EKF SLAM Implementation

This Notebook contains codes for implementing Extended Kalman Filter (EKF) SLAM on a JetBot.

In [ ]:
```
import os
import pickle
import cv2
import numpy as np
import math
import time

%matplotlib inline
import matplotlib.pyplot as plt
import ipywidgets.widgets as widgets
from IPython.display import display

from jetbot import bgr8_to_jpeg
from jetbot import ObjectDetector
from jetbot import Camera
from jetbot import Robot
```

Startup JetBot

In [ ]:
```
model = ObjectDetector('../Notebooks/object_following/ssd_mobilenet_v2_coco.engine')
camera = Camera.instance(width=300, height=300)
robot = Robot()
```

In [ ]:
```
# Load COCO labels
filename = "coco_labels.dat"
filehandler = open(filename, 'rb')
COCO_labels = pickle.load(filehandler)

# Load camera calibration data for undistort
filename = "calibration.dat"
filehandler = open(filename, 'rb')
camera_cal = pickle.load(filehandler)
mtx = camera_cal['mtx']
dist = camera_cal['dist']
f_u = mtx[0,0]  # focal lengths in u pixels (image plane horizontal)
f_v = mtx[1,1]  # focal lengths in v pixels (image plane vertical)
c_u = mtx[0,2]  # focal center in u pixels (image plane horizontal)
c_v = mtx[1,2]  # focal center in v pixels (image plane vertical)
focal_center = np.array([c_u, c_v])

# Open Image Widget
image_widget = widgets.Image(format='jpeg', width=300, height=300)
width = int(image_widget.width)
height = int(image_widget.height)

BLUE = (255, 0, 0)
GREEN = (0, 255, 0)
RED = (0, 0, 255)

diag_dir = 'diagnostics'
# we have this "try/except" statement because these next functions can throw an erro
try:
    os.makedirs(diag_dir)
except FileExistsError:
    print('Directories not created because they already exist')

# Mapping between set_motor "speed" and measured wheel angular velocity "omega"
# for 0.1 second motor running time
wheel_calibration = {
    "speed": [0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8],
    "omega": [0.0, 3.85, 9.23, 15.0, 25.8, 29.2, 35.4]
}

plt.plot(wheel_calibration["speed"],wheel_calibration["omega"])
plt.ylabel('Omega')
# fig = plt.figure(figsize=(4, 5))
plt.savefig('test.png')
plt.show()
plt.close()
```

