AE 482 FINAL PROJECT: DRAWING WITH THE UR3 ROBOT

Jadyn Chowdhury[1]

Jadynnc2 | Partner - Chris Sun | TA - Lukas Zscherpel | Tuesday 3PM Section | 10/08/24

AE 482: Introduction to Robotics, Urbana, IL, 61820, United States of America

This lab explores the intersection of computer vision and robotics, enabling the UR3 robot to draw input images with precision. Using OpenCV and ROS, the system produced detailed outputs, winning the best-drawn image competition, and showcasing robotics' potential in automation and practical applications.

I. Nomenclature

ROS	=	Robot Operating System used to control the UR3 robot via Python scripts.
$UR3$	=	A collaborative robotic arm used in the lab.
FK	=	Forward Kinematics-calculating position and orientation of end-effector, given joint angles.
T	=	Transformation Matrix-4x4 matrix encoding both rotation and translation.
S	=	Screw Axis-six vectors describe rotation and translation of each joint, relative to base frame.
θ	=	Joint Angles- angles for each of the six joints in the UR3 robot.
End-Effector	=	The tool attached to the end of the robotic arm, equipped with a suction device.
$T(\theta)$	=	Homogeneous Transformation Matrix- encodes both the orientation and position of end-effector.
IK	=	Inverse Kinematics - joint angles to position end-effector at a specified location and orientation.
DOF	=	Degrees of Freedom- six independent movements (or axes) available to the UR3 robot.
Yaw	=	Rotation around z-axis, representing the orientation of the end-effector in the horizontal plane.
$Elbow - Up$ $Configuration$	=	IK solution where robot's 'elbow' is oriented in an upward position to avoid obstacles.
$World\ Frame$	=	(w) Coordinate system where robot's base is used as the origin for end-effector positions.
$Base\ Frame$	=	(0) Local coordinate system fixed at base, to which joint angles and transformations are relative.

II. Introduction

The purpose of this project is to explore the practical applications of robotic systems in precision tasks such as drawing. Using the UR3 robot and ROS, we combine advanced robotics and image processing techniques to enable automated contour detection and replication. This lab demonstrates how robots can be programmed to interpret and replicate visual data, paving the way for innovative applications in fields such as manufacturing, art, and automation.

The motivation behind this project is to gain hands-on experience with forward and inverse kinematics, transformation matrices, and real-world applications of image processing. By using OpenCV for image analysis and ROS for robot control, the lab integrates software and hardware solutions to solve a complex task: transforming an input image into precise, physical movements for the robot to recreate that image on paper.

This project has real-world implications in domains where precision and automation are critical. For instance, contour-based tasks such as engraving, painting, and surgical procedures can benefit from similar methodologies. By controlling the UR3 robot to draw, we demonstrate its potential to transition from abstract data interpretation to tangible outputs, showcasing the versatility and accuracy of modern robotic systems.

[1]Aerospace Engineering and Computer Science, Grainger School of Engineering at UIUC

III. Method

The development of the robotic drawing pipeline required integrating image processing, robot kinematics, and execution logic to ensure precise replication of an input image. The final implementation was the result of iterative improvements, guided by the need to handle various levels of image complexity while maintaining scalability and efficiency.

A. Contour Detection and Optimization

After the image was resized (see section E), it underwent a series of preprocessing steps to extract meaningful contours. The `find_contours` function encapsulated these steps, starting with grayscale conversion and Gaussian blurring to reduce noise. Adaptive thresholding was then applied to segment the image into regions of interest,

```
grey_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(grey_img, (5, 5), 0)
thresh = cv2.adaptiveThreshold(blur, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 11, 2)
```

Morphological operations, including opening and closing, further refined the segmented image by removing small noise and closing gaps in contours. The edges were then detected using the Canny algorithm,

```
edges = cv2.Canny(morph, 50, 150, apertureSize=3)
contours, _ = cv2.findContours(edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
```

While these steps extracted initial contours, they often contained noise or redundant points. The `find_contours` function applied additional processing, such as filtering contours by area, interpolating points for smoother curves, and merging close contours to prevent overlapping outlines. These functions were written for this specific use case, for example for filling in gaps between contour renditions,

```
def interpolate_contour(contour, step=2):
    """Interpolate additional points along a contour for higher detail."""
    interpolated = []
    for i in range(len(contour)):
        start_point = contour[i]
        end_point = contour[(i + 1) % len(contour)]
        distance = np.linalg.norm(np.array(end_point) - np.array(start_point))
        num_steps = max(int(distance / step), 1)
        for j in range(num_steps):
            interpolated_point = (
                int(start_point[0] + (end_point[0] - start_point[0])
                * j / num_steps),
                int(start_point[1] + (end_point[1] - start_point[1])
                * j / num_steps)
            )
            interpolated.append(interpolated_point)
    return interpolated
```

Alongside this function, the other functions designed and implemented were:

- `smooth_path`: Apply a moving average to smooth the path points.
- `merge_close_contours`: Merge contours that are within a certain proximity to prevent double outlines.
- `adaptive_sample`: Adaptively sample points based on curvature. Used vector cosine analysis to identify linear contours.
- `group_straight_lines`: Group consecutive points that form a straight line. Allowed for just two points to be drawn rather than multiple in a line, greatly increasing speed of drawing.

These optimizations ensured that the detected contours were both meaningful and manageable for the robot to replicate.

B. Transforming Image Coordinates to World Coordinates

The UR3 robot operates in a three-dimensional workspace, making it essential to map pixel-based image coordinates to its world frame. The `IMG2W` function performed this transformation, scaling the image coordinates based on the physical dimensions of the drawing sheet and applying offsets to align the image with the robot's starting position.

```
def IMG2W(row, col):
    x = x_offset + col * scale
    y = y_offset + row * scale
    return x, y
```

This transformation allowed the robot to draw images of varying sizes and resolutions while ensuring the contours fit within the defined workspace. Parameters such as `x_offset`, `y_offset`, and `scale` were adjusted dynamically based on the input image's dimensions and the drawing sheet's size.

C. Path Planning for Efficient Drawing

After transforming the contours into world coordinates, the next step was to determine the most efficient path for the robot to follow. This involved sorting the contours to minimize travel distance between them, implemented in the `sort_contours_by_proximity` function,

```
def sort_contours_by_proximity(contours):
    sorted_contours = []
    current_position = (x_offset, y_offset)
    while contours:
        closest_contour = min(
            contours, key=lambda c:
np.linalg.norm(np.array(current_position) - np.array(IMG2W(c[0][1],
c[0][0])))
        )
        sorted_contours.append(closest_contour)
        contours.remove(closest_contour)
        current_position = IMG2W(closest_contour[-1][1],
closest_contour[-1][0])
    return sorted_contours
```

This optimization reduced unnecessary movements, improving both the speed and accuracy of the drawing.

