# ECE 9156 Report

Kehao Si (Student Number:251127808)

July 10, 2020

**Abstract**

This document is a report for project of course ECE 9156, in which a simulation of the algorithm of EKF-SLAM with known correspondences has been realized. The report is specified as 4 sections to demonstrate the design of the simulation.

## 1 Introduction of Simulation

In this project, there are 30 landmarks distributed at the edges of an equilateral triangle that the side length is set as 4 units and the center is the origin of the coordinates, which is shown in Figure 1.

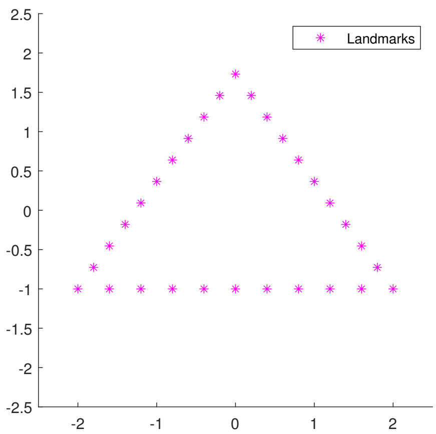


Figure 1: Locations of landmarks

The starting position of the robot has coordinates $(0, 1)$. Suppose the robot moves with the linear velocity $v$ and angular velocity $\omega$. Here, the velocities are both normally distributed. The mean of the linear velocity $\mu_v = 0.1\ unit/s$, and covariance $\sigma_v^2 = 0.01^2\ (unit/s)^2$. The mean of the angular velocity $\mu_\omega = 0.1 rad/s$, and covariance $\sigma_\omega^2 = 0.01^2\ (rad/s)^2$ . The robot measures distances and bearing angles to landmarks which is set with the sampling period $\Delta t = 0.5s$. The landmark measurement process is affected by additive Gaussian noise with zero mean and covariance $\sigma_r^2 = 0.1^2\ (unit/s)^2$, $\sigma_\phi^2 = 0.1^2\ (rad/s)^2$. The landmark correspondences are known.

## 2 System Analysis

With the configurations given in the section 1, we can write the control at time t:

$$u_t = \begin{pmatrix} v_t \\ \omega_t \end{pmatrix} = \begin{pmatrix} 0.1 \\ \frac{6\pi}{180} \end{pmatrix} \tag{1}$$

Also, the actual velocities can be written as

$$\begin{pmatrix} \hat{v}_t \\ \hat{\omega}_t \end{pmatrix} = \begin{pmatrix} v_t \\ \omega_t \end{pmatrix} + \begin{pmatrix} \sigma_v^2 \\ \sigma_\omega^2 \end{pmatrix}, \tag{2}$$

where $v_t = 0.1$, $\omega_t = 0.1$, $\sigma_v^2 = 0.01^2$, and $\sigma_\omega^2 = 0.01^2$. Thus, we can obtain the motion model which can represent the state of the robot

$$\begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \begin{pmatrix} -\frac{\hat{v}_t}{\hat{\omega}_t}\sin\theta_{t-1} + \frac{\hat{v}_t}{\hat{v}_t}\sin(\theta_{t-1} + \hat{\omega}_t\Delta t) \\ \frac{\hat{v}_t}{\hat{\omega}_t}\cos\theta_{t-1} - \frac{\hat{v}_t}{\hat{v}_t}\cos(\theta_{t-1} + \hat{\omega}_t\Delta t) \\ \hat{\omega}_t\Delta t \end{pmatrix} \tag{3}$$

with the robot initial state

$$\begin{pmatrix} x_0 \\ y_0 \\ \theta_0 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}. \tag{4}$$

The combined state vector which is presented by matrix $y$ in the program consists of the robot state and the map:

$$\mathbf{y_t} = \begin{pmatrix} \mathbf{x_t} \\ m \end{pmatrix}. \tag{5}$$

where the demension is 63 that the signatures of landmark are stored in another vector which is set as the vector *signature* in the program.

For convenience of expression and computation, $\theta$, which denotes the angle of the robot facing direction with respect to the positive direction of X-axis, is uniformed in the interval $[-\pi, \pi]$.

