# 4 Detailed Design

## 4.1 Software System

### 4.1.1 Software architecture

Cobot software system is a crucial part of enabling the overall system to perform pick and place operations safely in the presence of humans in the same workspace during operation. While architecting the software system, several aspects were taken into considerations starting from considering the hardware components including the RGB camera and motors and motor drivers and their software interfaces and real-time constraints moving to choosing the appropriate communication protocols between the different parts of the software system. Finally, a modular design was implemented.

Modular design in our cobot software enables us to break down the software functionality into separate modules that can be developed and tested independently. The software is structured into three main modules, where each module focuses on a specific functionality. The three modules are vision, motion planning and control, and task planning.

Independence and Reusability are key concern to us during the development of all modules to enable them to be reused in different applications or scenarios. For example, a motion control module developed for one cobot with certain specifications related to the mechanical and electrical parts can be reused in another cobot with minimal modifications, saving development time and effort.

The software's three main modules are developed independently and can be tested and maintained individually. This allows for easier debugging, troubleshooting, and updating of specific modules without impacting the entire software system.

Modular design facilitates flexibility and scalability in the cobot software. New modules can be added, or existing modules can be modified without disrupting the entire software structure. This allows for easy customization and adaptation to different cobot configurations or changing requirements.

The following diagram shows the system detailed architecture including the three main software modules and how they are connected with the rest of the parts in an abstract way that illustrates the data flow and sequence of operations.

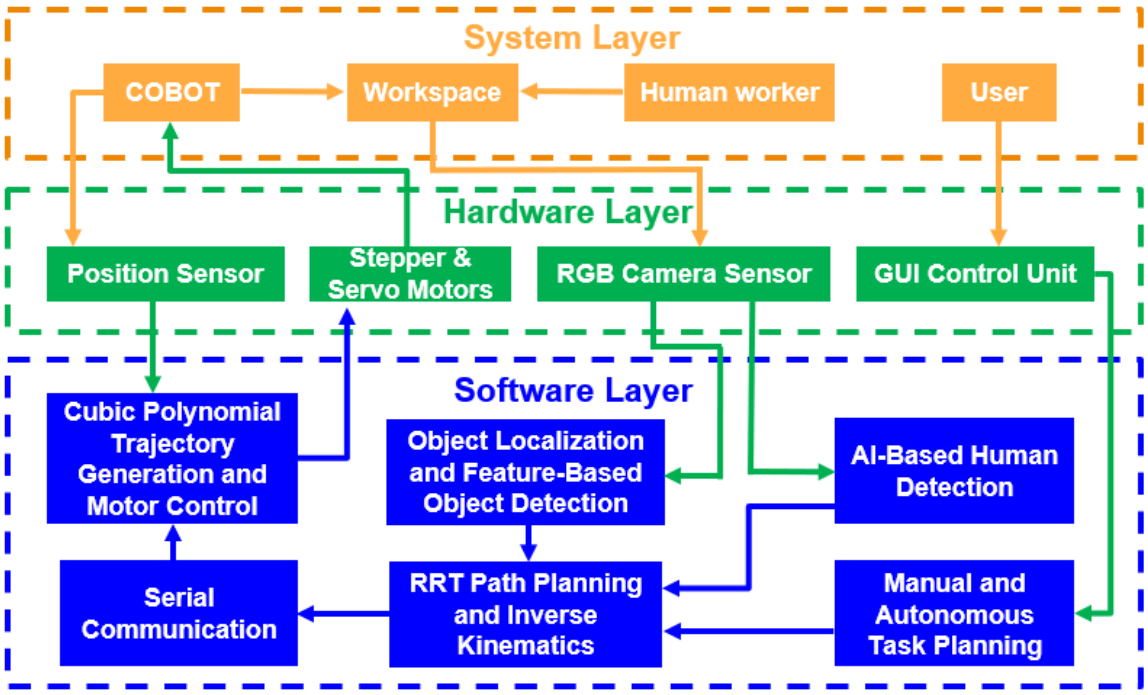



*Figure 1 Detailed system Architecture*

As shown in the diagram above, our **system layer** includes the **robot** and the **human worker** inside the **workspace** and the **user**. They interact with each other and with the robot in a shared workspace simultaneously (collaboration).

Along comes, in the middle of the diagram, our **hardware layer** which receives the data related to activities occurring in the workspace and user interaction with the robot and sends them to the software layer, then it receives motor control signals from the software layer and sends them to the cobot in system layer. The hardware layer consists of **stepper and servo motors, RGB camera sensor, position sensor, and GUI control unit**.

Finally, at the bottom of the diagram, our **software layer,** which receives data from the hardware layer, performs all the processing required by the system to perform its functions, and sends the data back to the hardware layer.

The software layer includes three main modules: **vision, motion planning and control, and task planning modules**.

The vision module is responsible for perceiving the environment and detecting the objects to be picked by the robot as well detecting the presence of humans inside the workspace in real time to ensure safety and collision avoidance as being key feature of the collaborative robot during operation.

The motion planning and control module is responsible for planning the robot path that is free from obstacles and then creating the appropriate trajectories between each waypoint on the path and controlling the motor movements.

The task planning module is responsible for receiving inputs from the operator or the user of the cobot, facilitating the task planning part through the GUI by either manual or autonomous modes, then communicating the data between the high and low level control components in the system to achieve seamless data flow and communication between the user and the internal software components.

The following sections will explain each module in the software system separately.

### 4.1.2 Vision module

#### Vision architecture

The following diagram shows the vision module architecture and how it is connected with the rest of the parts in an abstract way that illustrates the data flow and sequence of operations.

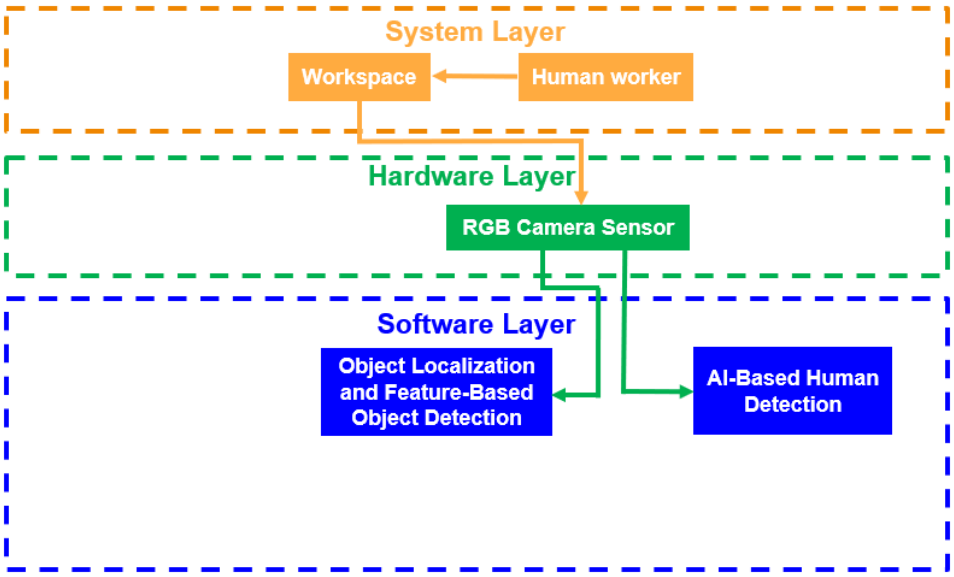



*Figure 2 Vision architecture*

The vision module deals with the workspace and the objects located inside it and deals with the presence of humans inside the workspace while the robot performs its operations.

The module uses RGB camera sensor to monitor the workspace and human activity in real time then feed the camera data streams into two parts: the first part is the objects localization and detection which is responsible for determining the pick position and the second part is the human detection which is responsible for ensuring the safety stop of the robot when there is a human in the workspace while operation.

## object localization and Camera transformation

Camera transformation and object localization play a crucial role in perceiving and understanding the workspace using an RGB camera. In this section, we will discuss the process involved in camera transformation and object localization, as implemented in the code.

The following diagram illustrates the steps involved in object localization and Camera transformation process:

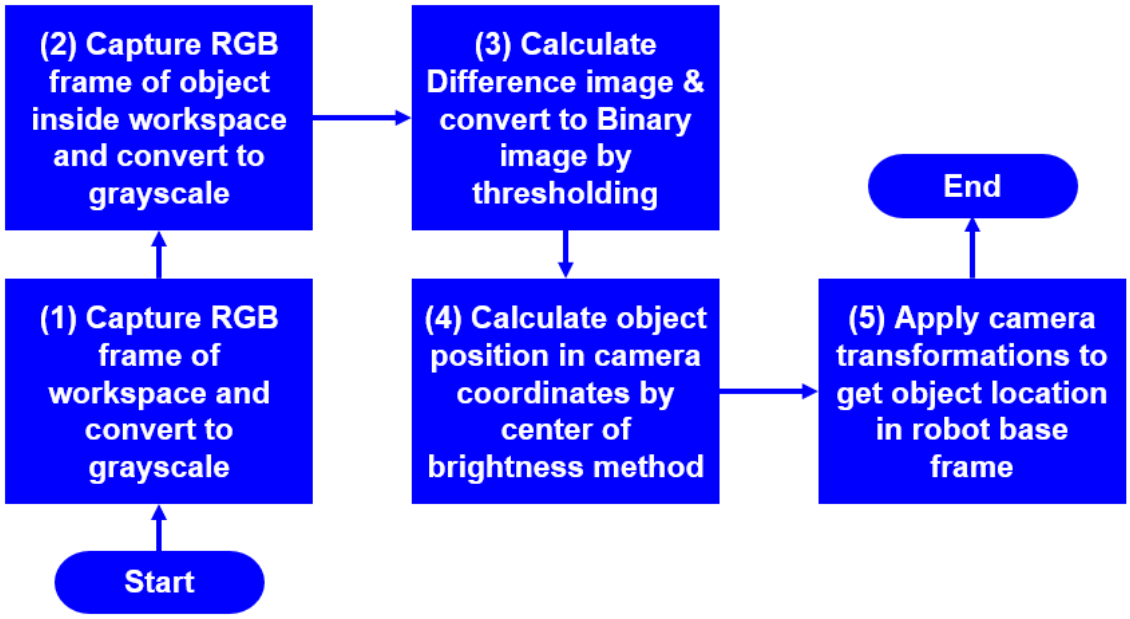



*Figure 3 Object localization algorithm*

In the first step: Capture RGB frame of workspace and convert to grayscale, the RGB camera captures a frame of the workspace. This frame represents the current state of the environment . To simplify further image processing steps, the captured RGB frame is converted to grayscale. Grayscale conversion removes color information, resulting in a single-channel image that retains the intensity values of the original frame. The step is implemented in python programming language using OpenCV library as followed:

```
import cv2
# Capture video from the camera
cap = cv2.VideoCapture(1)
_, frame = cap.read()
global gray_image1
gray_image1 = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

In the second step: Capture RGB frame of object inside workspace and convert to grayscale, a separate RGB frame is captured focusing on the object of interest in order to specifically analyze and localize objects within the workspace. Similar to the previous step, the RGB frame is converted to grayscale for subsequent processing. The step is implemented in python programming language using OpenCV library as followed:

```
import cv2
# Capture video from the camera
cap = cv2.VideoCapture(1)
global frame
_, frame = cap.read()
gray_image2 = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

In the third step: Calculate Difference image & convert to Binary by thresholding:The difference image is obtained by computing the pixel-wise absolute difference between the grayscale frames captured in steps 1 and 2. This difference image highlights the areas where changes have occurred, indicating the presence of the object against the background. To create a binary image for easier object segmentation, a thresholding technique is applied. Pixels above a certain threshold value are set to white (foreground), while those below the threshold are set to black (background). Thresholding is a common image processing technique used to separate objects or regions of interest from the background based on a specified threshold value. In our case, pixels with intensity values less than or equal to 100 are considered part of the background and are assigned a value of 0. Pixels with intensity values greater than 100 are considered part of the objects or regions of interest and are assigned a value of 1. By applying this thresholding technique, the resulting binary image (BW) becomes a binary mask where the background is represented by black pixels (0) and the objects or regions of interest are represented by white pixels (1). This binary image simplifies subsequent processing steps by facilitating object segmentation and further analysis of the objects within the workspace. The step is implemented in python programming language using OpenCV library as followed:

```
import cv2
Difference = np.absolute(np.matrix(np.int16(gray_image1))
                         - np.matrix(np.int16(gray_image2)))
Difference[Difference > 255] = 255
Difference = np.uint8(Difference)
BW = Difference
BW[BW <= 100] = 0
BW[BW > 100] = 1
```

In the fourth step: Calculate object position in camera coordinates by center of brightness method, Using the binary image generated in the previous step, the object's

position in camera coordinates is calculated. This method involves calculating the weighted average of the object's binary image, where the pixel intensities serve as weights. By computing the row and column averages of the object's binary image, the center of brightness is obtained, representing the object's position within the camera's field of view. This approach leverages the intensity values of the white pixels in the binary image as weights during the averaging process. The higher the intensity value of a pixel, the stronger its influence on the calculated center of brightness. This weighting mechanism ensures that pixels with higher intensities, which typically correspond to the object's core or brighter regions, have a greater impact on determining the object's position. The step is implemented in python programming language using numpy library for matrix calculations as followed:

```
import numpy as np
column_sums = np.matrix(np.sum(BW, 0))
column_numbers = np.matrix(np.arange(640))
column_mult = np.multiply(column_sums, column_numbers)
total = np.sum(column_mult)
total_total = np.sum(np.sum(BW))
column_location = total / total_total
X_Location = column_location * cm_to_pixels
row_sums = np.matrix(np.sum(BW, 1))
row_sums = row_sums.transpose()
row_numbers = np.matrix(np.arange(480))
row_mult = np.multiply(row_sums, row_numbers)
total = np.sum(row_mult)
total_total = np.sum(np.sum(BW))
row_location = total / total_total
Y_Location = row_location * cm_to_pixels
```

The variable **cm_to_pixels** represents the conversion factor between centimeters (cm) and pixels in the image. It is used to convert the calculated column and row locations from pixel coordinates to real-world coordinates in centimeters. In the code, **cm_to_pixels** is defined as (36.0 / 640.0) by mounting the camera 80 cm above the workspace. Here, the value 36.0 represents the length in centimeters that corresponds to the width of the image frame, and 640.0 represents the width of the image frame in pixels. Dividing the length in centimeters by the width in pixels provides the conversion factor, giving the number of centimeters represented by each pixel in the image. By multiplying the column and row locations (in pixels) by the **cm_to_pixels** conversion

factor, the code converts these values to the corresponding real-world X and Y coordinates in centimeters.

