Technitute_

Advanced Kalman Filtering and Sensor Fusion

2D Vehicle Unscented Kalman Filter: Prediction Step

UKF Exercise 1

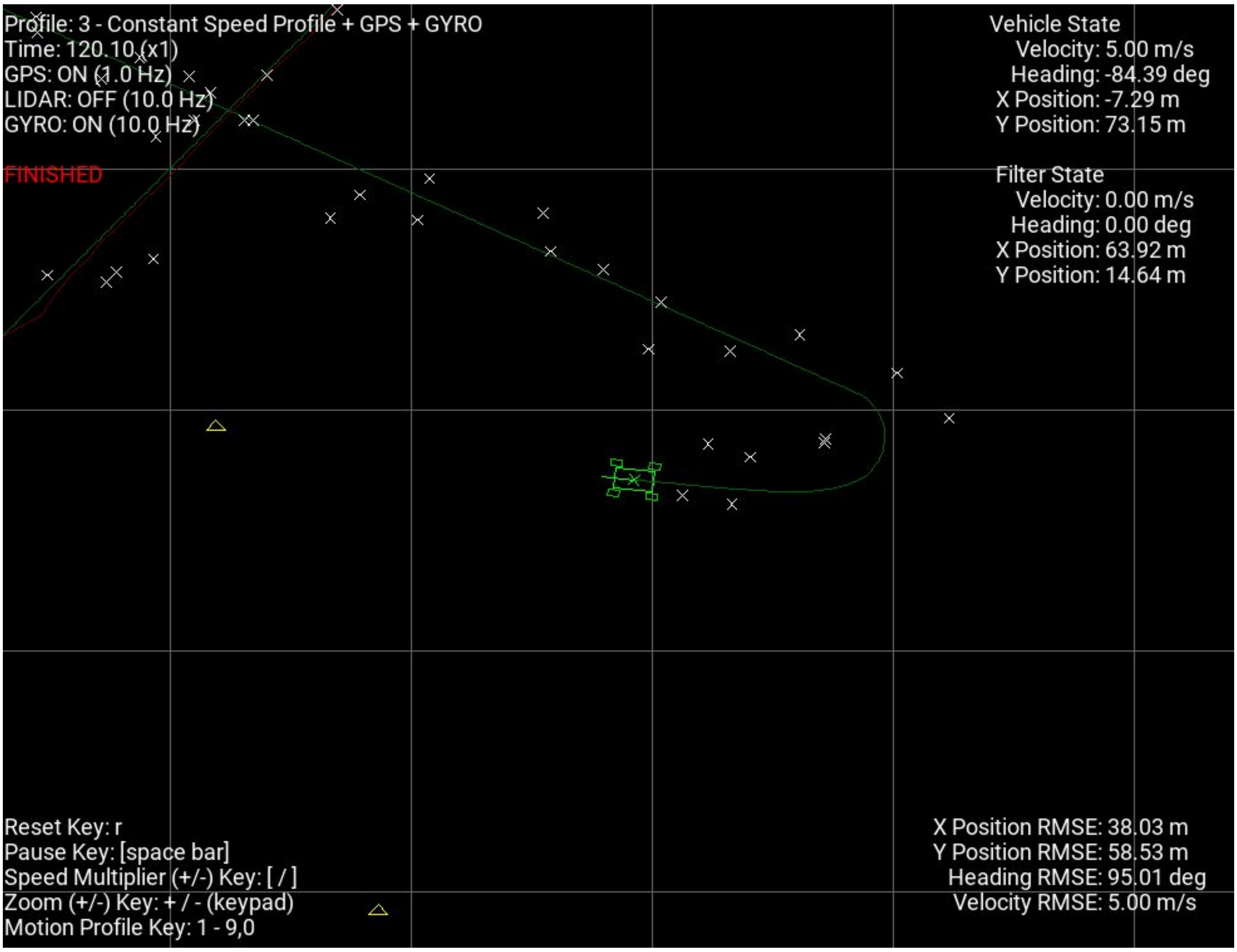
2D Vehicle UKF:
Prediction Step Exercise

Overview

Implement the ***Unscented Kalman Filter Prediction Equations*** and the ***2D Non-Linear Vehicle Process Model***.