Shared Functions

In [ ]:
```
def normalize_angle(angle):
    """ Normalize angle to between +pi and -pi """
    """ Important for EKF Correction Step !!!  """
    return (angle+math.pi)%(2*math.pi)-math.pi

def forward(wheel_speed, Rtime):

    robot.set_motors(wheel_speed, wheel_speed)
    time.sleep(Rtime)
    robot.stop()

    return

def control2robot(wheel_radius, axle_length):
    """ transform wheel speeds to robot motion in world frame """
    l = axle_length
    r = wheel_radius

    return np.array([[r/2, r/2],
                     [r/l, -r/l]])

def omega2speed(in_val, mapping, debug=False):
    """ Map wheel angular speed to motor speed setting based on a calibration mappin

    if in_val < 0:
        sign = -1
        in_val = abs(in_val)
    else:
        sign = 1

    out_lower = 0
    in_lower = 0
    out_val = 0

    for i, in_upper in enumerate(mapping["omega"]):
        if debug:
            print (i, in_upper)
        if in_val < in_upper:
            out_upper = mapping["speed"][i]
            out_val = out_lower + (in_val - in_lower)/(in_upper - in_lower) \
                *(out_upper-out_lower)
            if debug:
                print("yes", out_val)
            break
        else:
            if debug:
                print("no")
            out_lower = mapping["speed"][i]
            in_lower = in_upper

    if out_val is 0:
        print ("Input is too high!!!", in_val)
        out_val = 0

    return sign*out_val


def calc_wheel_velocities(direction='L', arc_radius=0.5, min_ang_vel=3.85, \
       wheel_radius=0.0325, axle_length=0.12, debug = False):
    """ Calculate wheel velocities to generate forward arc motion of provided radius

    radius = arc_radius
    axle = axle_length
```

```
    if direction is 'L':
        """ If left turn, angular velocity of right wheel should be higher.
        Set angular velocity of left wheel to minumum (e.g. 3.85--> motor setting of
        l_ang_vel = min_ang_vel
        r_ang_vel = (min_ang_vel*2)/(2*radius/axle-1)+min_ang_vel
    else:
        """ If right turn, angular velocity of left wheel should be higher.
        Set angular velocity of right wheel to minumum (e.g. 3.85--> motor setting o
        r_ang_vel = min_ang_vel
        l_ang_vel = (min_ang_vel*2)/(2*radius/axle-1)+min_ang_vel

    if debug:
        print ("Left angular velocity:",l_ang_vel, " Right angular velocity:",r_ang_
        T = control2robot(wheel_radius, axle_length)
        robot_velocities = np.dot(T, np.array([[r_ang_vel],[l_ang_vel]]))
        print ("Robot velocities:", robot_velocities)
        print("arc radius = ",abs(robot_velocities[0,0]/robot_velocities[1,0]))

    return np.array([[r_ang_vel],[l_ang_vel]])

def update_map(Mu, landmarks, Mu_prev=None, folder=None, ind=None, debug=False):
    """ Update robot position on map """
    plt.figure(figsize=(8,8))
    plt.xlim([-100,250])
    plt.ylim([-100,250])

    """ Display robot as line + triangle (arrow) """
    robot_x = Mu[0,0]*100
    robot_y = Mu[1,0]*100
    robot_theta = Mu[2,0]*180/math.pi - 90 # Adjust orientation to match matplotlib

    if debug:
        print("(x,y):{:.1f}, {:.1f}".format(robot_x,robot_y))
        print("Orientation: {:.1f}".format(normalize_angle(Mu[2,0])*180/math.pi))

    # robot = line + triangle
    plt.plot(robot_x, robot_y, marker=(2, 0, robot_theta), c='k',markersize=15, line
    plt.plot(robot_x, robot_y, marker=(3, 0, robot_theta), c='k',markersize=10, line

    """ Display landmark as green cross, uncorrected landmark as lime cross """
    for i, landmark in enumerate(landmarks):
        if landmark['observed'] is True:
            # Mark landmark's actual locations
            landmark_x_actual = landmark["actual_x"]*100
            landmark_y_actual = landmark["actual_y"]*100
            plt.plot(landmark_x_actual, landmark_y_actual, marker='*', markersize=8,
            # Mark landmark (post-Kalman correction)
            landmark_x = Mu[3+2*i]*100
            landmark_y = Mu[3+2*i+1]*100
            plt.plot(landmark_x, landmark_y,marker='x', markersize=12, color='blue')
            plt.text(landmark_x, landmark_y+10, landmark["obj_name"])

    if (folder is not None) and (ind is not None):
        file_path = os.path.join(folder, 'map_'+str(ind+1).zfill(3)+'.png')
        plt.savefig(file_path)
    else:
        plt.show()

    plt.close()
    return

def undistort(img, mtx, dist, crop=False):
    """Undistort camera image based on calibration data"""
    h,w = img.shape[:2]
    # print (h,w)
```

```
    newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))

    # undistort
    dst = cv2.undistort(img, mtx, dist, None, newcameramtx)

    # crop the image (optional)
    if crop:
        x,y,w,h = roi
        dst = dst[y:y+h, x:x+w]
    return dst

def draw_bbox(img, width, height, bbox, color, line_width):
    bbox_pixel = [(int(width * bbox[0]), int(height * bbox[1])),
                  (int(width * bbox[2]), int(height * bbox[3]))]
    cv2.rectangle(img, bbox_pixel[0], bbox_pixel[1], color, line_width)
    return bbox_pixel

def valid_bbox(bbox, width, height):
    """ Detect if an object borders the 4 edges of the camera image plane.
    Used to disregard a landmark for range and bearing estimation. """
