ETPoF Homework 2

Annemarie Linnenbank & Tommie Verouden

8 March 2024

N.B.: Apologies for including all code within the text below. We did not have time to properly bundle our code in an appendix, but we will make sure to do this next time.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import scipy.spatial as spatial
from sklearn.cluster import DBSCAN
from scipy.optimize import curve_fit
```

Part a: Optical Calibration (13 points)

In this part, we work towards a fully-fledged optical calibration algorithm. For clarity, we will go through all the steps contained within the function in the exercises, so we can show some intermediate results too. For future use, the steps in part a3 could be collected into a single function.

a1) **Image resolution** (5 pts)

We start by importing the image using the OpenCV library.

In [ ]:
```
# Import the first image in grayscale
img_path = "Images/Calibration_a/Clean.png"
img = cv.imread(img_path, 0)
```

(i) Circle detection

We apply a Hough circle transform to detect circles in the image, after which we remove circles that overlap with the image border. The parameters of the detector have been determined empirically, and we used the `HOUGH_GRADIENT_ALT` algorithm, since it is supposed to be more accurate (see https://docs.opencv.org/4.x/dd/d1a/group__imgproc__feature.html#gga073687a5b96ac7a3ab5802eb5510fe65aad57c72131c801de427f

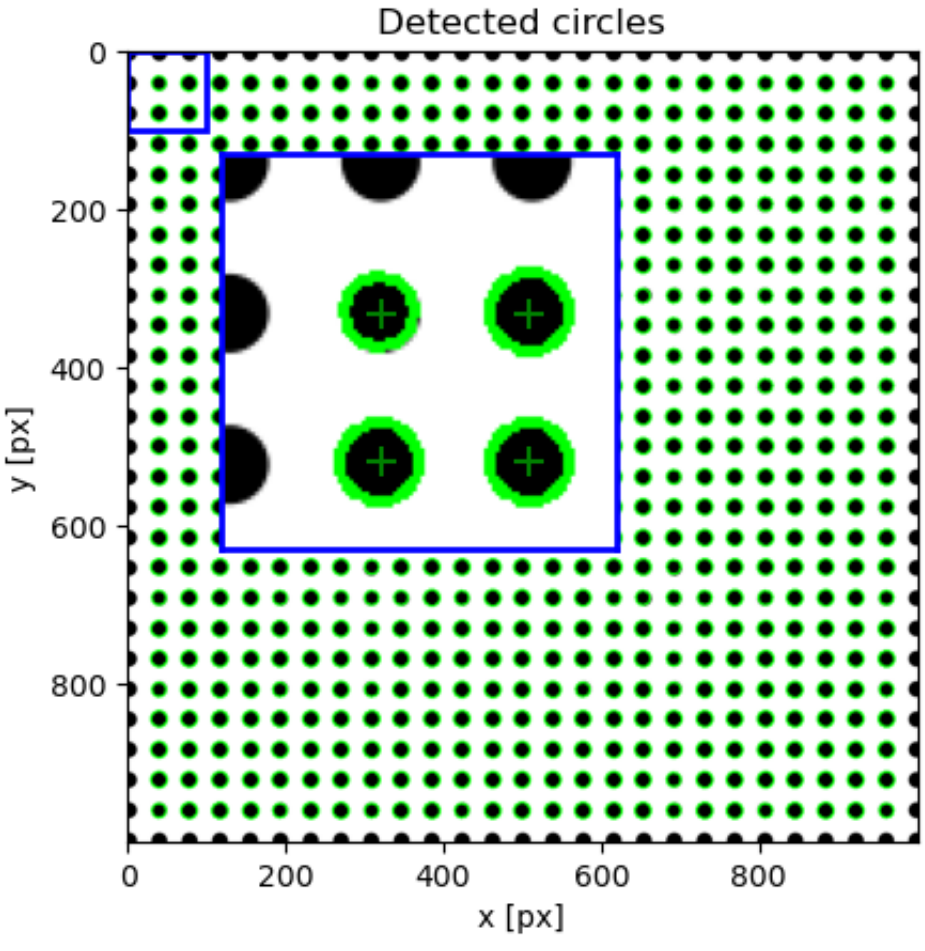
The distance and radii parameters are estimated by counting the circles per pixel by eye.

In [ ]:
```
def detect_circles(img_edges, min_dist, min_radius, max_radius, algorithm = cv.HOUGH_GRADIENT_ALT,
                   dP = 1.5, canny_thresh = 100, parameter2 = 0.8, boundary_px = 0,
                   verbose = False):
    '''Detect circles in an edge-detected image using the Hough transform

    PARAMETERS:
        img_edges (np.array): The edge-detected image
        min_dist (int): Minimum distance between the centers of the detected circles
        min_radius (int): Minimum radius of the detected circles
        max_radius (int): Maximum radius of the detected circles
        dP (int): Inverse ratio of the accumulator resolution to the image resolution
        canny_thresh (int): The upper threshold for the Canny edge detector
        parameter2 (int): How circular the circles should be in the ALT algorithm

    RETURNS:
        circles (np.array): The detected circles
    '''

    # === DETECTION ===
    # Apply Hough transform on the edge-detected image
    circles = cv.HoughCircles(img_edges, algorithm,
                              dP, min_dist, param1 = canny_thresh, param2 = parameter2,
                          minRadius = min_radius, maxRadius = max_radius)

    # Warn when no circles are detected
    if circles is None:
        print('No circles detected!')
        return None, None
    elif verbose:
        print(f'{len(circles[0, :, :])} circles detected')

    # Convert the circles to a list, split array
```

```
    circles = circles[0, :, :]
    centers = circles[:,0:2]
    radii = circles[:,2]


    # === MASKING ===
    # Get the indices of circles that overlap with the image boundary
    overlapping = (centers[:,0] - radii - boundary_px < 0) | \
                  (centers[:,0] + radii + boundary_px > img_edges.shape[0]) | \
                  (centers[:,1] - radii - boundary_px < 0) | \
                  (centers[:,1] + radii + boundary_px > img_edges.shape[1])

    # Remove the overlapping circles from the lists
    centers = centers[~overlapping]
    radii = radii[~overlapping]

    # Return the center coordinates and circle radii
    return centers, radii
```

In [ ]:
```
def plot_circles(img, centers, radii, inset = [0, 100], title = 'Detected circles'):
    '''Plot circles on an RGB image with inset zoom.

    PARAMETERS:
        img (np.array): The image
        centers (np.array): The center coordinates of the circles
        radii (np.array): The radii of the circles
        inset (list): The inset zoom parameters
    '''

    # Convert the image to RGB
    img_rgb = cv.cvtColor(img, cv.COLOR_GRAY2RGB)

    # Plot the circles on the original image
    fig, ax = plt.subplots()
    for center, radius in zip(centers, radii):
        cv.circle(img_rgb, (center[0].astype(int), center[1].astype(int)),
                  radius.astype(int), (0, 255, 0), 2)

    ax.imshow(img_rgb)

    ax.set_title(title)
    ax.set_xlabel('x [px]')
    ax.set_ylabel('y [px]')

    # Make an inset of the top left corner of the image
    if inset is not None:
        ax.add_patch(plt.Rectangle((inset[0], inset[0]), inset[1]-inset[0], inset[1]-inset[0],
                     fill=False, color='b', linewidth=2))
        axins = ax.inset_axes([0.12, 0.37, 0.5, 0.5], xlim=inset, ylim=np.flip(inset), xticks=[], yticks=[])
        axins.imshow(img_rgb, cmap='gray', origin="upper")
        axins.scatter(centers[:, 0], centers[:, 1], c="g", s=100, marker="+")
        axins.spines[:].set_color('blue')
        axins.spines[:].set_linewidth(2)

    # Show the image
    plt.imshow(img_rgb)
```

In [ ]:
```
# Detect circles in the image
cent_px, radii_px = detect_circles(img, min_dist=10, min_radius=5, max_radius=20)

# Plot the circles on the original image
plot_circles(img, cent_px, radii_px)
```



Note that the circles can only be drawn on the image with a full pixel accuracy on both the radius and the center. There is, however, some inaccuracy present in the Hough transform, since it is limited to 0.5 px when determining the center coordinates. A more accurate approach might be the FindCirclesGrid() algorithm in the OpenCV library, which we will later compare the result to.

First, we calculate the Euclidian distances between all points, $d = \sqrt{x^2 + y^2}$.

In [ ]:
```python
def calc_distance_between_all_points(x, y):
    '''Calculate the distance between all points in a 2D space

    PARAMETERS:
        x (np.array): The x-coordinates of the points
        y (np.array): The y-coordinates of the points

    RETURNS:
        distances_px (np.array): The distances between all points
    '''

    xy = np.column_stack([x, y])

    # Calculate the distance between all points
    distances_px = spatial.distance.cdist(xy, xy, 'euclidean')

    return distances_px

# Calculate the distance between all points
dist_px = calc_distance_between_all_points(cent_px[:, 0], cent_px[:, 1])
```

From these distances and the given calibration distance, we would like to calculate the resolution. However, for that, we need to know how many dots are actually spanned by the distances in the `dist_px` matrix. One problem: the output of the Hough transform is in an arbitrary order. Therefore, we first sort the coordinates by x and y values.

In [ ]:
```python
# Sort the coordinates by x and y
cent_px_sort = cent_px[np.lexsort((cent_px[:, 0], cent_px[:, 1]))]

# Calculate the distance between all points
dist_px_sort = calc_distance_between_all_points(cent_px_sort[:, 0], cent_px_sort[:, 1])

# Get the grid size from the uniqueness of the x and y coordinates
grid_size_id = [np.unique(cent_px_sort[:, 0]).size, np.unique(cent_px_sort[:, 1]).size]
```

This straightforward sorting operation only works in this case because, as it turns out, the coordinates could all be determined with 0.5 px precision, meaning that the x-coordinates of the dots in each column (and y-coordinates in each row) are exactly equal. Later, we will show a more robust method.

(ii) Resolution calculation

From the sorting indices of each circle, we can calculate the distance in "dot units". Combined with the calibration distance, we obtain the distances converted to real space, which, divided by the distance in pixel-space, give us a measure for the resolution of the image - calculated for every possible combination of circles.

In [ ]:
```python
def calc_resolution(dist_px, dist_id, calibration_distance = 1):
    '''Calculate the resolution of the image in pixels per unit length

    PARAMETERS:
        dist_px (np.array): The distance between all points in pixels
        dist_id (np.array): The distance between all points in index space
        calibration_distance (int): The distance between the calibration points in mm

    RETURNS:
        resolution (float): The resolution of the image in pixels per unit length
    '''

    # The real distance is given by the distance between
    #  two neighbouring points in real space times the distance in dot-space
    dist_re = calibration_distance * dist_id

    # Because the distance between a point and itself is 0, we ignore the division by zero warning
    with np.errstate(divide='ignore', invalid='ignore'):
        # Calculate the resolution of the image in pixels per unit length
        res = dist_re / dist_px

    return res
```

In [ ]:
```python
calibration_distance = 5 # mm

# Generate the grid in index space
cent_id_sort = np.fliplr(np.indices(grid_size_id).reshape(2, -1).T)

# Calculate the dot-space distances
dist_id_sort = calc_distance_between_all_points(cent_id_sort[:, 0], cent_id_sort[:, 1])

# Calculate the resolution for each distance
res = calc_resolution(dist_px_sort, dist_id_sort, calibration_distance)
```

Assuming that the grid is not distorted (for now), we can take the average resolution as a measure for the resolution of the entire image. Since the error scales with $1/d$, we can use a weighted average where we weigh the points by distance. So the resolution calculated from two neighbouring points weighs the least, and the diagonals the most.

In [ ]:
```python
def weighted_avg_and_std(values, weights, mask_diagonal=True,
                         verbose=False, precision=5):
    """
    Calculate the weighted average and standard deviation of a set of values.

    PARAMETERS:
        values (array-like): The values for which to calculate the weighted average and standard deviation.
        weights (array-like): The weights corresponding to each value.

    RETURNS:
        tuple: A tuple containing the weighted average and standard deviation.
    """

    # Mask the diagonal to avoid division by zero
    if mask_diagonal:
        mask = np.eye(values.shape[0], dtype=bool).__invert__()

    # Calculate the weighted average
    average = np.average(values[mask], weights=weights[mask])

    # Calculate the weighted variance
    variance = np.average((values[mask]-average)**2, weights=weights[mask])

    if verbose:
        print(f"The average resolution is {average:.{precision}f} mm/px,")
        print(f"with a standard deviation of {np.sqrt(variance):.{precision}f} mm/px ({100*np.sqrt(variance)/ave

    # Output the average and standard deviation
    return (average, np.sqrt(variance))
```

In [ ]:
```python
res_avg, res_std = weighted_avg_and_std(res, dist_id_sort, verbose=True)
```

```
The average resolution is 0.13054 mm/px,
with a standard deviation of 0.00020 mm/px (0.2%).
```

We think the standard deviation in the resolution gives a good estimate of the error in the calibration algorithm. It would be good if a measure of certainty could be extracted from the `cv.HoughCircles` function to compare the standard deviation to. Another option would be to extrapolate the position (in pixels) of the circles, based on the calculated resolution and the horizontal and vertical indices. The difference in position with the real position would also be a measure of the error in our algorithm.

(iii) Accuracy comparison

As a final check, we will compare the above error to the "off-the-shelf" `cv.findCirclesGrid` function, which compresses a few steps into one (with less control, however). Besides simplicity, the main advantage would be accuracy, because it is able to output the circle centres with sub-pixel precision. A disadvantage is that it requires *a priori* knowledge of the grid size, which is not unrealistic in a real experiment, but we did manage to circumvent that in the above steps.

In [ ]:
```python
def detect_circles_fancy(img, grid_size_id):
    '''Detect circles in an image using the openCV findCirclesGrid function

    PARAMETERS:
        img (np.array): The image
        grid_size_id (list): The grid size in index space

    RETURNS:
        cent_blackbox (np.array): The center coordinates of the detected circles
    '''

    # Find a symmetric circle grid using opencv
    grid_found, cent_blackbox = cv.findCirclesGrid(
        img, grid_size_id, flags=cv.CALIB_CB_SYMMETRIC_GRID)

    # Warn when no grid is found
    if grid_found is False:
        print('No grid found!')

        return None
    else:
        # Reshape the array
        cent_blackbox = cent_blackbox.reshape(-1, 2)

        return cent_blackbox

# Detect circles in the image
cent_blackbox = detect_circles_fancy(img, grid_size_id)

# Calculate the distance between all points
dist_px_blackbox = calc_distance_between_all_points(cent_blackbox[:, 0], cent_blackbox[:, 1])

# Calculate the resolution for each distance
res_blackbox = calc_resolution(dist_px_blackbox, dist_id_sort, calibration_distance)

# Calculate the weighted average and standard deviation
res_avg_blackbox, res_std_blackbox = weighted_avg_and_std(res_blackbox, dist_id_sort, verbose=True)
```

```
The average resolution is 0.13055 mm/px,
with a standard deviation of 0.00001 mm/px (0.0%).
```

This result is indeed more precise, but also well within the error margin of our "manual" calculation.

a2) **Noise, rotation & lens vignetting** (8 pts)

In this part we will elaborate on the steps taken in part a1. Start, again, by importing the image, and setting a new calibration size.

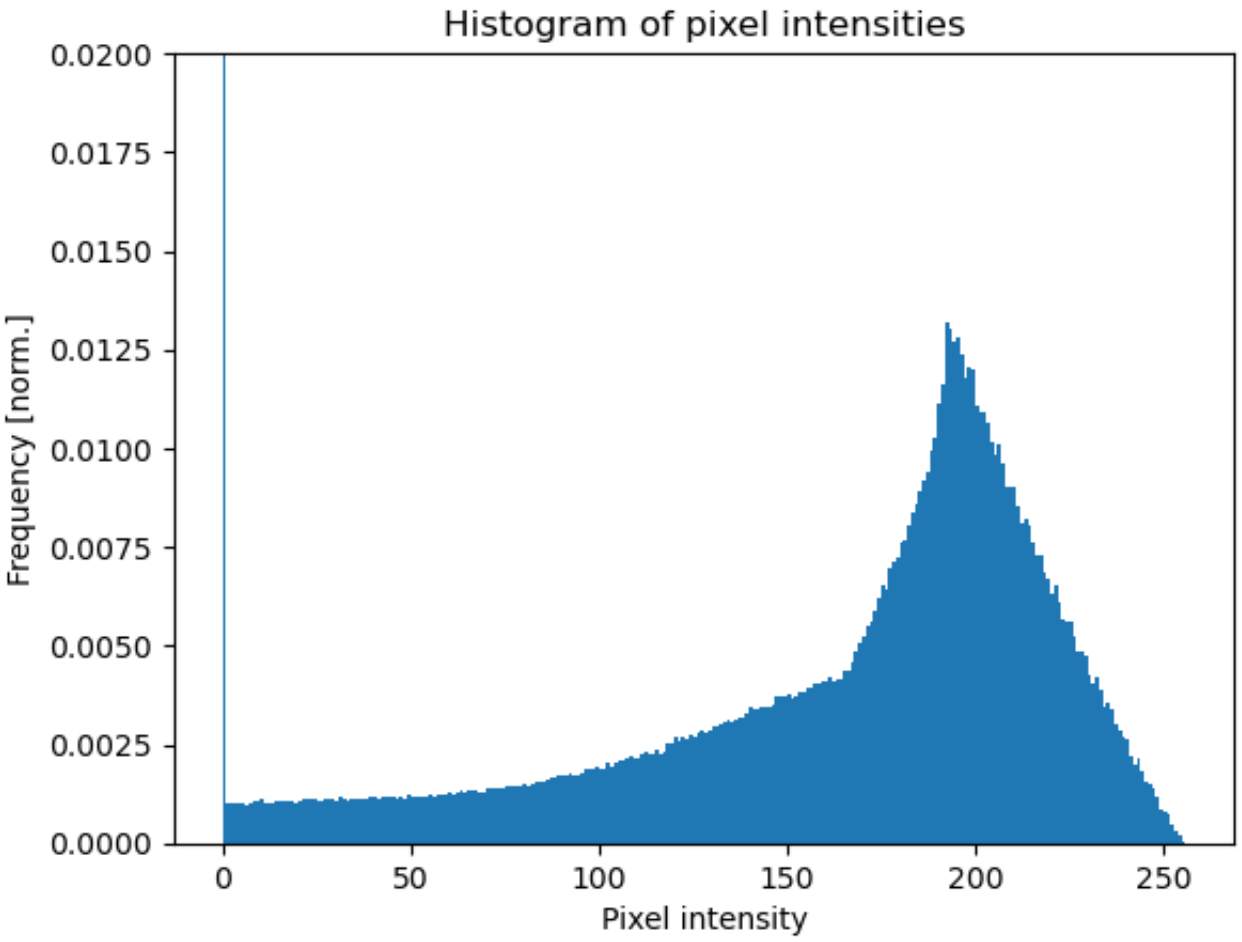
In [ ]:
```python
# Clear all variables without removing the imported modules
del [[cent_px, radii_px, cent_px_sort, dist_px, dist_px_sort,
      grid_size_id, cent_id_sort, dist_id_sort, res,
      calibration_distance, cent_blackbox,
      dist_px_blackbox, res_blackbox, res_avg,
      res_std, res_avg_blackbox, res_std_blackbox]]

