Kalman Filter 1D Implementation

1D Ball Falling Example

Ball being released at the height of 4000m. With constant acceleration -9.8m/s. The position of the ball is recorded every second from 0 to the 10th second.

Based on simple kinematic equation:

$$\mathbf{d = v_i * t + \frac{1}{2} * a * t^2}, \text{ where}$$

d = distance;

$\mathbf{v_i}$ = initial velocity;

t = time elapsed;

a = acceleration;

Given $\mathbf{v_i}$ = 0 m/s, **a** = -9.8 m/s^2,

At **t** = 1s, **d** = -4.9m, **Ball position** = 4000m – 4.9m = **3995.1m**

At **t** = 2s, **d** = -19.6m, **Ball position** = 4000m – 19.6m = **3980.4m**

At **t** = 3s, **d** = -44.1m, **Ball position** = 4000m – 44.1m = **3955.9m**

At **t** = 4s, **d** = -78.4m, **Ball position** = 4000m – 78.4m = **3921.6m**

At **t** = 5s, **d** = -122.5m, **Ball position** = 4000m – 122.5m = **3877.5m**

At **t** = 6s, **d** = -176.4m, **Ball position** = 4000m – 176.4m = **3823.6m**

At **t** = 7s, **d** = -240.1m, **Ball position** = 4000m – 240.1m = **3759.9m**

At **t** = 8s, **d** = -313.6m, **Ball position** = 4000m – 313.6m = **3686.4m**

At **t** = 9s, **d** = -396.9m, **Ball position** = 4000m – 396.9m = **3603.1m**

At **t** = 10s, **d** = -490.0m, **Ball position** = 4000m – 490.0m = **3510.0m**

In a perfect scenario where there are no external influences and noises, these are the expected results of the ball's position at each second.

Now assuming that we have a GPS sensor that can track the position of the ball at each second and that the GPS reading is not very accurate (let's assume it has a standard deviation of 20). The measurement shown by the GPS could be:

At t = 1s, 3975.1m (-20m)

At t = 2s, 4000.4m (+20m)

At t = 3s, 3935.9m (-20m)

At t = 4s, 3941.6m (+20m)

At t = 5s, 3857.5m (-20m)

At t = 6s, 3843.6m (+20m)

At t = 7s, 3739.9m (-20m)

At t = 8s, 3706.4m (+20m)

At t = 9s, 3583.1m (-20m)

At t = 10s, 3530.0m (+20m)

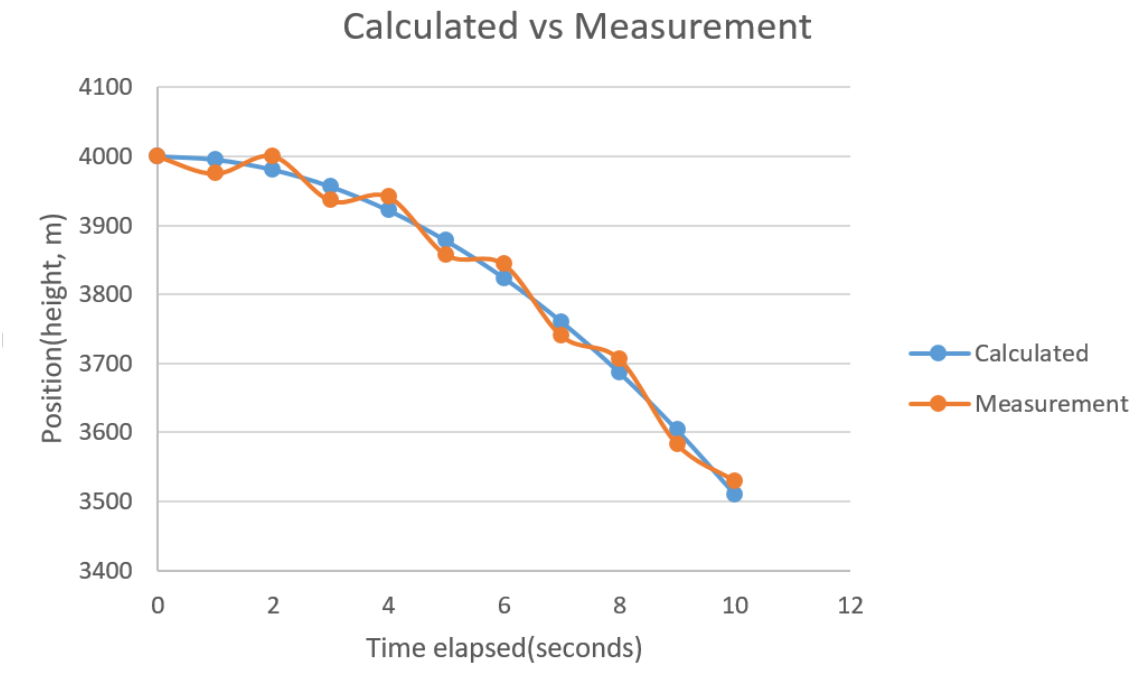

Figure 1: Graph of calculated results vs measurement results

Equations

1. Predict current state based on previous state.

$$\mathbf{X = A*X_{k-1} + B*u;}$$

2. Predict current process covariance matrix based on previous one.

$$\mathbf{p = A*P*A^T + Q;}$$

3. Calculate the Kalman gain.

$$\mathbf{K = p*\ C^T\ (C*P*\ C^T + R)^{-1};}$$

4. Update state with Kalman gain.

$$\mathbf{X_k = X + K(z-C*X);}$$

5. Update process covariance matrix.

$$\mathbf{P_{Xk} = (I-KC)p;}$$

X = predicted state;

$\mathbf{X_{k-1}}$ = previous state;

A, **B** & **C** = transition matrix;

$\mathbf{A^T}$ = transpose of matrix A;

$\mathbf{C^T}$ = transpose of matrix C;

u = control unit action (e.g. acceleration);

p = predicted process covariance matrix;

P = previous process covariance matrix;

Q = process covariance matrix that does not get updated;

K = Kalman gain;

R = measurement error variance;

$\mathbf{X_k}$ = updated state;

z = measurement from sensor(s);

$\mathbf{P_{Xk}}$ = updated process covariance matrix;

I = identity matrices;