    return (width * bbox[0]>10) and \
           (width * bbox[2]<width-10) and \
           (height * bbox[1]>10) and \
           (height * bbox[3]<height-10)

def display_landmarks(img, landmarks, width, height, debug=False):
    """ put blue bounding boxes on detected objects on image """

    for item in landmarks:
        label = COCO_labels[item['label']-1]
        bbox = item['bbox']
        bbox_pixel = draw_bbox(img, width, height, bbox, BLUE, 1)
        if debug:
            print(label,item['label'], bbox_pixel)
    return

def landmark_coordinates(item, width, height):
    """ calculate landmark's left, center and right in image pixel coordinates """
    u_left = item['bbox'][0] * width
    u_right = item['bbox'][2] * width
    u_center = (item['bbox'][0]+item['bbox'][2])*width/2
    return u_left, u_center, u_right


""" =================== """
""" EKF Functions       """
""" =================== """

def initialize_Mu_Sigma(x,y,theta,landmark_list):
    """ Initialize Mu and Sigma """
    # Initialize Mu
    Mu = np.array([[x],[y],[theta]])
    for object in range(len(landmark_list)):
        Mu = np.vstack((Mu,np.array([[0],[0]])))
    N = Mu.shape[0]   # N=3+2n, n=num of landmarks

    # Initialize Sigma - For Σ_mm, infinity (large num) along the diagonal and zero 
    Sigma = np.zeros((N,N))
    Sigma[3:,3:] = np.eye(N-3)*LARGE

    return Mu, Sigma

def robot_pose_delta(v,w,theta,dt):
    """ Calculate change in robot pose in world frame """
```

```
    """ An alternate way - can avoid divide by zero error if w is zero
    x_delta = v*dt*math.cos(theta)
    y_delta = v*dt*math.sin(theta)
    """
    arc_radius = v/w   # arc radius

    x_delta = arc_radius*(math.sin(theta+w*dt)-math.sin(theta))
    y_delta = arc_radius*(math.cos(theta)-math.cos(theta+w*dt))
    theta_delta = w*dt

    return x_delta, y_delta, theta_delta

def compute_G_t(v,w,theta,dt,N):
    """ Calculate G_t matrix """
    F = np.zeros((3,N))
    F[0:3,0:3] = np.eye(3)

    """ An alternate way to avoid divide by zero error if w is zero """
    """ TBD """

    arc_radius = v/w   # arc radius

    d_x_delta = arc_radius*(math.cos(theta+w*dt)-math.cos(theta))
    d_y_delta = arc_radius*(-math.sin(theta)+math.sin(theta+w*dt))

    G_x_t = np.array([[0,0,d_x_delta],[0,0,d_y_delta],[0,0,0]])
    G_t = np.eye(N)+ np.dot(np.dot(F.T, G_x_t),F)

    return G_t

def prediction_step_update(Mu, Sigma, x_delta, y_delta, theta_delta, G_t, R_t, N):
    """
    Implement:
         Mu_t = g(Mu_t-1, u_t)
         Sigma_t = G_t.Sigma_t-1.G_t^T + F^T.R_t.F
    """
    F = np.zeros((3,N))
    F[0:3,0:3] = np.eye(3)
    Mu = Mu + np.dot(F.T, np.array([[x_delta],[y_delta],[theta_delta]]))
    Sigma = np.dot(np.dot(G_t, Sigma),G_t.T) + np.dot(np.dot(F.T, R_t),F)

    return Mu, Sigma

def estimate_range_bearing(landmarks, index, u_l, u_c, u_r, c_u, focal_length,\
                           correct_factor, camera_offset):
    """ Estimate landmark range (r) and bearing (phi) """

    j = index  # j is landmark index
    landmark_width = landmarks[j]['width']   # we know landmark's real width

    # First calculate range and bearing from camera's focal center
    phi = math.atan2(c_u-u_c, f_u) # phi is +ve if landmark is left of focal center
    depth = f_u/(u_r-u_l)*landmark_width*correct_factor-focal_length
    r = depth/math.cos(phi)

    # Next calculate range and bearing from robot's center of motion
    true_phi = math.atan2(r*math.sin(phi), depth+focal_length+camera_offset)
    true_r = (depth+focal_length+camera_offset)/math.cos(true_phi)

    return true_r, true_phi

def update_landmark_Mu(Mu, j, r, phi):
    """ Update j-th landmark's x and y coordinate in Mu """

    x = Mu[0,0]
```

```
    y = Mu[1,0]
    theta = Mu[2,0]

    Mu[3+2*j,0] = x + r*math.cos(phi+theta)
    Mu[3+2*j+1,0] = y + r*math.sin(phi+theta)

    return Mu

def get_observation(Mu, j):
    """ Get landmark's updated observation """
    robot_x = Mu[0,0]
    robot_y = Mu[1,0]
    robot_theta = Mu[2,0]
    landmark_x = Mu[3+2*j,0]
    landmark_y = Mu[3+2*j+1,0]
    delta_x = landmark_x-robot_x
    delta_y = landmark_y-robot_y

    delta = np.array([[delta_x],[delta_y]])
    q = np.asscalar(np.dot(delta.T, delta))
    # important to normalize orientation (otherwise fatal error in EKF!!!)
    phi = normalize_angle(math.atan2(delta_y,delta_x)-robot_theta)
    z_t_hat = np.array([[math.sqrt(q)],[phi]])

    return delta, q, z_t_hat

def compute_H_t(delta, j, q, N):
    """ Compute H_t matrix """
    delta_x = delta[0,0]
    delta_y = delta[1,0]

    F = np.zeros((5,N))
    F[0:3,0:3] = np.eye(3)
    F[3:5,3+2*j:3+2*j+2] = np.eye(2)

    sqrt_q = math.sqrt(q)
    H_j = np.array([[-sqrt_q*delta_x, -sqrt_q*delta_y,   0, sqrt_q*delta_x, sqrt_q*de
                    [        delta_y,        -delta_x,  -q,       -delta_y,        de
    H_t = 1/q*np.dot(H_j,F)

    return H_t

def compute_Kalman_Gain(delta, q, j, H_t, Sigma, N, Q_t):
    """ Compute Kalman Gain K_t """
    delta_x = delta[0,0]
    delta_y = delta[1,0]

    # Compute Kalman Gain K_t
    L = np.dot(np.dot(H_t, Sigma),H_t.T) + Q_t
    K_t = np.dot(np.dot(Sigma, H_t.T),np.linalg.inv(L))

    return K_t
```