# Import the second image in grayscale
img_path = "Images/Calibration_a/Realistic.png"
img = cv.imread(img_path, 0)

# Set the calibration distance to 2 mm
calibration_distance = 2 # mm
```

(i) Noise & vignetting correction

Let us see what is going on in the image by making a histogram of the pixel intensities.

In [ ]:
```python
# Make a histogram of the image
fig, ax = plt.subplots()
ax.hist(img.ravel(), bins=256, range=[0,256], density=True)
ax.set_title('Histogram of pixel intensities')
ax.set_xlabel('Pixel intensity')
ax.set_ylabel('Frequency [norm.]')
ax.set_ylim([0, 0.02])
plt.show()
```

```
# Print the fraction of pixels at zero intensity
print(f'The fraction of pixels at zero intensity is {np.sum(img == 0) / img.size:.4f}.')
```




```
The fraction of pixels at zero intensity is 0.0904.
```

It is clear that the circles will be mostly contained in the large peak at zero intensity (which, in the plot above is cropped; it actually goes up to ~0.09.). The background noise is mostly spread out in intensity. Therefore, we can safely apply some contrast or brightness changes.

In the end, we settled on the following order of operations to enhance the image:

1. Start by applying a Gaussian blur to get rid of any sharp features in the background. The radius of the kernel was chosen empirically.
2. Increase the contrast of the image slightly using the OpenCV library to add and clip images.
3. Binarise the (greyscale) image with an adaptive threshold. This way, we can separate the dots from the background. To be extra safe, an adaptive threshold was chosen to account for the vignetting in the image, which causes the background to be darker at the edges. From the histogram, we could see that this might not have been necessary in this case, but it does not hurt either, as long as the kernel used to calculate the average brightness is large enough. We have chosen the constant which `cv.adaptiveThreshold` subtracts from this value (to determine the local threshold) empirically.
4. Next, we close the image with a circular kernel to remove some small artefacts from the background. This is done by first dilating and then eroding the image.
5. Then, we open the image up to fill small holes that are left in the black circles. The same (small) kernel is used as in step 4.

Note that, in the end, step 4 and 5 do not add much, since we managed to tweak the other parameters well enough that the circles were already well-separated after step 3. We have kept them here for future reference.

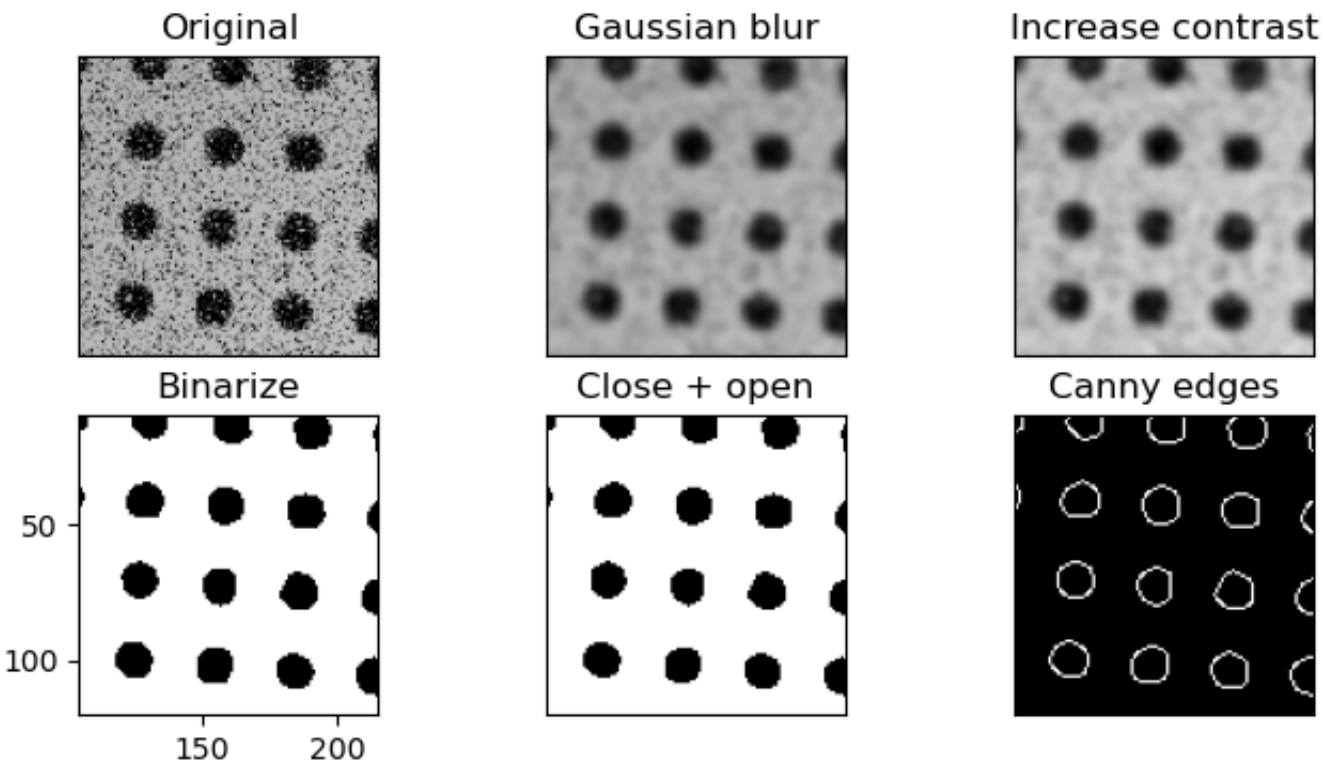
6. Optionally, we perform Canny edge detection on the image. This should not be necessary, since this step is also already included in the `cv.HoughCircles` function. However, in practice, it seemed to yield better results.

In [ ]:
```
def pre_process(img, blur_kernel = 9, contrast = 1.2, brightness = 0,
                binary_kernel_radius = 31, binary_constant = 51, morph_kernel_radius = 5,
                plot = False, canny_edges = False, canny_thresh = 70, canny_max = 255):
    '''Pre-process an image for circle detection

    PARAMETERS:
        img (np.array): The image to pre-process
        blur_kernel (int): The size of the Gaussian blur kernel
        contrast (float): The contrast enhancement factor
        brightness (int): The brightness enhancement factor
        binary_kernel_radius (int): The radius of the adaptive thresholding kernel
        binary_constant (int): The constant of the adaptive thresholding
        morph_kernel_radius (int): The radius of the morphological kernel
        plot (bool): Whether to plot the pre-processing steps
        canny_edges (bool): Whether to use Canny edge detection
        canny_thresh (int): The upper threshold for the Canny edge detector
        canny_max (int): The maximum value of the Canny edge detector

    RETURNS:
        img_proc (np.array): The pre-processed image
    '''
```

```
    # Make a copy of the image
    img_proc = img.copy()

    if plot:
        # Set up a plot, show original image (1)
        fig, ax = plt.subplots(2, 3, figsize=(8, 4))
        ax[0, 0].imshow(img_proc, cmap='gray')

    # Apply a Gaussian blur (2)
    img_proc = cv.GaussianBlur(img_proc, (blur_kernel, blur_kernel), 0)
    if plot: ax[0, 1].imshow(img_proc, cmap='gray')

    # Increase the contrast (3)
    img_proc = cv.addWeighted(
        img_proc, contrast, img_proc, brightness, 0)
    if plot: ax[0, 2].imshow(img_proc, cmap='gray')

    # Binarize the image with adaptive threshold (4)
    img_proc = cv.adaptiveThreshold(
        img_proc, 255, cv.ADAPTIVE_THRESH_MEAN_C, cv.THRESH_BINARY, binary_kernel_radius, binary_constant)
    if plot: ax[1, 0].imshow(img_proc, cmap='gray')

    # Close the image with a circular kernel to remove small components (5)
    kernel = cv.getStructuringElement(
        cv.MORPH_ELLIPSE, (morph_kernel_radius, morph_kernel_radius))
    img_proc = cv.dilate(img_proc, kernel)
    img_proc = cv.erode(img_proc, kernel)

    # Open the image with the same kernel to fill small holes
    img_proc = cv.erode(img_proc, kernel)
    img_proc = cv.dilate(img_proc, kernel)
    if plot: ax[1, 1].imshow(img_proc, cmap='gray')

    # Edge detection (6)
    if canny_edges:
        img_proc = cv.Canny(img_proc, canny_thresh, canny_max)
    else:
        img_proc = np.zeros_like(img_proc)
    if plot: ax[1, 2].imshow(img_proc, cmap='gray')

    if plot:
        # Finish the plot
        subplot_names = ['Original', 'Gaussian blur',
                         'Increase contrast', 'Binarize',
                         'Close + open', '']
        if canny_edges:
            subplot_names[5] = 'Canny edges'

        for ii, a in enumerate(ax.flatten()):
            a.set_xticks([])
            a.set_yticks([])
            a.set_xlim([105, 215])
            a.set_ylim([120, 10])
            a.set_title(subplot_names[ii])

        ax[1,0].set_xticks([150, 200])
        ax[1,0].set_yticks([50, 100])
        plt.show()

    return img_proc
```

The intermediate results are shown on a cutout of the top-left corner of the image in the plots below. The vertical axis seems to have flipped accidentally.

In [ ]:
```
# Pre-process the image
img_proc = pre_process(img, plot=True, canny_edges=True)
```



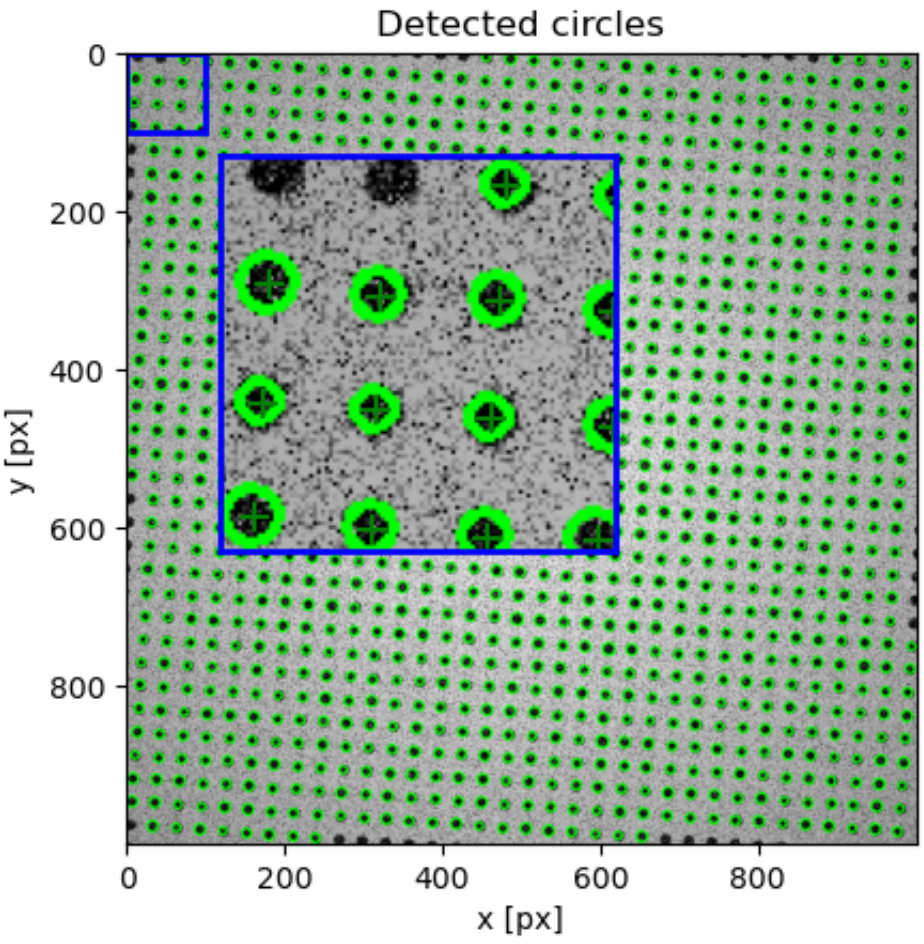
(ii) Circle detection

This time, we search for smaller circles. Therefore, we decrease the minimum radius, as well as the parameter dP, which accounts for how the accumulator space is defined relative to the image space. In the accumulator space, possible circle candidates are determined, so a higher resolution can help when detecting small circles (see https://docs.opencv.org/4.x/dd/d1a/group__imgproc__feature.html#ga47849c3be0d0406ad3ca45db65a25d2d).

We have also changed to the regular `cv.HOUGH_GRADIENT` algorithm, since it seems to be more robust to the jagged edges in our image, despite the pre-processing.

In [ ]:
```
# Detect circles in the processed image
cent_px, radii_px = detect_circles(img_proc, algorithm = cv.HOUGH_GRADIENT,
                                   dP = 1, min_dist = 10, min_radius = 5, max_radius = 14,
                                   canny_thresh = 140, parameter2 = 12,
                                   verbose=True)

# Plot the circles on the original image
plot_circles(img, cent_px, radii_px)
```

```
1145 circles detected
```



(iii) Rotation correction

Our code from part a1 requires a rectangular grid of circles. Due to the rotation in the image, the number of circles in each row and column varies now. Thus, we would like to remove the "incomplete" rows and columns. Let us try to find the angle.

We start by generating an image containing only the circle centres as white pixels on a black background. Because these coordinates are (often) in between pixels, we colour the pixels around the centres too, when necessary. Then we apply the Hough line transform with a 1 pixel and 0.1 degree resolution. A minimum of 40 votes is chosen such that only horizontal, vertical and diagonal lines show up (they hit at least 40 white pixels, which would amount to at least 10 circles where the centre is in between 4 pixels).

From these lines, the ones with the most common angle (after clustering) are used to calculate an average value and error. Because the circles centres can be a group of 2 or 4 pixels in the generated image, more horizontal and vertical lines than diagonals can be drawn through them. So for angles smaller than 45°, we are likely to avoid accidentally extracting the angle of the diagonals.

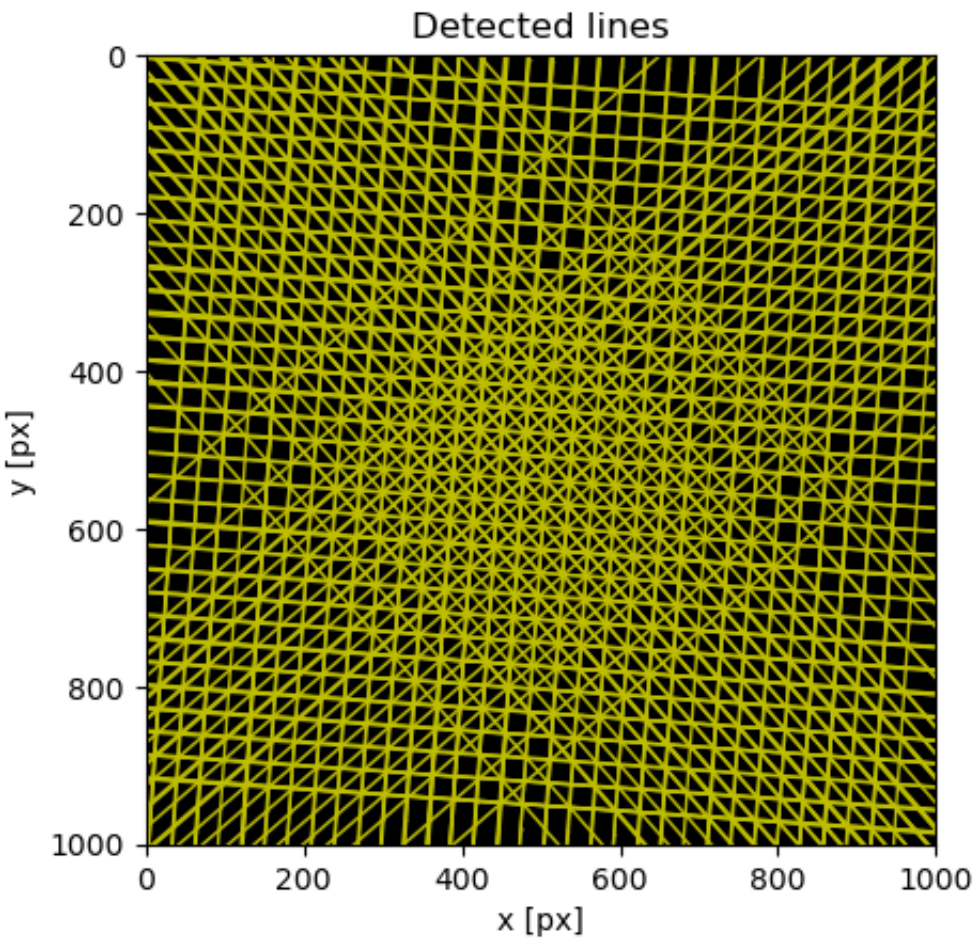
In [ ]:
```python
def calc_angle(img, centers, res_px = 1, res_deg = 0.1, min_votes = 40,
               plot = False, verbose = False):
    '''Calculate the angle of the circles in the image

