Model-Based Visual Tracking

Solution of Matlab exercises

Dr. Giorgio Panin

TUM Informatik – Robotics and Embedded Systems
(Prof. Alois Knoll)

Pose representations

Exercise 2.1.

Write a Matlab function that takes 3 Euler angles (with axes x-y-z) in degrees, and returns the rotation matrix R

- Input: Euler angles (α,β,γ)
- Output: Rotation matrix R

2.1. - Euler Angles representation

```
% 2.1 Euler Angles representation

function [R] = Euler2Mat(a,b,c)

% Angles must be in radians!
a = a*pi/180;
b = b*pi/180;
c = c*pi/180;

% Elementary rotation matrix around x
Rx = [1 0 0; 0 cos(a) -sin(a); 0 sin(a) cos(a)];

% Elementary rotation matrix around y
Ry = [cos(b) 0 -sin(b); 0 1 0; sin(b) 0 cos(b)];

% Elementary rotation matrix around z
Rz = [cos(c) -sin(c) 0; sin(c) cos(c) 0; 0 0 1];

% Rotation matrix
R = Rx*Ry*Rz;
```

2.1. - Euler Angles representation

Example

```
a = 10
b = 20
c = 30

R =

    0.8138   -0.4698   -0.3420
    0.4410    0.8826   -0.1632
    0.3785   -0.0180    0.9254


R*R' =

    1.0000   -0.0000    0.0000
   -0.0000    1.0000         0
    0.0000         0    1.0000
```

2.3. – Body to Screen Transformation

Exercises 2.3, 2.4, 2.5.

Write a Matlab script that computes the 3D/2D transformation from a point in body frame coordinates to a point on the screen, by using the (extrinsic+intrinsic) transformation models

- Input: a point in 3D space (body coordinates)
- Input: the 6 roto-translation (pose) parameters
- Input: intrinsic camera parameters (pinhole model)
- Output: the screen point coordinates of the projection

2.3. – Body to Screen Transformation

```
% World to screen transformation (3D/2D)

clear;
close all;

% Rotation angles (radians)
a = 10*pi/180;
b = 20*pi/180;
c = 30*pi/180;

% Translation parameters (mm)
tx = 50;
ty = 60;
tz = 500;

% Input: a point in Body frame coordinates
pb = [-20;30;50];
```

2.3. – Body to Screen Transformation

```
% 2.3. Extrinsic transformation

% Call the Euler Angles function (exercise 2.1.)
R = Euler2Mat(a,b,c);

% Put the roto-translation into the
% homogeneous transformation matrix (4x4)
T = [R, [tx;ty;tz]; 0,0,0,1];

% Homogeneous coordinates of pb
pbH = [pb;1];

% Extrinsic transformation
pcH = T*pbH;

% Result: point in camera frame coordinates
pc = pcH(1:3);
```

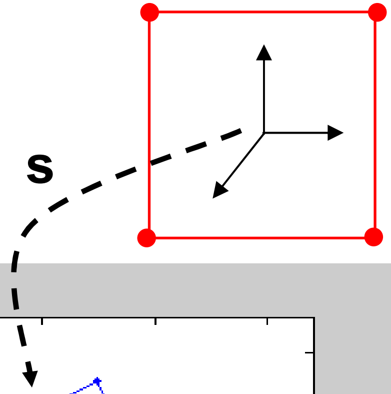
2.3. – Body to Screen Transformation

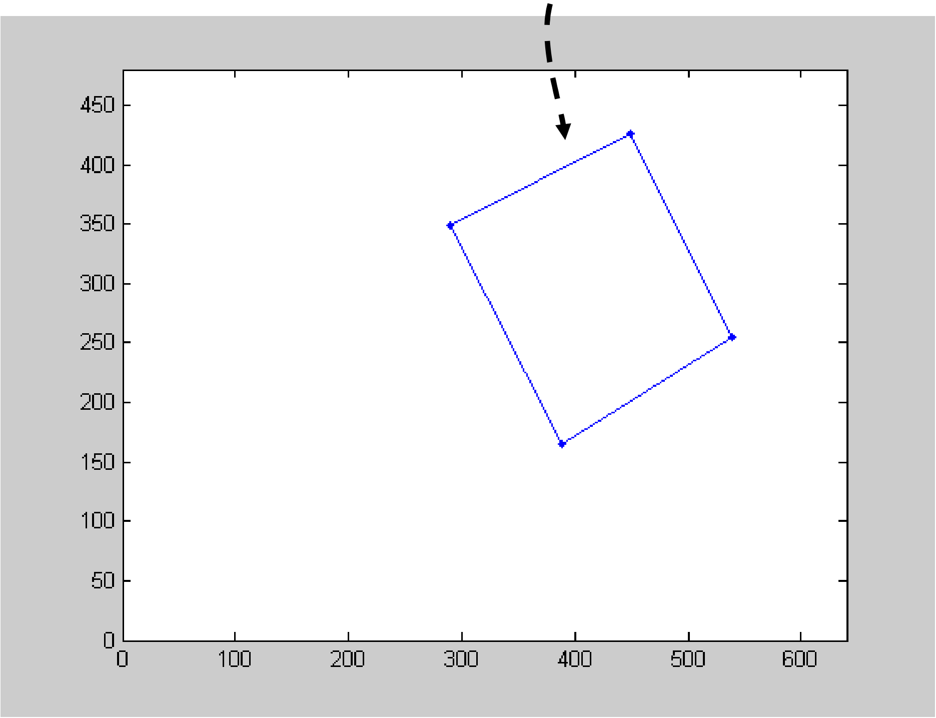
```
% 2.4. Intrinsic transformation

% Intrinsic parameters

% Focal length
f = 1000;
% Horizontal and vertical resolutions
rx = 640;
ry = 480;

% Put them together into the intrinsic transformation matrix
K = [f 0 rx/2; 0 f ry/2; 0 0 1];

% Do the 3D/2D projection, from camera to screen
qH = K*pc;

% Normalize the homogeneous coordinates
q = [qH(1)/qH(3); qH(2)/qH(3)]

% Result: screen point q
```

```
% 2.5. Global transformation

% The two transformations can be combined
% into the Projection matrix (3x4)
P = K*[R, [tx;ty;tz]];

% → The same process can be done in a single step
% (in homogeneous coordinates)
qH = P*pbH;

q = [qH(1)/qH(3); qH(2)/qH(3)]
```

2.3. – Body to Screen Transformation

```
% Body to screen projection: example

% The Body model consists of 4 points (a square) in space
pb1 = [-50; -50; 0];
pb2 = [50; -50; 0];
pb3 = [50; 50; 0];
pb4 = [-50; 50; 0];

f = 1000;
rx = 640;
ry = 480;

% Pose parameters (6 dof)
s = [10 20 30 50 30 500];

[q1] = GlobalT(s,pb1,f,rx,ry);
[q2] = GlobalT(s,pb2,f,rx,ry);
[q3] = GlobalT(s,pb3,f,rx,ry);
[q4] = GlobalT(s,pb4,f,rx,ry);
```





Pose estimation

Linear LSE

A) Solve the linear regression problem : $f_i = a_1 x_i + a_2$

for the set of points $(x_1,y_1),\ldots,(x_N,y_N)$

x	1	2	3	4	5	6	7	8	9	10
y	2.6892	3.3476	3.8128	5.1878	5.6323	6.5843	6.7507	7.2644	8.5423	9.0124

(the "true" line has coefficients $a_1 = 0.7$, $a_2 = 2$)

B) Plot the result, by indicating with a cross 'x' the points from the table, and draw the line resulting from LSE, together with the "true" one.

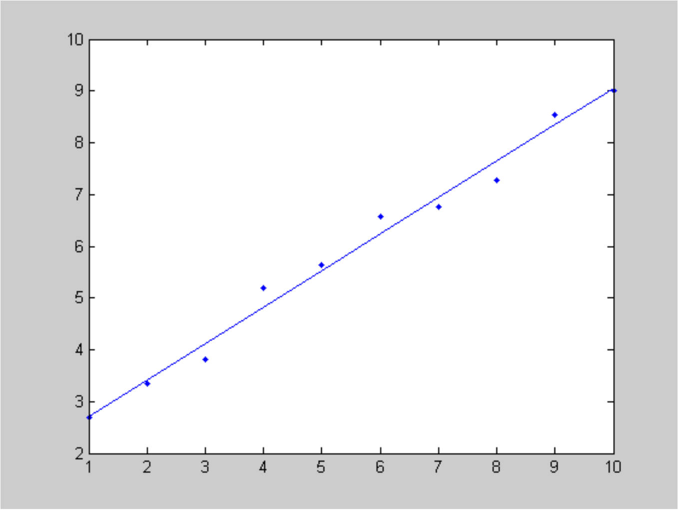
C) Try to modify significantly one of the measurements y, and run again the estimation to see the sensitivity to outliers.

3.1. Linear Regression (LSE)

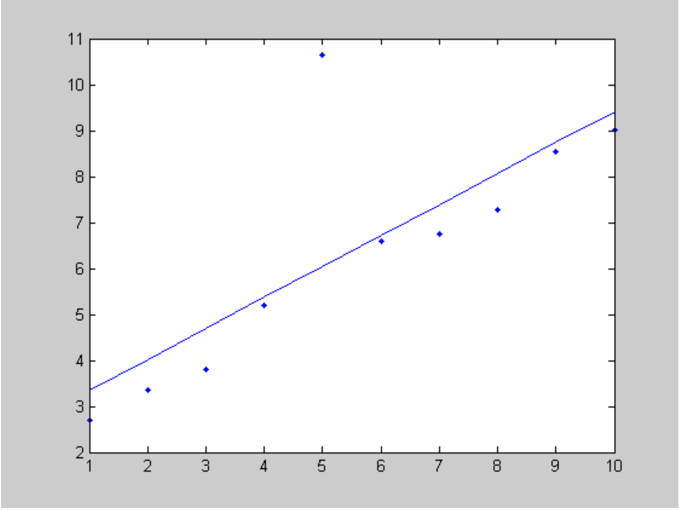
```
% 3.1 Linear Regression (LSE)

clear;
close all;

% Data (column vectors)
x = [1 2 3 4 5 6 7 8 9 10]';
y = [2.6892 3.3476 3.8128 5.1878 5.6323 6.5843 6.7507 7.2644
8.5423 9.0124]';

% Model: y = a1*x+a2
% State: s = [a1 a2];

% Construct the coefficient matrix (A)
% Ai = [xi 1]

l = length(x);
A = [x ones(l,1)];

% Solve for the minimum LSE error
s = inv(A'*A)*A'*y;
```

3.1. Linear Regression (LSE)

```
% Plot results
% The estimated line is: y = s(1)*x+s(2)
yest = s(1)*x+s(2);

figure;
plot(x,yest);
hold;
plot(x,y,'b.');
```




```
% Modify one of the values, and do the optimization again
ymod = y;
ymod(5) = ymod(5)+5;

smod = inv(A'*A)*A'*ymod;
yest2 = smod(1)*x+smod(2);

figure;
plot(x,yest2);
hold;
plot(x,ymod,'b.');
```



3.2. Weighted Linear LSE

Weighted linear LSE

For the same set of points (x_i,y_i), now introduce weights

x	1	2	3	4	5	6	7	8	9	10
w	1.0	0.5	0.4	0.3	0.1	0.8	0.6	0.8	0.9	1.0

and solve the weighted LSE problem by using the W matrix: $W = \mathrm{diag}(w_1,\ldots,w_{10})$.