Now, the predtion section of the EKF-SLAM algorithm can be implemented. Calculating the Jacobian matrix of the motion model which is denoted as *JacobianMotion* in the program, one obtains

$$G_t = \begin{pmatrix} 1 & 0 & -\frac{v_t}{\omega_t}\cos\theta_{t-1} + \frac{v_t}{v_t}\cos(\theta_{t-1} + \omega_t\Delta t) \\ 0 & 1 & -\frac{v_t}{\omega_t}\sin\theta_{t-1} + \frac{v_t}{v_t}\sin(\theta_{t-1} + \omega_t\Delta t) \\ 0 & 0 & 1 \end{pmatrix}. \tag{6}$$

The covariance matrix of the motion model which is denoted as $Q$ can be acquired as

$$Q = \begin{pmatrix} 0.01^2 & 0 \\ 0 & 0.01^2 \end{pmatrix}. \tag{7}$$

To map the covariance $Q$ into the state space, the Jacobian matirx which is denoted as $Vt$ in the program can be obtained:

$$V_t = \begin{pmatrix} \frac{-\sin\theta_t + \sin\theta_t + \omega_t\Delta t}{\omega_t} & \frac{v_t(\sin\theta_t - \sin\theta_t + \omega_t\Delta t)}{{\omega_t}^2} + \frac{v_t\cos\omega_t\Delta t\Delta t}{\omega_t} \\ \frac{\cos\theta_t - \cos\theta_t + \omega_t\Delta t}{\omega_t} & -\frac{v_t(\cos\theta_t - \cos\theta_t + \omega_t\Delta t)}{{\omega_t}^2} + \frac{v_t\sin\omega_t\Delta t\Delta t}{\omega_t} \\ 0 & \Delta t \end{pmatrix}. \tag{8}$$

In the last step of pridection, the predicted covariance at time $t$ can be computed:

$$\bar{\Sigma}_t = G_t\Sigma_{t-1}G_t^{\mathrm{T}} + V_tQV_t^{\mathrm{T}} \tag{9}$$

As the pridection section is accomplished, then we can step into the measurement update section. In this section, the robot repeats computing loop for each landmark which is sensored by the robot, and updates its state after everytime the loop is finished. Here, the measurement model is

$$\begin{pmatrix} r_t^i \\ \phi_t^i \\ s_t^i \end{pmatrix} = \begin{pmatrix} \sqrt{m_jx - {x_t}^2 + m_jy - {y_t}^2} \\ \operatorname{atan2}(m_jy - y_t, m_jx - x_t) - \theta_t \\ s_j \end{pmatrix} + \mathcal{N}(0, M), \tag{10}$$

where

$$M = \begin{pmatrix} 0.01 & 0 & 0 \\ 0 & 0.01 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{11}$$

If the $j-th$ landmark is detected for the first time, the prior location estimate could be imprecise. In this case, the esitmate is replaced with the projected location obtianed from the range and bearing measurements using the formula

$$\begin{pmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \\ \bar{\mu}_{j,s} \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \\ s_t^i \end{pmatrix} + \begin{pmatrix} r_t^i \cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i \sin(\phi_t^i + \bar{\mu}_{t,\theta}) \\ 0 \end{pmatrix} \tag{12}$$

The Jacobian of the noise-free $i$-$th$ measurement model with respect to the state vector and the map of the landmark $j$ is calculated at the predicted mean is

$$H_t^i = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\sigma_x & -\sqrt{q}\sigma_y & 0 & \sqrt{q}\sigma_x & \sqrt{q}\sigma_y & 0 \\ \sigma_y & -\sigma_x & -q & -\sigma_y & \sigma_x & 0 \\ 0 & 0 & 0 & 0 & 0 & q \end{pmatrix}, \tag{13}$$

where $\sigma_x = \bar{\mu}_{j,x} - \bar{\mu}_{t,x}, \sigma_y = \bar{\mu}_{j,y} - \bar{\mu}_{t,y}, q = {\sigma_x}^2 + {\sigma_y}^2$.

Therefore, we can obtain the Kalman Gain $K_t^i$ and upadate the predicted mean $\bar{\mu}_t$ and the predicted covariance $\bar{\Sigma}_t$ as

$$K_t^i = \bar{\Sigma}_t {H_t^i}^{\mathrm{T}} (H_t^i \bar{\Sigma}_t {H_t^i}^{\mathrm{T}} + M)^{-1}; \tag{14}$$

$$\bar{\mu}_t = \bar{\mu}_t + K_t^i (z_t^i - \hat{z}_t^i); \tag{15}$$

$$\bar{\Sigma}_t = (I - K_t^i H_t^i) \bar{\Sigma}_t. \tag{16}$$

If the $j-th$ landmark is not the last one which is sensored by the robot at time $t$, the robot will repeat the calculating process and upadate its state to reduce the uncertainty over agian, untill the last sensored landmark at time $t$ is processed.