    PARAMETERS:
        img (np.array): The image
        centers (np.array): The center coordinates of the circles
        res_px (int): The resolution of the Hough line transform in pixels
        res_deg (int): The resolution of the Hough line transform in degrees
        min_votes (int): The minimum number of votes for a line to be considered
        plot (bool): Whether to plot the detected lines
        verbose (bool): Whether to print the results

    RETURNS:
        angle_avg (float): The average angle of the grid in radians
        angle_std (float): The standard deviation of the angle in radians
        angle_ct (int): The number of lines used to calculate the angle
        lines_ct (int): The total number of lines detected
    '''

    # === CIRCLE CENTER GRID ===
    # Generate a black image
    img_centers = np.zeros_like(img)

    # Get the pixel indices surrounding the circle centres,
    # and clip them to the image size
    x = [np.clip(np.floor(centers[:,0]).astype(int), 0, img.shape[1]-1),
         np.clip(np.ceil(centers[:,0]).astype(int), 0, img.shape[1]-1)]
    y = [np.clip(np.floor(centers[:,1]).astype(int), 0, img.shape[0]-1),
         np.clip(np.ceil(centers[:,1]).astype(int), 0, img.shape[0]-1)]

    # Color these pixels white
    img_centers[y[0], x[0]] = img_centers[y[1], x[1]] \
        = img_centers[y[0], x[1]] = img_centers[y[1], x[0]] = 255

    # === HOUGH LINE TRANSFORM ===
    # Perform a Hough line transform
    lines = cv.HoughLines(img_centers, res_px, res_deg * np.pi / 180, min_votes)

    # Make a plot of the lines
    if plot:
        fig, ax = plt.subplots()
        ax.imshow(img_centers, cmap='gray')

        # Plot the lines
        for line in lines:
            rho, theta = line[0]
            a = np.cos(theta); b = np.sin(theta)
            x0 = a * rho; y0 = b * rho
            x1 = int(x0 + 2 * img.shape[0] * (-b))
            y1 = int(y0 + 2 * img.shape[1] * (a))
            x2 = int(x0 - 2 * img.shape[0] * (-b))
            y2 = int(y0 - 2 * img.shape[1] * (a))
            ax.plot([x1, x2], [y1, y2], 'y', linewidth=1)

        ax.set_title('Detected lines')
        ax.set_xlabel('x [px]')
        ax.set_ylabel('y [px]')
        ax.set_xlim([0, img.shape[0]])
        ax.set_ylim([img.shape[1], 0])
        plt.show()

    # === PROCESSING ===
    # Extract angles in degrees
    lines_angles = np.array([line[0][1] / np.pi * 180 for line in lines])

    # Round to the nearest 1 degree and consider multiples of 90 degrees
    lines_angles = np.round(np.mod(lines_angles, 90), 1)

    # Use the DBSCAN algorithm to cluster the angles
    dbscan = DBSCAN(eps=1, min_samples=2)
    dbscan.fit(lines_angles.reshape(-1, 1))

    # Get the number of clusters
    angles_ids, angles_cts = np.unique(dbscan.labels_, return_counts=True)

    # If only two angles are left
```

```
    if len(angles_cts) == 2:
        # Calculate the average of the angles in the largest bin
        # (assuming non-diagonal lines get more hits with non-singular points)
        angles = [lines_angles[dbscan.labels_ == angles_ids[angles_cts.argmax()]]]
        angle_avg = np.mean(angles)
        angle_std = np.std(angles)

        # Also get the number of elements in the smallest cluster
        angle_ct = angles_cts[angles_cts.argmax()]

        # If the angle is larger than 45 degrees, subtract 90 degrees
        if angle_avg > 45:
            angle_avg -= 90

        if verbose:
            print(f"The angle of the grid in the image is {angle_avg:.1f}°"\
                  f" with a standard deviation of {angle_std:.1f}°.")
            print(f"This was calculated using {angle_ct}/{len(lines_angles)} lines found.")

        # Convert back to radians
        angle_avg = angle_avg / 180 * np.pi
        angle_std = angle_std / 180 * np.pi

        return angle_avg, angle_std, angle_ct, len(lines_angles)

    # If no or more than two angles are found, return None
    else:
        if verbose:
            print('Too little or too many lines found, try changing the min_votes.')
        return None, None, None, None
```

Note that for the clustering, we added a small machine learning part. This solution might be a bit overkill, but it is quick and reliable: we simply cluster the "data points" (in this case line angles) into an unknown number of bins. The algorithm we used is called "Density-Based Spatial Clustering of Applications with Noise" or DBSCAN, which groups neighbouring points together without prior knowledge on the number of clusters (see also https://en.wikipedia.org/wiki/DBSCAN).

We first found it when sorting the grid points (see part iv), where it made slightly more sense to use this function - here the clustering can definitely be done more easily. But we had it imported anyway.

In [ ]:
```
# Calculate the angle of the circles in the image
angle_avg, angle_std, angle_ct, line_ct = calc_angle(img_proc, cent_px, plot=True, verbose=True)
```




```
The angle of the grid in the image is 4.0° with a standard deviation of 0.1°.
This was calculated using 183/259 lines found.
```

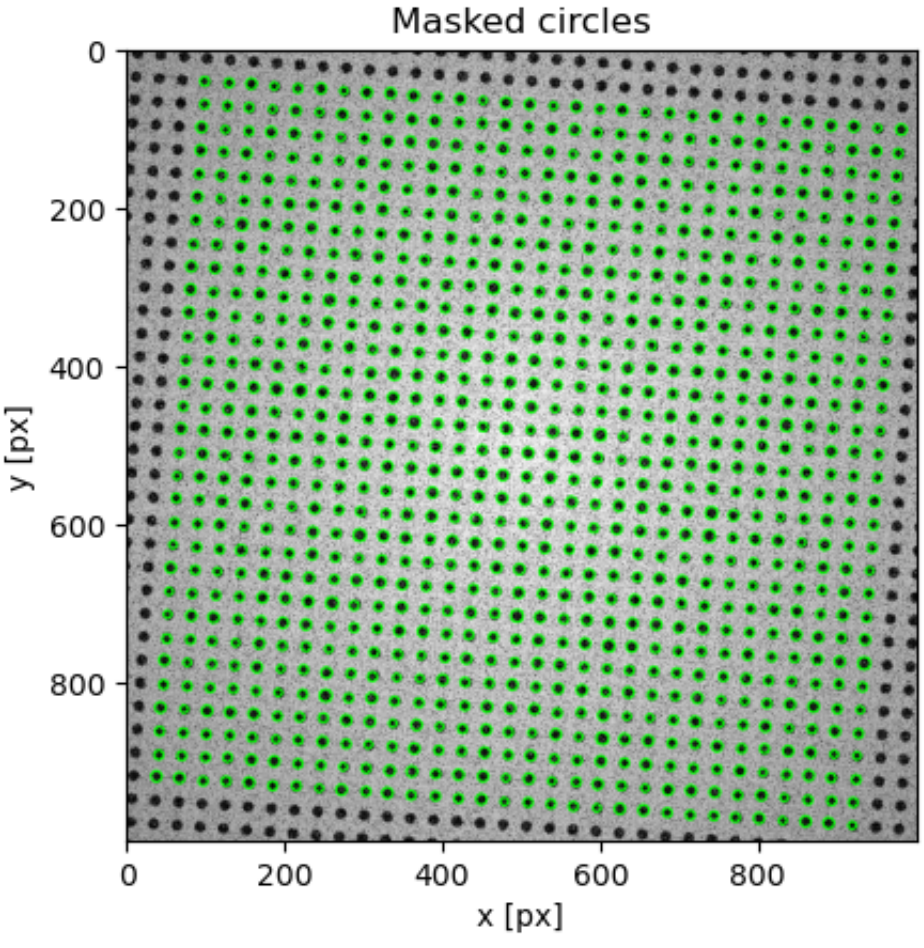
Even though we could exploit the embedded possibility of drawing multiple lines here nicely, it does result in a larger-than-necessary error (given that you could, e.g., draw a line from the left pixel of the top circle centre to the right pixel of the bottom centre). There are probably more accurate ways to distinguish diagonals (like matching crossings of orthogonal line sets to the circle centres themselves), but this works for now.

Next, we can use this angle to define a rectangular mask, rotated by the angle that was found, and cut off the incomplete rows. Note that the way we incorporate the circle radii is less than ideal, since they are also taken into account when there are no circles overlapping the image edges. Furthermore, the formulas do not work for negative angles. These bugs should be fixed for future use,

but for now the function suffices.

In [ ]:
```python
def mask_grid_rotation(img, cent_px, radii_px, angle,
                       boundary_px=0, plot=False):
    """
    Masks circles in an image that are outside the rotated image.

    PARAMETERS:
        img (numpy.ndarray): The input image.
        cent_px (numpy.ndarray): The coordinates of the circle centers in pixels.
        radii_px (numpy.ndarray): The radii of the circles in pixels.
        angle (float): The angle of rotation in radians.
        boundary_px (int): The boundary in pixels to consider around the rotated image. Default is 0.
        plot (bool): Whether to plot the masked circles. Default is False.

    RETURNS:
        cent_px_mask (numpy.ndarray): The masked circle centers.
        radii_px_mask (numpy.ndarray): The masked circle radii.
    """

    # Get the image size
    img_y, img_x = img.shape

    # Get the circle coordinates
    circles_x = cent_px[:, 0]
    circles_y = cent_px[:, 1]

    # Use the mean radius to calculate the boundary
    radius_px = np.mean(radii_px)

        # Mask the circles that are outside the rotated image
    mask = (circles_y > radius_px + boundary_px + np.tan(angle) * circles_x) \
               & (circles_y < img_y - radius_px - boundary_px - np.tan(angle) * (img_x - circles_x)) \
               & (circles_x > radius_px + boundary_px + np.tan(angle) * (img_y - circles_y)) \
               & (circles_x < img_x - radius_px - boundary_px - np.tan(angle) * circles_y)

        # Quick 'n' dirty fix for negative angle in part a3
    if angle < 0:
        mask = (circles_y > radius_px + boundary_px - np.tan(angle) * (img_x - circles_x)) \
            & (circles_y < img_y - radius_px - boundary_px + np.tan(angle) * circles_x) \
            & (circles_x > radius_px + boundary_px - np.tan(angle) * circles_y) \
            & (circles_x < img_x - radius_px - boundary_px + np.tan(angle) * (img_y - circles_y))

    # Get the masked circles
    cent_px_mask = cent_px[mask]
    radii_px_mask = radii_px[mask]

    # Make a plot of the masked circles
    if plot:
        plot_circles(img, cent_px_mask, radii_px_mask,
                     title='Masked circles', inset=None)

    return cent_px_mask, radii_px_mask
```

In [ ]:
```python
cent_px_mask, radii_px_mask = mask_grid_rotation(img, cent_px, radii_px, angle_avg, plot=True)
```

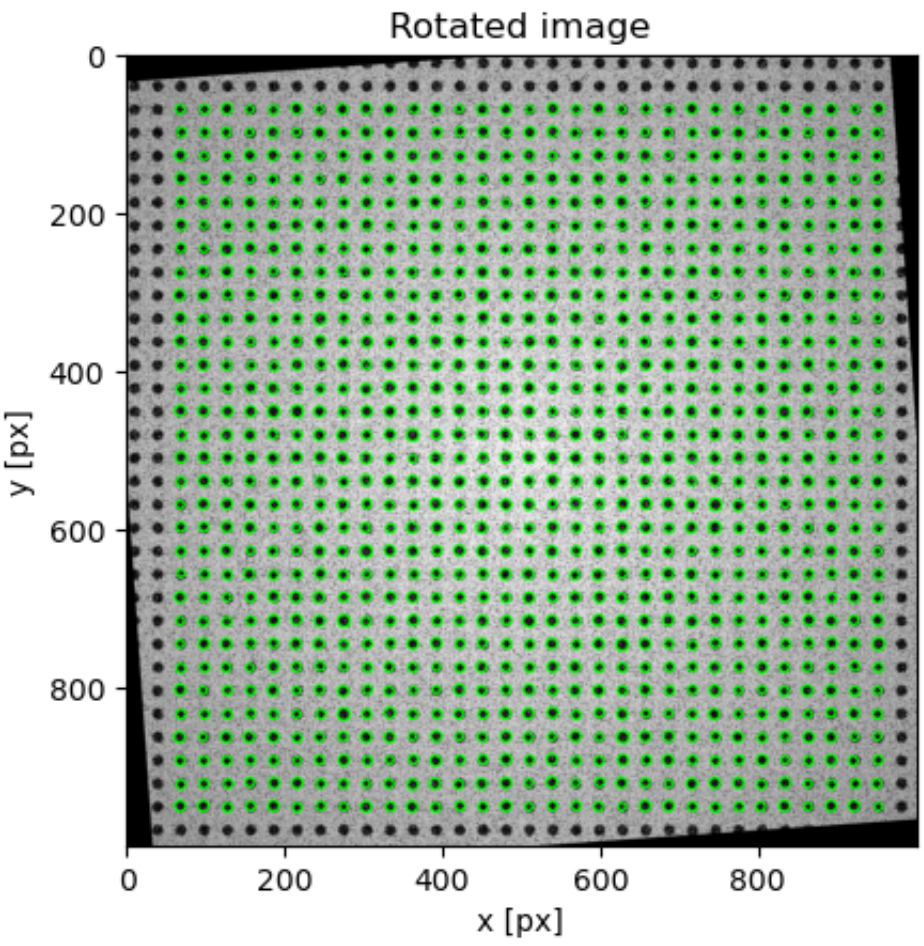


Now, we can rotate the grid back!

In [ ]:
```python
def rotate_grid(cent, angle, plot=False, img=None):
    """
    Rotate a grid of points around a center point by a given angle.

    PARAMETERS:
        cent (numpy.ndarray): The center point of the grid.
        angle (float): The angle (in radians) by which to rotate the grid.
        plot (bool): Whether to plot the rotated grid. Default is False.
        img (numpy.ndarray): The image to rotate around. Default is None.

