Active Inference and Kalman filter design applied to Jackal robot simulation

Dennis Benders and Martijn Wisse

***Abstract*— The goal of this paper is to contribute to the existing knowledge in the field of Active Inference (AI). AI is originally developed in the neuroscience field and recently mapped to the control engineering. This way of controlling systems should perform well in environments under the presence of coloured process and output noise. Until now, little practical experience exists when using this theory to control physical systems. Before controlling a physical system it is preferable to first have the whole setup working in a simulation environment. Therefore, this project considers the simulation of a skid-steering robot, called the Jackal robot, in the Gazebo simulator. An AI filter is developed of which the goal is to perform better than the benchmark: the Kalman filter. It can be concluded that the complete simulation and filter setup has been created in a Robot Operating System (ROS) environment. Further research is needed regarding the construction of the precision matrices, learning rate and prior in the AI filter. The Kalman filter performance is given by its Mean-Squared Error (MSE) with respect to the simulation data and can be used to compare with the AI filter in the future.**

Fig. 1. Jackal robot [2]

I. INTRODUCTION

In the neuroscience field a new theory has recently been developed, called Active Inference (briefly denoted as AI). This theory has the potential to become very useful within the field of control engineering. However, only a very limited amount of practical implementations exist at the moment, so still a lot of practical work has to be performed before practical conclusions can be drawn regarding the performance of the AI algorithm.

This paper contributes to the existing knowledge by analysing part of the AI algorithm: the filter. The final goal would be to compare the AI filter with a benchmark filter: the Kalman filter. This paper will not perform the comparison, but rather explains the design choices and implementation of both filters and leave the comparison as future work.

According to the expectations [1], the AI filter will perform well on state estimation when the system is subject to coloured (differentiable) noise, because it derives information from multiple derivative orders of the system state (thereby capturing the noise properties). This assumption can be quite useful in practice, since natural processes are assumed to produce coloured noise, which is added to the sensor signals of a physical system to be controlled.

A good system to test this filter on is the Jackal robot [2]. The Jackal robot (as shown in Figure 1) is a skid-steering robot, meaning that the wheels on one side of the robot have equal velocity. For example, the robot can turn left when the wheels on the left side have less rotational velocity in forward direction than the wheels on the right side. As one may notice, rotating this robot introduces quite some mechanical vibrations, which introduces a noisy sensor (Inertial Measurement Unit, abbreviated as IMU) signal.

Before testing the filter on the real Jackal robot, the filter should first be tested using a system simulation. Gazebo is used as simulation program as this is supported by the Jackal robot developers (Clearpath Robotics Inc.). One of the main contributions of this paper will thus be to analyse the Gazebo simulation and derive the necessary information needed to implement the AI and Kalman filters.

To summarise, the main contributions of this paper will be the analysis of the Jackal robot rotational velocity behaviour in a Gazebo simulation. Furthermore, the AI and Kalman filter frameworks will be designed, implemented and validated using the information derived from the Gazebo simulation.

This paper will be organised as follows: Section II describes all system requirements needed to fulfil the goals of this paper. Section III provides the design choices regarding the system setup, Jackal robot model, AI filter framework and Kalman filter framework. Thereafter, Section IV will describe the implementation and validation of the different components designed in Section III. The validation part of this section immediately describes the most important experimental results obtained during the project. Section V summarises the main conclusions of this paper and discusses the remaining issues and corresponding future work. As this paper is the results of a project, Section VI will finally describe the main project achievements.

II. REQUIREMENTS

This section describes all requirements applicable in case of the system used to create the results obtained in Section

IV. First of all, requirements exist regarding the software to be used. These are listed below:

1) The Jackal robot should be used as test system for both filters, because the strong points of the AI filter (performing state estimation under the effect of coloured noises) can be tested well with this robot
2) Gazebo 7 should be used as simulation program, because it is supported by Clearpath Robotics Inc. in case of the Jackal robot
3) ROS Kinetic should be used to control the simulation, because it is compatible with Gazebo 7 and the standard software framework for simulation experiments within the Cognitive Robotics department at TU Delft
4) Ubuntu 16.04 LTS should be used as operating system, because it is supported by ROS Kinetic
5) Python or C++ should be used as scripting language, because these are supported by ROS
6) MATLAB may be used to perform short/quick experiments with, because most students are more familiar with MATLAB than Python within the Active Inference group at TU Delft

III. DESIGN CHOICES

This section describes the design of different components needed to analyse the Jackal behaviour in the Gazebo simulation and to implement the AI and Kalman filter. It will be organised in a bottom-up fashion, so the filters will first be described in Sections III-A and III-B in order to know what kind of content these filters need. Thereafter, a small theoretical comparison of the filters is given in Section III-C, although it is not part of the main goal of this paper. Finally, the rest of the system design will be described following the information need of both filters. This includes the linear system model, designed in Section III-D, and the noise properties of the data, provided in Section III-E. From these designs, the necessary simulation data can be derived, which is used to come up with a proper ROS layout (containing all necessary ROS nodes to run the simulation and filters), as described in Section III-F.

A. AI filter

In order to derive the AI filter [1], several AI concepts need to be explained. First of all, the usage of the Free-Energy principle is shortly explained in Section III-A.1. Thereafter, the agent (combination of filter and controller) is described in Section III-A.2. Section III-A.3 provides the incorporation of noise information in the AI framework. Section III-A.4 describes the update rule being used to get a new state estimate every time the AI filter is run. Finally, Section III-A.5 gives an overview of the AI framework and puts Sections III-A.1 until III-A.4 in context.

1) Free-Energy principle: As already mentioned, Active Inference is a theory based on recent scientific results obtained in the neuroscience. It is based on the so-called Free-Energy principle. This principle essentially means that humans are able to make more accurate predictions while they are interacting with their continuously changing environment. When mapping this principle to the control engineering, it boils down to optimising (minimising) a cost function in order to estimate the system state and to calculate the next control input that cause the predictions of the environment states as accurate as possible. The AI algorithm thus consists of two parts: a filter and a controller. Both are based on an optimisation process of the Free-Energy.

