Short Project 2

Carlos Bergillos, Xavier Martín

1. Pose estimated

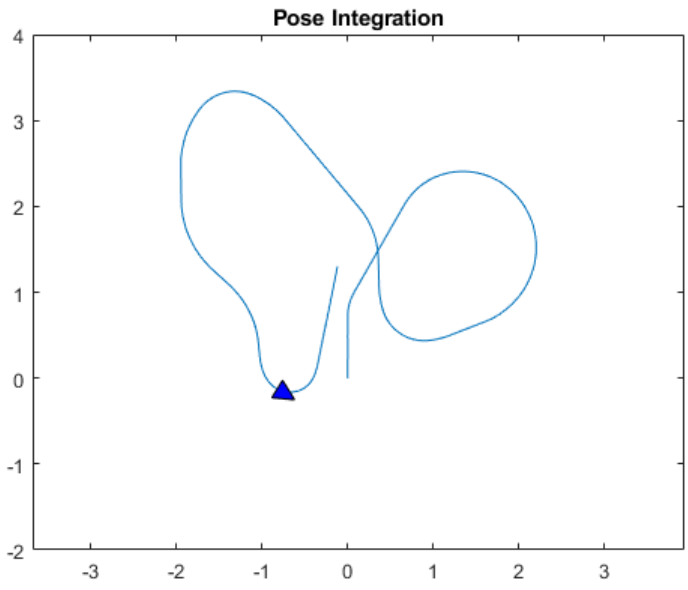
```
 function [ x, y, theta ] = pose_integration_ins( x_ini, y_ini,
theta_ini, L, R, width)
     S = width/(2 * 1000);

     delta_th = (R-L)/(2*S);
     delta_d = (R+L)/2;

     % error matrix
     V = [0.02^2       0;
          0          (0.5*pi/180)^2];

     V = randn(1, 2) * V;

     x = x_ini + (delta_d + V(1))*cos(theta_ini + delta_th + V(2));
     y = y_ini + (delta_d + V(1))*sin(theta_ini + delta_th + V(2));
     theta = mod((theta_ini + delta_th + V(2)), 2*pi);
 end
```

This function computes the pose integration of the robot.

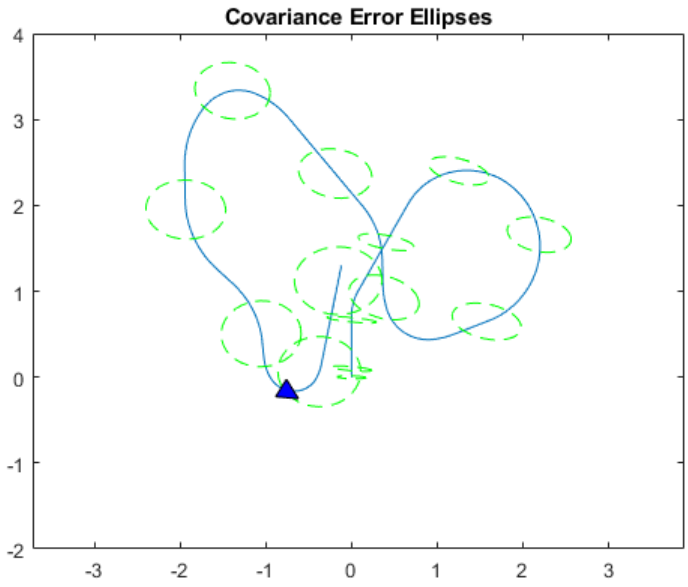
The function receives the distance travelled by the left and right wheels (`L`, `R`) as an input, as well as the last pose of the robot (`x_ini`, `y_ini`, `theta_ini`), and returns the pose estimation for the next instant (`x`, `y`, `theta`).

To obtain the different values for L and R, we are using columns 6 and 7 of the `data_enc` Workspace variable.

By calling this function many times sequentially (with all the available values for `L` and `R`), we can obtain an aproximate trajectory of the robot.

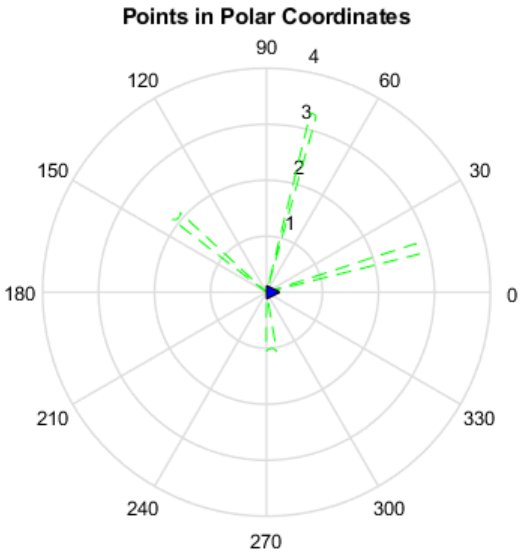


The function is also introducing an error to the pose integration (matrix variable V). To account for this error, we will also plot the the covariance error ellipses:



2. Polar to cartesian

```
function [x, y] = polar2cartesian_ins(data)
    [steps cols] = size(data);
    for j=2:361
        a = (j-1)/180*pi;
        x(j-1) = data(j)*cos(a);
        y(j-1) = data(j)*sin(a);
    end
end
```



Next, this function takes the points provided by the laser of the robot (as found in the `lds_dis` Workspace variable) and outputs the cartesian coordinates of the points (`x`, `y`) in robot frame.