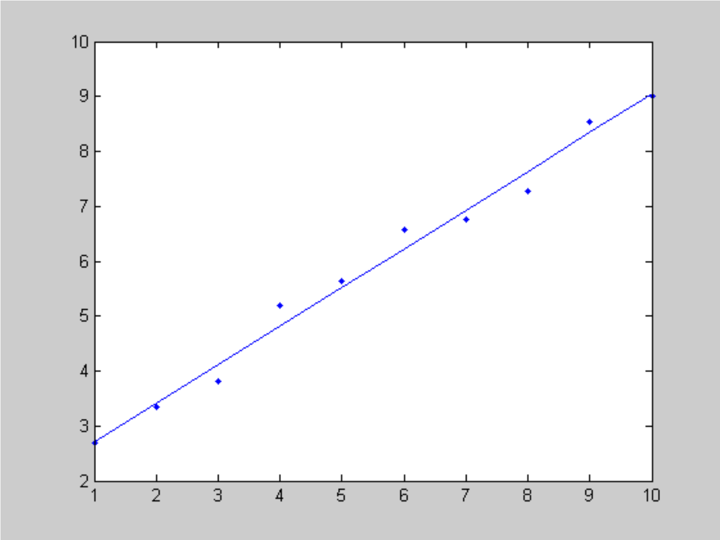
Try again to see the sensitivity to each measurement, which now is different for each point: for example, it should be much more sensitive to y_1 than y_5.

3.2. Weighted Linear LSE

```
% 3.2 Weighted Linear Least-Squares (WLSE)

clear;
close all;

% Data (column vectors)
x = [1 2 3 4 5 6 7 8 9 10]';
y = [2.6892 3.3476 3.8128 5.1878 5.6323 6.5843 6.7507 7.2644
8.5423 9.0124]';

% Weights
w = [1 0.5 0.4 0.3 0.1 0.8 0.6 0.8 0.9 1]';
W = diag(w);

% Model: y = a1*x+a2
% State: s = [a1 a2];
```

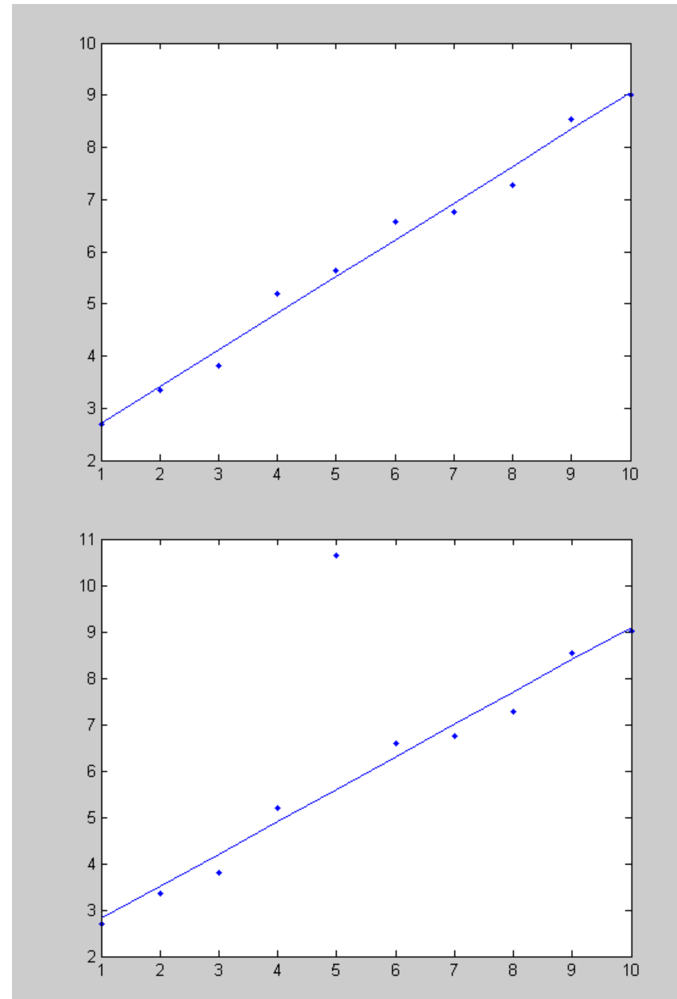
3.2. Weighted Linear LSE

```
% Construct the coefficient matrix (A)
% Ai = [xi 1]

l = length(x);
A = [x ones(l,1)];

% Solve for the minimum weighted error
s = inv(A'*W*A)*A'*W*y;

% Plot results
% The estimated line is: y = s(1)*x+s(2)
yest = s(1)*x+s(2);

figure;
plot(x,yest);
hold;
plot(x,y,'b.');
```



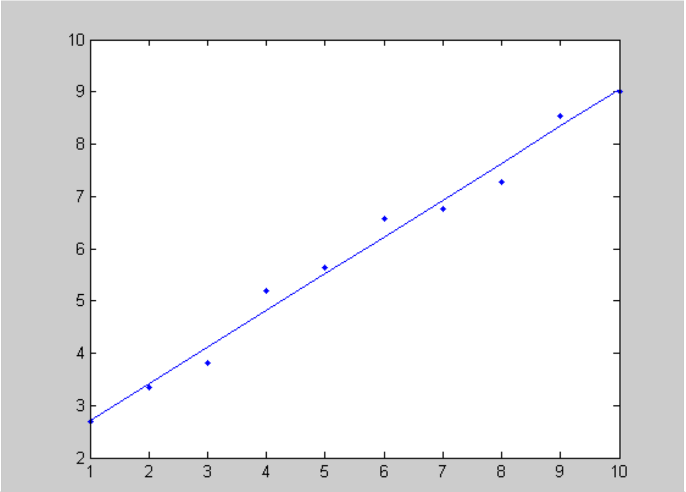
3.2. Weighted Linear LSE

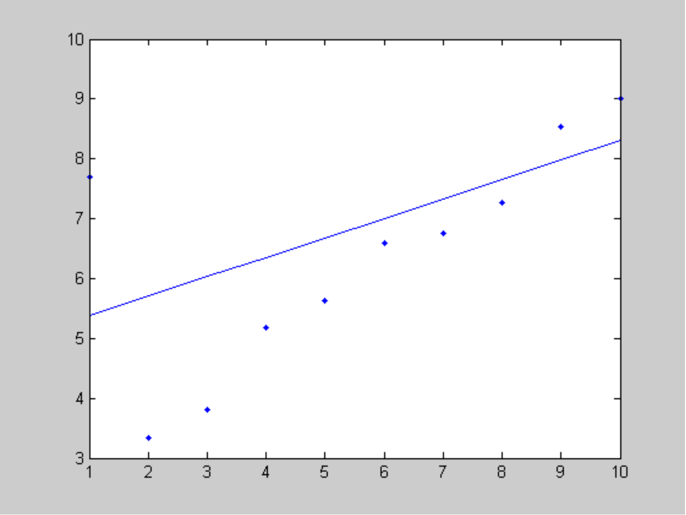
```
% Modify y5, and do the optimization again
% LSE should be less sensitive to an outlier in y5
ymod = y;
ymod(5) = ymod(5)+5;

smod = inv(A'*W*A)*A'*W*ymod;
yest2 = smod(1)*x+smod(2);

figure;
plot(x,yest2);
hold;
plot(x,ymod,'b.');
```



3.2. Weighted Linear LSE

```
% Modify y1, and do the optimization again
% LSE should be more sensitive to an outlier in y1
ymod = y;
ymod(1) = ymod(1)+5;

smod = inv(A'*W*A)*A'*W*ymod;
yest2 = smod(1)*x+smod(2);

figure;
plot(x,yest2);
hold;
plot(x,ymod,'b.');
```





Nonlinear LSE

Given the following data set

x	1	2	3	4	5	6	7	8	9	10
y	1.7321	-0.0509	-1.4885	-1.8603	-0.7881	1.5522	2.2165	0.8167	-1.0226	-1.9651

Consider now the non-linear model: $f_i = a_1 \sin(x_i + a_2)$, where the state is **s** = [a_1,a_2], to be estimated. This is the problem of fitting a sinusoid to a noisy sequence of data.

A) Write a Matlab function that computes the SSD cost function $E(s) = \sum_{i=1}^{N} (y_i - f_i(\mathbf{s}))^2$.

Input: the dataset (**x**,**y**) and the model hypothesis **s**

Output: the SSD value C(s)

3.3. Nonlinear LSE

B) Derive the (Nx2) Jacobian matrix of $\mathbf{f}$: $J = \begin{bmatrix} \frac{\partial f_1}{\partial a_1} & \frac{\partial f_1}{\partial a_2} \\ \cdots & \cdots \\ \frac{\partial f_{10}}{\partial a_1} & \frac{\partial f_{10}}{\partial a_2} \end{bmatrix}$ at a given state $[a_1,a_2]$

C) Write a Matlab function that makes this computation, that is:

Input: the dataset ($\mathbf{x}_i$) and model hypothesis $\mathbf{s}$
Output: the Jacobian matrix J($\mathbf{s}$)

3.3. Nonlinear LSE

```
function [E,J] = LSE_sinus(s,x,y)

% Model: f = a1*sin(x+a2)
% State: s = [a1,a2];

% Error vector (the SSD value is norm(E))

E = y-s(1).*sin(x+s(2));

% Derivatives of f:
% df/da1 = sin(x+a2)
% df/da2 = a1*cos(x+a2)

J(:,1) = sin(x+s(2));
J(:,2) = s(1)*cos(x+s(2));
```

3.3. Gauss-Newton Optimization

D) Finally, write a script that does the Gauss-Newton optimization:

0. Start from a state guess $\mathbf{s}=\mathbf{s}_0=[1,0]$
1. while(true)
   a. Compute the error vector ($\mathbf{y}$-$\mathbf{f}$($\mathbf{s}$))
   b. Compute the Jacobian matrix J($\mathbf{s}$)
   c. Compute the Gauss-Newton increment $\Delta s = \left[(J^T J)^{-1} J^T \right] (\mathbf{y} - \mathbf{f})$
   d. Increment the estimate $\mathbf{s} \rightarrow \mathbf{s}+\Delta\mathbf{s}$
   e. If the increment is $\Delta\mathbf{s} < 0.001$, exit the loop
2. Output = final estimate s.