Calculations

We will use one iteration as example, where:

t = 1s,

Previous position, x = 4000m,

Previous velocity, x_v = 0m/s

Acceleration = -9.8m/s^2

R = 20^2 (because our measurement has a standard deviation of 20)

z= 3975.1m

P/Q = $\begin{bmatrix} \text{variance} & 0 \\ 0 & \text{variance} \end{bmatrix}$

For this calculation we will just plug in some numbers for our P/Q in order to yield some visible results from our Kalman Filter, so we will just assume that:

P/Q = $\begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix}$

1. Predict current state based on previous state.

$$\mathbf{X = A*X_{k-1} + B*u;}$$

$$X = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ x_v \end{bmatrix} + \begin{bmatrix} 1/2\,\Delta T^2 \\ \Delta T \end{bmatrix} u$$

$$= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 4000 \\ 0 \end{bmatrix} + \begin{bmatrix} 1/2 \\ 1 \end{bmatrix} -9.8$$

$$= \begin{bmatrix} 3995.1 \\ -9.8 \end{bmatrix}$$

2. Predict current process covariance matrix based on previous one.

$$\mathbf{p = A*P*A^T + Q;}$$

$$p = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 22 & 4 \\ 4 & 8 \end{bmatrix}$$

3. Calculate the Kalman gain.

$$\mathbf{K = p^* C^T (C^*P^* C^T + R)^{-1};}$$

$$K = \frac{\begin{bmatrix} 22 & 4 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}}{\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 22 & 4 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 400}$$

$$= \begin{bmatrix} 0.0521 \\ 0.0095 \end{bmatrix}$$

4. Update state with Kalman gain.

$$\mathbf{X_k = X + K(z-C*X);}$$

$$X_k = \begin{bmatrix} 3995.1 \\ -9.8 \end{bmatrix} + \begin{bmatrix} 0.0521 \\ 0.0095 \end{bmatrix} \left\langle 3975.1 - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 3995.1 \\ -9.8 \end{bmatrix} \right\rangle$$

$$= \begin{bmatrix} 3994.058 \\ -9.99 \end{bmatrix}$$

5. Update process covariance matrix.

$$\mathbf{P_{Xk} = (I-KC)p;}$$

$$P_{Xk} = \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0.0521 \\ 0.0095 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} \right\rangle \begin{bmatrix} 22 & 4 \\ 4 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} 20.85 & 3.79 \\ 3.79 & 7.96 \end{bmatrix}$$

	Position of object		
Time elapsed(seconds)	Predicted	Measurement	Updated
0	4000	4000	4000
1	3995.1	3975.1	3994.1
2	3979.2	4000.4	3981.3
3	3957.2	3935.9	3953.6
4	3918.6	3941.6	3924.2
5	3881	3857.5	3873.9
6	3819.1	3843.6	3827.4
7	3764.7	3739.9	3755.8
8	3681.4	3706.4	3690.6
9	3608.3	3583.1	3598.9
10	3504.9	3530	3514.3

Figure 2: Results of ball position with Prediction, Measurement and Kalman Filter's update