The following is a breakdown of the localization calculations within the code step by step:

1. **Column Calculation (Object X location):**
    - **column_sums = np.matrix(np.sum(BW, 0))**: This line calculates the sum of pixel intensities along each column of the binary image. The resulting **column_sums** matrix represents the cumulative intensity values for each column.
    - **column_numbers = np.matrix(np.arange(640))**: This line generates a matrix **column_numbers** representing the column indices from 0 to 639.
    - **column_mult = np.multiply(column_sums, column_numbers)**: Here, the element-wise multiplication between **column_sums** and **column_numbers** is performed, resulting in a matrix **column_mult** that combines the cumulative intensity values with their respective column indices.
    - **total = np.sum(column_mult)**: This line sums up all the elements in the **column_mult** matrix, giving the total weighted sum of column positions.
    - **total_total = np.sum(np.sum(BW))**: The total number of white pixels in the binary image is computed by summing up all the elements in the binary image twice.
    - **column_location = total / total_total**: The weighted average column location is obtained by dividing the total weighted sum of column positions by the total number of white pixels.
    - **X_Location = column_location * cm_to_pixels**: Finally, the column location in pixels is converted to the corresponding real-world X-coordinate in centimeters by multiplying it with the **cm_to_pixels** conversion factor.
2. **Row Calculation (Object Y location):**
    - **row_sums = np.matrix(np.sum(BW, 1))**: This line calculates the sum of pixel intensities along each row of the binary image. The resulting **row_sums** matrix represents the cumulative intensity values for each row.

- **row_sums = row_sums.transpose()**: The **row_sums** matrix is transposed to ensure proper dimensions for subsequent calculations.
- **row_numbers = np.matrix(np.arange(480))**: This line generates a matrix **row_numbers** representing the row indices from 0 to 479.
- **row_mult = np.multiply(row_sums, row_numbers)**: The element-wise multiplication between **row_sums** and **row_numbers** is performed, resulting in a matrix **row_mult** that combines the cumulative intensity values with their respective row indices.
- **total = np.sum(row_mult)**: This line sums up all the elements in the **row_mult** matrix, giving the total weighted sum of row positions.
- **total_total = np.sum(np.sum(BW))**: Similar to the column calculation, the total number of white pixels in the binary image is computed.
- **row_location = total / total_total**: The weighted average row location is obtained by dividing the total weighted sum of row positions by the total number of white pixels.
- **Y_Location = row_location * cm_to_pixels**: Finally, the row location in pixels is converted to the corresponding real-world Y-coordinate in centimeters by multiplying it with the **cm_to_pixels** conversion factor.

By performing these calculations, the code determines the X Location and Y Location, representing the object's position within the camera's field of view in terms of the real-world coordinates in centimeters.

In the fifth step: Apply camera transformations to get object location in robot base frame, the transformations accounting for the camera's position and orientation relative to the robot base frame are calculated. By leveraging appropriate transformation matrices and calibration parameters, the object's camera-based coordinates are converted to the robot base frame's coordinates. These transformations enable the robot

arm to accurately perceive and interact with the object based on its location in the workspace. The step is implemented in python programming language using numpy library for matrix calculations as followed:

```
import numpy as np
# Rotation around y-axis by 180 degrees:
Rad_y = (180.0 / 180.0) * np.pi
RY = [[np.cos(Rad_y), 0, np.sin(Rad_y)],
      [0, 1, 0],
      [-np.sin(Rad_y), 0, np.cos(Rad_y)]]
#translation x_base by 13cm & in y_base by 20cm:
dO_C = [[13], [20], [0]]
H0_C = np.concatenate((RY, dO_C), 1)
H0_C = np.concatenate((H0_C, [[0, 0, 0, 1]]), 0)
# x y object location in camera coordinates
PC = [[X_Location],
      [Y_Location],
      [0],
      [1]]
# x y z in base coordinates
PO = np.dot(H0_C, PC)
#print(PO)
global XO
global YO
XO = PO[0]
YO = PO[1]
```

In our robot setup, we assume that the distance between the camera frame zero position within the workspace and the robot base frame zero position is 13 cm in the x-direction and 20 cm in the y-direction of the base frame. Also, we assume that the z-axes of camera and robot base frames are in the opposite directions. The following is a breakdown of the localization calculations within the code step by step:

1. **Rotation around y-axis by 180 degrees:**

- **Rad_y = (180.0 / 180.0) * np.pi**: Here, **Rad_y** is defined as the conversion factor for converting degrees to radians. It is set to π (pi) to represent a rotation of 180 degrees around the y-axis.
- **RY = [[np.cos(Rad_y), 0, np.sin(Rad_y)], [0, 1, 0], [-np.sin(Rad_y), 0, np.cos(Rad_y)]]**: The matrix **RY** represents the rotation matrix around the y-axis. It consists of three rows.

2. **Translation x_base by 13cm and in y_base by 20cm:**

   - **dO_C = [[13], [20], [0]]**: The matrix **dO_C** represents the translation vector from the origin of the camera frame to the origin of the base frame. It specifies the displacement in the x and y directions, with 13 cm in the x-direction and 20 cm in the y-direction. The third element is set to 0, assuming no translation in the z-direction.

3. **Transformation matrix from camera to base coordinates:**

   - **H0_C = np.concatenate((RY, dO_C), 1)**: The matrix **H0_C** is created by horizontally concatenating the rotation matrix **RY** and the translation vector **dO_C**. It represents the combined transformation of rotation and translation from the camera frame to the base frame.

   - **H0_C = np.concatenate((H0_C, [[0, 0, 0, 1]]), 0)**: The homogeneous transformation matrix **H0_C** is further extended by adding a row of [0, 0, 0, 1] at the bottom. This additional row represents the homogeneous coordinate [0, 0, 0, 1], accounting for translations in the homogeneous coordinate system.

4. **Object location in camera coordinates:**

   - **PC = [[X_Location], [Y_Location], [0], [1]]**: The matrix **PC** represents the object's location in the camera coordinates. It consists of four rows, where the first row represents the X_Location (column location) of the object, the second row represents the Y_Location (row location) of the object, the third row is set to 0 (assuming no displacement in the z-direction), and the fourth row is set to 1 to represent homogeneous coordinates.

5. **Object location in base coordinates:**

   - **PO = np.dot(H0_C, PC)**: The matrix **PO** is obtained by performing matrix multiplication between the camera transformation matrix **H0_C** and the

object's coordinates in the camera frame **PC**. This multiplication transforms the object's coordinates from the camera frame to the base frame.

6. **Object position variables:**

- **XO = PO[0]**: The variable **XO** represents the transformed X-coordinate of the object in the base frame. It is obtained by accessing the first row (index 0) of the matrix **PO**.
- **YO = PO[1]**: The variable **YO** represents the transformed Y-coordinate of the object in the base frame. It is obtained by accessing the second row (index 1) of the matrix **PO**.

Finally, by following these five steps of object localization and Camera transformation, the system captures and processes the RGB frames, analyzes the difference between the frames, extracts the object's position using the center of brightness method, and transforms the object's camera-based coordinates to the robot base frame. This enables the robot arm to effectively interact with the object, facilitating various collaborative tasks.

## Feature-based object detection

Feature-based object detection is a fundamental technique in computer vision that focuses on identifying and matching distinctive visual features, also known as keypoints, between an object of interest and a scene. These keypoints are locations in the image that exhibit strong variations in intensity or texture, making them highly distinguishable. The SIFT algorithm, which stands for Scale-Invariant Feature Transform, is a popular choice for feature extraction due to its robustness against changes in scale, rotation, and illumination.

The SIFT algorithm works by detecting keypoints at multiple scales and orientations within an image. These keypoints are then described by a set of descriptors that encode their appearance and local neighborhood information. In the feature detection phase, SIFT identifies potential keypoints based on their scale-space extrema

and eliminates unstable keypoints using a combination of scale-space theory and local image gradient properties.

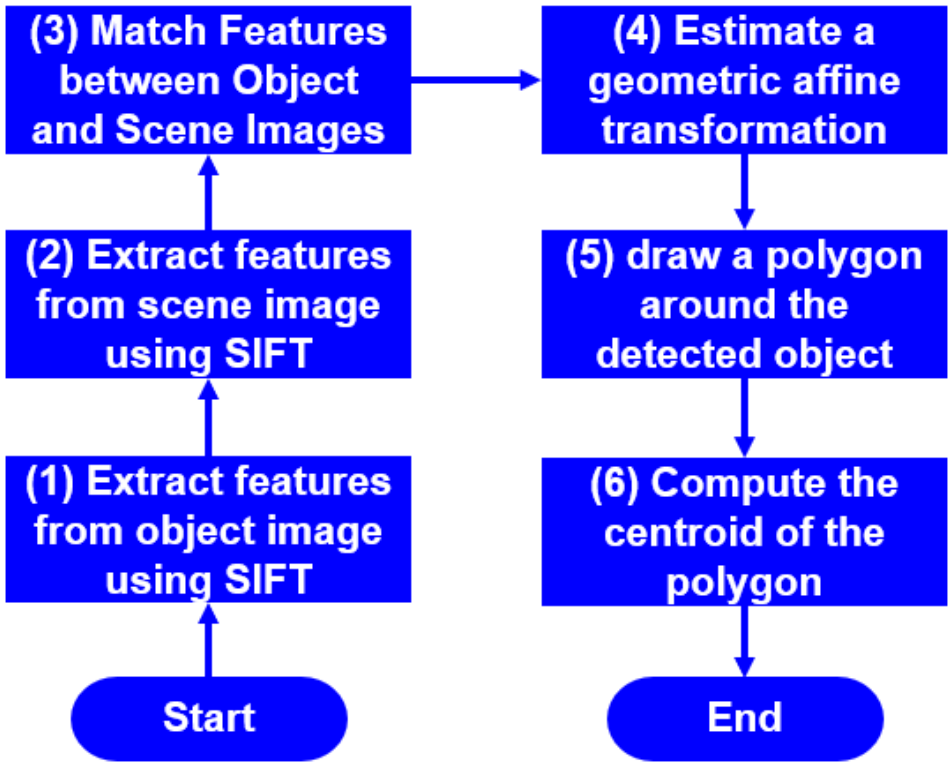
Once the keypoints and descriptors are extracted from the object image and the scene image, they can be matched to establish correspondences between the two. The matching process aims to find the best matches by comparing the similarity of the descriptors. This is typically done using distance metrics such as Euclidean distance or Hamming distance. By filtering the matches based on a distance threshold or using more advanced techniques like the ratio test, the algorithm identifies the most reliable matches.

After obtaining the matches, a geometric transformation is estimated to align the object with the scene. This transformation can involve translation, rotation, and scaling. In the case of SIFT, an affine transformation is commonly used. By applying this transformation, the object can be overlaid onto the scene, providing the basis for further analysis and localization.

One key advantage of feature-based object detection is its ability to handle occlusion, cluttered backgrounds, and viewpoint variations. It allows for robust and accurate detection even in challenging scenarios.

The feature-based object detection part plays a vital role in the project by enabling the detection and localization of specific objects within cluttered scenes full of different objects based on their visual features. By leveraging the power of computer vision techniques, this part provides a solution to the challenging task of identifying objects of interest. This part utilizes the SIFT (Scale-Invariant Feature Transform) algorithm for feature extraction and matching, allowing it to find corresponding keypoints between an object image and a scene image. Through the estimation of a geometric transformation, the code accurately aligns the object in the scene, enabling the computation of the object's centroid.