E) Run the script, and display the results, like for the linear case (estimated sinusoid + measurements y)

F) Repeat step (D) using Levenberg-Marquardt instead of Gauss-Newton, starting with λ=1, and compare the results

3.3. Gauss-Newton Optimization

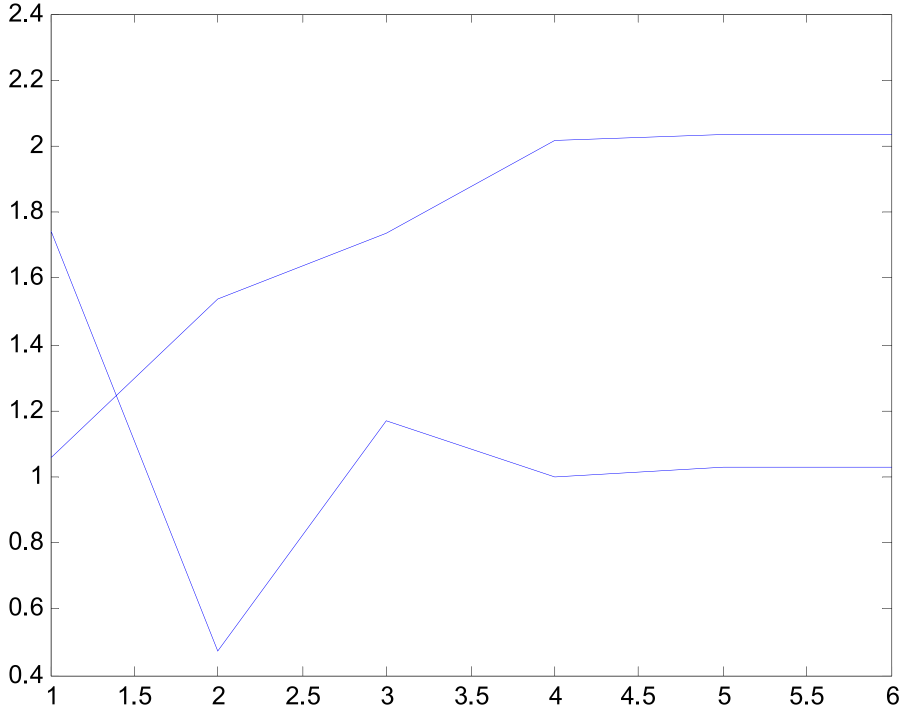
```
% 3.3 Nonlinear LSE optimization

clear;
close all;

% Data (column vectors)
x = [1 2 3 4 5 6 7 8 9 10]';
y = [1.7321 -0.0509 -1.4885 -1.8603 -0.7881 1.5522 2.2165 0.8167
-1.0226 -1.9651]';

% Model: f = a1*sin(x+a2)
% State: s = [a1,a2];

% Initial state
s0 = [1; 0];
```

3.3. Gauss-Newton Optimization

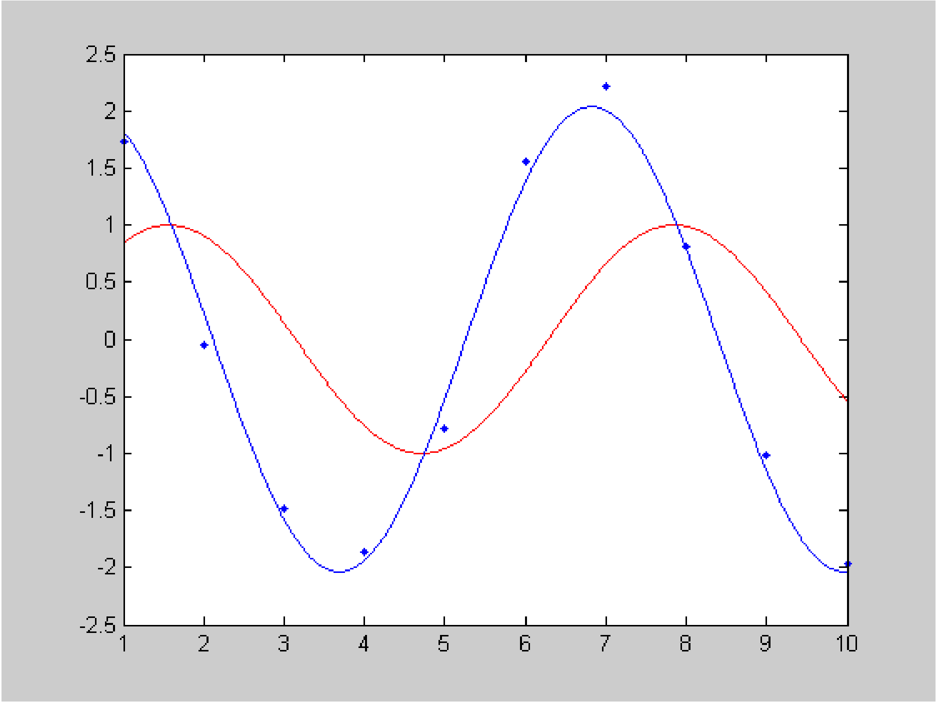
```
% Gauss-Newton Loop

% Initial state
s = s0;
loop_cond = true;
iter = 0;

while(loop_cond)

    iter = iter+1

    % Error vector and Jacobian matrix of f
    [E,J] = LSE_sinus(s,x,y);

    % Gauss-Newton increment
    ds = inv(J'*J)*J'*E;
    s = s+ds;

    if(norm(ds)<0.001)
        loop_cond = false;
    end
end
```

3.3. Gauss-Newton Optimization

```
s =
    1.0
    0.0

    1.0557
    1.7415

    1.5369
    0.4758

    1.7361
    1.1684

    2.0158
    1.0017

    2.0359
    1.0261

    2.0365
    1.0258

iter =

     6
```



```
% Plot results
% The estimated sinusoid is: f = s(1)*sin(x+s(2))

xx = 1:0.01:10;
y0 = s0(1)*sin(xx+s0(2));
yest = s(1)*sin(xx+s(2));

figure;
plot(xx,y0,'r');
hold;
plot(xx,yest);
plot(x,y,'b.');
```



3.3. Levenberg-Marquardt Optimization

```
% Levenberg-Marquardt Loop
% Initial LM parameter
lambda = 1;

% Initial state
s = s0;
loop_cond = true;
iter = 0;
while(loop_cond)

    % Error vector and Jacobian matrix of f
    [E,J] = LSE_sinus(s,x,y);

    % Levenberg-Marquardt increment
    ds = inv(J'*J+lambda*eye(2,2))*J'*E;

    [E2,J] = LSE_sinus(s+ds,x,y);

    % If the error decreases, accept new parameters and decrease lambda
    if(norm(E2)<norm(E))
        s = s+ds;
        lambda = lambda/10
    % Else, reject it and increase lambda
    else
        lambda = lambda*10
    end

    if(norm(ds)<0.001)
        loop_cond = false;
    end
end
```

3.10. RANSAC

Robust LSE – RANSAC

x	1	2	3	4	5	6	7	8	9	10
y	2.6892	3.3476	7.0	5.1878	5.6323	6.5843	6.7507	7.2644	8.5423	9.0124

Solve the linear regression problem (exercise 6) with an outlier, in two steps:

A) Implement the RANSAC algorithm on the dataset above, to remove the outliers (it should find only one):

REPEAT 100 times:

1 - Pick two points at random

2 - Fit a line (a_1,a_2) through the points

3 - Set a tolerance around the line ($y \pm \sigma$) to select outliers for this hypothesis

σ = standard deviation of the error

and keep the case with less outliers (if there is more than one, take just one).

B) After removing the outlier, do the standard LSE estimation (linear) with the remaining points

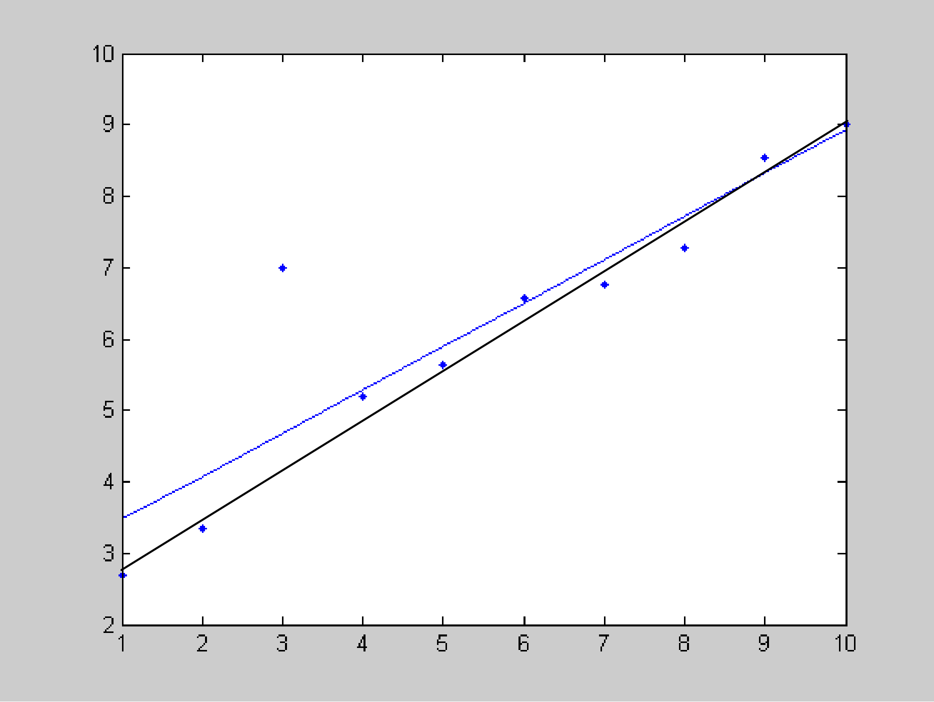
C) Compare the result with the (non-robust) standard LSE using the full dataset

