DESIGN OF A MOBILE ROBOT WITH SELF-LOCALIZATION FOR SEARCH AND RESCUE

THE HONG KONG POLYTECHNIC UNIVESITY

DEPARTMENT OF MECHANICAL ENGINEERING

FINAL YEAR CAPSTONE PROJECT REPORT

18 MAY 2020

SUBMITTED BY GROUP 2

XU XINRUI 16097906D
YANG CHENG 16096359D
ZHANG YUMING 16098712D

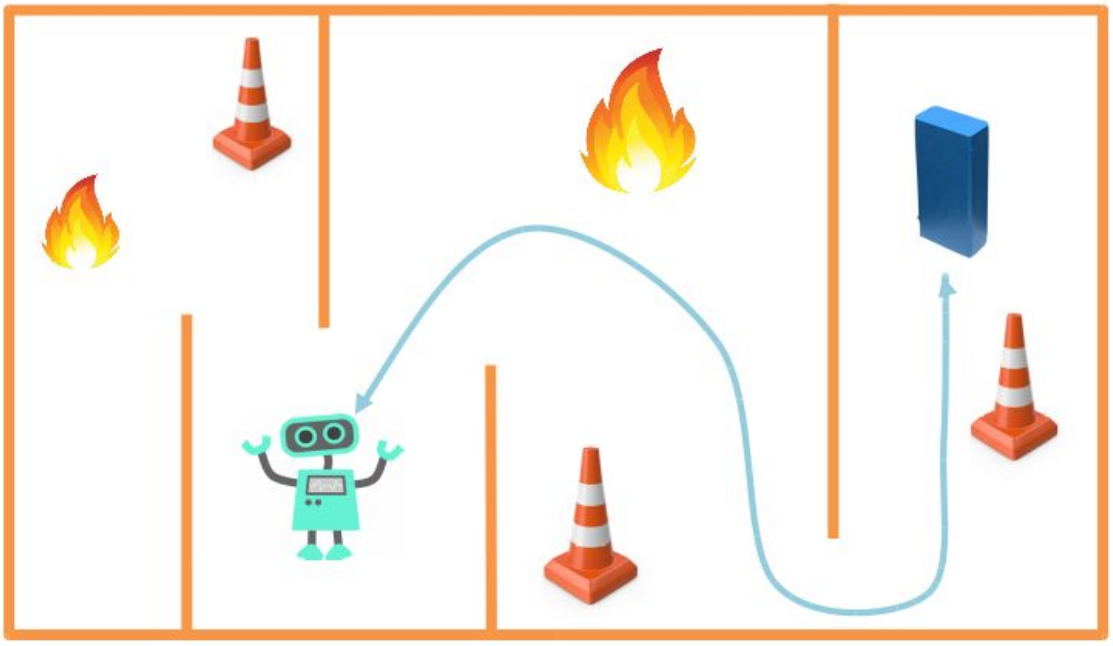
SUPERVISOR

DR. HENRY CHU

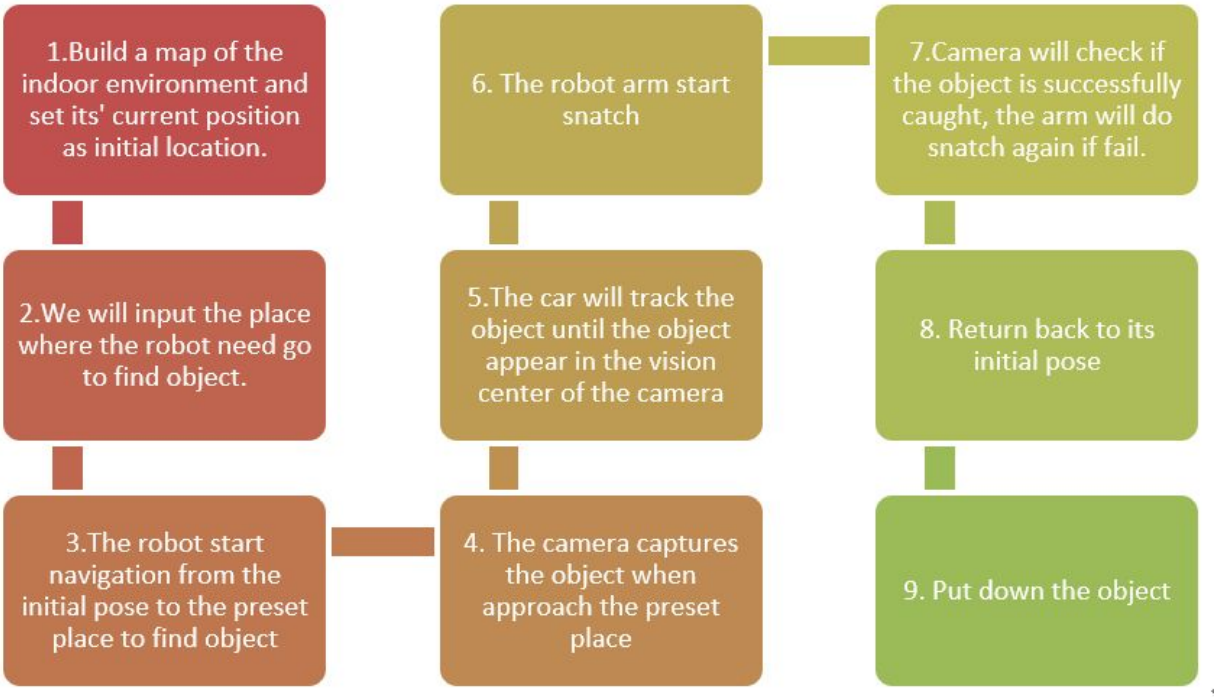
Final Year Project: Self-navigation robot for search and rescue

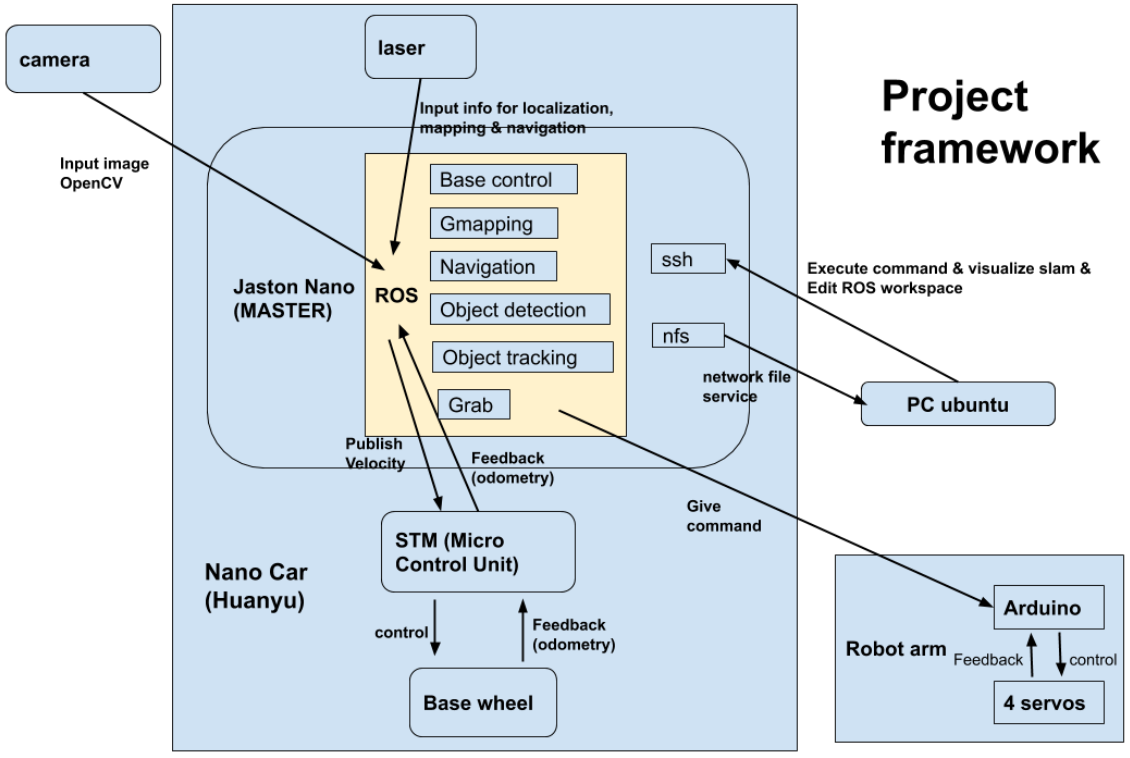
Overview:

In this report, we are going to show you how we combined a SLAM robot with vision and robotic arm to build a self-navigation rescue robot. We choose to use a SLAM robot as the development base with a gmapping algorithm. The whole architecture is built in ROS based on Ubuntu 18.04 on a Jaston Nano board. We first assume a simple scenario for performing rescue and specify the task. Our rescue target is a blue wooden block which needs to be picked up. Then, we divided the rescue task into several sub functions and experimented with them one by one. The rescue is successfully constructed by the integration of small functions. OpenCV is used to detect the target and a robotic arm with an arduino control board is connected to the SLAM robot to grab the blue block and perform the rescue mission. We designed the architecture and combined multiple functions together and enabled the robot to perform the automatic rescue task.

