Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Extended Kalman Filter & Particle Filter from MATLAB to Python

Seminar CSE Robotics

Noah Andrés Baumann

noabauma@student.ethz.ch

Institute for Dynamic Systems and Control
ETH Zürich

Supervisors:
Enrico Mion
Bhavya Sukhija
Jin Cheng
Prof. Raffaello D'Andrea

May 31, 2022

Acknowledgements

Thanks to Enrico, Bhavya and Jin for this fun project. I have learned much more than anticipated by translating the existing MATLAB scripts into python. Not only did I get a good sense for recursive estimations, but now I am fluent in MATLAB.

Abstract

In this project we will translate two existing MATLAB script from the ETH Zürich lecture: 'Recursive Estimation' into python, such that future students have the option to do the exercise in python instead of MATLAB. The two exercises are: Extended Kalman Filter and Particle Filter. We also improve an existing evaluation function, which compares performance and accuracy of hand-in submissions from students for the spring 2022 Recursive Estimation lecture. The new evaluation function now gives an overall performance and accuracy score of each submission. We compared both the MATLAB and python version regarding performance and accuracy. MATLAB outperforms python in speed as well as in accuracy.

Contents

Chapter 1

Introduction

Chapter 2 is all about the Extended Kalman Filter (EKF). Section 2.1 starts with understanding the existing 2021 EKF and a description of how we translated it into python. In section 2.2 we made performance and accuracy comparisons of both the MATLAB and python implementation. Chapter 3 describes the same as chapter 2 but for the particle filter (PF) instead of the EKF. Section 3.1 consists of the implementation and section 3.2 of the performance and accuracy measurements. Chapter 4 is the new evaluation function. In the final chapter we draw the conclusions of this study.

Chapter 2

Extended Kalman Filter

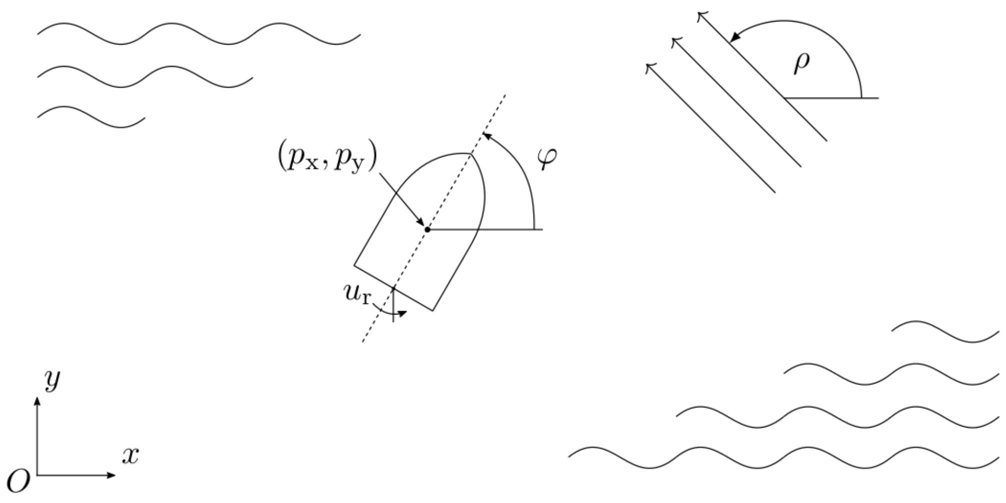


Figure 2.1: Schematic diagram of the boat and the fixed coordinate system (x, y) with origin O taken from the exercise sheet. The position of the boat is denoted by (p_x, p_y), the orientation with respect to the x-axis by φ and the control input to adjust the rudder by u_r. The wind blows at a constant absolute velocity and its direction with respect to the x-axis is given by ρ.