D. Robot Movements: Lifting and Drawing
The robot's movements were managed using two distinct commands: `MoveJ` for curves and `MoveL` for precise linear movements. These were implemented in helper functions such as `lift_pen` and `draw_path`.

The `lift_pen` function for example ensured the pen was raised to a safe height before moving to a new starting position,

```
def lift_pen(pub_cmd, loop_rate, target_point, height, vel, accel):
    x, y, _ = target_point
    lifted_point = np.array([x, y, height])
    move_arm(pub_cmd, loop_rate, lifted_point.tolist(), vel, accel,
'L')
```

The `draw_path` function guided the pen along the contours, transforming each point into joint angles using inverse kinematics,

```
def draw_path(pub_command, loop_rate, path, vel, accel):
    for point in path:
        xw, yw = IMG2W(point[1], point[0])
        drawing_point = lab_invk(xw, yw, height_sheet, 0)
        move_arm(pub_command, loop_rate, drawing_point, vel, accel,
'L')
```

These functions allowed the robot to seamlessly transition between different segments of the drawing, maintaining precision throughout. The `draw_image` function conducted these movements using a try-catch structure to skip points that may be invalid and avoid the robot stopping due to an error.

E. Integrating the Pipeline
The final integration of all components occurred in the `main` function. This function initialized the ROS environment, loaded the image, processed it to extract and optimize contours, and executed the drawing (simplified for readability),

```
def main():
    image = cv2.imread('./images/avengers7.jpg')
    contours = find_contours(image)
    sorted_contours = sort_contours_by_proximity(contours)
    draw_image(sorted_contours, pub_command, loop_rate, vel, accel)
```

By combining the preprocessing, coordinate transformation, and movement logic, this pipeline demonstrated the versatility and efficiency of the system. The results were evaluated using images of varying complexity, showcasing the scalability enabled by parameters.

F. Hyperparameters
The project relied on several hyperparameters to balance precision, efficiency, and scalability.

Hyperparameter	**Purpose**	**Impact**	**Why It's Important**
max_dimension	Defines the largest dimension (width or height) of the image after scaling.	Higher values retain finer details; lower values simplify contours for faster processing.	Controls scalability and allows the system to adapt to images of varying complexity.

merge_threshold	Distance threshold for merging close contours.	Larger values combine more contours; smaller values preserve individual contours.	Prevents redundant lines and ensures smooth, unified drawing paths.
step (in interpolation)	Defines the spacing between interpolated points along contours.	Smaller steps create smoother curves; larger steps reduce detail but improve processing speed.	Balances the need for detailed drawing with computational efficiency.
window_size	Window size for smoothing contours using a moving average.	Larger values smooth paths more aggressively; smaller values retain finer details.	Ensures contours are clean and free of noise, improving the robot's movement precision.
angle_threshold	Angle threshold for grouping points into straight lines.	Lower thresholds group more points; higher thresholds preserve individual points.	Helps optimize drawing paths by reducing unnecessary movements.
scale	Scales image coordinates to fit within the robot’s workspace.	Adjusted dynamically based on max_dimension and the size of the drawing surface.	Ensures that the image fits on the physical drawing area while maintaining the correct proportions.
vel (velocity)	Defines the speed of the robot arm during movements.	Higher values increase speed but reduce precision; lower values ensure accurate execution of fine details.	Balances execution time and drawing accuracy based on the task's requirements.
accel (acceleration)	Defines the acceleration of the robot arm during movements.	Higher values allow quicker transitions; lower values prevent jerky movements that might compromise drawing quality.	Ensures smooth and stable movements during drawing, especially for delicate tasks.

Table 1: Hyperparameter

SPIN_RATE was another parameter that controls the feedback response time of the robot, but it is not considered a hyperparameter as it was kept at 50Hz for all test cases. Focusing on one parameter in specific, `max_dimension`, is helpful in understanding their importance in scalability. The first step in the pipeline involved processing the input image to extract contours suitable for drawing. A key breakthrough in this phase was identifying `max_dimension` as a critical hyperparameter that directly influenced the quality of the drawing. This parameter controlled the scaling of the image, enabling the system to handle a wide range of image complexities. Larger values of `max_dimension` retained more detail, resulting in higher precision at the cost of increased processing time. Conversely, smaller values simplified the image, making it faster to process but less detailed. This was also directly correlated with drawing time.

The image was initially resized using `max_dimension` to scale its largest dimension to this value while maintaining its aspect ratio,

```
max_dimension = 2000  # Adjustable parameter for drawing quality
scaling_factor = max_dimension / float(max(height, width))
image = cv2.resize(image, None, fx=scaling_factor, fy=scaling_factor,
interpolation=cv2.INTER_AREA)
```

This approach was inspired by the idea that bringing an image closer (effectively increasing `max_dimension`) reveals finer details, while moving it further away (decreasing `max_dimension`) hides nuances. Take this example image,

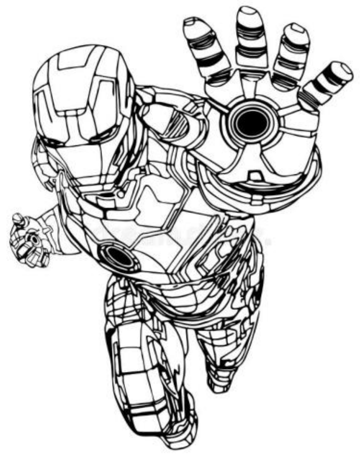

Figure 1: Ironman Reference Image

Before the image is drawn, the detected key points are shown, then the image is processed and the outline of the image to be drawn is shown. This allowed for verification and improvements to be made without the need for waiting for the robot to draw the whole image. Running this figure 1 with a `max_dimension` of 500 yields the following,

```
Image downscaled by a factor of 0.62 to reduce noise and processing
time.
Calculated scale: 0.2854
Detecting contours...
Detected 75 contours
```

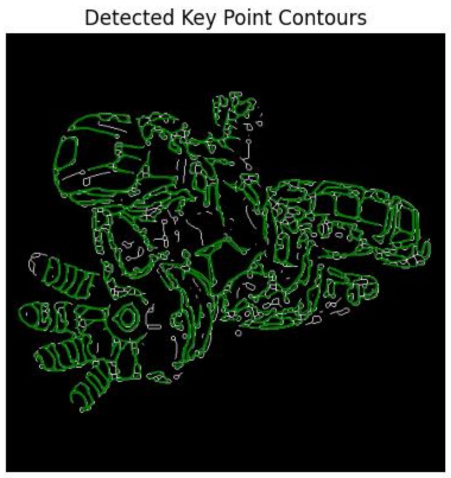

Figure 2: Figure 1 detected key point contours (dim 500)

