UKF Localisation of a Formula Student Car

Corentin Chauvin-Hameau

January 2019

***Abstract* — This report presents the implementation of an Unscented Kalman Filter used for the localisation of a driverless racing car taking part in Formula Student competitions. The UKF can fuse the data measured by the various sensors mounted on the car (IMU, odometry, GNSS and cones perception). The development is done under the Robot Operating System and using a custom Gazebo simulation providing realistic data.**

I. INTRODUCTION

A. *Problem description*

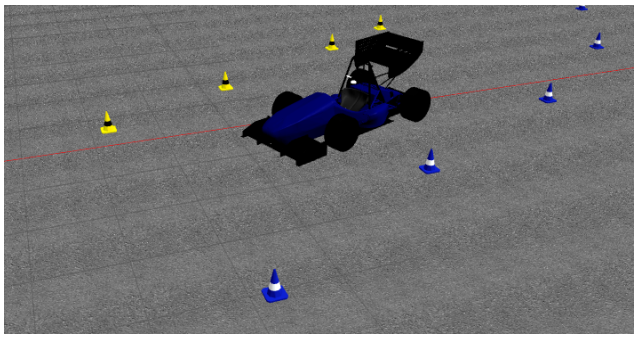


Figure 1: Gazebo simulation

Formula Student is an engineering competition where student teams from all over the world design and build formula racing cars. Since a few years it presents a driverless category, which requires the car to complete several laps autonomously. Here, determining the position of the car is a core issue, and is helped by several sensors embedded in the car. The Formula Student car of KTH (the one which is considered for this project) is equipped with:

- an Inertial Measurement Unit (IMU) measuring linear accelerations and orientation velocities
- wheel encoders providing odometry
- a standard GNSS giving position and velocity. Actually, the IMU and the GNSS measurements are provided by the same sensor, and the data can already be fused by its internal algorithms. However, the measurements will be handled as if it was given by separate sensors.
- cones position detection. The track is delimited by cones on its sides (see figure 1). During the competition, the position of the cones is not known at the beginning. However, the mapping and the data association won't be considered in this report, so the absolute position of the cones will be assumed to be known.

The car and its sensors are simulated in Gazebo. The car motion and dynamics are faithful to the real car. The IMU and the GNSS are simulated by plugins reproducing typical behaviours of these sensors, with noise parameters calibrated to fit the actual sensors mounted on the car. The odometry is generated taking the ground truth given by Gazebo, and adding noise, mainly according to the *odometry motion model* in [1] (section 5.4) (it won't be presented in this report). The real car can either detect the cones thanks to a stereo camera, or a lidar. The cones detection will be faked by adding a Gaussian noise of constant variance to the cones position given by Gazebo.

All the simulated output are in the same format and naming as for the real car. Therefore, the developed algorithm can be used on the real car without modification.

The main goal of the project is to fuse the different measurements with an Unscented Kalman Filter, to estimate the pose and the twist of the car. The structure of the algorithm is mainly based on [2] which describes a UKF for an autonomous car equipped with an IMU, a DMI (Distance Measuring Instrument) and a GNSS. The application and the sensors are very similar, so it seems to be a very good starting point.

Some additions have to be made to handle the extra data measured by the sensors that we have. For instance, DMI is providing information about the linear speed of the car, while wheels odometry gives information about the linear and angular speeds of the car. Moreover, the GNSS speed is not fused in the article, while the speed is often much more accurate than the position with this kind of sensors. Indeed, the speed can be determined accurately by using Doppler effect on the motion of the satellites, as explained in [3]. Finally, the cones detection can give information about the absolute position of the car, with relatively high accuracy (the lidar is for instance detecting cones with an accuracy of 3 cm).