# 3 Result Demonstration

In this simulation, the robot sensor is assumed to dectect the environment around the robot as a circle with radius of $1.5 units$. The uncertainties of the estimated locations of landmarks are performed as ellipses in the figure. For the landmark which is sensored for the first time, the covariance ellipse is blue. If the landmark has been sensored before and is sensored currently, the covariance ellipse is shown as red. If the landmark has been sensored before but is NOT detected currently, the covariance ellipse is in black. Additionally, the estimated location of landmark is represented by sign "+" in the figure.

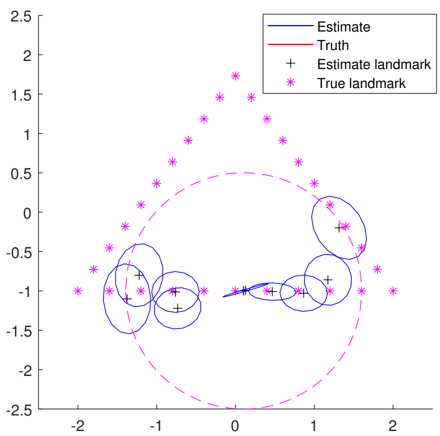


Figure 2: The robot state at Time $t_0$

The simulation of time $t_0 = 0s$ is shown as Figure 2. At time $t_0$ that all the landmarks in the sensor range are not detected, the uncertainty of the system is quite high and the robot localization of itself and the landmarks are both imprecise.

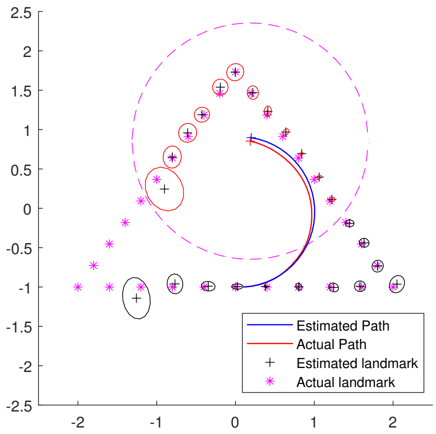


Figure 3: The robot state at Time $t_1$

Figure 3 illustrates the system state at time $t_1 = 30s$. The robot moves while observing landmarks. The estimated robot's path is shown as the red line. The uncertainty of robot's location and locations of landmarks increase as robot moves.

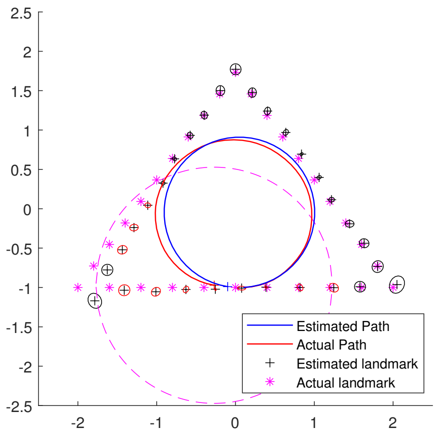


Figure 4: The robot state at Time $t_2$

At time $t_2 = 60s$, all the landmarks has been sensored which means the robot senses the first landmarks again. Consequently, the uncertainties of all landmarks decrease