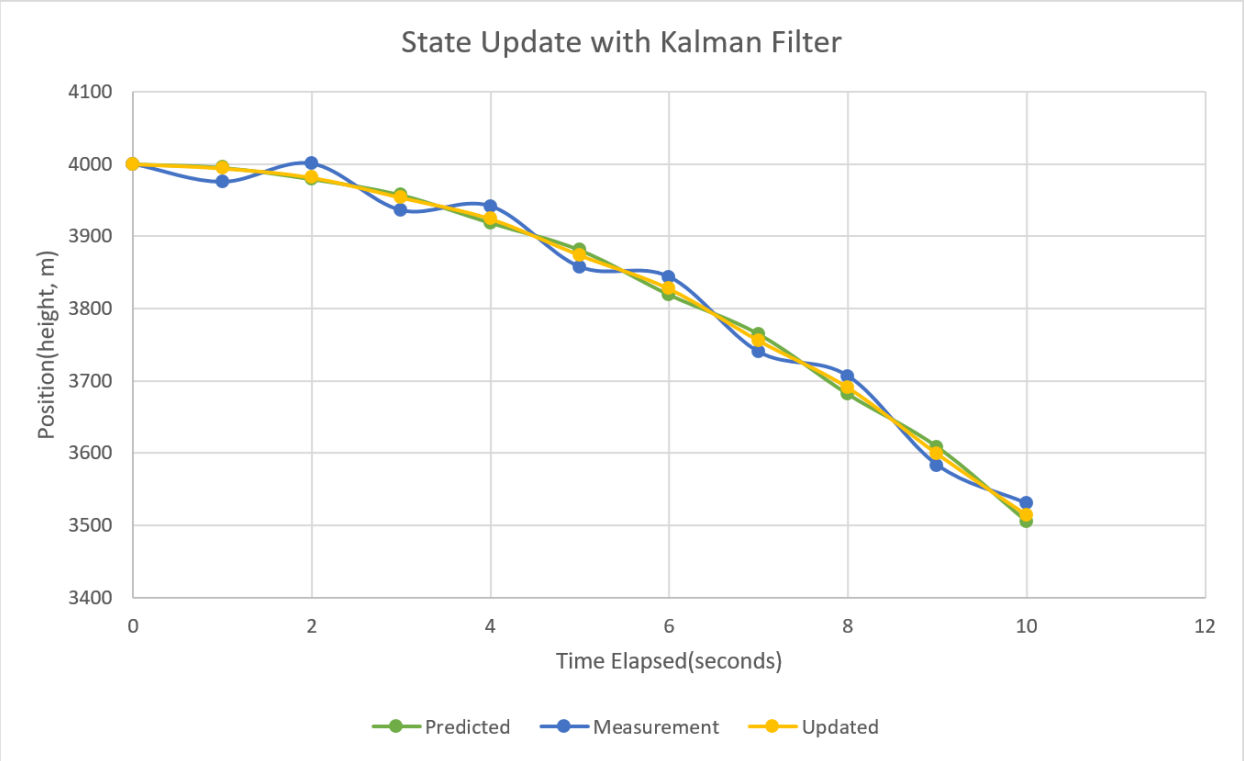

Figure 3: Graph plotted based on table in Figure 2

Implementation in C

```
float xLocErr=20;
float xLoc=4000, xVel=0;
float xLocMea;
float acc=(-9.8);
float x[2][1], p[2][2], P[2][2];
float upperK[2][1], lowerK1[1][2], lowerK2, inverseLowerK, K[2][1];
float y; //for updating the state

float prevP[2][2] = {
        {9,0},
        {0,4}
};

//needs to be fed
float prevX[2][1] = {
        {xLoc},
        {xVel}
};

float A[2][2] = {
        {1,TIME},
        {0,1}
};

float At[2][2] = {
        {1,0},
        {TIME,1}
};

float B[2][1] = {
        {(TIME*TIME)*0.5},
        {TIME}
};

float C[1][2] = {
        {1,0}
};


float Ct[2][1] = {
        {1},
        {0}
};

//constant
float Q[2][2] = {
        {9,0},
        {0,4}
};

//constant
float R = xLocErr*xLocErr;

float I[2][2] = {
        {1,0},
        {0,1}
};
```

```
int count = 1;

while(!feof(fp1)) {

    printf("At t = %ds:\n", count);
    fscanf(fp1, "%f", &xLocMea);

    float z = xLocMea;


    //1.Predict state
    x[0][0] = (A[0][0]*prevX[0][0])+(A[0][1]*prevX[1][0])+(B[0][0]*acc);
    x[1][0] = (A[1][0]*prevX[0][0])+(A[1][1]*prevX[1][0])+(B[1][0]*acc);

    printf("PREDICTED LOCATION: ");
    printf("%f\n", x[0][0]);

    //2.Predict process covariance matrix
    for(int i=0;i<=1;i++) {
        for(int j=0;j<=1;j++) {
            float sum = 0;
            for(int k=0;k<=1;k++) {
                sum = sum + A[i][k]*prevP[k][j];
            }
            p[i][j] = sum;
        }
    }

    for(int i=0;i<=1;i++) {
        for(int j=0;j<=1;j++) {
            float sum = 0;
            for(int k=0;k<=1;k++) {
                sum = sum + p[i][k]*At[k][j];
            }
            p[i][j] = sum + Q[i][j];
        }
    }


//3.Calculate Kalman Gain
for(int i=0;i<=1;i++) {
    for(int j=0;j<=0;j++) {
        float sum = 0;
        for(int k=0;k<=1;k++) {
            sum = sum + p[i][k]*Ct[k][j];
        }
        upperK[i][j] = sum;
    }
}

for(int i=0;i<=0;i++) {
    for(int j=0;j<=1;j++) {
        float sum = 0;
        for(int k=0;k<=1;k++) {
            sum = sum + C[i][k]*p[k][j];
        }
        lowerK1[i][j] = sum;
    }
}

lowerK2 = (lowerK1[0][0]*Ct[0][0])+(lowerK1[0][1]*Ct[1][0])+R;
inverseLowerK = 1/lowerK2;

K[0][0] = upperK[0][0]*inverseLowerK;
K[1][0] = upperK[1][0]*inverseLowerK;

//4.Update state
y = z-((C[0][0]*x[0][0])+(C[0][1]*x[1][0]));

prevX[0][0] = x[0][0]+(K[0][0]*y);
prevX[1][0] = x[1][0]+(K[1][0]*y);
```

```
    //5. Update process covariance matrix
    for(int i=0;i<=1;i++) {
        for(int j=0;j<=1;j++) {
            float sum = 0;
            for(int k=0;k<=0;k++) {
                sum = sum + K[i][k]*C[k][j];
            }
            P[i][j] = I[i][j]-sum;
        }
    }

    for(int i=0;i<=1;i++) {
        for(int j=0;j<=1;j++) {
            float sum = 0;
            for(int k=0;k<=1;k++) {
                sum = sum + P[i][k]*p[k][j];
            }
            prevP[i][j] = sum;
        }
    }

    printf("LOCATION MEASUREMENT: ");
    printf("%f\n", xLocMea);
    printf("UPDATED LOCATION: ");
    printf("%f\n",prevX[0][0]);
    printf("CURRENT STATE:\n");
    printf("%f\n",prevX[0][0]);
    printf("%f\n",prevX[1][0]);


    printf("PROCESS COVARIANCE MATRIX:\n");
    for(int i=0;i<=1;i++) {
        for(int j=0;j<=1;j++) {
            printf("%f, ",prevP[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    count++;
}
fclose(fp1);
```