B. *Contribution*

The contribution of this project is about:

- providing a full Gazebo simulation of the car and its sensors

- implementing the UKF of [2] and adapting it
- developing the measurement models for the odometry, the GNSS speed and the cones

The Gazebo simulation is highly based on the code of the Formula Student team of Edinburgh (sources on GitHub: `https://github.com/eufsa/eufs_sim`). What remained was to adapt the sensors used by the KTH Formula Student team, fake the cones detection and noise the odometry. I was asked to develop this simulation for the KTH team, so my work will not only be used for this project.

In the UKF algorithm, the prediction phase of [2] was to be adapted, since the state is slightly different here (see next section).

II. UKF LOCALISATION

A. Algorithm

1 **UKF_localisation(** $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$ **):**
2 *// Prediction*
3 $\chi_{t-1} = [\mu_{t-1}, \mu_{t-1} \pm (\sqrt{(n+\kappa)\Sigma_{t-1}})_i]$
4 $\bar{\chi}_t = predict(\chi_{t-1}, u_t)$
5 $\mu_t = \sum_{i=0}^{2n} w_m^{(i)} \bar{\chi}_t^{(i)}$
6 $\Sigma_t = \sum_{i=0}^{2n} w_c^{(i)} (\bar{\chi}_t^{(i)} - \mu_t)(\bar{\chi}_t^{(i)} - \mu_t)^T + Q$
7
8 *// Measurement update*
9 **foreach** *sensor measurement available* z_t **do**
10 $\bar{\chi}_t = [\mu_{t-1}, \mu_{t-1} \pm (\sqrt{(n+\kappa)\Sigma_{t-1}})_i]$
11 $Z_t = measurement\_model(\bar{\chi}_t, z_t)$
12 $\hat{z}_t = \sum_{i=0}^{2n} w_m^{(i)} Z_t^{(i)}$
13 $S_t = \sum_{i=0}^{2n} w_c^{(i)} (Z_t^{(i)} - \hat{z}_t)(\bar{z}_t^{(i)} - \hat{z}_t)^T + R$
14 $\Sigma_t^z = \sum_{i=0}^{2n} w_c^{(i)} (\bar{\chi}_t^{(i)} - \mu_t)(Z_t^{(i)} - \hat{z}_t)^T$
15 $K_t = \bar{\Sigma}_t S_t^{-1}$
16
17 $\mu_t = \mu_t + K_t(z_t - \hat{z}_t)$
18 $\Sigma_t = \Sigma_t - K_t S_t K_t^T$
19 **end**
20
21 **return** μ_t, Σ_t

Algorithm 1: Unscented Kalman Filter

This section describes the implemented algorithm. Its structure is the same as the classical one in [1], and the prediction is almost as in [2]. The state that we will try to estimate is $(x, y, \theta, v_x, v_y, \omega)$, where (x, y) is the absolute position of the car, θ is its heading, v_x is its longitudinal speed, v_y is its lateral speed, and ω is its angular speed.

As said previously, the state is slightly different from the one in [2]. Indeed, we will assume that the car is on a plane and work in 2D. Moreover, ω is added in the state since the UKF will fuse the angular speed from the odometry.

In the paper, the lateral speed is estimated in the prediction phase with the lateral acceleration given by the IMU, and never updated in the update phase. This is problematic because the IMU has always small biases, and it will accumulate errors through time. In our case, GNSS twist measurements will give additional information about the lateral speed, and will prevent these effects. However, when the GNSS twist is not fused, the lateral speed will be assumed to be zero.

At each time instant t, the UKF algorithm can be split into two different phases: a *prediction* phase where the filter tries to estimate the relative motion of the car, and an *update* phase where it takes into account sensors readings to correct the state.