3.10. RANSAC

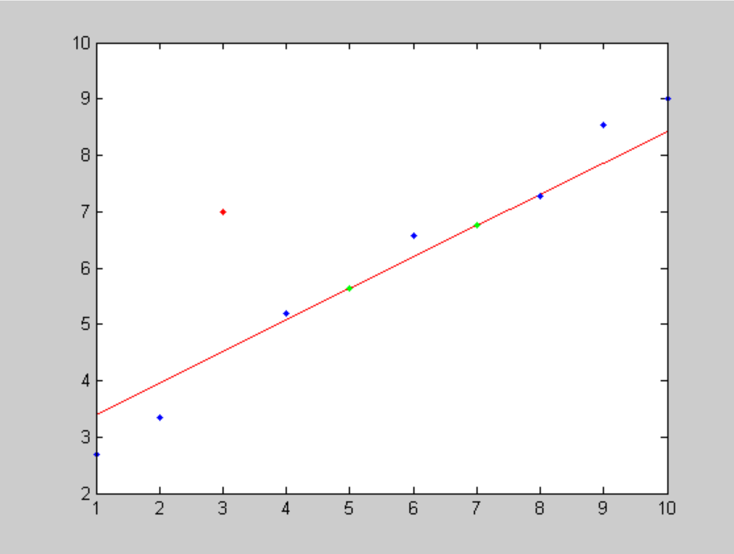
```
% 3.10 RANSAC

clear;
close all;

% Data (columns vectors)
x = [1 2 3 4 5 6 7 8 9 10]';
y = [2.6892 3.3476 7 5.1878 5.6323 6.5843 6.7507 7.2644 8.5423
9.0124]';

% Model: y = a1*x+a2
% State: s = [a1 a2];

l = length(x);

% Solve for the minimum LSE error with all points
A = [x ones(l,1)];
s = inv(A'*A)*A'*y;
```

3.10. RANSAC

```
% Plot the result of standard (non-robust) LSE

yest = s(1)*x+s(2);

figure;
plot(x,yest);
hold;
plot(x,y,'b.');
```



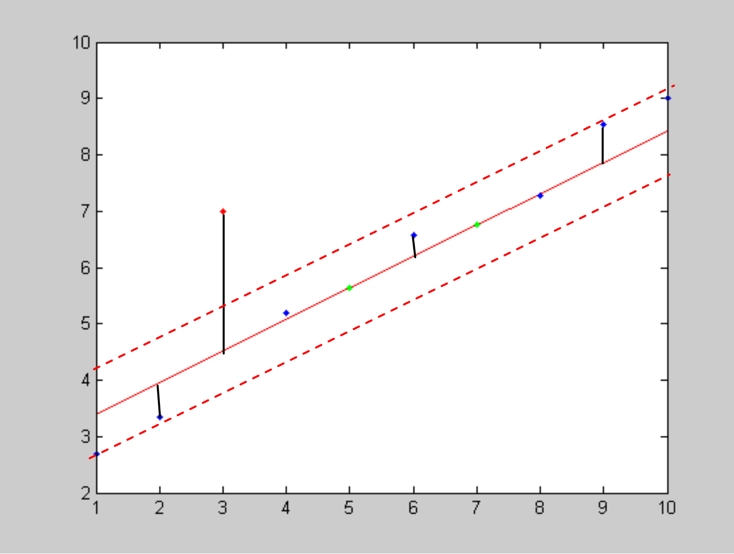
3.10. RANSAC




```
%%%% RANSAC Loop

nmin = l;
for(k=1:100)

    % Pick two points at random
    i1 = floor(l*rand(1,1)+1);
    i2 = floor(l*rand(1,1)+1);

    % Make sure that they are different!
    while(i2==i1)
        i2 = floor(l*rand(1,1)+1);
    end

    % Fit a line (exact solution with 2 points)
    A0 = [x(i1) 1; x(i2) 1];
    s0 = inv(A0)*[y(i1); y(i2)];
```

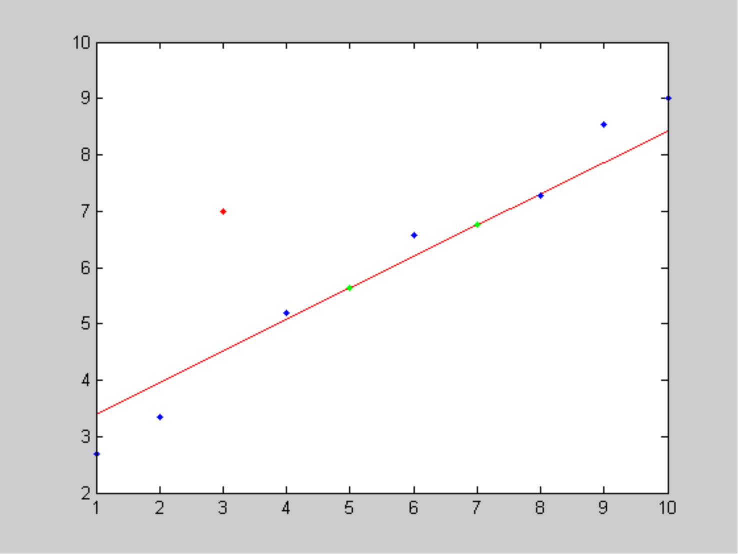
3.10. RANSAC

```
%%%% RANSAC Loop (continues)

% Compute the residual error
y0 = s0(1)*x+s0(2);
err = y0-y;
% Compute the standard deviation of the residual error
err_std = std(err);

% Compute the outliers:
% points with error higher than the standard deviation
outl = find(abs(err)>err_std);
nout = length(outl);

% Keep cases with least number of outliers
if(nout<nmin)
    i1ok = i1;
    i2ok = i2;
    sok = s0;
    yok = y0;
    out_ok = outl;
    inl_ok = find(abs(err)<=err_std);
    nmin = nout;
  end
end
```

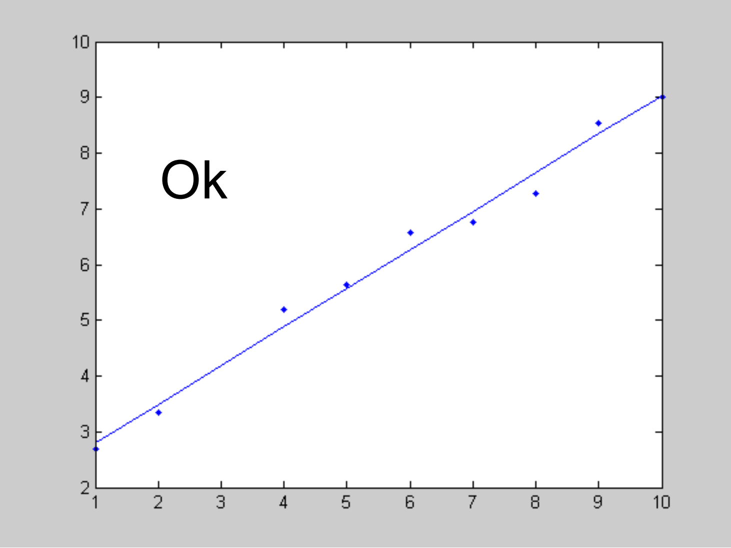


3.10. RANSAC

```
% Plot the result of RANSAC
% Estimated line with two points (best case)
yest = sok(1)*x+sok(2);
```




```
% Solve for the minimum LSE error
% with inlier points only
A = [x(inl) ones(length(inl),1)];
s = inv(A'*A)*A'*y(inl);

% Plot the result of LSE
% Final estimated line (with inliers only)
yest = s(1)*x(inl)+s(2);
```



LSE Estimation of 3D Pose

All of the previous exercises can be used to do a robust LSE estimation of 3D pose.
By defining with

- $\mathbf{x}_i$ = 3D body points
- $\mathbf{y}_i$ = 2D measured screen positions
- $\mathbf{s}$ = Body pose (6 parameters)
- $f(x_i,s)$ = the global 3D/2D projection
- $J(x_i,s)$ = Jacobian Matrices (2x6) of f_i

1. Do RANSAC using a simple 3-point fitting algorithm (P3P)

→ Result: a first pose estimate (best case) $\mathbf{s}_0$, and no outliers

2. With the inliers, apply Gauss-Newton (or Levenberg-Marquardt)

Dynamics and measurement models for Bayesian tracking

Motion Model

Write a function that generates a 1D random motion of the following type:

- Brownian motion
- WNA
- Constant acceleration ($a=9.81$) + perturbation

Where $w = \text{Gauss}(0,\sigma=1)$, $\Delta t=0.1$ for all the situations.

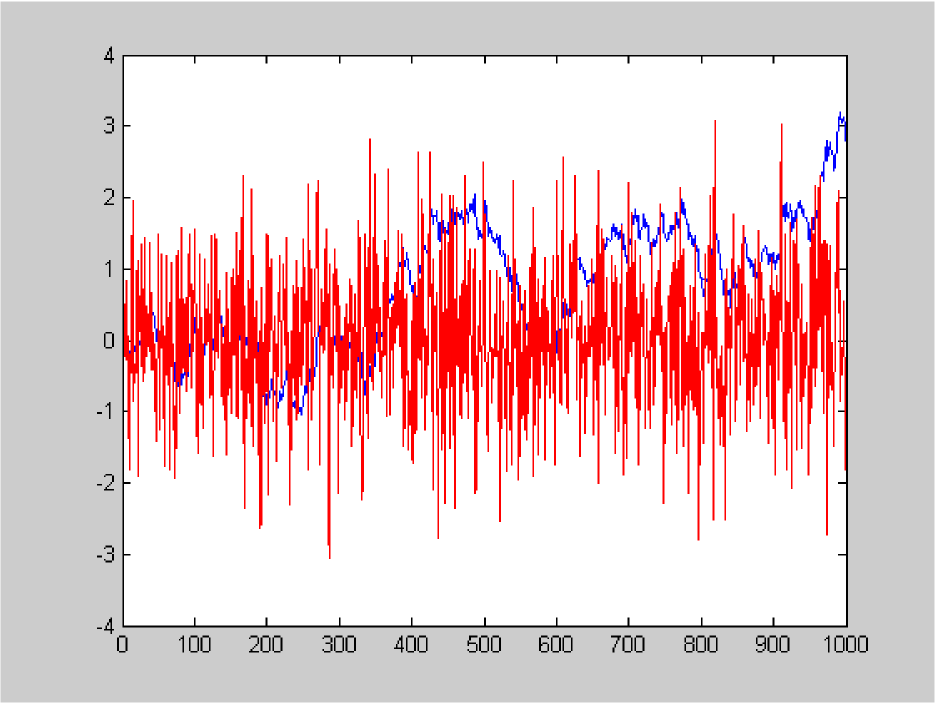
For each case, plot a corresponding trajectory in time: $p(t)$, with $t=(0, \Delta t, 2\Delta t, \ldots, N\Delta t)$, $N=1000$.