The EKF 2021 exercise consists of keeping track of all the states of a boat on a vivid lake using a hybrid Extended Kalman Filter. The five states of the boat are: x- and y-coordinates (p_x, p_y), x- and y-velocities (s_x, s_y) and orientation φ. The boat is drifting on the lake and gets pushed by a wind with constant velocity at a measured angle ρ. The boat also has a thrust u_t and a rudder u_r command. A schematic of the boat is shown in fig. 2.1. Measurements of five sensors are given: From the distance sensor A z_a, distance sensor B z_b, distance sensor C z_c, a gyro sensor z_g and a compass sensor z_n. In this simulation, the inputs are piecewise constant functions for a time sequence of random length. The distance sensor C is the only sensor which we do not get inputs at piecewise constant time. Instead, the inputs of sensor C are given at random time steps. All the inputs from the simulator are corrupted by noise, which is the reason why we use a hybrid EKF to estimate the state of the boat. The exercise is well documented on the webpage of the lecture 'Recursive Estimation'[1].

2.1 EKF 2021 implementation in python

For the whole EKF implementation, we only needed 5 different python libraries. The first one is NumPy for its amazing matrix/vector data structures, operations, methods and more. The second one is SciPy [1] for its ODE solver `solve_ivp`. The third one is Matplotlib [2] for its visualization. The fourth one is the `time` library for measuring the execution time. Lastly, we also needed the `copy` library to create deep copies of the `SimulationConst.py` and the `EstimatorConst.py` for calling the estimator and simulation function. We could have used another ODE solver library like CasADi [3], but we ended up using SciPy, due to the same integration method as the MATLAB ODE solver `ode45`, which both uses the explicit Runge-Kutta method of order 5(4). We split the whole code into specific files as it was done for the MATLAB implementation. The `run.py` function is used to execute a simulation of the true system: It runs the estimator, plots the results and reports the root-mean-squared error (RMSE) of the tracking, angular, velocity, wind and bias. The `Estimator.py` is the estimator where the EKF is implemented. For students, this file should be empty. The `EstimatorConst.py` is a class which consists of all the physical constants that are available to the estimator. `EstimatorState.py` is a class with the measurement update at time step k: $P_m(k)$, $x_m(k)$. `Simulator.py` consists of the simulator function used to simulate the motion of the boat and measurements. This function is called by `run.py` and should be obfuscated (i.e. its source code is not readable) when given to the students. `SimulationConst.py` is the class of the constants used in the simulation. These constants are not accessible to the estimator.

2.2 Performance & Accuracy Comparison

In this section we will explore the performance and accuracy of our python implementation with the MATLAB version of the EKF 2021 exercise. Due to MATLAB and NumPy having not the same random number generators (RNGs) we had to find another way to measure the similarities of both implementations, without loosing too much time. Only the uniform distribution RNG is identical in python and MATLAB. You can easily reproduce the same random numbers with the same seeds if you only use uniform distributions. For the normal distribution

[1]https://idsc.ethz.ch/education/lectures/recursive-estimation.html

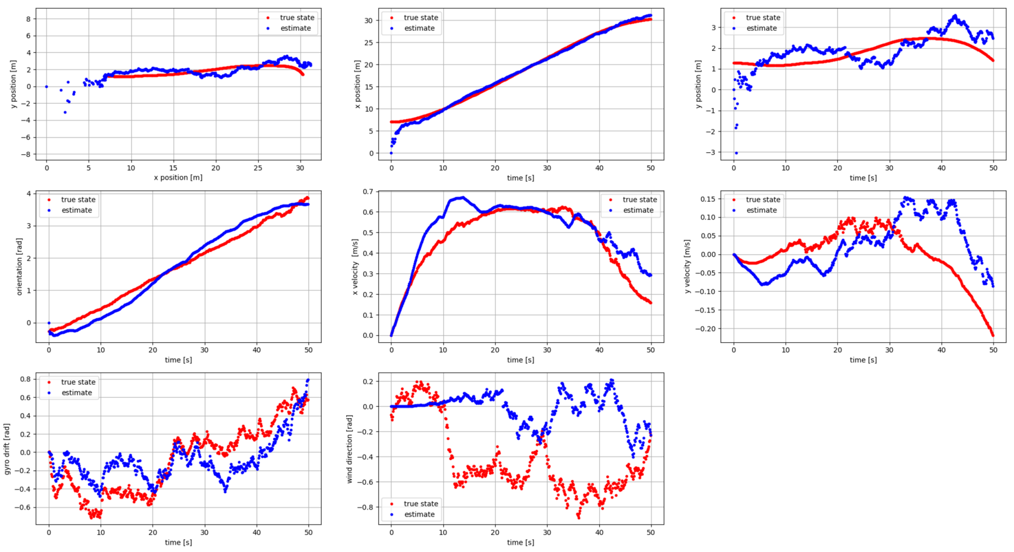


Figure 2.2: A random EKF 2021 plots. Red are the true states and blue are the estimated states. From top left to bottom right. y- and x-positions, x positions over time, y positions over time, orientiation [rad] over time, x velocity over time, y velocity over time, gyro drift [rad] over time and last wind direction [rad] over time.

