MARINE & MARITIME INTELLIGENT ROBOTICS

Marine Mechatronics Practical work n°1

Visual servoing on the BlueRov

by Associate Prof. Claire Dune and Prof. Vincent Hugel (COSMER, Université de Toulon)

Visual servoing steps

Basically, visual servoing implementation is based on these steps:

- Feature detection and tracking
- Interaction Matrix computation
- Control law in the camera frame
- Control law computation in the robot frame

Visual servoing usage

This is how visual servoing is generally used. We don't give the desired features to the system, we calculate them from the desired view. This eliminates the need for calibration.

1. Position the robot in front of the target and click to record you desired features **s***
2. Move the robot in another position ***where it can still see the target***
3. Arm the robot with the joystick start button
4. Launch the control law with pressing A button
5. Disarm the robot

1. Install / update autonomous_rov package

Prerequisite (start with the section how to BlueROV)

- ros installed and configured with catkin
- python >=2
- autonomous_rov package installed and configured

Install additional template files to autonomous_ros package

- go to you catkin workspace :

```
cd ~/catkin_ws
cd src
cd autonomous_rov
```

- get the template files on the moodle and add them to your autonomous_rov package
    - launch

    - run_image_processing.launch
  - script
    - camera_parameters.py
    - image_processing_mir.py

You will find TODO and FIXME in these codes, each time your action is needed.

- source your catkin workspace :

```
source devel/setup.bash
```

2. Test offline

Test offline on a rosbag to define your tracking code

The purpose of this first code is to define a tracked point and a desired point and broadcast them through a ros topic to the visual servoing node. The tracked point will simply be the geometric center of an orange buoy. First convert the image to Hue, Saturation, Value mode, then set all non-orange pixels to zero and all orange pixels to 255. In openCV, you can find out how to create a threshold mask. Next, calculate the barycentre of pixels at 255.

- Set the env variable in your .bashrc configuration file, should be in your home directory

```
gedit ~\.bashrc
```

- And edit those lines

```
export ROS_MASTER_URI=http://127.0.0.1:11311
export ROS_HOSTNAME=127.0.0.1
export ROS_IP=127.0.0.1
```

- Open a terminal and run a roscore

```
roscore
```

- Open a new terminal and run the bag testtracking.bag with loop option so that it never stops publishing (replace xxx with bag name)

```
cd ~/catkin_w/src/autonomous_rov/bags
rosbag play -l [xxxxxxxx].bag
```

```
example :
rosbag play -l 2023-12-18-17-24-28.bag
```

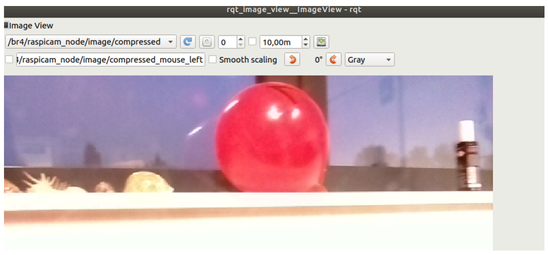
- Open a terminal and test the topics, you should see the camera topic

```
rostopic list
```

- Open a terminal and test the camera view,

```
rosrun rqt_image_view rqt_image_view
```

You should see a window opened with :


rqt_image_view__ImageView - rqt
Image View
/br4/raspicam_node/image/compressed
10,00m
/raspicam_node/image/compressed_mouse_left
Smooth scaling
0°
Gray

If not, select the right camera topic in the selection menu.

- Change the group name in the launch file to adapt it to your robot topic. Open the file run_image_processing.launch

  ```
  gedit ~/catkin_ws/src/autonomous_rov/launch/run_image_processing.launch
  ```

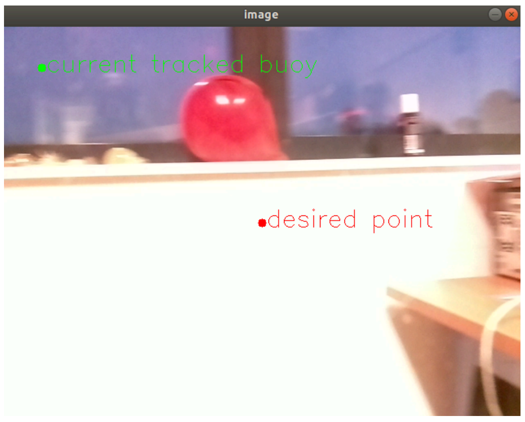
  Change the code with your rov number. Replace br5 with br1 , br2 , br3 or br4 …

  ```
  <group ns="br5">
  ```

