The system will have four launch files embedded in mavros_off_board and these will be used to launch the vehicle, the functionality of each launcher is:

- posix_sitl.launch: It launches PX4 SITL in Gazebo
- mavros_posix_sitl.launch: This launch file launches Mavros, PX4 SITL and Gazebo. This launch file will allow the control of the vehicle with ROS. The model launched will be in sdf format.
- mavros_rviz.launch: This launch file will only be for visualization purpose and shows the vehicle in Gazebo and Rviz. The model launched will be in urdf format.
- urdf_launcher.launch This launch file will start Mavros, PX4 SITL, Rviz and Gazebo. This launch file will allow the control of the vehicle with ROS. The model launched will be in xacro format.

Usage of launch files would be different based on needs. We would be using mavros_posix_sitl.launch for this pseudo code example.

#Launching simulated environment with the package mavros_off_board.

- make px4_sitl_default gazebo
- `roslaunch mavros_off_board mavros_posix_sitl.launch`

mavris_posix_sitl.launch:

- setting configurations: vehicle positions, vehicle model and world, gazebo configs, mavros and px4 configs

#This will deploy the Gazebo world created with the UAV, allowing further iteration of the vehicle.

#Once the simulation is running, Take-off the vehicle and start its movement. Place the drone 50m above the target and descent towards 20m height.

```
rosrun mavros_off_board offb_node
rosrun mavros_off_board teleop_node_pos  #uses mavros_off_board package
```

#Move the vehicle with the keyboard along the simulation space and locate it above the landing pad.

- Importing massage type PositionTarget from marvos_msgs.msg
- Defining keybinds and speedbinds as a dictionary
- Publishes to the topic /mavros/setpoint_raw/local (controls speed and pos)(rospy.Publisher)
- Rospy.init_node #for telling rospy the name of the node
- Create a proxy to call services and enable persistent connections for takeoff, landing, and changing mode service. (using rospy.ServiceProxy)
- Setting linear and angular speed (using rospy.get_param), and initializing other parameters

#Next step would be to use detection pipeline in object_detector. (for detecting aruco marker)

roslaunch object_detector detector.launch

this launch file will execute 4 nodes under the </node> section, each performing it's individual task. (under object_detector package)

Node1: corners_detector.py

- Importing libraries for Custom msg with the corners and centroid of the object, and to do the perspective transforms. (object_detector/corners.h,<Qtransform>)
- Creating a class called Detections

Under public, the class will have 3 main methods: Detections(), cornersDetectedCallback, assignCorners().

- Detections(): contains details of publishers and subscriber.
- Publishes to the topic /corners (Publisher type object_detector::corners) (using po_nh.advertise)
- Subscribers to the topic /objects from find_object_2d/ObjectsStamped (using po_nh.subscribe)
- cornersDetectedCallback(): method for subscriber callback
- Creation of a Corners object to publish the info.
- extract data given by the find object 2d topic.
- Find corners QT. QTrasform initialization with the homography matrix.
- Map the template coordinates to the current frame with the shape of the template and homography.
- Assign coordinates to the publishing object: assignCorners(points, coordinates)
- Assign centroid values: coordinates/centerX and centerY.
- Assign zero to all the members of the publishing object if nothing is detected
- Publish coordinates: pub.publish(coordinates)
- assignCorners():
- Assign the bottom left and top left corners.
- Assign the bottom right and top right corners.
- Ros_init to name the node "Corners_detected"

Node2: cent_calculator.py

Importing libraries for Custom msg of Corner and States (“object_detector/Corners.h”, "object_detector/States.h")

Creating a class called state

Under public the class 2 main methods: States() and centCalculator().

- 1. Sates (): contains details of publishers and subscriber.
- Publishes to the topic /states (Publisher type object_detector::States) (using po_nh.advertise)
- Subscriber to /corners topic from object_detector/Corners (using po_nh.subscribe)
- 2. centCalculator(): method for subscriber callback
- Creation of a States object to publish the info.
- Theta angle calculation (computeTheta())
- Right, left, bottom, top vactor magnitude calculation (computeEucDist()) ()using values form the node corners_detector.py)
- Compute Theta() and computeEucDist() for calculating the angle with respect to x and Euclidian distance between two points.
- Assigning centroid of the template in the image
- Computation of width and height as an average of the bottom and top magnitudes
- Correction of theta when the angle is bigger than 90 degrees to avoid bad orientation for the controller.

```
if(theta > 90)
        {
                int cont = theta / 90;
                st.Theta = theta - cont * 90;
        }
else if(theta < 90)
        {
                int cont = -1 * theta / 90;
                st.Theta = theta + cont * 90;
        }
```

- Ros_init to name the node "data_calculation"

Node3: Kalman_filter.py

- Importing library for custom msg of types States
- Creating a class called Kalman:

Under public, it will have 2 main methods: Kalman() (constructor) and predictionsDetectedCallback().