Prediction phase

In the prediction phase, 13 *sigma points* are generated around the state estimate μ_{t-1} (see line 3 in algorithm 1), two by dimensions (here $n = 6$) and the state estimate itself. The way the sigma points are initially spread depends on the state covariance Σ_{t-1}. If the uncertainty on the state is big, the points will be spread a lot. The motion of each sigma points is estimated by the prediction function, and a new state can be computed by averaging these sigma points (lines 4 and 5). The average is weighted so that the sigma point corresponding to μ_{t-1} is given more weight than the others. Likewise, a new state covariance Σ_t is computed by averaging the covariance of each sigma points (line 6). A way to compute w_c, w_m and κ can be found in [4].

predict(χ_{t-1}, u_t **):** 1
foreach *sigma point* $(x_t^i, y_t^i, \theta_t^i, v_{x,t}^i, v_{y,t}^i, \omega_t^i)$ **do** 2
$x_{t+1}^i = x_t^i + (v_{x,t}^i \cos(\theta_t^i) - v_{y,t}^i \sin(\theta_t^i))\delta_t$ 3
$y_{t+1}^i = y_t^i + (v_{x,t}^i \sin(\theta_t^i) + v_{y,t}^i \cos(\theta_t^i))\delta_t$ 4
$\theta_{t+1}^i = \theta_t^i + \omega_t^i \delta_t$ 5
$v_{x,t+1}^i = v_{x,t}^i + u_t[a_x]\delta_t$ 6
$v_{y,t+1}^i = v_{x,t}^i + u_t[a_y]\delta_t$ 7
$\omega_t^i = \nu\omega_t^i + (1-\nu)u_t[\omega]$ 8
end 9
10
return $\bar{\chi}_t$ 11

Algorithm 2: Prediction phase

The algorithm presented by [2] uses the data from the IMU for the prediction of the linear speeds. With the same idea, we will use the IMU angular speed $u_t[\omega]$ to also update ω (line 8 of algorithm 2). This weird thing with the ν factor is a way to incorporate the angular speed fused from the odometry. If ν is small, a big weight is given to the IMU, while if ν is near 1.0, more weight is given to the odometry.

In a different spirit, a better idea could have been to fuse the IMU data in the update phase, and not in the prediction. For example, [5] presents the implementation of the Extended Kalman Filter of a very common ROS package for localisation, `robot_localization` (which also has a UKF). In this implementation, the

state is also the full 3D pose and twist. The prediction is done by propagating the dynamical model, and the sensors readings are handled only in the update phase. This has the advantage of taking into account the covariance of the IMU and avoiding this ν trick.

Update phase

During the update phase, each sensor data is processed sequentially. Again, sigma points are generated with the same method as previously. A measurement model is used to predict the observation which could be done for each sigma point (more on the different measurement models used in the next subsection). An average $\hat{z}_t$ of these 13 observations is then computed (line 12 of algorithm 1). Its uncertainty S_t is computed using the measurement uncertainty matrix R, which depends on the sensor type (line 13). Σ_t^z is the cross-covariance between the sigma points and the observation, and is used to compute the Kalman gain K_t (line 15). The state is then corrected by comparing $\hat{z}_t$ to the actual measurement z_t, and its covariance decreased.

B. Measurement models

In the implementation, the different *measurement_model* functions return the predicted measurement Z_t of all the sigma points, but also the true measurement z_t itself and its uncertainty R. Since the lines 12 to 18 of algorithm 1 are handled by the same function for each sensor, the output of the *measurement_model* functions has to be of the same dimension.

The ROS measurement messages carry their own uncertainties, which has been tuned to be faithful to the data. This will be used for generating R. However, for some sensors (cones detection and GNSS position in particular), it was better not to use the uncertainties in the ROS messages, but rather have it as tunable parameters in order to adapt the influence of each measurement.

Odometry model

The odometry sensor is providing linear speed v_x and angular speed ω. The measurement vector is then $z_t = (0, 0, 0, v_x, 0, \omega)$. For each sigma point, the projected measurement is $Z_t^{(i)} = (0, 0, 0, v_{x,t}^{(i)}, 0, \omega_t^{(i)})$. Finally, the uncertainty matrix is $R = diag(a, a, a, \sigma_{v_x}^2, a, \sigma_\omega^2)$ where a is a big number preventing the measurement on the pose to have to much influence (for example 10^{15}).