MATLAB uses the Ziggurat algorithm, whereas NumPy uses the Box-Muller algorithm. We decided to measure the accuracy of both the implementations by running each a thousand times and get the mean ± std of all the errors. Doing this, we could also measure the performance of each implementation at the same time. We first started with trying to reproduce the same simulator boat measurements, to check if they are already accurate enough before trying to fit the EKF (see table 2.1). We can conclude that python and MATLAB generate similar output of their simulation function. Next, we tested the performance and accuracy of the whole program over 1000 runs (see table 2.2). All performance measurements where ran on a Ryzen R9 5950x at 3.4Ghz. The python version is less accurate and about 3x slower than the MATLAB version. Due to time reasons we could not figure out why the python script is less accurate. The runtime of the python script was slower than the MATLAB script as expected. MATLAB is known to be a relatively fast language. Figure 2.2 shows the EKF working properly in python.

2.3 from EKF 2021 to EKF 2022 in python

In this year Recursive Estimation lecture, the TAs decided to change up the EKF exercise from the previous year. This time not only the sensor C is getting

Table 2.1: EKF exercise 2021/2022 Simulator accuracy for 1000 runs with the `SimulationConst.py` values (see listing A.1).

		python		MATLAB	
		mean	variance	mean	variance
state means	p_x	16.174	6.883	17.149	6.610
	p_y	3.033	14.583	2.939	11.523
	s_x	0.436	9.025e-3	0.479	0.010
	s_y	8.735e-4	0.027	-0.006	0.019
	φ	-2.148e-3	2.772	-0.093	2.982
state vars	p_x	51.612	288.530	61.497	396.562
	p_y	7.242	5.659	5.362	39.039
	s_x	0.025	1.385e-4	0.029	3e-4
	s_y	0.014	1.341e-4	0.015	2e-4
	φ	0.953	0.4137	1.424	0.529
wind means	ρ	-0.006	0.180	-0.007	0.179
wind vars	ρ	0.082	0.005	0.083	0.005
drift means		-0.006	0.150	-0.007	0.164
drift vars		0.082	0.005	0.081	0.006
input means	u_t	0.050	2.423e-5	0.050	0.000
	u_r	-7.501e-5	1.128e-3	-0.002	0.001
input vars	u_t	8e-4	1.646e-8	8.050e-4	1.570e-8
	u_r	5e-4	1.242e-8	5.280e-4	1.210e-8
sense means	z_a	1.427e3	9.758	1.424e3	8.496
	z_b	1.984e3	6.908	1.983e3	6.665
	z_c	1.997e3	15.061	1.997e3	11.853
	z_g	-8.254e-3	2.938	-0.100	3.049
	z_n	-4.190e-3	2.779	-0.100	2.995
sense vars	z_a	49.697	342.056	54.859	357.930
	z_b	71.241	296.411	81.405	413.755
	z_c	12.250	62.927	10.529	45.475
	z_g	1.048	0.605	1.469	0.800
	z_n	1.445	0.415	1.918	0.534

Table 2.2: EKF exercise 2021 `run.py` RMSE and performance python vs MATLAB comparison with the same `SimulationConst.py` (see listing A.1) and `EstimatorConst.py` (see listing A.2) parameters. We made 1000 runs and measured the mean and variance of the RMSE as well as the execution time. Less is better.

	python		MATLAB	
	mean	variance	mean	variance
trackErrorNorm	1.336	0.128	1.096	0.060
angularErrorNorm	0.240	0.003	0.063	7.400e-4
velocityErrorNorm	0.140	0.002	0.100	0.002
windErrorNorm	0.440	0.031	0.399	0.023
biasErrorNorm	0.242	0.003	0.081	4.762e-4
Avg. exec time [s]	1.572	0.337	0.502	0.005