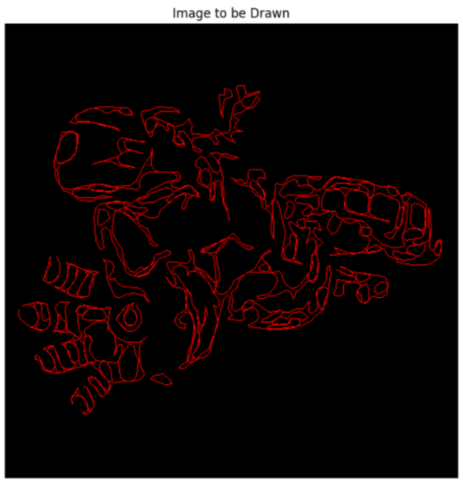

Figure 3: Figure 1 to be drawn (dim 500)

Evidently there are significant details missing. Now setting the dimension value to 2000,

```
Image downscaled by a factor of 2.5 to increase details.
Calculated scale: 0.0713
Detecting contours...
Detected 552 contours
```

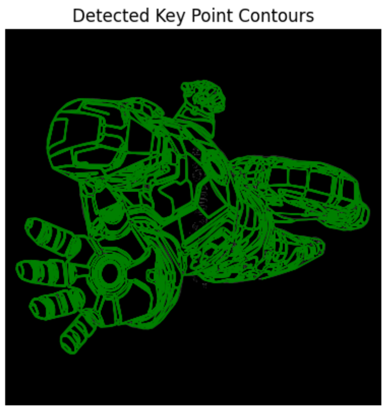

Figure 4: Figure 1 detected key point contours (dim 2000)

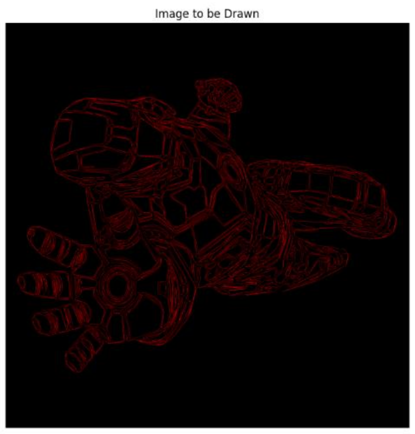

Figure 5: Figure 1 image to be drawn (dim 2000)

It becomes quickly apparent that leveraging this hyperparameter enables detail control. That being said, the number of contours has more than quadrupled, this adds significant drawing time to the image. It is also important to highlight that there are images where additional detail becomes redundant whilst still taking a significant amount of time to be drawn. It is important to establish the value dependent on the use case.

IV. Results

The results of this project demonstrate the effectiveness of the developed pipeline in enabling the UR3 robot to replicate input images as precise drawings. By carefully tuning hyperparameters in Tasks 1 and 2, the robot successfully produced visually accurate and aesthetically pleasing outputs. The success of the robot in the final demonstration is underscored by winning the voting competition for the best-drawn image. A pen was chosen as the drawing instrument to best reflect the fine details.

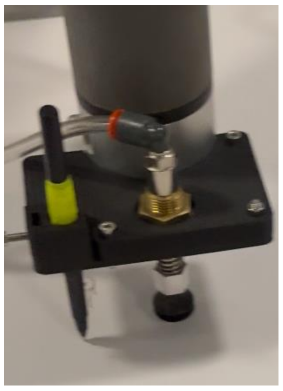

Figure 6: Pen in robot end effector

A. Task 1: Results and Hyperparameter Evaluation

In Task 1, the hyperparameters were adjusted to prioritize speed over detail, reflecting the need for rapid processing. The chosen settings reduced the complexity of the image while ensuring that essential features were preserved.

Hyperparameter	**Value**	**Purpose**
max_dimension	600	Reduced to simplify image details, prioritizing speed over precision.
merge_threshold	15	Increased to merge nearby contours aggressively, reducing redundant drawing paths.
step (in interpolation)	5	Larger steps reduced the number of interpolated points, accelerating the drawing process.
window_size	2	Smaller smoothing window retained enough detail to avoid oversmoothing.
angle_threshold	15°	Larger threshold grouped more points into straight lines, optimizing movement paths.
vel (velocity)	12	Increased speed of arm movement for faster execution.
accel (acceleration)	10	Increased acceleration for rapid transitions between points.

Table 2: Task 1 hyperparameters

The result of Task 1 is shown below, highlighting the robot's ability to produce a simplified yet clear representation of the input image. While the output lacked the finer details seen in later tasks, it demonstrated the effectiveness of a speed-prioritized approach, completing the drawing in less than 10 minutes as per the task specifications.

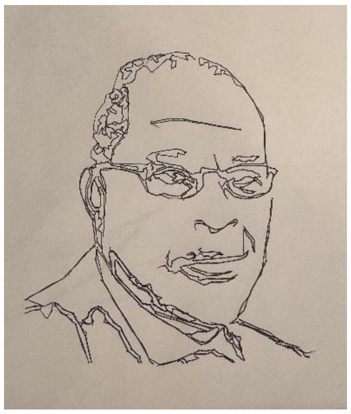

Figure 7: Task 1 image

B. Task 2: Results and Hyperparameter Evaluation

In Task 2, the focus shifted to achieving a balance between precision and efficiency.

Hyperparameter	*Value*	*Purpose*
max_dimension	2000	Higher value retained finer image details for more precise contour detection and drawing.
merge_threshold	10	Moderate threshold preserved individual contours, ensuring accurate path tracing.
step (in interpolation)	2	Smaller steps enhanced contour smoothness and drawing detail.
window_size	3	Balanced smoothing removed noise while preserving fine details in contours.
angle_threshold	10°	Stricter threshold reduced grouping, enabling detailed reproduction of curved paths.
vel (velocity)	8	Reduced speed ensured higher accuracy in executing fine details.
accel (acceleration)	8	Moderate acceleration avoided jerky movements, ensuring smooth transitions.

Figure 8: Task 2 hyperparameters

The final image drawn was chosen due to its complexity being able to showcase our algorithm, and since it would be recognizable and draw attention from voters. The image drawn by the robot was a clear and precise reproduction of the input, winning the competition for the best-drawn image. This success highlights the robustness of the pipeline and the effectiveness of the hyperparameter tuning.

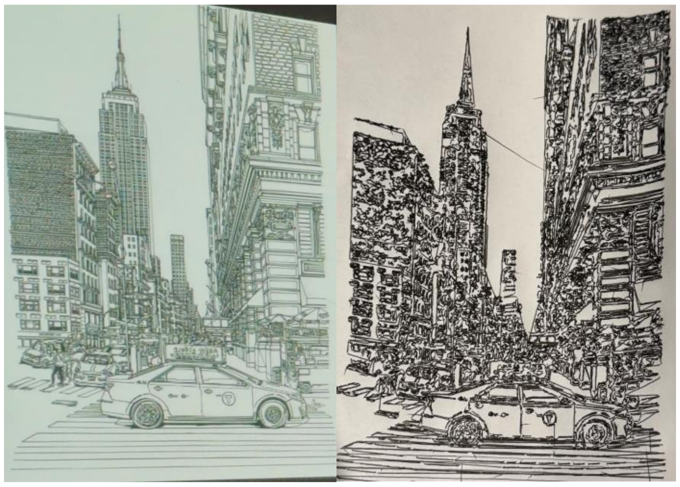

Figure 9: Reference and Drawn task 2 image

The key points and contours for the submitted image are visualized below, showing the processed data that guided the robot's movements.