    RETURNS:
        cent_rot (numpy.ndarray): The rotated grid of points.
        img_rot (numpy.ndarray): The rotated image.
    """

    # === CIRCLE CENTER ROTATION ===
    # If the image is not given, rotate around (0,0))
    if img is None:
        # Assume the center is at (0, 0)
        img_center = [0, 0]
    else:
        # Get the image size
        img_y, img_x = img.shape

        # Calculate the center of the image
        img_center = [img_x / 2, img_y / 2]

    # Generate a rotation matrix
    rot_matrix = np.array([[np.cos(angle), -np.sin(angle)],
                           [np.sin(angle), np.cos(angle)]])

    # Translate, multiply by the rotation matrix and translate back
    cent_tra = cent - img_center
    cent_rot = np.dot(rot_matrix, cent_tra.T)
    cent_rot = cent_rot.T + img_center

    # === IMAGE ROTATION ===
    if img is not None:
        # Rotate the image the openCV way
        rot_matrix = cv.getRotationMatrix2D(img_center, -angle / np.pi * 180, 1)
        img_rot = cv.warpAffine(img, rot_matrix, (img_x, img_y))

        # Make a plot of the rotated image
        if plot and cent is not None:
            # Plot the circles on the rotated image
            plot_circles(img_rot, cent_rot, radii_px_mask, title='Rotated image', inset=None)
    else:
        img_rot = None

    return cent_rot, img_rot
```

In [ ]:
```python
# Calculate the rotated grid of circle coordinates in pixels
```

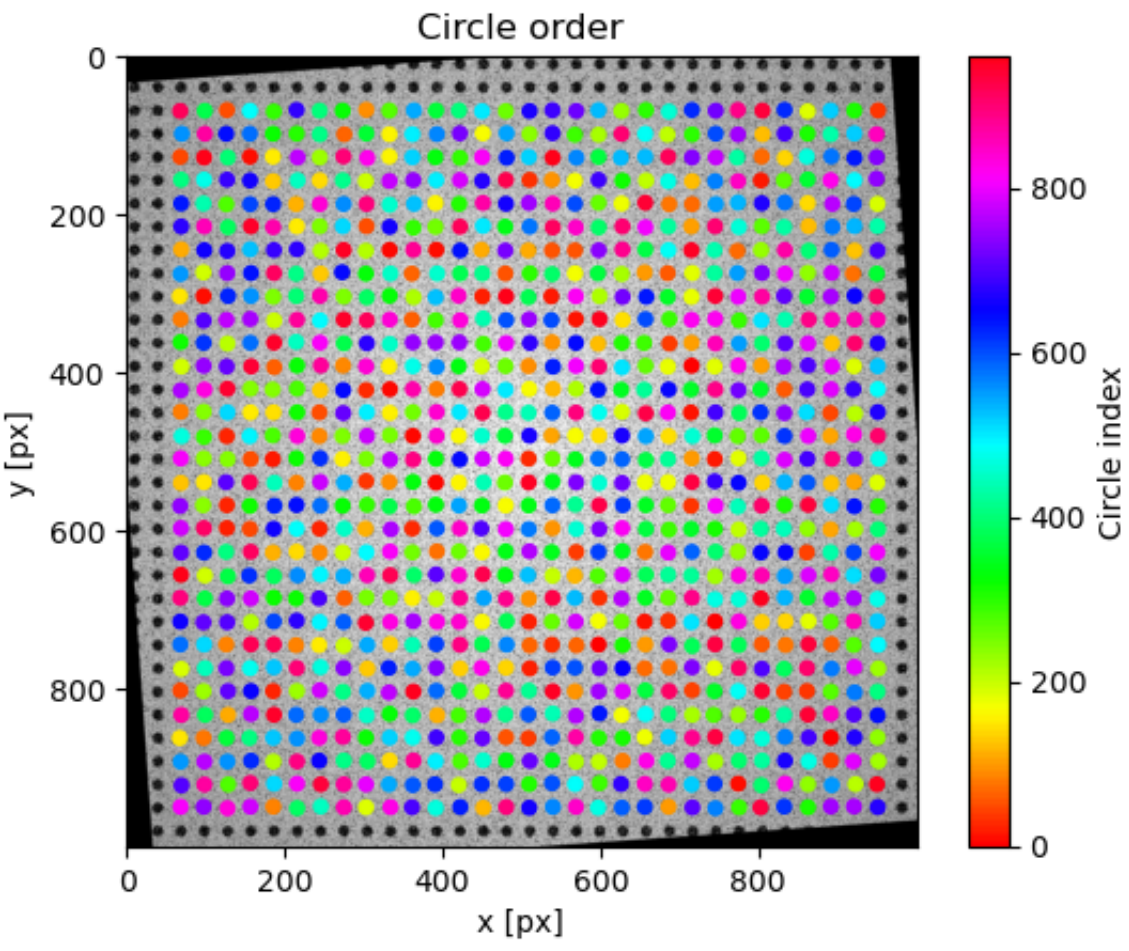
```
# and plot the image as verification
cent_px_rot, img_rot = rotate_grid(cent_px_mask, -angle_avg, plot=True, img=img)
```



(iv) Circle sorting

It is finally time to work towards a resolution calculation. However, due to the rotation, there are now slight inaccuracies in both coordinates, making this not so trivial.

In [ ]:
```
# Plot the circle centers with a color corresponding to the index
fig, ax = plt.subplots()
ax.imshow(img_rot, cmap='gray')
ax.scatter(cent_px_rot[:, 0], cent_px_rot[:, 1],
           c=range(len(cent_px_rot)), s=20,
           cmap = 'hsv')
ax.set_title('Circle order')
ax.set_xlabel('x [px]')
ax.set_ylabel('y [px]')

# Add a colorbar
cbar = plt.colorbar(ax.collections[0], ax=ax, orientation='vertical')
cbar.set_label('Circle index')
plt.show()
```



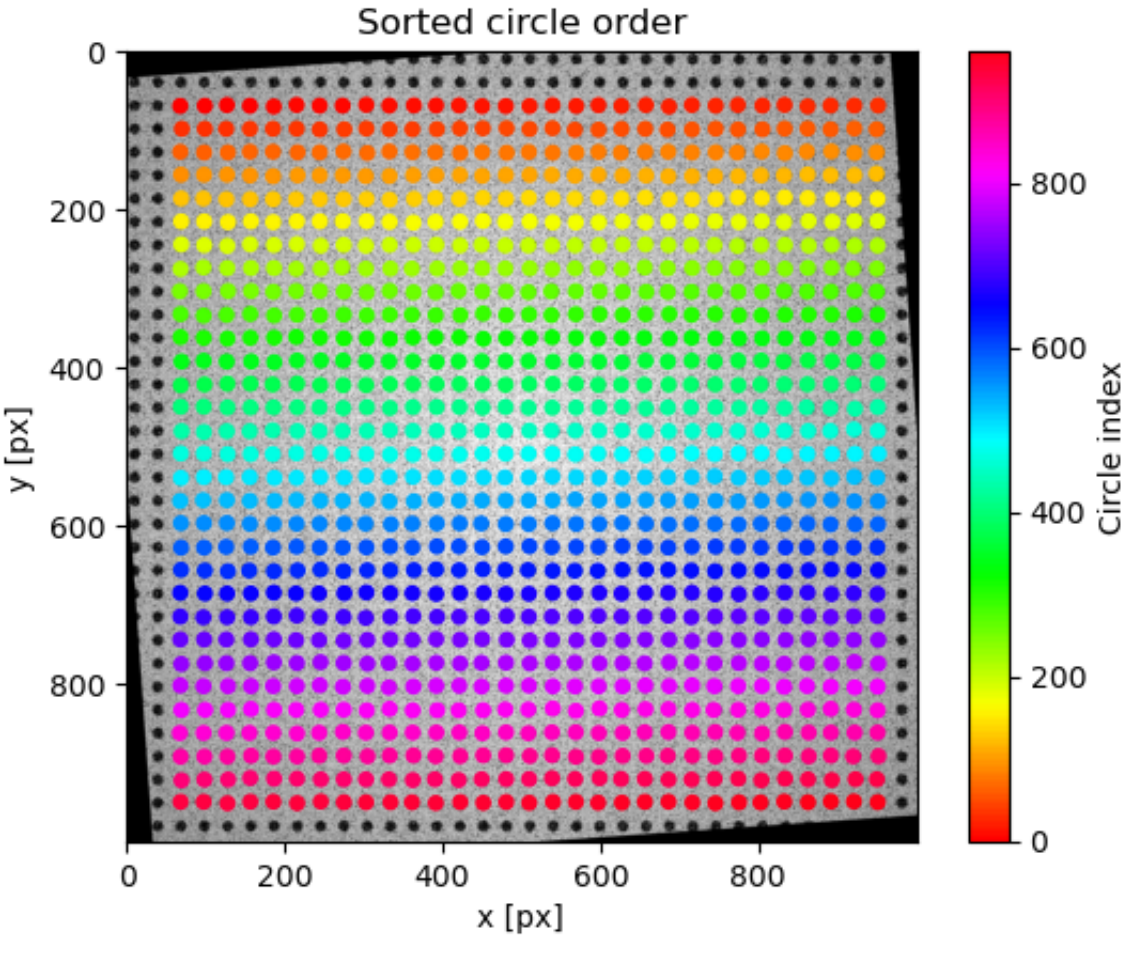
Like in part iii, we pull out the DBSCAN function to cluster the points twice: first in the x-direction, and then in the y-direction.

In [ ]:
```
def sort_circles(cent_px, radii_px, discard=False):
    # Sort the circle coordinates by x
```

```
    cent_px_sort = cent_px.copy()
    cent_px_sort = cent_px_sort[cent_px_sort[:, 0].argsort()]

    # Minimum distance between clusters is the average circle diameter
    min_dist = 2 * np.mean(radii_px)

    # Perform clustering in the x direction, fit model to data
    dbscan_x = DBSCAN(eps=min_dist, min_samples=2)
    dbscan_x.fit(cent_px_sort[:, 0].reshape(-1, 1))

    # Make an array of the cluster numbers in the x direction for each circle
    cent_id = dbscan_x.labels_

    # Sort the data points and the clusters array by y coordinate
    sort_y = cent_px_sort[:, 1].argsort()
    cent_px_sort = cent_px_sort[sort_y]
    cent_id = cent_id[sort_y]

    # Perform clustering in the y direction, fit model to data
    dbscan_y = DBSCAN(eps=min_dist, min_samples=2)
    dbscan_y.fit(cent_px_sort[:, 1].reshape(-1, 1))

    # Add a dimension to the clusters array
    cent_id = np.vstack((cent_id, dbscan_y.labels_)).T

    indices = np.lexsort((cent_id[:, 0], cent_id[:, 1]))

    # Discard the unclustered points
    if discard:
        # Remove points with index -1 from the indices array
        indices = indices[cent_id[indices].min(axis=1) != -1]

    # Apply the indices
    cent_id = cent_id[indices]

    # Sort the circle coordinates and radii by the cluster indices
    cent_px_sort = cent_px_sort[indices]
    radii_px_sort = radii_px[indices]

    # Count the number of clusters
    grid_size_id = [len(np.unique(dbscan_x.labels_)), len(np.unique(dbscan_y.labels_))]

    return cent_px_sort, radii_px_sort, cent_id, grid_size_id
```

In [ ]:

```
# Sort the circle coordinates and radii
cent_px_sort, radii_px_sort, cent_id, grid_size_id = sort_circles(cent_px_rot, radii_px_mask)

# Plot the circle centers with a color corresponding to the index
fig, ax = plt.subplots()
ax.imshow(img_rot, cmap='gray')
ax.scatter(cent_px_sort[:, 0], cent_px_sort[:, 1],
           c=range(len(cent_px_sort)), s=20,
           cmap = 'hsv')
ax.set_title('Sorted circle order')
ax.set_xlabel('x [px]')
ax.set_ylabel('y [px]')

# Add a colorbar
cbar = plt.colorbar(ax.collections[0], ax=ax, orientation='vertical')
cbar.set_label('Circle index')
plt.show()
```



(v) Resolution calculation

Now, we rotate the coordinates back, and perform the same steps as in part a1 to obtain the resolution of the image.

In [ ]:
```
# Rotate the grid back to the original orientation
cent_px_sort, _ = rotate_grid(cent_px_sort, angle_avg)

# Calculate the distance between all points in real space and index space
dist_px_sort = calc_distance_between_all_points(cent_px_sort[:, 0], cent_px_sort[:, 1])
dist_id_sort = calc_distance_between_all_points(cent_id[:, 0], cent_id[:, 1])

# Calculate the resolution for each distance
res = calc_resolution(dist_px_sort, dist_id_sort, calibration_distance)

# Calculate the weighted average and standard deviation
res_avg, res_std = weighted_avg_and_std(res, dist_id_sort, verbose=True)
```

```
The average resolution is 0.06803 mm/px,
with a standard deviation of 0.00016 mm/px (0.2%).
```

a3) **Radially-varying resolution** (3 bonus pts.)

Now, the image is also distorted! We start by repeating steps (i) - (iv) from part a2.

In [ ]:
```
# Clear all variables without removing the imported modules
del [[angle_avg, angle_ct, angle_std, cent_id,
      cent_px, cent_px_mask, cent_px_rot, cent_px_sort,
      dist_id_sort, dist_px_sort, grid_size_id,
      img_proc, img_rot, line_ct, radii_px,
      radii_px_mask, radii_px_sort, res, res_avg,
      res_std]]

# Import the third image in grayscale
img_path = "Images/Calibration_a/Distorted.png"
img = cv.imread(img_path, 0)

# Set the calibration distance to 4 mm
calibration_distance = 4 # mm
```

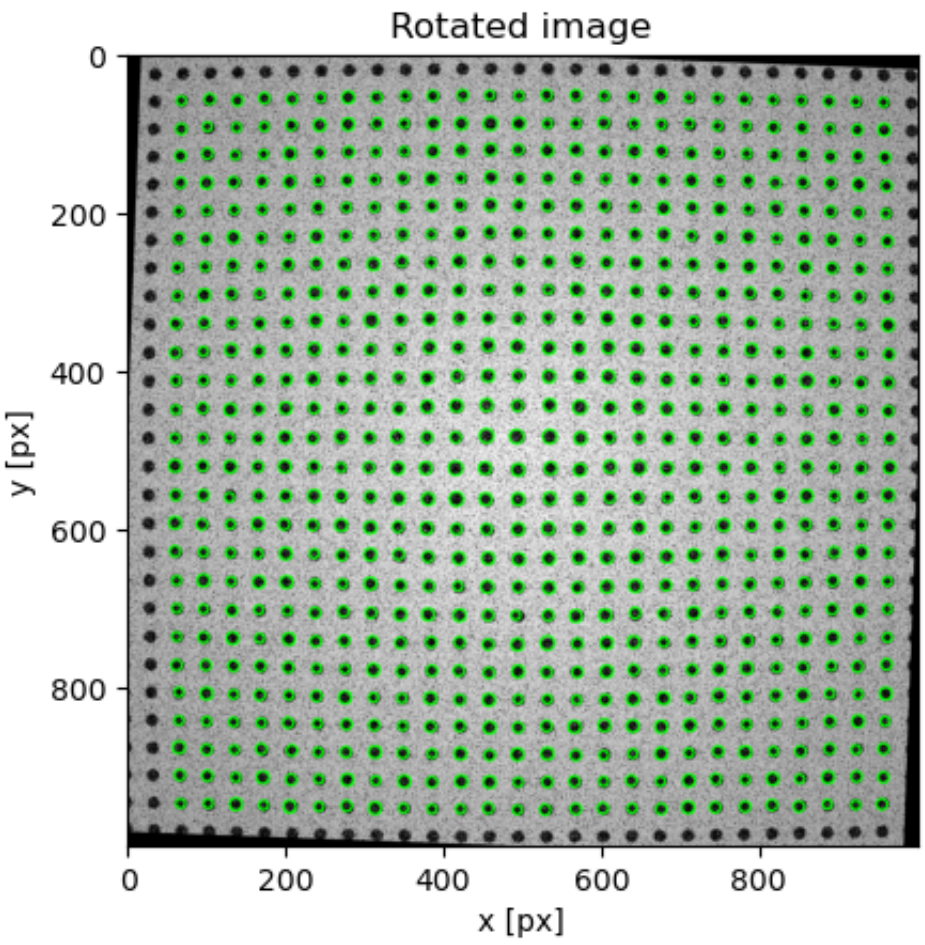
Some small notes:

- The circle detection parameters were adjusted slightly to account for the new
- In calculating the angle, we have opted to decrease the number of required votes, so that only the lines in the centre are found. Since the distortion looks to be larger at the edges, we can decrease the error this way.
- The grid masking function did not work for negative angles and turned out to be less robust when the circle radii are spread, so we applied a quick fix in the definition above.
- The rest of the functions worked without issues.

In [ ]:
```
# Pre-process the image
img_proc = pre_process(img, plot=False, canny_edges=True)
```

```
# Detect circles in the processed image
cent_px, radii_px = detect_circles(img_proc, algorithm = cv.HOUGH_GRADIENT,
                                   dP = 1, min_dist = 10, min_radius = 5, max_radius = 14,
                                   canny_thresh = 140, parameter2 = 12, verbose=True)

# Calculate the angle of the circles in the image
angle_avg, angle_std, angle_ct, line_ct = calc_angle(img_proc, cent_px, min_votes=30, plot=False, verbose=True)

# Mask the circles that are outside the rotated image
cent_px_mask, radii_px_mask = mask_grid_rotation(img_proc, cent_px, radii_px, angle_avg, boundary_px=8.5)

# Calculate the rotated grid of circle coordinates in pixels
cent_px_rot, img_rot = rotate_grid(cent_px_mask, -angle_avg, img=img, plot=True)

# Sort the circle coordinates and radii
cent_px_sort, radii_px_sort, cent_id, grid_size_id = sort_circles(cent_px_rot, radii_px_mask)

# Rotate the grid back to the original orientation
cent_px_sort, _ = rotate_grid(cent_px_sort, angle_avg)

# Calculate the distance between all points in real space and index space
dist_px_sort = calc_distance_between_all_points(cent_px_sort[:, 0], cent_px_sort[:, 1])
dist_id_sort = calc_distance_between_all_points(cent_id[:, 0], cent_id[:, 1])

# Calculate the resolution for each distance
res = calc_resolution(dist_px_sort, dist_id_sort, calibration_distance)

# Calculate the weighted average and standard deviation
res_avg, res_std = weighted_avg_and_std(res, dist_id_sort, verbose=True)
```