To do this, for each of the 360 columns of the input, corresponding to the readings of 360 degrees, we obtain the cartesian coordinates by multiplying the given distance by the *sin* and *cos* of the laser angle, to obtain *x* and *y* in robot frame.

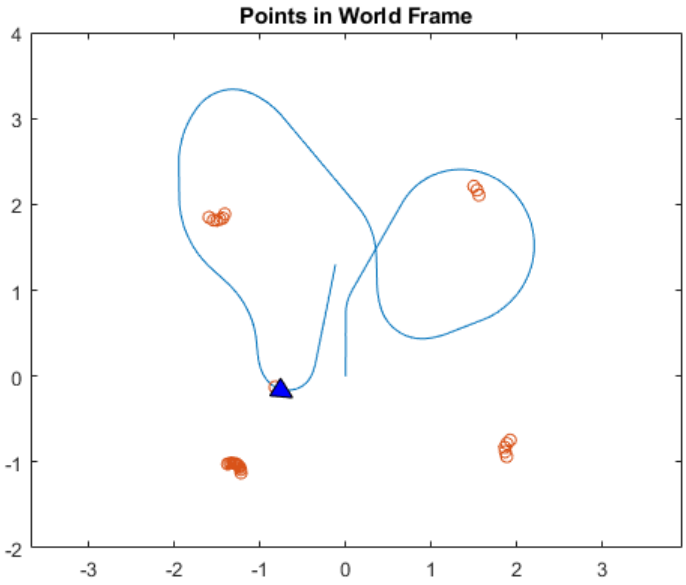
3. Robot to world

```
function [xs, ys] = robot2world_ins(data_x, data_y, x_ini, y_ini,
theta_ini)
    [steps cols] = size(data_x);
    for j=1:360
        RF = transl(x_ini,y_ini,0)*trotz(mod(theta_ini,2*pi));
        point = [data_x(j) data_y(j) 0];
        point = h2e(RF*e2h(point'));
        xs(j) = point(1);
        ys(j) = point(2);
    end
end
```

Now, we want to transform the points from robot frame to world frame. Thanks to the pose integration we did in section 1, we have the *x* and *y* position of the robot, as well as its angle *theta*.

The function takes the laser points in robot frame (*data_x*, *data_y*) and the robot's pose (*x_ini*, *y_ini*, *theta_ini*), and outputs the points in world frame (*xs*, *ys*).

We just need to apply a transformation to the points, with transl(*x_ini*, *y_ini*) and trotz(*theta_ini*).



4. Associated Land Mark

In this section we have calculated the points (in World Space) where the robot thinks the Land Marks are.

To do so, we have created a new function (nearest_to) which has the following inputs: X and Y coordinates of the real Land Marks (`LM_X`, `LM_y`), X and Y coordinates of the computed Land Marks by the robot in world space (*data_X*, *data_Y*), the number of points we will take into account to compute the center of a Land Mark (`n`) and the maximum distance in meters a point can be from a Land Mark (`thr`).

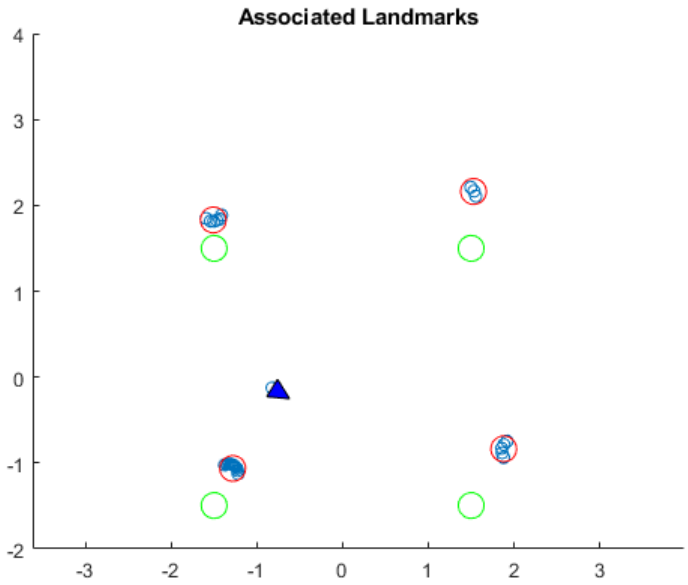
This function returns the X and Y coordinates in world space where the robot is detecting the Land Marks.