Table 2.3: Same as table 2.2 but for EKF exercise 2022. This exercise has the same `SimulationConst.py` (see listing A.1) and `EstimatorConst.py` (see listing A.2) parameters. Less is better.

	python		MATLAB	
	mean	variance	mean	variance
trackErrorNorm	2.288	0.372	1.923	0.207
angularErrorNorm	0.238	0.003	0.061	7.281e-4
velocityErrorNorm	0.172	0.005	0.103	0.004
windErrorNorm	0.430	0.030	0.209	0.005
biasErrorNorm	0.240	0.003	0.080	4.702e-4
Avg. exec time [s]	1.524	0.318	0.524	0.005

measurements at random, but also for A and B at the same time when/if the sensor C gets measured. If the noise/variance of the states measurements stay the same as for the exercise 2021. We will expect less accurate tracking of the boat, due to having less available measurements at each so time interval. Figure 2.3 depicts it very well. Now, there are jumps visible in the plots when the position measurements happened. In table 2.3, one notice the less accuracy from both implementations. Still, the python implementation is still less accurate, as well as 3x slower than the MATLAB version.

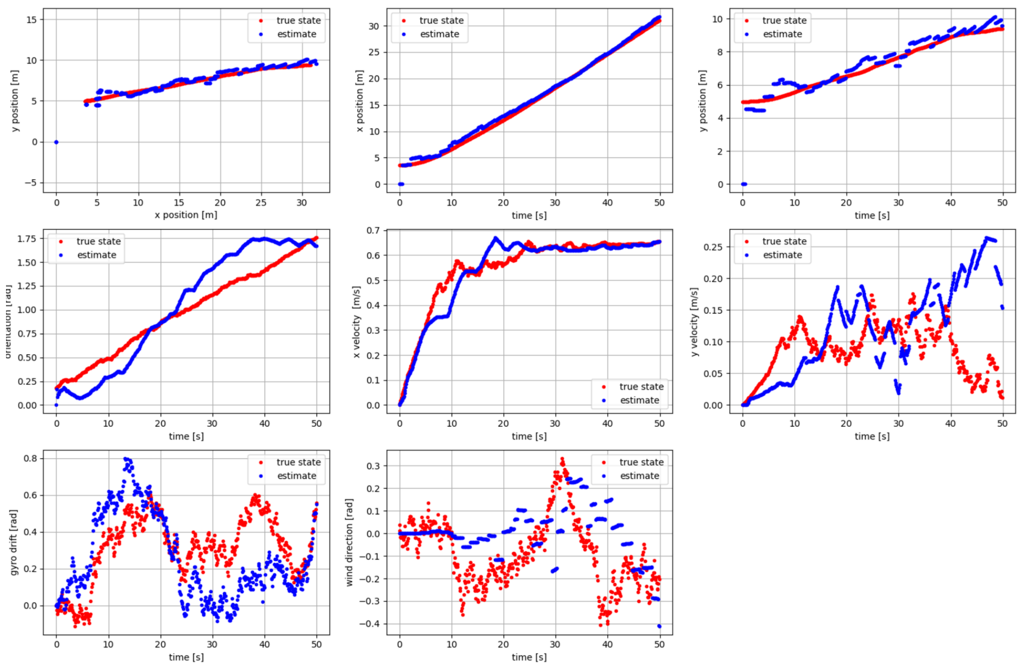
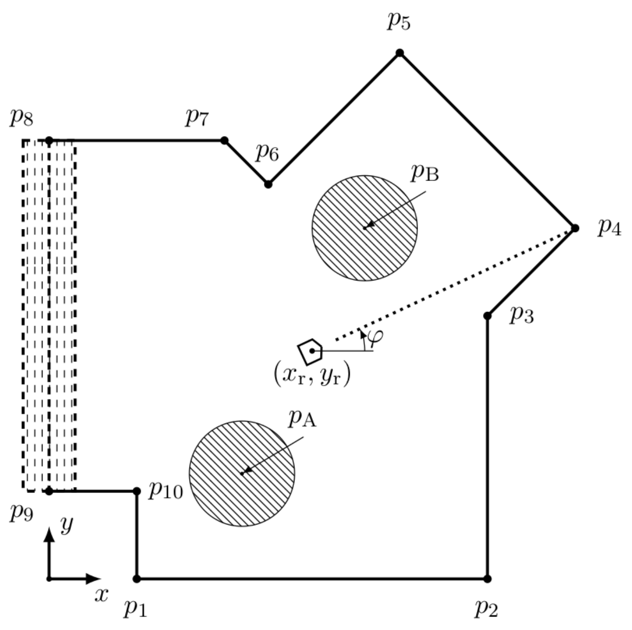