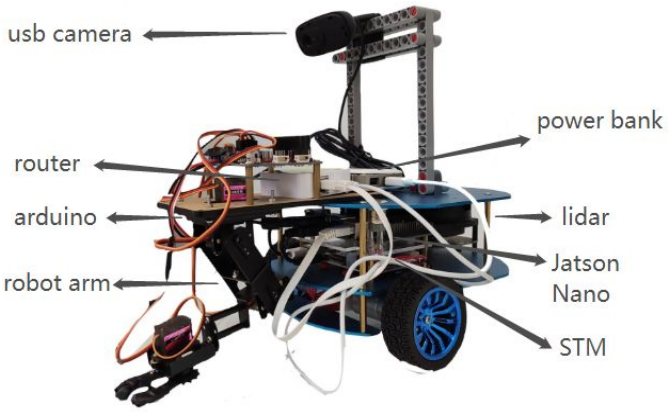


Workflow:


1.Build a map of the indoor environment and set its' current position as initial location.
2.We will input the place where the robot need go to find object.
3.The robot start navigation from the initial pose to the preset place to find object
4. The camera captures the object when approach the preset place
5.The car will track the object until the object appear in the vision center of the camera
6. The robot arm start snatch
7.Camera will check if the object is successfully caught, the arm will do snatch again if fail.
8. Return back to its initial pose
9. Put down the object


Project framework
camera
laser
Input info for localization, mapping & navigation
Input image OpenCV
Base control
Gmapping
Navigation
Object detection
Object tracking
Grab
Jaston Nano (MASTER)
ROS
ssh
nfs
Execute command & visualize slam & Edit ROS workspace
network file service
PC ubuntu
Publish Velocity
Feedback (odometry)
Give command
STM (Micro Control Unit)
Nano Car (Huanyu)
control
Feedback (odometry)
Base wheel
Arduino
Robot arm
Feedback
control
4 servos

Robot structure + Hardware + Software



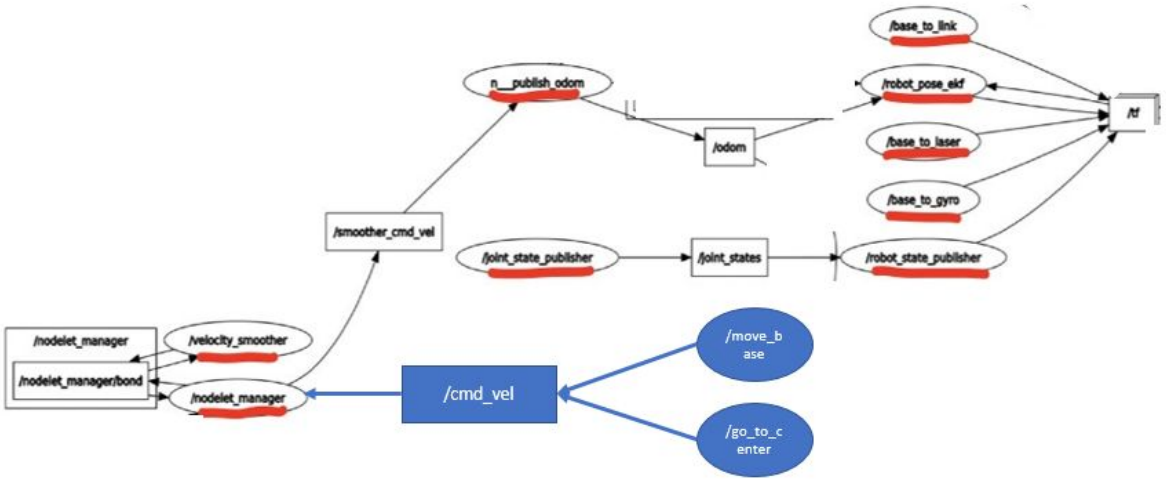
Hardware		
Item	**Provider**	**Function**
Jatson Nano Developer Kit	Huanyu-Hangzhou	Process data, main control center
Router	Huanyu-Hangzhou	Connect the Jetson Nano with PC and Internet
Lidar	Huanyu-Hangzhou	Scan
HiBot board	Huanyu-Hangzhou	Control board for Lidar and Motor
Camera	Logitech	Detect object
Arduino board	Taobao-1	Control board for robotic arm
Robotic arm + servo	Taobao-1	Grab object
PC	Self used PC	Remote control Jetson Nano, monitor task procedure

Software/tool/ Library	
Item	**Description**
Ubuntu 18.04	OS for Robot Operating system
Arduino	Platform to code robot arm
OpenCV	Image processing
Robot Operating System (ROS)	Platform to develop robot projects
Rviz	Visualization of images and topics

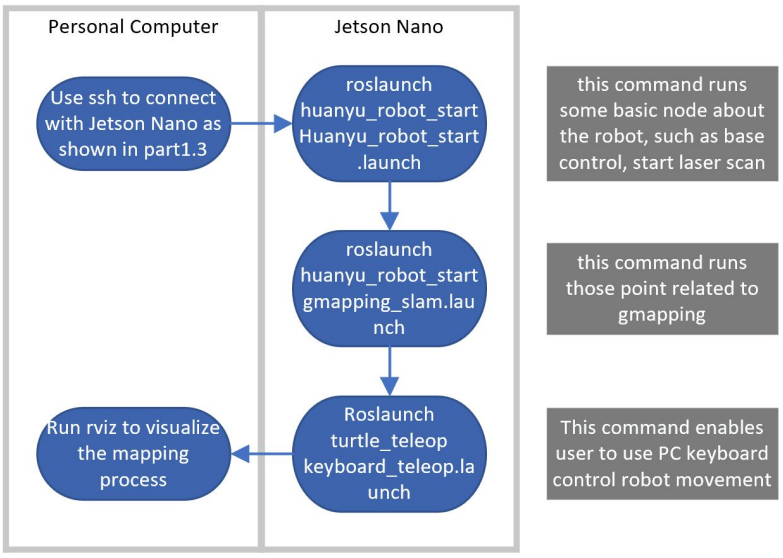
Functions

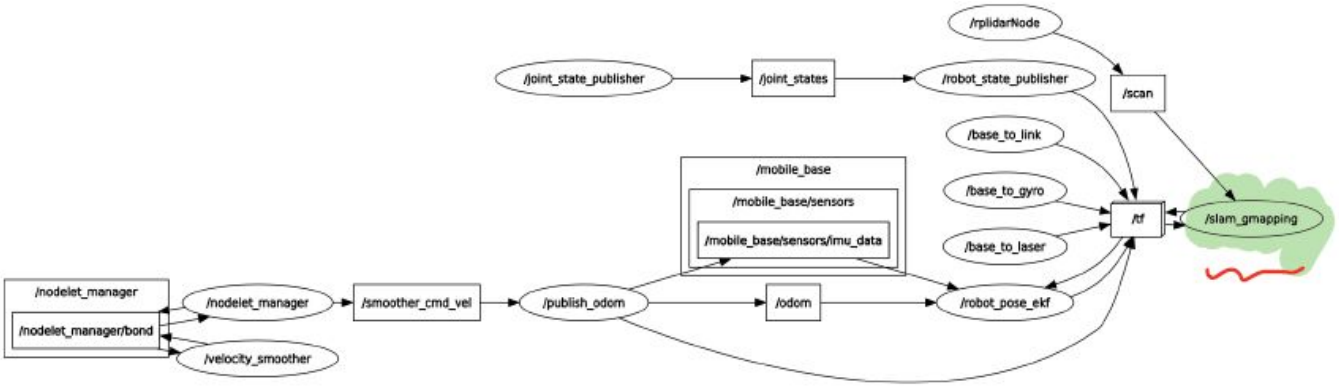
Function			
Number	Name of the function	Major Program provider	Description
1	Base control	Huanyu	Full code available and accessible inside Nano robot for base package control
2	Mapping (SLAM)	ROS	Open source : gmapping
3	Navigation (SLAM)	ROS	Open source package: move_base
4	Object detection	Ourselves	Detect the object and get the center location
5	Object tracking	Ourselves	Move Nano to an appropriate location for picking up the object
6	Grab	Ourselves	Pick up the object and put down the object (code in arduino)
7	Path point	Ourselves	Record points for searching target and navigate robot to return when grab is finish