Afterwards, compute and plot the corresponding probabilistic motion models $P(\mathbf{s}_t|\mathbf{s}_{t-1})$ (in 1 or 2 dimensions, depending on the case)

4.1. Motion Models

```
% 4.1 Motion models

clear;
close all;

% Brownian motion

dt = 0.1;

p = zeros(1000,1);
v = zeros(1000,1);

for t=2:1000
    w = randn(1,1);
    p(t) = p(t-1) + w*dt;
    v(t) = w;
end

figure;
plot(p,'b');
hold;
plot(v,'r');
```

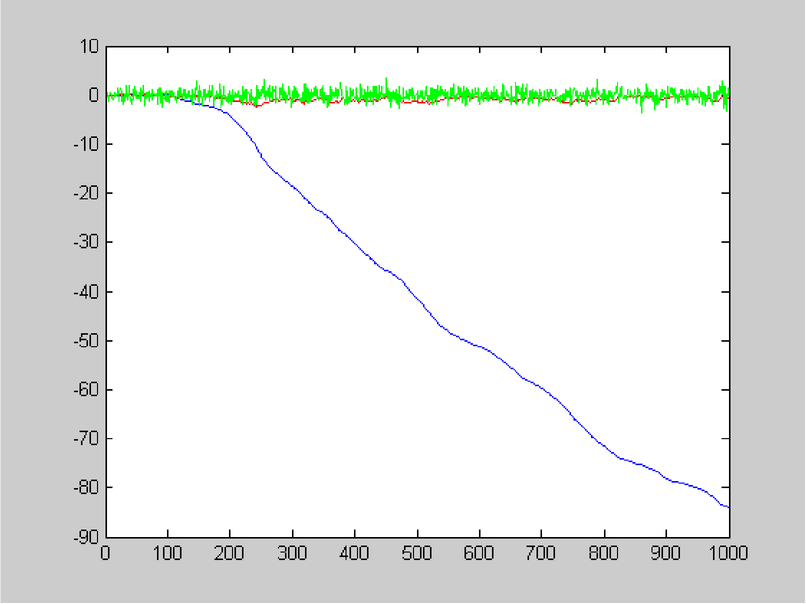
Position (blue) and velocity (red)



4.1. Motion Models

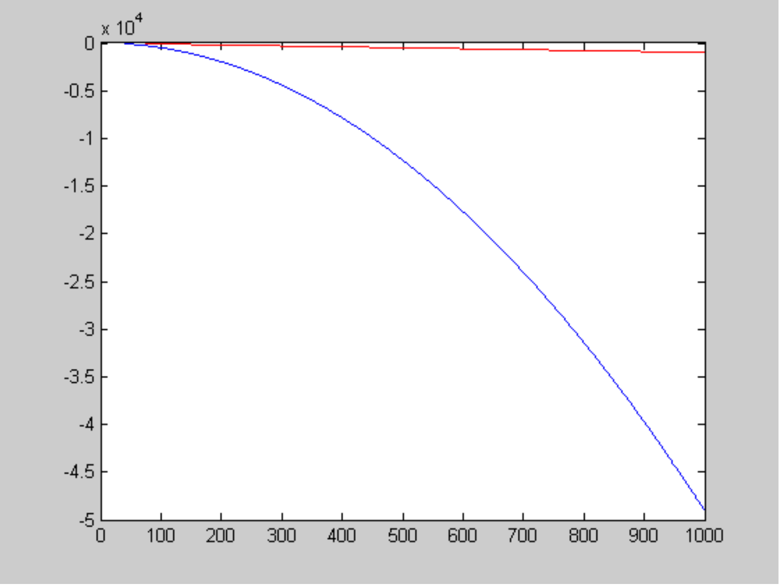
```
% WNA motion

dt = 0.1;

p = zeros(1000,1);
v = zeros(1000,1);
a = zeros(1000,1);

for t=2:1000
    w = randn(1,1);

    p(t) = p(t-1) + v(t-1)*dt + 0.5*w*dt^2;
    v(t) = v(t-1) + w*dt;
    a(t) = w;
end

figure;
plot(p,'b');
hold;
plot(v,'r');
plot(a,'g');
```

Position (blue)
Velocity (red)
Acceleration (green)



4.1. Motion Models

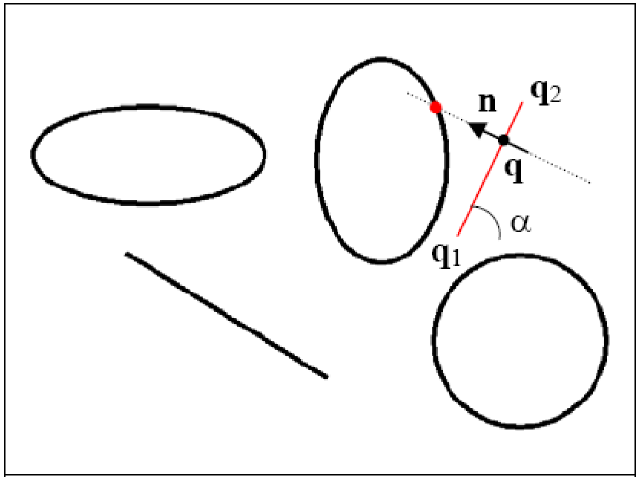
```
% Perturbed acceleration model

dt = 0.1;

p = zeros(1000,1);
v = zeros(1000,1);
a = zeros(1000,1);
a0 = -9.81;

for t=2:1000
    w = randn(1,1);

    p(t) = p(t-1) + v(t-1)*dt + 0.5*(w+a0)*dt^2;
    v(t) = v(t-1) + (w+a0)*dt;
    a(t) = w+a0;
end

figure;
plot(p,'b');
hold;
plot(v,'r');
plot(a,'g');
```



4.4. Likelihood model for edges

Consider now a segment model: the state s is (x_1, y_1, α) the position and orientation of the segment (the length is 10).

$$\mathbf{q} = k\mathbf{q}_1(s) + (1-k)\mathbf{q}_2(s)$$

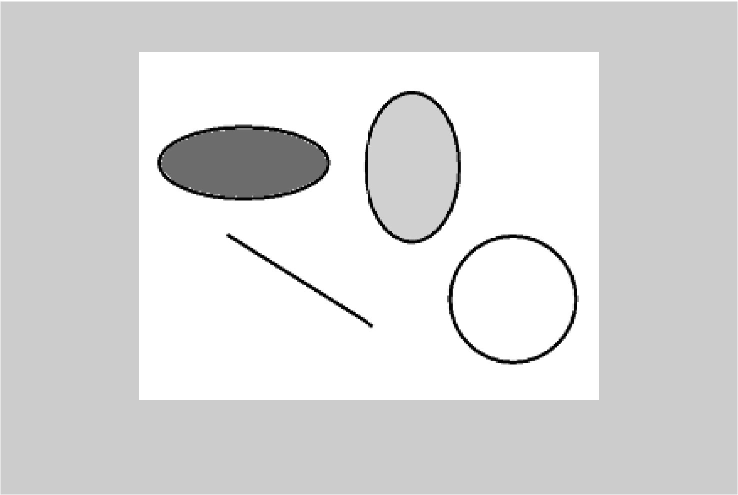


With this model, given an image (example above) compute the Likelihood $P(z|s)$:

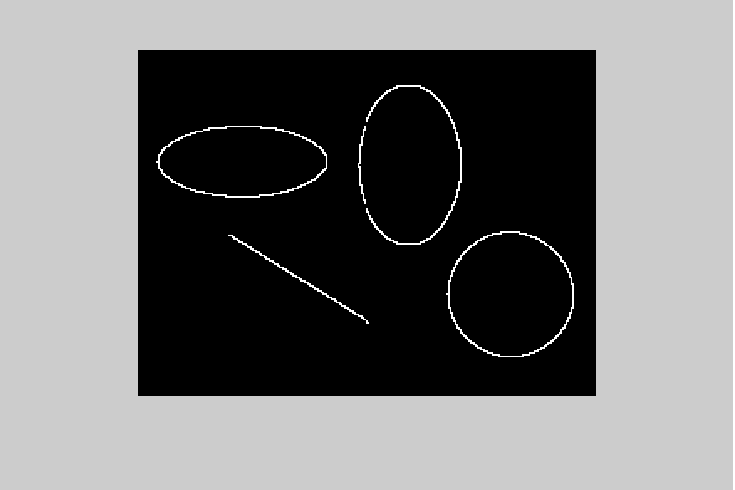
- take a set of 11 equidistant points in the segment (k=0,0.1,0.2,…,1)
- take the normal vector to the segment $\mathbf{n} = (-\sin(\alpha), \cos(\alpha))$
- From a point $\mathbf{q}(k_i)$, search along the normal directions $\mathbf{q}(k_i)+j\mathbf{n}$ where j= 0,-1,1,-2,2,…,-L,L (up to a length L=5 in the two directions)
- Stop the search if a black pixel is found (pixel coordinates → the points $\mathbf{q}+j\mathbf{n}$ need to be rounded to integers), or if the maximum distance L is reached
- The result is the distance of the nearest edge, l_{ok}, or L
- Now weight the distance l_{ok} with a Gaussian: $P(l_{ok})$ = Gauss(0,1)
- Finally, multiply all the 11 Gaussians, to get the Likelihood $P(\mathbf{z}|\mathbf{s})$

4.4. Likelihood model for edges

```
% 4.4 Likelihood function for edge maps

clear;
close all;

% The original image
I = imread('Image.bmp');
figure;
imshow(I);
```

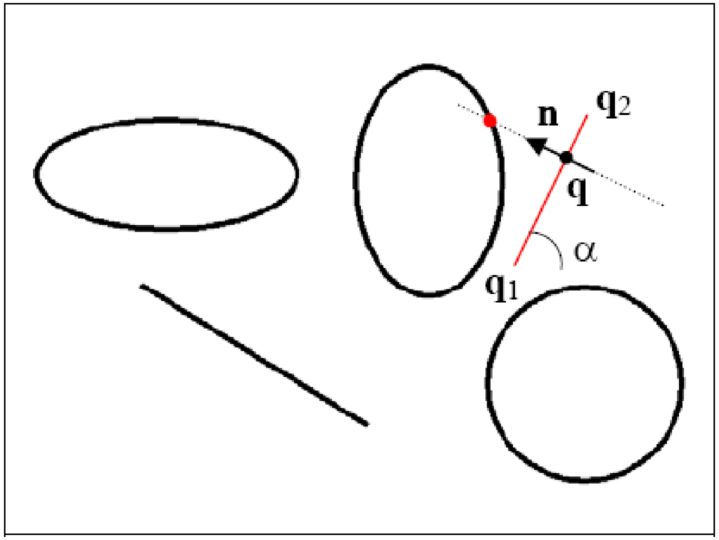


```
% The Canny Edge Map
E = edge(I,'canny');
```



