**3. Main Processing Loop**

Within an infinite loop, the script performs the following operations for each frame captured from the camera:

* **Frame Preprocessing**

Captures a frame from the camera and processes it using ***A\_utility.get\_processed\_frame***, which includes rotation and downsampling.

If it's the first iteration, it stores the initial frame (frame0) for later comparison.

* **Contact Area Calculation**

Applies inpainting on the current and initial frames using ***A\_utility.inpaint*** to remove certain features like markers.

Calculates the difference between the current and initial frames using ***A\_utility.difference*** to detect changes or contacts.

* **Convex Hull Analysis**

Extracts all contours from the difference image using ***A\_utility.get\_all\_contour***.

Identifies and analyzes the convex hull of the contact area, including its area and orientation (slope), using ***A\_utility.get\_convex\_hull\_area***.

* **Marker Detection and Optical Flow Analysis**

Detects markers in the current frame using ***A\_utility.marker\_center***.

Initializes and runs the optical flow or feature tracking algorithm using ***m.init(m\_centers)*** and ***m.run()***.

Retrieves the flow data (movement of markers) using ***m.get\_flow()***.

* **Flow Visualization**

Visualizes the flow of markers on the frame using ***A\_utility.draw\_flow***.

Overlays flow information specifically on the detected hull area using ***A\_utility.draw\_flow\_mask***.

* **Display and Data Publishing**

Displays the processed frame with visualized flow.

If a slope is calculated, sends the average flow change and slope as a ROS message for further use or analysis.

**4. Loop Termination and Cleanup**

* **Loop Control**

The script continuously processes frames until the 'q' key is pressed.

* **Resource Cleanup**

Releases the camera and closes all OpenCV windows.

**def get\_processed\_frame**

***get\_processed\_frame*** is designed to process the video stream captured by a camera. It performs two main operations: rotating the frame and downsampling it.

* **Camera Frame Capture**

***cam.read()***: This method captures a single frame from the video stream. The cam object is an instance of ***cv2.VideoCapture***, which handles the video capture functionality.

***ret, frame***: The ***read*** method returns two values:

***ret*** (a boolean): Indicates if the frame was successfully captured (***True***) or not (***False***).

***frame***: The actual image frame captured from the video stream.

* **Rotation**

***cv2.rotate(frame, cv2.ROTATE\_90\_CLOCKWISE)***: This line rotates the captured frame by 90 degrees clockwise.

Purpose of Rotation: The rotation is typically done to correct the orientation of the image. Depending on how the camera is mounted or positioned, the captured frames might need to be rotated to align them with a desired orientation or coordinate system.

* **Downsampling**

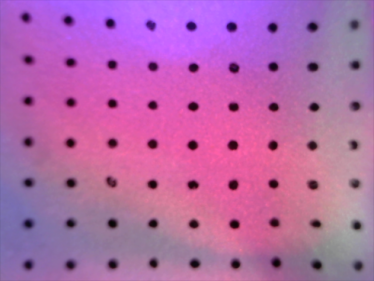
***cv2.pyrDown(rotated\_frame).astype(np.uint8)***: This line performs two operations:

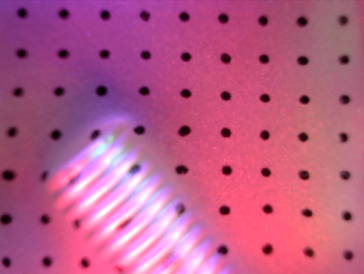
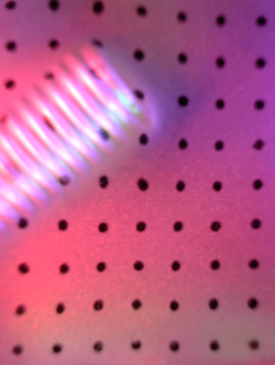
Downsampling: cv2.pyrDown is used to reduce the image size. Downsampling is a process of reducing the resolution of an image. In this case, the function reduces the width and height of the image to half of their original dimensions. It's part of OpenCV's image pyramids functionality, used for scaling images.

Type Conversion: ***.astype(np.uint8)***. After downsampling, the image is converted to a format with 8-bit pixel values (***np.uint8***). This ensures that the image has a standard data type, which is commonly used in image processing tasks.

* **Return Value**

The function returns ***downsampled***, which is the processed frame - rotated and downsampled, ready for further processing or analysis.

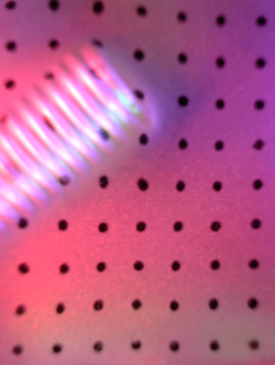
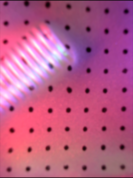
original(800\*600) —— (resize) —— rotate 90° and downsampling(400\*300)

**def mask\_marker**

The ***mask\_marker*** function is designed to identify and create a mask for the markers in the frame based on their visual characteristics.

