Assignment A10: Tracking

Clinton Fernandes

u1016390

04/20/2016

A10

CS 6320

1. Introduction

The aim of this assignment is to study the methods of object tracking i.e., given a video (sequence of images), determine the position of the object in each of the image frame. There are different strategies used in tracking, and in this assignment, tracking by Background subtraction and tracking motion models (using Kalman filter) will be studied. Also, we will compare the use of 3 distance measures for red ball tracking: (1) Euclidean distance, (2) Mahalanobis distance, (3) Gaussian probability.

Questions that can be answered while performing the exercise are:

1. how robust a threshold is for each of the distance functions (i.e., look at thresholds on the distance for the different measures and the resulting classification).
2. How does the performance of the methods change with the sensor noise?
3. How does the performance of the methods change if noise is added to the image sequence?

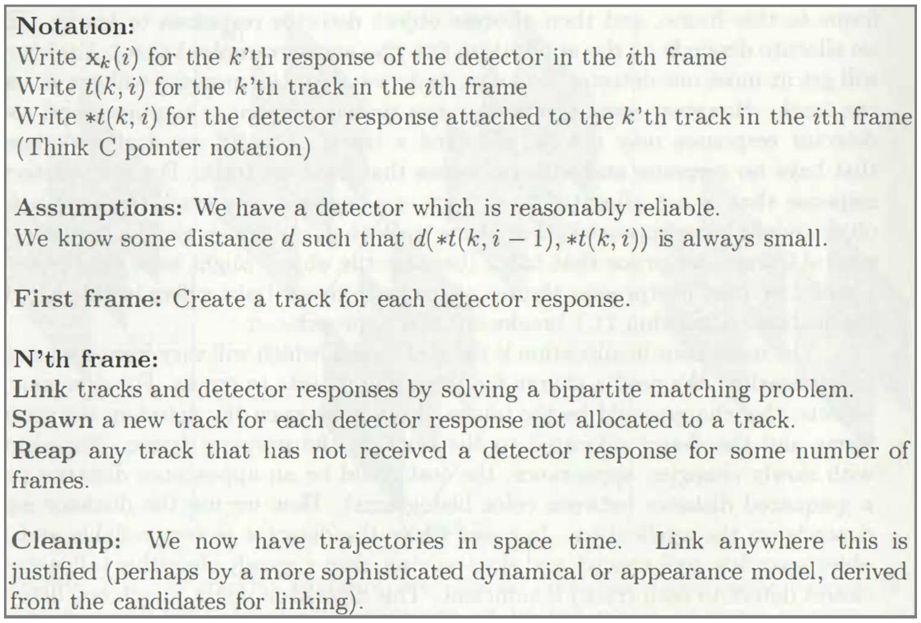
2. Method

As has been discussed in the introduction, there are different strategies that can be used for tracking, the two simple ways to track objects are tracking by detection and tracking by matching.

2.1. Tracking by detection (Background Subtraction)

In this, we have a strong model of the object, strong enough to identify it in each frame. We find it, link up the instances, and we have a track. We can build a reliable detector for the object we wish to track.

Background subtraction is often a good enough detector in applications where the background is known and all trackable objects look different from the back-ground. In such cases, it can be enough to apply background subtraction and regard the big blobs as detector responses. Although simple, this method has weaknesses - for example, if an object is still for a long time, it might disappear from tracking.



Notation:
Write $\mathbf{x}_k(i)$ for the k'th response of the detector in the ith frame
Write $t(k, i)$ for the k'th track in the ith frame
Write $*t(k, i)$ for the detector response attached to the k'th track in the ith frame
(Think C pointer notation)

Assumptions: We have a detector which is reasonably reliable.
We know some distance d such that $d(*t(k, i-1), *t(k, i))$ is always small.

First frame: Create a track for each detector response.

N'th frame:
Link tracks and detector responses by solving a bipartite matching problem.
Spawn a new track for each detector response not allocated to a track.
Reap any track that has not received a detector response for some number of frames.

Cleanup: We now have trajectories in space time. Link anywhere this is justified (perhaps by a more sophisticated dynamical or appearance model, derived from the candidates for linking).

Algorithm: Tracking by Detection.

To track an object by Background Subtraction, we need to use 2 functions: CS5320_background_sub_tracking function and CS5320_bead_tracking.

***F-1: CS5320_background_sub_tracking**

This function takes in the image sequence as the input and the output of this function is the subtracted images (subtraction of the average image from original images). Here an image's foreground is extracted for further processing.

Input: im(k).im (mxnxd array): k_th image

Output: t_im (mxnxp array): subtracted images

<u>Algorithm:</u>

```
• [row, col] = size of an image
• For i = 1:maxImageNumber
•         For plane = 1:3
•                 Sum(row, col, plane) = Sum(row, col, plane) + imSeq(i).im(row, col, plane)
•         end
• end
• For plane = 1:3
•         mean(: ,: , plane) = Sum(: ,: , plane) / maxImageNumber
• end
• For i = 1:maxImageNumber
•         For plane = 1:3
•                 Temp = imSeq(i).im(:,:, plane) - mean(:,:, plane)
•         end
•         For all pixels of Temp
•                 t_im = norm(Temp)
•         end
• end
```

We have now got t_im which represents the background subtracted images. After this, we have to use another function:

***F-2: CS5320_bead_tracking.**

This function tracks the position of the object in consideration (in our case, the bead) as it moves in the image frame.

Input to this function is t_im, the background subtracted image (i.e., the output of the CS5320_background_sub_tracking function). Output of the function is the position of the center (in rows, columns) of the object (Bead) as it moves in the image frame.

Input: t_im (mxnxp array): subtracted images (image from average)

Output: track (mxn array): [rows,cols] of the center of moving object

<u>Algorithm:</u>

```
• track = []
• set a threshold for the distance
• For all images
    o For all pixels of the image
        ▪ If t_im > threshold
            • set temp_im = 1
        ▪ end
    o end
    o Now find connected components: CC = bwconncomp(temp_im(:,:,imageNum))
    o Find the largest connected component
    o Find the Centroid of the largest connected component: S = regionprops(CC,'Centroid')
    o [r, c] == positions of the Centroid
    o Add [r,c] to track: track= [track;[r,c]];
• end
```

2.2. Tracking by Matching (with Kalman Filter)

Kalman filter is a technique for filtering and prediction in linear Gaussian systems. The Kalman filter implements belief computation for continuous states. The Kalman filter represents beliefs by the moments parameterization: At time t, the belief is represented by the the mean and the covariance.