4.4. Likelihood model for edges

```
% Model = a segment wiht fixed length
l = 40;

% State vector
% s = [x₁;y₁;α];

x1 = s(1);
y1 = s(2);
alpha = s(3)*pi/180;

q1 = [x1;y1];

q2 = [x1+l*cos(alpha);y1+l*sin(alpha)];

% Pre-compute the J 'exploration points' (along the normal)
J = 10;

jv(2:2:2*J+1) = -(1:J);
jv(3:2:2*J+2) = 1:J;
jv(1) = 0;
```

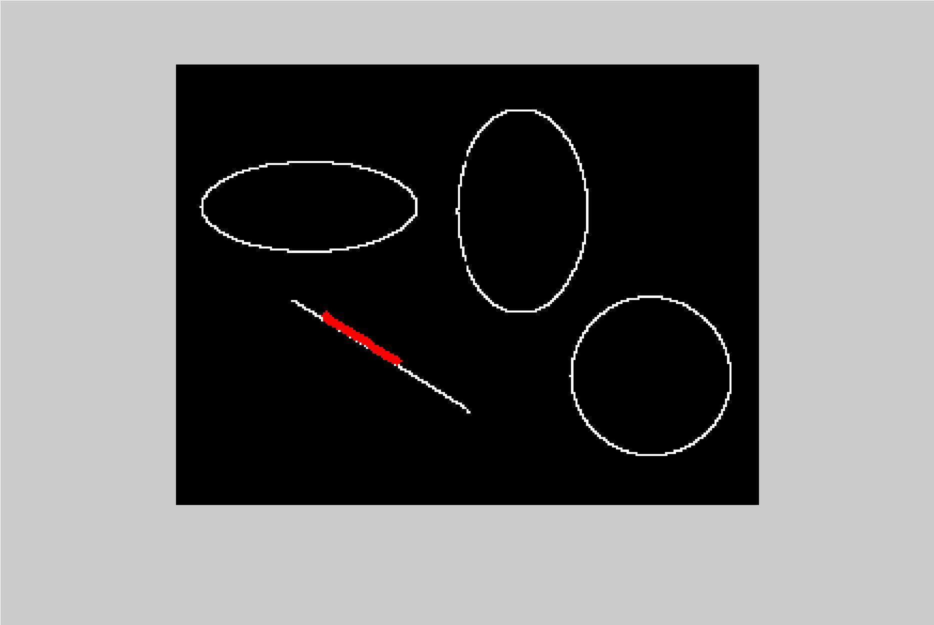


$\Leftarrow$ jv = [0,-1,1,-2,2,-3,3,…,-J,J]

4.4. Likelihood model for edges

```
sig = 8;                                % Variance of the Gauss. Likelihood
Lik = 1;                                % Initialize Likelihood to 1

for i=1:1:l                             % Loop: Take l points along the segment

    k = (i-1)/(l-1);
    q = k*q1+(1-k)*q2;
    n = [-sin(alpha);cos(alpha)];
    d_ok = J;                           % In case 'no edge', set distance to J
    j=1;

    while(j<=length(jv))                % Search along the normal
        d = jv(j);
        qp = round(q+d*n);

        if(E(qp(2),qp(1))~=0)                   % If an edge has been found…
                d_ok = norm(qp-q)*sign(d);      % Compute distance to the segment
                j = length(jv)+1;               % exit the loop
        end
        j = j+1;
    end

    Lik = Lik*exp(-0.5*d_ok^2/sig^2)            % Contribution to the Likelihood
end
```

4.4. Likelihood model for edges

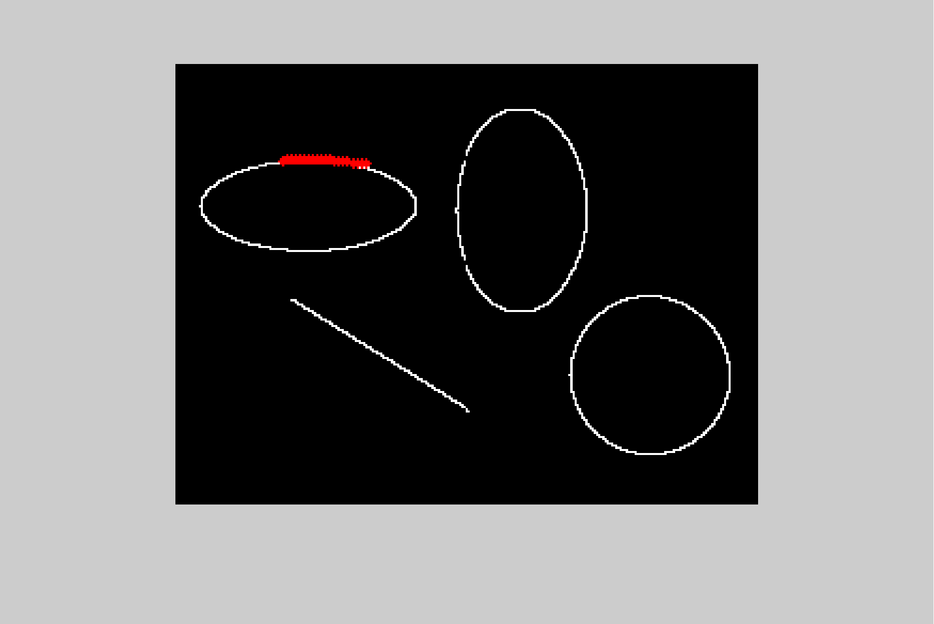


Likelihood = 1

```
% Best matching example
s = [70;118;32];

figure;
imshow(E);
hold;
plot([q1(1),q2(1)],[q1(2),q2(2)],'r');
for i=1:ind
    if(found(i)==1)
        plot(q_ok(1,i),q_ok(2,i),'r.');
    else
        plot(q_ok(1,i),q_ok(2,i),'b.');
    end
end
```

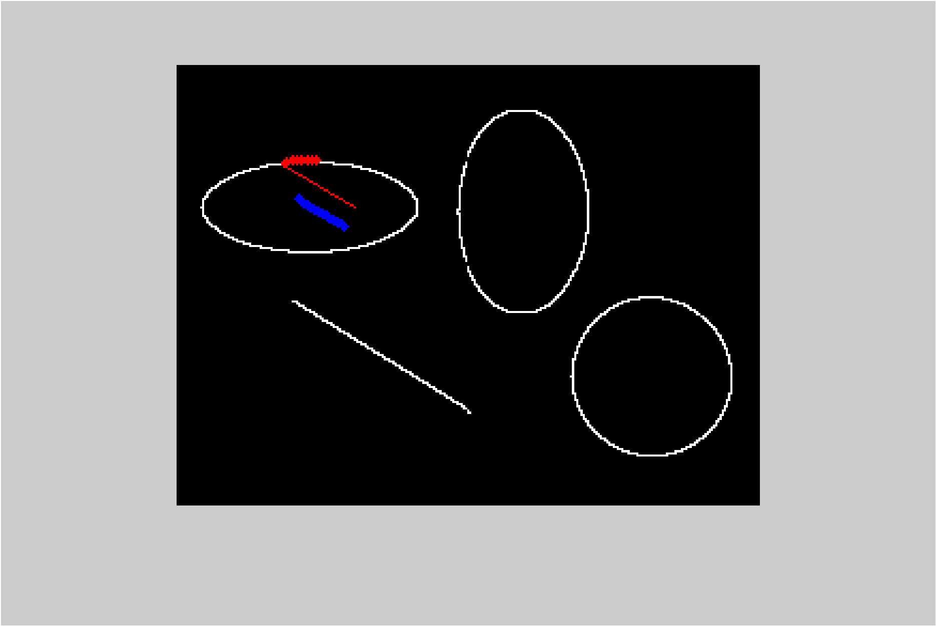
4.4. Likelihood model for edges

```
% Not perfect matching
s = [50;47;0];

figure;
imshow(E);
hold;
plot([q1(1),q2(1)],[q1(2),q2(2)],'r');
for i=1:ind
    if(found(i)==1)
        plot(q_ok(1,i),q_ok(2,i),'r.');
    else
        plot(q_ok(1,i),q_ok(2,i),'b.');
    end
end
```



Likelihood = 0.6735

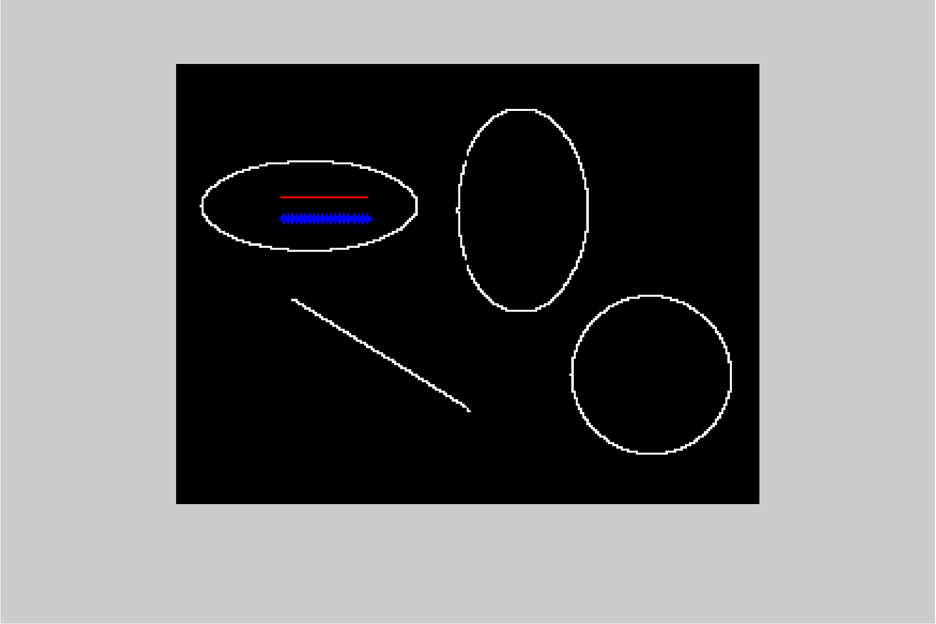
4.4. Likelihood model for edges



```
Likelihood = 2.9759e-006
```

```
% 'Intermediate' case
s = [50;47;30];

figure;
imshow(E);
hold;
plot([q1(1),q2(1)],[q1(2),q2(2)],'r');
for i=1:ind
    if(found(i)==1)
        plot(q_ok(1,i),q_ok(2,i),'r.');
    else
        plot(q_ok(1,i),q_ok(2,i),'b.');
    end
end
```