- Launch the test :

  ```
  roslaunch autonomous_rov run_image_processing.launch
  ```

  You should see this window



- Look at the topics (rostopic list in a terminal). You should see two new topics broadcasted :

  ```
  /br4/tracked_point of type std_msgs/Float64MultiArray
  /br4/desired_point of type std_msgs/Float64MultiArray
  ```

  They are published by the node **image_processing_mir.py**. They contained the tracked and the desired point positions in the image in meters : [x,y] and [x*,y*]. They will be listened to in the visual servoing node.

The code to connect to the publisher is defined in *publishers* function line 102. The corresponding topic are broadcasted in *cameracallback* function

```
current_point_msg = Float64MultiArray(data = current_point_meter)
pub_tracked_point.publish(current_point_msg)
```

```
desired_point_msg = Float64MultiArray(data = desired_point_meter)
pub_tracked_point.publish(desired_point_msg)
```

You can see that the current tracked point is fixed. In order to make it track the buoy, you have to insert some image processing code to find the buoy and track the center point.

- Optional : If you want, you can also modify the code, so that the desired point is not always in the center of the view but is defined with a position of a mouse click, by using a callback function on mouse click, such as :

```
cv2.setMouseCallback("image", click_detect)

def click_detect(event,x, y, flags, param):
       global set_desired_points
       if event == cv2.EVENT_RBUTTONDOWN:
       rese_desired_points = True
       print ("desired_points to update")
       return x,y
```

3. Real robot application

- Kill roscore and the running nodes
- Turn on and connect the robot (see How To BlueROV)
- Set your network to

```
IPv4 to be adress = 192.168.254.15 Netmask = 255.255.255.0 Gateway = [empty]
```

Be careful that your firewall is off

- Set the env variable in you bashrc

```
gedit ~\.bashrc
```

And edit those lines

```
export ROS_MASTER_URI=http://192.168.254.15:11311
export ROS_HOSTNAME=192.168.254.15
export ROS_IP=192.168.254.15
```

- Change the group name in the launch file to adapt it to your robot topic. Open the file run_image_processing.launch

  ```
  gedit ~/catkin_ws/src/autonomous_rov/launch/run_image_processing.launch
  ```

  Change the code with your rov number. Replace br5 with br1 , br2 , br3 or br4 …

  ```
  <group ns="br5">
  ```

- Open a terminal and run a roscore
  ```
  roscore
  ```

- Open another terminal and run the launch file
  ```
  roslaunch autonomous_rov run_image_processing.launch
  ```

- Look at the topic : you should see two new topics :

  ```
  /br5/tracked_point of type std_msgs/Float64MultiArray
  /br5/desired_point of type std_msgs/Float64MultiArray
  ```

4. Visual Servoing

1. Create a visual servoing script
- copy ***listener_MIR.py*** in the script folder
- save it as ***visual_servoing_MIR.py***

2. Create a visual servoing launch file
- Create file ***run_visual_servoing.launch*** in the launch folder
- Do not forget to set the file privileges to executable (chmod a+x)
- Adapt the code to launch the ***visual_servoing_MIR.py script***

```
<?xml version="1.0"?>
<launch>
  <!-- roscore automatically started by roslaunch -->
  <!-- start mavlink node -->
  <!-- px4.launch already run by ROV -->
  <!-- <include file="$(find bluerov)/launch/apm.launch" /> -->
  <!-- start joy_node -->
   <arg name="ns" default="br5"/>
   <group ns="$(arg ns)">
    <arg
        name="arg1"
        default="--device /dev/ttyUSB0"
        />
    <node
        pkg="autonomous_rov"
        type="visual_servoing_MIR.py"
        name="visual_servoing_MIR"
        output="screen" >
        <param
            name="points"
            value="$(arg npoints)"/>
        </node>
  </group>
</launch>
```

3. Subscribe to the tracked point topics

- In ***visual_servoing_MIR.py*** add these lines to the main function to subscribe to the point topic :

```
rospy.Subscriber("tracked_point",Float64MultiArray,trackercallback,
queue_size=1)
rospy.Subscriber("desired_point",Float64MultiArray,desiredpointscallback,
queue_size=1)
```

4. Define the associated callback functions

You will need to define two callback functions for the two topics “tracked_point” and “desired_point”.