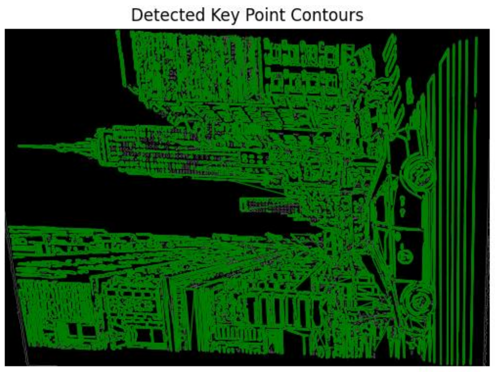

Figure 10: Submitted image contours (rotated as it is drawn portrait)

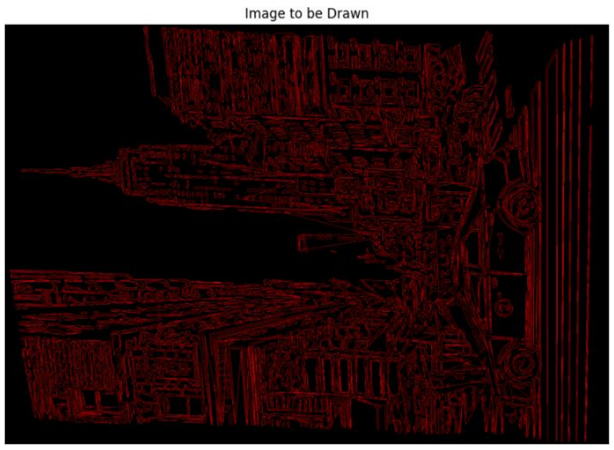

Figure 11: Submitted image to be drawn

C. Overall Performance

In addition to the submitted image, the robot performed well on a variety of other test images, showcasing its versatility. The system's scalability, achieved through the adjustable hyperparameters, enabled it to handle images of varying complexity. This adaptability was a key factor in the success of the project, allowing the pipeline to be fine-tuned for specific tasks and requirements.

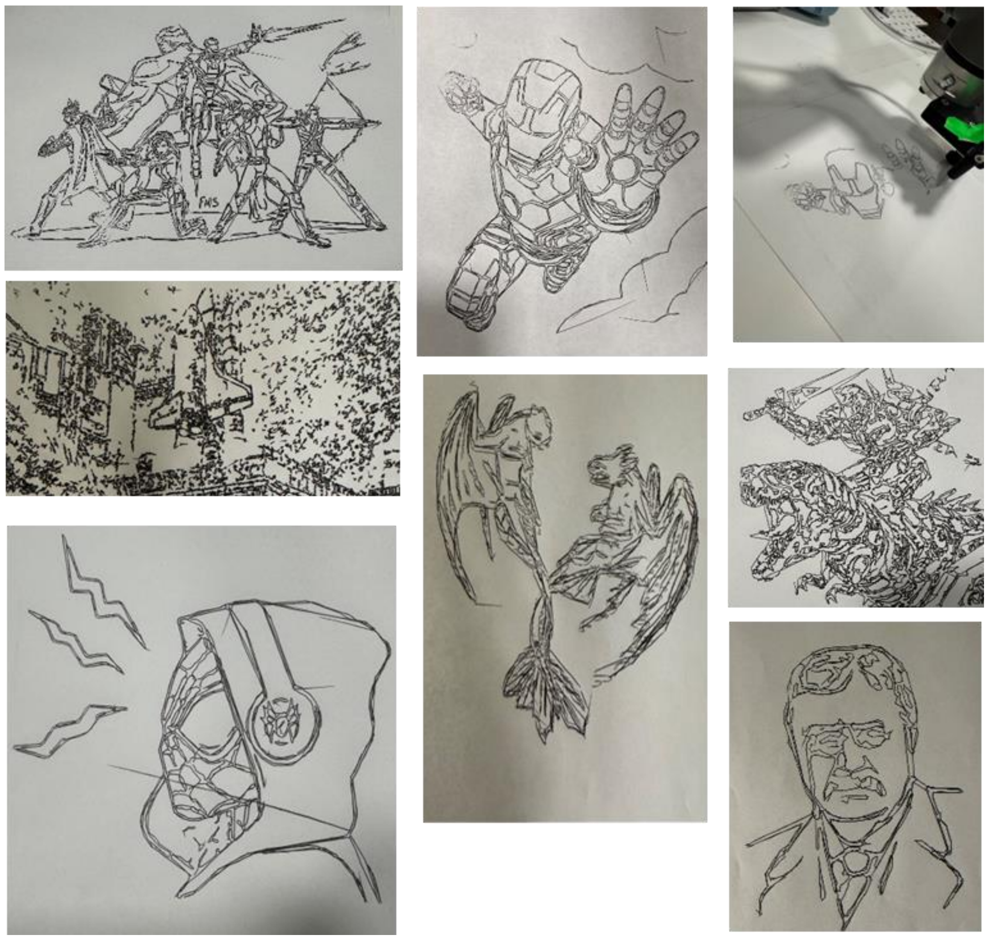

Figure 12: Other drawn images

The results demonstrate that the robot not only met but exceeded expectations, successfully integrating image processing and robotic control to achieve a high level of performance. The voting competition win further validates the quality of the final implementation.

D. Self-Evaluation

Throughout the process of development and testing there were continuous needs to go back and reapproach and even restart. This involved choosing different contour mapping algorithms, adding and removing helper functions, choosing a pen or sharpie and designing approaches that were based on intuition (for example the scaling mechanism for detail). The best evaluator for this progress was the contour map and final image drawn plots. It allowed for the comparison of different algorithms and hyperparameter tuning without the need to wait for the robot to draw some images.

V. Conclusion

The success of this project demonstrates the effectiveness of combining advanced image processing techniques with robotic kinematics to achieve precise and scalable task execution. Through iterative development and careful hyperparameter tuning, the UR3 robot was able to replicate input images as detailed and accurate drawings, culminating in a winning performance in the voting competition for the best-drawn image.

Key to this success was the ability to dynamically adjust parameters like `max_dimension` to balance detail and speed, ensuring the pipeline could handle a wide range of image complexities. In addition the implementation of a series of functions that enabled improvements of the raw key point data allowed for the most optimal placement of points that the robot could draw. This adaptability highlights the robustness of the system and its potential for application in fields such as manufacturing, art, and automation.