It can be shown that the Free-Energy can be written as:

$$\mathcal{F}(\boldsymbol{\mu}, \boldsymbol{y}) = \frac{1}{2}\left(\boldsymbol{\epsilon}_{\mu}^{T}\Pi_{w}\boldsymbol{\epsilon}_{\mu} + \boldsymbol{\epsilon}_{y}^{T}\Pi_{z}\boldsymbol{\epsilon}_{y}\right) \tag{1}$$

This equation will be used in Section III-A.4 to determine the state estimation update rule.

2) Agent: In the AI framework, the combination of filter and controller is called the agent. The agent's goal is to control the physical (or simulated) system as well as possible. The system model is described in Section III-D. In order to do so, the agent keeps a generative model up-to-date that resembles the physical (or simulated) system as well as possible. The generative model is given by the following generalised state and output equation:

$$\mathcal{D}\boldsymbol{\mu} = \tilde{A}\boldsymbol{\mu} + \boldsymbol{\xi} \tag{2}$$

$$\tilde{\boldsymbol{y}} = \tilde{C}\boldsymbol{\mu} \tag{3}$$

where:

- $\mathcal{D} \in \mathbb{R}^{(1+p)\cdot n_x \times (1+p)\cdot n_x}$ is the derivative (block) matrix operator and given by equation (4)
- $\boldsymbol{\mu} \in \mathbb{R}^{(1+p)\cdot n_x \times 1}$ is the expectation of the generalised state (block) vector and given by equation (5)
- $\tilde{\boldsymbol{y}} \in \mathbb{R}^{(1+p)\cdot n_y \times 1}$ is the generalised output (block) vector and given by equation (6)
- $\boldsymbol{\xi} \in \mathbb{R}^{(1+p)\cdot n_x \times 1}$ is the prior. The prior can be seen as an implementation of intelligence (obtained by experience), which is used as prior knowledge for state estimation and control input calculation. When controlling the system, it acts like a reference signal
- $\tilde{A} \in \mathbb{R}^{(1+p)\cdot n_x \times (1+p)\cdot n_x}$ is the generalised state (block) matrix and given by equation (7)
- $\tilde{C} \in \mathbb{R}^{(1+p)\cdot n_y \times (1+p)\cdot n_x}$ is the generalised output (block) matrix and given by equation (8)

$$\mathcal{D} = \left[\begin{array}{c|c|c|c|c} 0_{n_x \times n_x} & \mathrm{I}_{n_x \times n_x} & \dots & 0_{n_x \times n_x} & 0_{n_x \times n_x} \\ \hline 0_{n_x \times n_x} & 0_{n_x \times n_x} & \dots & 0_{n_x \times n_x} & 0_{n_x \times n_x} \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline 0_{n_x \times n_x} & 0_{n_x \times n_x} & \dots & 0_{n_x \times n_x} & \mathrm{I}_{n_x \times n_x} \\ \hline 0_{n_x \times n_x} & 0_{n_x \times n_x} & \dots & 0_{n_x \times n_x} & 0_{n_x \times n_x} \end{array}\right] \tag{4}$$

$$\boldsymbol{\mu} = E[\tilde{\boldsymbol{x}}] = E[\left[\begin{array}{c} \boldsymbol{x} \\ \hline \boldsymbol{x}' \\ \hline \vdots \\ \hline \boldsymbol{x}^{(p-1)} \\ \hline \boldsymbol{x}^{(p)} \end{array}\right]] \tag{5}$$

$$\tilde{\boldsymbol{y}} = \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y}' \\ \vdots \\ \boldsymbol{y}^{(p-1)} \\ \boldsymbol{y}^{(p)} \end{bmatrix} \tag{6}$$

$$\tilde{A} = \begin{bmatrix} A & 0_{n_x\times n_x} & \dots & 0_{n_x\times n_x} & 0_{n_x\times n_x} \\ 0_{n_x\times n_x} & A & \dots & 0_{n_x\times n_x} & 0_{n_x\times n_x} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0_{n_x\times n_x} & 0_{n_x\times n_x} & \dots & A & 0_{n_x\times n_x} \\ 0_{n_x\times n_x} & 0_{n_x\times n_x} & \dots & 0_{n_x\times n_x} & A \end{bmatrix} \tag{7}$$

$$\tilde{C} = \begin{bmatrix} C & 0_{n_y\times n_x} & \dots & 0_{n_y\times n_x} & 0_{n_y\times n_x} \\ 0_{n_y\times n_x} & C & \dots & 0_{n_y\times n_x} & 0_{n_y\times n_x} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0_{n_y\times n_x} & 0_{n_y\times n_x} & \dots & C & 0_{n_y\times n_x} \\ 0_{n_y\times n_x} & 0_{n_y\times n_x} & \dots & 0_{n_y\times n_x} & C \end{bmatrix} \tag{8}$$

The following dimension notations are used:

- p is the embedding order of the generalised states and outputs
- n_x is the system state dimension
- n_u is the system control input dimension
- n_y is the system output dimension

From equations (2) and (3), $\boldsymbol{\epsilon}_\mu$ and $\boldsymbol{\epsilon}_y$ can be derived. These quantities are given by the following two expressions:

$$\boldsymbol{\epsilon}_\mu = \mathcal{D}\boldsymbol{\mu} - \tilde{A}\boldsymbol{\mu} + \boldsymbol{\xi} \tag{9}$$

$$\boldsymbol{\epsilon}_y = \tilde{\boldsymbol{y}} - \tilde{C}\boldsymbol{\mu} \tag{10}$$

Note the $\tilde{}$-symbol in the equations described above. This symbol is used to indicate the usage of generalised motions. This means that not only the states, but also its derivatives until and including order p are used in the generative model to capture the coloured noise properties (see Section III-A.3). By taking these derivatives into account (as can be seen in equations (5) and (6)), a more accurate state estimate and control input can be calculated. Usually, noise information is useful until order six, so $p = 6$ is chosen to be used.

3) Noise modelling: Equation (1) includes Π_w and Π_z, the so-called precision matrices. These matrices weight the errors made by the agent. The process error, $\boldsymbol{\epsilon}_\mu$, indicates the difference between estimated generalised state and the prior and is weighted by Π_w. The output error, $\boldsymbol{\epsilon}_y$, indicates the difference between estimated generalised system output and real system output and is weighted by Π_z.