- Define the ***desiredpointcallback*** function. I suggest the desired data are spread through a global variable to access them in the trackercallback function

```
def desiredpointscallback(data):
 # print "desired point callback"
 global desired_point_vs
 desired_point_vs = data.data
```

- Define the ***trackercallback*** function. I suggest the visual servoing computation will be defined in this function. First use global variable desired_point_vs to get the desired points and get the current point as data.

```
def trackercallback(data):
    global desired_points_vs
    current_points = data.data
```

- Now, we will have to define the control law that gives us the robot velocity, but first, let us check that we are able to send a velocity screw in a direct robot frame

5. Ensure a direct robot frame in the trackercallback

   Consider the file listener_MIR.py and its ***setOverrideRCIN(channel_pitch, channel_roll, channel_throttle, channel_yaw, channel_forward, channel_lateral) (l.281).*** It send a velocity to the low level control of the robot. There is a mapping between velocity sent and the direction of the motion that depends on your robot thrusters.

   Let suppose I have finally defined a robot velocity screw vector

```
vrobot = [vx,vy,vz,wx,wy,wz]
```

   Now I want to convert it to a topic to send

```
vel = Twist()
vel.linear.x = vrobot[0]
vel.linear.y = vrobot[1]
vel.linear.z = vrobot[2]
vel.angular.x = vrobot[3]
vel.angular.y = vrobot[4]
vel.angular.z = vrobot[5]
```