4.4. Likelihood model for edges



Likelihood = 1.6374e-007

```
% Worst case (no matching points)
s = [50;63;0];

figure;
imshow(E);
hold;
plot([q1(1),q2(1)],[q1(2),q2(2)],'r');
for i=1:ind
    if(found(i)==1)
        plot(q_ok(1,i),q_ok(2,i),'r.');
    else
        plot(q_ok(1,i),q_ok(2,i),'b.');
    end
end
```

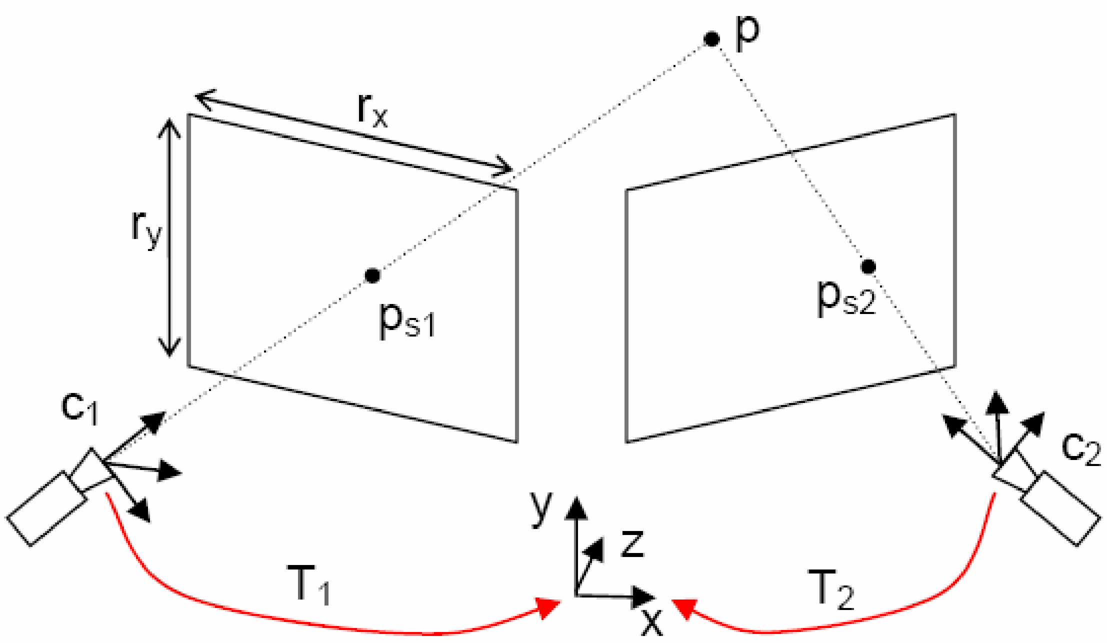
Bayesian Tracking

5.1 Extended Kalman Filter

3) **Extended Kalman Filter: track a flying ball (DLR system)**

Suppose to have two cameras (a stereo system), looking a ball thrown across the room.

The setup is the one described in Exercise 3-Lecture 4, as below indicated



5.1 Extended Kalman Filter

The ball $\mathbf{p}=(x,y,z)$ describes a parabolic trajectory during the flight, and its motion model can be described by a constant gravity acceleration towards the bottom ($-\mathbf{g}$) + a small random component w (e.g. air resistance in different points of the trajectory).
This motion (described in Lecture 4 - Slide 10) gives a probabilistic state model:
$P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}\,(A\,\mathbf{s}_{t-1}+C,\ B\Lambda_w B^T)$.

with

$$A = \begin{bmatrix} I & I\Delta t \\ 0 & I \end{bmatrix} \qquad B = \begin{bmatrix} I\Delta t^2 \\ I\Delta t \end{bmatrix} \qquad C = \begin{bmatrix} -\mathbf{g}I\Delta t^2 \\ -\mathbf{g}I\Delta t \end{bmatrix}$$

$(A\,\mathbf{s}_{t-1}+C)$ is the *prediction* of $\mathbf{s}_t$
$\mathbf{g} = [0\ 981\ 0]$ is the gravity acceleration (y direction)
$\Lambda_w = \text{diag}(0,0,0,1,1,1)$ is the covariance of motion noise (acceleration noise)
$\Delta t = 0.1$ is the time sampling interval (10 frames/sec).

5.1 Extended Kalman Filter

The state **s** is a (3+3)-vector (position+velocity), and positions are measured in [*mm*]. The measurement **z** (solution of the other exercise) is the collection of two positions located on the two camera images:
$\mathbf{z} = (\mathbf{p}_{s1}, \mathbf{p}_{s2})$, which are 4 image coordinates (x_1, y_1, x_2, y_2).

The measurement model, for a given hypothesis **p**, gives an expected measurement

$$p_{c1} = (x_{c1}, y_{c1}, z_{c1}) = T_1 p \qquad p_{c2} = (x_{c2}, y_{c2}, z_{c2}) = T_2 p$$

$$p_{s1,\exp}(p_{c1}) = \left(\frac{x_{c1}}{z_{c1}} f + \frac{r_x}{2}, \frac{y_{c1}}{z_{c1}} f + \frac{r_y}{2} \right) \qquad p_{s2,\exp}(p_{c2}) = \left(\frac{x_{c2}}{z_{c2}} f + \frac{r_x}{2}, \frac{y_{c2}}{z_{c2}} f + \frac{r_y}{2} \right)$$

where $\mathbf{p}_{c1}$ and $\mathbf{p}_{c2}$ are the coordinates of p in the two cameras (extrinsic transformations T_1, T_2), and $\mathbf{p}_{s1,\exp}$, $\mathbf{p}_{s2,\exp}$ are the projections on the screens (intrinsic transformation: **f**, r_x, r_y).

5.1 Extended Kalman Filter

EKF: Linearize motion *f* and measurement *h* models

→ Compute the Jacobian matrices of f and h.

f is linearized around $\mathbf{s}_{t-1}$: $A_t = \left.\frac{\partial f}{\partial s}\right|_{s_{t-1}}$

Prediction :

$$s_t^- = f(s_{t-1}, 0)$$

$$S_t^- = A_t S_{t-1} A_t^T + \Lambda_w$$

h is linearized around $\mathbf{s}_t$: $C_t = \left.\frac{\partial h}{\partial s}\right|_{s_t}$

Correction :

$$s_t = s_t^- + G_t(z_t - h(s_t^-, 0))$$

$$S_t = S_t^- - G_t C_t S_t^-$$

$$G_t = S_t^- C_t^T (C S_t^- C_t^T + \Lambda_v)^{-1}$$

5.1 Extended Kalman Filter

A. Compute the Jacobian matrix $J = \dfrac{\partial \mathbf{z}_{exp}}{\partial \mathbf{s}}$ at given hypothesis $\mathbf{s}$, (write a Matlab function returning J, with input $\mathbf{s}$)

B. Implement the Extended Kalman Filter (equations in Lecture 5-Slide 19).
NOTE: the motion model is already linear, so the Jacobian is just A.

C. Do a simulated experiment (real vs. estimated state), where the ball is thrown from the ground:

Real initial state $\mathbf{p}_0 = [0,0,0]$ with initial velocity $\mathbf{v}_0 = [0, 10, 10]$ (forward z, up y).
Initial state hypothesis: $\mathbf{p}_0 = [0,0,0]$, $\mathbf{v}_0 = [0,0,0]$ (no knowledge).
Perturbation of acceleration during the flight = Gaussian random $\mathbf{w}$, with covariance 1.

5.1 Extended Kalman Filter

```
function [z1exp, z2exp, J1, J2] = Meas_Stereo(s,f,rx,ry,T1,T2)

% The ball is a point in World coordinates
pw = s(1:3);

% We transform it into camera 1 coordinates (extrinsic T1)
% and then we apply the camera projection (exercise 2.3)

pbH = T1*[pw;1];

pb1 = pbH(1:3);

z1exp = GlobalT(zeros(6,1),pb1,f,rx,ry);

% The same for camera 2 (with transformation T2)

pbH = T2*[pw;1];

pb2 = pbH(1:3);

z2exp = GlobalT(zeros(6,1),pb2,f,rx,ry);

% (The Jacobian matrices are a bit long to write here...)
```

5.1 Extended Kalman Filter

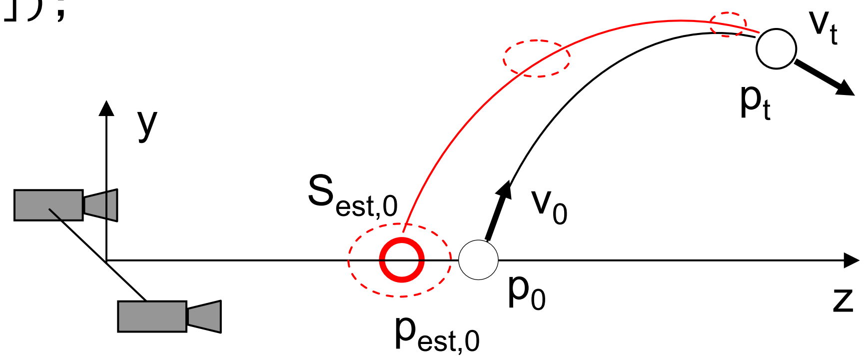
```
% 5.3 Extended Kalman Filter (stereo system)

clear;
close all;

% Intrinsic parameters for both cameras
f = 1000;
rx = 640;
ry = 480;

% Constant Transformation matrices, from World to Camera 1 and 2
T1 = [eye(3,3), [200; 0; 0]; 0 0 0 1];
T2 = [eye(3,3), [-200; 0; 0]; 0 0 0 1];

% Motion model:
% Perturbed acceleration

% Sample time (we do 100 steps = 10 sec. of simulation)
dt = 0.1;

% Constant (gravity) acceleration term
% a0 = g*dt (acceleration over a dt time interval)
a0 = [0; -981*dt; 0];
```

5.1 Extended Kalman Filter

```
% This is the real system state ("ground truth")
% used to do the simulation (in a real system, we do not know it!)

% Initial Position (real value)
p0 = [0;0;1200];

% Initial Velocity: forwards (z), and upwards (y)
% (real value)
v0 = [0;300;100];

% This is the motion model
A = [eye(3,3), eye(3,3)*dt; zeros(3,3), eye(3,3)];

B = [0.5*eye(3,3)*dt^2; eye(3,3)*dt];

% Covariance of acceleration noise
lw = 1;

% Covariance matrix of motion process
Qw = B*lw*B';

% Covariance matrix of measurement process
lv = 1;
Qv = lv*eye(4,4);
```

5.1 Extended Kalman Filter

```
% Real state (initial state)
p_real = p0;
v_real = v0;

% Initial state estimate and covariance matrix
% state = (position,velocity)
p_est = [0;0;1000];
v_est = [0;0;0];
s_est = [p_est;v_est];
S_est = diag([10 10 100 10 10 10]);
```