```
780 circles detected
The angle of the grid in the image is -2.1° with a standard deviation of 0.2°.
This was calculated using 10/16 lines found.
The average resolution is 0.11044 mm/px,
with a standard deviation of 0.00149 mm/px (1.3%).
```



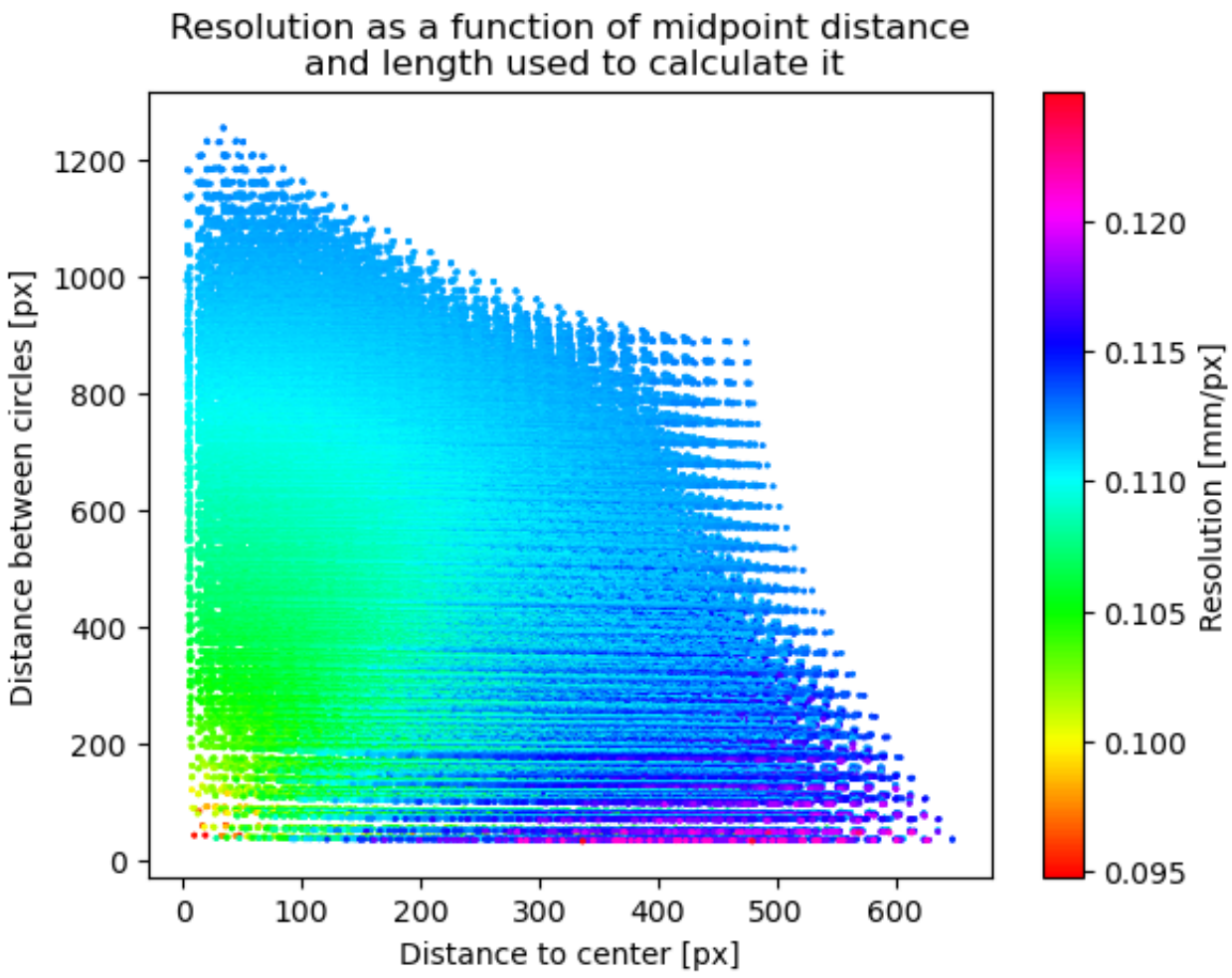
Although the average resolution is similar to the image in part a1, and the noise similar to the image of part a2, the standard deviation of the resolution is now about 10 times larger (1.3% of the average instead of 0.2%). This will indeed be due to the distortion. Let's investigate what the distortion looks like by considering the resolution as a function of the radial distance from the centre.

Above, we have calculated the resolution for all possible distances that exist in the grid, from all nearest neighbours (the largest error) to the diagonals (the smallest error). Where, for the average resolution, we gave the diagonals the largest weight, there is no use for them when calculating the local resolution. Let's see what the effect is of this length.

We start by giving each entry in the resolution matrix an associated coordinate given by the midpoint between the two circles used to calculate that entry.

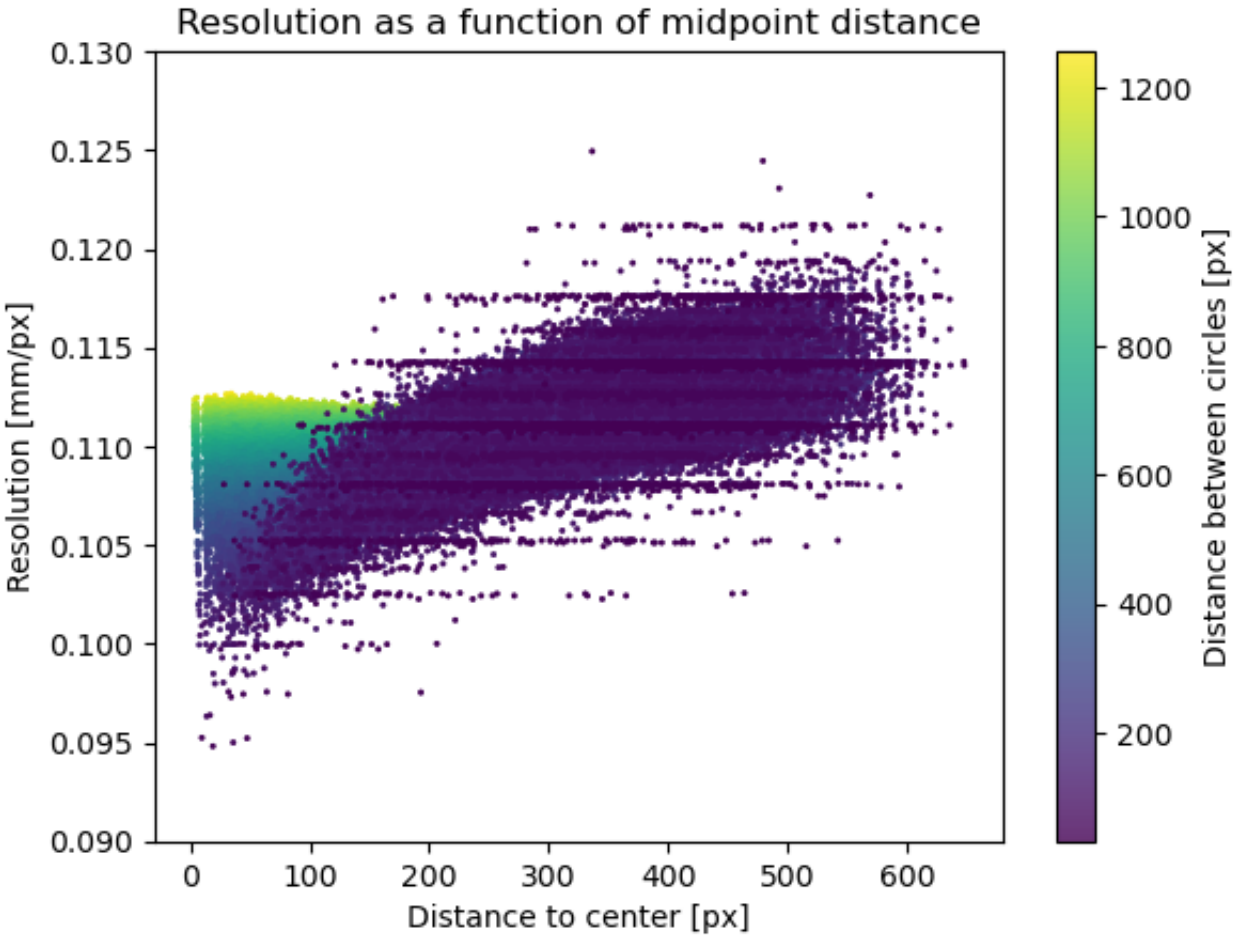
In [ ]:
```
# Verify the coordinates in pixel space
middle_img = [img_proc.shape[0] / 2, img_proc.shape[1] / 2]

# Calculate the midpoint coordinates of each combination of circles
midpts_px = (cent_px_sort[:, None] + cent_px_sort) / 2

# Calculate the distance of these midpoints to the center of the image
```

```
middist_px = np.sqrt(np.sum((midpts_px - middle_img)**2, axis=2))

# Calculate the angle of the midpoints in degrees
middist_angles = (np.arctan2(midpts_px[:, :, 0] - middle_img[0],
                             midpts_px[:, :, 1] - middle_img[1])\
                               * 180 / np.pi)

# Flatten dist_px_sort, middist_px and res and mask out the diagonal
mask = np.eye(middist_px.shape[0], dtype=bool)
dist_px_sort_flat = dist_px_sort[~mask].copy().flatten()
middist_px_flat = middist_px[~mask].copy().flatten()
middist_angles_flat = middist_angles[~mask].copy().flatten()
res_flat = res[~mask].copy().flatten()

# Sort by decreasing dist_px_sort_flat
sort_id = np.argsort(dist_px_sort_flat)[::-1]
dist_px_sort_flat = dist_px_sort_flat[sort_id]
middist_px_flat = middist_px_flat[sort_id]
res_flat = res_flat[sort_id]

# Plot the resolution as a function of the midpoint distance and length
fig, ax = plt.subplots()
ax.scatter(middist_px_flat, dist_px_sort_flat, c=res_flat, s=1, alpha=1, cmap='hsv')
cbar = plt.colorbar(ax.collections[0], ax=ax, orientation='vertical')

ax.set_title('Resolution as a function of midpoint distance\n and length used to calculate it')
ax.set_xlabel('Distance to center [px]')
ax.set_ylabel('Distance between circles [px]')
cbar.set_label('Resolution [mm/px]')
plt.show()
```



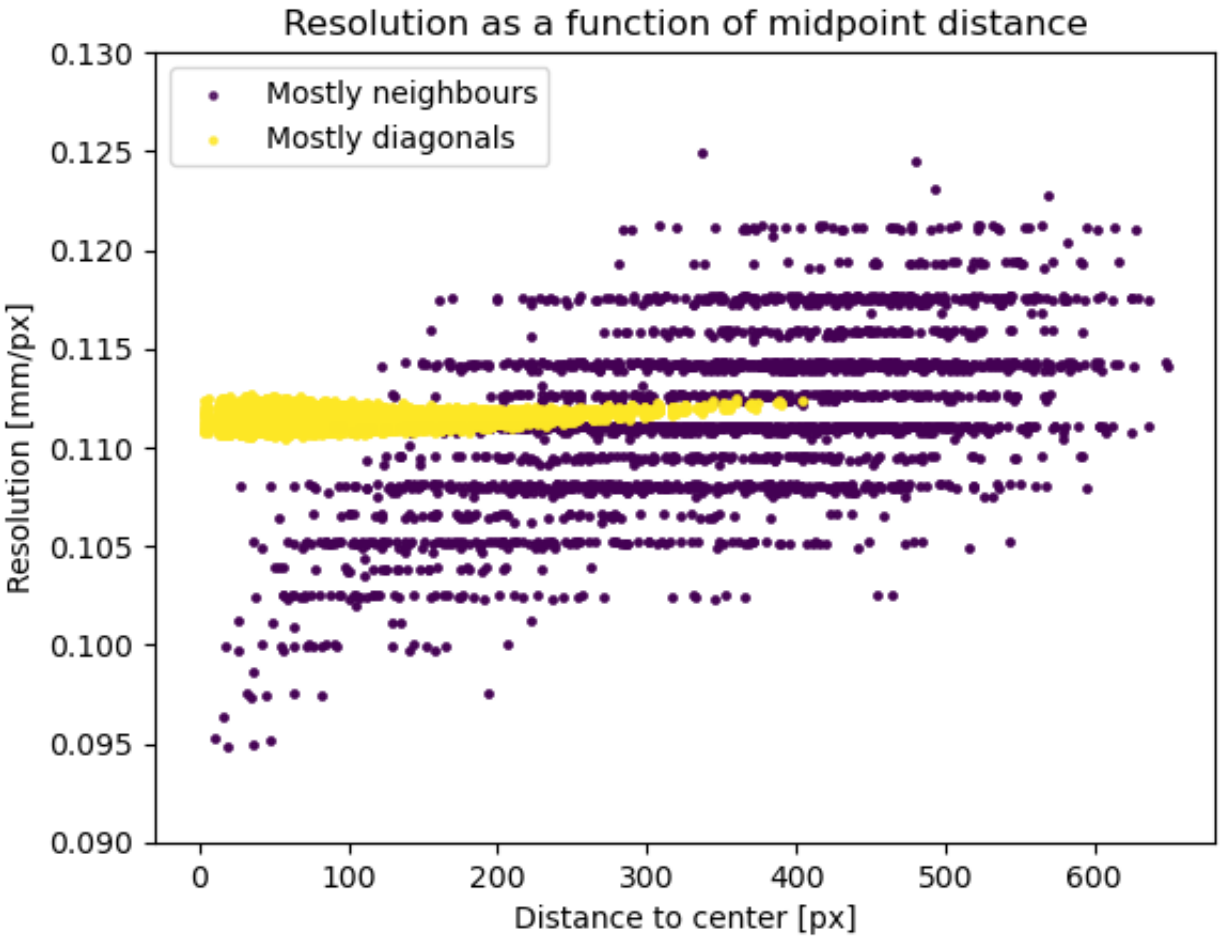
As suspected, the larger the distance between the two circles, the less variation in resolution. Let us discard all points with an associated distance larger than 200 px.

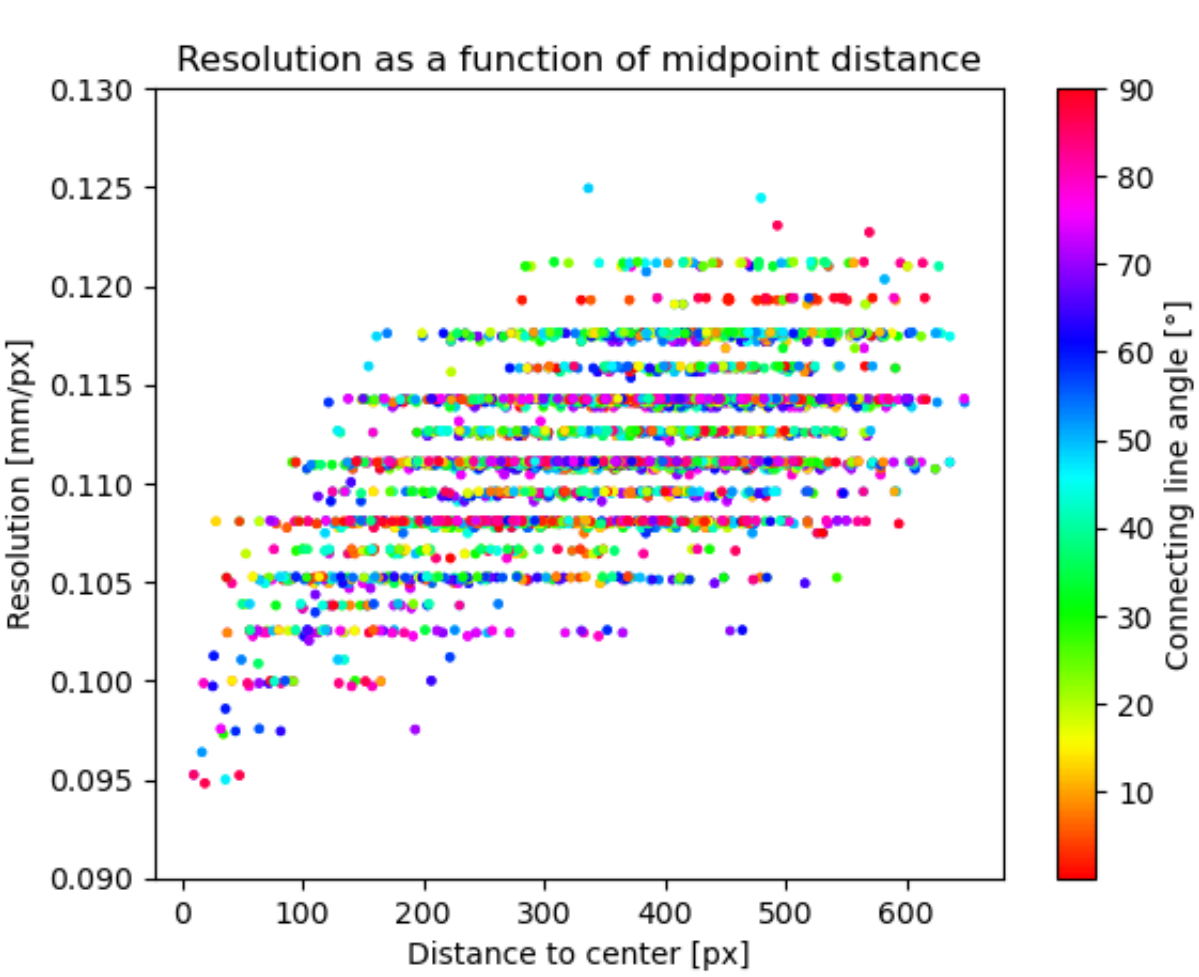
In [ ]:
```
# Make a plot of the resolution as a function of the midpoint distance
fig, ax = plt.subplots()
ax.scatter(middist_px_flat, res_flat, c=dist_px_sort_flat, s=1, alpha=0.8)
cbar = plt.colorbar(ax.collections[0], ax=ax, orientation='vertical')
ax.set_ylim([0.09, 0.13])
ax.set_title('Resolution as a function of midpoint distance')
ax.set_xlabel('Distance to center [px]')
ax.set_ylabel('Resolution [mm/px]')
cbar.set_label('Distance between circles [px]')
```



Here, the limits of the `cv.HoughCircles` algorithm also become clear, because the resolution calculated from nearest neighbours shows up in discrete stripes. So a subpixel accuracy circle finder would be ideal in this case.

In [ ]:
```python
# Now use only the nearest neighbours and diagonals by taking a smaller limit
mask_sm = dist_px_sort_flat < 60
mask_la = dist_px_sort_flat > 900