1.Base control

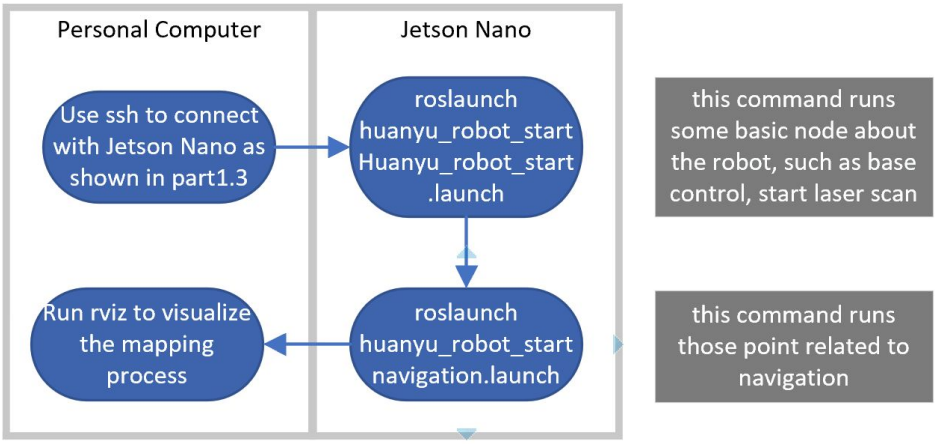


2.Mapping

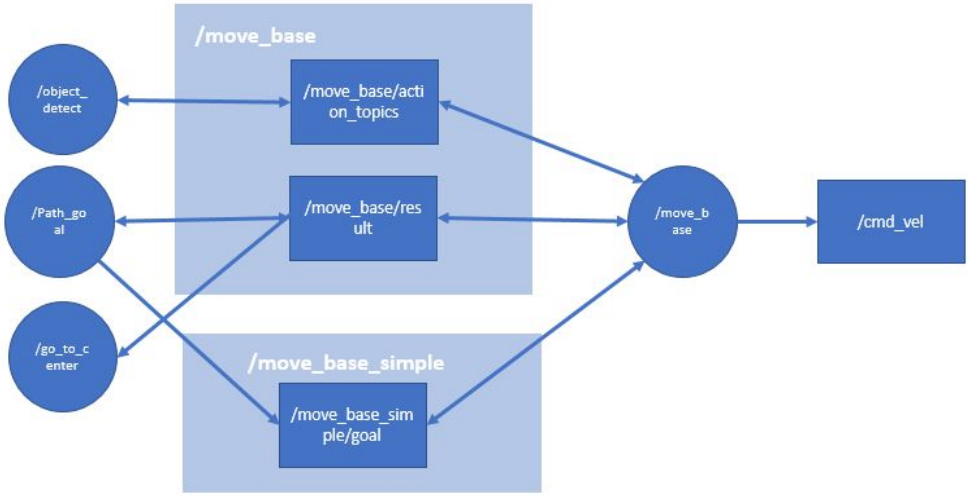




3.Navigation



Mapping is more independent from others. It is only used initially for map generation. After the map is generated, navigation function will be combined with other functions to perform rescue mission. Navigation move_base node needs to communicate with others in order to reach our final goal. The rqt graph and table for it is listed as low.



Node name: move_base			
Topic/Service	Direction	Message type	Function
/move_base_simple/goal	Subscriber	geometry_msgs/P oseStamped	Get a goal for move_base to pursue in the map
/move_base/result	Publisher	move_base_msgs/ MoveBaseAction Result	Indicate whether a navigation is finished
/cmd_vel	Publisher	geometry_msgs/T wist	Publish velocity to robot
SimpleActionClient<ActionSpec>::c ancelGoal(Action achieved through move_base/action_topics)	Server	actionlib::Simple ActionClient	Cancel the goal that are currently running

Move_base will receive the destination(search) points from path_goal by topic /move_base_simple/goal and it will navigate the robot to that location by publishing velocity information to /cmd_vel and once a navigation is completed, move_base is going to updates the result in /move_base/result by setting status = 3. This works during both searching for the object and

returning after the robot gets the object. When the robot finds the object, it will receive a request from client node obejct_detect for the service of stopping navigation and cancelling the current goal (actionlib::SimpleActionClient< ActionSpec > ::cancelGoal).

4. Obeject_detect

We have color detection to identify our target and programmed it with Opencv.

The digital image is by default using RGB representation. There are 5 steps in color detection:

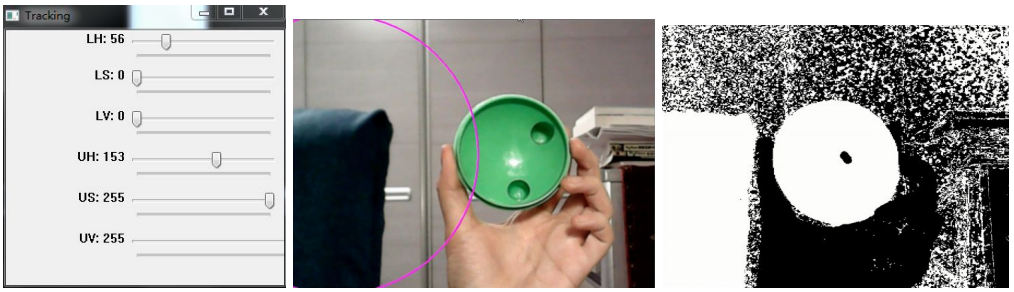
- Read the image and convert it to HSV
- Set the range for HSV boundary to detect a specific color and generate a mask
- Apply morphological operations to eliminate the noise on the mask
- Use bitwise operations and mask to segment out colors out of range
- Detect the area and draw the circle to identify the color concentration area

Step 1: Read the image and convert it to HSV

OpenCV reads the image by default in RGB format and we use the library function to convert it to HSV. As in any other signals, images also can contain different types of noise, especially because of the source (camera sensor). We applied image Smoothing techniques to help in reducing the noise.

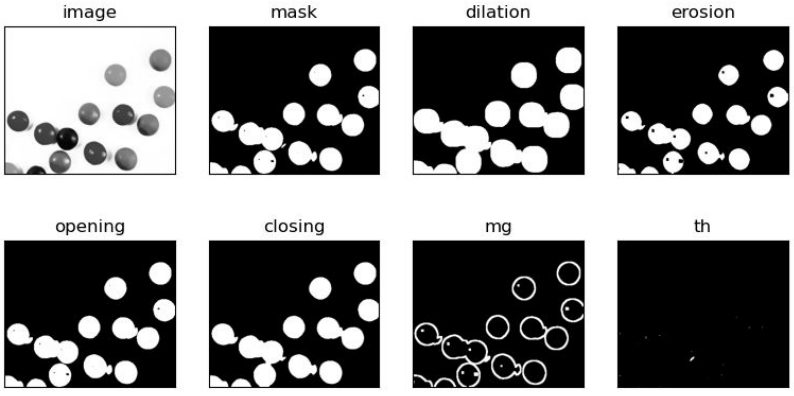
Step 2: Set the range for HSV boundary to detect a specific color and generate a mask

We need to get the HSV value of our target and then set an appropriate range to eliminate other values. To achieve this, we created six-track bars that control the lower and upper value of HSV correspondingly. The program will read the value from the trackbars and use it to generate a mask to segment out others. The cv.inRange() function simply returns a binary mask, where white pixels (255) represent pixels that fall into the upper and lower limit range and black pixels (0) do not.