Figure 2.3: Same as fig. 2.2 but this time for the EKF exercise 2022. It uses the same set of `EstimatorConst.py` and `EstimatorConst.py` parameters.

Chapter 3

Particle Filter



Point	x[m]	y[m]
p_1	0.5	0.0
p_2	2.5	0.0
p_3	2.5	1.5
p_4	3.0	2.0
p_5	2.0	3.0
p_6	1.25	2.25
p_7	1.0	2.5
p_8	κ	2.5
p_9	κ	0.5
p_{10}	0.5	0.5
p_A	1.1	0.6
p_B	1.8	2.0

Figure 3.1: Sketch of the robot moving in a closed room with a partially known contour. The distance measurement is indicated by the dotted line between the robot and the opposing wall in front of it. The shaded circles denote the locations where the robot is initially located. The shaded rectangle on the left indicates the region where the unknown vertical wall is placed. The table to right are the coordinates of the contour points.

The Particle Filter exercise is the 2nd one from the programming exercise. The goal of this exercise is to track the movement of are robot in a closed room. Contour of the room is known except one wall. Figure 3.1 is the sketch with the table of the coordinates of contour points from the exercise sheet. For more details we refer to the exercise sheet from the Recursive Estimation lecture[1].

[1]https://idsc.ethz.ch/education/lectures/recursive-estimation.html

3.1 Implementation in python

In this exercise, we only needed 4 different python libraries: NumPy, Matplotlib, time and copy. We used the same kind of libraries as described in section 2.1, except no need for SciPy due to no ODE needed for this PF exercise. The whole python script is build up into specified files same as the MATLAB implementation. `Animation.py` is the animation function for creating the video of the moving robot with the particles from the PF. `PostParticles.py` is the class consisting the posterior update of the particles at a specific timestep. `run.py` is the function that is used to execute a simulation of the true system, run the estimator, plot the results, and report the root-mean squared tracking error. `Estimator.py` is the function template to be used for the implementation of the PF. It should be initially empty for the student. `Simulator.py` is the function used to simulate the motion of the robot and measurements. This function is called by `run.py`, and should be obfuscated (i.e. its source code is not readable) when given to the student. `EstimatorConst.py` is a class with the constants known to the estimator. `SimulationConst.py` is a class with the constants used for the simulation. These constants are not known to the estimator. Both of those classes are identical except that `SimulationConst.py` also consists the number of samples of the simulation (see listing A.3).

3.2 Performance & Accuracy Comparison

Like done in section 2.2, we could not recreate the same type of runs, due to the different type of RNGs of NumPy and MATLAB for the normal distributions. But not the `Simulator.py`, it only consists of uniform RNGs which meant easy checking for reproducibility, which we ended matching the output with the MATLAB version of `Simulator.m`. On the other hand `Estimator.py` consisted with some normal distributions. We made 1000 runs with different seeds to compare the accuracy and performance of python and MATLAB implementations. Table 3.1 shows our results of the `run.py` performance compared to the MATLAB version. This time the we are not that far off in accuracy, but around 9x times slower than MATLAB. The slowdown has to do with the big for-loop over all the 1000 generated particles in the `Estimator.py` script. You could, if possible, optimize the performance in python, if everything would be written in NumPy operators without the use of for-loops.