Flowchart

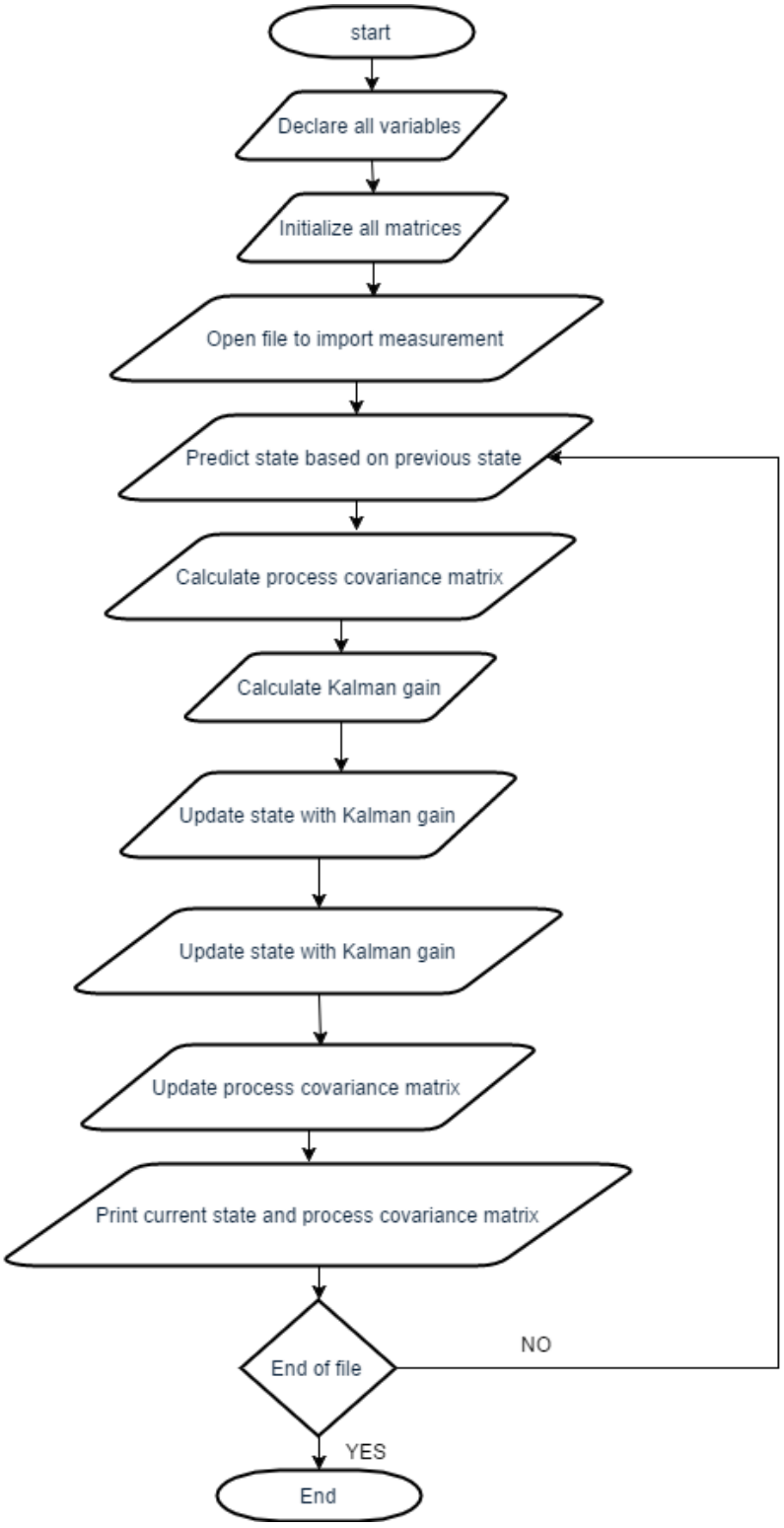

Figure 4: Flowchart of program

Fine Tuning the R

Realistically, we are unable to know what exactly the error(R) in the sensor is. In the previous example, we know that R has a standard deviation of 20 because we are the one who added the error into the sensor. In real life, most of the times we will need to fine tune the value of R. For this example, we will increase the R by the standard deviation of 5 and observe the changes.

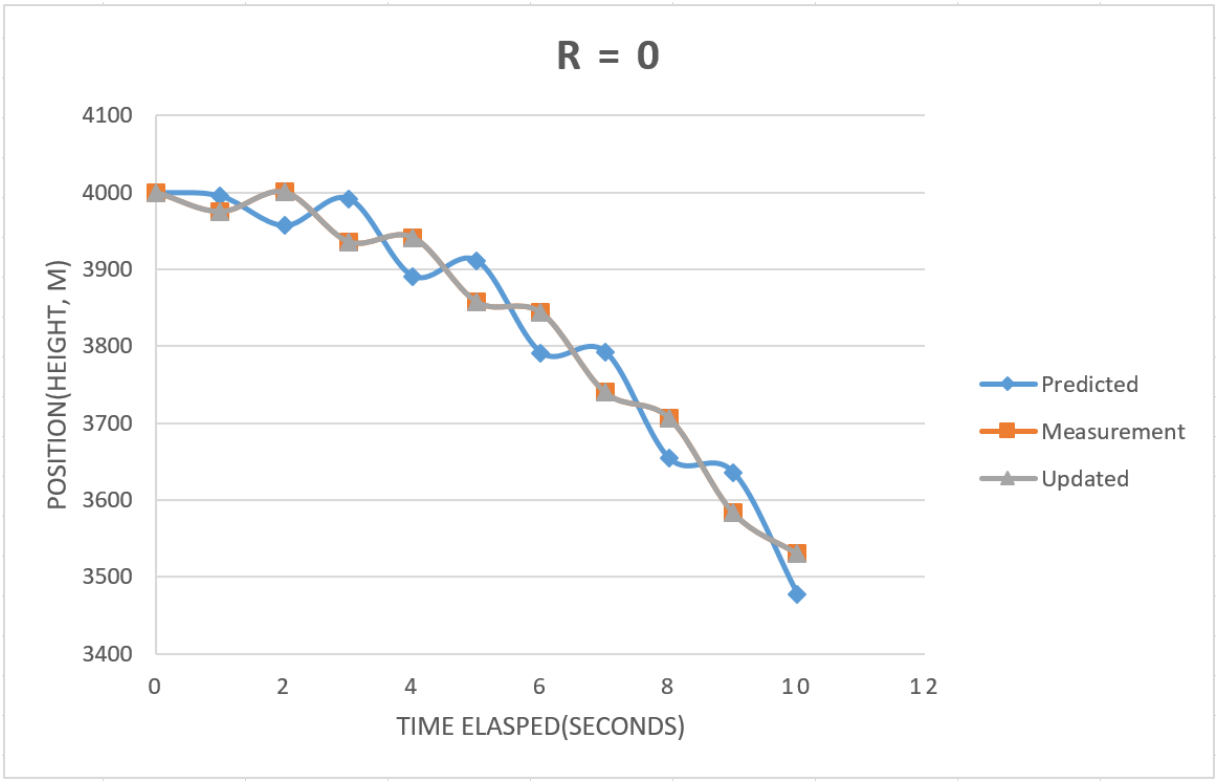

Figure 5: When R is 0, the updated positions tend to fall towards the measurements as they are considered perfect readings