The state transition probability p(xt | ut , xt -1) must be a linear function in its arguments with added Gaussian noise. This is expressed by the following equation:

$$x_t \;=\; A_t x_{t-1} + B_t u_t + \varepsilon_t$$

Here Xt and Xt -l are state vectors, and Ut is the control vector at time t.

$$x_t = \begin{pmatrix} x_{1,t} \\ x_{2,t} \\ \vdots \\ x_{n,t} \end{pmatrix} \quad \text{and} \quad u_t = \begin{pmatrix} u_{1,t} \\ u_{2,t} \\ \vdots \\ u_{m,t} \end{pmatrix}$$

At and Bt are matrices. At is a square matrix of size n x n, where n is the dimension of the state vector Xt · Bt is of size n x m, with m being the dimension of the control vector Ut. By multiplying the state and control vector with the matrices At and Bt, respectively, the state transition func-tion becomes linear in its arguments. Thus, Kalman filters assume linear system dynamics.

The random variable Et is a Gaussian random vector that mod-els the uncertainty introduced by the state transition. It is of the same dimension as the state vector. Its mean is zero, and its covariance will be denoted Rt.

The measurement probability p(zt I Xt) must also be linear in its argu-ments, with added Gaussian noise:

$$z_t \;=\; C_t x_t + \delta_t$$

Here Ct is a matrix of size k x n, where k is the dimension of the measure-ment vector Zt. The vector i5t describes the measurement noise. The distri-bution of dt is a multivariate Gaussian with zero mean and Qt Covariance.

```
Algorithm Kalman_filter($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):

    $\bar{\mu}_t = A_t\, \mu_{t-1} + B_t\, u_t$
    $\bar{\Sigma}_t = A_t\, \Sigma_{t-1}\, A_t^T + R_t$

    $K_t = \bar{\Sigma}_t\, C_t^T (C_t\, \bar{\Sigma}_t\, C_t^T + Q_t)^{-1}$
    $\mu_t = \bar{\mu}_t + K_t(z_t - C_t\, \bar{\mu}_t)$
    $\Sigma_t = (I - K_t\, C_t)\, \bar{\Sigma}_t$
    return $\mu_t, \Sigma_t$
```

We know that, here we require a state model that models the object. The states of the model can be position velocity or acceleration.

For a **constant velocity** model:

State vector: Xt = [x y vx vy]

Process matrix: A = [1 0 del_t 0 ; 0 1 0 del_t ;0 0 1 0 ; 0 0 0 1]

Here, x and y are positions, vx and vy are velocities in x and y direction. If we had to have accelerations in the state vector, then ax and ay represent the accelerations in x and y directions. del_t is the time step between image sequences.

***F-3:** The **CS5320_const_vel** function gives a constant velocity forward simulation.

Input to the function:

x0 (float): initial x location

y0 (float): initial y location

vx0 (float): x speed

vy0 (float): y speed

del_t (float): time step

max_t (float): maximum time for simulation

R (4x4 array): covariance of process

Output: trace (kx4 array): state trace {trace(i,:): state vector at step i}

Algorithm:

```
• trace = [x0, y0, vx0, vy0];
• for all time steps = del_t:max_time
    o A = [1 0 del_t 0; 0 1 0 del_t;0 0 1 0; 0 0 0 1];
    o temp = A*[x0; y0; vx0; vy0] + mvnrnd(zeros(1,size(R,2)),R)';
    o trace = [trace; temp' ];
    o Update x0, y0, vx0, vy0 to values in temp;
• end
```

***F-4:** For a **constant acceleration** model:

State vector: Xt = [x y vx vy ax ay]

Process matrix: A = [1 0 del_t 0 0.5*del_t^2 0; 0 1 0 del_t 0 0.5*del_t^2;...

0 0 1 0 del_t 0; 0 0 0 1 0 del_t; 0 0 0 0 1 0; 0 0 0 0 0 1];

The **CS5320_const_acc** function gives a constant acceleration forward simulation.

Algorithm:

```
• trace = [x0, y0, vx0, vy0, ax0, ay0]
• for all time steps = del_t:max_time
    o A = [1 0 del_t 0 0.5*del_t^2 0; 0 1 0 del_t 0 0.5*del_t^2;...
           0 0 1 0 del_t 0; 0 0 0 1 0 del_t; 0 0 0 0 1 0; 0 0 0 0 0 1];
    o temp = A*[x0; y0; vx0; vy0; ax0; ay0] + mvnrnd(zeros(1,size(R,2)),R)';
    o trace = [trace; temp' ];
    o Update x0, y0, vx0, vy0, ax0, ay0 to values in temp;
• end
```

***F-5:** The **CS5320_process_step** function represents one step in linear process in the Kalman Filter

Input: p_im1 (nx1 vector): state vector at (i-1)_th step

D (nxn array): linear process matrix

R (nxn array): covariance matrix for process

Output: p_i (nx1 vector): state vector at i_th step

Algorithm:

```
• p_i = D*p_im1 + mvnrnd(zeros(1,size(R,2)),R)';
```

***F-6:** Similarly, the **CS5320_observe** represents one step linear observation

Input: x (nx1 vector): state vector

M (kxn array): observation matrix

Q (kxk array): observation covariance

Output: y (kx1 vector): observation (sensor) vector

Algorithm:

```
• y = M*x + mvnrnd(zeros(1,size(Q,2)),Q)';
```

Now, as we know about the model and the process and control steps, we can go ahead to look how one step in the Kalman Filter takes place.

***F-7:** This is represented by the CS5320_Kalman_step function

% On input:

% x_im1 (nx1 vector): state vector at step i-1

% Sigma_im1 (nxn array): state covariance array

% D (nxn array): linear process matrix

% R (nxn array): process covariance matrix

% M (kxn arraay): linear observation matrix

% Q (kxk array): observation covariance array

% y (kx1 vector): observation vector

% On output:

% x_i_plus (nx1 vector): updated state vector

% Sigma_i_plus (nxn array): state covariance matrix

The <u>Algorithm</u> to this function is the one discussed above (figure on page 4)

We can now simulate Kalman for constant acceleration scenario. This can be done by using

***F-8: CS5320_const_acc_Kalman** function

% On input:

% x0 (float): initial x locaiton

% y0 (float): initial y location

% vx0 (float): initial x speed

% vy0 (float): initial y speed

% ax0 (float): x acceleration

% ay0 (float): y acceleration

% del_t (float): time step

% max_t (float): max time for simulation

% R (6x6 array): process covariance matrix

% Q (2x2 array): observation covariance matrix

% On output:

% ta (px6 array): actual state values for p steps

```
%     ty (px2 array): observation values for p steps

%     te (px6 array): estimated state values for p steps
```

