**lab\_live**

**1. Initialization and Camera Setup**

* 1. **Publishing 3D coordinates**
  2. **Camera Detection and configuration**

The script begins by detecting the UVC camera connected to the system, extracting its device number for use. Initializes the camera with specific width and height settings and creates a ***cv2.VideoCapture*** object for frame capture.

1. **Marker Tracking Initialization**
   1. **Configuration Setup**

Calls ***setting.init()*** to configure global settings related to marker tracking.

**2.2. Marker Tracking Instance**

Creates an instance of the ***Matching*** class from the ***find\_marker*** library. This class is likely responsible for tracking the movement of markers on the tactile sensor.

**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.

**1. Initialization and Camera Setup**

* 1. **Publishing 3D coordinates**

**def publish\_image(real\_x, real\_y, real\_z)**

***publish\_image*** is a function designed to publish a 3D point (with coordinates ***real\_x***, ***real\_y***, and ***real\_z***) to a ROS topic. This function is integral for communicating spatial data within a robotic system.

* **Point Initialization**

***detect\_result = Point()***：A instance of the Point class is created and assigned to the variable ***detect\_result***. ***Point*** is likely imported from the ***geometry\_msgs.msg*** package, which is commonly used in ROS (Robot Operating System) for representing a point in a 3D space.

* **ROS Rate Configuration**

***rate = rospy.Rate(100)***：Sets up a ROS rate object. This is used to control the frequency of the loop execution, here specified as 100 Hz.

* **Header Configuration**

***header = Header(stamp=rospy.Time.now()):*** Initializes a ***Header*** object with the current time as the timestamp.(the ***stamp*** attribute is used to record the exact time when the message was created or the data was recorded. This is crucial in time-sensitive applications, such as those involving real-time robotic operations, where knowing the precise time of data capture is necessary for synchronization and temporal analysis.)

***header.frame\_id = 'map':*** Sets the frame ID of the header to 'map', indicating the coordinate frame of reference.(By setting ***frame\_id*** to 'map', it indicates that the point coordinates being published are in the 'map' coordinate frame. This helps other nodes or components in the ROS system understand the context and reference frame of the published data, ensuring that the spatial information is correctly interpreted and utilized.)

* **Assigning Point Coordinates**

Assign the values of real\_x, real\_y, and real\_z to the respective coordinates of the Point object, ***detect\_result***.

* **Publishing the Point**

***pub.publish(detect\_result):*** Publishes the ***Point*** object to a ROS topic. The ***pub*** variable is assumed to be a ROS publisher initialized elsewhere in the code.

* **ROS Node and Publisher Initialization**

***rospy.init\_node('tactile\_senser')***: Initializes a new ROS node named '***tactile\_senser***'. This is the first step in any ROS program and registers the script as a node in the ROS network, allowing it to communicate with other nodes.

***pub = rospy.Publisher("/tactile", Point, queue\_size=10)***: Creates a ROS publisher that will publish messages of type ***Point*** on the "***/tactile***" topic. The ***queue\_size=10*** argument specifies the size of the outgoing message queue, keeping the last 10 messages before discarding older ones.

* 1. **Camera Detection and Configuration**
* **Command Execution for Camera List**

***output = subprocess.check\_output(...)***: Executes the command ***v4l2-ctl --list-devices*** using the subprocess module. This command lists all video devices connected to the system. The ***universal\_newlines=True*** argument ensures that the output is in text format, not bytes.

屏幕上有字

描述已自动生成

* **Camera Identification**

***camera\_sections = output.strip().split('\n\n')***: Splits the command output into separate sections for each camera device. (The ***strip()*** method is used to remove any leading and trailing whitespace characters (like spaces, tabs, and newline characters) from the ***output*** string. This is often used to clean up the text before further processing. The ***split()*** method is then used to divide the output string into a list of substrings. Here, the method splits the string at every occurrence of ***\n\n***, which is a double newline character.)

The subsequent for loop searches through these sections for a section containing 'UVC', indicating a UVC (USB Video Class) camera. If found, ***target\_camera*** is set to this section.

文本

中度可信度描述已自动生成

* **Camera Device Number Extraction**

The ***if*** statement checks if a ***UVC*** camera was found. If yes, it extracts the device numbers associated with the UVC camera.

***device\_lines = target\_camera.strip().split('\n')***: Splits ***target\_camera*** into lines, resulting in:

***device\_lines = ["UVC Camera (046d:0825):", "/dev/video1", "/dev/video2"]***

***device\_numbers = [line.split()[-1] for line in device\_lines[1:]]***: Extracts the device numbers from device\_lines, skipping the first line. The comprehension splits each line and takes the last element:



* **Camera Initialization**

***cam = cv2.VideoCapture(device\_numbers[0])***: Initializes a video capture object ***cam*** using the first device number (/dev/video1 in this case) of the UVC camera.

The camera's frame width and height are set to 800x600 using

***cam.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 800)*** and

***cam.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 600)***.

This configures the camera to capture video frames of 800x600 pixels.

1. **Marker Tracking Initialization**
   1. **Configuration Setup**

***setting.init()***: Initializes settings for marker tracking. In which:  
N\_, M\_: the row and column of the marker array

x0\_, y0\_: the coordinate of upper-left marker (original size before deformation)

dx\_, dy\_: the horizontal and vertical interval between adjacent markers (original size before deformation)

fps\_: the desired frame per second, the algorithm will find the optimal solution in 1/fps seconds

**2.2. Marker Tracking Instance**

***m = find\_marker.Matching(...)***: Creates an instance of a Matching class (presumably from the ***find\_marker*** module) with the parameters defined in the ***setting*** module.