The project also underscored the importance of a structured development approach, integrating preprocessing, transformation, and execution phases seamlessly. Overall, this lab provided valuable insights into real-world robotic applications, blending technical precision with creative problem-solving to achieve outstanding results.

A. Reflection on the Labs

The lab sequence was an incredibly rewarding experience. As a computer science major, I appreciated the opportunity to apply what I have learned in a hands-on setting, exploring how programming and robotics intersect. The Tower of Hanoi lab was particularly exciting, as it allowed me to see the recursive algorithm I had studied extensively come to life through the robot's movements.

Lab 5 stood out for its integration of computer vision and robotics, which was both challenging and inspiring. Using OpenCV to detect objects and map pixel coordinates to the world frame sparked ideas about real-world applications, such as self-driving cars and automation processes.

The final project lab was the culmination of all the skills learned, combining computer vision, kinematics, and drawing execution. Seeing the robot accurately replicate an image and winning the best-drawn image competition was immensely satisfying. However, accessing workstations was a challenge, with labs overcrowded even at 2am. I recommend reducing class sizes, adding more stations, or introducing a scheduling system to address this issue. Despite this, the labs were highly enjoyable and instrumental in sparking further interest in robotics and computer vision.

VI.Acknowledgments

The author wishes to express their gratitude to TA Lukas and Lab Partner Chris for their aid in completing the lab as per the specifications.

VII.References

[1] Department of Electrical Engineering, *Lab 5 Manual: Image Contour Detection and Drawing*, University of Illinois at Urbana-Champaign, Urbana, IL, 2024.

[2] Lynch, K. M., and Park, F. C., *Modern Robotics: Mechanics, Planning, and Control*, Preprint version, Cambridge University Press, Cambridge, U.K., 2017.

VIII. Appendices

Appendix A: Kinematics Code

```
#!/usr/bin/env python

import numpy as np

import math

from scipy.linalg import expm

from lab4_header import *


"""

Use 'expm' for matrix exponential.

Angles are in radian, distance are in meters.

"""

def Get_MS():

  # =================== Your code starts here ====================#

  # Fill in the correct values for a1~6 and q1~6, as well as the M matrix

  s1 = np.array([[0],[0],[1],[150],[150],[0]])

  s2 = np.array([[0],[1],[0],[-162],[0],[-150]])

  s3 = np.array([[0],[1],[0],[-162],[0],[94]])

  s4 = np.array([[0],[1],[0],[-162],[0],[307]])

  s5 = np.array([[1],[0],[0],[0],[162],[-260]])

  s6 = np.array([[0],[1],[0],[-162],[0],[390]])

  S = np.column_stack([s1, s2, s3, s4, s5, s6])



  M = np.array([[0,-1,0,390],[0,0,-1, 401],[1,0,0,215.5],[0,0,0,1]])




  # ==============================================================#
  return M, S



def skew_symmetric_6x1(screw_axis):
    """
    Create the skew-symmetric matrix
    """
    omega = screw_axis[:3]
    v = screw_axis[3:]

    omega_skew = np.array([[0, -omega[2], omega[1]],
                           [omega[2], 0, -omega[0]],
                           [-omega[1], omega[0], 0]])

    skew_matrix = np.zeros((4, 4))
    skew_matrix[:3, :3] = omega_skew
    skew_matrix[:3, 3] = v

    return skew_matrix

def calculate_T01(S, M, theta):
    """
    Calculate the pose T_01

    Parameters:
    - S: 6xN srcre matrix
    - M: 4x4 initial config matrix
    - theta: 1xN the joint angles
    Returns:
    - T_01: 4x4 transformation matrix
    """
    T = np.eye(4)

    for i in range(S.shape[1]):
        screw_axis = S[:, i]
        skew_matrix = skew_symmetric_6x1(screw_axis)
        exp_S_theta = expm(skew_matrix * theta[i])
        T = np.dot(T, exp_S_theta)

    T_01 = np.dot(T, M)

    return T_01

"""
Function that calculates encoder numbers for each motor
"""
def lab_fk(theta1, theta2, theta3, theta4, theta5, theta6):

  # Initialize the return_value
  return_value = [None, None, None, None, None, None]

  print("Foward kinematics calculated:\n")

  # =================== Your code starts here ====================#
  M, S = Get_MS()

  theta = np.array([theta1, theta2, theta3, theta4, theta5, theta6 ])
```

```
    T = calculate_T01(S, M, theta)





    # ==============================================================#

    return_value[0] = theta1 + PI
    return_value[1] = theta2
    return_value[2] = theta3
    return_value[3] = theta4 - (0.5*PI)
    return_value[4] = theta5
    return_value[5] = theta6

    return return_value

"""
Function that calculates an elbow up Inverse Kinematic solution for the UR3
"""
def lab_invk(xWgrip, yWgrip, zWgrip, yaw_WgripDegree):
    # =================== Your code starts here ====================#

    # Convert degrees to radians
    yaw_WgripDegree_rad = np.deg2rad(yaw_WgripDegree)

    # Linear change from corner to centre
    x_grip = xWgrip + 150
```

```
y_grip = yWgrip - 150

z_grip = zWgrip - 10


x_cen = x_grip - 53.5*np.cos(yaw_WgripDegree_rad)

y_cen = y_grip - 53.5*np.sin(yaw_WgripDegree_rad)

z_cen = z_grip


phi = math.atan2(y_cen, x_cen)

theta1 = phi - math.asin(110/(np.sqrt(x_cen**2 + y_cen**2)))


theta6 = np.pi/2 + theta1 - yaw_WgripDegree_rad


x_3end = x_cen + (27 + 83)*np.sin(theta1) - 83*np.cos(theta1)

y_3end = y_cen - (27 + 83)*np.cos(theta1) - 83*np.sin(theta1)

z_3end = z_cen + 59 + 82


L1 = 152

L3 = 244

L5 = 213

A = z_3end - L1

B = x_3end

C = np.sqrt(A**2 + B**2)

alpha = math.acos((L3**2 + L5**2 - C**2)/(2*L3*L5))

beta = math.acos((L3**2 + C**2 - L5**2)/(2*L3*C))

gamma = math.atan2(A, B)

psi = math.acos((L5**2 + C**2 - L3**2)/(2*L5*C))

theta2 = -(beta + gamma)

theta3 = np.pi - alpha

theta4 = -(np.pi - np.absolute(theta2) - alpha)
```

```
    theta5 = -np.pi/2

    print("Thetas:")
    temp = np.array([theta1, theta2, theta3, theta4, theta5, theta6])
    print(temp.reshape(6,1))

    print("T Matrix")
    print(lab_fk(theta1, theta2, theta3, theta4, theta5, theta6))
    # =============================================================#
    return lab_fk(theta1, theta2, theta3, theta4, theta5, theta6)

  import numpy as np

  # Define the target (input) and measured positions
  input1 = np.array([100, 100, 150, 90])  # Includes X, Y, Z, and Yaw
  measured1 = np.array([110, 108, 150])   # Measured X, Y, Z positions

  # Extract X, Y, Z coordinates from input and measured arrays
  i_x, i_y, i_z, _ = input1.flatten()
  m_x, m_y, m_z = measured1.flatten()

  # Calculate scalar (Euclidean) error
  scalar_error = np.sqrt((i_x - m_x)**2 + (i_y - m_y)**2 + (i_z - m_z)**2)

  # Display the result
scalar_error
```