The actual system model is given by the generative process state and output equations:

$$\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u} + \boldsymbol{w} \tag{11}$$

$$\boldsymbol{y} = C\boldsymbol{x} + \boldsymbol{z} \tag{12}$$

When making these equations generalised, replacing the term $B\boldsymbol{u}$ by $\boldsymbol{\xi}$ and removing the noise terms $\boldsymbol{w}$ and $\boldsymbol{z}$, the generative model is obtained. Important is the difference in the equations caused by the state and output noises $\boldsymbol{w}$ and $\boldsymbol{z}$. A significant part of the error $\boldsymbol{\epsilon}_\mu$ is determined by the properties of the process noise $\boldsymbol{w}$. The same holds for $\boldsymbol{\epsilon}_y$ and properties of the output noise $\boldsymbol{z}$.

It is thus intuitive that the the precision matrices contain information of the noises in some way or another. However, more specific details of the noise should be specified to be able to construct the precision matrices. [3], [4] and [5] are used to set up the precision matrices for the case where the coloured noise signals (for both state and output) can be represented by a convolution of a white noise signal (represented by its mean μ and covariance matrix Σ) with a Gaussian filter having a certain kernel width s. To summarise:

$$\boldsymbol{\omega_w} \sim \mathcal{N}(\mu_w, \Sigma_w) \tag{13}$$

$$\boldsymbol{w} = \boldsymbol{\omega_w} * h_w(t) \tag{14}$$

$$\boldsymbol{\omega_z} \sim \mathcal{N}(\mu_z, \Sigma_z) \tag{15}$$

$$\boldsymbol{z} = \boldsymbol{\omega_z} * h_z(t) \tag{16}$$

where:

$$h_a(t) = \sqrt{\frac{\Delta t}{s_a\sqrt{\pi}}} \cdot e^{\frac{-t^2}{2s_a^2}} \tag{17}$$

Using this noise information, the precision matrix for the coloured process noise can be written as:

$$\begin{aligned} \Pi_w &= \mathcal{S}(s_w^2)^{-1} \otimes \Sigma_w^{-1} \\ &= \begin{bmatrix} 1 & 0 & -\frac{1}{2s_w^2} \\ 0 & \frac{1}{2s_w^2} & 0 \\ -\frac{1}{2s_w^2} & 0 & \frac{3}{4s_w^2} \end{bmatrix}^{-1} \otimes \begin{bmatrix} \frac{1}{\sigma_{w_1}^2} & 0 & \dots & 0 & \\ 0 & \frac{1}{\sigma_{w_2}^2} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \frac{1}{\sigma_{w_{n_x}}^2} \end{bmatrix} \end{aligned} \tag{18}$$

Note: equation (18) is for notation convenience written for the case where $p = 2$ (meaning that only the 1^{st}- and 2^{nd}-order derivatives of the states are taken into account in the generalised state). Π_z can be constructed in a similar way.

4) State estimation update rule: Now that all terms in equation (1) have been explained, this function can be minimised. Fortunately, minimising the Free-Energy is a convex optimisation problem. Therefore, the gradient descent method can be used to calculate both state estimate and control input, causing the Free-Energy to ideally decrease to its global minimum. As already mentioned, only the AI filter is discussed in the paper, so this section only covers the state estimation update rule. This update rule is given by:

$$\dot{\boldsymbol{\mu}} = \mathcal{D}\boldsymbol{\mu} - \alpha_\mu \frac{\partial \mathcal{F}(\boldsymbol{\mu}, \boldsymbol{y})}{\partial \boldsymbol{\mu}} \tag{19}$$

By working out $\frac{\partial \mathcal{F}(\boldsymbol{\mu},\boldsymbol{y})}{\partial \boldsymbol{\mu}}$ using the chain rule (details are explained in [1]), the state estimation update rule is given by:

$$\dot{\boldsymbol{\mu}} = \mathcal{D}\boldsymbol{\mu} \\ - \alpha_\mu \left((\mathcal{D} - \tilde{A})^T \Pi_w (\mathcal{D}\boldsymbol{\mu} - \tilde{A}\boldsymbol{\mu} - \boldsymbol{\xi}) - \tilde{C}^T \Pi_z (\tilde{\boldsymbol{y}} - \tilde{C}\boldsymbol{\mu}) \right) \tag{20}$$

5) AI framework for filtering and control: Figure 2 shows the general AI framework used to create a closed-loop control loop with a physical system, applied to the Jackal robot simulation in Gazebo. From this figure, a few things can be concluded.

First of all, a general control loop usually consists of three parts: a physical system to be controlled, a filter and a controller. This also holds for the AI framework. The physical system is represented by the top part of the figure, the AI filter by the bottom-left part and the AI controller by the bottom-right part. However, in this case, there is no physical system to be controlled, so the top part of the figure becomes the Gazebo simulation (as indicated in the figure). The AI filter maintains its functionality. Furthermore, the AI controller is not used in this case, since only the filter part is analysed and not the closed-loop AI control behaviour. It can thus be concluded that the system involved in this paper only consists of the 'Gazebo' and 'Filter' part, which is open-loop.

Secondly, one may notice that the Gazebo simulation information should somehow be incorporated in the AI filter. This entails the linear system model (represented by system matrices A, B and C, which are derived in Section III-D and used in the generative model as described in Section III-A.2). This also entails the process noise w and output noise z. As Figure 2 shows, the coloured process and output noise are constructed by filtering (convoluting) a white noise signal with standard deviation σ with a Gaussian filter ($H(s)$, in time-domain denoted as $h(t)$), which is in accordance with Section III-A.3.

Thirdly, one can see that inside the filter, first the Free-Energy is minimised using the gradient descent algorithm to obtain the new value for the generalised state estimate $\boldsymbol{\mu}$ using information of the prior, as given by equation (20). Note, however, that equation (20) only provides the formula for the rate of instantaneous change of $\boldsymbol{\mu}$. Therefore, it should also be integrated to get the real value of $\boldsymbol{\mu}$, which is indicated in the figure. Section IV-F provides the exact implementation of this block.

