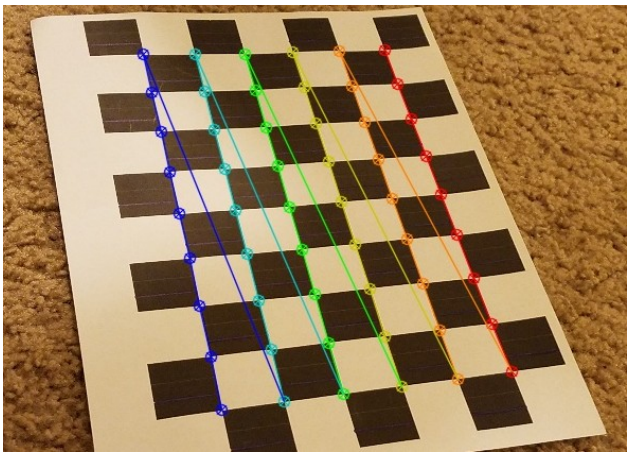
distortion coefficients =

$$\begin{bmatrix} 0.363 & -1.171 & 0.003 & -0.001 & 1.051 \end{bmatrix}$$

rms error: 1.7228



captured chessboard images



example 'findChessCorners' chessboard output

Part 2 Rectification

A few candidate pictures were captured around the UCSC campus. The picture I settled on was of the rear side of Porter B:

Deliverable 2: Raw image

opencv provides the undistort() function, which takes in the parameters generated in part1, and applies the appropriate transform to 'undo' the lens distortions. Undistortions like this are more drastic in the case of wide angle ('fisheye') lenses, but my camera didn't seem to introduce much distortion, as the undistorted image shows:

Deliverable 2: Rectified image

Part 3a Finding Lines

A short program was written to manually mark lines in an image, described under 'Using the Script' above. The program uses CV's mouse callback function and captures the start- and end-points of lines when the user drags them. A simple function was written to take 2 points (captured with mouse-drag) and draw the corresponding line to the edges of the frame. The user can specify H or V bundles. These points are written to separate files (one for each bundle) and drawn on the image. I used it to draw the following lines:

Deliverable 3.1: Rectified image with drawn lines

Part 3b Vanishing Directions

All relevant lever and vanishing-direction vectors were calculated according to the project instructions. For ease of use, I defined a getLeverVectors function, as well as an LMEDS function which takes in a full bundle of lines. My LMEDS implementation is as follows:

```
#perform least median of squares on a list of line points.
#implementing algorithm given in project assignment
def LMEDS(lineBundle):
    numLines = len(lineBundle)
    best_i = -1
    best_j = -1
    best_m = 1e9
    for rep in range(1000):
        i = random.randrange(numLines)
        j = random.randrange(numLines)
        #stops divide-by-zero when rand i==j:
        if i==j:
            continue
        n_i = getLeverVector(lineBundle[i])
        n_j = getLeverVector(lineBundle[j])
        m_ij = np.cross(n_i,n_j)
        m_ij = m_ij/np.linalg.norm(m_ij)

        #now look through the lines that are not i,j and get r for each:
        r = np.empty(0)
        for line in lineBundle:
            if line is not lineBundle[i] and line is not lineBundle[j]:
                n_k = getLeverVector(line)
                rk = abs(n_k.dot(m_ij))
                r = np.append(r,rk)

        M_m = np.median(r)
        if M_m < best_m:
            #print('new best median:', M_m)
            best_m = M_m
            best_i = i
            best_j = j

    #now get the best ones back out
    n_i_best = getLeverVector(lineBundle[best_i])
    n_j_best = getLeverVector(lineBundle[best_j])
    m_ij_best = np.cross(n_i_best,n_j_best)
    m_ij_best = m_ij_best/np.linalg.norm(m_ij_best)
    if NORM_FIX:
        m_ij_best = np.abs(m_ij_best)
    return (m_ij_best, best_m)
```