The following diagram illustrates the steps involved in feature-based object detection process:




*Feature-based object detection algorithm4 Figure*

The algorithm is implemented in python programming language using numpy library for as followed:

In the first step, the features are extracted from the object image using SIFT. The object image is loaded, converted to grayscale, and SIFT keypoints and descriptors are computed:

```
# (1) Extract features from object image using SIFT
box_image = cv2.imread('object.jpg')
box_gray = cv2.cvtColor(box_image, cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create()
box_keypoints, box_descriptors = sift.detectAndCompute(box_gray, None)
```

In the second step, features from the scene image are extracted using SIFT. The scene image is loaded, converted to grayscale, and SIFT keypoints and descriptors are computed:

```
# (2) Extract features from scene image using SIFT
scene_image = cv2.imread('scene.jpg')
scene_gray = cv2.cvtColor(scene_image, cv2.COLOR_BGR2GRAY)
scene_keypoints, scene_descriptors = sift.detectAndCompute(scene_gray, None)
```

In the third step, features between the object image and the scene image are matched. A brute force matcher (BFMatcher) is used to find matching descriptors between the two sets of descriptors. The BFMatcher works by comparing each descriptor in one set with all descriptors in the other set and finding the best matches based on a distance measure. In the code, the BFMatcher is initialized with the Euclidean distance (cv2.NORM_L2) and the crossCheck parameter set to True. The crossCheck parameter ensures that the matching is done symmetrically, meaning a descriptor from the object image must match a descriptor from the scene image, and vice versa, to be considered a valid match:

```
# (3) Match Features between Object and Scene Images
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
matches = bf.match(box_descriptors, scene_descriptors)
```

In the fourth step, the geometric transformation (affine transformation) between the object and the scene is estimated using the matched keypoints. The keypoints from the object and scene images are used to estimate the transformation matrix. To estimate the affine transformation, the matched keypoints are used as corresponding points. The coordinates of these keypoints in the object image and the scene image are collected and passed as input to the **cv2.estimateAffine2D()** function. This function calculates the affine transformation matrix that best aligns the keypoints of the object image with the keypoints of the scene image. The resulting transformation matrix, denoted as **transform**, represents the translation, rotation, and scaling required to align the object with the scene. This matrix can then be used to transform other geometric entities, such as drawing a polygon around the detected object or computing its centroid:

```
# (4) Estimate a geometric affine transformation
box_points = np.float32([box_keypoints[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
scene_points = np.float32([scene_keypoints[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
transform, inlier_mask = cv2.estimateAffine2D(box_points, scene_points)
```

In the fifth step, a polygon is drawn around the detected object in the scene image. The box_polygon represents the coordinates of a predefined polygon in the object image. It is defined as a float32 numpy array, with each row representing the (x, y) coordinates

of a vertex of the polygon. The **.reshape(-1, 1, 2)** operation is used to reshape the **box_polygon** array to the desired shape before applying the transformation. In this case, the **box_polygon** is reshaped to have dimensions **(4, 1, 2)**. Here's what each dimension represents: The first dimension **(4)** represents the number of vertices in the polygon, which is 4 in this case. The second dimension **(1)** represents a single polygon, as we are dealing with one object. The third dimension **(2)** represents the (x, y) coordinates of each vertex. By reshaping the **box_polygon** in this way, we ensure that it is compatible with the transformation operation and can be passed as input to the **cv2.transform()** function. To transform the box_polygon coordinates from the object image to the scene image, the cv2.transform() function is used. The cv2.transform() function takes the box_polygon and the affine transformation matrix transform as inputs and applies the transformation, resulting in the transformed polygon coordinates new_box_polygon. To visualize the polygon in the scene image, a copy of the scene image is created as scene_image_with_box. The cv2.polylines() function is then used to draw the polygon on scene_image_with_box. The np.int32() function is used to convert the transformed polygon coordinates to integer values as required by the cv2.polylines() function. The color (0, 255, 255) represents the color of the polygon (in this case, yellow), and the thickness parameter sets the thickness of the polygon lines. After executing this code snippet, the scene_image_with_box will show the scene image with a polygon drawn around the detected object:

```
# (5) Draw a polygon around the detected object
box_polygon = np.float32([[0, 0], [box_gray.shape[1], 0],
                          [box_gray.shape[1], box_gray.shape[0]],
                          [0, box_gray.shape[0]]]).reshape(-1, 1, 2)
new_box_polygon = cv2.transform(box_polygon, transform)
scene_image_with_box = scene_image.copy()
cv2.polylines(scene_image_with_box, [np.int32(new_box_polygon)],
              True, (0, 255, 255), thickness=3)
```

In the sixth step, the centroid of the polygon is computed using the moments of the transformed polygon. We first calculate the moments of the transformed polygon using the **cv2.moments()** function. Once we have the moments, we can extract the centroid coordinates. The centroid of a shape is the center of mass, which can be

calculated as the average position of all the points in the shape. In our case, the centroid coordinates correspond to the average x and y positions of the points in the polygon. The centroid coordinates can be computed by dividing the sum of the x moments (**m10**) and y moments (**m01**) by the zeroth moment (**m00**). The zeroth moment represents the total area or mass of the shape. The centroid coordinates are then converted from pixel units to centimeters:

```
# (6) Compute the centroid of the polygon
M = cv2.moments(np.int32(new_box_polygon))
centroid_x = int(M['m10'] / M['m00'])
centroid_y = int(M['m01'] / M['m00'])
cm_to_pixels = 36.0 / 640.0
centroid_x_cm = centroid_x * cm_to_pixels
centroid_y_cm = centroid_y * cm_to_pixels
```

Finally, by following these six steps, the system performs feature-based object detection using the SIFT algorithm. It extracts features from both the object image and the scene image, matches these features to establish correspondences, and estimates a geometric transformation between the two. This transformation allows for accurate alignment of the object in the scene. A bounding box is then drawn around the detected object, providing a visual representation of its location. Additionally, the centroid of the object is computed using moments, allowing for precise localization. Overall, this approach enables the detection and localization of specific objects within cluttered scenes based on their visual features.

## AI-based human detection

The AI-based human detection algorithm plays a crucial role in ensuring safety in collaborative workspaces. By leveraging computer vision techniques, this part of the system enables real-time detection and localization of humans within the robot's workspace. This capability is essential for implementing safety measures and ensuring that the cobot operates within ISO/TS 15066 safety standards. ISO/TS 15066 is a technical specification that provides guidance on the safety requirements for collaborative robot systems. One of the key concepts emphasized in ISO/TS 15066 is the implementation of

safety-rated monitored stop functions. This function allows the collaborative robot (cobot) to detect the presence of a human in its workspace and automatically stop its motion to ensure the safety of the human. The AI-based human detection algorithm aligns with the safety-rated monitored stop concept. By using the YOLO algorithm to detect humans in real-time, the system is able to identify the presence of humans within the cobot's workspace. When a person is detected, the system triggers a safety response by sending a signal (in our case, using serial communication) to the cobot, indicating the need to stop. This implementation is crucial for complying with ISO/TS 15066 guidelines, as it helps ensure that the cobot operates in a safe manner around human operators. The safety-rated monitored stop function, enabled by the human detection algorithm, minimizes the risk of collisions or other accidents between the cobot and humans in the workspace. By incorporating this AI-based human detection system into the cobot's safety framework, the cobot can effectively adhere to ISO/TS 15066 guidelines. This not only enhances the safety of the collaborative workspace but also promotes the successful integration of human-robot collaboration, providing a safe and efficient working environment.

The following diagram illustrates the safety system and human detection process:

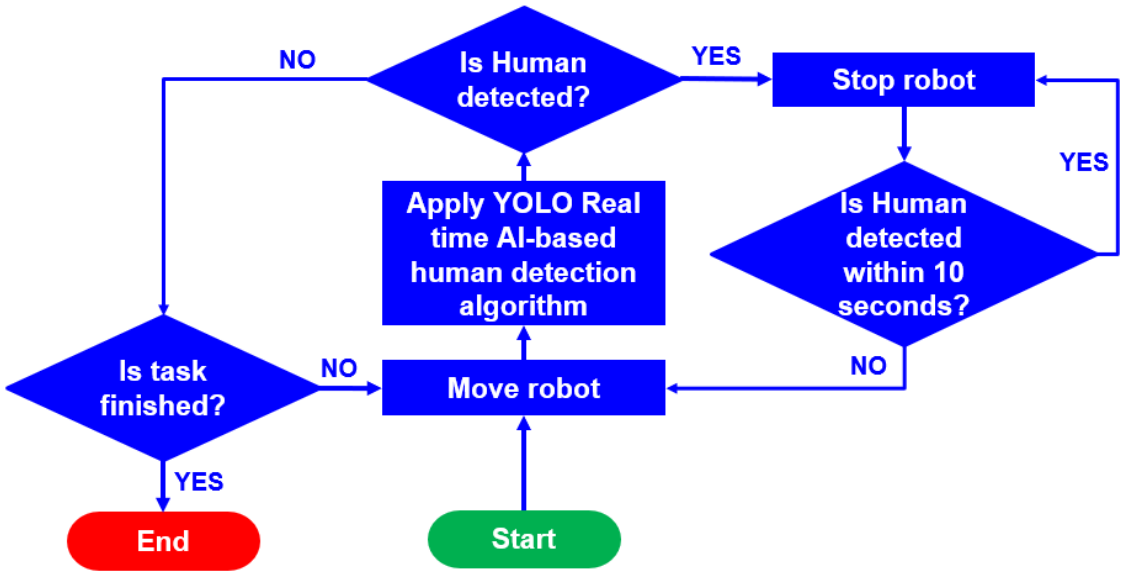



*Figure 5 Safety Rated Monitored Stop Algorithm*

When the robot starts moving to perform a specific task, the YOLO algorithm starts checking for human presence in real time, and as long as there is no human detected, the robot will continue to move until finishing the task. If the YOLO algorithm detects a human within the workspace, the robot will stop and start to monitor the human presence for ten seconds. If the system continues to detect the human during the 10 seconds period, the system will reset the time period and start the ten second monitoring from the beginning, thus making the absence of human detections for ten continuous seconds the only case for the robot to continue its movement to ensure safe operations.

The model used for human detection is The YOLO (You Only Look Once) V3 pretrained model which is a deep learning-based object detection model that can detect objects in real-time with high accuracy. It is a popular and widely used model due to its efficiency and ability to process images in real-time. The YOLO V3 model is built on a deep convolutional neural network (CNN) architecture. It divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell. Unlike other object detection models that use region proposal methods, YOLO V3 performs object detection in a single pass through the network, which makes it extremely fast. The YOLO V3 model has been trained on a large dataset with various object classes, including people, cars, animals, and everyday objects. It can detect multiple objects in an image and assign class labels and confidence scores to each detected object.

To use the YOLO V3 model in python and OpenCV library, the following steps are typically performed:

1. Load the Model: The YOLO V3 model consists of a configuration file (**.cfg**) that defines the network architecture and a set of pre-trained weights (**.weights**) that contain learned parameters. These files are loaded using the **cv2.dnn.readNetFromDarknet()** function:

```
yolo_model = cv2.dnn.readNetFromDarknet('model/yolov3.cfg','model/yolov3.weights')
```

2. Preprocess the Input: The input image is preprocessed to meet the requirements of the YOLO V3 model. This involves resizing the image to a fixed size, converting it to a blob, and normalizing pixel values:

```
img_blob = cv2.dnn.blobFromImage(img_to_detect, 0.003922, (320, 320), swapRB=True, crop=False)
```

3. Perform Object Detection: The preprocessed image is passed through the YOLO V3 model to obtain object detection results. The model predicts bounding boxes, class probabilities, and confidence scores for each detected object:

```
# input preprocessed blob into model and pass through the model
yolo_model.setInput(img_blob)
# obtain the detection layers by forwarding through till the output layer
obj_detection_layers = yolo_model.forward(yolo_output_layer)
```

4. Filter Detections: Detections with confidence scores below a certain threshold are filtered out to remove weak detections:

```
# Loop over each detection
for detection in obj_detection_layers:
    # Extract confidence and class predictions
    scores = detection[5:]
    class_id = np.argmax(scores)
    confidence = scores[class_id]

    # Filter detections by confidence threshold
    if confidence > confidence_threshold:
        # Process the detection
        # ...
```

5. Apply Non-Maximum Suppression (NMS): Non-Maximum Suppression is applied to remove overlapping bounding boxes and retain only the most confident and non-overlapping detections:

```
max_value_ids = cv2.dnn.NMSBoxes(boxes_list, confidences_list, 0.5, 0.4)
```

After having the real time detections working, a human presence check is done during the cobot motion. If a human is detected, the system sends a signal to the low level control module to stop the motors immediately. In our system, when the low level control module receives I letter through the serial communication, it stops the motors:

```python
if predicted_class_label[0:6] =='person':
    serialInst.write('I'.encode('utf-8'))
```

The low level control system checks for incoming data from a serial connection. If the incoming data is 'I', it stops the motors, waits for 10 seconds, and then resumes motor movement unless another 'I' is received during the waiting period, which restarts the loop. The Arduino board controls motors that need to be stopped when a person is detected. The Python code can send the 'I' signal to the Arduino board when a person is detected and then wait for a specific duration before resuming motor movement.

This integration allows the human detection algorithm in Python to communicate with an external device, such as an Arduino board, to control motor behavior based on the detection results. For example, if a person is detected, the Python code can send the signal to stop the motors, ensuring safety in the presence of a human, and resume motor movement after a specific duration or based on additional signals from the Arduino board:

```
// Check for incoming data and stop the motors if an interrupt is received
 if (Serial.available() > 0) {
   char incoming = Serial.read();
   if (incoming == 'I') {
     stopMotors();
     unsigned long startTime = millis();
     unsigned long duration = 10000; // 10 seconds duration
     while (millis() - startTime < duration) {
       if (Serial.available() > 0 && Serial.read() == 'I') {
         // Restart the loop from the beginning
         startTime = millis(); // Reset the start time
        continue; // Restart the loop
       }
     }
     // Resume motor movement
     continue;
   }
 }
```

### 4.1.3 Motion planning and control module

#### Motion planning and control architecture

The following diagram shows the Motion planning and control module architecture and how it is connected with the rest of the parts in an abstract way that illustrates the data flow and sequence of operations.