B. Kalman filter

For the Kalman filter, the conventional linear, time-varying design approach is taken [6], because the simulated system is modelled linearly. Using the assumptions that the process is stationary and the control input is constant during each sampling period, the Kalman filter can be written in discrete form, consisting of two stages: predicting and

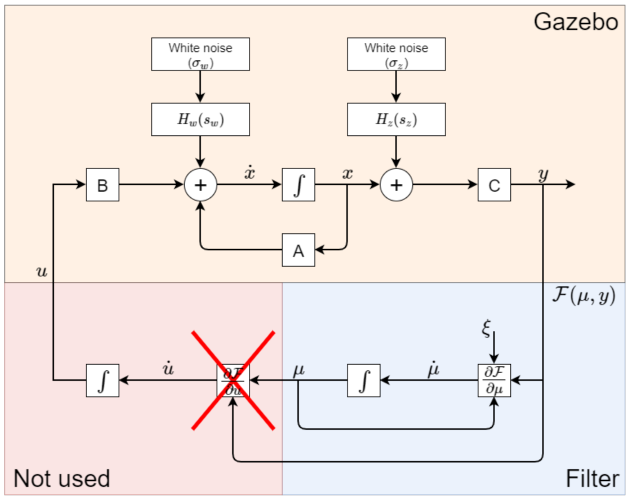

Fig. 2. General AI framework applied to the Jackal simulation in Gazebo

updating. These will each be described in the following sections.

1) Prediction stage: The purpose of the prediction stage is to estimate the system states at discrete timestep k without having measured this quantity at timestep k. Only measurements of timestep $k-1$ and earlier are available. Given the fact that the simulated system will be represented by a Markov Decision Process (MDP), only the system state at time $k-1$ is needed to predict the current state value. Furthermore, the Kalman filter takes the uncertainty of those state estimates into account by modelling the process and output noise as white noise. Therefore, the covariance matrix of the noises (as will be derived in Section III-E) should be given as input argument to the Kalman filter.

The outputs of this stage are:

- $\hat{\boldsymbol{x}}_{k|k-1} \in \mathbb{R}^{n_x \times 1}$: estimate of the system state at the current time instance given the measurement value of the system state at the previous time instance
- $P_{k|k-1} \in \mathbb{R}^{n_x \times n_x}$: covariance matrix of the state estimate at the current time instance given the covariance matrix of the state estimate at the previous time instance. This matrix indicates the uncertainty of the state estimate: the lower the values in the matrix, the lower the uncertainty and the better the Kalman filter predicts

$\hat{\boldsymbol{x}}_{k|k-1}$ is calculated as follows:

$$\hat{\boldsymbol{x}}_{k|k-1} = \Gamma_k \hat{\boldsymbol{x}}_{k-1|k-1} + \Theta_k \boldsymbol{u}_{k-1} \tag{21}$$

$P_{k|k-1}$ is given by the following equation:

$$P_{k|k-1} = \Gamma_k P_{k-1|k-1} \Gamma_k^T + Q_k \tag{22}$$

Note that Γ_k and Θ_k are the time-varying system matrices, discretised at time instance k. Given the fact that the system model is Linear and Time-Invariant (LTI), Γ_k and Θ_k will be replaced with Γ and Θ from now on. The same holds for Q_k (the process noise covariance matrix): the process noise covariance is also assumed to be time-invariant. Therefore, Q_k will be replaced with Q.

2) Update stage: The update stage can be executed once the current system output $\boldsymbol{y}_k$ is measured. It uses the information of the output noise and the system measurement itself to update the state estimate and its corresponding covariance matrix by calculating the so-called innovation, the innovation covariance and the optimal Kalman gain, respectively.

The innovation signal is given by the difference between the real system output and the estimated system output. At timestep k, the innovation pre-fit residual equals:

$$\tilde{\boldsymbol{y}}_k = \boldsymbol{y}_k - C_{d,k}\hat{\boldsymbol{x}}_{k|k-1} \tag{23}$$

In this equation $C_{d,k}$ is the discretised C matrix of the linear system model at timestep k. Also for this matrix holds that subscript k will be omitted further on, because the system model is LTI.

The innovation covariance at timestep k is given by:

$$S_k = C_d P_{k|k-1} C_d^T + R_k \tag{24}$$

In this equation R_k represents the output noise covariance matrix.

The optimal Kalman gain at timestep k can now be calculated using the innovation covariance and equals:

$$K_k = P_{k|k-1} C_d^T S_k^{-1} \tag{25}$$

Using this information, the new updated state estimate can be constructed:

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + K_k \tilde{\boldsymbol{y}}_{\boldsymbol{k}} \tag{26}$$

The updated estimate covariance matrix given is now given by:

$$P_{k|k} = (I_{n_x} - K_k C_d) P_{k|k-1} \tag{27}$$

where I_{n_x} is the identity matrix with size n_x.

C. Comparison of AI and Kalman filter

When comparing the Kalman filter and the AI filter, a few conclusions can be drawn.

First of all, both filters use the same LTI state-space model to base their estimations on.

There exists a difference regarding the inputs both filters get, however. Most important inputs in this paper are the noise properties. The AI filter takes into account the coloured noise properties, which are used for weighting the process and output error in the Free-Energy function. The Kalman filter, on the other hand, assumes w and z to be white. One could say that the AI filter is better informed about the circumstances under which the system is operating. Therefore, it can be expected that the AI filter is able to perform better when it comes to state estimation under the presence of coloured noise. However, the Kalman filter gets the control input as input argument, which is replaced by the prior in the AI filter. Since the control input is constant and the system will be linearised around this constant operating point, it is not expected that the real control input gives a bigger advantage than the prior, provided that the prior is a realistic value. It can thus be concluded that the AI filter will probably be able to perform better than the Kalman filter based on the inputs they get.

Another difference between the filters is the way the state estimation is performed. The AI filter performs state estimation by minimising the Free-Energy function, thereby minimising the prediction error with respect to the prior (which acts as pre-knowledge on the system and could therefore be seen as a kind of intelligence, as mentioned before) and with respect to the real system output. Using Π_w and Π_z it balances between the two signals. On the other hand, the Kalman filter performs state estimation using an observer structure where the Kalman gain ensures that the observer is asymptotically stable.

D. Linear system model

As Figure 2 shows, the filters treat the simulation data as if it contained the results of simulating a LTI state-space model of the system. This is of course not the case. In fact, Gazebo uses a physics engine to calculate all necessary model states at a relatively high frequency. In case of Gazebo 7, this frequency defaults to 1 kHz. It means that somehow the LTI state-space model should be fitted to resemble the Gazebo simulation as well as possible, so the filters can properly handle the data coming from the Gazebo simulation. The main goal of this section is to make clear how the linear system model should be designed in order to match the simulation results. Section III-D.1 provides the LTI state-space model to be used and Section III-D.2 describes the tuning of the parameters from the state-space equation in order to let the linear model fit the Gazebo results as much as possible.