* **Downsampling and Type Conversion**

***frame = cv2.pyrDown(frame).astype(np.float32)***: This line downsamples the image to reduce its size, which helps in reducing the noise. Downsampling is performed using a Gaussian pyramid, which blurs and then subsamples the image. The conversion to ***float32*** is for maintaining precision in subsequent operations, especially as blurring and difference calculations can produce non-integer values.(Note that when output image for visualization, ***float32*** should be converted to ***uint8.***)

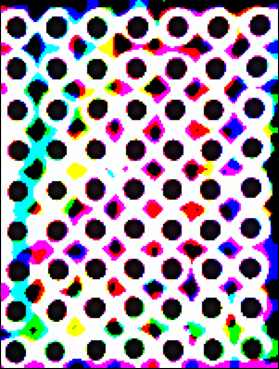
   

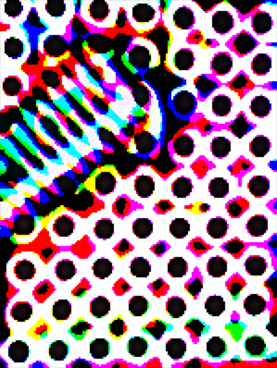
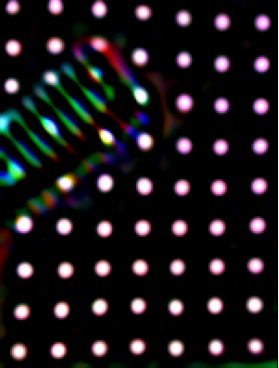
further downsampling(400\*300—200\*150)

* **Difference of Gaussians (DoG)**

***blur = cv2.GaussianBlur(frame, (25, 25), 0)*** and ***blur2 = cv2.GaussianBlur(frame, (15, 15), 0)***: These lines apply Gaussian blurring to the frame with two different kernel sizes. The larger kernel size (25x25) results in more blurring compared to the smaller one.

***diff = blur - blur2***: This line computes the difference between the two blurred images. This technique, known as Difference of Gaussians (DOG), is a common method for edge detection and feature enhancement. It highlights regions of the image where there are rapid intensity changes, which are typical of edges and contours. (Consider a white paper with black markers. With larger kernel size, the color (or say, brightness value compared to the blackmarker) of each pixel is averaged more. So the pixel values of the marker areas of ***blur*** are higher than those of the ***blur2, while*** the pixel values of the colored (bright) areas of ***blur*** are lower than those of the ***blur2***. Therefore, the pixels in ***diff*** yield negative value in the colored (bright) areas while positive in marker areas.)

Blur 25\*25 —— Blur 15\*15 —— Blur difference —— Diff amplification

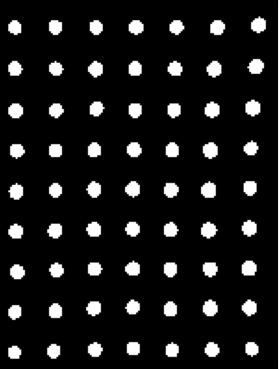
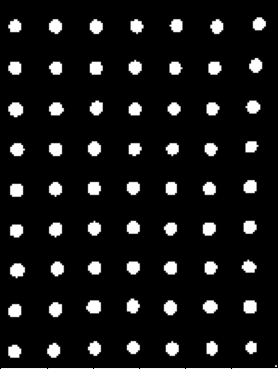
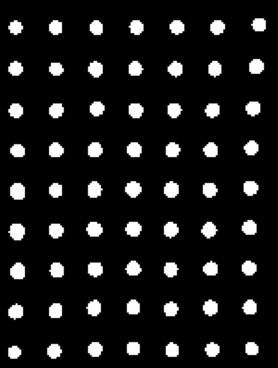
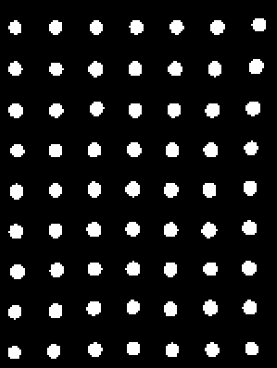
* **Thresholding and Mask Creation**

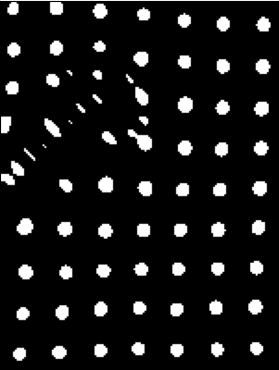
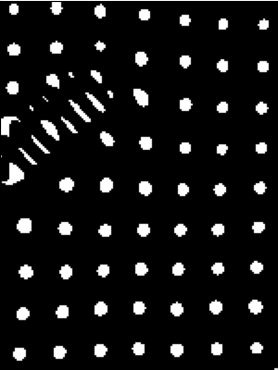
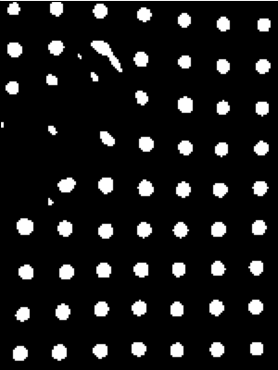
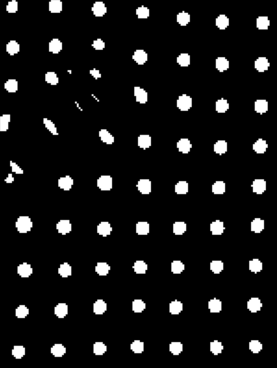
The code then amplifies the differences (***diff \*= 20***) and ensures the values stay within the 0-255 range.

***THRESHOLD = 120***: A threshold is set to identify significant differences (>6).

***mask\_b***, ***mask\_g***, and ***mask\_r*** are binary masks created for each color channel (blue, green, red), where pixels with differences above the threshold are marked.