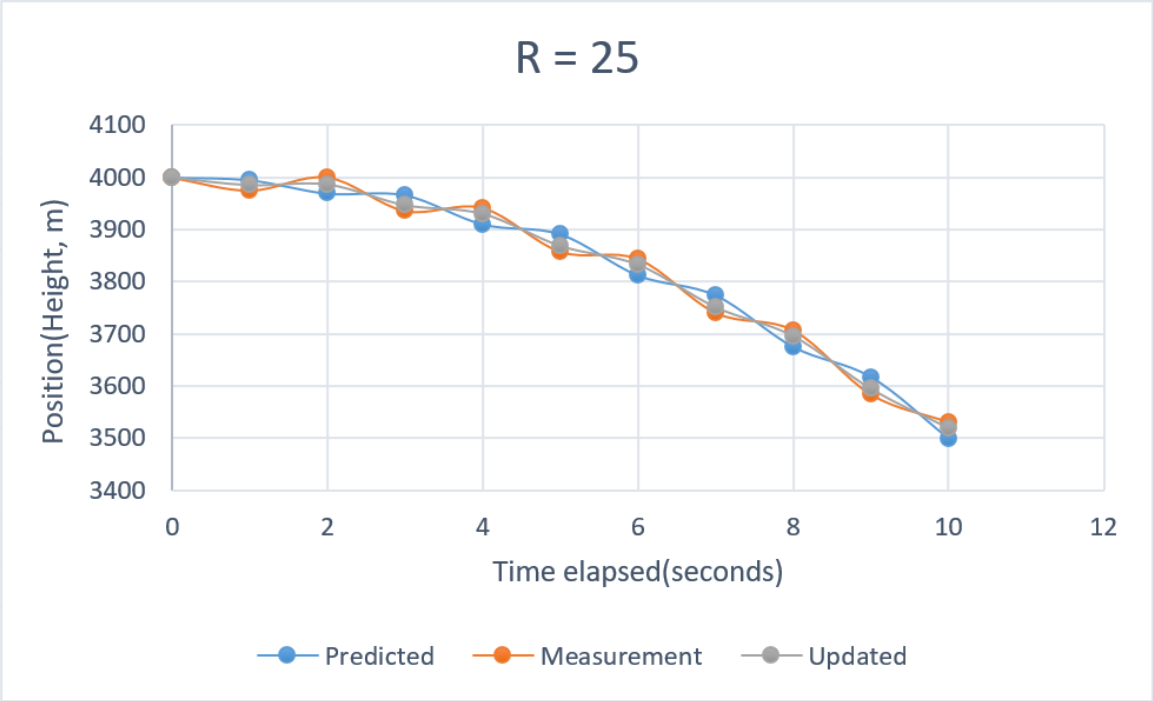

Figure 6: The updated positions fall a tiny bit away from the measurement

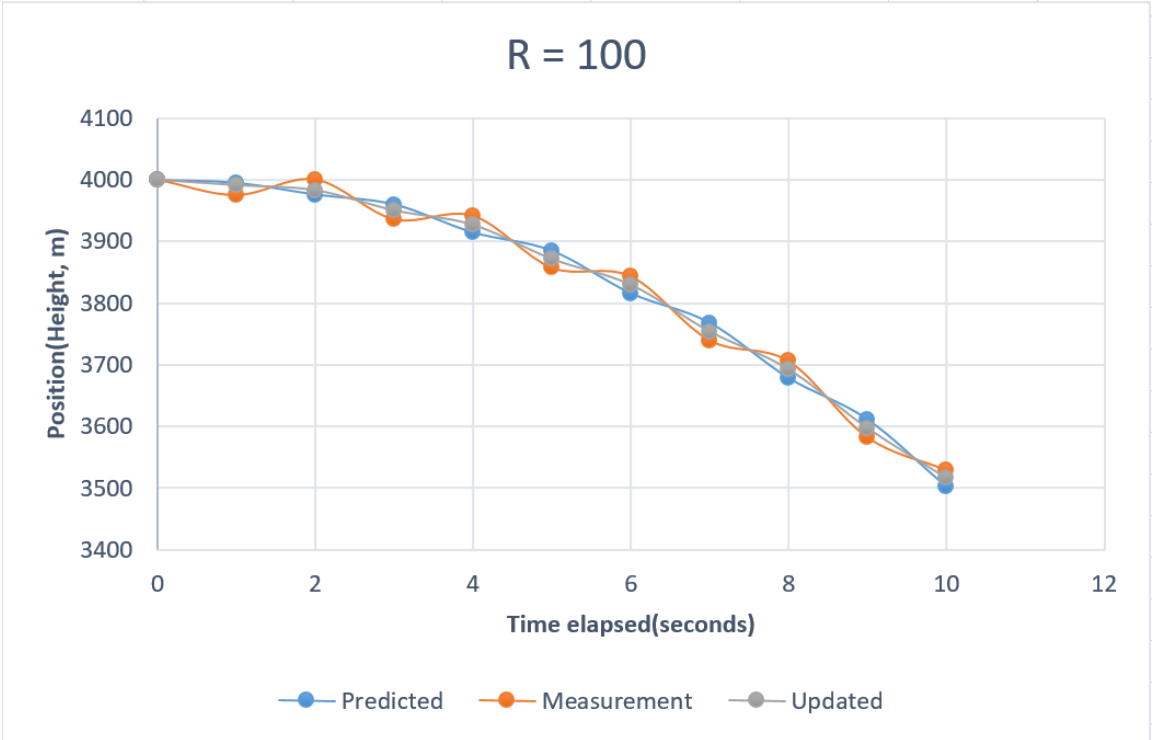

Figure 7: The updated positions move closer to the middle

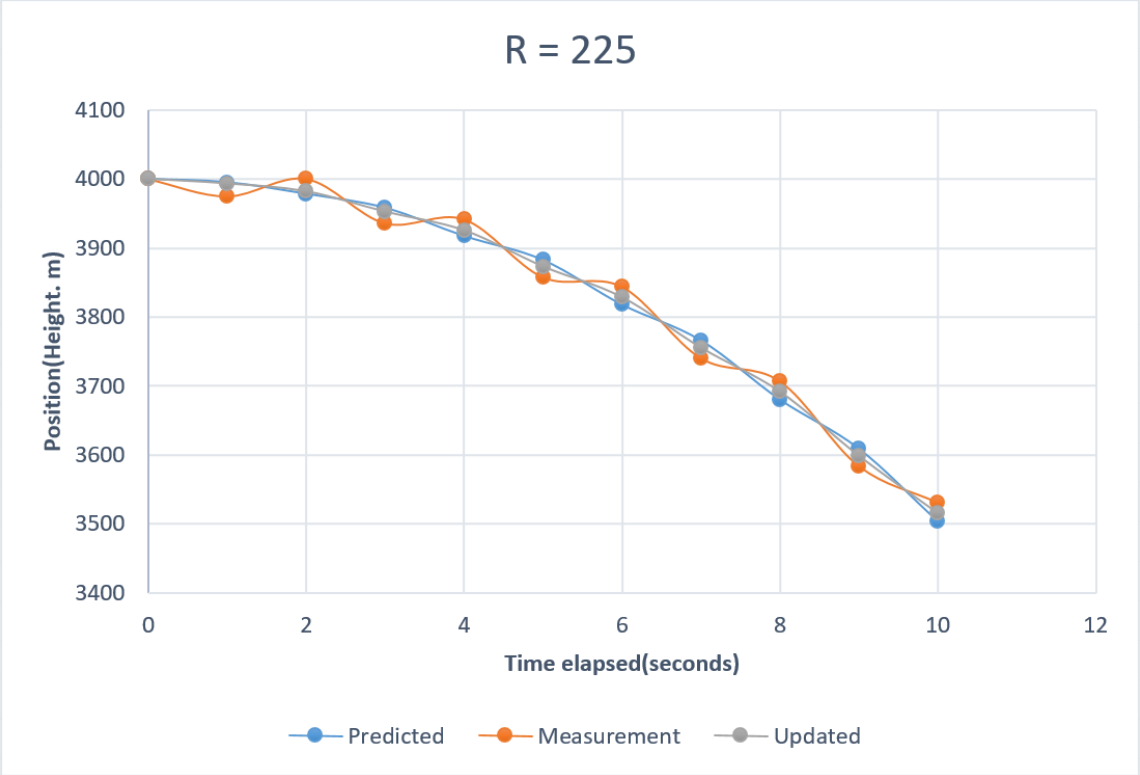

Figure 8: The updated positions move closer to the predictions and creating a smoother curve

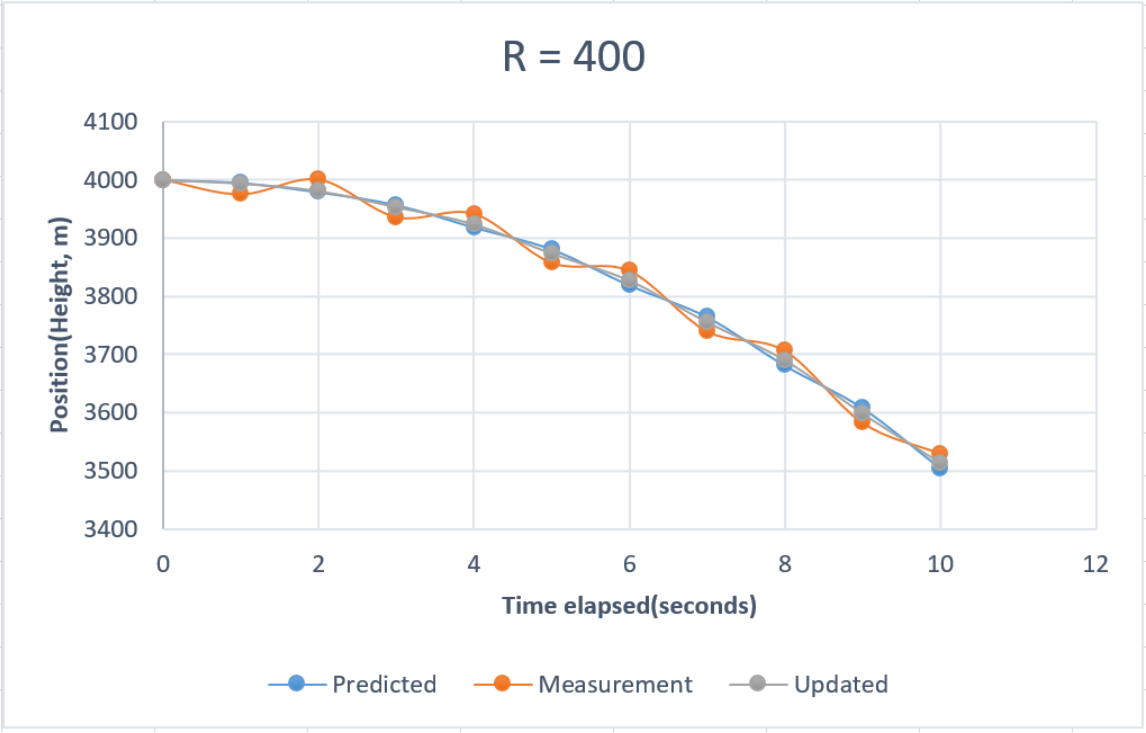

Figure 9: The updated positions and predictions lie almost together and closing up the gap

A Realistic Example

Let's assume our ball is a generic basketball with:

Mass (m) = 0.6kg

Diameter = 9inches/0.2286m

Radius (r) = 4.5inches/0.1143m

When an object is free falling on earth, other than the gravitational force that acts upon it, another force that acts against it is air resistance.

$D = \frac{1}{2} \rho * A * Cd * V^2$;

$a = F/m$;

$F = mg - D$;

D = Air resistance;

ρ = density of air ($1.225 kg/m^3$);

A = cross-sectional area of the ball (πr^2);

a = acceleration of object;

Cd = coefficient of drag of object, for a sphere it's about 0.47;

F = force;

g = gravitational force ($-9.8 m/s^2$);

Knowing these formulas give us a new way to calculate a more realistic measurement of the ball's position rather than basing our calculation on simple kinematics.

1. Calculate estimated velocity, **Vi**.

$$\mathbf{Vi = Vi_{(t-1)} + a_{(t-1)} * \Delta t;}$$

2. Calculate acceleration, **a** based on **Vi**.

$$\mathbf{a = (m*g - \tfrac{1}{2}\,\rho * A * Cd * V^2)/m;}$$

3. Update velocity, **V** based on acceleration, **a**.

$$\mathbf{V = a*\Delta t + V_{(t-1)};}$$

4. Calculate position, **X** with updated velocity, **V**.

$$\mathbf{X = X_{(t-1)} - V*\Delta t;}$$

We will be using the same example from previous situation where the basketball is dropped at a height of 4000m where the position is recorded every second for 10seconds, with initial values of:

$X = 4000;$

$Vi_{(t-1)} = 0;$

$a_{(t-1)} = 0;$

And other constant values of:

$g = 9.8;$

$m = 0.6;$

$\rho = 1.225;$

$A = 0.041;$

$Cd = 0.47;$

Let's calculate the **1st** position of our ball.