Step 1 (Setup)

- Open the c++ file "**kalmanfilter.cpp**" which will be the file used in this exercise (it should be a new copy of the file "**kalmanfilter_ukf_student.cpp**" file).
- Compile the run the simulation as is, using profile 3. See that the car starts at the origin (0,0) and moves at 5 m/s while performing a series of turns.
- Note: The GPS measurement model and state initialisation is copied from the previous.Linear Kalman Filter exercises as the filter equations and logic are the same, so the filter will start to run (but using GPS only and without an process model)



2D Vehicle UKF:
Prediction Step Exercise

```
// ------------------------------------------------------------------------------- //
// Advanced Kalman Filtering and Sensor Fusion Course - Unscented Kalman Filter
//
// ####### STUDENT FILE #######
//
// Usage:
// -Rename this file to "kalmanfilter.cpp" if you want to use this code.

#include "kalmanfilter.h"
#include "utils.h"

// ------------------------------------------------------------------------ //
// YOU CAN USE AND MODIFY THESE CONSTANTS HERE
constexpr double ACCEL_STD = 1.0;
constexpr double GYRO_STD = 0.01/180.0 * M_PI;
constexpr double INIT_VEL_STD = 10.0;
constexpr double INIT_PSI_STD = 45.0/180.0 * M_PI;
constexpr double GPS_POS_STD = 3.0;
constexpr double LIDAR_RANGE_STD = 3.0;
constexpr double LIDAR_THETA_STD = 0.02;
// ------------------------------------------------------------------------ //
```

Constants that you can modify and use

```
void KalmanFilter::handleGPSMeasurement(GPSMeasurement meas)
{
    // All this code is the same as the LKF as the measurement model is linear
    // so the EKF update state would just produce the same result.
    if(isInitialised())
    {
        VectorXd state = getState();
        MatrixXd cov = getCovariance();

        VectorXd z = Vector2d::Zero();
        MatrixXd H = MatrixXd(2,4);
        MatrixXd R = Matrix2d::Zero();

        z << meas.x,meas.y;
        H << 1,0,0,0,0,1,0,0;
        R(0,0) = GPS_POS_STD*GPS_POS_STD;
        R(1,1) = GPS_POS_STD*GPS_POS_STD;

        VectorXd z_hat = H * state;
        VectorXd y = z - z_hat;
        MatrixXd S = H * cov * H.transpose() + R;
        MatrixXd K = cov*H.transpose()*S.inverse();

        state = state + K*y;
        cov = (Matrix4d::Identity() - K*H) * cov;

        setState(state);
        setCovariance(cov);
    }
```

GPS Measurement is already done (Same as LKF)

2D Vehicle UKF: Prediction Step Exercise

```
else
{
    VectorXd state = Vector4d::Zero();
    MatrixXd cov = Matrix4d::Zero();

    state(0) = meas.x;
    state(1) = meas.y;
    cov(0,0) = GPS_POS_STD*GPS_POS_STD;
    cov(1,1) = GPS_POS_STD*GPS_POS_STD;
    cov(2,2) = INIT_PSI_STD*INIT_PSI_STD;
    cov(3,3) = INIT_VEL_STD*INIT_VEL_STD;

    setState(state);
    setCovariance(cov);
}
```

State Initialization is automatically done on ***FIRST*** GPS measurement

$$\hat{x}_0 = \begin{bmatrix} p_{x\,gps} \\ p_{y\,gps} \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{P}_0 = \begin{bmatrix} \sigma^2_{gps} & 0 & 0 & 0 \\ 0 & \sigma^2_{gps} & 0 & 0 \\ 0 & 0 & \sigma^2_{\psi_0} & 0 \\ 0 & 0 & 0 & \sigma^2_{V_0} \end{bmatrix}$$

NOTE:
The code assumes that the state vector has the following form!!

$$\hat{x} = \begin{bmatrix} p_x \\ p_y \\ \psi \\ V \end{bmatrix}$$

Helper Functions

```
VectorXd normaliseState(VectorXd state)
{
    state(2) = wrapAngle(state(2));
    return state;
}
VectorXd normaliseLidarMeasurement(VectorXd meas)
{
    meas(1) = wrapAngle(meas(1));
    return meas;
}
```

2D Vehicle UKF: Prediction Step Exercise

Step 2 (Implement the SigmaPoint Functions)