GNSS pose model

This one is the same as in [2]. A GNSS is able to estimate its latitude and longitude, and then a ROS package converts it in a Cartesian absolute frame (for example pointing to the East). This absolute frame has to be the same as the frame in which the state is estimated (or conversion has to be done). The true measurement is therefore $z_t = (x, y, 0, 0, 0, 0)$. For each sigma point, $Z_t^{(i)} = (x_t^{(i)}, y_t^{(i)}, 0, 0, 0, 0)$.

The uncertainty matrix is $R = diag(\sigma_x^2, \sigma_y^2, a, a, a, a)$ where the σ are tunable parameters in this case. We will see it in the experimentations, the influence of this measurement is very important. For low values of σ, the estimated pose of the car is very shaky, as the measurement. For high values, the influence of the GNSS becomes negligible. A good balance has to be found.

GNSS twist model

The GNSS gives (v_x^0, v_y^0) in the absolute frame, while the estimated speed of the car is aligned with its heading. To bring the measurement back in the good frame, we will use the track (the direction of the speed) $\varphi = \text{atan2}(v_y^0, v_x^0)$ which we will assume equal to the heading of the car. This assumption holds when the amount of slip is not too important. Therefore,

$$\begin{aligned} v_x &= v_x^0 \cos(\varphi) + v_y^0 \sin(\varphi) \\ v_y &= -v_x^0 \sin(\varphi) + v_y^0 \cos(\varphi) \end{aligned}$$

We can also use this track to update the heading of the car. This is very valuable because it is the only sensor giving absolute information about the heading. The measurement vector is then $z_t = (0, 0, \varphi, v_x, v_y, 0)$. For each sigma point, $Z_t^{(i)} = (0, 0, \theta^{(i)}, v_{x,t}^{(i)}, v_{y,t}^{(i)}, 0)$.

The uncertainty on the speeds is contained in the ROS message, but not the uncertainty on the track. By differentiating the formula computing φ, we can compute $\sigma_\varphi^2 = ((v_x \sigma_{v_y})^2 + (v_y \sigma_{v_x})^2)/(v_x^2 + v_y^2)$. Then $R = diag(a, a, \sigma_\varphi^2, \sigma_{v_x}^2, \sigma_{v_y}^2, a)$. If the speed is small, the uncertainty on φ is very big, and it is probably the same for the uncertainties on v_x and v_y. Thus, the update is done only if the speed is above a specific threshold (1 m/s for example).

Cones detection model

The cones detection system is providing the noised position of each cones in range of detection in the sensor frame. The sensor is placed at the front of the car, on its x axis. (x_m, y_m) is then the position of the measured cone transformed into the car frame. We assume that we know the absolute position of the cones at the beginning and we don't care about data association. So for each detected cone, we have its absolute position (x_b, y_b).

We will process each cone sequentially. If it is easier than taking into account all the cones at the same time, it presents two drawbacks:

- for each cone update, the covariance of the state will decrease, which gives more weight to the first cones processed.
- an outlier could shift the state a lot, leading to bad data association when processing the next cones. This is not a problem in our assumptions.

A first idea could be to take $z_t = (x_b, y_b, 0, 0, 0, 0)$ as observation vector. For each sigma point, the predicted observation would then be the absolute position

of the cone: $Z_t^{(i)} = (x^{(i)} + x_m \cos\theta^{(i)} - y_m \sin\theta^{(i)}, y^{(i)} + x_m \sin\theta^{(i)} + y_m \cos\theta^{(i)}, 0, 0, 0, 0)$. However, if the cone is a bit far, the measurement is very sensitive to $\theta^{(i)}$. This causes a problem in the update phase.