- 1. Kalman():
- Publisher type object_detector::States, it publishes in /predicted_states topic
- Subscriber to /states topic from object_detector/States
- Posteriori estimate covariance matrix initialization
- Initialization of values like Covariance of innovation, State Vector, Kalman gain, State transition matrix, and innovation vector.
- // code example of State vector initialization
  this->X = MatrixXd::Zero(10,1);
- 2. predictionsDetectedCallback(): method for subscriber callback
- Creation of a States object to publish the info.
- Assigning a vector to store the observations
- initialize the states with the observation
- Example:

```
if(this->first_iter==0)
        {
          this->X(0,0) = msg.Xc; // State xc
          this->X(1,0) = msg.Yc; // State Yc
          this->X(2,0) = msg.W; // State Width
          this->X(3,0) = msg.H; // State Height
          this->X(4,0) = msg.Theta; // State Theta
          this->X(5,0) = 0; // State Xc'
          this->X(6,0) = 0; // State Yc'
          this->X(7,0) = 0; // State Width'
          this->X(8,0) = 0; // State Height'
          this->X(9,0) = 0; // State Theta'
          this->first_iter = 1;
        }
```

- assign the observations to the measurement vector
- Assign the predictions to the publisher object (pub.publish(predictions))
- Ros_init to name the node "KF_predictor"

Node4: plot_estimation.py

- Importing necessary libraries for custom msg of type states
- Creating a class called Drawer

Under public, the class will have 3 main methods:

- 1. Drawer(): it is a constructor used for details regarding publisher and subscriber
- Define two subs: sub and image_sub. Image_sub will subscriber to /image_raw topic and 'sub' will subscribe to /predicted_states topic from object_detector/States.
- Publisher type sensor_msgs, it publishes in /rectangle_draw/raw_image topic.
- 2. estimationsDetectedCallback():Subscriber callback to assign the values of the states to an array
- Example:

```
{
  this->data[0] = msg.Xc;
  this->data[1] = msg.Yc;
  this->data[2] = msg.W;
  this->data[3] = msg.H;
  this->data[4] = msg.Theta;
}
```

- 3. imageCb(): To draw object in the image frame
- Copy image to cv_bridge pointer (cv_bridge::toCvCopy)
- Draw the object detected in the image frame
- Example:
  rectangle_draw(cv_ptr->image);
  image_pub_.publish(cv_ptr->toImageMsg());

To land the vehicle we will use a drone_controller package which will be already defined. The process variable of the controller is the output of the detection pipeline.

rosrun drone_controller pid_controller_final

- Importing libraries for custom msg of types States and Error
- Import service for landing <mavros_msgs/CommandTOL.h>
- Define descent factor (#define FACTORZ 0.02)
- Creating a class controller

Under public, the class will have 2 main methods: Controller (constructor) and controllerCallback

- 1. Controller:
- Define PID controller objects (max, min, kp, kd, ki)
- Define a descend factor (define FACTORZ 0.025)

- Initialize two publishers as pub and pub1
- Pub: Publisher type mavros_msgs::PositionTarget, it publishes in /mavros/setpoint_raw/local topic
- Pub1: Publisher type drone_controller::Error, it publishes in /error topic
- Subscribing to topic to /predicted_states from object_detector/Corners
- Important: Defining landing client
- land_client = po_nh.serviceClient<mavros_msgs::CommandTOL>("mavros/cmd/land");
- And definiging initial altitude (20m in our case)
- 2. callbackController: method for subscriber callback
- Error Calculation between image and template's center (X,Y,Theta etc)
- Publish the error (drone_controller::Error er;)
- If the error between width and height is less than 4 pixels and height
  {
  Descend Z based on the factor (zpos = zini - FACTORZ;)
  }
  Else
  {
  If there is more than 3 pixels of error, hold pos
  }
- Drone service for automatic langing when it reaches an specific altitude and centroid conditions.
- If distance between drone and the landing zone (zpose) reaches very close to zero
  {
  Set all the descend parameters to Zero (land_cmd.request.latitude = 0;
  land_cmd.request.longitude = 0;
  land_cmd.request.altitude = 0;)
  }
  # When it lands, everything goes to zero
- Publish the service for landing (ROS_INFO("Landing"))
- Print and publish the final error (pub1.publish(er))
- Shutdown the node
- // Update vehicle's position
  zini = zpos;
- Compute controller output for each axis
- Position target object to publish (mavros_msgs::PositionTarget pos)
- Ros-init to name the node "controller_node"