# 4 Copy of Code

## 4.1 Main Program

The main program starts with the configuration initialization such as the motion model noise, the measurement noise, the control input and the location of landmarks.

```
%% I. Initialization
% motion model noise
q = [0.01;0.01];
Q = diag(q.^2);
% measurement noise
m = [.1; 0.1];
M = diag(m.^2);
% R: robot initial pose
% u: control
% ******************************************%
R = [0;-1;0];
u = [0.1;6/180*pi];
% ******************************************%
% set landmarks
Marks = landmarks();
% sensor radius
% ******************************************%
sensor_r = 1.5;
% ******************************************%
% i-th landmark is sensored before, LANDMARK(i) = 1;
% j-th landmark is not sensored, LANDMARK(j) = 0;
LANDMARK = zeros(1,size(Marks,2));
% y_news -- landmarks first seen
% y_olds -- landmarks has been sensored before
y_olds = zeros(3,size(Marks,2));
y_news = zeros(3,size(Marks,2));
%   State and covariance intialization
y = zeros(numel(R)+numel(Marks), 1); %** State && Map **%
P = zeros(numel(y),numel(y));
signature = zeros(1,size(Marks,2));
r = [1 2 3];
y(r) = R;
sigma = 0;
% Map starts at 4&5
s = [4 5];
% 60 s/circle
loop =120;
% poses_ -- store the actual state
poses_ = zeros(3,loop);
% poses -- store the estimated state
poses = zeros(3,loop);
```

As the initialization is accomplished, the EKF-SLAM with known correspondences algorithm reflecting the simulation into the figure is implemented as fellow:

```
for t = 1:loop
% control noise
n = q.*randn(2,1);
% return the robot actual position
R = move(R, u, n);
i_olds=1;
i_news=1;
for i = 1:size(Marks,2)
```

```
    %mesurement error
    v = m.*randn(2,1);
    yi= project(R, Marks(:,i)) + v;
    if yi(1) < sensor_r && LANDMARK(i) == 1
        y_olds(:,i_olds) = [yi(1);yi(2);i];
        i_olds = i_olds + 1;
    elseif  yi(1) < sensor_r &&  LANDMARK(i) == 0
            y_news(:,i_news) = [yi(1);yi(2);i];
            i_news = i_news + 1;
            LANDMARK(i) = 1;
    end
end
for i = i_olds:size(Marks,2)
    y_olds(:,i) = [100;0;0];
end
for i = i_news:size(Marks,2)
    y_news(:,i) = [101;0;0];
end
% EKF
%  prediction
[y(r), JacobianMotion, Vt] = move(y(r), u, [0 0]);
P_rr = sigma;
P(r,:) = JacobianMotion*P(r,:);
P(:,r) = P(r,:)';
sigma = JacobianMotion*P_rr*JacobianMotion' + Vt*Q*Vt';
%  update
end_old = find(y_olds(1,:)==100,1);
if isempty(end_old)
    end_old=size(y_olds,2)+1;
end
for j = 1:(end_old-1)
    % expectation
    if isempty(j)
        break
    end
    id = find(signature==y_olds(3,j),1);
    v = [id*2+2 id*2+3];
    [e, E_r, E_l] = project(y(r), y(v));
    H = [E_r E_l];
    rl   = [r v];
    E    = H * P(rl,rl) * H';
    % measurement
    yi_1 = y_olds(:,j);
    yi1 = yi_1(1:2,1);
    % innovation
    z = yi1 - e;
    if z(2) > pi
        z(2) = z(2) - 2*pi;
    end
    if z(2) < -pi
        z(2) = z(2) + 2*pi;
    end
    S = E+M;
    % Kalman gain
    K = P(:, rl) * H' * S^-1;
    % update
    y = y + K * z;
    P = P - K * S * K';
end
% for the landmarks which are never seen before
end_new = find(y_news(1,:)==101,1);
if isempty(end_new)
    end_new=size(y_news,2)+1;
```

```
end
for m1 = 1:(end_new-1)
    if isempty(m1)
        break
    end
    id = find(signature==0,1);
    signature(id) = y_news(3,m1);

    % measurement
    yi_2 = y_news(:,m1);
    yi2 = yi_2(1:2,1);
    [y(s), L_r, L_y] = backProject(y(r ), yi2);
    P(s,:) = L_r * P(r,:);
    P(:,s) = P(s,:)';
    P(s,s) = L_r * sigma * L_r' + L_y * M * L_y';
    s = s + [2 2];
end
% obtain states info
% estimated
poses(1,t) = y(1);
poses(2,t) = y(2);
poses(3,t) = y(3);
% actual
poses_(1,t) = R(1);
poses_(2,t) = R(2);
poses_(3,t) = R(3);
% 5. PLOT
% Actual positon and the range of sensor
set(RG, 'xdata', R(1), 'ydata', R(2));
circle_x = linspace((R(1)-0.9999*sensor_r),(R(1)+0.9999*sensor_r));
circle_y1 = sqrt(sensor_r^2 - (circle_x - R(1)).^2) + R(2);
circle_y2 = R(2) - sqrt(sensor_r^2 - (circle_x - R(1)).^2);
set(sensor1,'xdata',circle_x,'ydata',circle_y1);
set(sensor2,'xdata',circle_x,'ydata',circle_y2);
% Estimated position
set(rG, 'xdata', y(r(1)), 'ydata', y(r(2)));
Circle_x = linspace((y(r(1))-0.9999*sensor_r),(y(r(1))+0.9999*sensor_r));
Circle_y1 = sqrt(sensor_r^2 - (Circle_x - y(r(1))).^2) + y(r(2));
Circle_y2 = y(r(2)) - sqrt(sensor_r^2 - (Circle_x - y(r(1))).^2);
% Robot actual&estimated path
set(estimate_pose,'xdata',poses(1,1:t),'ydata',poses(2,1:t));
set(true_pose,'xdata',poses_(1,1:t),'ydata',poses_(2,1:t));

legend([estimate_pose true_pose lG WG],{'Estimated Path','Actual Path' ...
    'Estimated landmark' 'Actual landmark'})
if s(1)==4
    continue
end
% The estimated locations of landmark
w = 2:((s(1)-2)/2);
w = 2*w;
landmark_estimated_x = y(w);
landmark_estimated_y = y(w+1);
set(lG, 'xdata', landmark_estimated_x, 'ydata', landmark_estimated_y);

%%%%% 1- the landmark is never sensored before (BLUE)
for g1 = 1:(end_new-1)
    if isempty(g1)
            break
    end
    o1 = y_news(3,g1);
    h1 = find(signature==o1,1);
    temp1 = [2*h1+2;2*h1+3];
```

```
    le = y(temp1);
    LE = P(temp1,temp1);
    [X,Y] = cov2elli(le,LE,3,16);
    set(eG1(o1),'xdata',X,'ydata',Y,'color','b');
end
%%%% 2- the landmark has been sensored and is sensored again (RED)
for g2 = 1:(end_old-1)
    if isempty(g2)
            break
    end
    o2 = y_olds(3,g2);
    h2 = find(signature==o2,1);
    temp2 = [2*h2+2;2*h2+3];
    le = y(temp2);
    LE = P(temp2,temp2);
    [X,Y] = cov2elli(le,LE,3,16);
    set(eG1(o2),'xdata',X,'ydata',Y,'color','r');
end
%%%% 3- the landmark has been sensored and is NOT sensored now (BLACK)
v = find(signature==0,1);
if isempty(v)
    v = size(signature,2)+1;
end
for g3 = 1:v-1
    if isempty(g3)
            break
    end
    a = find(y_olds(3,:)==signature(g3),1);
    b = find(y_news(3,:)==signature(g3),1);
    if (isempty (a)) && (isempty(b))
        temp3 =  [2*g3+2;2*g3+3];
            le = y(temp3);
    LE = P(temp3,temp3);
    [X,Y] = cov2elli(le,LE,3,16);
    set(eG1(signature(g3)),'xdata',X,'ydata',Y,'color','k');
    end
end
drawnow;
pause(0.5);
end
```