To see that, let's assume the state of the car is perfectly known, and that the cone measurement has no noise. $Z_t^{(i)} - z_t = 0$ for $i = 0$ (the sigma point corresponding to the mean) and $i = 7$ to 12. And $Z_t^{(i)} + Z_t^{(i+1)} - 2z_t = 0$ for $i = 1$ and 3 (the sigma points where x and y is shifted by the same opposite constant). However, for the sigma points where the heading is shifted, we have $Z_t^{(4)} + Z_t^{(5)} - 2z_t \neq 0$. Therefore, $z_t - \hat{z}_t$ will be quite big, and the state will change a lot during the update.

A better idea is to take for each sigma point $Z_t^{(i)} = (x_t^{(i)}, y_t^{(i)}, 0, 0, 0, 0)$. The measurement is then the position of the car given the absolute position of the cone, the relative measurement and the estimated heading: $z_t = (x_b - x_m \cos(\theta) + y_m \sin(\theta), y_b - x_m \sin(\theta) - y_m \cos(\theta), 0, 0, 0, 0)$. This leads to much more stable behaviour.

An other problem was that the update was not only influencing the position, but also the heading with a big magnitude. A solution inspired by the `robot_localization` package is to manually select the elements of the state which should be updated (in line 17 of algorithm 1). Thus, only the position is updated, and the uncontrolled bad effects on the heading are removed. I however let the covariance update unchanged since the bad effects on the covariance were negligible.

III. SIMULATED RESULTS

In this section, the differences between the different sensors and their effect on the estimation are studied. The comparison will be done for one lap (about a hundred meters) where the car is driven manually, and with the Root Mean Square Errors (RMSE) in distance, linear speed, orientation and angular speed between the output of the UKF and the ground truth provided by the simulator: $RMSE_d = \sqrt{\sum_{t=1}^{T}[(x_0 - x)^2 + (y_0 - y)^2]/T}$ ($RMSE_v$ is computed the same way) and $RMSE_\theta = \sqrt{\sum_{t=1}^{T}(\theta_0 - \theta)^2/T}$ ($RMSE_\omega$ is computed the same way).

For all the presented figures, the green curve corresponds to the ground truth, the blue curve corresponds to the output of the UKF, and the red curve corresponds to the noised odometry input. The scale of the grid is one meter per tile. The grey points are the undetected cones, and the colored points are the detected cones. In the tables, the values are given in International System of Units.

Often, at one point in the runs (in the bottom left corner of the pictures), the estimated orientation is doing something weird. The reason is not very clear, there is probably a mistake with a modulo somewhere in the code. Fortunately, it doesn't seem to affect the remaining of the runs.

A. *Dead reckoning*

Using only the prediction phase with the IMU leads to very poor results in position, even when the experiment is conducted with a noise free IMU. If the position is often very badly determined, the orientation is way better. This difference comes from different factors:

- the position is obtained by integrating two times the linear acceleration given by the IMU, while the orientation is only the result of one integration of the angular speed.
- the simulated car is mounted on springs and dampers, which plays on its orientation during accelerations and decelerations. If the IMU is not perfectly horizontally oriented, gravity introduces additional biases. Taking into account the orientation of the IMU in the predict phase would be a good idea to deal with it.

It is hard to quantify the errors between the ground truth and the output of the filter because it varies a lot between each run. Figure 2 shows two typical runs, and table 1 the corresponding offsets and RMSE.

Run	**Final offsets**		**RMSE**			
	d	θ	d	θ	v	ω
1	11.41	0.01	12.96	0.01	1.78	0.02
2	0.31	0.46	9.17	0.91	7.94	0.04

Table 1: Final offsets and RMSE for dead reckoning

It becomes clear here that using the IMU in the prediction step was not necessarily a good idea. Using the linear speed provided by the odometry instead of the acceleration of the IMU could be better.