In [ ]:
```
""" ======================== """
""" Open-End Circular Motion """
""" ======================== """

def take_circ_step(robot_params, direction, radius, debug=False, motion=True):
    """ Open-end control for circular motion - Taking one step in the trajectory """

    # load robot control parameters
    start_x = robot_params["start_x"]
    start_y = robot_params["start_y"]
    wheel_radius = robot_params["wheel_radius"]
    axle_length = robot_params["axle_length"]
    motor_on_time = robot_params["motor_on_time"]
    motor_off_time = robot_params["motor_off_time"]
    min_ang_velocity = robot_params["min_ang_velocity"]

    """ Generate clamped wheel velocities based on turn direction and radius """
    wheel_velocities = calc_wheel_velocities(direction='L', arc_radius=radius, \
        min_ang_vel=min_ang_velocity, \
        wheel_radius=wheel_radius, axle_length=axle_length, debug = debug)

    """ Map wheel angular velocities to motor setting, then run motors """
    w_r = omega2speed(wheel_velocities[0,0],wheel_calibration)
    w_l = omega2speed(wheel_velocities[1,0],wheel_calibration)
    if debug:
        print ("L motor:", w_l," R motor:", w_r)

    """ Run motor step motion """
    if motion:
        robot.set_motors(w_l, w_r)  # left, right
        time.sleep(motor_on_time)
        robot.stop()
        time.sleep(motor_off_time)

    return wheel_velocities
```