***mask = (mask\_b \* mask\_g + mask\_b \* mask\_r + mask\_g \* mask\_r) > 0***: This line combines the binary masks from the three color channels. A pixel in the final mask is marked (***True***) if it is marked in at least two of the three channel-specific masks. This step ensures that only strong features (idealy, markers) common across multiple channels are selected.

mask\_b, mask\_g, mask\_r, mask

* **Final Mask Upsampling and Output**

The final binary mask (***mask***) is upsampled back to the original size of the frame and converted to ***uint8*** type.

The mask is returned with its values multiplied by 255 (***return mask \* 255***). This step converts the binary mask (with ***True/False*** values) into a grayscale mask where the marked pixels are white (255), and unmarked pixels are black (0).

In summary, ***mask\_marker*** uses blurring and difference calculations to detect markers in the image, and then applies thresholding to create a binary mask. This mask indicates the locations of the markers (where marker areas are white, and the rest is black), which can then be used by the ***inpaint*** and ***marker\_center*** functions to remove or alter these regions in the frame.

**def marker\_center**

***marker\_center*** aims to locate the center points of markers in a given frame for tracking or analysis purposes.

* **Initialization**

***areaThresh1*** and ***areaThresh2***: These variables define the range of acceptable areas for a contour to be considered a valid marker. The function will only consider contours whose areas are between ***areaThresh1*** (lower bound) and ***areaThresh2*** (upper bound).

***centers***: An empty list initialized to store the center points of detected markers.

* **Marker Mask Creation**

***mask = mask\_marker(frame, debug=debug)***: This line calls the ***mask\_marker*** function to create a mask that highlights the marker areas in the frame. The ***mask\_marker*** function returns a binary image where marker areas are white, and the rest is black.

* **Contour Detection**

***cv2.findContours(...)***: This function detects contours in the mask. Contours are the curves joining all continuous points along the boundary of white areas in the mask. The function returns a list of all detected contours.

***contours[0]***: OpenCV's ***findContours*** function returns a tuple where the first element is the list of contours. That's why ***contours[0]*** is used.

* **Marker Centers Calculation**

The loop iterates over each detected contour:

***cv2.boundingRect(contour)***: Calculates the bounding rectangle for each contour. The bounding rectangle is the smallest rectangle that encloses the contour.

***AreaCount = cv2.contourArea(contour)***: Computes the area of the contour.

The if condition checks if the contour's area is within the defined range and if the bounding rectangle is approximately square or regular in shape.

***cv2.moments(contour)***: This function calculates image moments, which are useful for finding the center of mass (centroid) of the contour.

***mc = [t["m10"] / t["m00"], t["m01"] / t["m00"]]***: These calculations derive the coordinates of the centroid from the moments.

***centers.append(mc)***: Adds the centroid coordinates to the centers list.

* **Return Value**

The function returns ***centers***, a list of coordinates representing the center points of each detected marker.

In essence, ***marker\_center*** effectively identifies and locates markers in a given image, providing valuable data for tracking or analyzing specific features within the frame.

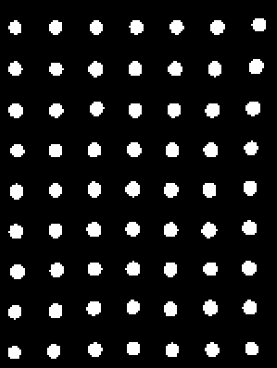
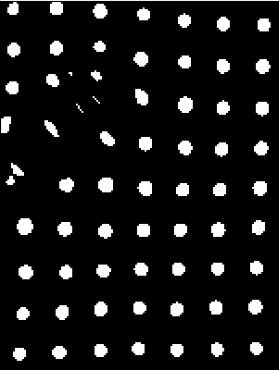
**def inpaint**

The ***inpaint*** function is used for removing markers or small unwanted objects from an image. It uses a mask to identify the regions to be inpainted (reconstructed).

* **Mask Creation**

Calls ***mask\_marker(frame)*** to create a mask. This mask identifies the pixels corresponding to the markers in the image.

The mask is binary, where the marker pixels are set to a high value (white) and the rest are low value (black).

mask\_marker(get\_processed\_frame(frame))

* **Inpainting**

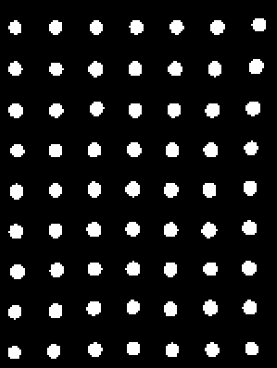
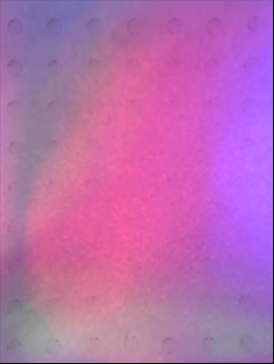
***cv2.inpaint(frame, mask, 7, cv2.INPAINT\_TELEA)*** is used for the inpainting process.

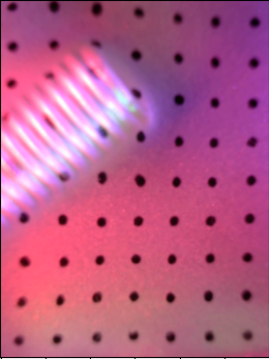
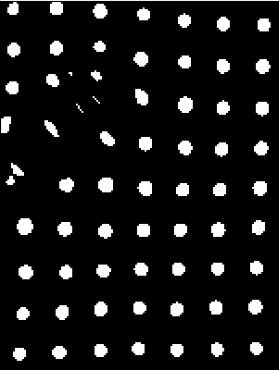
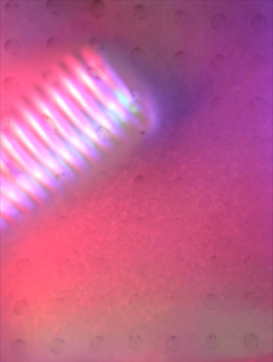
***frame*** is the original image from which markers are to be removed.