Step 3: Apply morphological operations to eliminate the noise on the image

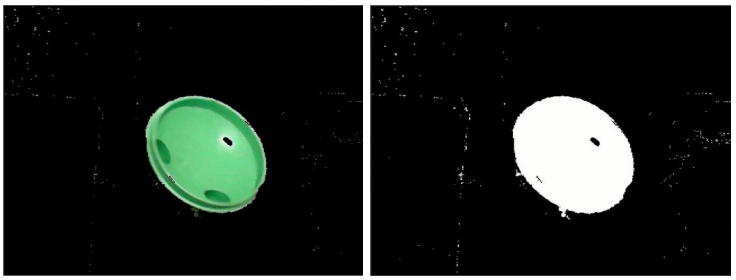
Before using the mask the filter other colors, some morphological transformation is performed to eliminate the noise and minors in the mask. Morphological transformations are some simple operations based on the image shape. It is normally performed on binary images. It needs two inputs - our original image and kernel which decides the nature of the operation (same as the one used in image smoothing). Two basic morphological operators are Erosion and Dilation. There are also other operations that developed based on these two, such as opening and closing. The result after different morphological transformation is shown below:



Step 4: Use bitwise operations to segment out colors that are not in range

Bitwise provides simple binary logic for and, or, xor operations. By applying the mask in the operation, the image left(res) would be the parts that have 1 value in the mask.

X	Y	X&Y
0	0	0
0	1	0
1	0	0
1	1	1



Step 5: Detect the area and draw the circle to identify the color concentration area

OpenCV provides the function to find the contour of white color in the mask (cnt). We assume that the object is in a circular shape. cv2.minEnclosingCircle(cnt) returns the center position and radius of the area of white in the mask. From that, we can draw it out using cv2.circle() for visualization.

Result

Below shows the processing of color identification using a green cup lid.

As we adjust the HSV range, the result is getting more accurate and the final result is very close to reality. The algorithm for finding the center is based on the calculation of the distribution of white color in the mask. This indicates that only a unique set of x, y will be returned by it and this is extremely important for our application. As this part - object_detection needs to pass the center

location of the target to ROS for object_tracking. Multiple values will lead to confusion and malfunctions. In addition, we are using single-colored objects for grab and detection.

```
-----new-----
 x: 461.79095458984375 y: 194.4076385498047 radius: 128.74227905273438
-----new-----
 x: 462.06732177734375 y: 194.34530639648438 radius: 128.99905395507812
-----new-----
 x: 462.9173583984375 y: 194.09104919433594 radius: 128.97369384765625
-----new-----
 x: 463.4538879394531 y: 192.1886444091797 radius: 129.3709716796875
-----new-----
 x: 463.8381652832031 y: 193.64895629882812 radius: 128.59420776367188
```

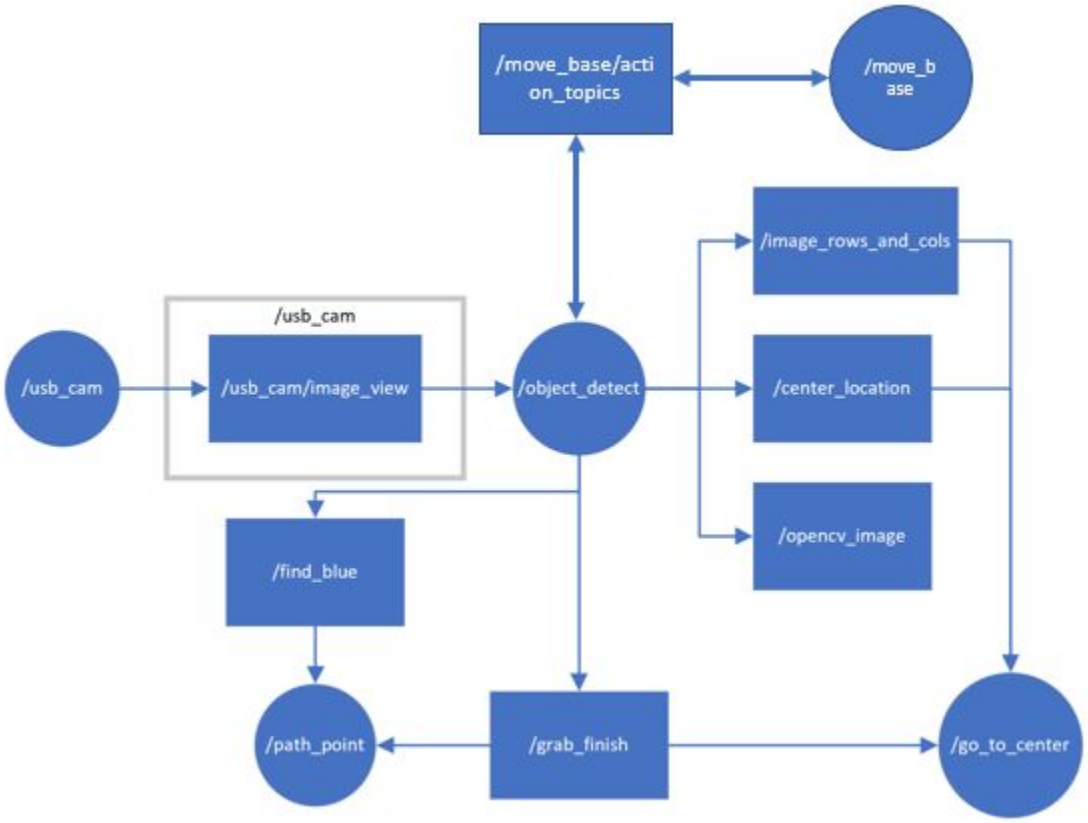
However, color detection is quite limited for object detection. Typically, an object with a pure color that is distinguishable with the environment is required for an accurate result. This makes it not applicable to most of the daily life items. In addition, in real practice, it is hard to ensure that the color is always unique and separable from the environment unless the environment is intentionally designed. Moreover, the luminance condition may vary, but the specified HSV range needed to be adjusted manually to adapt to the new situation. This means frequent adjustments will be needed.

OpenCV and ROS

To integrate OpenCV in ROS, we first need a node to capture the image and it is called /usb_cam.

Node name: usb_cam		
Topic	Direction	Message type
/usb_cam/image_raw	Publisher	sensor_msgs/Image

Then, OpenCV can be integrated into ROS. It is written in the python script vision.py and during the process, it runs in the node object_detect. The structure of it is shown below in the rqt graph:

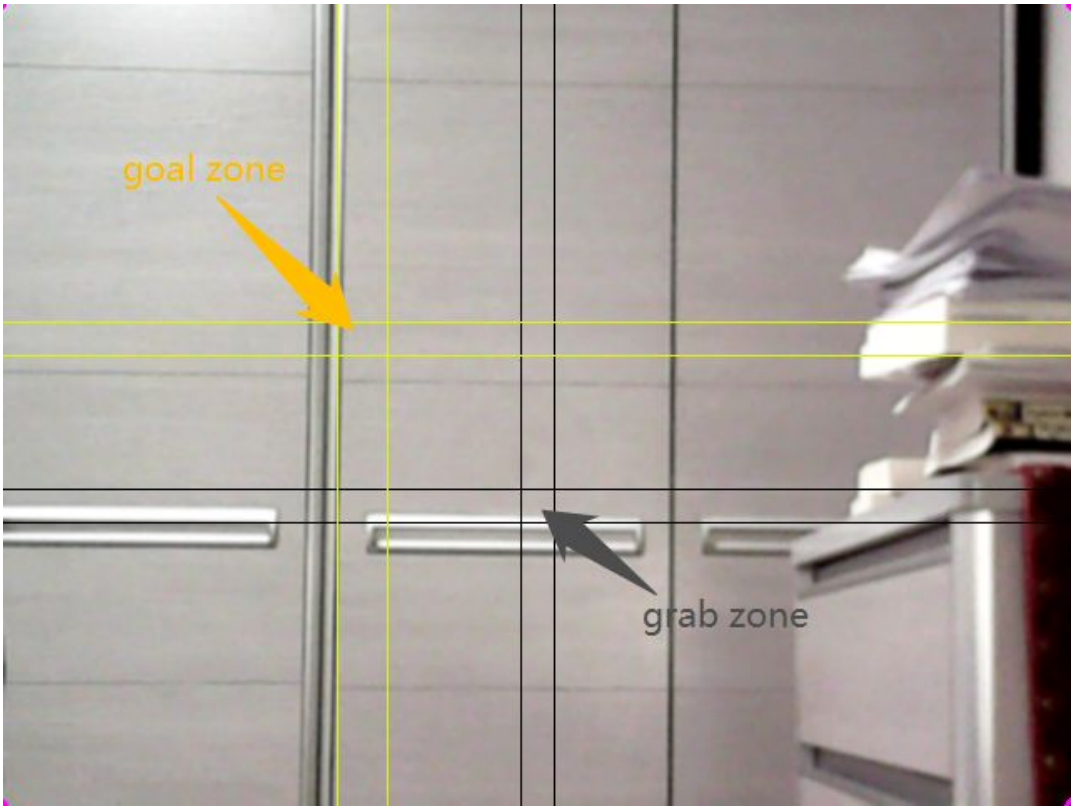


Node name: object_detect			
Topic/Action	Direction	Message type	Function
/usb_cam/image_raw	Subscriber	sensor_msgs/Image	get camera image
/opencv_image	Publisher	sensor_msgs/Image	visualize OpenCV processed image
/image_rows_and_cols	Publisher	geometry_msgs/Point	output image info
/center_location	Publisher	geometry_msgs/Point	reveal object center location
/find_blue	Publisher	geometry_msgs/Point	Indicate the object first appeared in the image
/grab_finish	Publisher	geometry_msgs/Point	Indicate the completion of grab task
SimpleActionClient<ActionSpec>::cancelGoal() (Action achieve through move_base/action_topics)	Client	actionlib::SimpleActionClient	Cancel the goal that are currently running

This node has multiple functions and integrates navigation, object-tracking, and grab functions. OpenCV is embedded inside for performing object detection. ROS passes around images in its own sensor_msgs/Image message format, but it can not be used directly in OpenCV. Here, we apply CvBridge to solve this problem. CvBridge is a ROS library that provides an interface between ROS and OpenCV. It can be found in the cv_bridge package in the vision_opencv stack. (cv_bridge)

```
1 from cv_bridge import CvBridge
2 bridge = CvBridge()
3 image_message = bridge.cv2_to_imgmsg(cv_image, encoding="passthrough")
```

After getting the correct format of the image, we perform color detection using the functions in OpenCV. As soon as a blue object appears, it will publish a message (X = 1) to /find_blue to indicate the need for pausing the navigation. After then, it will keep sending the center location of the target to /center_location until the object is in the zone for grabbing. Then, it will pause and wait for grabbing. As shown in the image below, the black square enclosed by the four black lines is the zone for grabbing and the yellow square represents the expected goal zone where the center of the target is located after successfully grabbing the object.



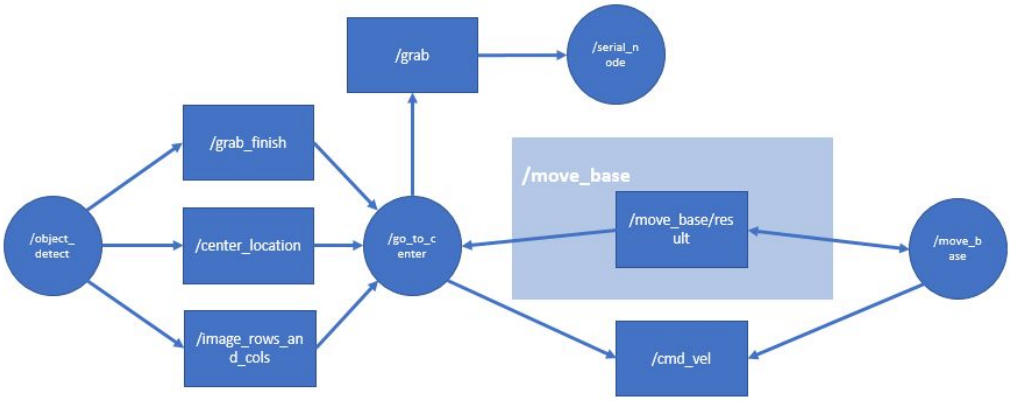
```
# x, y >> location of center
if x_range[0]<x<x_range[1] and y_range[0]<y<y_range[1]: # if the object is in the grab zone(center)
    rospy.loginfo("object in center!")
    rospy.sleep(10)  # sleep for 10 sec
    center = True    # the robot are expected to finish grab, and set center = True

if center == True: # if the object is previously in the grab zone(center) and grab is performed
    #check if the new center location is within the goal expectation zone or not
    if x_goal_range[0] < x < x_goal_range[1] and y_goal_range[0] < y < y_goal_range[1]:
        # yes, publish x=1 to grab_finish to indicate
        self.grab_finish.x = 1
        self.grab_finish_pub.publish(self.grab_finish)
        try_again = 0
        rospy.loginfo("grab complete!")
        rospy.signal_shutdown("grab finish") # shutdown this node
    else:
        # no, grab is failed, need another trial
        center = False
```

After giving the command for grabbing, the node will sleep for 10 seconds to wait for the robot arm to finish the grab task. Here, we set center = True to indicate that the robot has finished grabbing. Then, The node will detect the location of the target again. When Nano car is unmoved, the robot arm is fixed and the final location of the object on the screen after grabbing is therefore fixed as well. If the center location falls in the goal expectation zone, the object_detect node will publish a message to /grab_finsih to indicate the success of grab. Then, it will shutdown itself as all the tasks related to it have been completed. If the center is out of range, then we will reset the center = False and another iteration for color detection and tracking will move on.

5.Object Tracking

The mission of ‘Object Tracking’ is to identify the position of the object in the video frame and use the movement of the wheel to move the object to a specific small region of the frame. Once the object is located into that region, the robotic arm will do a pre-designed movement to grab the block. Once the robot is unmoved, the relative position between Robot and our target is fixed.



Node name: object_detect			
Topic/Action	Direction	Message type	Function
/move_base/result	Subscriber	move_base_msgs/MoveBaseActionResult	Check if navigation complete
/image_rows_and_cols	Subscriber	geometry_msgs/Point	get frame(image) info
/center_location	Subscriber	geometry_msgs/Point	get object center location
/grab_finish	Subscriber	geometry_msgs/Point	Check if grab task is finished
/cmd_vel	Publisher	geometry_msgs/Twist	controls the velocity of robot
/grab	Publisher	geometry_msgs/Point	Make the arm to pick up the target

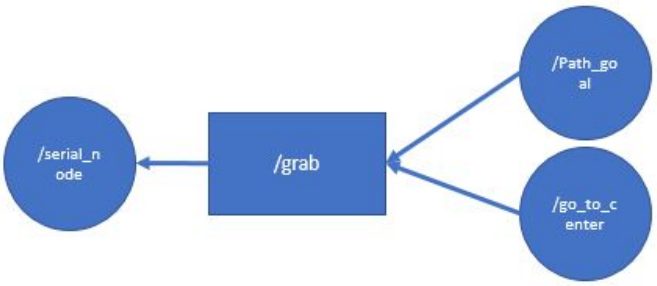
The Object Tracking is to track the object, so we only need to transfer the /center_location got from object tracking to the velocity of the robot (/cmd_vel) and publish /grab=2 when it is suitable for grabbing. /image_rows_and_cols is mainly used for design the velocity. However, we did not really know the control base and how to control the robot movement in the initial design stage of velocity. So we seek help from a Huanyu technician. Huanyu technician told us that we could send the velocity through /cmd_vel topic and arrange the velocity in 2X3 matrix. Then the robot base will follow the /cmd_vel value to move. Therefore, we divided the frame into 9 parts and each part corresponding to a particular velocity to be published.