- You can use the Cholesky Decomposition [cov.llt()]

```
std::vector<VectorXd> generateSigmaPoints(VectorXd state, MatrixXd cov)
{
    std::vector<VectorXd> sigmaPoints;

    // ----------------------------------------------------------------------- //
    // ENTER YOUR CODE HERE

    // ----------------------------------------------------------------------- //

    return sigmaPoints;
}

std::vector<double> generateSigmaWeights(unsigned int numStates)
{
    std::vector<double> weights;

    // ----------------------------------------------------------------------- //
    // ENTER YOUR CODE HERE

    // ----------------------------------------------------------------------- //

    return weights;
}
```

$$\kappa = 3 - n$$

$$\begin{aligned} x^{(0)} &= x \\ x^{(i)} &= x + \sqrt{(n+\kappa)P}_i \\ x^{(n+i)} &= x - \sqrt{(n+\kappa)P}_i \end{aligned} \quad i = 1, \ldots, n$$

$$W^{(0)} = \frac{\kappa}{n+\kappa}$$

$$W^{(i)} = \frac{1}{2(n+\kappa)} \quad i = 1, \ldots, 2n$$

2D Vehicle UKF: Prediction Step Exercise

Step 3 (Implement the Process Model)

- Modify the ***vehicleProcessModel()*** function
- Same model as EKF, however take in the Augmented State and Return the new State.
- Do Not Normalise the heading angle! (Discontinuities here interfere with results)!

```
VectorXd vehicleProcessModel(VectorXd aug_state, double psi_dot, double dt)
{
    VectorXd new_state = VectorXd::Zero(4);

    // ----------------------------------------------------------------------- //
    // ENTER YOUR CODE HERE

    // ----------------------------------------------------------------------- //

    return new_state;
}
```

$$x^a = \begin{bmatrix} p_x & p_y & \psi & V & w_{\dot{\psi}} & w_a \end{bmatrix}^T$$

$$\begin{bmatrix} p_x \\ p_y \\ \psi \\ V \end{bmatrix}_k = \begin{bmatrix} p_x \\ p_y \\ \psi \\ V \end{bmatrix}_{k-1} + \Delta t \begin{bmatrix} V_{k-1}\cos(\psi_{k-1}) \\ V_{k-1}\sin(\psi_{k-1}) \\ \dot{\psi}_k + w_{\dot{\psi}} \\ w_a \end{bmatrix}$$

2D Vehicle UKF:
Prediction Step Exercise

Step 4 (Implement the Unscented Transform Prediction Step)

- Modify the ***predictionStep()*** function
- Augment the State and Covariance Matrix with process noise
- Generate the Sigma Points (using augmented data) and Weights
- Transform Sigma Points with Vehicle Process Model
- Calculate the mean of the transformed Sigma Points
- Calculate the covariance of the transformed Sigma Points

```
void KalmanFilter::predictionStep(GyroMeasurement gyro, double dt)
{
    if (isInitialised())
    {
        VectorXd state = getState();
        MatrixXd cov = getCovariance();

        // Implement The Kalman Filter Prediction Step for the system in the
        // section below.
        // HINT: Assume the state vector has the form [PX, PY, PSI, V].
        // HINT: Use the Gyroscope measurement as an input into the prediction step.
        // HINT: You can use the constants: ACCEL_STD, GYRO_STD
        // HINT: Use the normaliseState() function to always keep angle values within correct range.
        // HINT: Do NOT normalise during sigma point calculation!
        // ----------------------------------------------------------------------- //
        // ENTER YOUR CODE HERE



        // ----------------------------------------------------------------------- //

        setState(state);
        setCovariance(cov);
    }
}
```

$$x^a_{k-1} = \begin{bmatrix} \hat{x}^+_{k-1} \\ 0 \\ 0 \end{bmatrix}$$

$$P^a_{k-1} = \begin{bmatrix} P^+_{k-1} & 0 & 0 \\ 0 & \sigma^2_{gyro} & 0 \\ 0 & 0 & \sigma^2_{accel} \end{bmatrix}$$