1. $Vi = Vi_{(t-1)} + a_{(t-1)} * \Delta t$

   $= 0 + 0(1)$

   $= 0$

2. $a = (m*g - ½\ \rho*A*Cd*V^2)/m$

   $= (0.6*(9.8) - ½(1.225*0.041*0.47*0^2)/0.6$

   $= 9.8$

3. $V = a*\Delta t + V_{(t-1)}$

   $= 9.8*1 + 0$

   $= 9.8$

4. $X = X_{(t-1)} – V*\Delta t$

   $= 4000 – 9.8$

   $= 3990.2$

Let's calculate the **2nd** position of our ball.

1. $Vi = Vi_{(t-1)} + a_{(t-1)} * \Delta t$

   $= 0 + 9.8(1)$

   $= 9.8$

2. $a = (m*g - ½\ \rho*A*Cd*V^2)/m$

   $= (0.6*(9.8) - ½(1.225*0.041*0.47*9.8^2)/0.6$

   $= 7.9$

3. $V = a*\Delta t + V_{(t-1)}$

   $= 7.9*1 + 9.8$

   $= 17.7$

4. $X = X_{(t-1)} – V*\Delta t$

   $= 3990.2 – 17.7$

   $= 3972.5$

After 10 seconds, the results are shown as below (2 decimal places):

3990.20
3972.49
3951.15
3928.97
3906.66
3884.35
3862.03
3839.71
3817.39
3795.07

Using the idea as before, that our measurement represents our “sensor reading” and that it has an error of the standard deviation ±20. Hence, we will get something like:

4010.20
3992.49
3971.15
3908.97
3886.66
3904.35
3882.03
3859.71
3797.39
3815.07

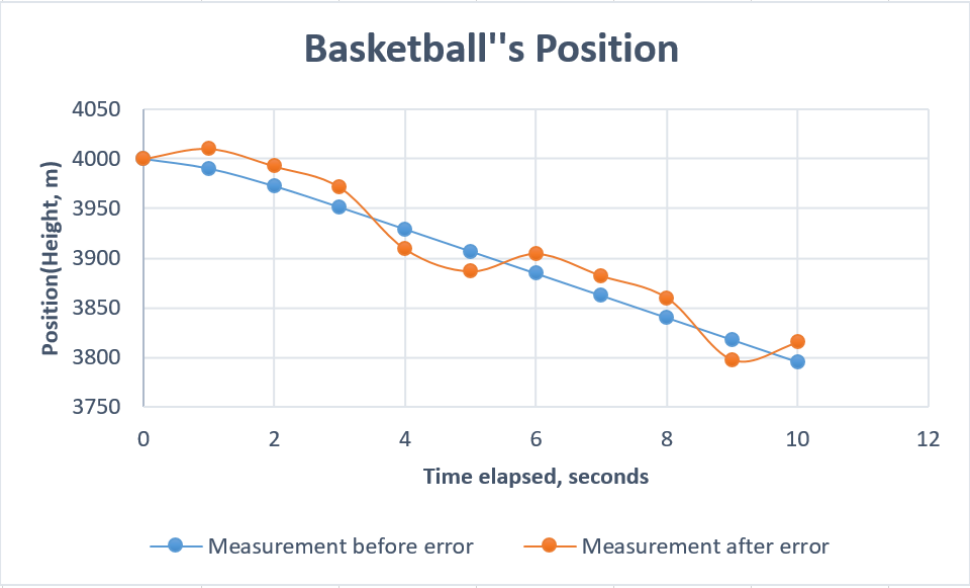

Figure 10: Graph showing both measurement before and after error

Using Kalman Filter with Realistic Example

Using the same C code from the *Implementation in C* section (page 8-10), the measurements feeding into the program will be the measurements after adding the ±20 error.

**Q (process error) is unknown for this situation, hence we will assume that the process is near perfect and put the minimum amount of Q = {(1,0)(0,1)}*