EKF Setup and Initialization

In [ ]:
```
robot_params = {
    # pose
    "start_x": 1.25,
    "start_y": 0.75,
    "start_theta": math.pi/2,
    # physical dimensions
    "wheel_radius": 0.0325,
    "axle_length": 0.12,
    "camera_offset": 0.06, # camera is +6cm from center of wheel axle
    # stepwise motor control
    "motor_on_time": 0.1,
    "motor_off_time": 0.2,
    # wheel velocity control
    "min_ang_velocity": 6.5, # Equivalent to motor speed setting of 0.3
    "focal_length": 0.00315 # camera focal lenght in meter
}

control_params = {
    "num_iter": 201,
    "interval": 5,
    "debug": False,
    "motion": True,
    "radius": 0.50,    # radius of circular trajectory
}

landmarks = [
    {
    "label": 19,
    "obj_name": 'horse',
    "width": 0.394,
    "observed": False,
    "actual_x": 1.50,
    "actual_y": 1.50,
    "Mu": [],
    },
    {
    "label": 13,
    "obj_name": 'stop sign',
    "width": 0.12,
    "observed": False,
    "actual_x": 1.25,
    "actual_y": 0,
    "Mu": [],
    },
    {
    "label": 44,
    "obj_name": 'bottle',
    "width": 0.10,
    "observed": False,
    "actual_x": 0.75,
    "actual_y": 1.50,
    "Mu": [],
    },
    {
    "label": 63,
    "obj_name": 'couch',
    "width": 1.68,
    "observed": False,
    "actual_x": -0.50,
    "actual_y": 1.21,
    "Mu": [],
    },
    {
    "label": 72,
```

```
    "obj_name": 'TV',
    "width": 1.00,
    "observed": False,
    "actual_x": 2.0,
    "actual_y": 0.9,
    "Mu": [],
    },
    {
    "label": 64,
    "obj_name": 'potted plant',
    "width": 0.55,
    "observed": False,
    "actual_x": 0.5,
    "actual_y": -0.10,
    "Mu": [],
    }
]

landmark_item_list = []
for item in landmarks:
    landmark_item_list.append(item['label'])

# Display camera image with bounding boxes for detected objects
display(widgets.HBox([image_widget]))
image = undistort(camera.value, mtx, dist) # undistort camera image
image_widget.value = bgr8_to_jpeg(image)  # update image widget with camera image

LARGE = 1e6
t_delta = 0.1  # motor on time
R_t = np.eye(3)*0.001  # Assume small constant control noise for now
Q_t = np.eye(2)*0.001  # Assume small constant  measurement noise for now

diag_dir = 'diagnostics'
np.set_printoptions(precision=5)

# Load camera parameters
focal_length = robot_params["focal_length"]
correct_factor = 1.0 # 0.769
camera_offset = robot_params["camera_offset"]

# Load robot parameters
wheel_radius = robot_params["wheel_radius"]
axle_length = robot_params["axle_length"]
T = control2robot(wheel_radius,axle_length)
x = robot_params["start_x"]
y = robot_params["start_y"]
theta = robot_params["start_theta"]

# load control parameters
num_iter = control_params["num_iter"]
interval = control_params["interval"]
motion = control_params["motion"]
debug = control_params["debug"]
radius = control_params["radius"]

""" Initialize Mu and Sigma """
Mu, Sigma = initialize_Mu_Sigma(x,y,theta,landmark_item_list)
# Mu_prev = Mu
if debug:
    print(Mu)
    print(Sigma)

# Place robot (and landmark)on map
update_map(Mu, landmarks, debug=True)
```