# Get the colors at the end of the colormap
cmap = plt.get_cmap('viridis');
color1 = cmap(0);
color2 = cmap(255);

# Make a plot of the resolution as a function of the midpoint distance (nearest neighbours)
fig, ax = plt.subplots()
ax.scatter(middist_px_flat[mask_sm], res_flat[mask_sm], color=color1, s=5, alpha=0.8, label='Mostly neighbours')
ax.scatter(middist_px_flat[mask_la], res_flat[mask_la], color=color2, s=5, alpha=0.8, label='Mostly diagonals')
ax.set_ylim([0.09, 0.13])
ax.set_title('Resolution as a function of midpoint distance')
ax.set_xlabel('Distance to center [px]')
ax.set_ylabel('Resolution [mm/px]')
ax.legend()
plt.show()

# Cycle around 90 degrees and round to the nearest degree
middist_angles_flat = np.round(np.mod(middist_angles_flat, 90), 1)

# Make another plot using angle as colour
fig, ax = plt.subplots()
ax.scatter(middist_px_flat[mask_sm], res_flat[mask_sm],
           c=middist_angles_flat[mask_sm], s=5, cmap='hsv')
cbar = plt.colorbar(ax.collections[0], ax=ax, orientation='vertical')
cbar.set_label('Connecting line angle [°]')

ax.set_ylim([0.09, 0.13])
ax.set_title('Resolution as a function of midpoint distance')
ax.set_xlabel('Distance to center [px]')
ax.set_ylabel('Resolution [mm/px]')
plt.show()
```





Above, we hoped to be able to see whether the distortion was angle-dependent. It seems like this is not the case, but it might as well be the inaccuracies of our grid finder disguising any pattern.

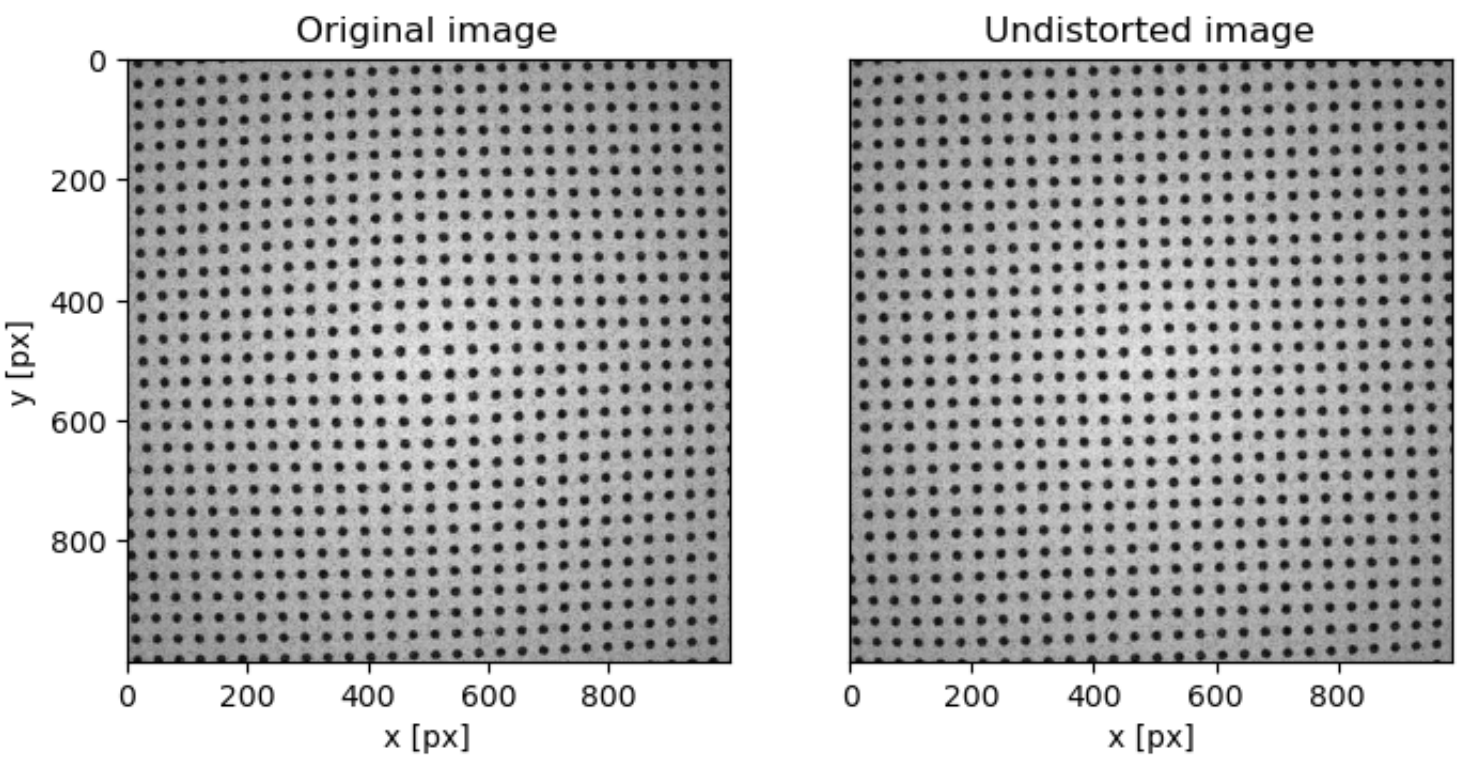
Undistorting the image

Using the grid coordinates and the pinhole distortion model as seen on the OpenCV site (https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html), we can undistort the image. Given more time, these parameters could be compared to verify whether the pinhole model is appropriate or not.

In [ ]:
```
# Reshape the centers array
cent_px_undist = cent_px_sort.reshape((-1,1,2)).astype(np.float32)

# Prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((grid_size_id[0] * grid_size_id[1], 3), np.float32)
objp[:,:2] = np.mgrid[0:grid_size_id[0], 0:grid_size_id[1]].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

objpoints.append(objp)

# Termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

imgpoints.append(cent_px_undist)
```

```
imgpoints = np.array(imgpoints, dtype=np.float32)

ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, img.shape[::-1], None, None)
h,  w = img.shape[:2]
newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))

# Undistort
img_dst = cv.undistort(img, mtx, dist, None, newcameramtx)

# Crop the image
x, y, w, h = roi
img_dst = img_dst[y:y+h, x:x+w]

# Compare original and undistorted
fig, ax = plt.subplots(1, 2, figsize=(8, 5))

ax[0].imshow(img, cmap='gray')
ax[0].set_title('Original image')
ax[0].set_xlabel('x [px]')
ax[0].set_ylabel('y [px]')

ax[1].imshow(img_dst, cmap='gray')
ax[1].set_title('Undistorted image')
ax[1].set_xlabel('x [px]')
ax[1].set_yticks([])
plt.show()

# Calculate undistortion error
```

```
mean_error = 0
for i in range(len(objpoints)):
    imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, dist)
    error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2)/len(imgpoints2)
    mean_error += error
print(f"Total undistortion error: {mean_error/len(objpoints):.4f} px")
```




```
Total undistortion error: 0.0558 px
```

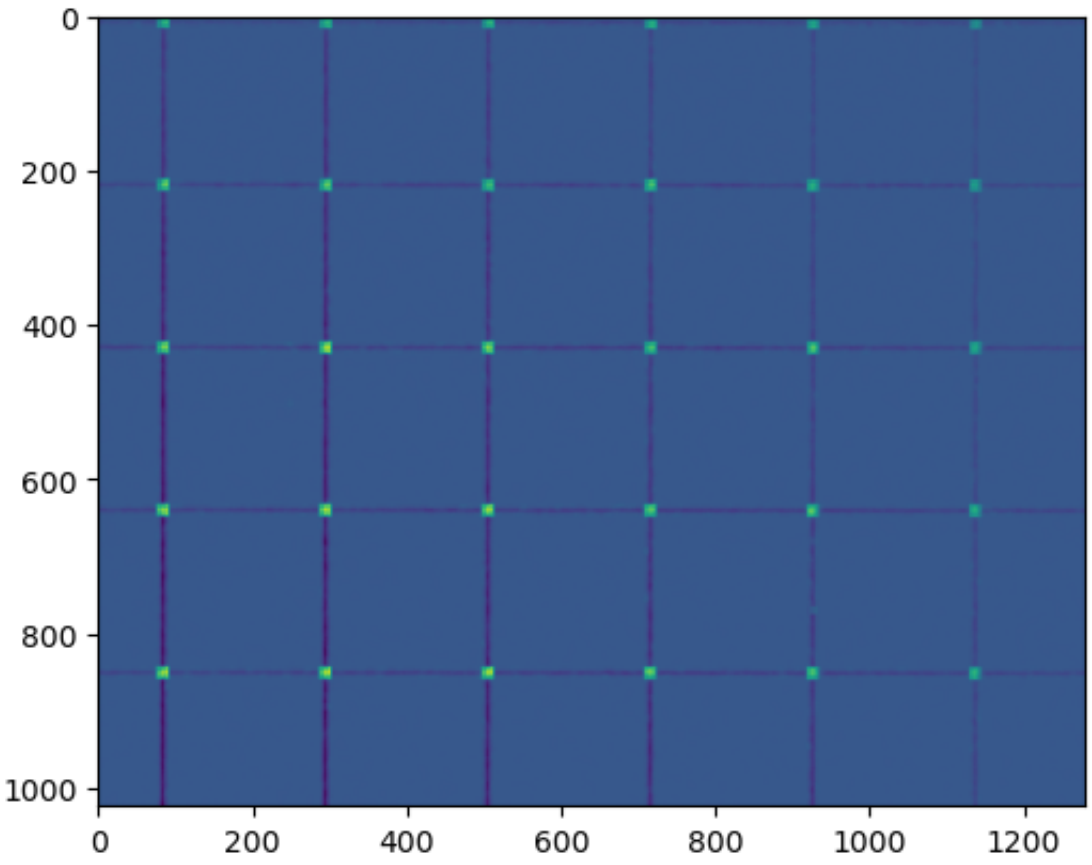
Part b: Droplet evaporation and Contact angle measurement (12 points)

b1) **Calibration** (2 points)

We start by finding the calibration of the camera. We use a Harris corner detector to find the crossing points of the lines in our calibration image, which can then be fed into our calibration function from (a). We start with the larger grid.

In [ ]:
```
#import the image, converted to grayscale
gray = cv.imread('Images/Calibration_b/20201204-2x-500um.tif', 0)

#find corners and plot on original image
img_corner = cv.cornerHarris(gray,10,3,0.04)
img_corner1 = np.copy(img_corner)

#to detect the corners, we have to get rid of the rest
# the image, then we can find the points using connectedComponents function
ret, thresh = cv.threshold(img_corner,0.1*img_corner.max(),255,0)
thresh = np.uint8(thresh)
ret, labels, stats, centroids = cv.connectedComponentsWithStats(thresh)
plt.imshow(img_corner)

#define calibration distance
```

```
calibration_distance = 0.5 #mm/px
```



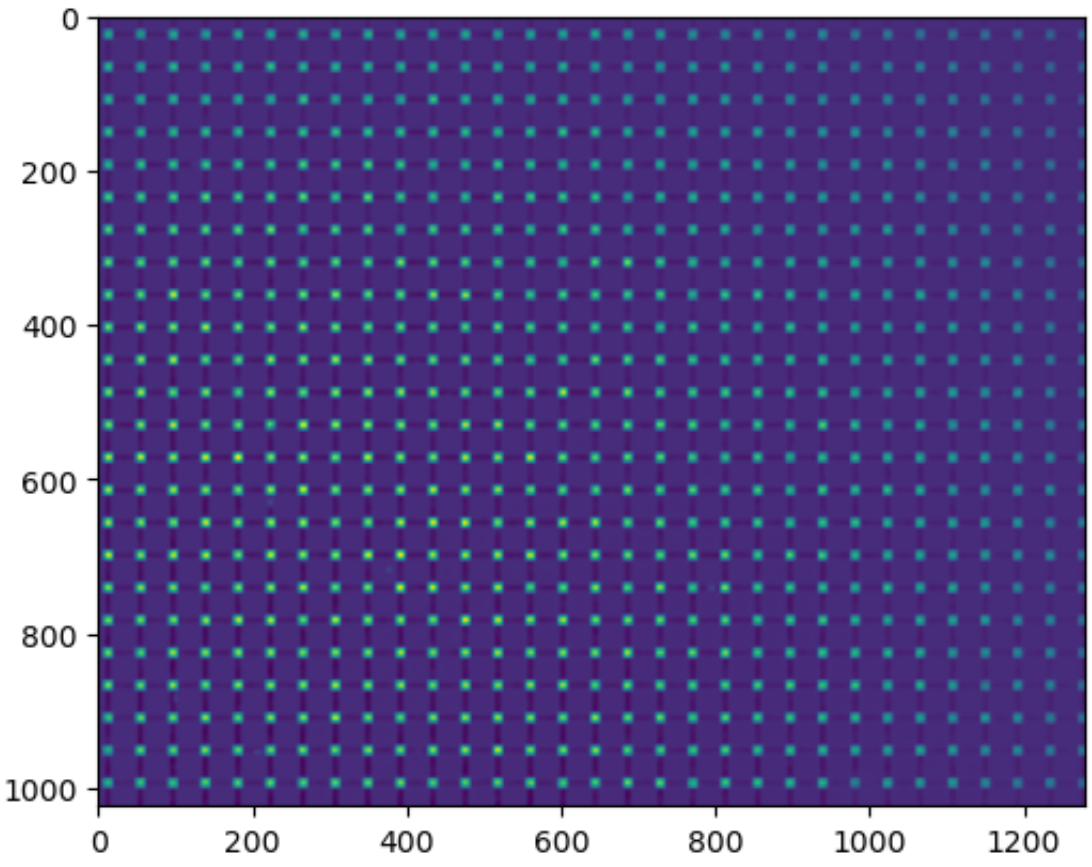
In [ ]:
```
# Calculate the angle of the circles in the image
angle_avg, angle_std, angle_ct, line_ct = calc_angle(gray, centroids, min_votes=10, verbose=True)

# Sort the circle coordinates, take radii as 1
radii_dummy = 1*np.ones(centroids.shape[0])
cent_px_sort, _, cent_id, grid_size_id = sort_circles(centroids, radii_dummy, discard=True)

# Calculate the distance between all points in real space and index space
dist_px_sort = calc_distance_between_all_points(cent_px_sort[:, 0], cent_px_sort[:, 1])
dist_id_sort = calc_distance_between_all_points(cent_id[:, 0], cent_id[:, 1])

# Calculate the resolution for each distance
res = calc_resolution(dist_px_sort, dist_id_sort, calibration_distance)

# Calculate the weighted average and standard deviation
calibration1, dc1 = weighted_avg_and_std(res, dist_id_sort, precision=7, verbose=True)
```

```
The angle of the grid in the image is 0.0° with a standard deviation of 0.0°.
This was calculated using 8/11 lines found.
The average resolution is 0.0023726 mm/px,
with a standard deviation of 0.0000011 mm/px (0.0%).
```

Convinced that the smaller grid will definitely not be angled either, we calculate the resolution given by that grid directly.

In [ ]:
```
# Follow the same steps as above
gray = cv.imread('Images/Calibration_b/20201204-2x-100um.tif', 0)
img_corner = cv.cornerHarris(gray,10,3,0.04)
img_corner1 = np.copy(img_corner)
ret, thresh = cv.threshold(img_corner,0.1*img_corner.max(),255,0)
thresh = np.uint8(thresh)
ret, labels, stats, centroids = cv.connectedComponentsWithStats(thresh)
plt.imshow(img_corner)

#define new calibration distance
calibration_distance = 0.1 #mm/px

# Continue, take radii as 1 this time
radii_dummy = 1*np.ones(centroids.shape[0])
cent_px_sort, _, cent_id, grid_size_id = sort_circles(centroids, radii_dummy, discard=True)
dist_px_sort = calc_distance_between_all_points(cent_px_sort[:, 0], cent_px_sort[:, 1])
dist_id_sort = calc_distance_between_all_points(cent_id[:, 0], cent_id[:, 1])
res = calc_resolution(dist_px_sort, dist_id_sort, calibration_distance)
calibration2, dc2 = weighted_avg_and_std(res, dist_id_sort, precision=7, verbose=True)
```