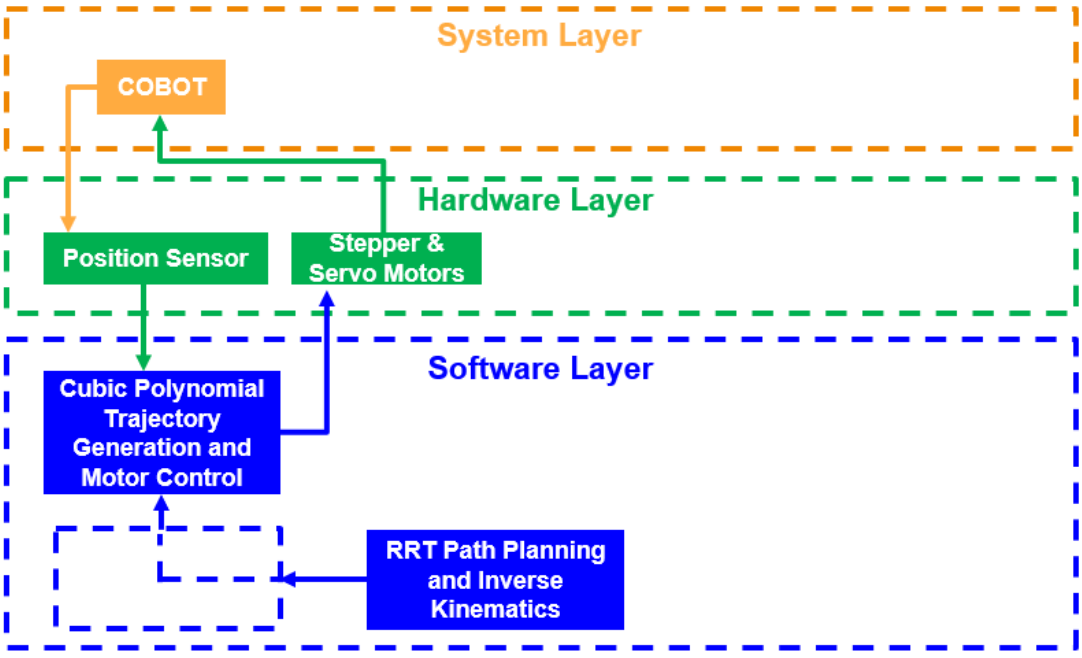



*Figure 6 Motion planning and control architecture*

The path planning module determines the via points between the initial and goal position to generate a free collision path for the end effector. Inverse kinematics module

gives the joint angle configurations to achieve each via point. Configurations of each two adjacent via point are defined as initial and final joint angles and set to the trajectory generation and motor control module. The appropriate motion commands are sent to the motors to obtain the desired position of end effector. The second joint motor (servo motor) involves a potentiometer that measures the current position and sends a feedback signal to the controller.

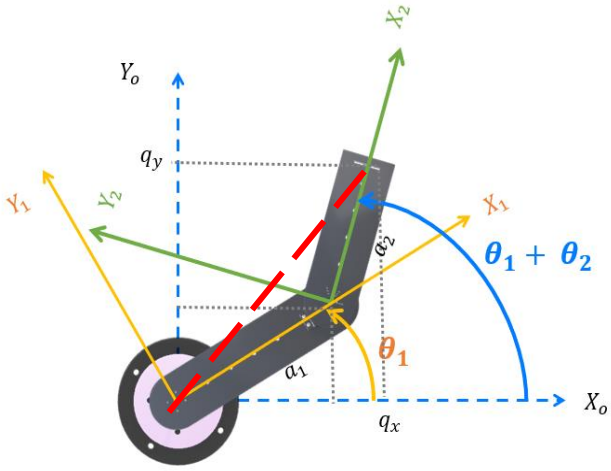
## Inverse Kinematics

The problem of inverse kinematics of the two-link robot is straightforward. It is to determine the required two joint angles that move the end-effector to a predefined desired position. We are concerned with the motion in the XY plane of the base frame; therefore, we can use the geometrical solution. For manipulators with more complex geometries, it is easier to use the algebraic solution with known joint parameters, as the geometrical solution could be complex or even impossible. Appendix A provides additional information on the algebraic solution for inverse kinematics, including an explanation of the Denavit-Hartenberg convention and its application to deriving equations for joint angles.

From the following geometry in fig. and the geometric solution of the joint angles $\theta_1, and\ \theta_2\ are$:

$$\theta_2 = \ cos^{-1}(\frac{q_x^2 + q_y^2 - a_1^2 - a_2^2}{2a_1a_2})$$

$$\theta_1 = tan^{-1}(\frac{q_y}{q_x}) - tan^{-1}(a_1 + a_2cos\ (\theta_2))$$




As shown, we need predefined constants and variables to obtain the desired joint angles. We consider that the robot has rigid links with constant lengths and offsets. The predefined constants are:

$$a_1 = 0.20075\ m$$

$$a_2 = 0.159\ m$$

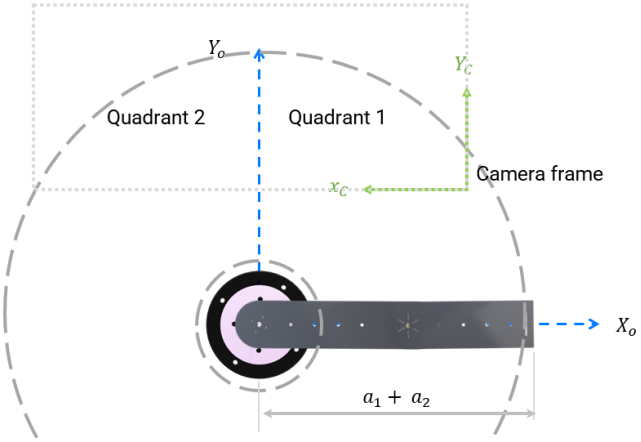
The predefined variables $q_x\ and\ q_y$ are obtained either through camera analysis or manually using the graphical user interface (GUI).

The inverse kinematics solution is implemented through module (inverse_k_2D) as follows:

```
def calculate_inverse_kinematics(end_effector_pose):
    #Define Position and Orientation of the end-effector
    u_x = end_effector_pose[0] #Oreintation between X of EE and X of Base
    u_y = end_effector_pose[1] #Oreintation between X of EE and Y of Base
    u_z = end_effector_pose[2] #Oreintation between X of EE and Z of Base
    #The oreintation is added as a generalized form to add more flexibility to the module
    #However this code is built for our 2DOF Cobot
    q_x = end_effector_pose[3] #X position of the end-effector
    q_y = end_effector_pose[4] #Y position of the end-effector
    q_z = end_effector_pose[5] #Z position of the end-effector

    #Calculate theta_2
    theta_2_cos = (q_x**2 + q_y**2 - a1**2 - a2**2) / (2*a2*a1) #cos(theta_2)
    theta_2 = np.arccos(theta_2_cos) #rad

    #the following print is to test the module and display results clearly
    print("theta 2 is {}".format(np.degrees(theta_2)))

    #Calculate theta_1
    b1 = np.arctan(q_y / q_x) #frist term
    b2 = np.arctan((a2 * np.sin(theta_2)) / (a1 + a2 * np.cos(theta_2))) #second term
    theta_1 = b1 - b2 #read
    if q_x < 0 and q_y > 0:
        theta_1 = theta_1 + np.pi #if the angle lies in the second quarter add 180 deg

    #the following print is to test the module and display results clearly
    print("theta 1 is {}".format(np.degrees(theta_1)))

    #store joint angles in an array
    joint_angles = np.array([[theta_1],
                             [theta_2]
                            ])
    #convert joint angles into degree
    joint_angles_degree = np.degrees(joint_angles)
    return joint_angles_degree
```

The module acquires six input parameters that represent the robot variables. The first three parameters correspond to the orientation of the end-effector's X-axis with respect to the base axes. The remaining three parameters correspond to the position of the end-effector in the X, Y, and Z axes of the base frame. This generalized parameterization enables the module to accommodate additional parameters in cases where the robot possesses more than three degrees of freedom. As the complexity of the inverse kinematics problem increases with the number of degrees of freedom, this

generalized form enhances the module's versatility and adaptability to varying robotic systems. NumPy package is imported to the module as we have operation on arrays. For more information about NumPy package and its application, please check Appendix B.




Note that the desired position of the end-effector always lies within either quadrant one or quadrant two as shown in fig.. The solution for the second joint angle is obtained using the arccos function, which provides a unique solution in either quadrant one or quadrant two. To determine the first joint angle, we utilize the arctan function, which yields a solution in quadrant one if the result is positive, or quadrant four if negative. Consequently, we incorporate a conditional statement in line 63 to verify whether q_x is negative and q_y is positive; if so, we add 180 degrees to the solution.

It is important to note that we are only concerned with the XY position, and not the Z axis. To grasp an object with known dimensions in a SCARA configuration, we can simply use a prismatic joint with a constant parameter, without needing to solve for the inverse kinematics of all three degrees of freedom. Furthermore, the previous two-link configuration can be used to grasp an object located at the same position in Z as the end-effector.

## Trajectory Generation

The approach which involves sending a single motion command with desired angles, constant velocities, and acceleration over time can result in sudden shocks when the robot comes to a stop. To avoid this issue, a more sophisticated approach involves defining the desired positions, velocities, and acceleration of the robot over time. By generating a trajectory that specifies the robot's motion profile in this way, it is possible to achieve smoother and more controlled movements, while also reducing the risk of

damage to the robot or its surroundings. Our project utilizes a joint-space implementation of the cubic polynomial trajectory. For further details regarding the joint space, please refer to Appendix A, Section 2. With this joint-space scheme, the inverse kinematics is only implemented once, or a few times if there are predefined via points. This approach enables faster processing times compared to the task-space scheme.

Our problem is to move the end-effector form predefined initial position to a desired final goal position $\begin{bmatrix} q_x \\ q_y \end{bmatrix}$ in a certain amount of time $t_f$. The initial position of the end effector is $\begin{bmatrix} q_x = 0 \\ q_y = 0 \end{bmatrix}$, however, for the joint-space scheme we use joint angles that we define initial joint angles as: $\begin{bmatrix} \theta_{1,o} = 0 \\ \theta_{2,o} = 0 \end{bmatrix}$ and the desired joint angles, which are obtained from the inverse kinematics, as: $\begin{bmatrix} \theta_{1,f} \\ \theta_{2,f} \end{bmatrix}$. What is required is a function for each joint that define positions, velocities and acceleration at each instant of time $t$ between the initial time instant $t_o = 0$ and the final time instant $t_f$.

Now, we have two constraints of motion for each joint which are the initial and final positions, we need four constraints as minimum to obtain a cubic polynomial function. The cubic polynomial function for each joint is represented by:

$$\theta_i(t) = a_{i,0} + a_{i,1}t + a_{i,2}t^2 + a_{i,3}t^3$$

Therefore, the velocity and acceleration functions are:

$$\dot{\theta}_i(t) = a_{i,1} + 2a_{i,2}t + 3a_{i,3}t^2$$

$$\ddot{\theta}_i(t) = 2a_{i,2} + 6a_{i,3}t$$

An additional two constraints for each joint are that the function be continuous in velocity which in this case means that the initial and final velocities are zero $\begin{bmatrix} \dot{\theta}_{1,0} = 0 & \dot{\theta}_{1,f} = 0 \\ \dot{\theta}_{2,0} = 0 & \dot{\theta}_{2,f} = 0 \end{bmatrix}$.

From the previous constraints we obtain the function parameter as:

$$a_{i,0} = 0 \ , \ a_{i,1} = 0 \ , \ a_{i,2} = \frac{3}{t_f^2}\theta_{i,f} \ , \ a_{i,3} = -\frac{2}{t_f^3}\theta_f$$

The trajectory generation functions are implemented within the microcontroller (ATMega32) uploaded code. The trajectory is represented as a struct (Trajectory) that contains joint angles (q), joint velocities (qdot) and joint accelerations (qddot) as attributes.

```
20  //Define trajectory struct with motion attributes
21  struct Trajectory {
22    float q;
23    float qdot;
24    float qddot;
25  };
```

Trajectory generation is implemented through a function that has a return data type of the struct. The function arguments are (an array of initial joint angles q_0[], an array of final joint angles q_f[], an array of initial velocities qdot_0[], an array of final velocities qdot_f[], and the certain amount of time for joints to move in t_f, as well as, the number of degrees of freedom which represents the number of joints.