Algorithm:

```
• First, find the actual states by using acceleration simulation:
  ta = CS5320_const_acc(x0,y0,vx0,vy0,ax0,ay0,del_t,max_t,R);
• Now find sensor states, ty

  For all time_steps : del_t:max_t

    o temp = M*ta(index,:)' + mvnrnd(zeros(1,size(Q,2)),Q)';
    o ty = [ty;temp'];
• end
• Now, estimate states
• te = [x0, y0, vx0, vy0, ax0, ay0];
• x_im1 = [x0;y0;vx0;vy0;ax0;ay0];
• Sigma_im1 = zeros(6);
• index = 1;
• for t = del_t:del_t:max_t
    o   index = index + 1;
    o   D = [1 0 del_t 0 0.5*del_t^2 0; 0 1 0 del_t 0 0.5*del_t^2;...
               0 1 0 del_t 0; 0 0 0 1 0 del_t; 0 0 0 0 1 0; 0 0 0 0 0 1];
    o   y = ty(index,:)';
    o   %Now perform the Kalman step here
    o   [x_i_plus,Sigma_i_plus] = CS5320_Kalman_step(x_im1,Sigma_im1,D,R,M,Q,y);
    o   te = [te; x_i_plus'];
    o   x_im1 = x_i_plus;
    o   Sigma_im1 = Sigma_i_plus;
• end
```

As we need to detect a falling ball, which follows the constant acceleration model, we can need to use the **CS5320_red_ball_Kalman** function. However, now there is no sensor input, ty. But there are images and we need to detect the position of the ball in the image first, before estimating the states using Kalman.

***F-9:** For this, we need the **CS5320_detect_red_ball** which finds the red ball in an image.

```
% On input: im (mxnx3 array): rgb image

%              model (1x3 vector): rgb model

% On output: row (int): row of red ball centroid

%               col (int): col of red ball centroid
```

Algorithm:

- for all pixels
    - obsVect = double([im(i,j,1) im(i,j,2) im(i,j,3)]);
    - dist(i,j) = CS5320_Euclidean_distance(obsVect, model);end
- [row,col] = find(dist==min(min(dist)));

The distance function can be changed. Here it is the Euclidean_distance

CS5320_Euclidean_distance

```
% On input:
%     im (1x3 array): rgb Vector
%     model (1x3 vector): rgb model
% On output:
%     d = distance;
```

Code:

- d = norm(obsVect - model);

CS5320_Mahalanobis_distance

```
% On input:
%     obsVect (1x3 array): rgb Vector
%     meanRGB (1x3 vector): mean of the sampled pixels in the ball
%     covarRGB (1x3 vector): covariance matrix
%       call [meanRGB, covarRGB, RGBstore] = mean_covar_of_pixels()
%       before this function
% On output:
%     d = distance;
```

Code:

- d = (obsVect - meanRGB)*inv((covarRGB))*(obsVect - meanRGB)';

***F-10: CS5320_red_ball_Kalman**

% On input:

%   im_seg (struct vector): image sequence of falling ball (p

%   images)

%   ax (float): acceleration in x

```
%    ay (float): acceleration in y
%    del_t (float): time step
%    R (6x6 array): process covariance matrix
%    Q (2x2 array): observation covariance matrix
% On output:
%    ty (px6 array): observation values for p steps
%    te (px6 array): estimated state values for p steps
```

Algorithm:

```
• For all images in sequence
    o [r,c] = CS5320_detect_red_ball(im_seq(i).cdata,[250,0,0]);
    o   x=c;
    o   y =rows -r+1;
    o   ty = [ty;[x,y]];
• End
• %pixel to meters conversion: each pixel is 1/6 m in height
    ty  = ty/6;

• Initialize state , Sigma, M, te
• for index = 2:70
  o     D = [1 0 del_t 0 0.5*del_t^2 0; 0 1 0 del_t 0 0.5*del_t^2;...
            0 0 1 0 del_t 0; 0 0 0 1 0 del_t; 0 0 0 0 1 0; 0 0 0 0 0 1];
    o   y = ty(index,:)';
    o   [x_i_plus,Sigma_i_plus] = CS5320_Kalman_step(x_im1,Sigma_im1,D,R,M,Q,y);
    o   te = [te; x_i_plus'];
    o   x_im1 = x_i_plus;
    o   Sigma_im1 = Sigma_i_plus;
• end
```

It is noticed that the final position in the Kalman estimation is trailing the detected (sensor) value. This is happening due to improper value of gravity constant for the model. To take care of this case, we have to produce an estimate of the gravitational acceleration constant for the sequence.

The **CS5320_acceleration_estimation** function has been developed to answer this query

Algorithm:

```
• ay1 = -5;
• ay2 = -20;
• While error > 1e-7 or error < - 1e-7
```

- ay=(ay1+ay2)/2;
    - [ty,te] = CS5320_red_ball_Kalman(Falling_Ball, 0, ay, 1/30, R, Q)
    - error_in_ty_te = ty_rc(70,2)-te_rc(70,2);
    - if error_in_ty_te>0
        - ay1 = ay;
    - else
        - ay2=ay;
    - End
- Show error_in_ty_te

• Verification

Codes used for Verification can be found in the function: ‘test_all_functions’.

1. CS5320_background_sub_tracking

To verify this function, use the following commands:

```
%Veifying CS5320_background_sub_tracking
clear all
load Bead_data% All 10 images of Bead motion  are stored in Bead_data
t_im = CS5320_background_sub_tracking(Bead);
imagesc(t_im(:,:,1)); colormap gray;
title 'Background Subtraction';
imagesc(t_im(:,:,4)); colormap gray;
title 'Background Subtraction'
```

When plotted, the images should show the background subtracted (black) with the object in white.

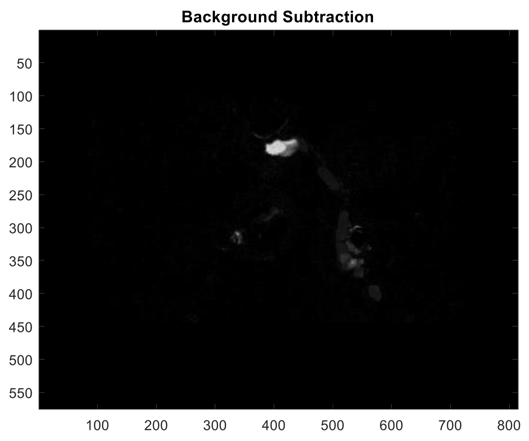


Fig: imagesc(t_im(:,:,1))