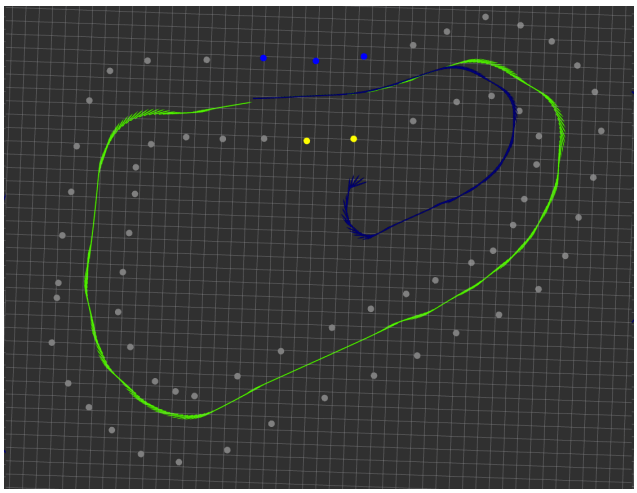
B. *IMU + odometry*

Fusing the odometry in the update phase helps to reduce the errors, but is not enough to reach an acceptable level. The differences between runs with same parameters are less important, so the results are easier to trust.

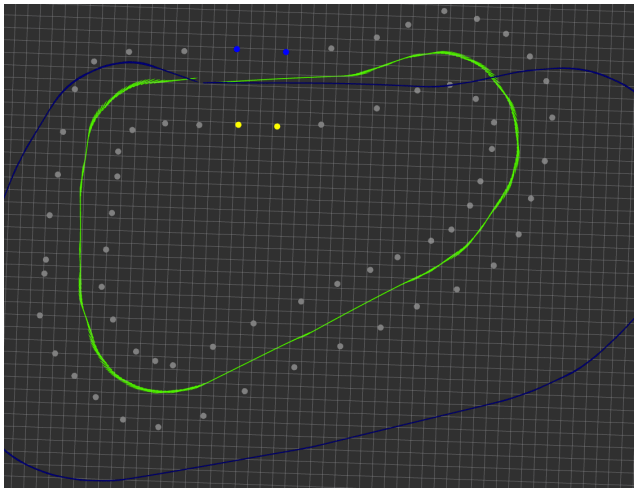
A big ν gives more weight to the angular speed of the odometry, and leads to slightly better results. However, even artificially decreasing the covariance of the odometry is not enough to get the output of the UKF looks like the odometry input.

ν	**Final offsets**		**RMSE**			
	d	θ	d	θ	v	ω
0.2	17.98	-0.65	12.16	0.43	0.07	0.06
0.8	10.00	0.34	8.77	0.75	0.08	0.41

Table 2: Final offsets and RMSE for IMU + odometry



(a)



(b)

Figure 2: Dead reckoning (IMU only) for two different runs

C. *IMU + odometry + GNSS position*

Adding the GNSS position information has huge effects on the error in position, which is now bounded. The GNSS mounted on the car has a standard deviation of 2.5 m, so the measurement is quite noisy. Playing with the standard deviation in the measurement model allows to find a balance between trusting the GNSS and correcting a lot the position based on its measurements (for low values of σ), and filtering its noise (high values of σ). For example, for $\sigma = 2.5$, the output of the filter is very shaky, while for $\sigma = 7.5$, the output is more reasonable (see figure 4).