lmeds

3b deliverables

The following vectors were generated. Note that since lmeds is a random-based process, each run generates a slightly different output; so these vectors were the result of only one run:

$$\mathbf{m}_H^C = \begin{bmatrix} 0.719 & -0.038 & -0.694 \end{bmatrix}$$

$$\text{residual} = 0.00184$$

$$\mathbf{m}_V^C = \begin{bmatrix} -0.049 & 0.979 & -0.198 \end{bmatrix}$$

$$\text{residual} = 0.00423$$

$$\mathbf{m}_T^C = \begin{bmatrix} 0.688 & 0.177 & 0.703 \end{bmatrix}$$

Since LMEDS is a random process, I would occasionally have an issue where the output image would be mirrored around the y-axis, or flipped upside down, or both. I think this is due to the projection error being a magnitude rather than a signed quantity, meaning occasionally the vectors with a smaller error may be disparate enough from the actual plane that they come out backwards in the math. The solution to this problem is to always force the vanishing directions to have the same signs as those shown above, since these directions are the correct 3D-representations of those lines; forcing those directions to have the right signs gives a correct homography every time.

The angle between the horizontal and vertical directions is given below. This should be close to 90°... not bad!:

$$\measuredangle(\mathbf{m}_H^C, \mathbf{m}_V^C) = 86.28°$$

We know from the lecture topics that the vanishing point is simply the projection matrix applied to the directional vector; therefore; the vanishing points, in the sensor frame, are calculated with $K\mathbf{m}_i^C$, followed by a conversion to augmented coordinates (indicated by the overbar ie, normalizing to make the third component one) for each vector. So, the vanishing points in the sensor frame (pixel units) are:

$$[\overline{K\mathbf{m}_H^C}]_2 = x_H^S = \begin{bmatrix} -1727.859 & 2188.604 \end{bmatrix}$$

$$[\overline{K\mathbf{m}_V^C}]_2 = x_V^S = \begin{bmatrix} 2228.261 & -13088.534 \end{bmatrix}$$

$$[\overline{K\mathbf{m}_T^C}]_2 = x_T^S = \begin{bmatrix} 4471.659 & 2792.691 \end{bmatrix}$$

Part 4

Using the vectors determined above with LMEDS, the homography was obtained with the method given in the project assignment (generating W with the vanishing directions, then calculating $KW^{-1}K^{-1}$) to obtain the homography for the perspective transform. Each element in the homography is divided by H(3,3) to normalize the last element to 1:

$$H = \begin{bmatrix} 4.051 & -0.069 & -9926.458 \\ 1.370 & 4.245 & -6924.065 \\ 0.001 & 0.000 & 1.000 \end{bmatrix}$$