Appendix B: Drawing Code

```
#!/usr/bin/env python


import sys

import copy

import time

import rospy


import numpy as np

import cv2

import matplotlib.pyplot as plt

from scipy.interpolate import CubicSpline

from skimage.morphology import skeletonize

from shapely.geometry import LineString, Polygon

from shapely.ops import unary_union


from final_header import *

from final_func import *


################ Pre-defined parameters and functions below ################


# Constants

SPIN_RATE = 50  # Hz

PI = np.pi


# UR3 home location (in radians)

home = [270 * PI / 180.0, -90 * PI / 180.0, 90 * PI / 180.0,

        -90 * PI / 180.0, -90 * PI / 180.0, 135 * PI / 180.0]
```

```
# UR3 current position, using home position for initialization
current_position = copy.deepcopy(home)

thetas = [0.0] * 6

digital_in_0 = 0
analog_in_0 = 0.0

suction_on = True
suction_off = False

current_io_0 = False
current_position_set = False

# Define image processing parameters
sheet_x_len = 186.7  # mm
sheet_y_len = 142.7  # mm
x_offset = 227.0      # mm
y_offset = 177.0      # mm
scale = 1.0
height_sheet = 10.15  # mm
height_free = height_sheet + 4  # mm

################ Callback Functions ################

def input_callback(msg):
    global digital_in_0
    digital_in_0 = msg.DIGIN & 1  # Only look at least significant bit
```

```
def position_callback(msg):
    global thetas, current_position, current_position_set

    thetas = msg.position[:6]
    current_position = copy.deepcopy(thetas)
    current_position_set = True

################ Control Functions ################

def gripper(pub_cmd, loop_rate, io_0):
    global SPIN_RATE, thetas, current_io_0, current_position

    error = 0
    spin_count = 0
    at_goal = False

    current_io_0 = io_0

    driver_msg = command()
    driver_msg.destination = current_position
    driver_msg.v = 1.0
    driver_msg.a = 1.0
    driver_msg.io_0 = io_0
    pub_cmd.publish(driver_msg)

    while not at_goal and not rospy.is_shutdown():
        if all(abs(thetas[i] - driver_msg.destination[i]) < 0.0005 for i in
range(6)):
            rospy.loginfo("Goal is reached!")
```

```
              at_goal = True

          loop_rate.sleep()

          spin_count += 1
          if spin_count > SPIN_RATE * 5:
              pub_cmd.publish(driver_msg)
              rospy.loginfo("Re-published driver_msg")
              spin_count = 0

      return error

  def move_arm(pub_cmd, loop_rate, dest, vel, accel, move_type):
      global thetas, SPIN_RATE

      error = 0
      spin_count = 0
      at_goal = False

      driver_msg = command()
      driver_msg.destination = dest
      driver_msg.v = vel
      driver_msg.a = accel
      driver_msg.io_0 = current_io_0
      driver_msg.move_type = move_type  # Move type ('J' for Joint, 'L' for
Linear)
      pub_cmd.publish(driver_msg)

      loop_rate.sleep()
```

```
    while not at_goal and not rospy.is_shutdown():
        if all(abs(thetas[i] - driver_msg.destination[i]) < 0.0005 for i in
range(6)):
            at_goal = True
            rospy.loginfo("Goal is reached!")

        loop_rate.sleep()

        spin_count += 1
        if spin_count > SPIN_RATE * 5:
            pub_cmd.publish(driver_msg)
            rospy.loginfo("Re-published driver_msg")
            spin_count = 0

    return error

################ Helper Functions ################

def lift_pen(pub_cmd, loop_rate, target_point, height, vel, accel):
    """Lift or lower the pen to a specified height."""
    x, y, _ = target_point
    new_z = height
    lifted_point = np.array([x, y, new_z])
    move_arm(pub_cmd, loop_rate, lifted_point.tolist(), vel, accel, 'L')

def draw_path(pub_command, loop_rate, path, vel, accel):
    """Draw a continuous path by following the given list of points."""
    if not path:
        return

    # Move to the start of the path (free position)
    try:
        start_xw, start_yw = IMG2W(path[0][1], path[0][0])
        start_free = lab_invk(start_xw, start_yw, height_free, 0)
        start_free = [float(val) for val in start_free]

        # Move to the start of the path (drawing position)
        start_drawing = lab_invk(start_xw, start_yw, height_sheet, 0)
        start_drawing = [float(val) for val in start_drawing]

        move_arm(pub_command, loop_rate, start_free, vel, accel, 'J')
        move_arm(pub_command, loop_rate, start_drawing, vel, accel, 'L')
    except ValueError as e:
        rospy.logwarn(f"Skipping invalid start point: {e}")
        return

    # Draw the path point by point
    for point in path:
        try:
            xw, yw = IMG2W(point[1], point[0])
            drawing_point = lab_invk(xw, yw, height_sheet, 0)
            drawing_point = [float(val) for val in drawing_point]
            move_arm(pub_command, loop_rate, drawing_point, vel, accel, 'L')
        except ValueError as e:
            rospy.logwarn(f"Skipping invalid keypoint during drawing: {e}")

def find_contours(image):
    """Find and preprocess contours from the given image."""
    if image is None:
        raise ValueError("Error: Input image is None. Check if the image file
exists and is valid.")

    # Convert to grayscale
    grey_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian Blur to reduce noise
    blur = cv2.GaussianBlur(grey_img, (5, 5), 0)

    # Adaptive Thresholding to handle varying lighting conditions and reduce
noise
    thresh = cv2.adaptiveThreshold(blur, 255,
                                   cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                   cv2.THRESH_BINARY_INV, 11, 2)

    # Morphological Operations to remove small noise and close gaps
    kernel = np.ones((3, 3), np.uint8)
    morph = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=1)
    morph = cv2.morphologyEx(morph, cv2.MORPH_CLOSE, kernel, iterations=1)

    # Edge Detection using Canny
    edges = cv2.Canny(morph, 50, 150, apertureSize=3)

    # Dilate and Erode to close gaps further
    edges = cv2.dilate(edges, kernel, iterations=1)
    edges = cv2.erode(edges, kernel, iterations=1)
```

```
        # Apply thinning to merge thick lines
        skeleton = skeletonize(edges > 0).astype(np.uint8) * 255

        # Find contours with hierarchy
        contours, hierarchy = cv2.findContours(skeleton, cv2.RETR_TREE,
cv2.CHAIN_APPROX_NONE)

        # Filter contours by area to remove noise
        min_contour_area = 100
        filtered_contours = [cnt for cnt in contours if cv2.contourArea(cnt) >
min_contour_area]

        # Simplify contours using RDP algorithm with minimal simplification
        simplified_contours = []
        for cnt in filtered_contours:
            arc_len = cv2.arcLength(cnt, True)
            epsilon = 0.001 * arc_len  # Very small epsilon to retain maximum
detail
            approx = cv2.approxPolyDP(cnt, epsilon, True)
            points = [tuple(point[0]) for point in approx]
            if len(points) > 1:
                interpolated = interpolate_contour(points, step=2)
                smoothed = smooth_path(interpolated, window_size=3)
                simplified_contours.append(smoothed)

        # Merge similar or close contours to prevent double outlines