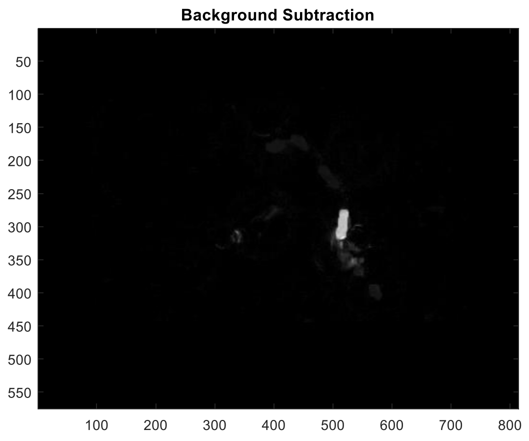


Fig: imagesc(t_im(:,:,4))

2. CS5320_bead_tracking

To verify this function, use t_im(Subtracted background images) as input. Plot the track on the top of an image and check if the object follows the same trace.

```
% Verifying bead_tracking
track = CS5320_bead_tracking(t_im);
close all;imagesc(Bead(1).cdata);
hold on;
plot(track(:,2),track(:,1),'ro-');
```

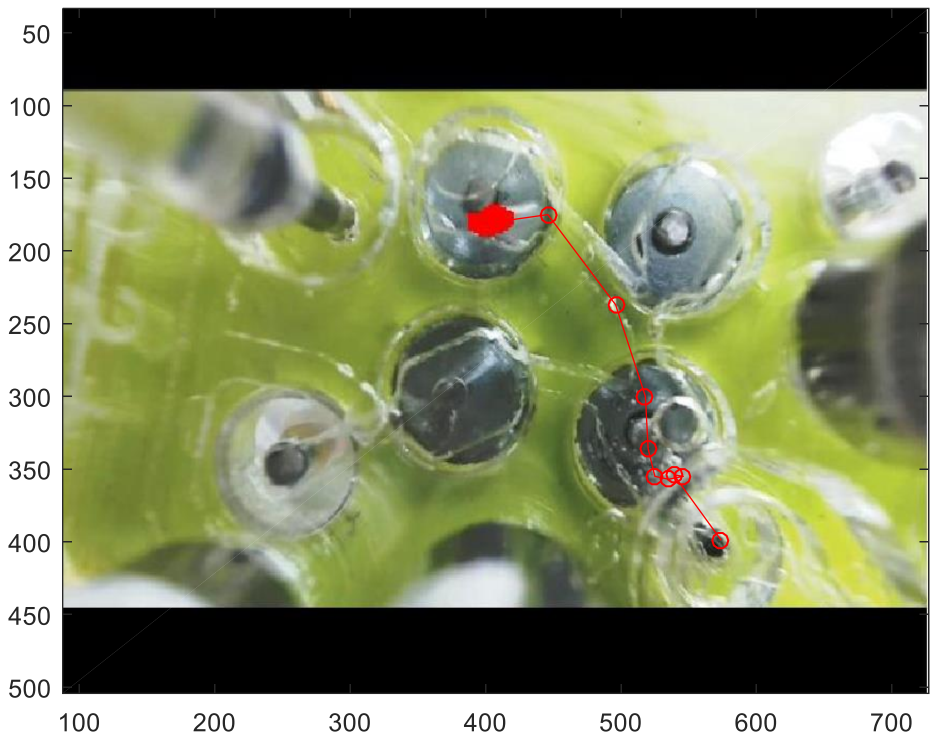


Fig: Bead Tracking

3. CS5320_const_vel

To verify, give velocity of 1 in both x and y direction, therefore trajectory should be at 45 degrees with constant difference between sequential positions.

```
%Verifying constant velocity with R = zeros(4)
R = zeros(4);
R(3:4,3:4)=0;
tr = CS5320_const_vel(0,0,1,1,0.1,2,R);
close all;plot(tr(:,1),tr(:,2),'r*');
```

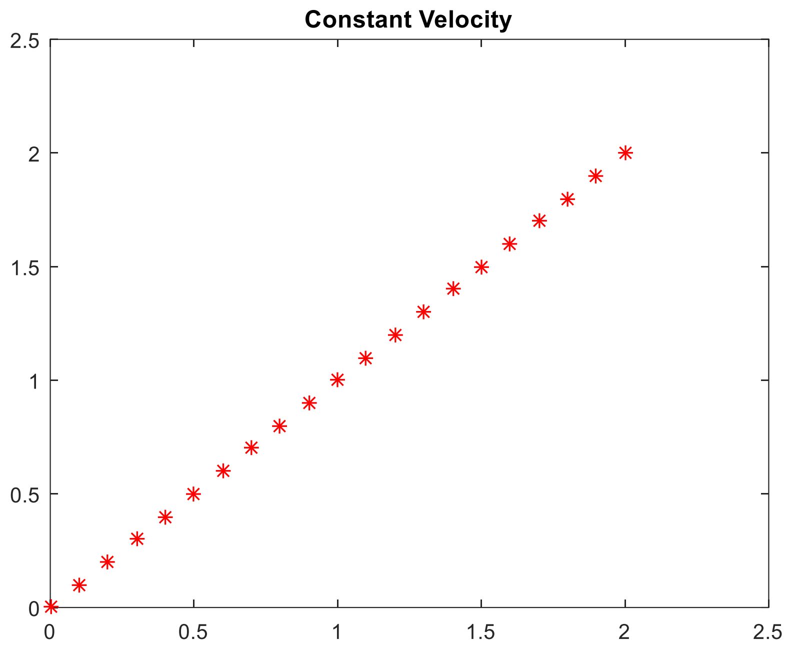


Fig: plot(tr(:,1),tr(:,2),'r'); for R=0*

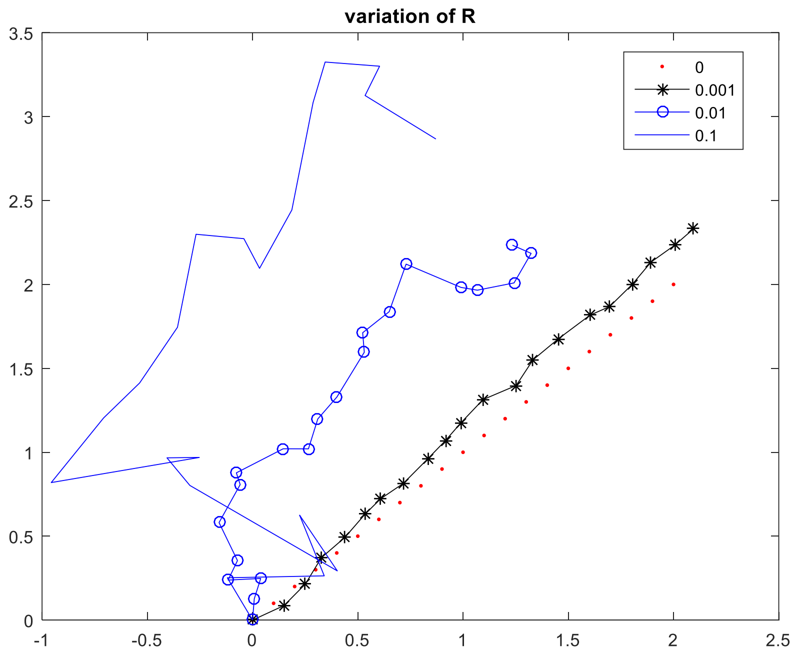


Fig: Variation of R (shown in legend)