σ	**Final offsets**		**RMSE**			
	d	θ	d	θ	v	ω
2.5	1.02	0.04	1.12	0.14	0.07	0.08
7.5	0.43	0.00	0.97	0.05	0.07	0.04

Table 3: Final offsets and RMSE when fusing GNSS position

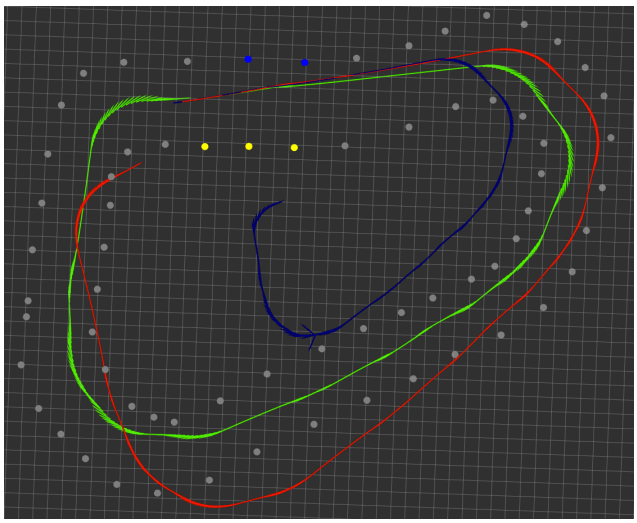


Figure 3: Fusing IMU and odometry with $\nu = 0.8$

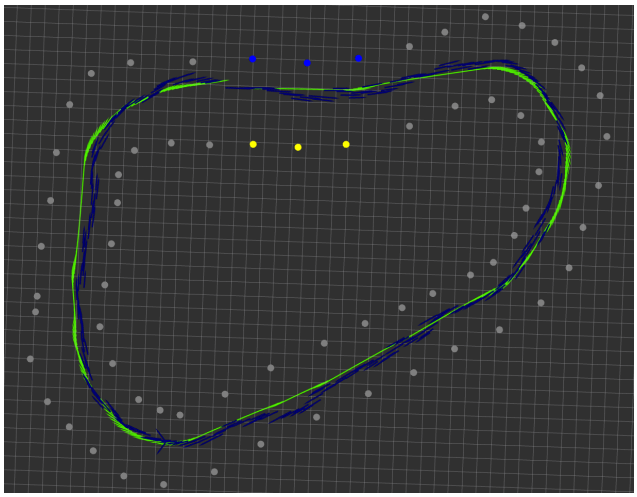


Figure 4: Fusing the GNSS position with $\sigma = 7.5$

D. *IMU + odometry + GNSS speeds*

It is interesting here not to fuse the GNSS position, to see clearly what happens without the shakes in the position. We can notice in figure 5 that even if the error in position still accumulates a bit, the output is cleaner than only fusing IMU and odometry. Moreover, thanks to the track information, the error in orientation is bounded.

Final offsets		**RMSE**			
d	θ	d	θ	v	ω
0.35	0.03	0.39	0.77	0.06	0.05

Table 4: Final offsets and RMSE when fusing GNSS speeds

E. *IMU + odometry + GNSS speeds + cones*

Since the standard deviation of the cones detection is of 3 cm, the accuracy of this measurement is of great value for the estimation of the position. In straight line,

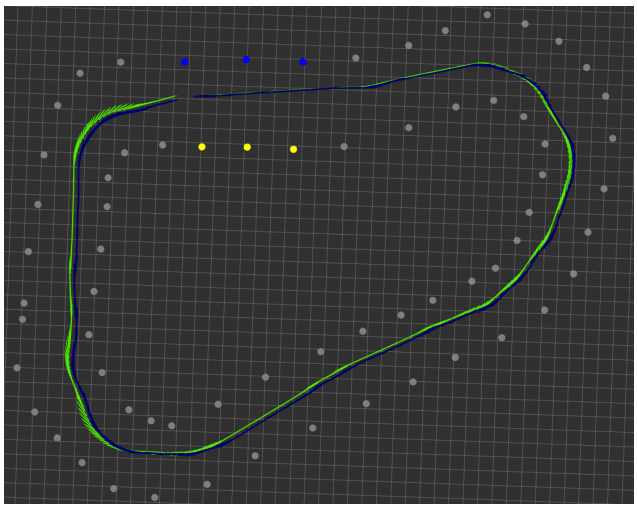


Figure 5: Fusing the GNSS speeds

the output of the filter is very good. But in turns, the cones update is badly shifting the position, probably because of small orientation errors to which the cones measurement is very sensitive. A quick fix could be to do the update only if the angular speed is under some threshold.