3.3 from PF 2021 to PF 2022 in python

We also implemented the PF 2022 exercise version into python, which also is a little bit different than the last years exercise. Shown in fig. 3.2, not only the

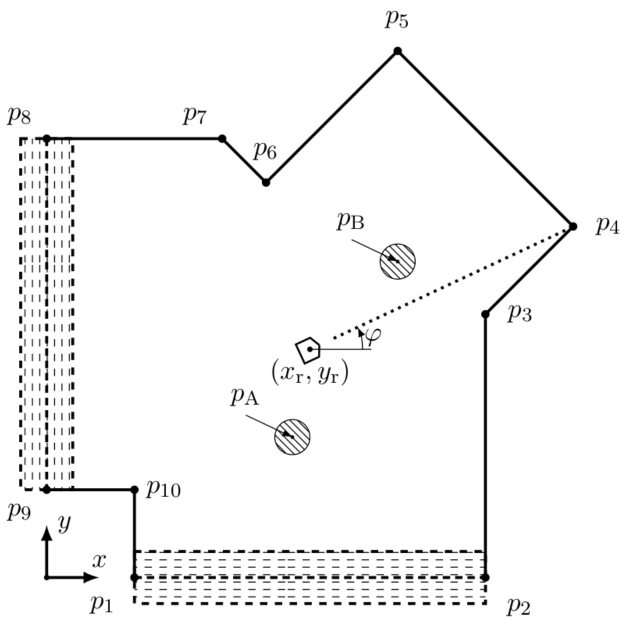
Table 3.1: 1000 runs of the PF exercise 2021 `run.py` with the RMSE of the trackErrorNorm and performance using the `SimulationConst.py` parameters from listing A.3 and 1000 particles.

	python		MATLAB	
	mean	variance	mean	variance
trackErrorNorm	1.122	0.116	0.928	0.132
Avg. exec time [s]	48.480	76.811	5.446	1.163

Table 3.2: 200 runs in python and 1000 runs in MATLAB of the PF exercise 2022 with the RMSE of the trackErrorNorm and performance using the `SimulationConst.py` parameters from listing A.4 and 1000 particles.

	python		MATLAB	
	mean	variance	mean	variance
trackErrorNorm	0.987	0.196	0.664	0.226
Avg. exec time [s]	34.891	0.285	5.255	0.017

left wall is unknown, now the bottom one as well. This makes it a little bit more challenging. The performance and accuracy are shown in table 3.2. Note that due to time reasons, we could not perform 1000 test runs in python. We ended up doing 200 runs. In first sight, one would think the RMSE got lower, this is due to a different set of parameters in `SimulationConst.py` from listing A.4. The python version is still worse than the MATLAB version in accuracy and performance. Python implementation is about 6x slower than MATLAB's implementation.



Point	x[m]	y[m]
p_1	0.5	ρ
p_2	2.5	ρ
p_3	2.5	1.5
p_4	3.0	2.0
p_5	2.0	3.0
p_6	1.25	2.25
p_7	1.0	2.5
p_8	κ	2.5
p_9	κ	0.5
p_{10}	0.5	0.5
p_A	1.4	0.8
p_B	2.0	1.8

Figure 3.2: Same as fig. 3.1 but for the 2022 exercise version the bottom wall is as well not known.

Chapter 4

Evaluation function for the lecture Recursive Estimation 2022

The Students, who participate in this exercise, can submit their code to the TAs. The TAs will then compare their code and the other competitors to see, who did the best implementation of the EKF and/or PF. Last year, the TAs already had a very good functioning evaluation function for both the exercises, but it did not conclude the overall accuracy of the EKF and PF over multiple different set of `EstimatorConst.py` and `SimulationConst.py` parameters, as well as the EKF overall error of all the five RMSE measurements. The last year implementation gave a spreadsheet of all the measured errors and execution time off all the submissions over all the different set of constant parameters.

The goal was to extend the evaluation function such that it makes an overall accuracy score of the submitted EKF implementation, same done for the PF submissions.

For the EKF evaluation we did the following. First take the mean of each RMSE over each set of constant parameters. The second step is normalizing all of the error separately over the columns. Like seen in the table 2.2 and **??** the errors are differently worst. By normalizing we mean normalizing over all submissions (including the master solution). In the last step we take the sum off all the normalized errors which call the 'overall error'. For the execution times, we only took the mean over all the variation set of constant parameters. Now when opening the spreadsheet, only the last three columns are the most important one. The 'mean Avg. Ex. Time of thread pool', the 'mean Avg. Ex. Time only the Estimator' and the 'overall error'. We measured the time two times different. The first one is measured over how long each thread pool thread takes to execute the `run.m` and the second one is the measurement of only the submitted `Estimator.m` performance. In our opinion, both measurements are fine.