4. CS5320_const_acc

To verify this, give a constant acceleration of 9.8 in both x and y. Along with the trajectory being at 45 degrees, the difference in sequential positions should increase.

```
%Verifying constant acceleration with 3 different R
R = 0.0000*eye(6);
R(5:6,5:6) = 0;
tr = CS5320_const_acc(0,0,0,0,9.8,9.8,0.1,2,R);
close all;plot(tr(:,1),tr(:,2),'r*');
```

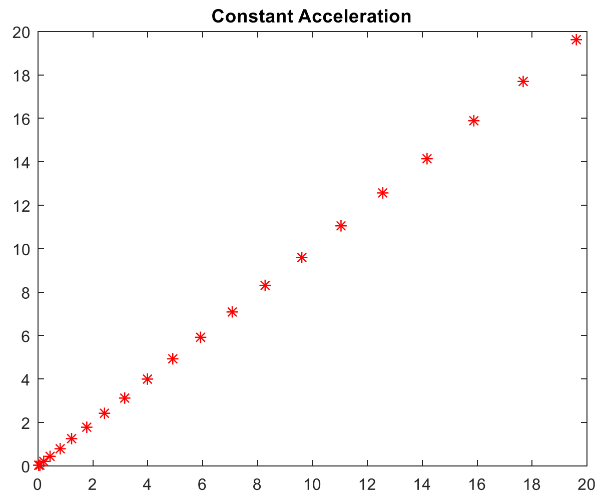
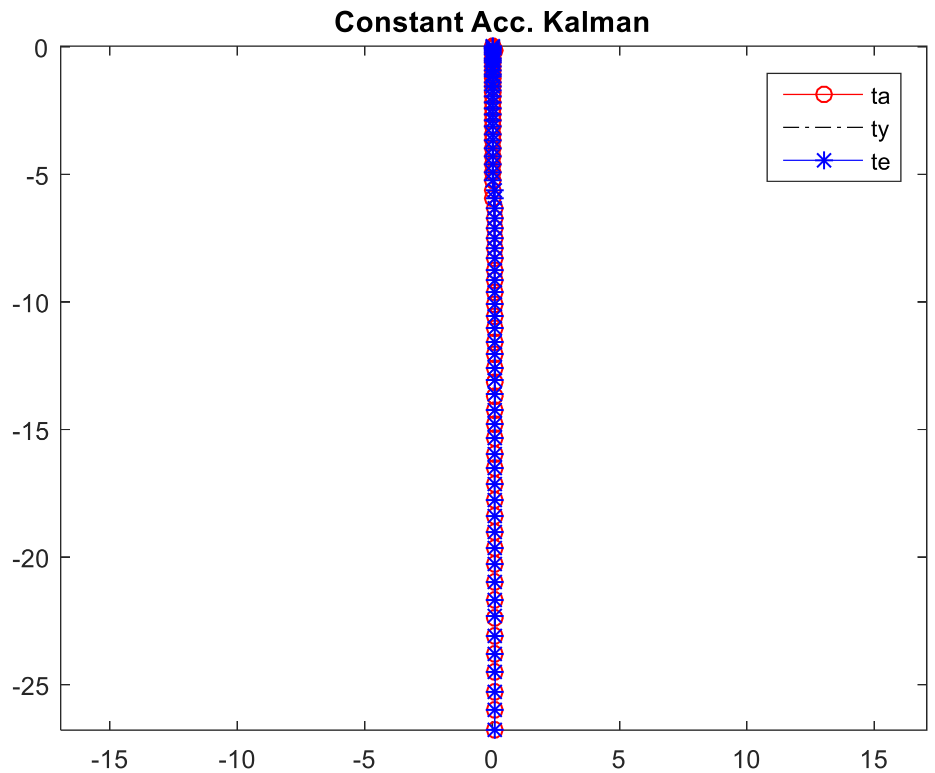


Fig: Constant Acceleration

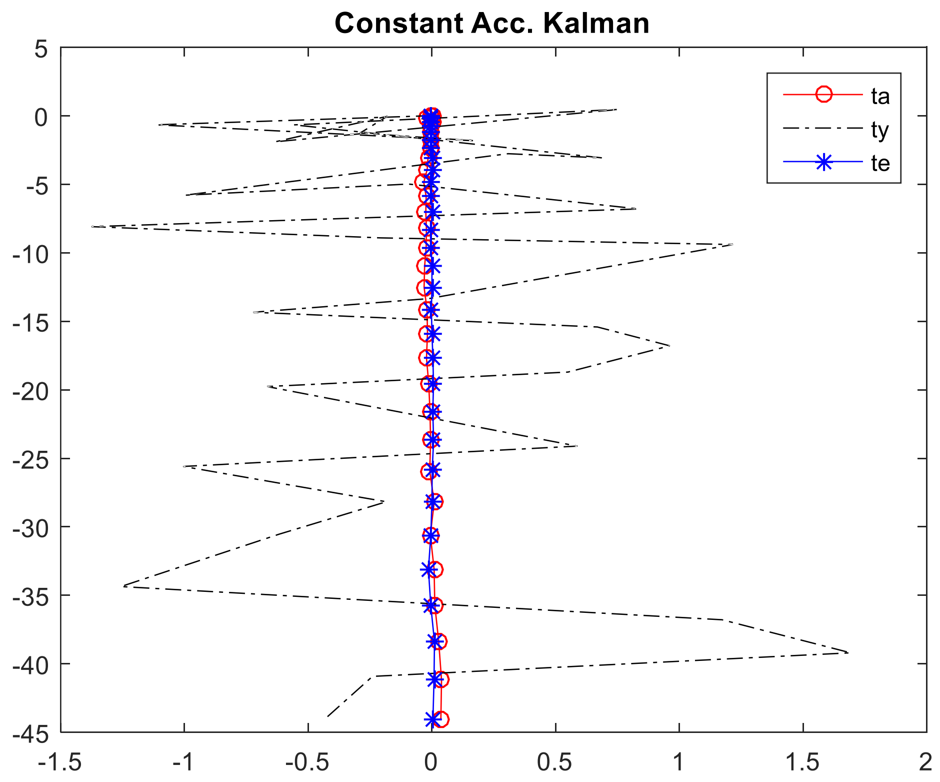
5. CS5320_const_acc_Kalman

This function can be verified by changing the values of R and Q. It is to be seen that with R and Q, the state estimate is very good. At high process noise (Q=1), the estimate is comparably good. At high process noise, R=0.1, the estimate is very bad.