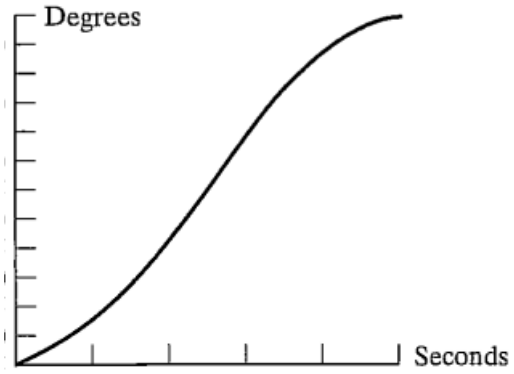
```
27  //Defining a function to generate the cubic polyomial trajectory
28  Trajectory cubic_polynomial_trajectory(float q_0[],
29   float q_f[], float qdot_0[], float qdot_f[], int t_f, int DOF) {
30    float a0[DOF];
31    float a1[DOF];
32    float a2[DOF];
33    float a3[DOF];
```

The function parameters for each joint by obtained by two iterations (DOF = 2) as follows:

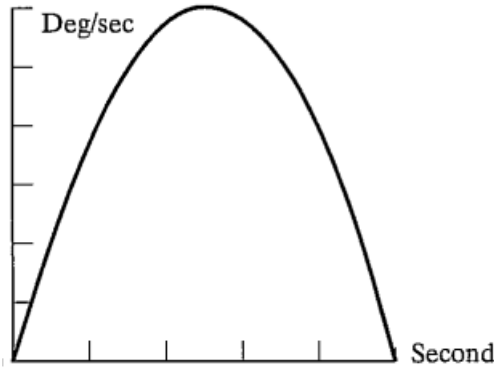
```
37    for (int i = 0; i < DOF; i++) {
38      a0[i] = q_0[i];
39      a1[i] = qdot_0[i];
40      a2[i] = (3.0 / pow(t_f, 2)) * (q_f[i] - q_0[i]);
41      a3[i] = -(2.0 / pow(t_f, 3)) * (q_f[i] - q_0[i]);
42    }
```

Motion Profiles of cubic polynomial trajectory

**Position Profile**




**Velocity Profile**




Acceleration Profile

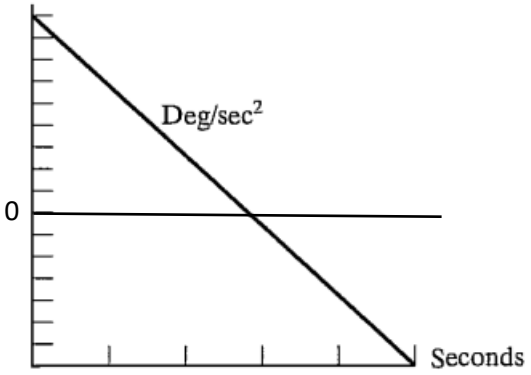



*Figure 7 Motion profiles of cubic polynomial trajectory*

The trajectory is implemented to move the motors in the Arduino code as follows:

```
void moveOnTrajectoryWaypoints(){
  for (int t = 0; t <= t_f; t++) {
    float desired_pos_1 = qOutPut[0][t];
    float desired_vel_1 = qdotOutPut[0][t];
    float desired_acc_1 = qddotOutPut[0][t];
    float desired_acc_2 = qddotOutPut[1][t];
    float desired_pos_2 = qOutPut[1][t];
    float desired_vel_2 = qdotOutPut[1][t];
    current_angle_2 = joint_2.read();
    float angular_distance_1 = desired_pos_1 - current_angle_1;
    float angular_distance_2 = desired_pos_2 - current_angle_2;
    int delay_time_1 = delay_velocity_stepper(angular_distance_1, desired_vel_1, desired_acc_1);
    delay_time_1 < min_delay ? delay_time_1 = min_delay : Serial.println("Velocity is within range");
    int steps_1 = convert_to_steps(desired_pos_1, step_); // convert desired position to steps
    int delay_time_2 = delay_velocity_servo(angular_distance_2, desired_vel_2, desired_acc_2);
    joint_move(steps_1, desired_pos_2, delay_time_1, delay_time_2);
    current_angle_1 = current_angle_1 + desired_pos_1;
    current_angle_2 = joint_2.read();
    Serial.print("done waypoint ");
    Serial.println(t);

  }
}
```

## Path Planning

Path planning involves finding an optimal path from a starting point to a goal point, taking into account various constraints and obstacles. It finds out the shortest free-obstacle path to the goal after receiving data from the perception module **(how the environment looks like).**

Motion planning has three categories:

- **Global planning** : a path planning algorithm ( RRT , A*)
- **Local Replanning** : Trajectory Generation
- **Behavior planning** : high level decision planning

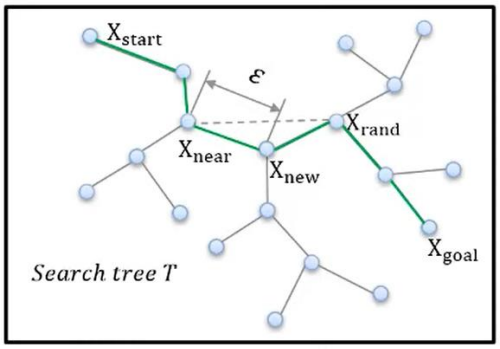
Global planning (Path planning algorithm)

There are two types of planning algorithm:

- Search/Grid based such as: A* algorithm
- Sampling based such as: Probabilistic Roadmap algorithm (PRM) or Rapidly exploring Random Tree (RRT).

RRT Rapidly exploring random tree

- Sampling algorithm
- Nodes -> state **(position & orientation)**
- RRT creates tree incrementally in the state space with randomly sampled nodes (states)
- It gives a valid path but not necessarily the shortest



*State Space X*

The modified version RRT* gives the shortest path

To implement any of the path planning algorithm we need to determine the configuration space of joint angles. The configuration space represents all the possible configurations of a robot arm, including the joint angles and other parameters that define the position and orientation of the end-effector. To plan a collision-free path for a robot arm, we need to consider both the configuration space and the obstacles in the environment. This involves generating a representation of the obstacles in the environment, such as a point cloud or a mesh, and then checking whether a given configuration of the robot arm would result in a collision with any of the obstacles.

Generating the configuration space of the robot

In the config_space module function collision_check takes in the joint angles theta1 and theta2 of the two-link robotic arm, as well as the positions and sizes of

obstacles in the environment. It then computes the positions of the robotic arm links and checks whether they intersect with any of the obstacles.

```python
def collision_check(theta1, theta2, obstacle_positions, obstacle_sizes):
    # Define the robot arm links
    link1_length = 0.20075
    link2_length = 0.159

    # Compute the end effector position
    x1 = link1_length * np.cos(theta1)
    y1 = link1_length * np.sin(theta1)
    x2 = x1 + link2_length * np.cos(theta1 + theta2)
    y2 = y1 + link2_length * np.sin(theta1 + theta2)

    # Create the polygons for the arm links
    link1_vertices = np.array([[0, 0], [link1_length, 0], [x1, y1]])
    link2_vertices = np.array([[x1, y1], [link2_length, 0], [x2, y2]])

    link1_polygon = Polygon(link1_vertices)
    link2_polygon = Polygon(link2_vertices)

    # Check for collision with each obstacle
    for obstacle_position, obstacle_size in zip(obstacle_positions, obstacle_sizes):
        obstacle_rectangle = Polygon([
            (obstacle_position[0] - obstacle_size[0] / 2, obstacle_position[1] - obstacle_size[1] / 2),
            (obstacle_position[0] - obstacle_size[0] / 2, obstacle_position[1] + obstacle_size[1] / 2),
            (obstacle_position[0] + obstacle_size[0] / 2, obstacle_position[1] + obstacle_size[1] / 2),
            (obstacle_position[0] + obstacle_size[0] / 2, obstacle_position[1] - obstacle_size[1] / 2)
        ])

        if link1_polygon.intersects(obstacle_rectangle) or link2_polygon.intersects(obstacle_rectangle):
            return True

    return False
```

The function generate_configuration_space takes in the ranges of joint angles theta1_range and theta2_range, as well as the positions and sizes of obstacles in the environment. It then generates a binary configuration space matrix that represents the collision-free configurations of the robotic arm, using the collision_check function to check for collisions at each configuration.

```python
def generate_configuration_space(theta1_range, theta2_range, obstacle_positions, obstacle_sizes, resolution=1):
    nrows = len(theta2_range)
    ncols = len(theta1_range)

    cspace = np.ones((nrows, ncols), dtype=bool)

    for i, theta2 in enumerate(theta2_range):
        for j, theta1 in enumerate(theta1_range):
            if collision_check(theta1, theta2, obstacle_positions, obstacle_sizes):
                cspace[i, j] = False

    return cspace
```

**Implementing RRT algorithm in the python environment**

To implement the RRT algorithm, we defined two classes. One class is the tree_node which has (locationX, locationY, parent, children) as attributes. The other class is RRTAlgorithm which has attributes of randomTree, goal, nearest_node, iterations, grid, stepSize,

path_distance, nearest_distance, num_waypoints, waypoints. Also, it involves methods that represent the steps of te RRT algorithm (addChild, sampleApoint, steerTopoint, isInObstacle, unitVector, findNearest, distance, goalFound, resetNearestValues and moveViaWaypoints)

```
#tree class
class tree_node():
    def __init__(self, locationX, locationY):
        self.locationX = locationX
        self.locationY = locationY
        self.children = []
        self.parent = None


class RRTAlgorithm():
    def __init__(self, start, goal, numIterations, grid, stepSize):
        self.randomTree = tree_node(start[0], start[1])
        self.goal = tree_node(goal[0], goal[1])
        self.nearest_node = None
        self.iterations = min(numIterations, 400)
        self.grid = grid
        self.stepSize = stepSize
        self.path_distance = 0
        self.nearest_distance = 1000
        self.num_waypoints = 0
        self.waypoints = []
```

The used packages are NumPy, matplotlib and random. For more information, please refer to Appendix section 3. We also imported the config_space module that generates the configuration space with predefined obstacle position.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import rcParams
import random
from config_space import *
```

We define the parameters the class would get as its attributes:

```
#load the grid, set start and goal <x, y> positions, number of iterations, step size
grid = np.load('configuration_space.npy')
start = np.array([0, 0])
goal = np.array([0.0, 150])
numIterations = 400
stepSize = 20
link1_length = 0.20075
link2_length = 0.159

goalRegion = plt.Circle((goal[0], goal[1]), stepSize, color='b', fill = False)
```

We declare an object (rrt) of class RRTAlgorithm and start iterations:

```
rrt = RRTAlgorithm(start, goal, numIterations, grid, stepSize)
# plt.pause(2)

#iterate
for i in range(rrt.iterations):
    #Reset nearest values, call the resetNearestValues method
    rrt.resetNearestValues()
    print("Iteration: ",i)
    #algorithm begins here-----------------------------------------------------------------------------

    #sample a point
    point = rrt.sampleAPoint()
    print(point)
    #find the nearest node w.r.t to the point
    rrt.findNearest(rrt.randomTree, point)
    new = rrt.steerToPoint(rrt.nearest_node, point) #steer to a point, return as 'new'
    #if not in obstacle
    if not rrt.isInObstacle(rrt.nearest_node, new):
        #add new to the nearestnode (addChild)
        rrt.addChild(new[0], new[1])

        plt.pause(0.10)
        plt.plot([rrt.nearest_node.locationX, new[0]], [rrt.nearest_node.locationY, new[1]],'go', linestyle="--")
        #if goal found (new is within goal region)
        if (rrt.goalFound(new)):
            #append goal to path
            rrt.addChild(rrt.goal.locationX, rrt.goal.locationY)

            break
```

## Motor control

In robotic manipulators, motors play a vital role in controlling the position and velocity of the joints, which ultimately determine the position and orientation of the end-effector. A microcontroller or computer is typically responsible for sending signals to the motors to adjust their speed and direction of rotation. In the case of a two-link robotic manipulator, each joint is typically controlled by a specific type of motor. For example, the first joint utilize a stepper motor NEMA 23, while the second joint use a servo motor MG996 . To achieve the desired motion, the MC ATmega32 sends motion commands to the stepper motor through a TB6600 drive, while sending commands to the servo motor

directly through Pulse Width Modulation (PWM). The control of the stepper motor NEMA 23 is achieved using open-loop control techniques, as it has precise control of position and ability to move very accurately in small, incremental steps. Whereas the servo motor MG996 can provide positional feedback. It contains a built-in potentiometer. As the motor rotates, the potentiometer generates a voltage signal that is proportional to the rotation angle of the motor shaft. This voltage signal can be read by the Arduino microcontroller.

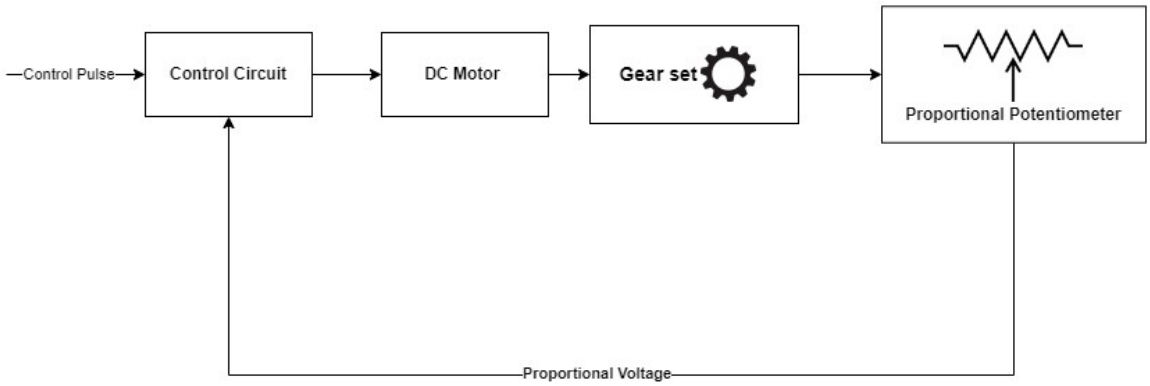



*Figure 8 Servo motor MG996 diagram*

To control the stepper and servo motors in a robotic manipulator, a common approach is to use a combination of vision and inverse kinematics modules implemented in a Python environment. The vision module is responsible for acquiring real-time image data from the camera, processing it, and extracting the relevant information about the environment (the position of the object and the motion interruption; person detected). The inverse kinematics module is responsible for calculating the joint angles required to achieve a specific position or trajectory of the end-effector, based on the information provided by the vision module.

To communicate with the stepper and servo motors, the Arduino board is used as an interface between the Python environment and the motors. The Python environment sends desired joint angles to the Arduino through the serial port, which in turn sends the appropriate signals to the motors to control their speed, direction, and position. Through

the Arduino the physical constraints of the robotic manipulator are considered, such as joint limits. Also, the trajectory is generated within the Arduino code.

The hardware implementation of the stepper motor is simple as it doesn't have a feedback sensor. However, we must consider the initial position as a defined value ($\theta_1 = 0$) and calculate the new position after each iteration through the Arduino code. It is like an incremental encoder. We use iteration to control both the stepper and servo motor to increase time instants and apply interruption.

```
277  void joint_move( int steps_1, float desired_pos_2, int delay_time_1, int delay_time_2) {
278    float current_time = millis();
279    //joint 1 (stepper motor)
280    int current_1 = current_pos_1;
281    int delay1 = delay_time_1;
282    int delay2 = delay_time_2;
```

This approach provides a flexible and customizable solution for controlling the motors in a robotic manipulator, allowing for real-time adjustments based on the information provided by the vision module. The function joint_move takes four arguments: steps_1, desired_pos_2, delay_time_1, and delay_time_2.

steps_1: desired position for the stepper motor

desired_pos_2: desired position of the servo motor

delay_time_1: the delay time between turning each step ON and OFF and before turning it ON in the next iteration. It is used to adjust the desired velocity by a predefined function.