***mask*** identifies the regions in frame to be inpainted.

***7*** is the inpainting radius. It specifies the neighborhood of each point in the mask to be considered for inpainting.

***cv2.INPAINT\_TELEA*** refers to the inpainting algorithm used. Alexandru Telea’s method is an advanced and efficient inpainting algorithm.

frame, mask, inpaint result

* **Return**

Returns ***frame\_marker\_removed***, which is the inpainted image where markers have been removed and the regions have been reconstructed based on surrounding pixels.

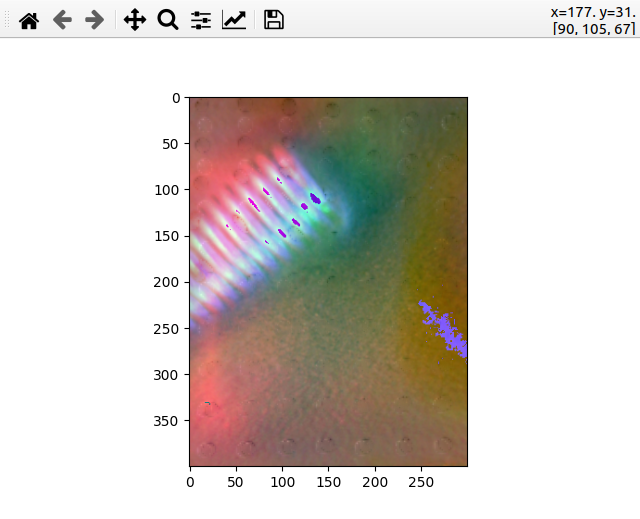
This function is particularly useful in computer vision tasks where markers are used for tracking or calibration but are not desired in the final processed image. The inpainting helps to restore the underlying image content.

**def difference (Only contact case is studied here)**

The ***difference*** function calculates the difference between two frames (images) and processes this difference to highlight the areas of change.

* **Calculate Difference**

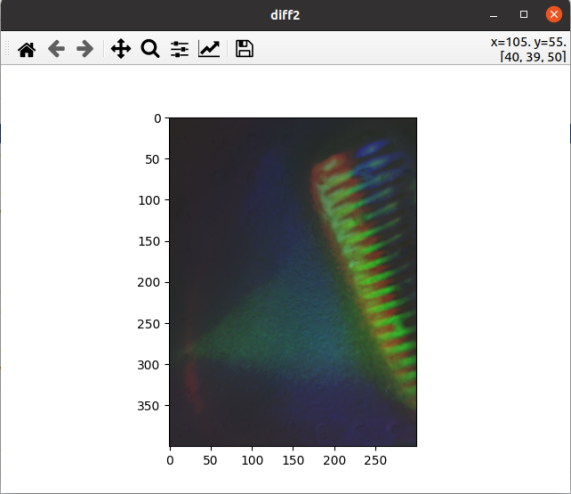
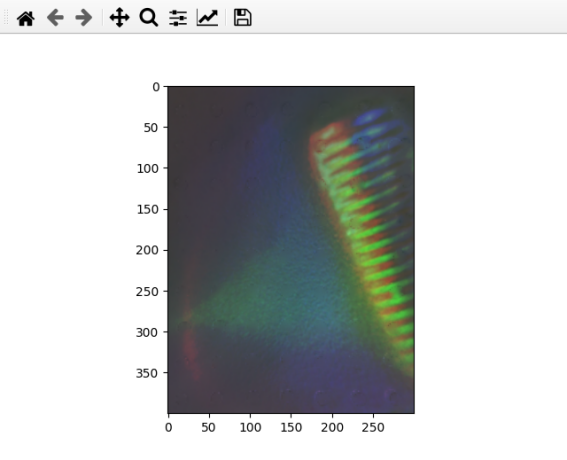
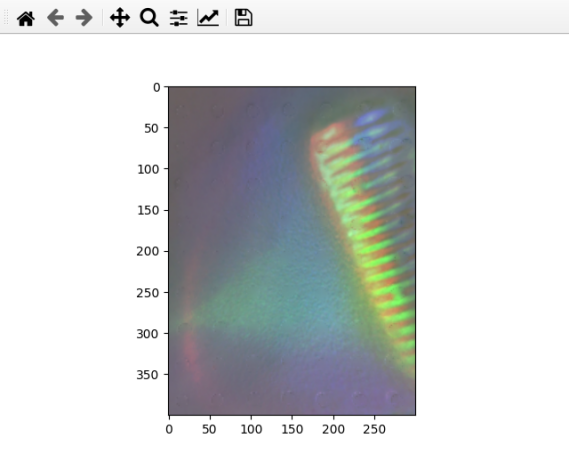
***(frame \* 1.0 - frame0) / 255.0 + normalized\_lb***: Computes the normalized difference between the current frame (***frame***) and the reference frame (***frame0***). The result is a float image where pixel values represent the intensity of change, normalized between -0.5 and 1.5.(Multiplying by 1.0 ensures that the operation is carried out in floating-point arithmetic instead of integer arithmetic, which is important to avoid overflow or underflow errors when the pixel values are subtracted.)