$$\hat{x}^{(i)}_k = f(\hat{x}^{(i)}_{k-1}, u_k, w^{(i)}_k)$$

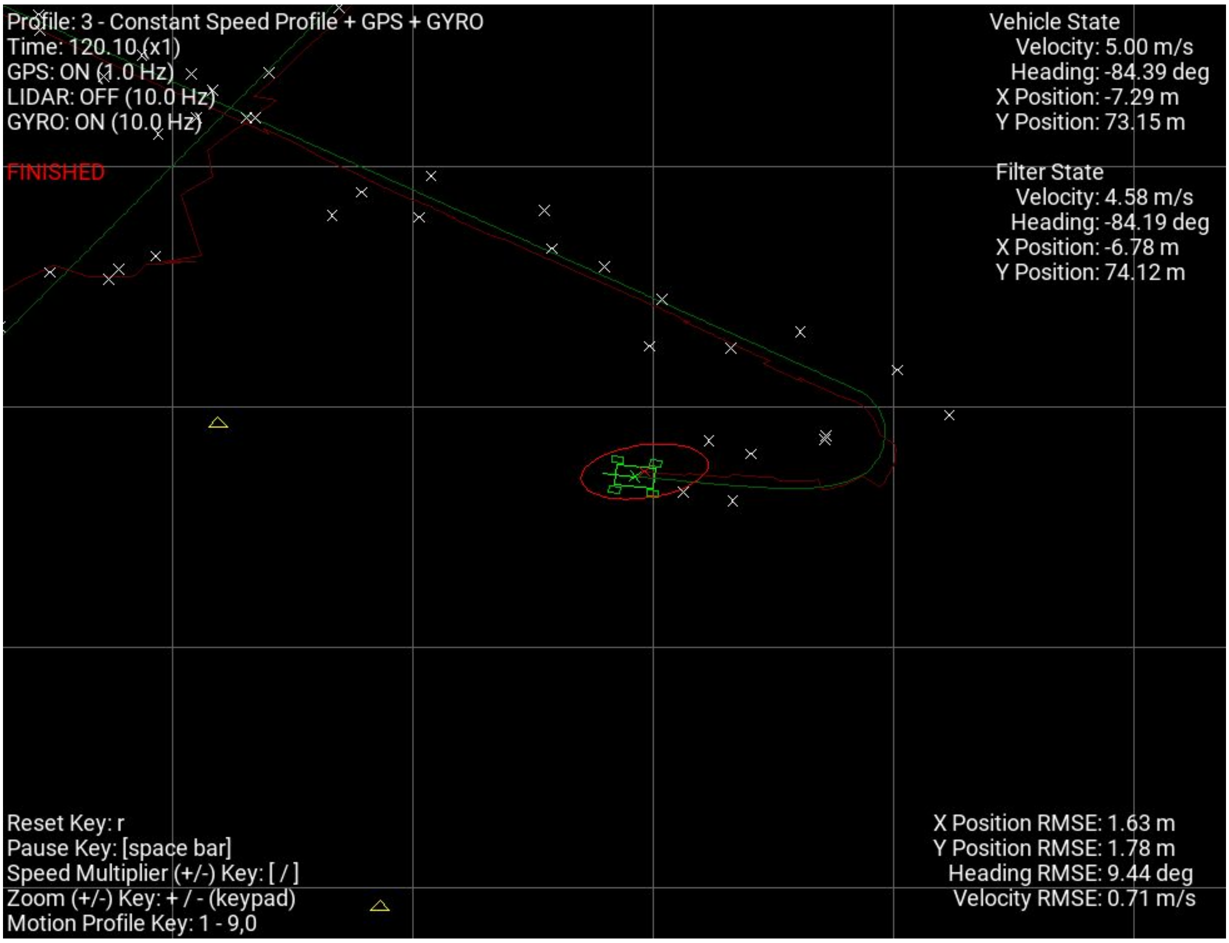
$$\hat{x}^-_k = \sum_{i=0}^{2n} W^{(i)} \hat{x}^{(i)}_k$$

$$P^-_k = \sum_{i=0}^{2n} W^{(i)} \left(\hat{x}^{(i)}_k - \hat{x}^-_k\right)\left(\hat{x}^{(i)}_k - \hat{x}^-_k\right)^T$$

2D Vehicle UKF: Prediction Step Exercise

Step 5 (Run the Simulation)

- Check out how the simulation runs with profiles 1 - 4.



2D Vehicle UKF:
Prediction Step Exercise

Step 6 (You can quickly compare to the EKF Solution for the Error Statistics)

Note: There will be slight differences in the results due to tuning, etc.

2D Vehicle UKF: Prediction Step Exercise

Step 7 (Play around with your code and make sure it is working as expected)

- Is this filter performing better than the EKF? Is that to be expected? Is it more robust?
- Is this filter method easier or harder to implement compared to the EKF?
- Do you think this filter is computationally more or less expensive than the EKF?