**We will observe the situation for R = 0^2, 5^2, 10^2, 15^2, 20^2, 40^2, 60^2 and 80^2*

R = 0^2

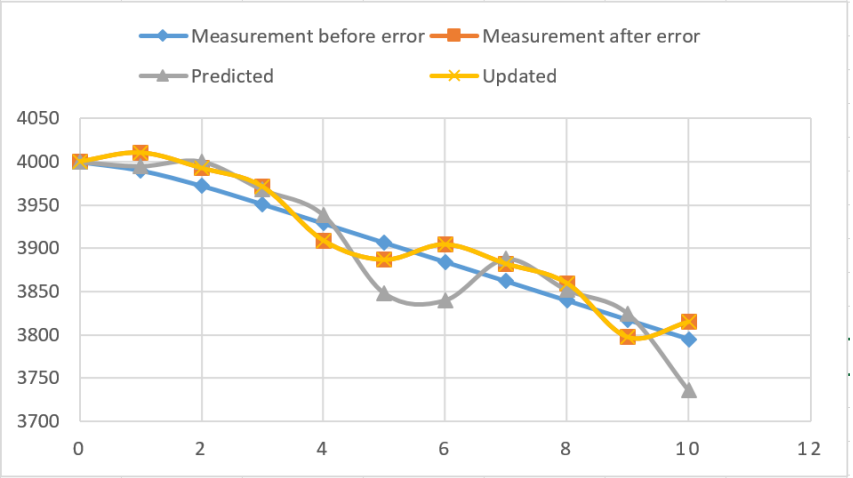

Figure 11

R = 5^2

Figure 12

R = 10^2

Figure 13

R = 15^2

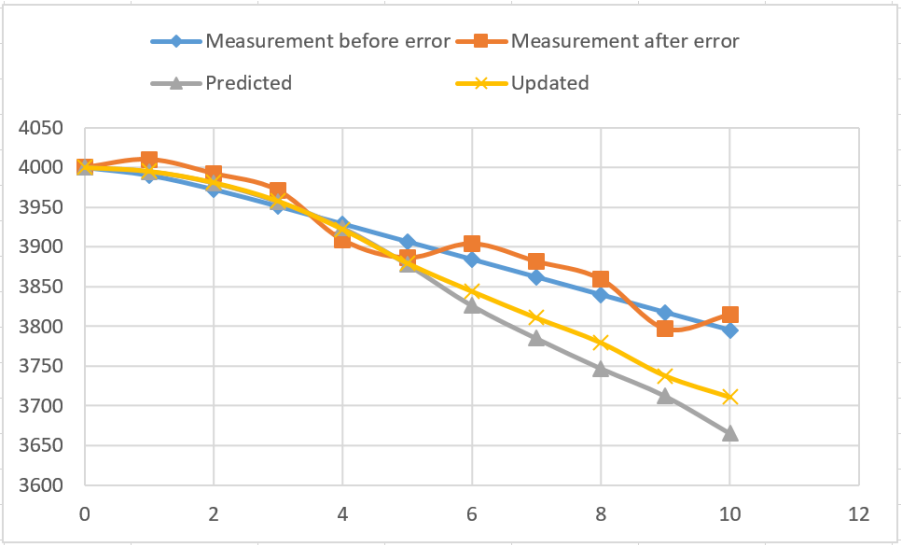

Figure 14

R = 20^2

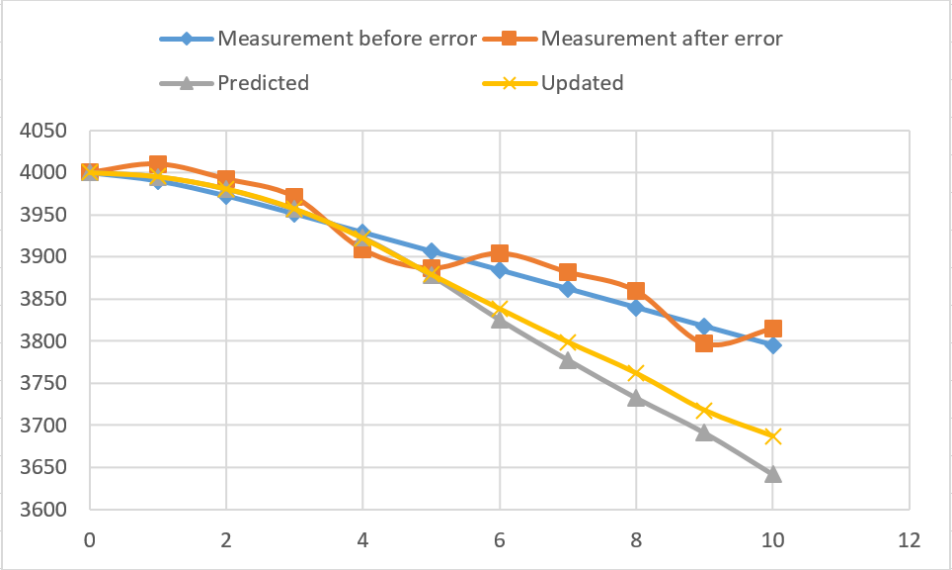

Figure 15

R = 40^2

Figure 16

R = 60^2

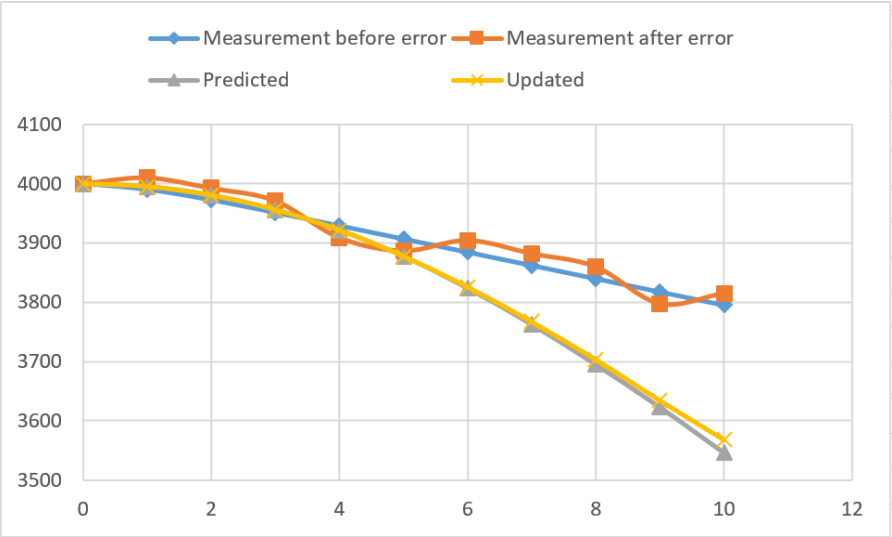

Figure 17

R = 80^2

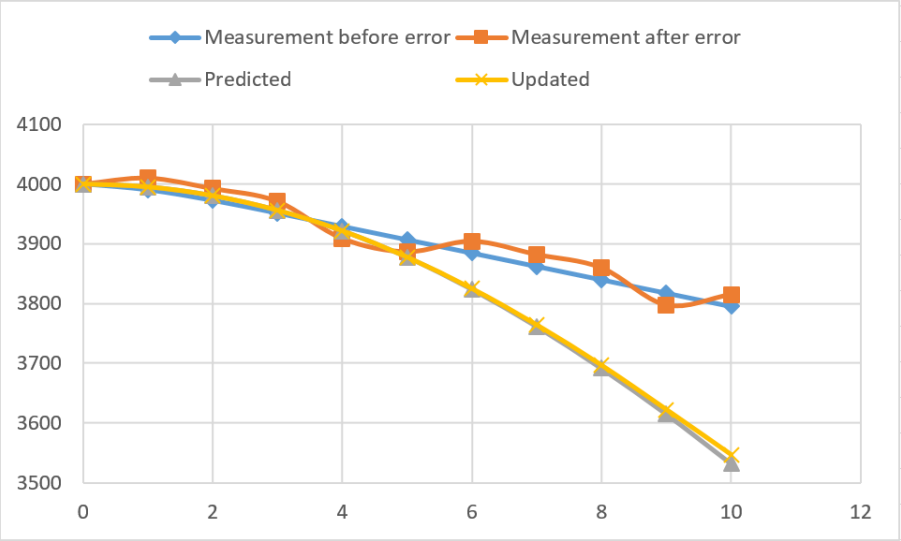

Figure 18

From the figures, we can observe that as R becomes bigger, the updated values move further away from the values before error.

In our case, when **R = 20^2**(the "sensor's error" that we know), the ideal situation would be having the updated values as close to the values before error as this indicates that our Kalman Filter is working as intended. However, as we can observe from *Figure 15,* it implies differently. This is because we have assumed that the process error, **Q** is near perfect which in reality, it is impossible.

Therefore, the next step to this project is to find the right Q with a suitable mathematical explanation for it.