## 4.2 Functions

Function `landmarks()`:

```
function f = landmarks()
    step = 0.4;
    n1 = 4/step;
    n2 = n1*2;
    n3 = n1*3;
    f = zeros(2,n3);
    for i = 1:n1
        f(1,i) = -2+step*(i);
        f(2,i) = -1;
    end
    for i = 1:n1
        f(1,i+n1) = -2+0.5*step*(i-1);
        f(2,i+n1) = (sqrt(3)+1)/2 * (-2+0.5*step*(i-1)) + sqrt(3);
    end
    for i = 1:n1
        f(1,i+n2) = 0.5*step*(i-1);
```

```
        f(2,i+n2) = (-(sqrt(3)+1)/2 * (0.5*step*(i-1)) + sqrt(3));
    end
end
```

Function `toFrame2D()`:

```
function [p_r, PR_r, PR_p] = toFrame2D(r , p)
    t = r(1:2);
    a = r(3);
    R = [cos(a) -sin(a) ; sin(a) cos(a)];
    p_r = R' * (p - t);
    if nargout > 1
        px = p(1);
        py = p(2);
        x = t(1);
        y = t(2);
        %Jacobian
        PR_r = [...
               [ -cos(a), -sin(a),   cos(a)*(py - y) - sin(a)*(px - x)]
               [  sin(a), -cos(a), - cos(a)*(px - x) - sin(a)*(py - y)]];
        PR_p = R';
    end
end
```