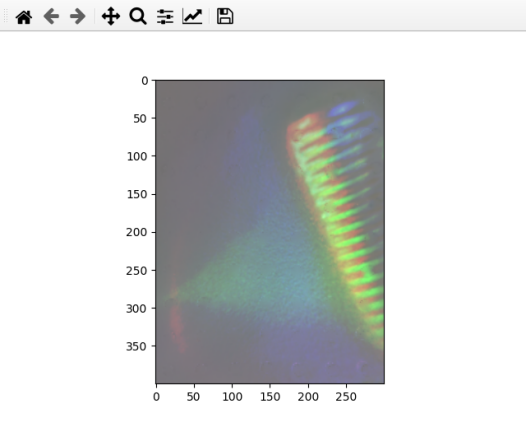
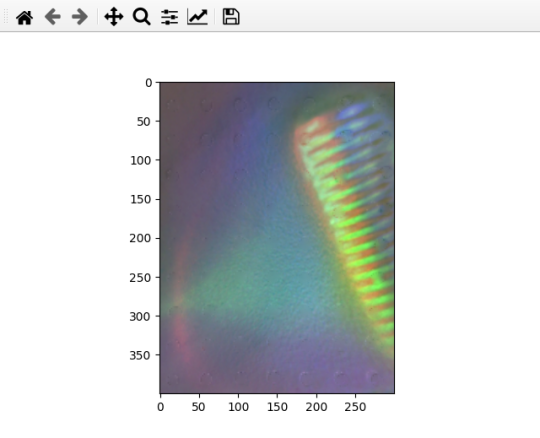
diff

* **Reduce noncontact differences**

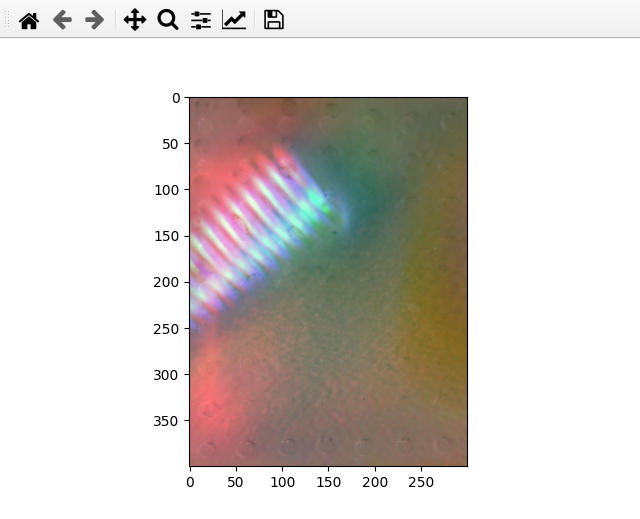
***diff[diff < normalized\_lb] = (diff[diff < normalized\_lb] - normalized\_lb) \* SF + normalized\_lb***: Reducing the negative differences (***diff[diff < 0.5],*** or say the pixels where frame - frame0 are negative) by adjusting their intensity with scale factor = 0.7. The negative area is usually not contact area. (By normalizing the pixels value to [-0.5, 1.5], we essentiall enhance those parts with positive difference by downscale the negative-difference part within a scale factor of 0.7 while efficiently fitting the results into [0,1])

normalized\_lb, SF = 0.2, 0.3, 0.5

normalized\_lb = 0.5 , SF = 0.2 normalized\_lb = 0.5 , SF = 0.7

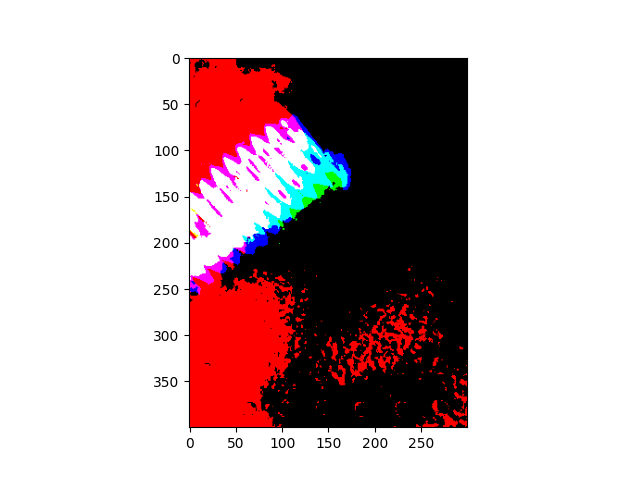


diff\_reduce

* **Convert to uint8, Apply Threshold, Gray Conversion and Thresholding**

***diff\_uint8 = (diff \* 255).astype(np.uint8)***: The normalized difference is scaled back to a range of 0-255 and converted to an 8-bit image.