**3. Main Processing Loop**

**3.1. Frame Preprocessing**

* **Initialization**

***count = 0***: A variable to track the number of iterations or frames processed.

***frame0 = None***: A variable to store the initial frame, which might be used as a reference or baseline in subsequent processing.

* **Iterating Over Sorted Image Files**

***for image\_file in sorted(image\_files)***: This loop iterates through all the image files in the ***image\_files*** list. The ***sorted()*** function is used to process these files in alphabetical or numerical order, which is crucial for maintaining chronological order in image sequences.

* **Loading and Verifying Images**

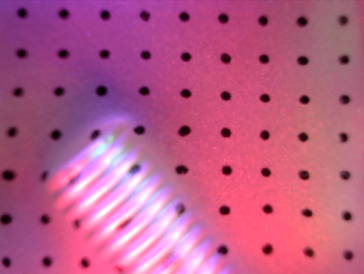
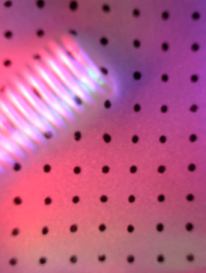
***frame = cv2.imread(os.path.join(image\_folder, image\_file))***: This line loads an image into the variable ***frame*** using OpenCV's ***imread*** function. The os.path.join is used to construct the full path to the image file by combining the directory path (***image\_folder***) and the filename (***image\_file***). Below shows an example:

The subsequent ***if*** statement checks if the image was successfully loaded (***frame is None***). If not, it prints an error message and uses ***continue*** to skip to the next iteration.

* **Processing the Image Frame**

***frame = A\_utility\_test.get\_processed\_frame(frame)***:

The loaded image frame is processed using a function ***get\_processed\_frame*** from the module ***A\_utility\_test.*** An example of the the processed output is shown below:

original —— A\_utility\_test.get\_processed\_frame(frame)

**3.2. Contact Area Calculation**

* **Remove the markers**

if count == 1::

This conditional block checks if it's the first iteration (count == 1). If so, it performs initial setup for contact area detection.

frame0 = copy.deepcopy(frame): Creates a deep copy of the first frame. This is crucial to avoid modifications to the original frame, which is used as a reference for detecting changes in subsequent frames.

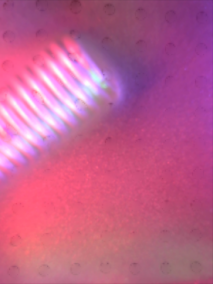
frame0\_final = A\_utility\_test.inpaint(frame0): Applies an inpainting operation to the copied frame. Inpainting is typically used to fill in missing or corrupted areas of an image.



A\_utility\_test.inpaint(frame0)

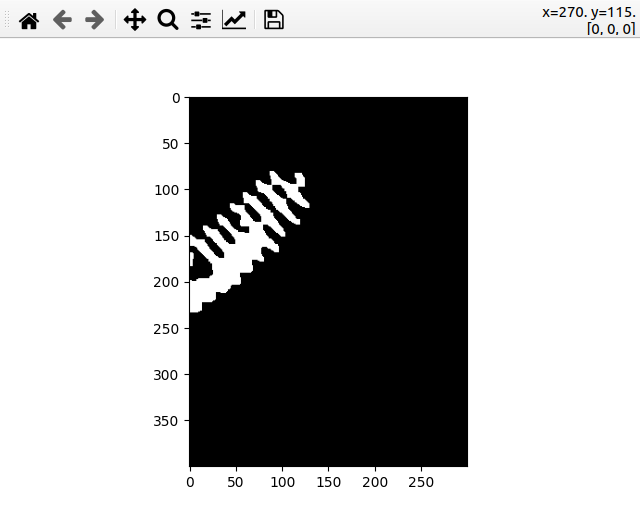
* **Frame Finalization and Contact Area Detection**

frame\_final = A\_utility\_test.inpaint(frame): Same inpainting operation is applied to the current frame.



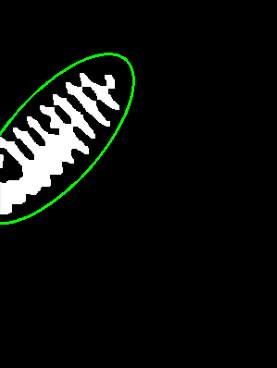
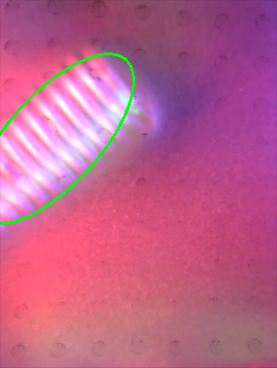
A\_utility\_test.inpaint(frame)

contact\_area\_dilated = A\_utility\_test.difference(frame\_final, frame0\_final, debug=False): Calculates the difference between the current frame and the reference frame to identify contact areas for further analysis.



A\_utility\_test.difference(frame\_final, frame0\_final)

contours = A\_utility\_test.get\_all\_contour(contact\_area\_dilated, frame, debug=False): Extracts contours from the detected contact area. Contours are useful in object detection and shape analysis.

**3.3. Convex Hull and Area Calculation**

hull\_area, hull\_mask, slope, center = A\_utility\_test.get\_convex\_hull\_area(...): This line computes the convex hull of the contact area, along with its area, a mask of the hull, the slope, and the center point. The convex hull is the smallest convex shape that encloses the contact area, and its properties like area and slope can be crucial in understanding the nature of the contact.