```
The average resolution is 0.0024013 mm/px,
with a standard deviation of 0.0000646 mm/px (2.7%).
```



Because calibration using the larger grid gives a smaller error, we use that resolution value, and convert to um/px for further use.

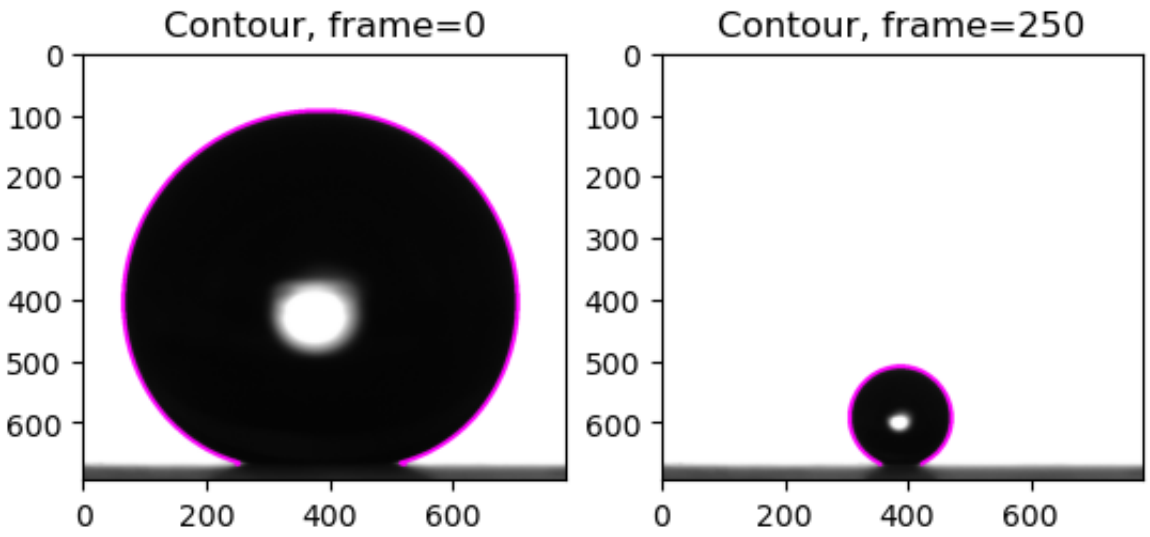
In [ ]:
```
calibration, dc = 1000*calibration1, 1000*dc1
```

b2) **Drop Shape and Contact Angle** (4 points)

Finding the drop shape takes a few steps. First we import the image, then convert it to gray-scale. We then threshold it, to get a binary-image. The contour function is then used to find the contour of the drop. We then have to do some masking of the contours to extract the drop shape. One could simply slice the image in a normal situation, but for some reason the function kept detecting the edges of the frame and drop shape as one large contour. So we would have to get rid of these via masking anyways. The height of the substrate was determined by thresholding the image and finding the height of the substrate manually.

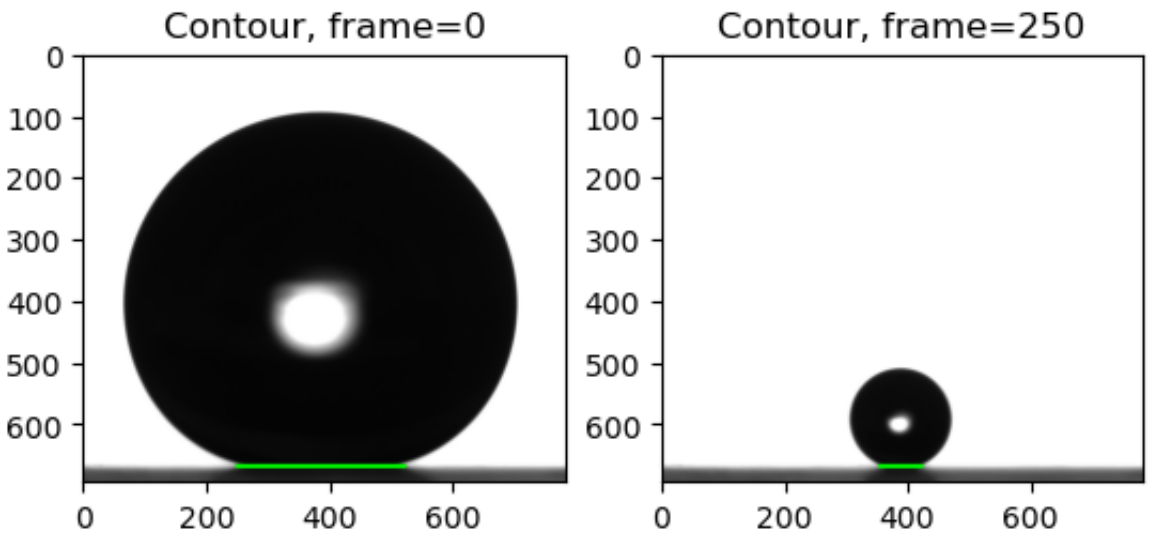
Another option which we chose not to use, is to use the final images which only show the base and subtract these from our image we want to analyse. We chose not to do this, because this did not perfectly extract the drop and we would still have to slice the contours to properly get the drop shape. Finally, because our edge detection detected the edge of the image, we still have to slice that off. So in our case masking was easier and works well. In the second set of images, we also have no base to substract, as we have no empty images. This means we would have to rewrite part of our code, which we now do not have to do.

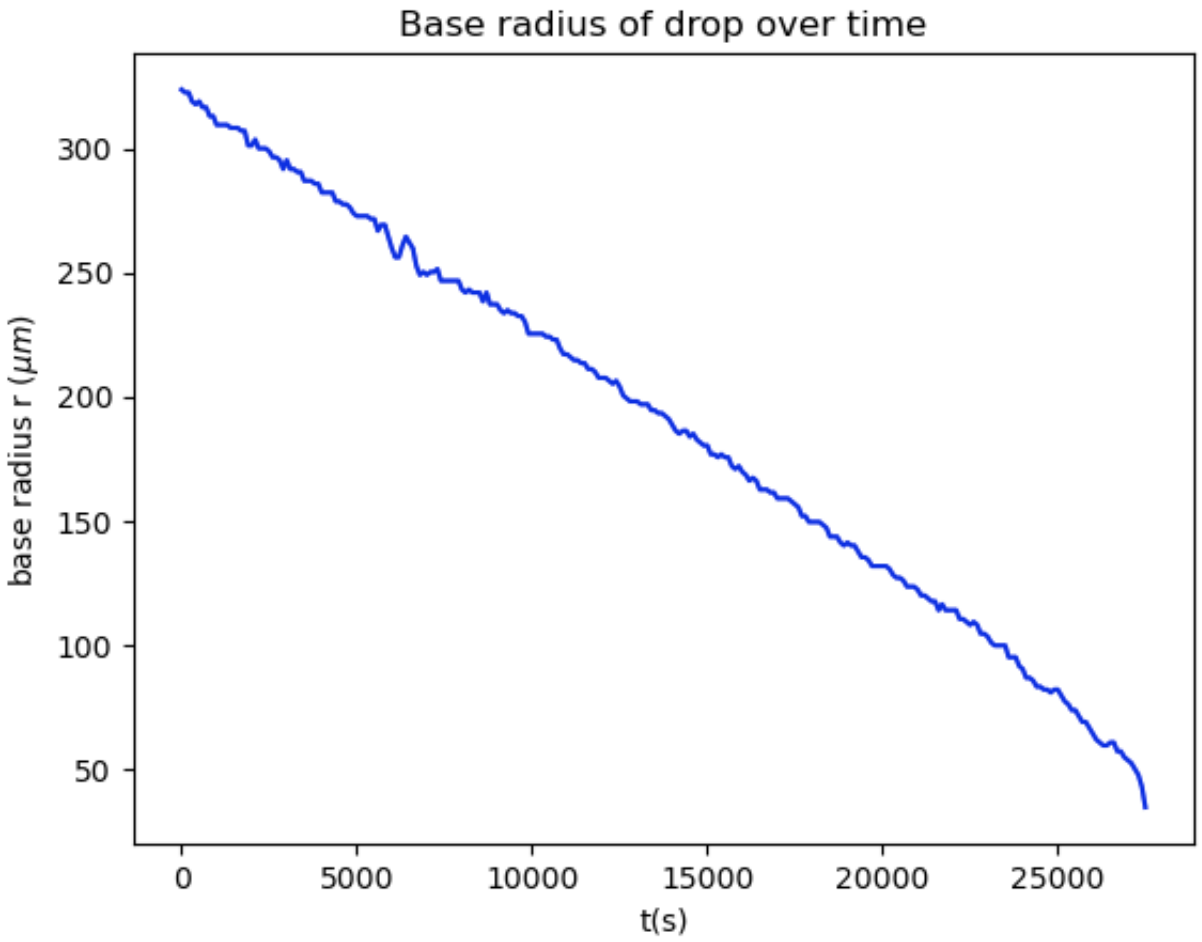
In [ ]:
```
ysubstrate = 670
for i in range(290+1):
    num = f'{i}' #pad number to 0007 shape
    # Import the image, convert to gray
    img = cv.imread(f'Images/EvaporatingDroplet/experiment{num.zfill(4)}.tif')
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    #threshold
    _,thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

    #find contours
    contours, _ = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

    points = np.vstack(contours[-1]).squeeze()
    x, y = points[:,0], points[:,1]

    # taking off the edges of the frame it keeps detecting >:(, and the bottom substrate
    mask = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<ysubstrate)
    points = points[mask]

    #reshape into a contour
    ctr = np.array(points).reshape((-1,1,2)).astype(np.int32)

    cv.drawContours(img, ctr, -1, (255,0,255), 5)

    if i == 0:
        f = plt.figure()
        f.add_subplot(1,2, 1)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
    if i == 250:
        f.add_subplot(1,2, 2)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
plt.show()
```



To fit the base radius, we find the contours at the height of the substrate and find the minimum and maximum values of said contour at that height. The difference between these two is $2r$. We do this up until image 275, where the drop has completely evaporated and no contours are detected anymore.

In [ ]:
```
stop = 275
base_radius = np.zeros(stop+1)
fps = 0.01
dt = 1/fps

for i in range(stop+1):
    num = f'{i}' #pad number to 0007 shape
    # Import the image, convert to gray
    img = cv.imread(f'Images/EvaporatingDroplet/experiment{num.zfill(4)}.tif')
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    #threshold
    _,thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

    #find contours
    contours, _ = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

    points = np.vstack(contours[-1]).squeeze()
    x, y = points[:,0], points[:,1]

    # finding base radius, find points at height of substrate
    mask2 = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y==ysubstrate-1)
    xbase = x[mask2]
    x1 = np.min(xbase)
    x2 = np.max(xbase)

    #base radius is half of base
    base_radius[i] = abs(x1-x2)/2

    img = cv.line(img, (x1, ysubstrate-1), (x2, ysubstrate-1), (0,255,0), 5)
    if i == 0:
        f = plt.figure()
        f.add_subplot(1,2, 1)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
    if i == 250:
        f.add_subplot(1,2, 2)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
plt.show()

#error calculation
dr = np.sqrt((base_radius**2)*(dc**2))

t1 = np.linspace(0, len(base_radius)*dt-dt, len(base_radius))
plt.figure()
plt.errorbar(t1, base_radius*calibration, yerr=dr, alpha=0.4)
plt.plot(t1, base_radius*calibration, 'b')
plt.xlabel('t(s)')
plt.ylabel(r'base radius r ($\mu m$)')
plt.title('Base radius of drop over time')
plt.show()
```





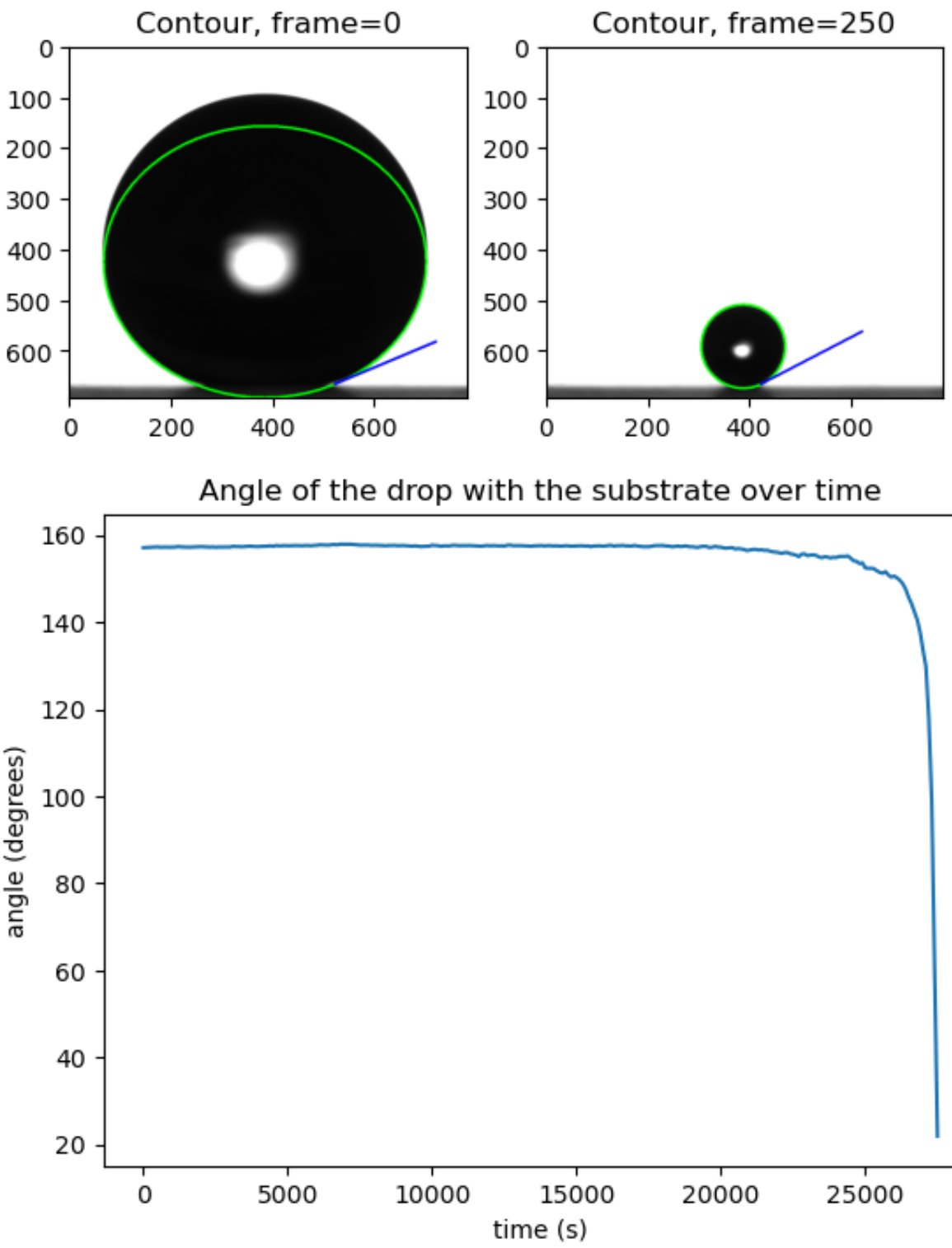
When we fit an ellipse or circle through the whole drop, this ellipse usually does not match the drop well near the bottom. The drop is not an ellipse/circle. So for the first 250 frames, only the bottom part of the drop is fitted to the ellipse, so it fits well at the bottom. Then after 250 frames the bubble becomes rounded enough to fit the whole drop again. We then use the fitted ellipse parameters to calculate a bunch of points at and slightly above the substrate on one side, then fit a line through these points. Then we calculate the angle using this line. When the drop shrinks and has an angle of more than 90 degrees, we have to subtract the found angle from 180 degrees, because of how the angle is defined in regards to the axis.

Note that in the code below, we first calculated the "wrong" angle. This is corrected in the plot below by subtracting the calculated angle from 180 degrees.

In [ ]:
```
def func(x, a, b):
    return a*x + b


h, w = 694, 784
ysubstrate = 670

angle = np.zeros(275+1)
da = np.zeros(275+1)

for i in range(275+1):
    num = f'{i}' #pad number to 0007 shape
    # Import the image, convert to gray
    img = cv.imread(f'Images/EvaporatingDroplet/experiment{num.zfill(4)}.tif')
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    #threshold
    _,thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

    #find contours
    contours, _ = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

    points = np.vstack(contours[-1]).squeeze()
    x, y = points[:,0], points[:,1]

    # taking off the edges of the frame it keeps detecting >:(, and the bottom substrate
    mask = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<ysubstrate)
```

```
    points = points[mask]

    #reshape into a contour
    ctr = np.array(points).reshape((-1,1,2)).astype(np.int32)

    #draw contours on image
    #cv.drawContours(img, ctr, -1, (255,0,0), 5)

    ## Draw
    canvas = np.zeros_like(img)
    cv.drawContours(canvas , ctr, -1, (0, 255, 0), 1)

    circles = cv.HoughCircles(canvas[:,:,1],cv.HOUGH_GRADIENT,1,300,
                              param1=50,param2=10,minRadius=0, maxRadius=0)

    r = circles[0,0,2]
    xc = circles[0,0,0]
    yc = circles[0,0,1]

    #cv.circle(img,(int(round(xc)),int(round(yc))), int(round(r)), (0,0,255), 3)

    if i < 250:
        points = np.vstack(contours[-1]).squeeze()
        x, y = points[:,0], points[:,1]

        # taking off the edges of the frame it keeps detecting >:(, and the bottom substrate
        mask = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<ysubstrate) & (y>yc)
        points = points[mask]

        #reshape into a contour
        ctr = np.array(points).reshape((-1,1,2)).astype(np.int32)

    # finding base radius
    mask2 = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<ysubstrate) & (y==ysubstrate-1)
    xbase = x[mask2]
    x1 = np.min(xbase)
    x2 = np.max(xbase)

    ellipse = cv.fitEllipse(ctr)

    u=ellipse[0][0]         #x-position of the center
    v=ellipse[0][1]         #y-position of the center
    a=ellipse[1][1]/2       #radius on the x-axis
    b=ellipse[1][0]/2      #radius on the y-axis

    t = np.linspace(0, 2*np.pi, 10000)

    y = v+b*np.sin(t)
    x = u+a*np.cos(t)

    # for bonus, fit both circle and ellipse
    #cv.ellipse(img, ellipse, (255,0,0), 3)

    #fit line and determine angle
    mask = (y<ysubstrate) & (y>660) & (x>xc)
    y = y[mask]
    x = x[mask]


    popt, pcov = curve_fit(func, x, y)
    de = np.sqrt(np.diag(pcov))[0]

    y2 = np.round(func(x2, *popt))
    y3 = np.round(func(x2+200, *popt))

    angle[i] = abs(np.arctan(popt[0])*180/np.pi)
    da[i] = np.sqrt( ((1/((popt[0]**2) + 1))**2)*(de**2)  )
    if i >= 274:
        angle[i] = 180 - abs(np.arctan(popt[0])*180/np.pi)
    #print(i, angle[i])
    #draw baseradius, fitted line and ellipse on circle
    cv.ellipse(img, ellipse, (0,255,0), 3)
    img = cv.line(img, (x2, int(y2)), (x2+200, int(y3)), (0,0,255), 3)

    if i == 0:
        f = plt.figure()
        f.add_subplot(1,2, 1)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
    if i == 250:
        f.add_subplot(1,2, 2)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
plt.show()
```