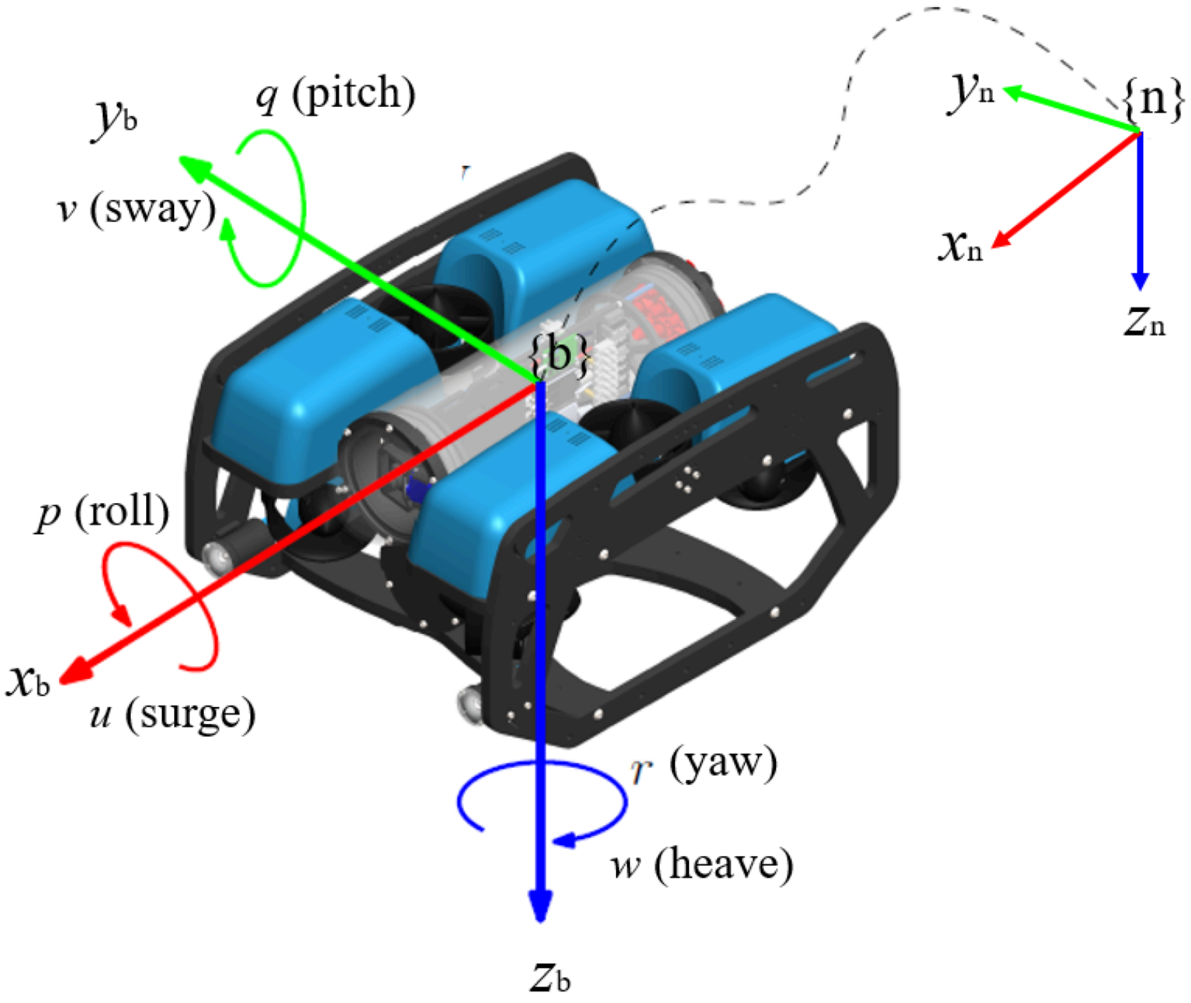
   And I want to convert the twist to a

```
forward_reverse = mapValueScalSat(cmd_vel.linear.x)
lateral_left_right = mapValueScalSat(-cmd_vel.linear.y)
ascend_descend = mapValueScalSat(cmd_vel.linear.z)
roll_left_right = mapValueScalSat(cmd_vel.angular.x)
pitch_left_right = mapValueScalSat(cmd_vel.angular.y)
yaw_left_right = mapValueScalSat(-cmd_vel.angular.z)
```

```
setOverrideRCIN(pitch_left_right, roll_left_right, ascend_descend,
                yaw_left_right, forward_reverse, lateral_left_right)
```

You might have noticed the - and + in the mapValueScaleSat fonction.
They map the orientation of the thruster with the low level.

To define those sign, you have to check that the obtained motion corresponds to a direct frame. Just test unit motion to find it !



To have a direct frame, if I set

- vrobot = [1,0,0,0,0,0], then the robot should move FORWARD, if not, change the sign of forward_reverse
- vrobot = [0,1,0,0,0,0], then the robot should move LATERALY ON THE RIGHT, if not, change the sign of lateral_left_right
- vrobot = [0,0,1,0,0,0], then the robot should dive DEEPERif not, change the sign of ascend_descend

        - vrobot = [0,0,0,1,0,0], then the robot should ROLL to RIGHT side down, if not, change the sign of roll_left_right
        - vrobot = [0,0,0,0,1,0], then the robot should PITCH noze UP, if not, change the sign of pitch_left_right
        - vrobot = [0,0,0,0,0,1], then the robot should TURN from LEFT to RIGHT, if not, change the sign of yaw_left_right

6. Compute visual servoing

- Compute the visual servoing error as the difference between the current point and the desired point
  `error_vs = …`
- Compute interaction matrix, test three scenario : current L, desired L, and half of both.
    - `currentL = interactionMatrix(currentPoint,Z=1)`
    - `desiredL =interactionMatrix(desiredPoint,Z=1)`
    - `mixteL = (currentL+desiredL)/2`
- Compute velocity of the camera
  `vcam_vs = …`

  Remember what we have seen in class. This control law computation depends on the number of degrees of freedom we want to control. Do not forget to prune the columns of L you do not need before inverting it.