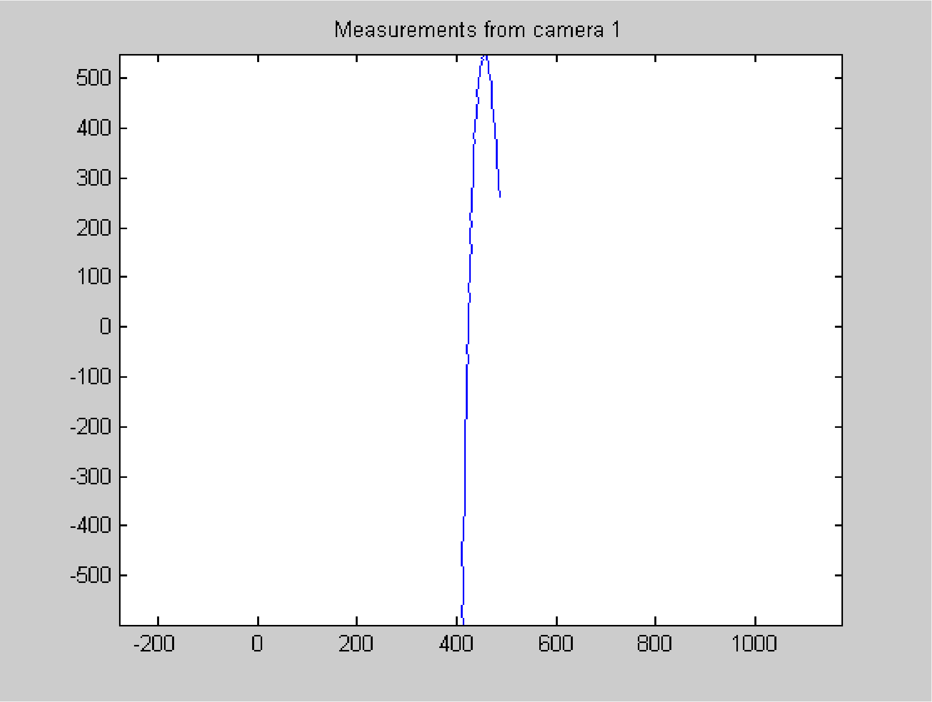
5.1 Extended Kalman Filter

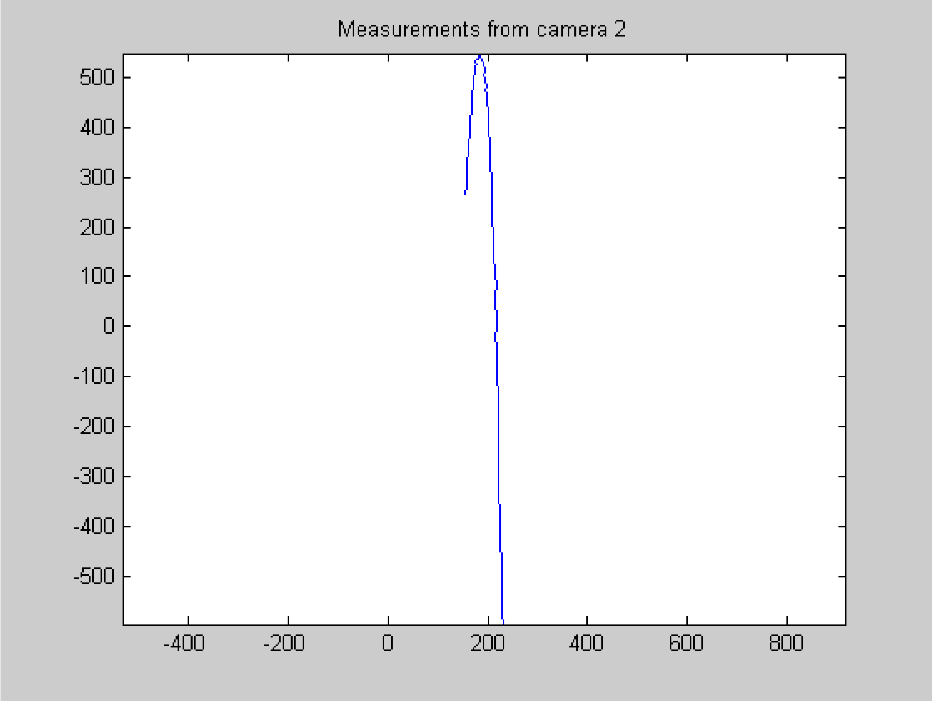
```
% Main loop
for t=2:100

    % Real motion (simulate the random process)
    w = sqrt(lw)*randn(3,1);

    p_real = p_real + v_real*dt + 0.5*w*dt^2 + 0.5*a0*dt^2;
    v_real = v_real + w*dt + a0*dt;
    a_real = w + a0;

    % Extended Kalman Filter: Prediction Step

    % Predicted state (=prior) is obtained applying motion model
    % without noise (expected motion)
    p_pred = p_est + v_est*dt + 0.5*a0*dt^2;
    v_pred = v_est + a0*dt;

    % Covariance matrix of the prediction
    S_pred = A*S_est*A' + Qw;

    % Expected measurement h(p_pred,0) and Jacobian matrix in (p_pred)
    [z1exp, z2exp, J1, J2] = Meas_Stereo(p_pred,f,rx,ry,T1,T2);
    % C is the Jacobian matrix of the full measurement vector (4x4)
    C = [J1;J2];
```

5.1 Extended Kalman Filter

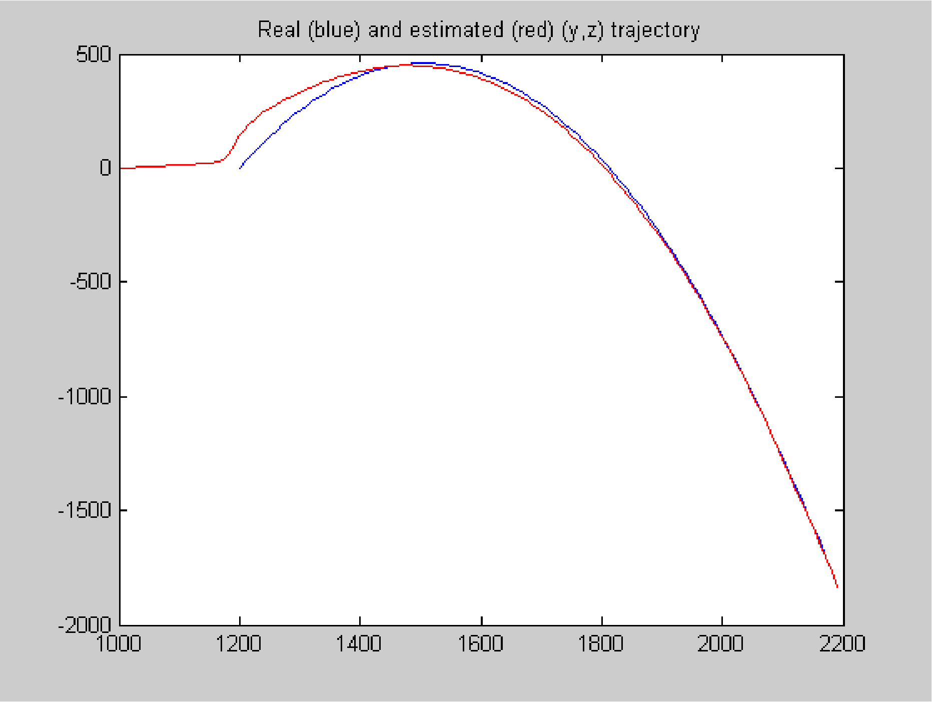
```
        (continues)

    % 'Simulate' the real measurement:
    % At the real position, compute the expected measurement
    % and add the Gaussian measurement noise
    [z1real, z2real, J1, J2] = Meas_Stereo(p_real,f,rx,ry,T1,T2);
    z1real = z1real+sqrt(lv)*randn(2,1);
    z2real = z2real+sqrt(lv)*randn(2,1);

    % Extended Kalman Filter: Correction step

    % Kalman Gain
    G = S_pred*C'*inv(C*S_pred*C'+Qv);

    % Update the state estimate (=posterior)
    s_est = [p_pred; v_pred] + G*([z1real; z2real] - [z1exp; z2exp]);
    p_est = s_est(1:3);
    v_est = s_est(4:6);

    % Update the covariance matrix
    S_est = S_pred-G*C*S_pred;

end
```

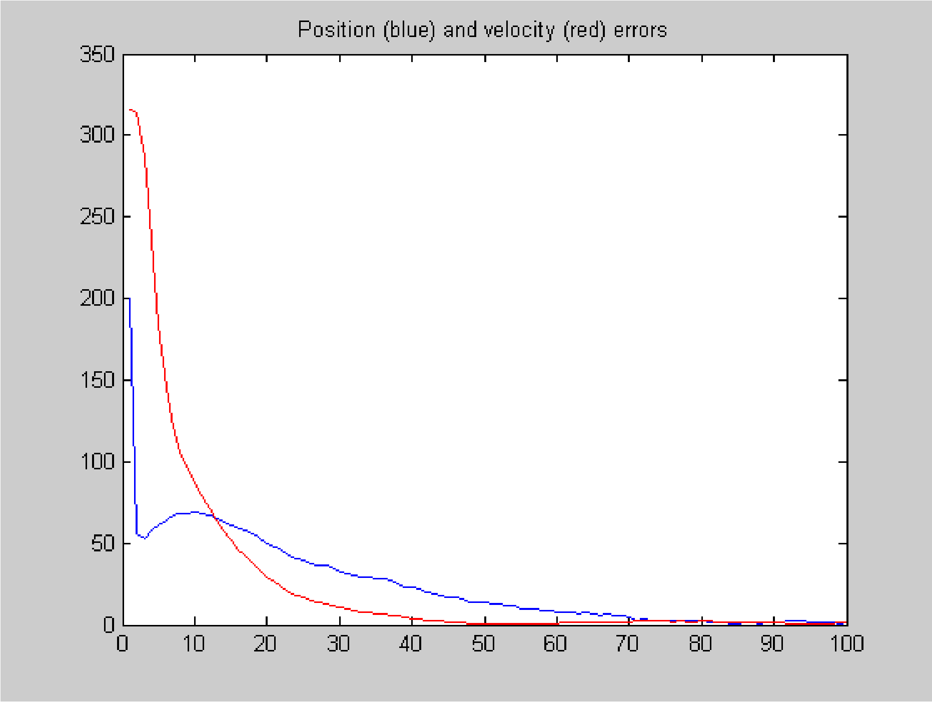
5.1 Extended Kalman Filter





5.1 Extended Kalman Filter





Video example (DLR)