```
int delay_velocity_stepper(float delta_theta, float omega_0, float acc,)
{
  int delay_;
  if (acc != 0) //We apply newton equations of motion
  {
    //by applying the equation delat_theta = omega_0 * dt + 0.5 * acc * dt^2
    //We have a quadric polynomial equation with parameters: a, b, c
    float a = 0.5 * acc; //coeffient of dt^2
    float b = omega_0; //coefficient of dt
    float c = -1 * delta_theta; //coefficient of dt^0
    int dt = (-b + sqrt((pow(b, 2) - 4 * a * c))) / (2 * a); //solution of dt

    delay_ = abs(0.5 * (dt / (delta_theta/step_))); //finding the delay time for each half step
    return (delay_*1e+6); //mircosecond

  }
  else
  {
    delay_ = abs(135000 / omega_0); //where the velocity is maximum and acceleration is zero
    return(delay_); //millisecond
  }
```

As shown, we have two cases of delays. When acceleration is non-zero, we use the motion equation. When the acceleration is zero and its zero at a single point, we apply a tested delay.

delay_time_2: the delay time between rotating a degree at a time.  It is used to adjust the desired velocity by a predefined function.

```
int delay_velocity_servo(float delta_theta, float omega_0, float acc)
{
  int delay_;
  if (acc != 0) //We apply newton equations of motion
  {
    //by applying the equation delat_theta = omega_0 * dt + 0.5 * acc * dt^2
    //We have a quadric polynomial equation with parameters: a, b, c
    float a = abs(0.5 * acc); //coeffient of dt^2
    float b = omega_0; //coefficient of dt
    float c = -1 * delta_theta; //coefficient of dt^0
    int dt = (-b + sqrt((pow(b,2) - 4 * a * c))) / (2 * a); //solution of dt
    delay_ = abs(dt / (delta_theta)); //finding the delay time for each degree
    return(delay_*1e+3); //milliseconds
  }
  else
  {
    delay_ = abs(354 / omega_0);
    return (delay_); //milliseconds

  }
}
```

For both motors as we define the current position within the code as well as using iterations to control the motors, we must add two conditions; one condition checks if the desired angle is in positive direction to the current position, the other condition checks if the desired angle is in negative direction to the current position, so that we determine the proper direction to rotate as well as the used conditions within the loops. For the stepper motor we have to increment the current_position variable  by 1 step after each iteration. For the servo motor, it has a feedback sensor (potentiometer) that we can read the current position and use it to start the loop and continue the motor motion.

```
286   if (steps_1 > current_1)
287   {
288     digitalWrite(dir_1, HIGH);
289     for (int i = current_pos_1; i <  steps_1; i++)
290     {
291       // Check for incoming data and stop the motors if an interrupt is received
292       if (Serial.available() > 0) {
293         char incoming = Serial.read();
294         if (incoming == 'I') //a person detected {
295           stopMotors();
296           unsigned long startTime = millis();
297           unsigned long duration = 10000; // 10 seconds duration
298           while (millis() - startTime < duration) {
299             if (Serial.available() > 0 && Serial.read() == 'I') {
300               // Restart the loop from the beginning
301               startTime = millis(); // Reset the start time
302             continue; // Restart the loop
303             }
304           }
305           // Resume motor movement
306           continue;
307         }
308       }
309       digitalWrite(step_1, HIGH);
310       delayMicroseconds(delay1);
311       digitalWrite(step_1, LOW);
312       delayMicroseconds(delay1);
313       current_pos_1 = current_pos_1 + 1;
314       if (current_pos_1 > upperlimit_steps_1) break;
315     }
316   }
```

The interrupt condition is added within the loops of both stepper and servo motors. The motors stops through stopMotors() function:

```
425  void stopMotors() {
426    // Stop the motors by setting the step pins low
427    digitalWrite(step_1, LOW);
428    current_pos_2 = joint_2.read();
429    joint_2.write(current_pos_2);
430  }
```

At the end of each movement the joint motors return to home position which is defined as zero for each joint angle.

In terms of iterations and interruption, controlling the servo motor in Arduino code is similar to controlling the stepper motor. However, the servo motor receives a single command specifying the desired joint angles. To reach the desired position, we increment or decrement the current position one degree at a time while monitoring feedback to ensure we have reached the desired position.

```
353  //joint 2 (servo motor)
354  current_pos_2 = joint_2.read();
355  if (desired_pos_2 > current_pos_2)
356  {
357    for (int i = current_pos_2; i <= desired_pos_2; i++)
358    {
359      // Check for incoming data and stop the motors if an interrupt is received
360     if (Serial.available() > 0) {
361       char incoming = Serial.read();
362       if (incoming == 'I') {
363         stopMotors();
364         unsigned long startTime = millis();
365         unsigned long duration = 10000; // 10 seconds duration
366         while (millis() - startTime < duration) {
367           if (Serial.available() > 0 && Serial.read() == 'I') {
368             // Restart the loop from the beginning
369             startTime = millis(); // Reset the start time
370            continue; // Restart the loop
371           }
372         }
373         // Resume motor movement
374         continue;
375       }
376     }
377
378       joint_2.write(i);
379       delay(delay2);
380    }
381  }
```

```
void home()
{
  //homing joint 1
  current_pos_1 > 0? digitalWrite(dir_1, LOW) : digitalWrite(dir_1, HIGH);
  for (int i = 0; i <= abs(current_pos_1); i++)
  {
    // Check for incoming data and stop the motors if an interrupt is received
    if (Serial.available() > 0) {
      char incoming = Serial.read();
      if (incoming == 'I') {
        stopMotors();
        unsigned long startTime = millis();
        unsigned long duration = 10000; // 10 seconds duration
        while (millis() - startTime < duration) {
          if (Serial.available() > 0 && Serial.read() == 'I') {
            // Restart the loop from the beginning
            startTime = millis(); // Reset the start time
          continue; // Restart the loop
          }
        }
        // Resume motor movement
        continue;
      }
    }
    digitalWrite(step_1, HIGH);
    delayMicroseconds(10000);
     digitalWrite(step_1, LOW);
    delayMicroseconds(10000);
  }
```

### 4.1.4 task planning module

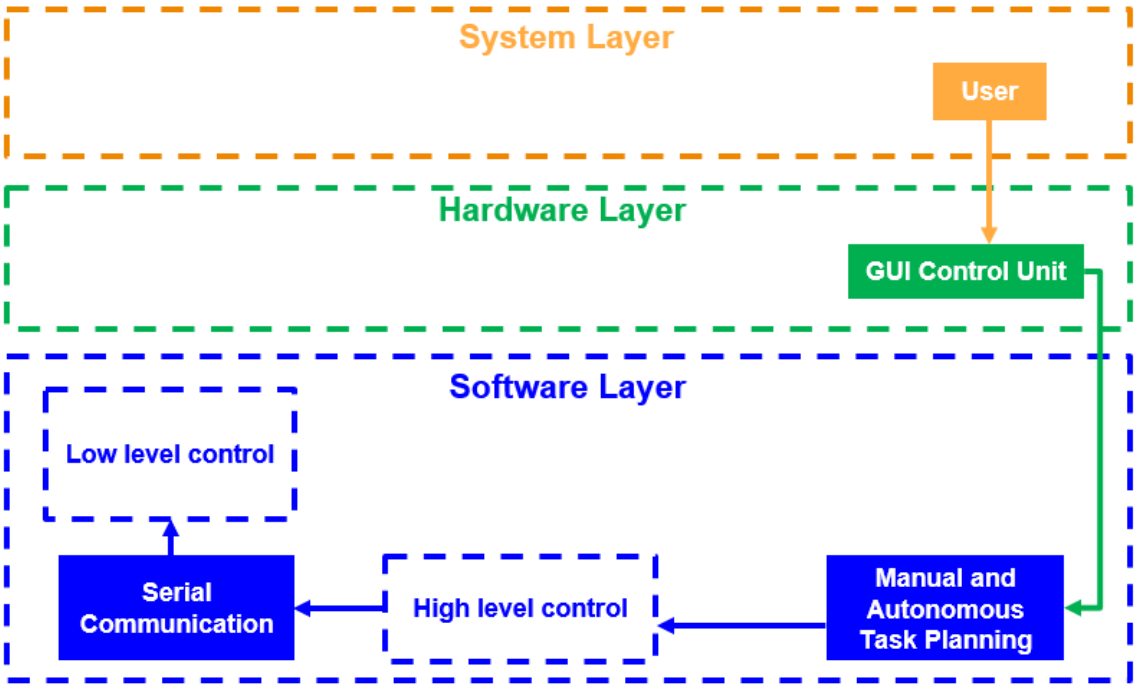



*Figure 9 Task planning architecture*

In the task planning process, the system layer plays a central role by integrating various components to ensure seamless coordination. The user interacts with the system layer, providing information and inputs. This information flows down to the hardware layer, where the GUI control unit receives and processes user commands. The GUI control unit then transfers the data to the software layer through manual and autonomous task planning mechanisms.

The software layer utilizes the task planning algorithms to determine the optimal sequence of actions for the robotic system. This layer combines both manual inputs from the user and autonomous decision-making to execute tasks efficiently and effectively.

Once the task planning is complete, signals are transmitted from the software layer to the high-level control system. This high-level control system acts as the bridge, relaying the signals to the serial communication interface. The serial communication interface transfers the signals to the low-level control system, which governs the specific hardware components.

In the final stages of the task planning process, the low-level control system sends the signals to the stepper and servo motors. These motors actuate the physical movements and actions required to carry out the designated tasks. By precisely controlling the motors, the robotic system executes the planned actions with accuracy and precision.

Through this coordinated process, the task planning system ensures a seamless flow of information and control, enabling effective interaction between the user, software, and hardware layers. This collaborative approach maximizes the capabilities of the robotic system and enhances its performance in executing complex tasks.

## GUI (Graphical User Interface)

### Introduction

GUI, or Graphical User Interface, is a visual interface that enables users to interact with a computer system or software application through graphical elements such as

buttons, menus, and icons. It provides a more intuitive and user-friendly way of controlling and monitoring complex systems, like collaborative robots.

The Graphical User Interface (GUI) is an integral part of the collaborative robot system, designed to facilitate user interaction and control. In this documentation, we will outline the workflow of the GUI for your collaborative robot project, which involves manual control, vision-based object detection, and object feature search functionality.

For the implementation of the GUI in our project, we have utilized the Python programming language and leveraged the capabilities of the CustomTkinter and Tkinter libraries for GUI layout and design. Additionally, you have integrated the OpenCV library to enable AI-based object detection and image processing functionalities within the GUI.

Flowchart showing how to create a GUI:

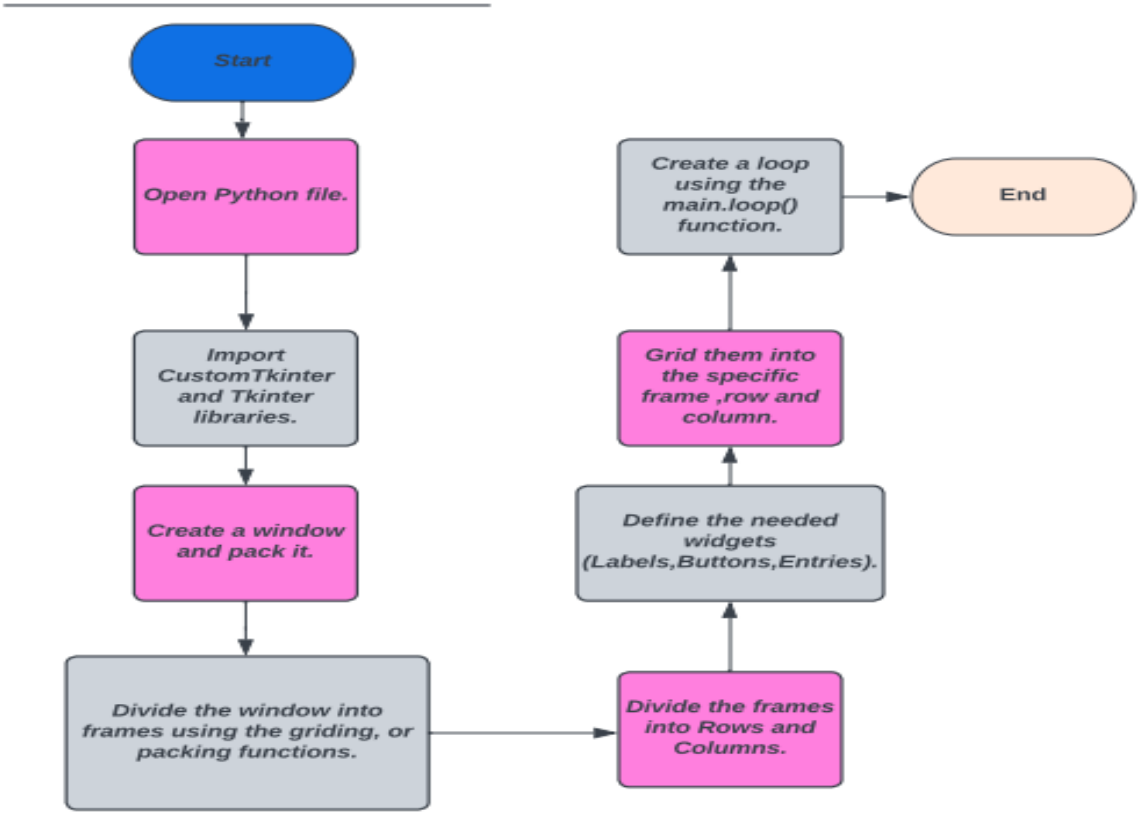



*Figure 10 Steps of building a GUI*

## Manual Control

The first tab of the GUI focuses on manual control of the collaborative robot. It provides users with the ability to command the robot's movements through forward and inverse kinematics. The following workflow outlines the steps involved:

a. Display: Upon launching the GUI, the manual control tab is initially displayed, presenting the relevant control elements and a visual representation of the robot's position in the workspace.

b. Forward Kinematics: Users can input desired joint angles or end-effector coordinates using the GUI interface. The GUI calculates the corresponding position and orientation of the robot in the workspace using forward kinematics equations. This information is then sent to the robot for execution.

c. Inverse Kinematics: Users can input desired position and orientation coordinates using the GUI interface. The GUI calculates the corresponding joint angles required to achieve the specified end-effector pose using inverse kinematics algorithms. The calculated joint angles are then sent to the robot for execution.

d. Command Buttons: Users can change the theme from dark to light or exit the GUI easily by clicking the buttons.

Here is part of the code of the Manual Tab