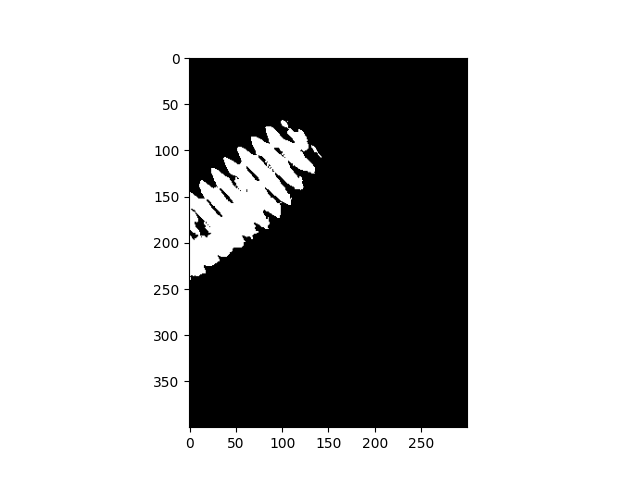
***diff\_uint8[diff\_uint8 > 140] = 255; diff\_uint8[diff\_uint8 <= 140] = 0***: A threshold is applied to create a binary image where the significant changes are marked as white (255) and others as black (0).



diff\_threshold

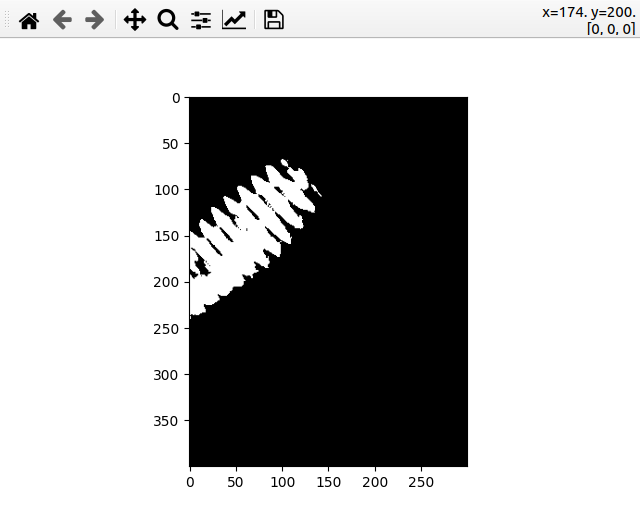
***Here starts the modified part, In order to convert the 3-channel binary image into grayscale image. The following procedures are conducted***:

1. Split the image into its channel components.
2. Calculate the dominance for each color.
3. Determine non-dominant areas (where no single channel is dominant).
4. Convert boolean mask to uint8.
5. Apply the mask to the image to retain only non-dominant areas.



diff\_gray

***\_, diff\_thresh = cv2.threshold(diff\_gray, 50, 255, cv2.THRESH\_BINARY)***: Further thresholding is applied to refine the binary image.

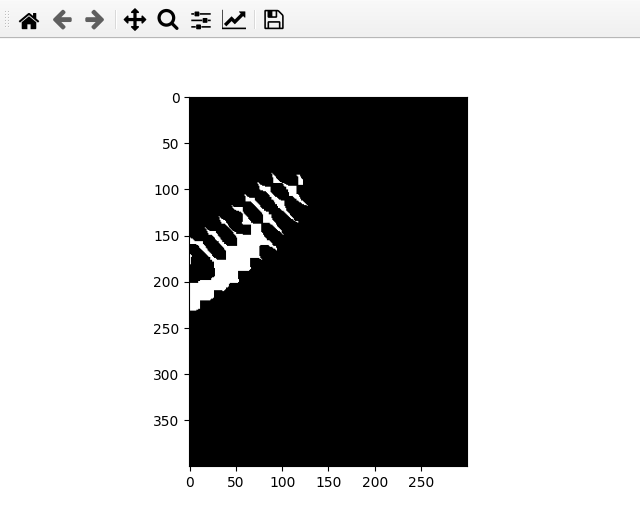
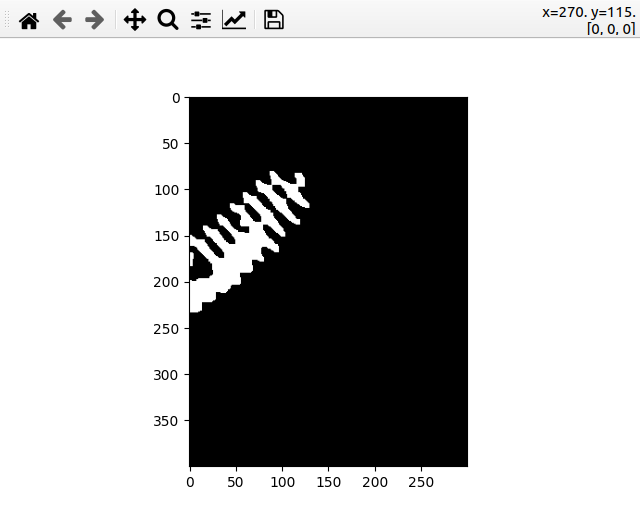


diff\_graythresh

* **Erosion and Dilation**

***diff\_thresh\_erode = cv2.erode(diff\_thresh, np.ones((5, 5), np.uint8), iterations=2):*** Erosion is applied to reduce noise and small irrelevant details.

***diff\_thresh\_dilate = cv2.dilate(diff\_thresh\_erode, np.ones((5, 5), np.uint8), iterations=1)***: Dilation is applied to restore the eroded main features.

diff\_thresh\_erode diff\_thresh\_dilate

The final processed image ***diff\_thresh\_dilate*** shows the areas of significant change between the two frames. This image can be used for further analysis, such as contour detection, to identify and analyze the regions of change.

**def get\_all\_contour**

This function is designed to find and process contours in an image. It takes a pre-processed image ***diff\_thresh\_dilate***, the original ***frame***, and a ***debug*** flag.