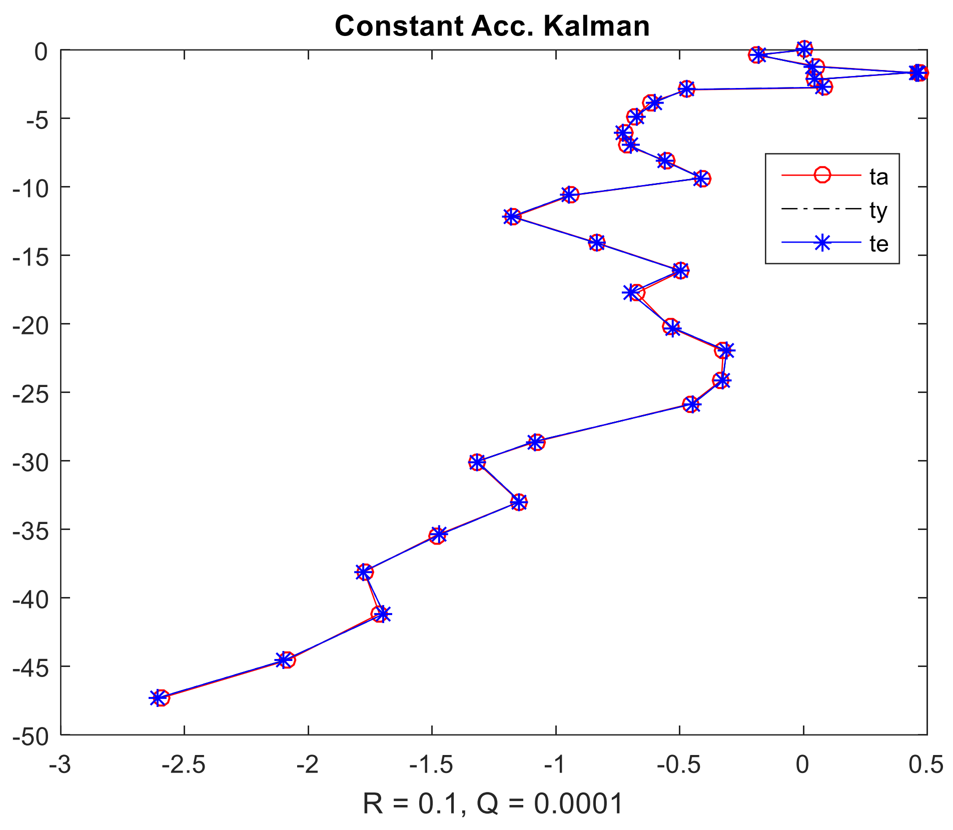
```
%Verifying CS5320_const_acc_Kalman
clear all;
R = 0.0001*eye(6);
R(5:6,5:6) = 0;
Q = 0.0001*eye(2);
[ta,ty,te] = CS5320_const_acc_Kalman(0,0,0,0,0,-9.8,1/30,2.34,R,Q);
close all;plot(ta(:,1),ta(:,2),'r-o');
hold on;plot(ty(:,1),ty(:,2),'k-.');
hold on;plot(te(:,1),te(:,2),'b-*');
```



R = 0.0001, Q = 0.0001



R = 0.0001, Q = 1



6. CS5320_detect_red_ball

In order to verify this function, the function was run multiple times in a loop (inside the CS5320_red_ball_Kalman function), the input was a sequence of images of the red ball. ty stored the rows and columns found in each iteration.

Code in CS5320_red_ball_Kalman

```
for i =1:numIm
    [r,c] = CS5320_detect_red_ball(im_seq(i).cdata,[250,0,0]);%[230,30,30]
    x=c;
    y =rows -r+1;
    ty = [ty;[x,y]];
end
```

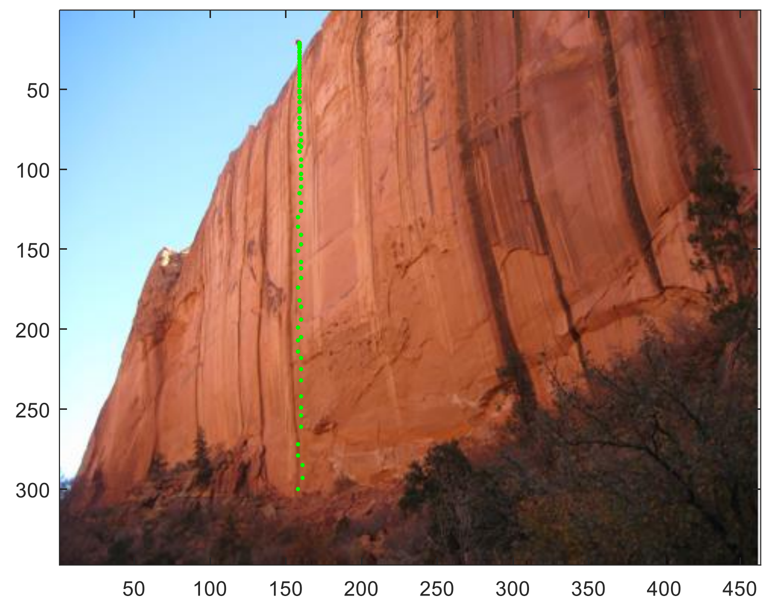


Fig: detected red ball

7. CS5320_red_ball_Kalman

To verify the function, it was run using the code below, the outputs, ty and te were plotted onto the original image. When compared to the image sequence, it was seen that the ball was detected at the correct positions (ty) and the Kalman estimation was good (te).

```
function test_red_ball_Kalman

load A10_data;
R = 0.0001*eye(6,6);
R(5:6,5:6) = 0;
Q = eye(2,2);
[ty,te] = CS5320_red_ball_Kalman(Falling_Ball,0,-17.64575,1/30,R,Q);% use g = -17??
%te is in meters: multiply by 6
te_met=te;
te_pix=6*te;
%now convert from xy -> rows and columns
te_rc = zeros(size(te,1),2);
te_rc(:,1) = te_pix(:,1);
te_rc(:,2) = 347*ones(size(te,1),1) - te_pix(:,2) + ones(size(te,1),1);
% similarly sensor
ty_met=ty;
ty_pix=6*ty;
%now convert from xy -> rows and columns
ty_rc = zeros(size(ty,1),2);
ty_rc(:,1) = ty_pix(:,1);
ty_rc(:,2) = 347*ones(size(ty,1),1) - ty_pix(:,2) + ones(size(ty,1),1);
%plot image with the ball motion trace
close all;
imshow(Falling_Ball(1).cdata);
hold on;plot(te_rc(:,1),te_rc(:,2),'b');
hold on;plot(ty_rc(:,1),ty_rc(:,2),'g.');
%below line to adjust the final position of ball te==ty, by adjusting ay0
```