- Change frame to compute the velocity of the robot. The visual servoing is defined in the camera frame. You have to find the transform between camera velocity screw and robot velocity screw. Position your camera model in the robot frame and find the homogeneous transform between the frame. Define the function ***transform*** that allows to compute vrobot_vs.
  `vrobot_vs = tranform(vcam_vs, homogeneous_transform_from_cam_to_robot)`

- Add publishers for
    - visual servoing error
    - vcam_vs
    - vrobot_vs

- Let us try the whole set up **on bags images !**
    - Launch the code
    - Display these topics in PlotJuggler
    - Record the bag **with all topics !**
    - **If the robot velocity displayed is coherent with the ball motion in the view, then turn on the robot and stop the rosplay**

- Let us try the whole set up **in air !**
    - Launch the code
    - Display these topics in PlotJuggler
    - Record the bags **with all topics !**
    - **If the robot velocity displayed is coherent with the ball motion in the view, then go into the water :)**

- Let us try the whole set up **in water !**

    - Launch the code
    - Display these topics in PlotJuggler
    - Record the bags **with all topics !**

These bags are important for your report ! Also save the plotjuggler plots.

5. Shut Down

- Disarm the robot
- Connect to the robot with
  `ssh ubuntu@192.168.254.xx`
- Shut it down
  `sudo shutdown -h 0`

APPENDIX A: Color detector

```
hsv = cv2.cvtColor(image_np, cv2.COLOR_BGR2HSV)

lower_bound = np.array([120, 177, 190])
upper_bound = np.array([192, 233, 255])

mask = cv2.inRange(hsv, lower_bound, upper_bound)
cv2.imshow('Mask', mask)

non_zero_pixels = cv2.findNonZero(mask)
if non_zero_pixels is not None and len(non_zero_pixels) > 0:
        mean_point = np.mean(non_zero_pixels, axis=0, dtype=np.int)
        cx, cy = mean_point[0]

        # Draw a circle at the center
        cv2.circle(image_np, (cx, cy), 5, (0, 255, 0), -1)

        print("Center Pixel of Blob: ({}, {})".format(cx, cy))
```

APPENDIX B: Get a pixel HSV on click

```
#To pick the right value for the thresholds


get_hsv = False

def click_detect(event,x, y, flags, param):
   global get_hsv,mouseX,mouseY
   if event == cv2.EVENT_LBUTTONDOWN:
          get_hsv =True
          mouseX, mouseY =x,y

def cameracallback(image_data):
…
         # get image data
         np_arr = np.fromstring(image_data.data, np.uint8)
         image_np = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
         hsv = cv2.cvtColor(image_np, cv2.COLOR_BGR2HSV)
         if get_hsv==True :
                  hsv_value = hsv[mouseY, mouseX]
                  print("HSV Value at ({}, {}): {}".format(mouseX, mouseY, hsv_value))
                  get_hsv=False
```

APPENDIX C: Add track bar to tune the thresholds

```
def on_trackbar_change(x):
    pass
# Create a window to display the result
cv2.namedWindow('Result')

# Create trackbars for threshold values
cv2.createTrackbar('Hue_Lower', 'Result', 0, 255, on_trackbar_change)
cv2.createTrackbar('Hue_Upper', 'Result', 30, 255, on_trackbar_change)
cv2.createTrackbar('Saturation_Lower', 'Result', 100, 255, on_trackbar_change)
cv2.createTrackbar('Saturation_Upper', 'Result', 255, 255, on_trackbar_change)
cv2.createTrackbar('Value_Lower', 'Result', 100, 255, on_trackbar_change)
cv2.createTrackbar('Value_Upper', 'Result', 255, 255, on_trackbar_change)


def cameracallback(image_data):

….

        hue_lower = cv2.getTrackbarPos('Hue_Lower', 'Result')
        hue_upper = cv2.getTrackbarPos('Hue_Upper', 'Result')
        saturation_lower = cv2.getTrackbarPos('Saturation_Lower', 'Result')
        saturation_upper = cv2.getTrackbarPos('Saturation_Upper', 'Result')
        value_lower = cv2.getTrackbarPos('Value_Lower', 'Result')
        value_upper = cv2.getTrackbarPos('Value_Upper', 'Result')

        # Define the lower and upper bounds of the orange color in HSV based on trackbar
values
        lower_bound = np.array([hue_lower, saturation_lower, value_lower])
        upper_bound = np.array([hue_upper, saturation_upper, value_upper])
```