        merged_contours = merge_close_contours(simplified_contours,
merge_threshold=10)
```

```
        # Further simplify paths using Shapely
        further_simplified_contours = []
        for contour in merged_contours:
            line = LineString(contour)
            # Simplify the path with a tolerance
                          simplified_line     =     line.simplify(tolerance=1.0,
preserve_topology=False)
            if simplified_line.is_empty:
                continue
            # Extract points from the simplified LineString
            simplified_points = list(simplified_line.coords)
            # Convert to integer tuples
            simplified_points = [(int(x), int(y)) for x, y in simplified_points]
            # Apply adaptive sampling
            adaptively_sampled = adaptive_sample(simplified_points, max_step=10,
min_step=5)  # Adjusted steps
            further_simplified_contours.append(adaptively_sampled)

        # Visualization for debugging
        visualization = cv2.cvtColor(skeleton, cv2.COLOR_GRAY2BGR)
        for contour in further_simplified_contours:
              cv2.polylines(visualization,  [np.array(contour)],  isClosed=True,
color=(0, 255, 0), thickness=1)  # Green contours
        cv2.imshow('Filtered and Simplified Contours', visualization)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        return further_simplified_contours
```

```
  def interpolate_contour(contour, step=2):
      """Interpolate additional points along a contour for higher detail."""
      interpolated = []
      for i in range(len(contour)):
          start_point = contour[i]
          end_point = contour[(i + 1) % len(contour)]
                  distance    =    np.linalg.norm(np.array(end_point)    -
np.array(start_point))
          num_steps = max(int(distance / step), 1)
          for j in range(num_steps):
              interpolated_point = (
                  int(start_point[0] + (end_point[0] - start_point[0]) * j /
num_steps),
                  int(start_point[1] + (end_point[1] - start_point[1]) * j /
num_steps)
              )
              interpolated.append(interpolated_point)
      return interpolated


  def smooth_path(path, window_size=3):
      """Apply a moving average to smooth the path points."""
      if len(path) < window_size:
          return path
      smoothed = []
      for i in range(len(path)):
          window  =  path[max(i  -  window_size,  0):min(i  +  window_size  +  1,
len(path))]
          avg_x = int(np.mean([p[0] for p in window]))
          avg_y = int(np.mean([p[1] for p in window]))
```

```
            smoothed.append((avg_x, avg_y))
        return smoothed


    def merge_close_contours(contours, merge_threshold=10):
        """Merge contours that are within a certain proximity to prevent double
outlines."""
        merged_contours = []
        while contours:
            base = contours.pop(0)
            to_merge = []
            for i, contour in enumerate(contours):
                # Calculate distance between the end of base and start of contour
                        distance   =   np.linalg.norm(np.array(base[-1])   -
np.array(contour[0]))
                if distance < merge_threshold:
                    base.extend(contour)
                    to_merge.append(i)
            # Remove merged contours from the list
            for index in sorted(to_merge, reverse=True):
                contours.pop(index)
            merged_contours.append(base)
        return merged_contours


    def adaptive_sample(path, max_step=10, min_step=5):
        """Adaptively sample points based on curvature."""
        if len(path) < 3:
            return path

        sampled = [path[0]]
        i = 1
        while i < len(path) - 1:
            p0 = np.array(path[i - 1])
            p1 = np.array(path[i])
            p2 = np.array(path[i + 1])

            # Calculate the angle between segments p0->p1 and p1->p2
            v1 = p1 - p0
            v2 = p2 - p1
            if np.linalg.norm(v1) == 0 or np.linalg.norm(v2) == 0:
                angle_deg = 0
            else:
                angle = np.arccos(
                             np.clip(np.dot(v1,  v2)  /  (np.linalg.norm(v1)  *
    np.linalg.norm(v2)), -1.0, 1.0)
                )
                angle_deg = np.degrees(angle)

            # If the angle is significant, keep the point
            if angle_deg > 10:  # Threshold angle to determine curvature
                sampled.append(tuple(p1))
                i += 1
            else:
                 # Merge points in straight segments by skipping intermediate
    points
                j = i + 1
                while j < len(path) - 1:
                    p_prev = np.array(path[j - 1])
                    p_curr = np.array(path[j])
```

```
                    p_next = np.array(path[j + 1])
                    if np.linalg.norm(p_curr - p_prev) == 0 or np.linalg.norm(p_next - p_curr) == 0:
                        angle_deg = 0
                    else:
                        v1 = p_curr - p_prev
                        v2 = p_next - p_curr
                        angle = np.arccos(
                            np.clip(np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2)), -1.0, 1.0)
                        )
                        angle_deg = np.degrees(angle)
                    if angle_deg > 10:
                        break
                    j += 1
                sampled.append(tuple(path[j]))
                i = j + 1
        sampled.append(path[-1])
        return sampled

    def group_straight_lines(path, angle_threshold=10):
        """Group consecutive points that form a straight line."""
        if len(path) < 3:
            return [path]

        grouped = []
        current_group = [path[0], path[1]]

        for i in range(2, len(path)):
            p0 = np.array(current_group[-2])
            p1 = np.array(current_group[-1])
            p2 = np.array(path[i])

            v1 = p1 - p0
            v2 = p2 - p1

            # Calculate the angle between the two vectors
            if np.linalg.norm(v1) == 0 or np.linalg.norm(v2) == 0:
                angle_deg = 0
            else:
                angle = np.arccos(
                    np.clip(np.dot(v1, v2) / (np.linalg.norm(v1) *
np.linalg.norm(v2)), -1.0, 1.0)
                )
                angle_deg = np.degrees(angle)

            if angle_deg < angle_threshold:
                current_group.append(path[i])
            else:
                grouped.append(current_group)
                current_group = [path[i - 1], path[i]]

        grouped.append(current_group)
        return grouped

    def IMG2W(row, col):
        """Transform image coordinates to world coordinates within valid
bounds."""
```

```
        if not (0 <= row < image_y_len and 0 <= col < image_x_len):
            raise ValueError(f"Invalid image coordinates: ({row}, {col}). Expected
range: "
                                      f"(0  <=  row  <  {image_y_len},  0  <=  col  <
{image_x_len})")

        # Map to world coordinates
        x = x_offset + col * scale
        y = y_offset + row * scale