In this configuration, since the data association is provided, the absolute localisation problem can be tackled: starting with a wrong initial state, the filter quickly converges towards the true state.

Final offsets		**RMSE**			
d	θ	d	θ	v	ω
0.06	0.01	0.42	0.50	0.06	*

Table 5: Final offsets and RMSE when fusing the cones (because of the orientation bug, the RMSE in ω is not relevant. The RMSE in θ is probably affected as well)

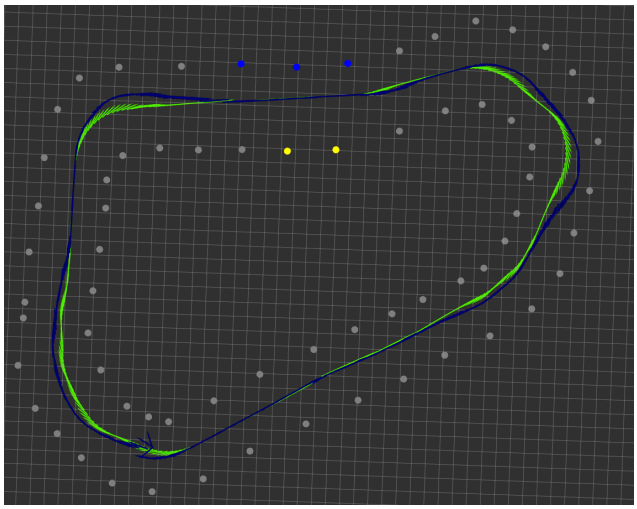


Figure 6: Fusing the cones position

IV. THE SOURCE CODE

The source code can be found on GitHub at: `https://github.com/CorentinChauvin/ukf-localisation`. All the instructions to install and launch it can be found in the README file.

The interesting files to look at are:

- the UKF implementation, in *ukf_localisation / src / ukf_localisation.py*
- the cones detection faking, in *gazebo_simulation / eufs_gazebo / nodes / cones_perception.py*
- the odometry noising, in *gazebo_simulation / eufs_gazebo / nodes / odometry_publisher.py*
- a video of the running UKF, in *REPORT / demo_ukf_localisation.mp4*

V. CONCLUSION

The presented Unscented Kalman Filter is able to fuse the various sensors mounted on the Formula Student car of KTH. The behaviour of its implementation has been studied in a realistic Gazebo simulation. It has been showed that even if the filter provides nice looking output, it is not perfect, and the accuracy of the state estimate wouldn't be enough in a driverless context. The main problem comes from the prediction phase using the IMU, which leads to very bad predictions. Some alternatives have been mentioned.

The next steps of this work could be to set up `robot_localization` for the car. This ROS package contains a robust implementation of a UKF which could give a good comparison of what we could expect from a good filter. Then, one could use this knowledge to investigate weaknesses of the implementation and fix it (some fixes have been mentioned). Finally, it would be a nice ending to try the code on the real car. The code is already ready to work on the current system, and new interesting issues would probably surface.

References

[1] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics*. MIT Press.

[2] Xiaoli Meng, Heng Wang, and Bingbing Liu. (2017). A Robust Vehicle Localization Approach Based on GNSS/IMU/DMI/LiDAR Sensor Fusion for Autonomous Vehicles.

[3] Salvatore Gaglione. (2015). How does a GNSS receiver estimate velocity?. *InsideGNSS*.

[4] Eric A. Wan, and Rudolph van der Merwe. (2000). The Unscented Kalman Filter for Nonlinear Estimation.

[5] Thomas Moore, and Daniel Stouch. (2015). A Generalized Extended Kalman Filter Implementation for the Robot Operating System.