```
# Additional Inner Frames
IK_frame      =ctk.CTkFrame(Tab_1,width=400,height=200)
FK_frame      =ctk.CTkFrame(Tab_1,width=400,height=200)
display_frame=ctk.CTkFrame(Tab_1,width=400,height=200)
other_frame   =ctk.CTkFrame (Tab_1,width=400,height=200)
# grid of frames
IK_frame.grid_propagate(False)
IK_frame.grid(row=0,column=0,padx=5,pady=5,sticky='nsew')
FK_frame.grid_propagate(False)
FK_frame.grid(row=0,column=1,padx=5,pady=5,sticky='nsew')
display_frame.grid_propagate(False)
display_frame.grid(row=1,column=0,padx=5,pady=5,sticky='nsew')
other_frame.grid_propagate(False)
other_frame.grid(row=1,column=1,padx=5,pady=5,sticky='nsew')
# IK frame Widgets
IK_frame.columnconfigure((0,1,2),weight=1)
IK_frame.rowconfigure((0,1,2),weight=1)

x_e      =ctk.CTkEntry(IK_frame,width=50)
y_e      =ctk.CTkEntry(IK_frame,width=50)
z_e      =ctk.CTkEntry(IK_frame,width=50)
```

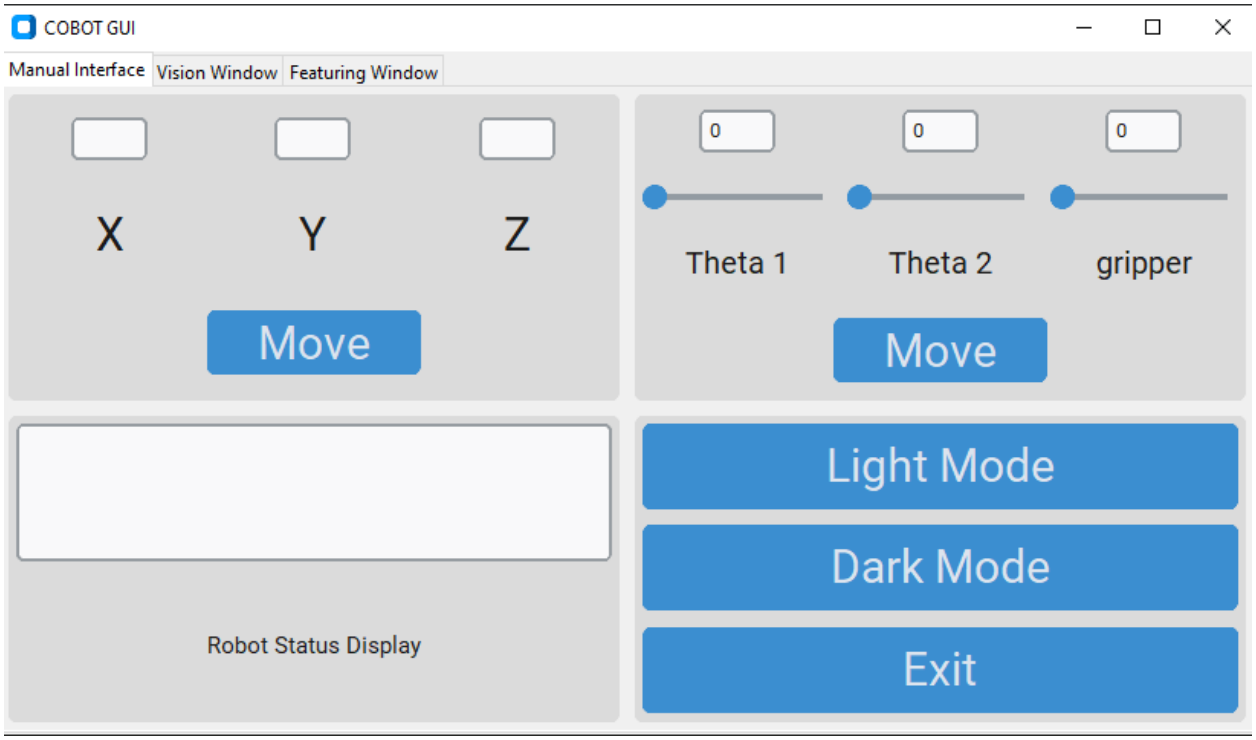



Figure: Manual Interface Window

## Vision-Based Object Detection

The second tab of the GUI focuses on vision-based object detection in the shared workspace. It allows the robot to detect objects and their coordinates using an RGB camera. The workflow for this tab is as follows:

a. Image Capture: Users initiate the object detection process by capturing an image of the workspace using the integrated RGB camera. The image is then processed and displayed in the GUI for visualization.

b. Object Detection: The GUI utilizes the OpenCV library to analyze the captured image and identify any objects present in the workspace. Through image processing techniques, such as color filtering, edge detection, and contour analysis, the system detects objects based on predefined characteristics.

c. Object Coordinates: Once an object is detected, the GUI determines its x and y coordinates relative to the robot's reference frame. This information is crucial for subsequent operations, such as inverse kinematics-based movement planning.

d. Buttons: Users can use these buttons to set the background image, start the detection of the live stream by the RGB camera, move the robot to the required position, or exit the program.

Here is part of the code for the Vision-Based tab:

```
############### Second Window ########################################
Tab_2.columnconfigure((0,1),weight=1)
Tab_2.rowconfigure(0,weight=4)
Tab_2.rowconfigure(1,weight=1)


background_frame=ctk.CTkFrame(Tab_2,width=50,height=100)
processing_frame=ctk.CTkFrame(Tab_2,width=50,height=100)
obj_loc_frame   =ctk.CTkFrame(Tab_2,width=50,height=50)
btns_frame      =ctk.CTkFrame(Tab_2,width=50,height=50)



background_frame.grid_propagate(False)
background_frame.grid(row=0,column=0,padx=5,pady=5,sticky='nsew')
processing_frame.grid(row=0,column=1,padx=5,pady=5,sticky='nsew')
obj_loc_frame.grid(row=1,column=0,rowspan=2,padx=5,pady=5,sticky='nsew')
btns_frame.grid(row=1,column=1,rowspan=2,padx=5,pady=5,sticky='nsew')
```

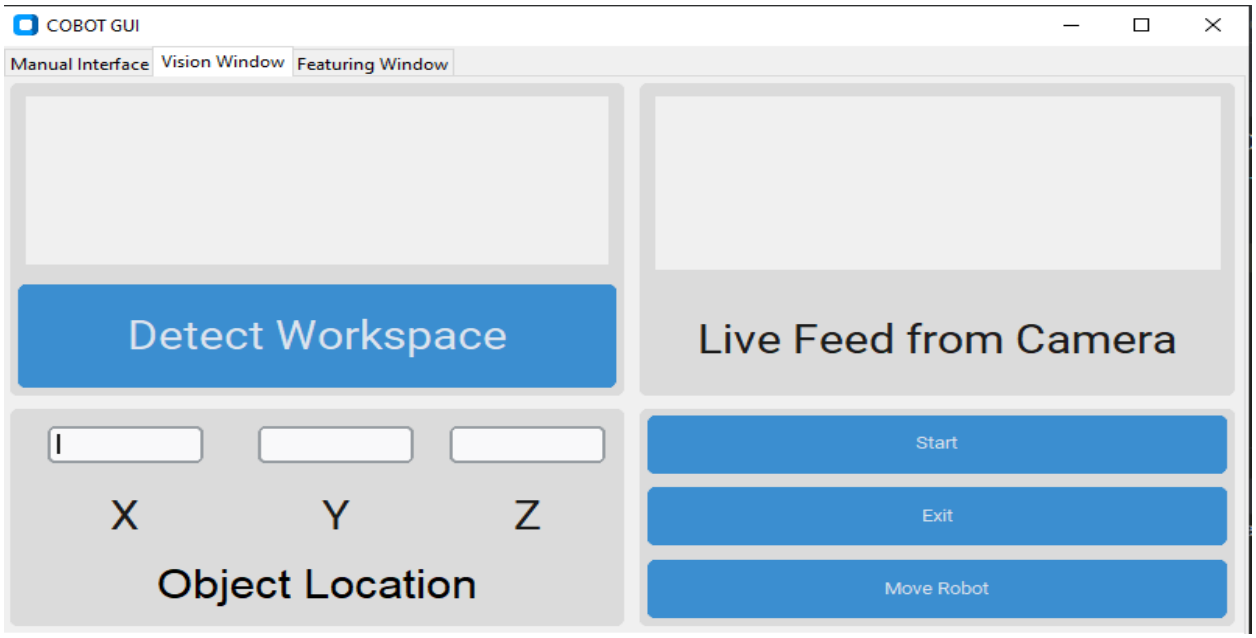



Figure: Vision-Based Window

## Object Feature Search

The third tab of the GUI focuses on object feature search functionality, allowing the robot to search for a specific object among multiple objects present in the workspace. Unlike the previous tab, where objects were detected using the RGB camera, in this tab, the reference object is provided as a JPG image.

a. Reference Object Selection: Users select a reference object by providing a JPG image of the desired object through the GUI interface. The reference image represents the object that the robot will search for in the workspace.

b. Object Comparison: The GUI utilizes the OpenCV library's image recognition and matching algorithms to compare the features of the reference object with the captured image of the workspace. Through feature extraction and matching techniques, the system searches for similar objects.

c. Object Coordinates: When the desired object is identified among the objects in the workspace, the GUI determines its x and y coordinates relative to the robot's reference frame. These coordinates enable the robot to perform inverse kinematics-based movements to reach the object.

d. Buttons: Users can use these buttons to start the detection of the workspace by the RGB camera, move the robot to the required position, or exit the program.

- Here is part of the code of the Featuring window.

```
########################## Third Tab Frames  ##################################
Tab_3.columnconfigure((0,1),weight=1)
Tab_3.rowconfigure(0,weight=4)
Tab_3.rowconfigure(1,weight=1)

feature_frame       =ctk.CTkFrame(Tab_3,width=300,height=300)
workspace_frame     =ctk.CTkFrame(Tab_3,width=300,height=300)
location_frame      =ctk.CTkFrame(Tab_3,width=300,height=300)
homing_frame        =ctk.CTkFrame(Tab_3,width=100,height=100)