For the PF evaluation function, we only took the error over all constant parameters, because there is only one type of error measurement in the PF exercise, hence no normalization needed. The last three columns in the spreadsheet are the same as for the EKF evaluation function.

Chapter 5

Conclusion

Overall, I am happy how it turned out. Some of the implementations are not as accurate as the MATLAB ones, but they are still good enough. It was a last minute decision to also EKF and PF exercise 2022 into python, due to this spontaneous decision, we were not really able to test their implementation thoroughly. Future TAs could maybe take a look, why all of the python implementations are not as accurate. The performance of python compared to MATLAB was expected to be slower, but this is not a big problem. What surprised me the most of the whole translating was, implementing a whole new class for the PF for making animations (see `Animation.py`).

Bibliography

[1] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, “Scipy 1.0: fundamental algorithms for scientific computing in python,” *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[2] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *IEEE Annals of the History of Computing*, vol. 9, no. 03, pp. 90–95, 2007.

[3] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, “Casadi: a software framework for nonlinear optimization and optimal control,” *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

Appendix A

Source Code

Listing A.1: EKF 2021/2022 SimulationConst class

```python
class SimulationConst:
    """ Boat dynamics constants """
    dragCoefficientHydr = 0.1 # C_d,h
    dragCoefficientAir = 0.06  # C_d,a
    rudderCoefficient = 2 # C_r
    windVel = 0.75  # C_w

    # control input constraints
    MaxThrust = 0.1
    MaxRudderCommand = 0.03

    # Minimum time for a segment, in seconds.
    # Should be multiple of sampling time.
    minSegTime = 0.5

    # Maximum time for a segment, in seconds.
    # Should be multiple of sampling time.
    maxSegTime = 2

    """ Radio measurement constants """
    pos_radioA = [-1000, 1000] #[x_a,y_a]
    pos_radioB = [2000, 0] #[x_b,y_b]
    pos_radioC = [0, 2000] #[x_c,y_c]

    """ Noise properties """
    # process noise
    # defined by its power spectral density Q_
    DragNoise = 0.1 # Q_d
    RudderNoise = 0.01 # Q_r
    WindAngleNoise = 0.01 # Q_rho
    GyroDriftNoise = 0.01 # Q_b

    # measurement noise
    # normally distributed with zero mean and variance \sigma_
    DistNoiseA = 20.0 # DistNoiseA = \sigma_a^2
    DistNoiseB = 20.0 # DistNoiseB = \sigma_b^2
    DistNoiseC = 5.0 # DistNoiseC = \sigma_c^2

    GyroNoise = 0.01 # \sigma_g^2

    CompassNoise = 0.5 #\sigma_n^2

    """ Starting point """
    # The boat start with equal probility in a cirlce of radius R0 around the
    # origin
    StartRadiusBound = 10.0 # R_0

    # The initial orientation is uniformly distributed in the range
    # [-\bar{\phi}, \bar{\phi}]
    RotationStartBound = np.pi/8.0 # \bar{\phi}

    # The initial wind direction is uniformly distributed in the range
    # [-\bar{\rho}, \bar{\rho}]
    WindAngleStartBound = np.pi/16.0 # \bar{\rho}

    # The initial gyro drift is exactly 0
    GyroDriftStartBound = 0.0

    """ Times """
    # Number of samples of the simulation, the total duration of the simulation
    # is then N*sampleContinuous in seconds.
    N = 500

    # The sample time for the continuous dynamics, in seconds.
    sampleContinuous = 0.1

    # The min sample time for distance measurement C, in seconds.
    sampleDistanceCmin = 0.5

    # The max sample time for distance measurement C, in seconds.
    sampleDistanceCmax = 2.0
```