1) LTI state-space model: Let's first start with the linear system model for the Jackal robot, which is given by [7]:

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{\phi}\end{bmatrix} = \begin{bmatrix}\frac{-4d_{long}}{m} & 0 & 0\\ 0 & \frac{-4d_{lat}}{m} & 0\\ 0 & 0 & \frac{-4}{I}(b^2 d_{long} + a^2 d_{lat})\end{bmatrix}\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\phi}\end{bmatrix} + \begin{bmatrix}\frac{2d_{long}r_{wheel}}{m} & \frac{2d_{long}r_{wheel}}{m}\\ 0 & 0\\ \frac{2bd_{long}r_{wheel}}{I} & -\frac{2bd_{long}r_{wheel}}{I}\end{bmatrix}\begin{bmatrix}\omega_r\\ \omega_l\end{bmatrix} \tag{28}$$

where:

- $\dot{x} \in \mathbb{R}$ is the robot's linear longitudinal velocity
- $\dot{y} \in \mathbb{R}$ is the robot's linear lateral velocity
- $\dot{\phi} \in \mathbb{R}$ is the robot's angular velocity around the z-axis
- $\omega_r \in \mathbb{R}$ is the rotational velocity of the wheels on the right side of the robot. Remember that the velocities of both wheels on one side of the robot are equal

Parameter	Value
a	$0.131\ \mathrm{m}$
b	$0.2078\ \mathrm{m}$
m	$18.431\ \mathrm{kg}$
I	$0.5485\ \mathrm{kg\,m^2}$
r_{wheel}	$0.098\ \mathrm{m}$

TABLE I
MEASURABLE CONSTANTS IN EQUATION 29

- $\omega_l \in \mathbb{R}$ is the rotational velocity of the wheels on the left side of the robot
- $a \in \mathbb{R}$ is half of the robot's wheel base
- $b \in \mathbb{R}$ is half of the robot's track width
- $m \in \mathbb{R}$ is the robot's mass
- $I \in \mathbb{R}$ is the robot's total moment of inertia around the z-axis (including the body and the four wheels)
- $r_{wheel} \in \mathbb{R}$ is the radius of the robot's wheels
- $d_{long} \in \mathbb{R}$ is the longitudinal viscous damping coefficient used to model the contact point between wheel and ground
- $d_{lat} \in \mathbb{R}$ is the lateral viscous damping coefficient used to model the contact point between wheel and ground

The final goal would be to perform state estimation on the real Jackal robot (not the Gazebo simulation). As mentioned before, this is outside the scope of this paper, but this fact is taken into account here. The main source of the LTI model states when driving the physical robot is the IMU. The IMU only outputs linear accelerations and rotational velocities related to the states provided in equation (28). Unfortunately, equation 28 is partially observable, because the linear velocities can not directly be measured and observed. As can be concluded from this state-space model, the states are decoupled, meaning that each state could be modelled independently from the other states. This gives the possibility to make the system somewhat simpler, observable and hence easier to analyse by only taking care of the rotational velocity around the z-axis of the robot (denoted by $\dot{\phi}$). By reducing the system dimensionality using this approach, the following state-space equations (state and output equation) are obtained:

$$\begin{aligned}\left[\ddot{\phi}\right] &= \left[\tfrac{-4}{I}(b^2 d_{long} + a^2 d_{lat})\right]\left[\dot{\phi}\right] \\ &\quad + \begin{bmatrix}\frac{2bd_{long}r_{wheel}}{I} & -\frac{2bd_{long}r_{wheel}}{I}\end{bmatrix}\begin{bmatrix}\omega_r \\ \omega_l\end{bmatrix}\end{aligned} \tag{29}$$

$$\left[\dot{\phi}\right] = \left[1\right]\left[\dot{\phi}\right] \tag{30}$$

Note that this system model is observable: the output is equal to the state. Furthermore, note that this equation contains some measurable constants and some tunable parameters. Table I provides the constants belonging to the Jackal robot [7]:

What remains are the tunable parameters d_{long} and d_{lat}. These parameters can be used to match the linear system with the Gazebo simulation results, which will be described in the next section.

2) Parameter tuning: An easy and effective way to match the model with the simulation results is by solving the least-squares problem given in equation (31), thereby minimising the Mean-Squared Error (MSE) between the Gazebo simulation and MATLAB simulation. The linear system will be simulated in MATLAB and its results will be compared with the Gazebo simulation results for each combination of realistic values for d_{long} and d_{lat}. The range to use is recommended to be [1, 1000] [7]. However, to make sure the minimum does not lie just outside this range, the range is taken to be [1, 1500]. The comparison should be performed using the same control input signal and at the same simulation frequency (1 kHz). The motion (which can be translated to the control input) used to tune the model parameters is the following: ($\dot{x}_{lin} = 0.3\mathrm{m\,s^{-1}}$, $\dot{\phi}_z = 22.5^\circ\,\mathrm{s^{-1}}$), which is also used in [7].

Note that here a linear system will be fitted to simulation results which show non-linear behaviour (as can be concluded from Figure 5 in Section IV, for instance). However, in general a linear system can only account for small variations in the neighbourhood of a certain operating point around which a non-linear system model is linearised. In this case only a linear system model is available, but it still holds that this system description becomes increasingly inaccurate when using it for operating points farther away from the one it is optimised for. Therefore, the operating point will be determined by taking the average values of the inputs and outputs that are recorded from the Gazebo simulation. d_{long} and d_{lat} will be tuned in such a way that the linear system can describe as much of the variations around that predefined operating point as possible. For further simulation results, always the same motion is used, such that the linear system optimised for that point can always be used.

$$\begin{aligned}\underset{d_{long},d_{lat}}{\text{minimise}} \quad & \frac{1}{N-1}\sum_{k=1}^{N-1}\left(\dot{\phi}_{k+1} - \Gamma\dot{\phi}_k - \Theta\begin{vmatrix}\omega_{r,k} \\ \omega_{l,k}\end{vmatrix}\right)^2 \\ \text{subject to} \quad & d_{long} \in [1, 1500] \\ & d_{lat} \in [1, 1500]\end{aligned} \tag{31}$$

where:

- N is the amount of input-output samples obtained from the Gazebo simulation
- $\dot{\phi}_k$ is the output sample of the Gazebo simulation at timestep k
- $\omega_{r,k}$ is one of the input samples (rotational velocity of the wheels on the right side of the robot) of the Gazebo simulation at timestep k
- $\omega_{r,k}$ is one of the input samples (rotational velocity of the wheels on the left side of the robot) of the Gazebo simulation at timestep k
- Γ is the discretised state matrix of the linear system model in MATLAB
- Θ is the discretised input matrix of the linear system model in MATLAB

Algorithm 1 summarises the process of tuning the parameters d_{long} and d_{lat}. How this is exactly implemented will be described in Section IV-D.

```
Algorithm 1 Parameter tuning process
Input: Correctly functioning Gazebo simulation software
    and LTI state-space model simulation in MATLAB
Output: Optimal combination of $d_{long}$ and $d_{lat}$ for the
    corresponding simulation
    Obtain and process Gazebo data
 1: Record input-output data of Gazebo simulation
    Construct MATLAB simulation and compare to Gazebo
    data
 2: for every combination of $d_{long}$, $d_{lat} \in$ [1, 1500] do
 3:     Construct continuous-time matrices using the current
        values for $d_{long}$ and $d_{lat}$
 4:     Discretise the continuous-time matrices at the same
        sampling time as the Gazebo simulation
 5:     Calculate the MSE for the whole sequence of discrete
        simulation samples
 6: end for
 7: Find the combination of $d_{long}$ and $d_{lat}$ that has minimum
    MSE value
```

E. Noise model

The Gazebo part in Figure 2 not only shows the linear system matrices that should be fitted to the simulation, but also the process and output noises (as explained in Section III-A.3). By analysing the rotational velocity of the Jackal robot in Gazebo, the process noise properties can be derived, as given in Section III-E.1. The output noise can be freely chosen within certain bounds, because Gazebo directly outputs the model states. Analysis and design of the output noise is the main topic of III-E.2.

1) Process noise: The process noise, denoted by $\boldsymbol{w}$ in equation (11), actually represents the noise present when calculating the next state when simulating the system in discrete-time. As described in the previous section, the linear system will be fitted to the Gazebo simulation. However, this system is not able to account for all variations around the operating point. Therefore, the state values of the fitted linear system differ from the 'ground truth' states coming from the Gazebo simulation. This difference can be considered as the process noise. It will be assumed in Section IV-E that the process noise is approximately white. Therefore, the model of the process noise needed in both filters is given by the standard deviation (σ_w) of the difference between Gazebo simulation results (only variations around operating point) and fitted linear system results. The final value chosen for σ_w will be provided in Section IV-E.

2) Output noise: The output noise is denoted as $\boldsymbol{z}$ in equation (12). This can be interpreted as the noise the IMU sensor adds to the system state. However, in case of the Gazebo simulation, the IMU sensor gives output values at a much lower frequency (50 Hz) than the simulation update frequency (1 kHz). Therefore, this simulated IMU sensor already adds some filtering behaviour to the values it outputs. Furthermore, the IMU on the real Jackal robot may show different behaviour than the simulated IMU sensor. It can be concluded that it does not really make sense to use the IMU data in the simulation as system output.

The output noise can thus manually be added to the Gazebo data before passing the data to the filters. Given the fact that the process noise is approximately white and the fact that the big advantage of the AI filter lies in the coloured noise aspect, the output noise will be made coloured. This will be done by filtering a white noise signal (with standard deviation σ_z) with a Gaussian filter (with kernel width s_z), as described in [5].

The value of σ_z (related to the amplitude of the white noise signal) should be chosen in such a way that the coloured noise signal will be clearly visible, but not too large. Therefore, a value of approximately $5\sigma_w$ is chosen to be used. Moreover, the kernel width should satisfy two conditions [5]:

1) In order to satisfy the Nyquist criterion, the simulation frequency should at least be twice as high as the coloured noise frequency. Therefore, s_z is safely chosen to be at least 10 times higher than the simulation sampling time
2) In order to have proper signal convolution with the constructed white output noise signal, s_z should be chosen quite smaller than the total simulation time. To be on the safe side, this value is chosen at least 100 times lower than the total simulation time

The exact values characterising the output noise are also provided in Section IV-E.

F. ROS layout

This section provides the overview of all ROS nodes needed to be able to perform the comparison between the AI and Kalman filter. As already mentioned, the system is designed for this comparison, but the comparison itself will not be performed, since it needs further research.

Figure 3 gives the interconnection of the relevant ROS nodes needed to run the filters. The dashed lines are related to obtaining experimental results on the filter comparison and simulating the AI filter in closed-loop. They indicate future work that is not covered in this paper, but will be covered later on.

As can be concluded from this figure, the simulation needs a controlling element: *Simulation control*. The simulation control node should indicate to Gazebo what the desired movement of the Jackal robot is by publishing these values on the */cmd_vel* topic. This movement is already given in Section III-D.2 and equals: ($\dot{x}_{lin} = 0.3\mathrm{m\,s^{-1}}$, $\dot{\phi}_z = 22.5^\circ\,\mathrm{s^{-1}}$).

The *Gazebo simulation* node represents the Gazebo simulation. For the sake of convenience, all Gazebo nodes are captured in this single block. Gazebo subscribes to the */cmd_vel* to get the desired robot motion. During simulation, the wheel velocities on both sides of the robot are published to the */joint_states* topic. This topic is used in the Kalman