# feature_frame.grid_propagate(False)

feature_frame.grid(row=1,column=1,rowspan=2,padx=5,pady=5,sticky='nsew')
workspace_frame.grid(row=0,column=1,padx=5,pady=5,sticky='nsew')
location_frame.grid(row=1,column=0,rowspan=2,padx=5,pady=5,sticky='nsew')
homing_frame.grid(row=0,column=0,padx=5,pady=5,sticky='nsew')
```

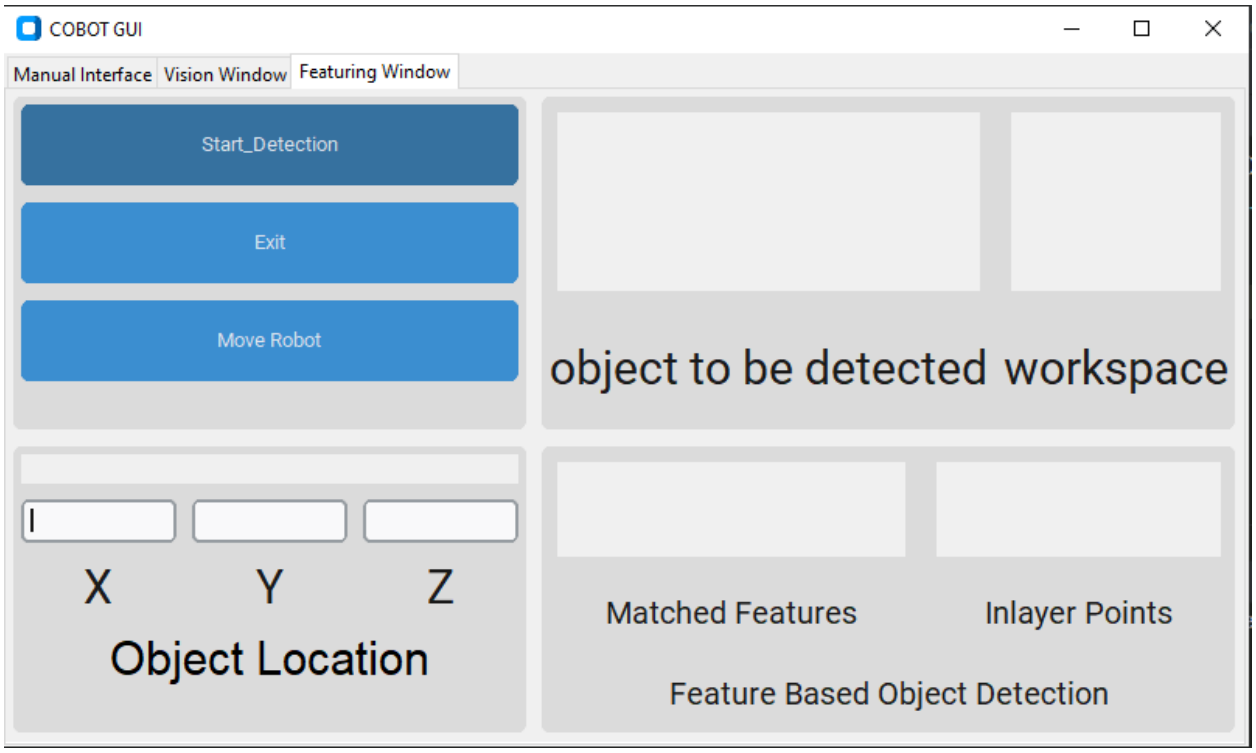



Figure: Featuring Window

## Functions

a. Forward_k: this function is in the manual tab and is used to apply forward kinematics, the user gives a value for the joint angles (theta 1 and theta 2) then the robot moves by these angles and the alternative x, y, z coordinates are displayed.

b. Move_robot_f1, Move_robot_f2, Move_robot_f3: these 3 functions are used to apply the inverse kinematics to move the robot in the manual, vision and featuring tabs respectively.

c. Set Background: this function is used in the vision tab to take a reference image from the workspace and then compare with it when the live feed is on. Any object that enters the workspace will be detected.

d. Start Processing: this function is used in the vision tab to start the live feed from the camera and apply image processing on it. The live feed is contentiously compared with the image captured from the set background function so that any object enters the workspace can be detected. If any object is detected the camera can easily state its x and y coordinates according to its coordinate system and then transform it into the robot coordinate system so that the robot can move to it.

e. Safety: in the safety function we use an AI model named YOLO to detect any human in the workspace. If any human is detected the robot stops directly, this instantaneous stop

is called Self-Monitored stop. If the human is no more detected in the workspace the robot waits for 10 seconds, then continue its last movement.

f. Feature Detection: this function is used in the feature tab to take the jpg image given by the user and compare it with screenshot image taken form the workspace. The comparison is based on the features extracted from the jpg image given from the user, so the more features the image has the easier it is to detect it.

g. Exit_Program: this function is in all the tabs, and it is used to exit the GUI program.

h. Move_homing: this function is used in all tabs to return the robot to its home position after applying the pick operation.

i. SendData: this function is used in all the tabs to send the data from the python script to the Arduino so that the Arduino can move the motors as needed.

Functions used in the GUI code:

```
def forward_k():...
def move_robot_f1():...
def move_robot_f2():...
def move_robot_f3():...
def set_background():...
def start_processing():...
def safety():...
def FeatureDetection():...
def exit_program():...
def move_homing():...
def sendData():...
```

## Conclusion

The GUI workflow outlined in this book provides an overview of the collaborative robot system's control and visualization capabilities. By leveraging the CustomTkinter and Tkinter libraries for GUI layout and design, and integrating the OpenCV library for object feature search, we have created a powerful interface for controlling the robot and performing advanced object recognition tasks.

With the manual control tab, users can command the robot's movements using forward and inverse kinematics. The vision-based object detection tab enables the robot to detect objects in the shared workspace. In the object feature search tab, users can select a reference object image, and the GUI performs feature comparison to locate similar objects in the workspace. By integrating these functionalities into a user-friendly GUI, you have created an efficient and collaborative robot system that enhances productivity, safety, and user experience.

# Communication

## Introduction

This chapter explores the communication between the Arduino Mega and a Python script and then between the Arduino Mega and the motors. It focuses on establishing a reliable and efficient method for controlling motors and performing various actions.

## Python Serial Library

The serial library in Python is a powerful tool for establishing serial communication between a Python script and the Arduino. It provides a simple and convenient way to send data from Python to the Arduino Mega. With the serial library, you can configure the port settings, such as baud rate and parity, for successful communication. The library offers functions like serial.Serial() to create a serial connection, write() to send data to the Arduino, and close() to gracefully terminate the connection. It also supports features like timeouts, allowing for flexible data transmission. By utilizing the serial library, you can effectively control the motors and trigger desired actions on the Arduino Mega from your Python script.

## Identifying the Arduino Mega Port

In order to establish communication, it is necessary to identify the correct serial port associated with the Arduino Mega. The serial.tools.list_ports library allows us to retrieve

a list of available ports and obtain information about them. By analyzing the port details such as name, description, and hardware ID, we can determine the appropriate port for successful communication.

## Establishing the Serial Connection

Once the correct port is identified, we establish a serial connection between the Python script and the Arduino Mega. This connection enables bidirectional data transfer. To establish the connection, we specify the identified port and set the baud rate, which determines the speed of data transmission between the two devices.

### Sending Data to the Arduino Mega

With the serial connection established, the Python script can send data to the Arduino Mega. This allows us to control motors and trigger desired actions. We use the write() method provided by the serial.Serial object to send data from the Python script to the Arduino Mega. Data is typically encoded in a compatible format, such as UTF-8, before transmission.

### Receiving Data from the Arduino Mega (optional)

If required by our project, the Python script can also receive data from the Arduino Mega. This enables us to obtain sensor readings, status information, or other relevant data. We utilize the read() method provided by the serial.Serial object to receive data from the Arduino Mega. The received data is then typically decoded from the chosen encoding format.

### Sending Data from Arduino to Stepper and Servo Motors

After receiving data from the Python script, the Arduino Mega can process and send instructions to control stepper and servo motors. The process for sending data to these motors involves the following steps:

Connecting Motors: Connect the stepper and servo motors to the Arduino Mega using appropriate motor control circuitry. This circuitry allows the Arduino to control the motion and positioning of the motors.

Data Processing: Process the received data from the Python script to extract relevant information for motor control. This includes extracting target positions, angles, or other parameters specific to stepper and servo motor control.

Stepper Motor Control: Utilize the appropriate libraries or functions in the Arduino programming language to control the stepper motor. For stepper motors, we use libraries like the AccelStepper library to control speed, direction, and steps. Functions like setSpeed(), moveTo(), and run() enable precise control over the stepper motor's movement.

Servo Motor Control: Use the appropriate libraries or functions to control the servo motor. Libraries such as the Servo library provide functions like attach() and write() to set the servo motor's position or angle. By specifying the desired position or angle, we can control the servo motor's movement.

Sending Motor Control Signals: Utilize the motor control functions or methods to send the corresponding control signals to the motor control circuitry. This could involve setting the number of steps, specifying the speed or direction for the stepper motor, or setting the angle for the servo motor.

Adjusting Motor Parameters: Dynamically adjust motor parameters, such as speed, acceleration, or servo angle, based on the received data. This allows the Arduino to respond to changing requirements or instructions from the Python script.

By following these steps, the Arduino Mega can effectively send data to control stepper and servo motors. This enables precise positioning and controlled movement based on the instructions received from the Python script.

**Serial Communication**

Serial communication is a method of data transfer that involves sending data one bit at a time over a single wire or channel. It is commonly used for short-distance communication between devices. In contrast to parallel communication, where multiple bits are sent simultaneously over separate wires, serial communication uses fewer wires, making it more suitable for applications with limited physical connections.

For example, consider sending the value "1011" from a computer to an Arduino Mega using serial communication. The data is transmitted one bit at a time, such as "1", "0", "1", and "1", over a single wire. This sequential transmission allows for simpler hardware requirements and is often more cost-effective.

**USB Port and Arduino Mega**

The USB port on the Arduino Mega facilitates the serial communication between the board and the computer.

The USB (Universal Serial Bus) protocol itself is based on synchronous communication, where data is transmitted in a synchronized manner with the help of a clock signal. However, the USB connections between the Arduino Mega and the computer typically operate in an asynchronous manner.

To be more precise, the USB connection between the Arduino Mega and the computer follows an asynchronous serial communication method. This means that the data transmission over the USB connection is not continuously synchronized with a clock signal, as in synchronous communication. Instead, it relies on start and stop bits to delimit data packets during transmission.

In summary, the USB connection between the Arduino Mega and the computer is asynchronous, even though the underlying USB protocol is synchronous. This asynchronous communication method allows for flexible and efficient data transfer between the two devices.

Here are some screenshots from our Arduino communication code.

```
for (int i = 0; i < DOF; i++) {
  for (int t = 0; t < t_f + 1; t++) {
    traj[i][t].q = a0[i] + a2[i] * pow(t, 2) + a3[i] * pow(t, 3);
    traj[i][t].qdot = 2 * a2[i] * t + 3 * a3[i] * pow(t, 2);
    traj[i][t].qddot = 2 * a2[i] + 6 * a3[i] * t;
    qOutPut[i][t] = traj[i][t].q;
    qdotOutPut[i][t] = traj[i][t].qdot;
    qddotOutPut[i][t] = traj[i][t].qddot;
    // Print the trajectory values
    Serial.print("DOF ");
    Serial.print(i);
    Serial.print(", t = ");
    Serial.print(t);
    Serial.print(": q = ");
    Serial.print(traj[i][t].q);
    Serial.print(", qdot = ");
    Serial.print(traj[i][t].qdot);
    Serial.print(", qddot = ");
    Serial.println(traj[i][t].qddot);

  }
}
```

```
#include <Servo.h>
#include <math.h>

// trajectory parameters
const int t_f = 6;
const int DOF = 2;
float q_0[DOF] = {0, 109};
float q_f[DOF];
float qdot_0[DOF] = {0, 0};
float qdot_f[DOF] = {0, 0};
int min_delay = 6000;
float qOutPut[DOF][t_f + 1];
float qdotOutPut[DOF][t_f + 1];
float qddotOutPut[DOF][t_f + 1];

// Global variables
int current_angle_1 = 0; // current p
int current_angle_2 ; // current posi

struct Trajectory {
  float q;
  float qdot;
  float qddot;
};
```

Flowchart showing the communication process.

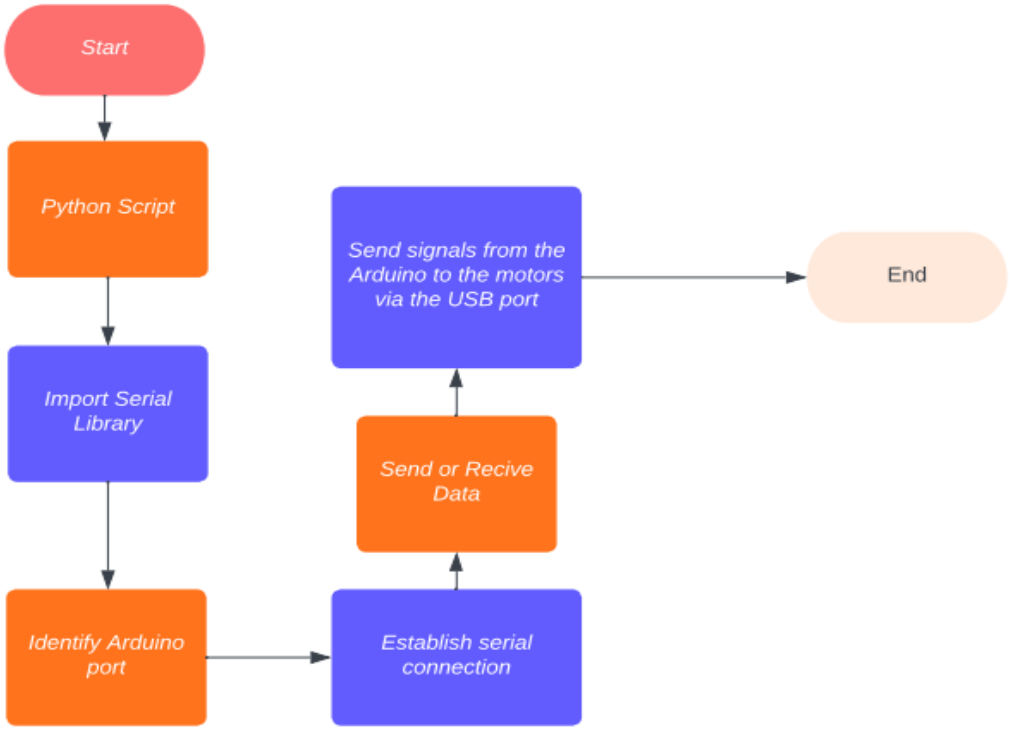



*Figure 11 Communication process*

**Conclusion**

In conclusion, the communication between the Arduino Mega and a Python script plays a vital role in controlling motors and achieving desired functionality. Serial communication, specifically through the USB port, enables the exchange of data between the two devices.

By leveraging the serial.tools.list_ports library in Python, we can identify the correct serial port associated with the Arduino Mega. Establishing the serial connection allows us to send data from the Python script to the Arduino Mega using the write() method provided by the serial.Serial object.

The USB port on the Arduino Mega follows an asynchronous serial communication method, where data transmission is not continuously synchronized with a clock signal. Instead, it utilizes start and stop bits to delimit data packets during transmission.

With the serial connection established, the Arduino Mega can process the received data and send instructions to control motors. The specific method of sending data to motors depends on the motor type and the motor control circuitry used in the project. Examples include using libraries like the AccelStepper library for stepper motors or the Servo library for servo motors.

Closing the serial connection gracefully terminates the communication and releases system resources.

Overall, by effectively utilizing serial communication and the serial library in Python, we can establish a reliable and efficient communication channel between the Python script and the Arduino Mega, enabling precise control over motors and facilitating the desired actions.