Region	Motion	Linear velocity	Angular velocity
A	turn left and go straight	x=-0.01	z=0.01
B	go straight	x=-0.01	z=0.01
C	turn right and go straight	x=-0.01	z=-0.01
D	turn left	x=0.00	z=0.01
E	locate successfully	x=0.00	z=0.00
F	turn right	x=0.00	z=-0.01
G	go back	x=0.01	z=0.00
H	go back	x=0.01	z=0.00
I	go back	x=0.01	z=0.00

6. Grab - Arduino Robot Arm

The first action for the robotic arm is to catch. This action happens after the camera detects the object and moves the robot car to a specific distance from the object.(move the car until the object appears in the center of our camera). Since every time the distance of the object from the arm is constant, we can design a series of actions for each sero. First of all, we need to locate the start pose of the arm so after the servo gets the single, it won't rotate sharply and damage the servo. Then we also need to locate the final position of the arm, which is the pose we want when it catches the objection. After testing for many times, we get the suitable initial pose and final pose for our robotic arm.

After the pretest of our Arduino robot arm, it is then connected to ROS by rosserial. Rosserial is a protocol for wrapping standard ROS serialized messages and multiplexing multiple topics and services over a character device such as a serial port or network socket. The rosserial protocol aims at point-to-point ROS communications over a serial transmission line.



The arduino now becomes the node in the diagram (/serial_node) and it is synchronized with object detection and navigation functions. The robot arm has two functions - pick up and put down which we named as severalServoControl() and putdownServoControl() correspondingly in the code. Both need to be performed at a certain time, and the timing is regulated by other nodes. To control this, we make Arduino subscribe to a topic called '/grab' so that other nodes can send messages to inform the arduino board when to pick up and when to put down. Grab is a Point message which gives three values: x, y and z. We use the value of x for making judgements where 1 for picking up and 2 for putting down.

```
ros::Subscriber<geometry_msgs::Point> sub("grab", move_or_not);

void move_or_not(const geometry_msgs::Point &indicator){
  if(indicator.x == 1){
    severalServoControl(); //pick up the object
    delay(5000);
    a=1;
  }
  if(indicator.x ==2){
    putdownServoControl(); //put down the object
    delay(5000);
    a=0;
  }
}
```

The condition when X = 1 :

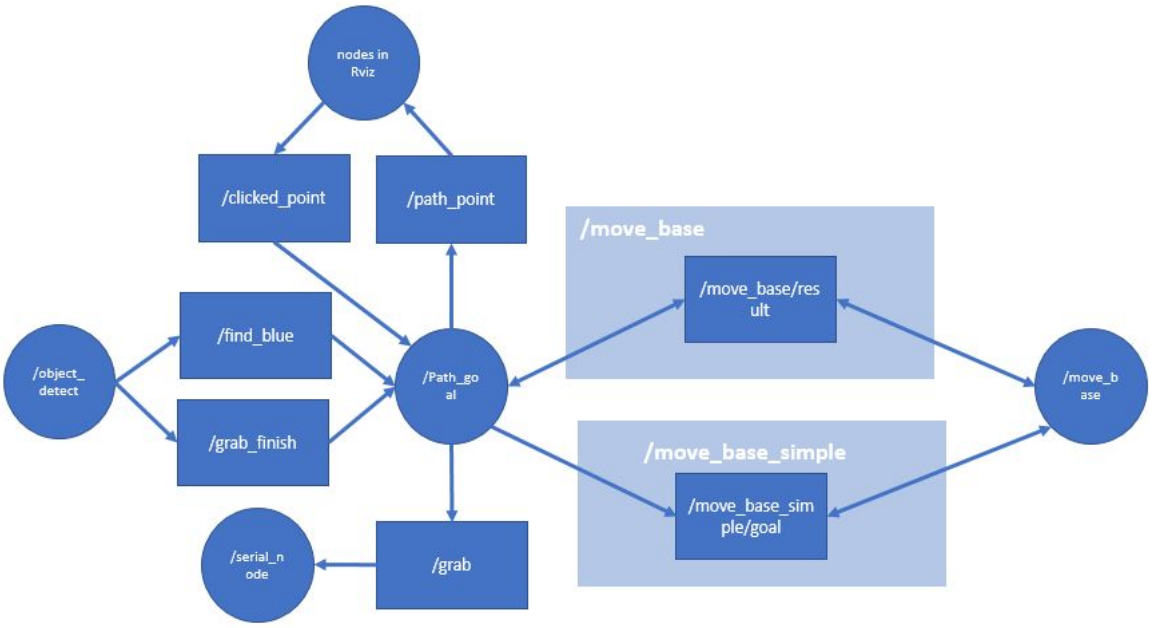
Once the object is located correctly in the image, the object detection program will send a point value as x=1 to /grab, and the robot will start working and picking up the object

The condition when X = 2 :

After the object is successfully grabbed, the robot will navigate to the destination (initial location). Once reached, it will publish x=2, then the arduino will implement the function of putting down the object..

7.Path point

By using the combination of functions, including object detection, object tracking and grab, we can now successfully find and grab the object if it appears inside the capture of the camera. It needs to be further integrated with navigation to complete the whole rescue task and therefore we introduce another function called path point. Path point provides a record of the whole navigation. It is mainly responsible for 2 things: 1. Record the input search point (destination points input by user in Rviz) 2.Navigate the robot to initial position if the robot has successfully grabbed the object. Below rqt and table summarise how it works with others.

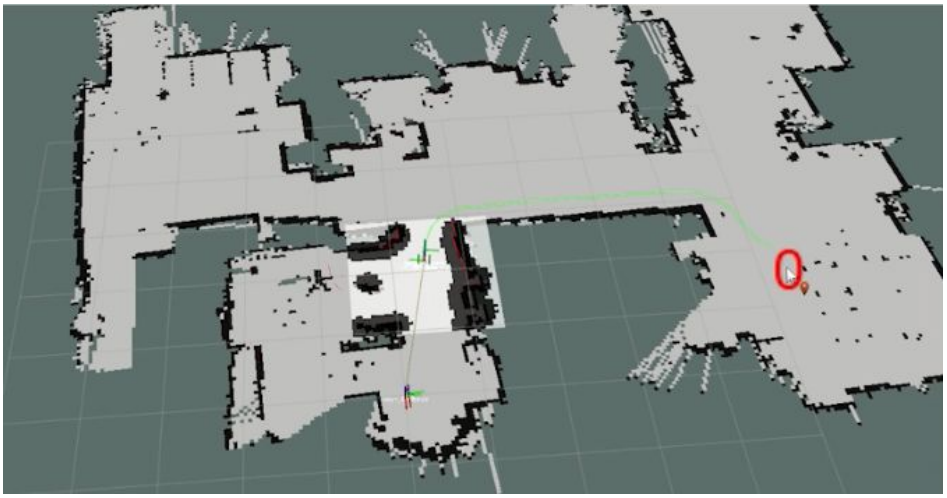


Node name: path_point			
Topic	Direction	Message type	Function
/clicked_point	Subscriber	geometry_msgs/PointStamped	Record the point for searching
/path_point	Publisher	visualization_msgs/MarkerArray	Publish all the search points
/move_base_simple/goal	Publisher	geometry_msgs/PoseStamped	Publish the navigation destinations
/move_base/result	Publisher & Subscriber	move_base_msgs/MoveBaseActionResult	Indicate whether a navigation is finished
/find_blue	Subscriber	geometry_msgs/Point	Indicate of the blue target appeared in cam
/grab_finish	Subscriber	geometry_msgs/Point	Indicate if the robot grab the object successfully
/grab	Publisher	geometry_msgs/Point	Ask the robot to put down the object in the end

a.record the input search point

```
marker.pose.position.x = msg.point.x # msg stores the clicked point info
marker.pose.position.y = msg.point.y
marker.pose.position.z = msg.point.z
marker.text = str(count)
markerArray.markers.append(marker) # add all clicked points to an array
id = 0
for m in markerArray.markers:  # record total number of clicked points
    m.id = id
    id += 1
mark_pub.publish(markerArray) # publish the array for visualization
```