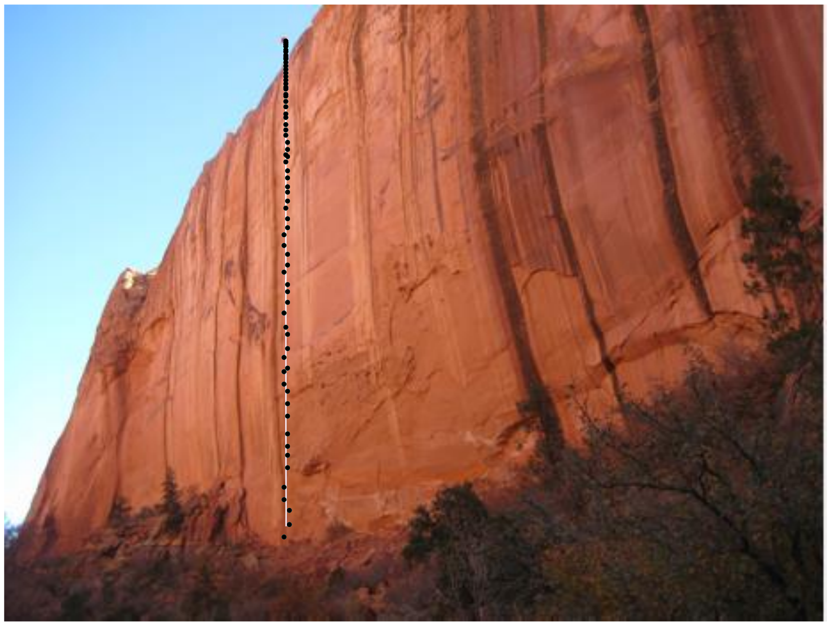


Fig: verifying CS5320_red_ball_Kalman

8. CS5320_acceleration_estimation

To verify the Function and also to produce an estimate of the gravitational acceleration constant for the sequence, the function was run. The acceleration was found out to be -17.5391

```
function [te_rc,ty_rc] = CS5320_acceleration_estimation()

load A10_data;

ay1 = -5;
ay2 = -20;
error_in_ty_te = 5;
R = 0.0001*eye(6,6);
R(5:6,5:6) = 0;
Q = eye(2,2);

while (1)
    ay=(ay1+ay2)/2;

    [ty,te] = CS5320_red_ball_Kalman( Falling_Ball, 0, ay, 1/30, R, Q);% use ay = -
17.64575
    %te is in meters: multiply by 6
    te_pix=6*te;
    %now convert from xy -> rows and columns
    te_rc = zeros(size(te,1),2);
    te_rc(:,1) = te_pix(:,1);
    te_rc(:,2) = 347*ones(size(te,1),1) - te_pix(:,2) + ones(size(te,1),1);
    % similarly sensor
    ty_pix=6*ty;
    %now convert from xy -> rows and columns
    ty_rc = zeros(size(ty,1),2);
    ty_rc(:,1) = ty_pix(:,1);
    ty_rc(:,2) = 347*ones(size(ty,1),1) - ty_pix(:,2) + ones(size(ty,1),1);

    error_in_ty_te = ty_rc(70,2)-te_rc(70,2);
    if error_in_ty_te>0
        ay1 = ay;
    else
        ay2=ay;
    end
    if abs(error_in_ty_te) < 1e-7
        break;
    end
end

error_in_ty_te
close all;
imshow(Falling_Ball(1).cdata);
hold on;plot(te_rc(:,1),te_rc(:,2),'b');
hold on;plot(ty_rc(:,1),ty_rc(:,2),'r.');
```

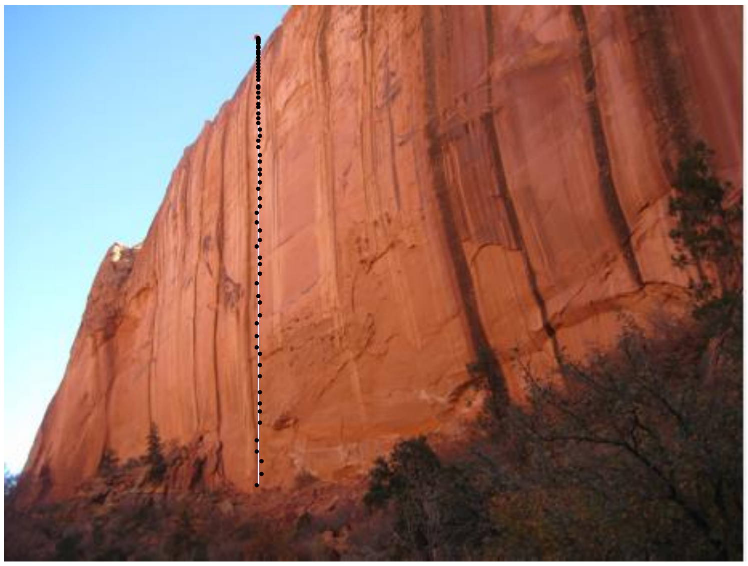


Fig: Better estimation of the states, via a good estimate of gravitational constant

- **Data**

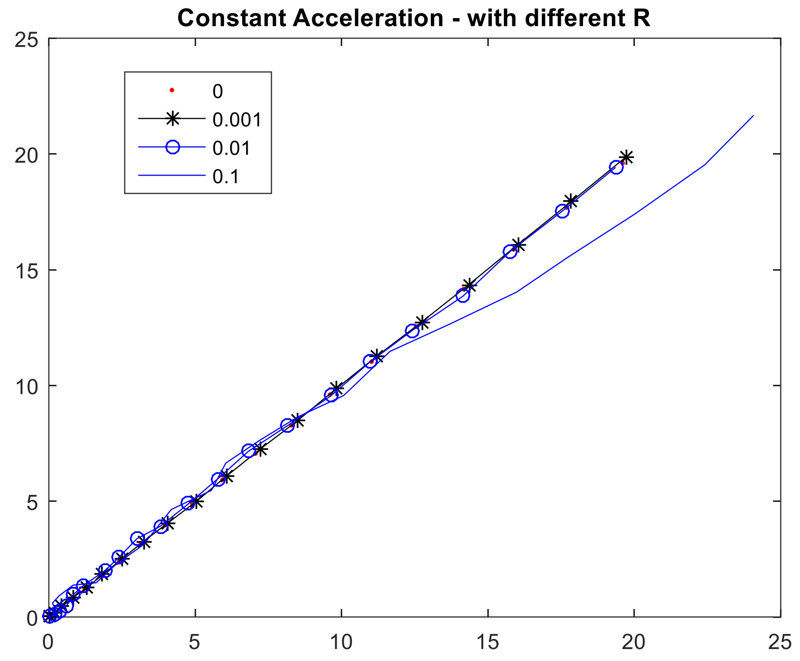
It can be seen how the Process noise and the sensor noise affect the state estimation. For this, first the Process noise can be held constant with Rfactor==0.0001 and the Gfactor can be varied from 0 to 1 by increments of 0.01. And then the final position of ty and te can be checked for error.