In [ ]:
```
# Display camera image with bounding boxes for detected objects
display(widgets.HBox([image_widget]))

""" Robot moves stepwise in a circle """
for i in range(num_iter):

    """ Move robot - Take 1 step in a left circular trajectory of radius 0.4m """
    wheel_velocities = take_circ_step(robot_params, 'L', radius, debug=debug, motion
    robot_velocities = np.dot(T,wheel_velocities)   # calculate (v,omega)
    if debug:
        print("Step: ", i+1)
        print("(w_r,w_l): {}".format(wheel_velocities))
        print("(v,omega): {}".format(robot_velocities))

    """ =================== """
    """ EKF Prediction Step """
    """ =================== """

    v = robot_velocities[0,0]
    w = robot_velocities[1,0]
    theta = Mu[2,0]%(2*math.pi)  # robot orientation (normalize to 2*pi)
    N = Mu.shape[0]   # N=3+2n, n=num of landmarks

    # Calculate delta in robot's pose in the world frame
    x_delta, y_delta, theta_delta = robot_pose_delta(v,w,theta,t_delta)

    G_t = compute_G_t(v,w,theta,t_delta,N)  # Generate G_t

    # Update Mu and Sigma based on change in robot pose
    Mu, Sigma = prediction_step_update(Mu, Sigma, x_delta, y_delta, theta_delta, G_t

    if debug:
        print("(dx,dy,dtheta):{:.2f},{:.2f},{:.2f}".format(x_delta,y_delta,theta_del
        print("Mu:",Mu)
        # np.set_printoptions(suppress=True)
        print("G:",G_t)
        print("Sigma:",Sigma)

    """ Grab camera image, undistort and detect objects """
    image = undistort(camera.value, mtx, dist) # undistort camera image
    detections = model(image) # Use SSD model to detect objects

    """ Identify landmarks and estimate range(r)/bearing(phi) from robot """
    items = []
    for det in detections[0]:

        coco_id =det['label']
        if coco_id in landmark_item_list and valid_bbox(det["bbox"], width, height):
            """ If object detected is a landmark and its bounding box does not
            borders the edges of the camera video """

            """ =================== """
            """ EKF Correction Step """
            """ =================== """

            # Obtain landmark's left, center and right in horizontal pixel coordinat
            u_l, u_c, u_r = landmark_coordinates(det, width, height)
            j = landmark_item_list.index(coco_id)   # get landmark's index

            # Estimate landmark's range and bearing(r,phi) from robot
            r, phi = estimate_range_bearing(landmarks, j, \
                                            u_l, u_c, u_r, \
                                            c_u, focal_length, correct_factor, camer
            z_t = np.array([[r],[phi]])  # z_t - actual observation
```

```
            """ If landmark j encountered for 1st time, update (Mu_j_x, Mu_j_y) """
            if landmarks[j]['observed'] is False:
                # Update Mu's landmark coordinate
                landmarks[j]['observed'] = True
                Mu = update_landmark_Mu(Mu, j,  r, phi)

            # Get landmark j's expected observation
            delta, q, z_t_hat = get_observation(Mu, j)  # z_t_hat - expected observa

            # Compute H_t
            H_t = compute_H_t(delta, j, q, N)

            # Compute Kalmain Gain K_t
            K_t = compute_Kalman_Gain(delta, q, j, H_t, Sigma, N, Q_t)

            if debug:
                print("Mu before Correction:", Mu)
                print("Kalman Gain", K_t)

            # Mu_prev = Mu
            # Mu = Mu_prev + np.dot(K_t, (z_t-z_t_hat))

            Mu = Mu + np.dot(K_t, (z_t-z_t_hat))
            Sigma = np.dot((np.eye(N)-np.dot(K_t,H_t)), Sigma)

            if True:
                print("{} (left,center, right):{:.1f},{:.1f}, \
                {:.1f}".format(COCO_labels[coco_id-1], u_l, u_c, u_r))
                print("{} estimated phi:{:.1f} degree".format(COCO_labels[coco_id-1]
                print("{} estimated range:{:.1f}cm".format(COCO_labels[coco_id-1],r*
                print("Mu after Correction:", Mu)
                print("robot: {:.1f},{:.1f}".format(Mu[0,0], Mu[1,0]))
                print("{}: {:.1f},{:.1f}".format(COCO_labels[coco_id-1], Mu[3+2*j,0]
                print("delta:", delta)
                print("q: {:.2f}".format(q))
                print("observation:", z_t)
                print("expected observation:", z_t_hat)

            items.append(det)  # save item to display bounding box in image

    # Keep track of Mu of landmarks over time
    for j,landmark in enumerate(landmarks):
        x = np.asscalar(Mu[3+2*j,0])
        y = np.asscalar(Mu[3+2*j+1,0])
        landmark["Mu"].append([x,y])

    # Update robot and landmark on map
    if i%interval==0:
        start = time.perf_counter()
        update_map(Mu, landmarks,folder=diag_dir,ind=i,debug=debug)
        end = time.perf_counter()
        print ("Plot time: {:.1f}".format(end-start))

    cv2.line(image,(int(c_u),0),(int(c_u),300),GREEN,1)
    display_landmarks(image, items, width, height, debug) # put bounding boxes on de
    image_widget.value = bgr8_to_jpeg(image)  # update image widget with camera imag
```