```
plt.figure()
plt.errorbar(t1, 180-angle, yerr=da)
plt.xlabel('time (s)')
plt.ylabel('angle (degrees)')
plt.title('Angle of the drop with the substrate over time')
plt.show()
```

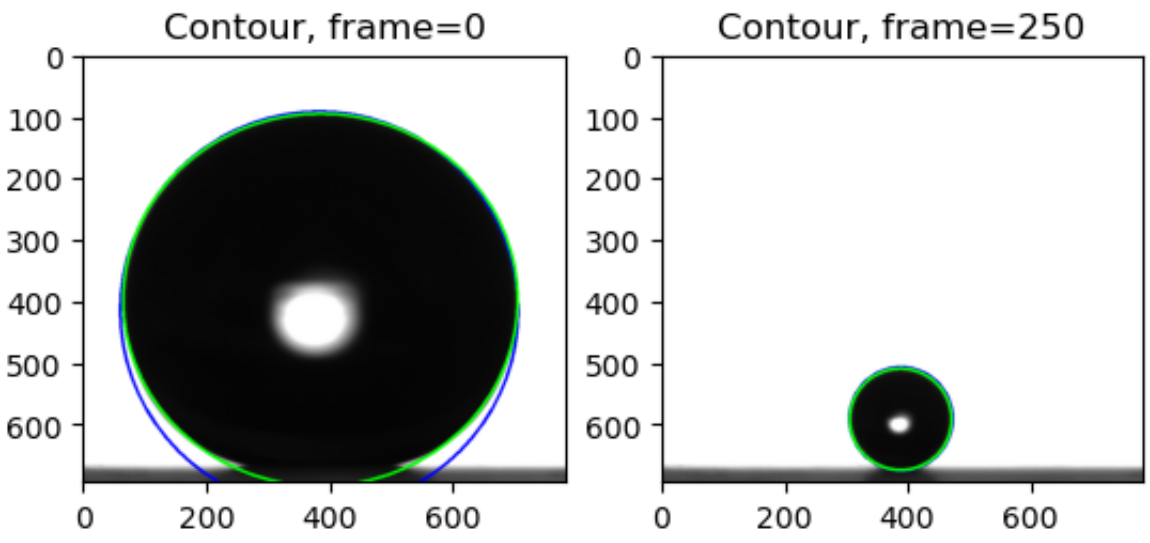


BONUS (2 points)

We now fit both a circle and an ellipse to the dropshape. The circle is fitted using a Hough transform. As we can see from the figures below, the circle (BLUE) fits well at the top, but not at the bottom. The same goes for the ellipse (GREEN), it fits well at the top, but not really at the bottom. As the drop evaporates and becomes more rounded, the circle and ellipse fit better. The better choice for fitting would be to use an ellipse, we decided to use an ellipse only fitted through the bottom half of the drop, because we are interested in the angle, so the top of the bubble is not of much interest.

In [ ]:
```
for i in range(275+1):
    num = f'{i}' #pad number to 0007 shape
    # Import the image, convert to gray
    img = cv.imread(f'Images/EvaporatingDroplet/experiment{num.zfill(4)}.tif')
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    #threshold
    _,thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

    #find contours
    contours, _ = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

    points = np.vstack(contours[-1]).squeeze()
    x, y = points[:,0], points[:,1]

    # taking off the edges of the frame it keeps detecting >:(, and the bottom substrate
    mask = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<ysubstrate)
    points = points[mask]

    #reshape into a contour
    ctr = np.array(points).reshape((-1,1,2)).astype(np.int32)
```

```
    ## Draw and fit circle
    canvas = np.zeros_like(img)
    cv.drawContours(canvas , ctr, -1, (0, 255, 0), 1)

    circles = cv.HoughCircles(canvas[:,:,1],cv.HOUGH_GRADIENT,1,300,
                              param1=50,param2=10,minRadius=0, maxRadius=0)

    r = circles[0,0,2]
    xc = circles[0,0,0]
    yc = circles[0,0,1]

    cv.circle(img,(int(round(xc)),int(round(yc))), int(round(r)), (0,0,255), 3)

    # for bonus, fit both circle and ellipse
    ellipse = cv.fitEllipse(ctr)
    cv.ellipse(img, ellipse, (0,255,0), 3)

    if i == 0:
        f = plt.figure()
        f.add_subplot(1,2, 1)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
    if i == 250:
        f.add_subplot(1,2, 2)
        plt.imshow(img)
        plt.title(f'Contour, frame={i}')
plt.show()
```



b3) **Interpretation of the plots** (1 point)

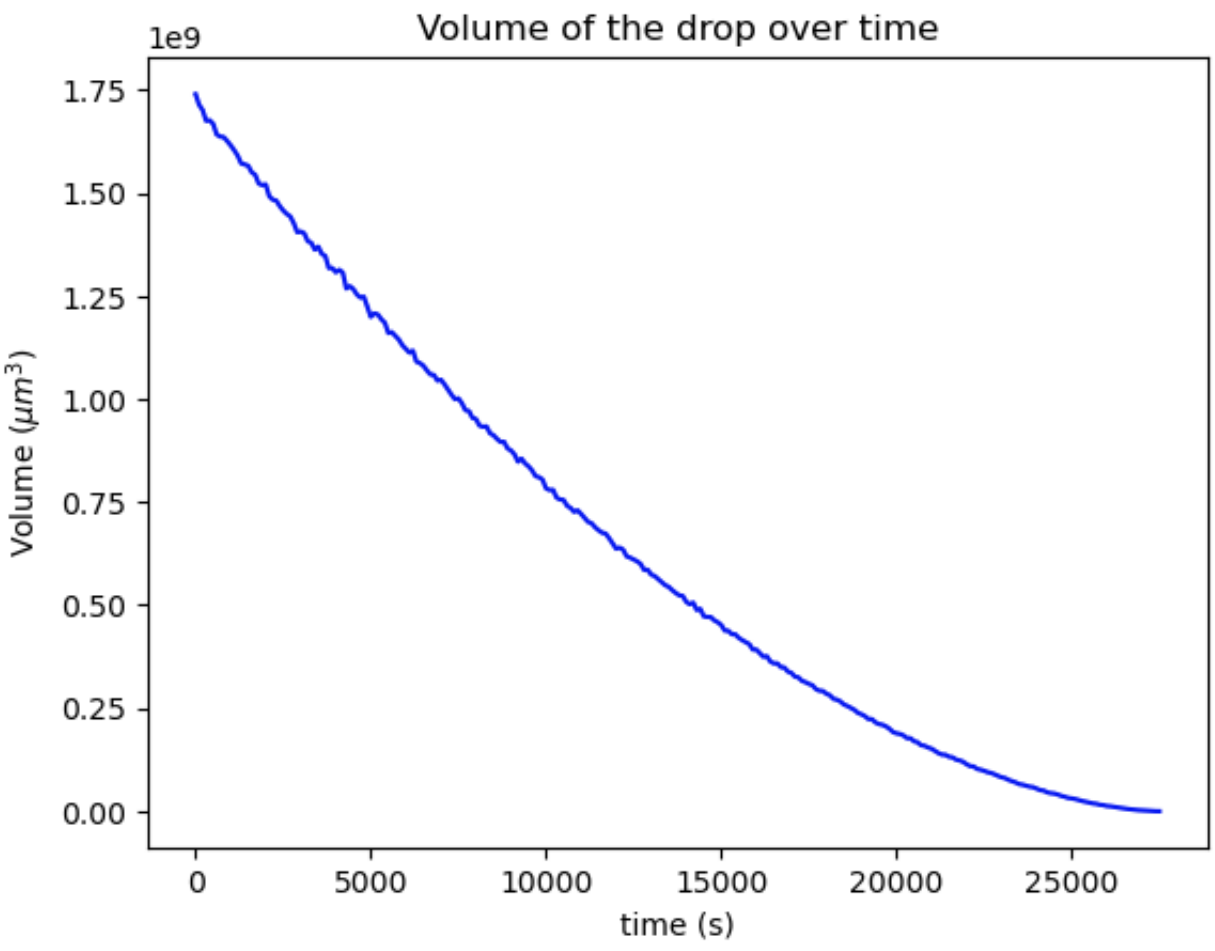
At around 6000 seconds, we can see a wobble in the plotted base-radius, where the drop seems to oscillate for some reason. We can also see that the base radius goes down quite linearly, except at the end where it suddenly goes down faster. We see something similar in the angle, which goes up very slowly at first, but then around the time where the base radius shoots down, the angle shoots up. We also see the angle shoot up when the drop goes from being a sphere to being more of a dome on top of the substrate.

b4) **Drop volume** (2 points)

We calculate the drop voluming by taking the formula for a solid of revolution and plugging in the contour of the drop. Instead of rotating around the x-axis, we rotate around the y-axis. We use the equation: $\pi \int x^2 dy$, where we have shifted our contour to have its center at the y-axis and only calculate for the points to the right of the y-axis, and rotate around the y-axis.

In [ ]:
```
volume = np.zeros(275+1)

for i in range(275+1):
    num = f'{i}' #pad number to 0007 shape
    # Import the image, convert to gray
    img = cv.imread(f'Images/EvaporatingDroplet/experiment{num.zfill(4)}.tif')
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    #threshold
    _,thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

    #find contours
    contours, _ = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

    points = np.vstack(contours[-1]).squeeze()
    x, y = points[:,0], points[:,1]

    # taking off the edges of the frame it keeps detecting >:(, and the bottom substrate
    mask = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<ysubstrate)
    points = points[mask]
```

```
    #reshape into a contour
    ctr = np.array(points).reshape((-1,1,2)).astype(np.int32)

    ## Draw and fit circle
    canvas = np.zeros_like(img)
    cv.drawContours(canvas , ctr, -1, (0, 255, 0), 1)

    circles = cv.HoughCircles(canvas[:,:,1],cv.HOUGH_GRADIENT,1,300,
                              param1=50,param2=10,minRadius=0, maxRadius=0)

    r = circles[0,0,2]
    xc = circles[0,0,0]
    yc = circles[0,0,1]

    #calculate volume
    x, y = x[mask], y[mask]
    mask = x > xc
    volume[i] = np.pi*np.trapz((x[mask]-xc)**2, y[mask])

#error calculation
dV = np.sqrt(   ((3*volume*(calibration**2))**2 )  *(dc**2) )

plt.figure()
plt.errorbar(t1, volume*(calibration**3), yerr=dV, alpha=0.2)
plt.plot(t1, volume*(calibration**3), 'b')
plt.xlabel('time (s)')
plt.ylabel('Volume ($\mu m^3$)')
plt.title('Volume of the drop over time')
plt.show()
```



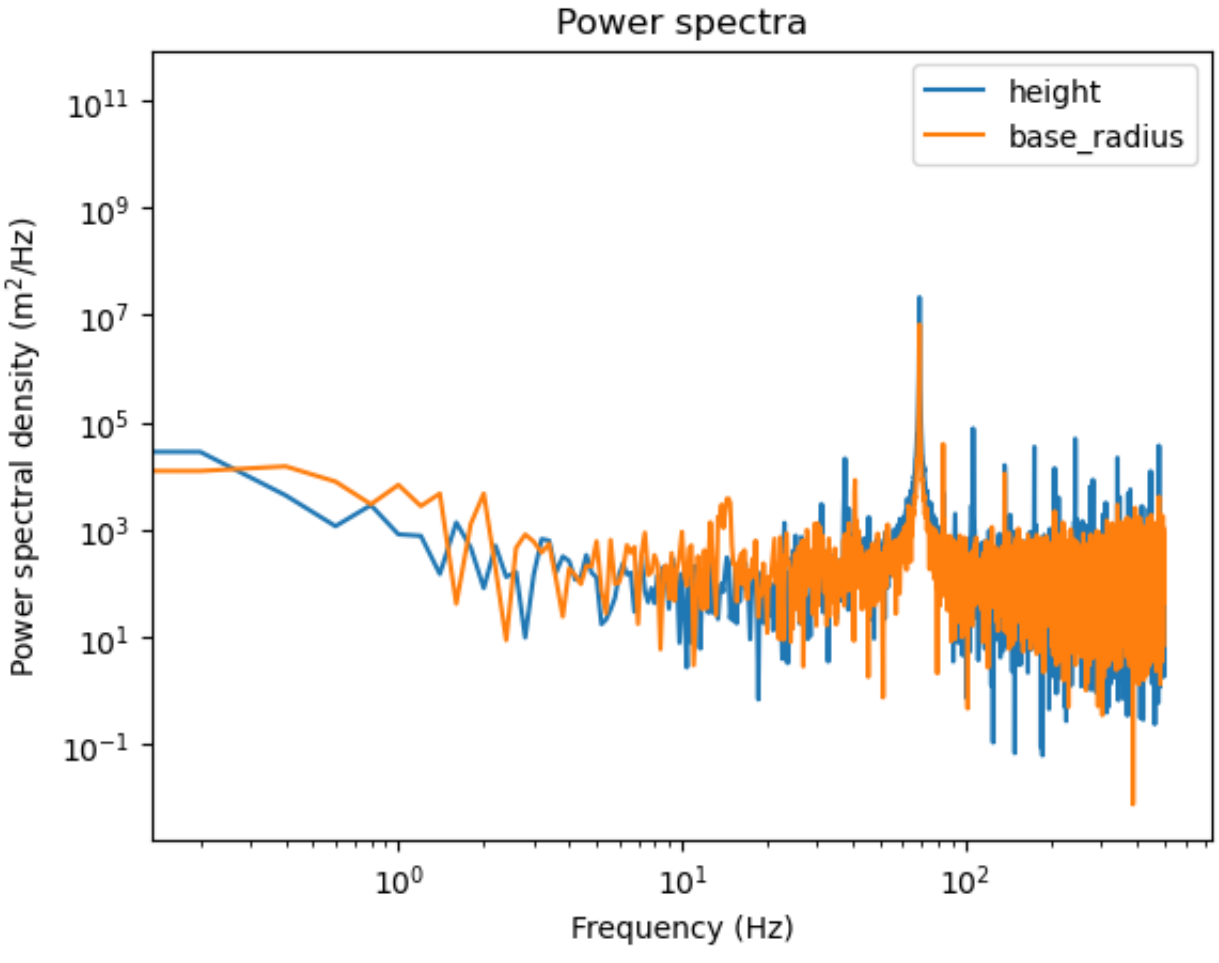
b5) **Oscillating droplet** (3 points)

We use two parameters for the drop, to check if both give approximately the same spectrum. We use the base radius over time and the height of the drop. We then do a Fourier transform, this is the same one we used for the previous assignment. The height of the substrate has now changed too. It was determined in a similar way as before.

As we can see in the image below, the peak in the power spectrum is found at 69Hz, which is the oscillation frequency of the droplet.

In [ ]:
```
nN = 4999#290 #number of images
fps = 1000
dt = 1/fps
y_arr = np.zeros(nN+1)
base_radius = np.zeros(nN+1)

for i in range(nN+1):
    num = f'{i}' #pad number to 0007 shape
    img = cv.imread(f'Images/OscillatingDroplet/experiment{num.zfill(4)}.tif')
    img2 = np.copy(img)
    #turn image gray
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    #gaussian blur
    gray = cv.GaussianBlur(gray,(5,5),0)
```

```
    #threshold
    _,thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

    #find contours
    contours, _ = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)

    points = np.vstack(contours[-1]).squeeze()
    x, y = points[:,0], points[:,1]


    # taking off the edges of the frame it keeps detecting >:(, and the bottom substrate
    mask = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<436)# & (y>300)
    points = points[mask]
    y_arr[i] = np.min(y[mask])

    # finding base radius
    mask2 = (y != 0) & (x != 0) & (x != np.max(x)) & (y != np.max(y)) & (y<436) & (y==436-1)
    xbase = x[mask2]
    x1 = np.min(xbase)
    x2 = np.max(xbase)
    base_radius[i] = abs(x1-x2)/2


def bartlett_fft(signal, fs, n_pieces):
    # Cut the signal into n equally-sized pieces
    signal_list = np.split(signal, n_pieces)
    P_list = []

    # Calculate power spectrum of each piece
    for piece in signal_list:
        P_list.append(np.abs(np.fft.fft(piece))**2)

    # Calculate the average power spectrum
    P_av = np.mean(P_list, axis=0)

    # Calculate the frequency array
    f = np.fft.fftfreq(piece.size, 1/fs)
    idx = np.argsort(f)

    # Get only positive frequencies
    f, P_av = f[idx], P_av[idx]

    return f, P_av


fs = fps # Hz
n_pieces = 1

f1, PS1 = bartlett_fft(base_radius, fs, n_pieces)
f, PS = bartlett_fft(y_arr, fs, n_pieces)
plt.figure()
plt.loglog(f, PS, '-', label='height')
plt.loglog(f1, PS1, '-', label='base_radius')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power spectral density (m$^2$/Hz)')
plt.title('Power spectra')
plt.legend()
plt.show()
```


Power spectra
height
base_radius
Power spectral density (m²/Hz)
Frequency (Hz)
10¹¹
10⁹
10⁷
10⁵
10³
10¹
10⁻¹
10⁰
10¹
10²