Whenever the user clicks a point in Rviz, it will be stored as a pointstamp message and published by /clicked_point. Path point subscribes to it and stores every clicked in array form. To visualize the clicked point and the sequel for clicked, it publishes this array to /path_point. In Rviz, the result would be shown as below:



b.Navigate to these search point

```
if index < count:  # count if all clicked point has been used for navigation
    # publish the next point when the previous one finished
    pose = PoseStamped()
    pose.header.frame_id = 'map'
    pose.header.stamp = rospy.Time.now()
    pose.pose.position.x = markerArray.markers[index].pose.position.x
    pose.pose.position.y = markerArray.markers[index].pose.position.y
    pose.pose.orientation.w = 1
    goal_pub.publish(pose)
    index += 1
elif index == count:  # all point is finished
    print
    'finish all point'
```

All the search(clicked) point is recorded in an array in an order. This node publishes the points one by one to move_base for navigation by topic /move_base_simple/goal. After the navigation of a point is finished, it receives a message from /move_base/result (status = 3) implying that the robot has reached the input point and then it will publish the next point for navigation. If there is no further point, the robot will stop and wait.

c.Stop navigation when object appeared and navigate the robot to initial position if the robot has successfully grabbed the object

```
if find_blue == 1:   # if object appeared in screen

    if go_back == 0: # grab is not complete
        rospy.sleep(1) # sleep

    if go_back == 1:  # # grab is not complete, publish initial position
        index = count
        pose = PoseStamped()
        pose.header.frame_id = 'map'
        pose.header.stamp = rospy.Time.now()
        # 0 in markerArray indicates the first/initial point
        pose.pose.position.x = markerArray.markers[0].pose.position.x
        pose.pose.position.y = markerArray.markers[0].pose.position.y
        pose.pose.orientation.w = 1
        # publish this point
        goal_pub.publish(pose)
        find_blue = 2 # indicate the robot is returning
```

Once the object appears, find_blue will contain messages x = 1. Path point receives this message and sleeps until the grab is finished. Then, if the grab task is complete, path_point will get grab_finsih = 1 and then publish the initial position to move_base for return navigation.

d.Put down the object and mission complete

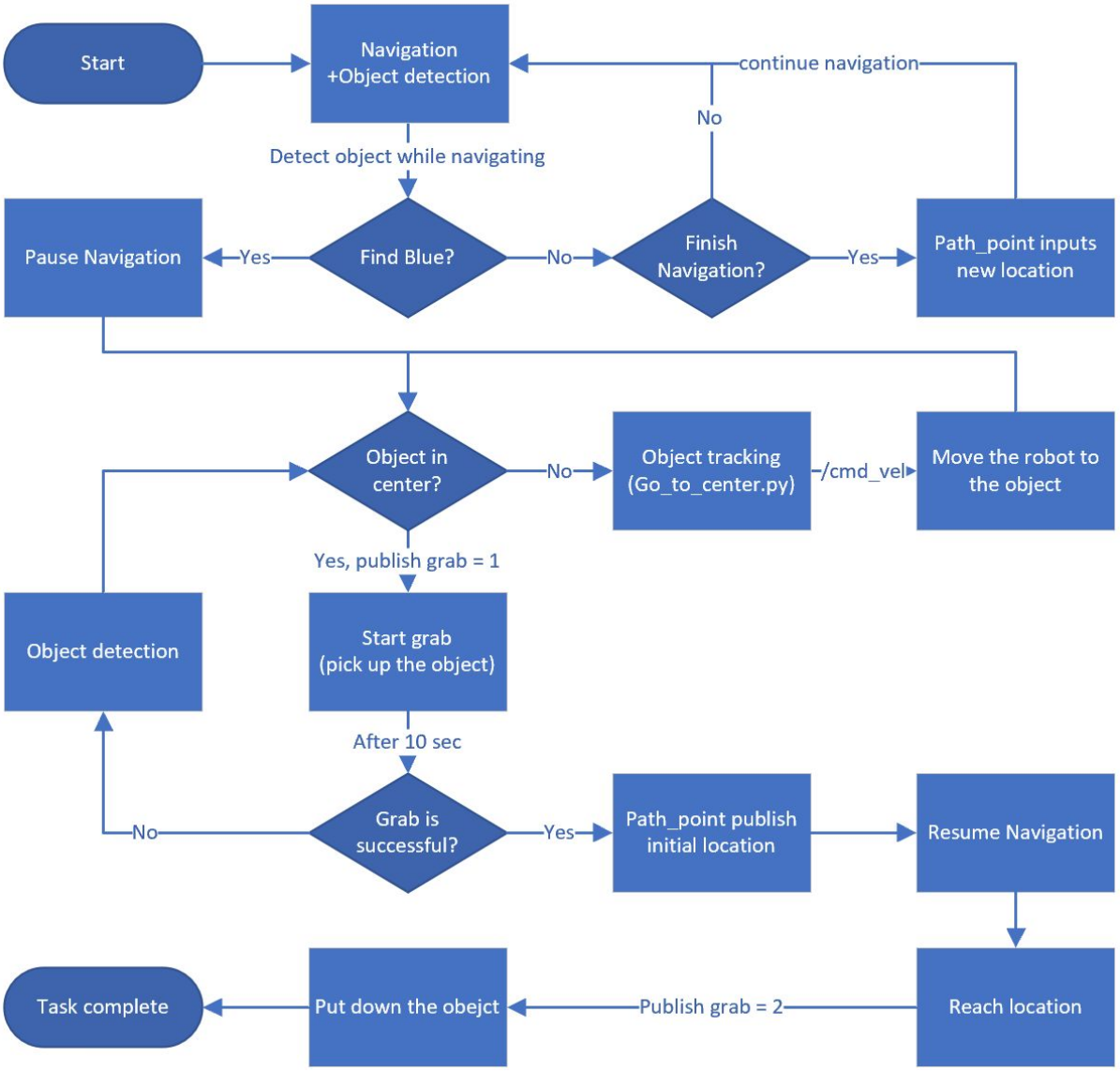
```
elif find_blue == 2: # when robot returns to initial position
    grab = Point()
    grab.x = 2
    grab_pub.publish(grab) # publish grab = 2 for putting down the object
    rospy.loginfo("task complete!")
    rospy.signal_shutdown("task finish") # all tasks are completed and kill the nodes
```

After the robot returns, path_point asks robot arm to put down the object by publishing to grab X = 2. Then, all tasks in the rescue mission are completed.

Integration

- SLAM
- Object Detection
- Object Tracking & Grab

The logic flow of this project:



The logic of our project is drawn in the above diagram. Since mapping is more independent from other functions, we simply ignore it here. We assume that we already had a map. Then it goes in four steps:

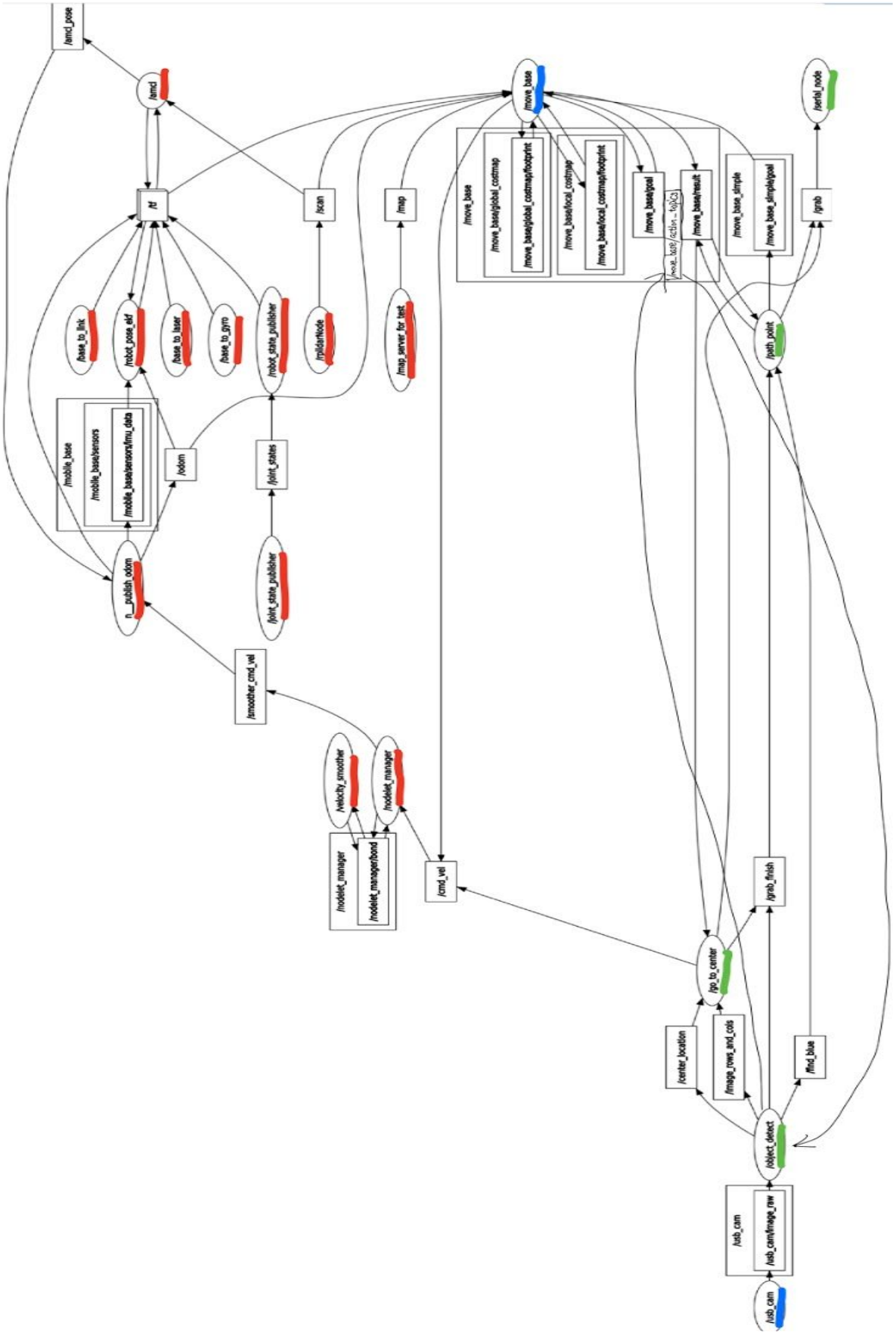
1. Mission started by navigation (node move_base) where the user input some possible points for searching. Node Path_point will store those points and publish them for navigation one after another.

2. During searching, if the object appears in the cam (find_blue=1), then navigation will pause and object detection will check if it is located in the center (zone for grab). Object tracking will drive Nano to move until the target reaches the grab zone. Once it is reached, the arm will receive the command (/grab=1) for picking up the target.

3. After grappling we would like object detection program to check the position of the object, if the object appears within the goal zone, we can tell the arm has catched the object (grab_finsih=1), else the arm will repeat catching until the object moves to a specific higher region in the camera.

4. If the target is successfully catched, path_point will send the initial location as the final goal for returning. And after Nano returns, the arm will receive the command (/grab=2) for putting down the target.

The information flow of the project Rqt_graph:



Node Explanation:

Node	Provider	Description
/nodelete_mangaer	ROS	Subscribe to required velocity and use it to move the base
/move_base	ROS	1.Get goal input by node path_point and navigate the robot to the goal 2.Cancel navigation when receiving the request 3.Indicate if a point is reached or not by /move_base/result
/usb_cam	ROS	publish the topic contains the real-time image information
/obejct_detect	ourselves	This is the node created by vision.py. It has 2 functions: 1.Subscribe the image message in /usb_cam and publish the object center position to /center_location 2.indicate if the target is grabbed successfully by /grab_finsih
/go_to_center	ourselves	This is the node created by go_to_center.py. 1.subscribe to the topic (/center_location) and publish the relative velocity topic (/cmd_vel) to track the object 2.order the arm to pick up the object(/grab=1)
/serial_node	Arduino	Arduino board communicated with Jetson Nano through “rosserial” and showed in rqt as /serial_node. This node subscribes to the topic (/grab). If the node receives X=1 in /grab, the robot will perform the grab action. If the node receives X=2 in /grab, the robot will perform the put down action.
/path_point	ourselves	This is the node created by show_mark.py. 1.record and publish searching points for navigation to the topic (/move_base_simple/goal) 2.make the robot return to initial position (/move_base_simple/goal) when grab finish (/grab_finish=1).

Topic Explanation:

Topic (msg type)	Publisher	Subscriber	Description
/cmd_vel (Twist)	Move_base & go_to_center	/nodelet manager	Contains three linear velocity and three angular velocity
/find_blue (Point)	/object_detect	/path_point	If X=1, it means the blue object exists in the video frame. The only value X can have is 1. Only published one time in the whole rescue mission.
/grab(Point)	/go_to_center & /path_point	/serial_node	If X=1, it means the robot can perform the grab action. If X=2, it means the robot can put down the object.
/center_location (Point)	/object_detect	/go_to_center	It contains the object center position in the frame.
/grab_finish (Point)	/object_detection	/go_to_center & /path_point	If X = 1, it means the grab task is finished.
/raw_image (image)	/usb_cam	/object detect	Original image captured by cam
/opencv_image (image)	/object detect	N/A	Image proceed by openCV, used for testing and visualization on PC
/image_rows_and_cols (Point)	/object detect	/go_to_center	The height and width of image
/move_base/result (MoveBaseActionResult)	/move_base &/path_point & /object detect	/path_point	If status = 3, navigation of current point is complet. If status=1, robot is going to the next goal position
/move_base_simple/goal (PoseStamped)	/path_point	/move_base	Destination for navigation

Action:

Action	Client	Server	Service
SimpleActionClient<ActionSpec>::cancelGoal()	/object_detect	/move_base	Cancel current goal and stop navigation

Limitation & Further development

- Remote control range problem
- Servo fast reset problem
- Self-Mapping
- Use Better Servo to Judge Grab Success
- Improve Battery System
- Improve Vision
- Improve Integrity of Robot Structure
- Control Grabbing Process base on Vision
- Implement to Logistic
- A more integrated structure

Command

Step1: Generate a map

roscore

roslaunch huanyu_robot_start Huanyu_robot_start.launch

roslaunch huanyu_robot_start gmapping_slam.launch

roslaunch turtlebot_teleop keyboard_teleop.launch

rviz

cd robot_ws/src/hunayu_robot_start/map >> open terminal

rosrun map_server map_saver -f map_name

Step2: Rescue robot

cd robot_ws/src/huanyu_robot_start/launch

gedit navigation_slam.launch >> change map filename

roslaunch huanyu_robot_start Huanyu_robot_start.launch

roslaunch huanyu_robot_start navigation.launch

rviz

roslaunch usb_cam usb_cam-test.launch

rqt_image_view

rosrun object detect go_to_center.py

rosrun huanyu_robot_start show_mark.py

rosrun rosserial_python serial_node.py /dev/ttyUSB0

rostopic pub /cmd_vel

Packages

Package			
Number	Name of the package	Major Provider	Description
1	huanyu_robot _start	Huanyu	Robot driver for base control, it communicates with Hibot (STM32) We add show_mark.py in it for recording navigation goals
2	hunayubot_description	Hunayu	Detail dimension of hunayu robot and transformations between components >> used in mapping and navigation
3	turtlebot_teleop	ROS	Keyboard control of robot velocity >> used in mapping
4	rplidar_ros	ROS	rplidar driver: get lidar sensory input >> used in mapping and navigation
5	robot_pose_ekf	ROS	Estimate the 3D pose of a robot, based on (partial) pose measurements coming from different sources >> used for mapping and navigation
6	slam_gmapping	ROS	Mapping
7	Navigation melodic	ROS	Navigation
8	rosserial_python	ROS	Connect Arduino to ROS
9	usb_cam	ROS	Camera driver, capture real-time image >> used for object detection and tracking
10	Object detect	Ourselves	Object detection and tracking
11	publisher	Ourselves	(For testing only) publish message to topic /grab and /grab_finish
12	opencv_move	Ourselves	(For testing only) object detection and tracking in python, independent from ROS

Note

1. Inside package object_detect/src, we add a file called find_hsv.py, it is used for finding the lower and upper hsv boundary for the target (the boundary values are then used for color detection)