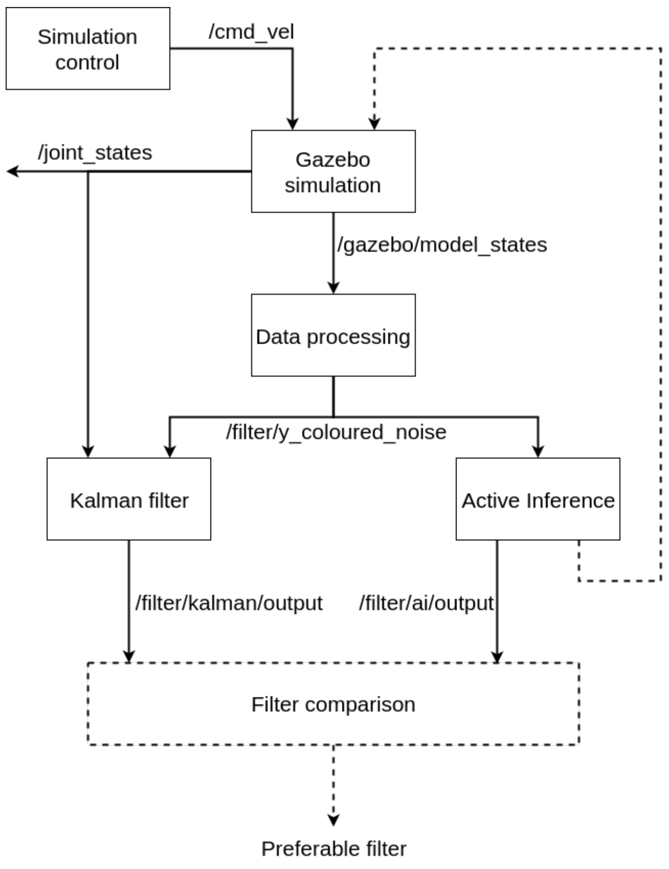

Fig. 3. Design of ROS layout

filter, while it is also used for system linearisation (therefore also not being connected to any ROS node).

Moreover, the *Gazebo simulation* node outputs the rotational velocity of the Jackal robot (defined as system output in equation (12)) on the */gazebo/model_states* topic.

Preferably, the data being published to the */joint_states* and */gazebo/model_states* topics arrive at exactly the same time in order for the Kalman filter to work properly.

The *Data processing* node is used to create coloured output noise. This node first needs an initialisation phase in which it generates the coloured noise by executing the convolution between the white output noise signal and its corresponding Gaussian filter (as explained in Section III-E). When running 'live', it means that the node subscribes to the */gazebo/model_states* to get the system output, adds the generated coloured output noise signal and publishes this result to the */filter/y_coloured_noise* topic.

The *Kalman filter* node implements the Kalman filter functionality (explained in Section III-B), which is in need of the changing values for the control input and system output during simulation. It outputs the state estimate of the Jackal robot's rotational velocity.

The *Active Inference* node implements the AI filter functionality, according to Section III-A. The AI filter also needs the system output and it outputs the state estimate. However, the *Active Inference* node is expected to also calculate the new control input for the Gazebo simulation in the future. This is indicated by the dashed line connecting the *Active Inference* with the *Gazebo simulation* node.

IV. IMPLEMENTATION AND VALIDATION

This section describes the implementation and validation of the design provided in Section III. First of all, the software being used is described in Section IV-A. Thereafter, in contrast to the previous section, a top-down approach is used to explain the system setup using the content of Figure 3 in order to have a clear overview of the system.

First of all, the implementation of the designed ROS layout, including message types and contents, will be the topic of Section IV-B. Thereafter, the implementation of the *Simulation control* node and the adjustments made in the *Gazebo simulation* block will be described in Section IV-C. As mentioned in Section III, both filters need a linear system model. The implementation for linearising the system around the defined operating point is provided in Section IV-D. Section IV-E derives the values for the properties of the process and output noise, which are used in the *Data processing* node. Finally, the implementations for the AI and Kalman filter are given in Sections IV-F and IV-G, respectively.

A. Software installation

In order to be able to run ROS and Gazebo, an operating system is required. As stated in the requirements (Section II), in order to simulate the Jackal robot, Gazebo 7 together with ROS Kinetic is required. ROS Kinetic is again only supported on Ubuntu 16.04 LTS. Due to a lack of memory, this operating system is installed on an external Solid State Drive (SSD) and the laptop has become dual-boot. After installing ROS Kinetic and supported Gazebo 7, the Jackal ROS packages for Gazebo were installed. What was further needed was MATLAB to perform the system linearisation and filter design, among other small test codes. Within MATLAB there exists an option to read in ROS bags (recordings of data being published on certain topics over time). However, when creating custom messages for the filters, the standard functions in MATLAB do not work anymore. Therefore, dedicated MATLAB code was generated using the `rosgenmsg` command. Furthermore, some changes in the ROS configuration files were applied, which will be described in Section IV-B.

B. ROS layout

Figure 4 shows the graph of all important ROS nodes that are active when the simulation is running. The node names correspond to Figure 3. Furthermore, some other nodes and topics are displayed, which are needed to properly run the simulation, but they are not important when obtaining the experimental results presented below.

All ROS nodes are either implemented in C++ or Python, because those are the languages that ROS allows. For this project, the nodes are programmed in Python, since this

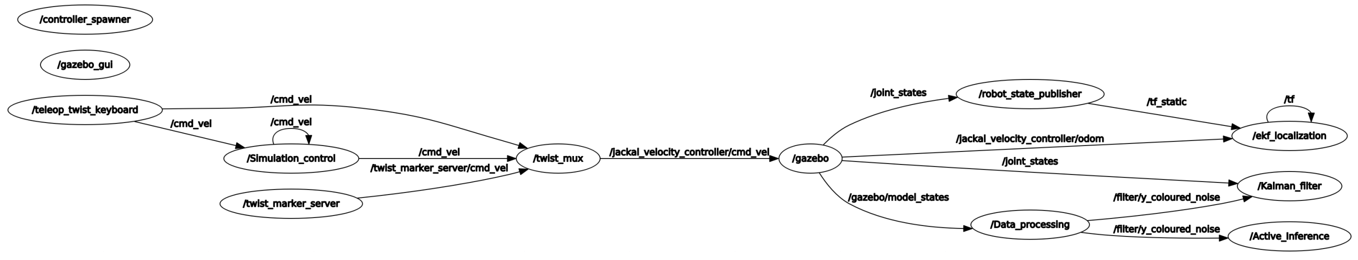

Fig. 4. Overview of all important running ROS nodes

is a relatively efficient programming language in terms of amount of code lines. The drawback is a bit less real-time performance. However, it is not expected to cause any problems in case of this Gazebo simulation at 1 kHz. The simulation even slows down its real-time factor when the computational load becomes too big to satisfy all timing deadlines.

First of all, the Gazebo simulation gets the desired robot movement via the */jackal_velocity_controller/cmd_vel* topic given by the *twist_mux* node. As the figure shows, this signal can be generated in three different ways, which will be explained in Section IV-C.