Example: Rfacror means : Rfactor*eye(4), Similarly Gfactor

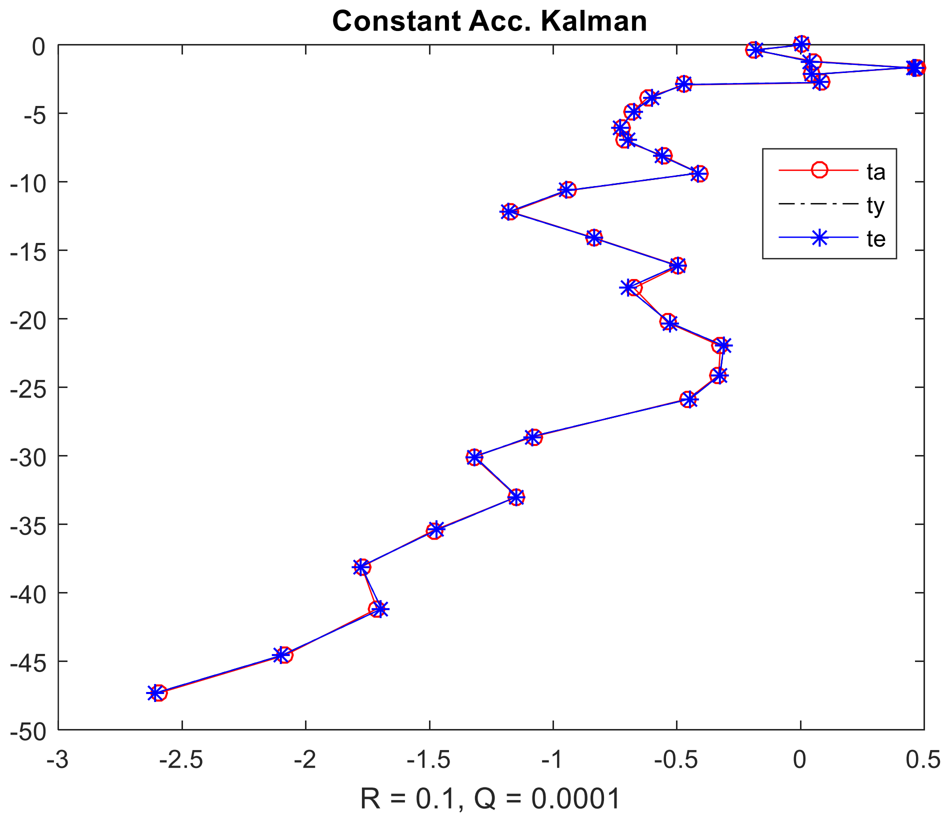
Rfactor		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Gfactor		0.00	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00
te	row	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14
	col	299.94	299.94	299.94	299.94	299.94	299.94	299.94	299.94	299.94	299.94	299.94
ty	row	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00
	col	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14	159.14
error		0.00	-0.99	-0.73	-0.53	-0.38	-0.27	-0.17	-0.10	-0.04	0.02	0.06

Rfactor		0.00	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00
Gfactor		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
te	row	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00
	col	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00
ty	row	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00	300.00
	col	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00	158.00
error		0.65	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

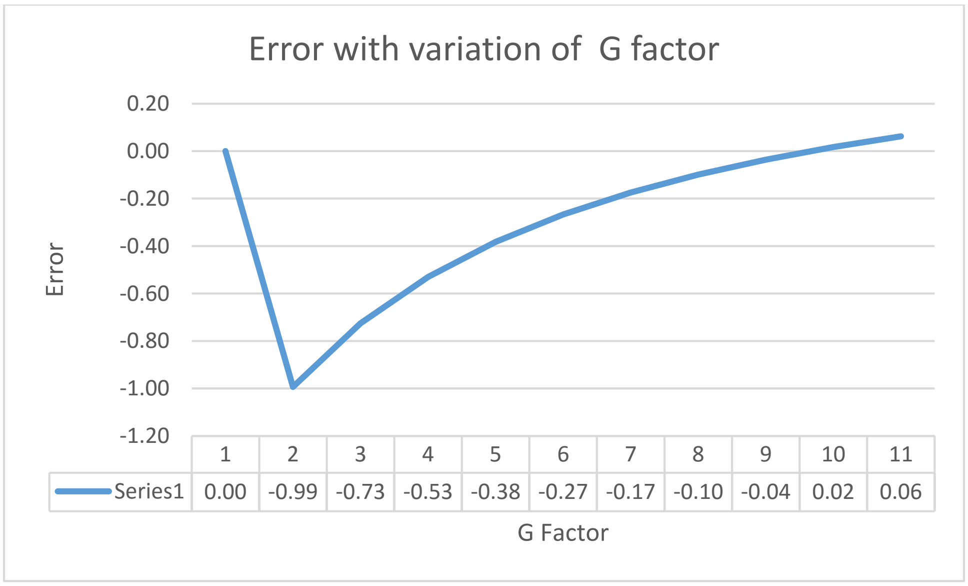
- **Analysis**



Constant Acceleration - with different R
0
0.001
0.01
0.1



Constant Acc. Kalman
ta
ty
te
R = 0.1, Q = 0.0001



	1	2	3	4	5	6	7	8	9	10	11
Series1	0.00	-0.99	-0.73	-0.53	-0.38	-0.27	-0.17	-0.10	-0.04	0.02	0.06

- **Interpretation**

With increase in Sensor noise, the estimation is not affected much but with the Process noise, the estimation is affected.

- **Critique**

Although I spent the most time for this assignment, I did not understand how to go ahead with many things, such as how the threshold is to be used as a measure. Because, I used the minimum distance to obtain the required point on the ball.

I tried to compute the Gaussian probability distance with the formula

$$d = 255 * (1 - e^{-(\frac{x-m}{\sigma^2})})$$

But there were dimension errors, as x and m(mean) were 1x3 vectors and σ was 3x3 covariance.

- **Log**

Coding/debugging: 25 hrs

Report: 5 hrs