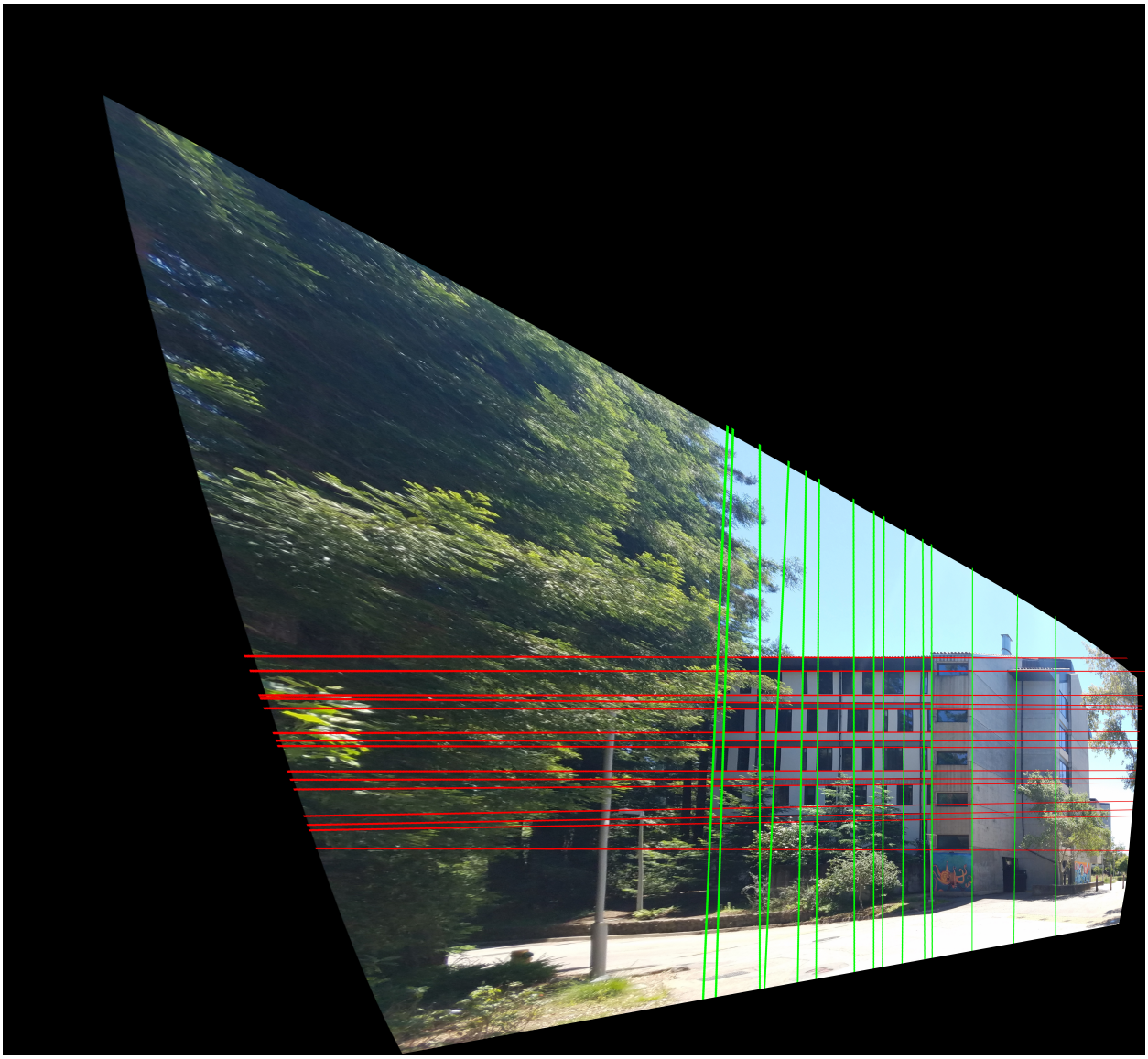
To determine the new location of the four corners after this mapping, and to make sure the new image is 'in frame', we simply transform the four corners $(0, 0), (width, 0), (0, height), (width, height)$ by augmenting them, applying the transform $H\bar{c}$, then normalizing by the third element to return to augmented:

```
corner  [0 0 1] mapped to  [-9926.458 -6924.065 1.000]
corner  [4032    0    1] mapped to  [1432.792 -313.091 1.000]
corner  [   0 3024    1] mapped to  [-6052.327 3531.706 1.000]
corner  [4032 3024    1] mapped to  [1204.684 2222.145 1.000]
```

The minimum x and y values of the corners are used to form the translation vector (b_x, b_y) which is negated and inserted into an otherwise identity matrix to form the translation matrix:

$$H_{translate} = \begin{bmatrix} 1.000 & 0.000 & 9926.458 \\ 0.000 & 1.000 & 6924.065 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

and the maximum-minimum (x,y) of the shifted corners determines the new end pixels in the image since the translation transform will already offset them. So applying the transform $H_t H$ and cropping the pixels accordingly gives the image below, which was generated using the opencv warpPerspective() function:



Deliverable 4: Perspective-shifted image. The new dimensions are (11359, 10455)