Function `fromFrame2D()`:

```
function [p, Jaco_pos, P_pr] = fromFrame2D(r, p_r)
    t = r(1:2);
    a = r(3);
    R = [cos(a) -sin(a) ; sin(a) cos(a)];
    p = R*p_r + t;
    if nargout > 1
        px = p_r(1);
        py = p_r(2);
        Jaco_pos = [...
                   [ 1, 0, - py*cos(a) - px*sin(a)]
                   [ 0, 1,   px*cos(a) - py*sin(a)]];
        P_pr = R;
    end
end
```

Function `scan()`:

```
function [y, Y_x] = scan(x)
    px = x(1);
    py = x(2);
    q = sqrt(px^2 + py^2);
    a = atan2(py, px);
    y = [q;a];
    if nargout > 1
        Y_x =[...
             [      px/(px^2 + py^2)^(1/2), py/(px^2 + py^2)^(1/2)]
             [ -py/(px^2*(py^2/px^2 + 1)), 1/(px*(py^2/px^2 + 1))]];
    end
end
```

Function `invScan()`:

```
function [p, P_y] = invScan(y)
    d = y(1);
    a = y(2);
    px = d * cos(a);
    py = d * sin(a);
    p = [px;py];
    if nargout > 1
       P_y = [cos(a) -d*sin(a)
              sin(a)  d*cos(a)];
end
```

Function `move()`

```
function [ro, JacobianMotion, RO_n] = move(r, u, n)
    a = r(3);
    dx = u(1) + n(1);
    da = u(2) + n(2);
    ao = a + da;
    dp = [dx;0];
    %uniform the angle [-pi,pi]
    if ao > pi
        ao = ao - 2*pi;
    end
    if ao < -pi
        ao = ao + 2*pi;
    end
    if nargout == 1
        to = fromFrame2D(r, dp);
    else
        [to, Jaco_pos, TO_dp] = fromFrame2D(r, dp);
        AO_a  = 1;
        AO_da = 1;
        JacobianMotion = [Jaco_pos ; 0 0 AO_a];
        RO_n = [TO_dp(:,1) zeros(2,1) ; 0 AO_da];
    end
    ro = [to;ao];
end
```

Function `cov2elli()`

```
function [X,Y] = cov2elli(x,P,ns,NP)
    % Ellipsoidal representation of multivariate Gaussian variables (2D). ...
       Different
    % sigma-value ellipses can be defined for the same covariances ...
       matrix. The most useful
    % ones are 2-sigma and 3-sigma
    %Ellipse points from mean and covariances matrix.
    %   [X,Y] = COV2ELLI(X0,P,NS,NP) returns X and Y coordinates of the NP
    %   points of the the NS-sigma bound ellipse of the Gaussian defined by
    %   mean X0 and covariances matrix P.
    %
    %   The ellipse can be plotted in a 2D graphic by just creating a line
    %   with line(X,Y).
    persistent circle
    if isempty(circle)
        alpha = 2*pi/NP*(0:NP);
        circle = [cos(alpha);sin(alpha)];
    end
```

```
    [R,D]=svd(P);
    d = sqrt(D);
   %circle -> aligned ellipse -> rotated ellipse -> ns-ellipse
    ellip = ns*R*d*circle;
   X = x(1)+ellip(1,:);
   Y = x(2)+ellip(2,:);
end
```

Function `project()`

```
function [y, Y_r, Y_p] = project(r, p)
    if nargout == 1
        p_r = toFrame2D(r, p);
        y   = scan(p_r);
    else
        [p_r, PR_r, PR_p] = toFrame2D(r, p);
        [y, Y_pr]    = scan(p_r);
        Y_r = Y_pr * PR_r;
        Y_p = Y_pr * PR_p;

    end
end
```

Function `backProject()`

```
function [p, P_r, P_y] = backProject(r, y)
    if nargout == 1
        p_r = invScan(y);
        p   = fromFrame2D(r, p_r);
    else
        [p_r, PR_y]     = invScan(y);
        [p, P_r, P_pr] = fromFrame2D(r, p_r);
        P_y = P_pr * PR_y;
    end
end
```