* **Finding Contours**

***cv2.findContours*** is used to find contours in the thresholded image ***diff\_thresh\_dilate***.

The method ***cv2.RETR\_EXTERNAL*** retrieves only extreme outer contours, which is useful for this application.The ***contours*** variable is a list where each element is a numpy array representing a contour with its (x, y) coordinates.

***cv2.CHAIN\_APPROX\_SIMPLE*** compresses horizontal, vertical, and diagonal segments and leaves only their end points, optimizing memory usage.

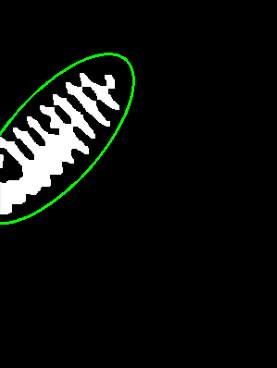
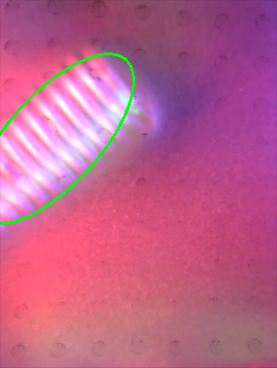


Contour

* **Merging and Ellipse Fitting**

If contours are found, they are merged into one array using ***np.concatenate***. ***merged\_contour*** thus contains all the points of all contours.

***ellipse = cv2.fitEllipse(merged\_contour)***: An ellipse is fitted to these merged contours using ***cv2.fitEllipse***. The returned structure contains the center of the ellipse, the major and minor axis lengths, and the rotation angle.This step assumes the contours form an elliptical shape, which can be modified according to the application's needs.

ellipse drawn with a scale factor of 1.5

* **Debug Visualization**

If ***debug*** is True, the function visualizes the fitted ellipse on both the original frame and the contour image for verification.

This visualization helps in understanding how well the ellipse fits the identified contours.

* **Error Handling**

The function uses a ***try-except*** block to handle any errors that might occur during contour merging and ellipse fitting (e.g., no contours are found in the first iteration when frame = frame0).

This function is for understanding the shape and position of objects in a given image.The function’s robustness is enhanced by its error handling and debug visualization capabilities.

**def regress\_line**

The ***regress\_line*** function is designed to fit a straight line to a set of points in an image using linear regression. This is typically useful for analyzing the orientation and position of linear features in images.

* **Line Frame Preparation**

***line\_frame = frame***: Any drawing (like lines or text) will be directly on the original frame, not on a copy.

* **Line Fitting**

***vx, vy, x, y = cv2.fitLine(all\_points, cv2.DIST\_L2, 0, 0.01, 0.01)***: This line uses OpenCV's ***fitLine*** function to fit a line to the set of points (***contours***). The line is fitted using the least-squares method (***cv2.DIST\_L2***), and the function returns the parameters of the line (***vx, vy, x, y***) where vx and vy are normalized vector coordinates that are collinear to the line, and (***x, y***) is a point on the line.

* **Calculating Slope and Angle**

***slope = vy / vx***: Calculates the slope of the line.

***angle = np.degrees(np.arctan(slope))***: Converts the slope to an angle in degrees.

* **Determining Line Endpoints for Drawing**

***lefty = int((-x \* vy / vx) + y)***: Calculates the y-coordinate of the line at the left edge of the image (***x=0***). The term ***-x \* vy / vx*** represents the vertical distance from the reference point (***x, y***) to the left edge when following the slope ***vy / vx***. Adding ***y*** offsets this distance to the actual coordinate system of the image.

***righty = int(((line\_frame.shape[1] - x) \* vy / vx) + y)***: Calculates the y-coordinate of the line at the right edge of the image (***x = line\_frame.shape[1]***, which is the width of the image). The term (***line\_frame.shape[1] - x) \* vy / vx*** represents the vertical distance from the reference point ***(x, y)*** to the right edge. Adding y offsets this distance to the actual coordinate system of the image.

***pt1 = (line\_frame.shape[1] - 1, righty):***Specify the left endpoint coordinates.

***pt2 = (0, lefty):*** Specify the right endpoint coordinates.

* **Calculating Midpoint**

***midx*** and ***midy*** represent the midpoint of the line segment being drawn. This is calculated as the average of the x and y coordinates of ***pt1*** and ***pt2***.

* **Drawing on the Frame**

The function draws the line, a circle at its midpoint, and text showing the midpoint coordinates and angle on ***line\_frame***.



* **Debug Display**

If ***debug*** is True, the modified frame with the drawn line is displayed in a window titled "Line".

* **Return Values**

The function returns the negative of the calculated angle and the midpoint coordinates. The reason for returning ***-1 \* angle*** is specific to how angles are interpreted in the context of application.

This function is to understand the orientation or alignment of the contact area.

**def get\_convex\_hull\_area**

The ***get\_convex\_hull\_area*** function performs several key operations to identify and analyze the convex hull of the contact area in the tactile sensor image.

* **Find Contours**

The function starts by finding contours in the ***diff\_thresh\_dilate*** image using ***cv2.findContours***. This step isolates the edges or boundaries of the contact area. It uses the ***RETR\_EXTERNAL*** mode to retrieve only the extreme outer contours and ***CHAIN\_APPROX\_SIMPLE*** to compress horizontal, vertical, and diagonal segments, leaving only their end points.