Due to the noise we have included (simulating that the floor is not completely even) and other errors, the pose integration does not provide the real location of he robot. As a consequence, the calculated Land Marks are moved a bit, and are not in the same place than the real Land Marks. If there are no points around a Land Mark, the robot will have any information about where that Land Mark is. This is because the value of our threshold of maximum distance (last input parameter).

We compute the average (center of gravity) of the nearest points to a given Land Mark to obtain the new position of that LandMark.

In this section we have used the given information about the real Land Marks (`LandMark` variable in the Workspace).

```
 function [x_mean, y_mean] = nearest_to(LM_x, LM_y, data_x, data_y,
n, thr)
     steps = length(data_x);
     p1 = [LM_x LM_y];
     for i=1:steps
         p2 = [data_x(i) data_y(i)];
         distances(i, 1) = sqrt(sum((p1 - p2).^2));
         distances(i, 2) = i;
     end
     distances = distances(distances(:, 1)<=thr, :);

     distances = sortrows(distances, 1);
     [rows cols] = size(distances);
     n = min(n, rows);
     ids = distances(1:n, 2);

     % we return the center of gravity of the closest points
     x_mean = mean(data_x(ids(:)));
     y_mean = mean(data_y(ids(:)));
 end
```



In the previous image, we can see the real Land Marks in green (which do not move) and the Land Marks that the robot has detected in red.

5. Similarity Transform

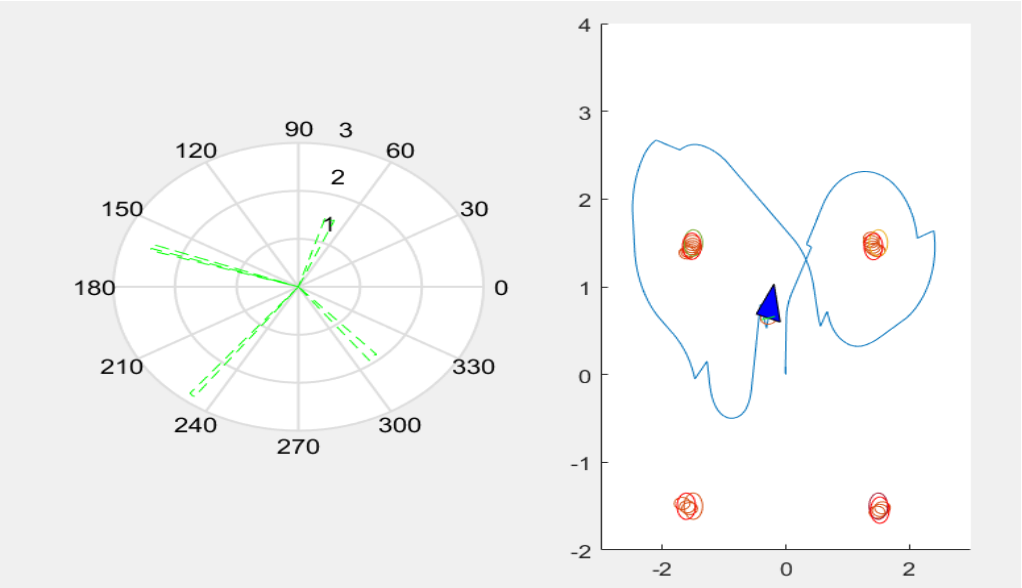
Next, we correct the robot's pose integration and trajectory with the information provided by the laser. To do so, every X steps (64 approx. in our case) we compute the error in the pose of the robot (*X*, *Y* and *theta)*. Then, we apply a transformation to the pose integration to correct this error.

With this, the robot is again in the given trajectory and the predicted Land Marks are also in the same place as the real Land Marks.

In this section we have used again the given information of the real Land Marks (`LandMark` variable in the workspace).

```
 function [tx, ty, theta] = similarity_transform(LandMarks,
CalculatedLandMarks)
     A = [];
     for i=1:size( LandMarks , 2)
         A = [A;[ LandMarks(1,i), LandMarks(2,i),1,0]];
         A = [A;[ LandMarks(2,i),-LandMarks(1,i),0,1]];
     end
     B = [];
     for i=1:size( CalculatedLandMarks , 2)
         B = [B; CalculatedLandMarks(1,i); CalculatedLandMarks(2,i)];
     end

     X = inv((A'*A))*A'*B;
     tx = X(3);
     ty= X(4);
     theta = atan2(X(2),X(1));
 end
```

The `similarity_transform` function computes this error. It takes as inputs the real Land Marks and the predicted Land Marks by the robot. This function computes two matrices: one with the real Land Marks (`A`) and the other with the predicted Land Marks by the robot (`B`). Then it applies a matrix operation: `inv((A'*A))*A'*B`. The result of this multiplication gives the *theta*, *X* and *Y* errors of the robot. Then if we apply these values to update the robot values we redirect the robot to a good trajectory.



As we can see in this figure, the trajectory (blue line) changes every X steps due to the corrections done thanks to the `similarity_transform` function. We can also see how the real Land Marks and the perceived Land Marks overlap, as this image was taken right after one of these corrections.