        return x, y

    def sort_contours_by_proximity(contours):
        """Sort contours based on proximity to minimize travel distance."""
        sorted_contours = []
        current_position = (x_offset, y_offset)

        while contours:
            closest_contour = min(
                contours,
                    key=lambda  c:  np.linalg.norm(np.array(current_position)  -
np.array(IMG2W(c[0][1], c[0][0])))
            )
            sorted_contours.append(closest_contour)
            contours.remove(closest_contour)
             current_position = IMG2W(closest_contour[-1][1], closest_contour[-
1][0])

        return sorted_contours
```

```
  def add_hatching(contour, spacing=10):
      """Add hatching lines within a contour to simulate shading."""
      # Create a mask for the contour
      mask = np.zeros((int(image_y_len), int(image_x_len)), dtype=np.uint8)
       cv2.drawContours(mask, [np.array(contour)], -1, 255, -1)  # Filled
contour

      # Generate horizontal lines with specified spacing
      hatching = []
      for y in range(0, int(image_y_len), spacing):
          # Find the edges of the hatching line within the mask
                  _, cols = cv2.findNonZero(mask[y:y+1, :]).T if
cv2.findNonZero(mask[y:y+1, :]) is not None else ([], [])
          if cols.size == 0:
              continue
          min_col = np.min(cols)
          max_col = np.max(cols)
          # Add start and end points of the hatching line
          hatching.append((min_col[0], y))
          hatching.append((max_col[0], y))

      return hatching

  def draw_image(contours, pub_command, loop_rate, vel, accel):
      """Draw the image by sending commands to the UR3 robot."""
      # Sort contours to minimize travel distance
      sorted_contours = sort_contours_by_proximity(contours)
      rospy.loginfo("Sorted contours by proximity.")
```

```
        # Draw sorted contours on a blank image for verification
          visualization  =  np.zeros((int(image_y_len),  int(image_x_len),  3),
dtype=np.uint8)
        for contour in sorted_contours:
              cv2.polylines(visualization,  [np.array(contour)],  isClosed=True,
color=(0, 0, 255), thickness=1)  # Red contours
        cv2.imshow('Sorted Contours', visualization)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        for contour in sorted_contours:
            # Group points into straight lines
            grouped_segments = group_straight_lines(contour, angle_threshold=10)

            for segment in grouped_segments:
                if len(segment) < 2:
                    continue

                # Move to the start of the segment (free position)
                try:
                    start_xw, start_yw = IMG2W(segment[0][1], segment[0][0])
                    start_free = lab_invk(start_xw, start_yw, height_free, 0)
                    start_free = [float(val) for val in start_free]

                    # Move to the start of the segment (drawing position)
                     start_drawing = lab_invk(start_xw, start_yw, height_sheet,
0)
                    start_drawing = [float(val) for val in start_drawing]
                   move_arm(pub_command, loop_rate, start_free, vel, accel, 'J')
                    move_arm(pub_command, loop_rate, start_drawing, vel, accel,
   'L')
                except ValueError as e:
                    rospy.logwarn(f"Skipping invalid start point: {e}")
                    continue

                # Define the end of the segment
                try:
                    end_xw, end_yw = IMG2W(segment[-1][1], segment[-1][0])
                    end_drawing = lab_invk(end_xw, end_yw, height_sheet, 0)
                    end_drawing = [float(val) for val in end_drawing]
                      move_arm(pub_command, loop_rate, end_drawing, vel, accel,
   'L')
                except ValueError as e:
                    rospy.logwarn(f"Skipping invalid end point: {e}")
                    continue

    def main():
        global image_x_len, image_y_len, scale, x_offset, y_offset, height_sheet,
   height_free

        # Initialize ROS node
        rospy.init_node('lab5node')

        # Initialize publisher for ur3/command with buffer size of 10
        pub_command = rospy.Publisher('ur3/command', command, queue_size=10)
```

```
    # Initialize subscribers
    rospy.Subscriber('ur3/position', position, position_callback)
    rospy.Subscriber('ur3/gripper_input', gripper_input, input_callback)

    # Wait until ROS is ready
    while not rospy.is_shutdown() and not current_position_set:
        rospy.loginfo("Waiting for initial position...")
        rospy.sleep(0.1)

    # Initialize the rate to publish to ur3/command
    loop_rate = rospy.Rate(SPIN_RATE)

    # Velocity and acceleration of the UR3 arm
    vel = 8.0
    accel = 8.0

    # Move to the home position
    move_arm(pub_command, loop_rate, home, vel, accel, 'J')

    ##========= Image Processing and Drawing =========##

    # Load the image
    image_path = './images/avengers7.jpg'
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError(f"Image at path '{image_path}' could not be loaded.
Check the file path and format.")
```

```
        # Downscale the image to reduce processing time and noise/increase for
opposite
        # Controls effective resolution of frawn image
        max_dimension = 2000
        height, width = image.shape[:2]

        scaling_factor = max_dimension / float(max(height, width))
        image = cv2.resize(image, None, fx=scaling_factor, fy=scaling_factor,
interpolation=cv2.INTER_AREA)
        rospy.loginfo(f"Image downscaled by a factor of {scaling_factor:.2f} to
reduce noise and processing time.")

        # Flip and rotate image (landscape/portrait)
        image = cv2.flip(image, 0)
        image = cv2.rotate(image, cv2.ROTATE_90_CLOCKWISE)

        # Get image dimensions
        image_y_len = float(image.shape[0])  # Height
        image_x_len = float(image.shape[1])  # Width

        # Calculate scaling factor to fit the image within the sheet
        scale = min(sheet_x_len / image_x_len, sheet_y_len / image_y_len)
        rospy.loginfo(f"Calculated scale: {scale:.4f}")

        # Detect contours using the improved find_contours function
        rospy.loginfo("Detecting contours...")
        contours = find_contours(image)
        rospy.loginfo(f"Detected {len(contours)} contours after simplification
and filtering.")
```

```
    if not contours:
        rospy.logwarn("No contours detected. Exiting.")
        return

    # Sort contours by proximity to optimize drawing path
    rospy.loginfo("Sorting contours by proximity...")
    sorted_contours = sort_contours_by_proximity(contours)
    rospy.loginfo("Sorted contours.")

    # Draw the image
    rospy.loginfo("Starting to draw the image...")
    draw_image(sorted_contours, pub_command, loop_rate, vel, accel)
    rospy.loginfo("Image drawing completed.")

    # Return to the home position
    move_arm(pub_command, loop_rate, home, vel, accel, 'J')

    rospy.loginfo("Task Completed!")
    rospy.loginfo("Use Ctrl+C to exit program")
    rospy.spin()

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```