* **Initialize Variables**

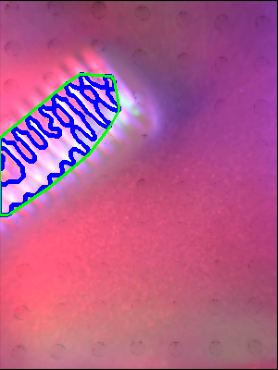
It prepares an empty image ***img\_hull*** for visualizing the convex hull. It initializes variables like ***hull\_area***, ***slope***, ***center***, and ***hull\_mask***. The ***hull\_mask*** is a blank image with the same shape as ***diff\_thresh\_dilate***, used to create a mask of the convex hull.

* **Loop Over Contours**

***contourpoints = np.vstack([cnt for cnt in contours])***: Uses np.vstack (vertical stack) to concatenate a sequence of arrays along the first axis (vertically).This array will be used to calculate the convex hull.

* **Convex Hull Calculation**

Using ***cv2.convexHull***, the function calculates the convex hull of the points in ***contourpoints***, which is the smallest convex shape that encloses all the points. The ***cv2.contourArea*** function then calculates the area of this convex hull.



blue line depits the contour, green line depits the hull

* **Line Regression (See def regress\_line)**

The function calls ***regress\_line***, passing the ***hull*** points and ***img\_hull*** to calculate the line regression of the convex hull. This function returns the slope and the center coordinates of the regression line, which helps in understanding the orientation and central position of the contact area.



regressline

* **Create Hull Mask**

cv2.fillPoly(hull\_mask, pts=[hullpoints], color=(255, 255, 255)): Fills the area of the convex hull in the ***hull\_mask*** image, creating a mask that can be used to isolate the convex hull from the rest of the image.



Hullmask

* **Debugging Visualization**

If debugging is enabled, the function uses ***cv2.putText*** and ***cv2.imshow*** to display the convex hull and related information, like the area of the hull, on the ***img\_hull*** image.

* **Return Values**

Finally, the function returns the area of the convex hull, the mask image, the slope of the regression line, and the center coordinates of the hull.

This process allows the system to quantify and visualize the area of contact and its orientation, which can be used to understand the interaction between the sensor and the object being touched.

**def draw\_flow**

The draw\_flow function is designed to visualize the flow of markers on the tactile sensor's surface. Let's delve into the function's details:

* **Inputs**

***frame***: The original image frame from the tactile sensor.

***flow***: A tuple containing flow data (Ox, Oy, Cx, Cy, Occupied), representing the original (Ox, Oy) and current (Cx, Cy) positions of markers, and a status array (***Occupied***).

* **Calculating arrow vectors**

Arrows are drawn to represent the movement of each marker from its original position (Ox, Oy) to its current position (Cx, Cy).

The ***K*** value is used as a scaling factor for the arrow length. A value of 1 implies that the arrow length is directly proportional to the displacement of the marker.

* **Color Coding**

The ***color*** variable is set to yellow (0, 255, 255) for most markers.

If the ***Occupied*** status for a marker is less than or equal to -1, the arrow is colored white (255, 255, 255). This could indicate markers that are no longer active or have a specific status.

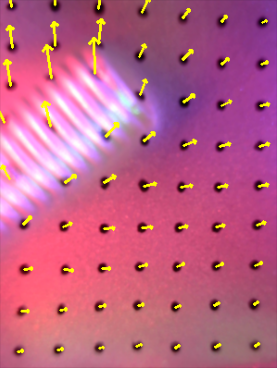
* **Drawing Arrows**

***cv2.arrowedLine*** is used to draw arrows on ***drawn\_frame***.

***pt1*** and ***pt2*** are the starting and ending points of each arrow, showing the marker's movement.

* **Output**

The function returns ***drawn\_frame***, which is the original frame augmented with visual representations of marker movements.



flow\_drawing

**def draw\_flow\_mask**

***draw\_flow\_mask*** focuses on visualizing marker movements within a specific masked area of a tactile sensor's surface. Unlike ***draw\_flow***, which visualizes flow across the entire frame, ***draw\_flow\_mask*** targets a localized region, providing insights into finer details of contact and pressure points.

* **Frame Preparation**

The function starts by copying the input frame to ***drawn\_frame*** for overlaying visualization without altering the original frame.

The mask is dilated to enhance the area of interest.

* **Combining Mask and Frame**

Applies the mask to ***drawn\_frame*** using ***cv2.bitwise\_and***, which isolates the flow visualization to the area defined by the mask.

* **Flow Visualization Within Masked Area**

It initializes variables ***change*** (to store total displacement) and ***counter*** (to count markers within the mask).

Iterates each marker's original (***Ox***, ***Oy***) and current (***Cx***, ***Cy***) coordinates in ***flow***.

For each marker, checks if it falls within the masked area (where mask value is 255).

If inside the masked area, calculates displacement (***dx***, ***dy***) and adds to change.

Draws an arrow from the marker's original to the current position, color-coded based on the ***Occupied*** status of each marker.

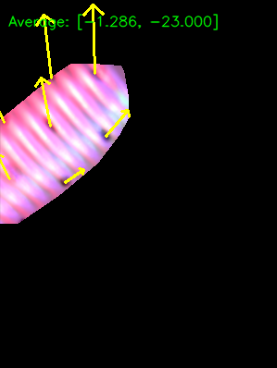
* **Average Flow Calculation**

If at least one marker falls within the masked area, calculates the average flow change (***change[0]*** for x-direction, ***change[1]*** for y-direction).

Displays the average flow change on the frame using ***cv2.putText***.

* **Output**

The function returns a frame (***drawn\_frame\_and***) that visually represents the average marker flow within the specified masked area.



flow\_drawing\_withmask