Listing A.2: EKF 2021/2022 EstimatorConst class

```python
class EstimatorConst:
    """ Boat dynamics constants """
    dragCoefficientHydr = 0.1 #C_d,h
    dragCoefficientAir = 0.06 #C_d,a
    rudderCoefficient = 2.0 #C_r
    windVel = 0.75 # C_w

    """ Radio measuremtn constants """
    pos_radioA = [-1000, 1000] #[x_a, y_a]
    pos_radioB = [2000, 0] #[x_b, y_b]
    pos_radioC = [0, 2000] #[x_c, y_c]

    """ Noise properties """
    # process noise
    # defined by its variance Q_
    DragNoise = 0.1 # Q_d
    RudderNoise = 0.01 # Q_r
    WindAngleNoise = 0.01 # Q_rho
    GyroDriftNoise = 0.01 # Q_b

    # measurement noise
    # normally distributed with zero mean and variance \sigma_
    DistNoiseA = 20.0 # DistNoiseA = \sigma_a^2
    DistNoiseB = 20.0 # DistNoiseB = \sigma_b^2
    DistNoiseC = 5.0 # DistNoiseC = \sigma_c^2

    GyroNoise = 0.01 # \sigma_g^2

    CompassNoise = 0.5 #\sigma_n^2

    """ Starting point """
    # The boat start with equal probility in a circle of radius R0 around the
    # origin
    StartRadiusBound = 10.0 # R_0

    # The initial orientation is uniformly distributed in the range
    # [-\bar{\phi}, \bar{\phi}]
    RotationStartBound = np.pi/8.0 # \bar{\phi}

    # The initial wind direction is uniformly distributed in the range
    # [-\bar{\rho}, \bar{\rho}]
    WindAngleStartBound = np.pi/16.0 # \bar{\rho}

    # The initial gyro drift is exactly 0
    GyroDriftStartBound = 0.0
```

Listing A.3: PF 2021 SimulationConst class

```
class SimulationConst:
    """ The wall contour - (x,y) coordinates of corners as in Table 1 """
    contour = np.array([[0.50, 0.00],
                        [2.50, 0.00],
                        [2.50, 1.50],
                        [3.00, 2.00],
                        [2.00, 3.00],
                        [1.25, 2.25],
                        [1.00, 2.50],
                        [0.00, 2.50],
                        [0.00, 0.50],
                        [0.50, 0.50]])

    """ Initialization """
    pA = [1.1, 0.6] # Center point pA of the initial position distribution
    pB = [1.8, 2.0] # Center point pB of the initial position distribution
    d = 0.2  # Radius of shaded regions for initialization

    phi_0 = np.pi/4.0 # Initial heading is uniformly distr. in [-phi_0,phi_0]

    l = 0.2  # Uniform distribution parameter of points p9 and p10

    """ Noise properties """
    # process noise
    sigma_phi = 0.05 # Parameter for process noise v_phi
    sigma_f = 0.01 # Parameter for process noise v_f

    # measurement noise
    epsilon = 0.01 # Parameter for measurement noise w

    """ Times """
    # Number of samples (discrete time steps) of the simulation.
    N = 500
```

Listing A.4: PF 2022 SimulationConst class

```
class SimulationConst:
    """ The wall contour - (x,y) coordinates of corners as in Table 1 """
    contour = np.array([[0.50, 0.00],
                        [2.50, 0.00],
                        [2.50, 1.50],
                        [3.00, 2.00],
                        [2.00, 3.00],
                        [1.25, 2.25],
                        [1.00, 2.50],
                        [0.00, 2.50],
                        [0.00, 0.50],
                        [0.50, 0.50]])

    """ Initialization """
    pA = [1.4, 0.8] # Center point pA of the initial position distribution
    pB = [2.0, 1.8] # Center point pB of the initial position distribution
    d = 0.1  # Radius of shaded regions for initialization

    phi_0 = np.pi/3.0 # Initial heading is uniformly distr. in [-phi_0,phi_0]

    m = 0.1  # Uniform distribution parameter of points p1 and p2
    l = 0.2  # Uniform distribution parameter of points p9 and p10

    """ Noise properties """
    # process noise
    sigma_phi = 0.05 # Parameter for process noise v_phi
    sigma_f = 0.01 # Parameter for process noise v_f

    # measurement noise
    epsilon = 0.01 # Parameter for measurement noise w

    """ Times """
    # Number of samples (discrete time steps) of the simulation.
    N = 500
```