In [ ]:
```
robot.stop()
```

Generate a Map of the Final State

Add actual measured pose of the robot.

In [ ]:

```
""" Update robot position on map """
plt.figure(figsize=(8,8))
plt.xlim([-100,250])
plt.ylim([-100,250])

""" Display robot as line + triangle (arrow) """
robot_x = Mu[0,0]*100
robot_y = Mu[1,0]*100
robot_theta = Mu[2,0]*180/math.pi - 90 # Adjust orientation to match matplotlib

if debug:
    print("(x,y):{:.1f}, {:.1f}".format(robot_x,robot_y))
    print("Orientation: {:.1f}".format(normalize_angle(Mu[2,0])*180/math.pi))

robot_x_actual = 150
robot_y_actual = 70
robot_theta_actual = (math.pi*3/8)*180/math.pi - 90 # Adjust orientation to matc

# robot = line + triangle
plt.plot(robot_x, robot_y, marker=(2, 0, robot_theta), c='k',markersize=15, line
plt.plot(robot_x, robot_y, marker=(3, 0, robot_theta), c='k',markersize=10, line

# robot = line + triangle
plt.plot(robot_x_actual, robot_y_actual, marker=(2, 0, robot_theta_actual), \
         c='r',markersize=15, linestyle='None')
plt.plot(robot_x_actual, robot_y_actual, marker=(3, 0, robot_theta_actual), \
         c='r',markersize=10, linestyle='None')

""" Display landmark as green cross, uncorrected landmark as lime cross """
for i, landmark in enumerate(landmarks):
    if landmark['observed'] is True:
        # Mark landmark's actual locations
        landmark_x_actual = landmark["actual_x"]*100
        landmark_y_actual = landmark["actual_y"]*100
        plt.plot(landmark_x_actual, landmark_y_actual, marker='*', markersize=8,
        # Mark landmark (post-Kalman correction)
        landmark_x = Mu[3+2*i]*100
        landmark_y = Mu[3+2*i+1]*100
        plt.plot(landmark_x, landmark_y,marker='x', markersize=12, color='blue')
        plt.text(landmark_x, landmark_y+10, landmark["obj_name"])

plt.savefig('circle_run03/map_final.png')
plt.show()
plt.close()
```

In [ ]:

```
```

Present | Slides | Themes | Help