Secondly, the Gazebo simulation outputs different measurements of states of which the most important messages are published on the */joint_states* and */gazebo/model_states* topics. The */gazebo/model_states* topic is subscribed to by the *Data_processing* node, which adds the output noise to the signal and publishes the result on the */filter/y_coloured_noise* topic, which is used by both filters to obtain the state estimate of the Jackal robot's rotational velocity. Furthermore, the */joint_states* topic is subscribed to by the *Kalman_filter* node, because the control input is needed in the Kalman filter to calculate the state estimate, as explained in Section III-B.

Finally, the Kalman and AI filter publish their results on the */filter/kalman/output* and */filter/ai/output* topics, respectively. These topics, however, are not shown, because they are not used in another ROS node, but merely recorded in ROS bags to analyse the data in MATLAB after the simulation has completed.

Note that all topics introduced in this project are called */filter/...*, which is done by purpose to clearly separate these topics from the other topics automatically present when running the simulation. Furthermore, in order to transfer data via these topics, the data format needs to be specified. This is done by generating custom ROS messages.

The *Gazebo_model_states_noise* message is used as data format for the */filter/y_coloured_noise* topic and defined in the *msg* folder of the ROS package. The following variables are stored in this message:

- $delta\_t \in \mathbb{R}$: time difference between two successive updates of the simulation model states
- $sigma\_w \in \mathbb{R}$: standard variation of the white process noise as defined in Section III-E.1
- $s\_w \in \mathbb{R}$: kernel width of the Gaussian process noise filter as defined in Section III-E.1
- $sigma\_z \in \mathbb{R}$: standard variation of the white output noise as defined in Section III-E.2
- $s\_z \in \mathbb{R}$: kernel width of the Gaussian output noise filter as defined in Section III-E.2
- $y\_model \in \mathbb{R}^3$: model states of the Gazebo simulation. This includes the linear velocity along the x-axis and y-axis and the angular velocity around the z-axis
- $y\_noise \in \mathbb{R}^3$: generated noise by the *Data processing* node to be added to the Gazebo model states data
- $y\_model\_noise \in \mathbb{R}^3$: sum of y_model and y_noise. This signal is considered to be the 'real' output signal of the system to be controlled

Note that y_model, y_noise and y_model_noise are arrays containing the linear velocities and angular velocity of the Jackal robot, while in this paper only the angular velocity is analysed. However, in the future maybe more states should be observed, as will be described in Section V. Therefore, the message is already defined to be universally applicable.

Moreover, the *Filt_output* message is used as data format for the */filter/kalman/output* and */filter/ai/output* topics and also defined in the *msg* folder in the ROS package. This message contains the variables listed below:

- $x\_filt \in \mathbb{R}$: state estimate calculated by either of the filters
- $u \in \mathbb{R}^2$: control input signal (from the */joint_states* topic), which is only needed in case of the Kalman filter
- $u\_lin \in \mathbb{R}^2$: 'linearised' control input signal, which only contains the control input variations around the operating point
- $y \in \mathbb{R}^3$: system output signal (from the */filter/y_coloured_noise* topic) containing only the angular velocity component of y_model, y_noise and y_model_noise
- $y\_lin \in \mathbb{R}^3$: 'linearised' system output signal, which only contains the system output variations around the operating point

Actually, only x_filt is needed to provide as filter output. However, for debugging and plotting purposes in the post-processing stage in MATLAB, both the filter inputs and outputs are given to design the filter in MATLAB.

Finally, a ROS *launch* file has been defined in the package's *launch* folder in order to launch Gazebo with the desired properties and to be able to run all described ROS nodes 'at the same time', so it is not needed to run them separately in a new terminal tab. It contains the following functionality:

- Load Gazebo simulation information (see Section IV-C)
  - Load the *world* file containing model, surface, link and physics engine information for Gazebo to properly simulate the desired model and environment
  - Load the Jackal robot model as specified in the installed Jackal packages
  - Load the *yaml* file containing the adjusted publish rate of the wheel rotational velocities
- Run Gazebo simulation with specified information
- Run ROS node *Simulation_control*
- Run ROS node *Data_processing*
- Run ROS node *Kalman_filter*
- Run ROS node *Active_Inference*

Given the described ROS layout in this section, it can be concluded that the design described in Section III-F is properly implemented.

C. Simulation and simulation control

This section describes the system setup used to obtain the data based on which the AI and Kalman filter will be compared with each other in the future. Section IV-C.1 first describes how the Jackal robot simulation in Gazebo can be controlled. Thereafter, Sections IV-C.2, IV-C.3, IV-C.4 and IV-C.5 give observations and possible adjustments made to the Gazebo simulation in order to better fulfil the goal of this project. Finally, Section IV-C.6 provides an overview of how the simulation is controlled with all important parameters listed.

1) Simulation control: As mentioned in the previous section, the Gazebo simulation is controlled using the information on the */jackal_velocity_controller* topic. This data can be generated using three different inputs:

1) *Simulation_control* node
2) Keyboard
3) *twist_marker_server* node

of which the *twist_marker_server* node is outside the scope of this project. The main way of controlling the simulation is by using the *Simulation_control* node, which publishes the desired robot movement to the */cmd_vel* topic at a predefined frequency. This frequency is set equal to 50 Hz in order to not let the robot slow down or stop between two successive messages, due to a too low frequency. A higher frequency also works fine, but this would cause the simulation to slow down.

Moreover, the keyboard can be used to steer the robot in simulation. Once the *teleop_twist_keyboard* node is started up and a key is pressed, this node will continuously publish the desired robot movement to the */cmd_vel* topic. However, this command would destroy the predefined movement by the *Simulation_control* node. Therefore, the keyboard is only used to debug the simulation and see whether it behaves as expected. In that case the *Simulation_control* block checks on the frequency of data being published to the */cmd_vel* topic to see whether a key is pressed (which is published at a higher frequency than 50 Hz) and stops publishing until the key is released. This is done in order to prevent the two sources of commands to interfere. Figure 4 shows this check with the arrow of */cmd_vel* going into the *Simulation_control* node again.

It should be noted that a specific data format is used for the */cmd_vel* topic: the *Twist* message. The *Twist* message is defined within the *geometry_msgs* package, which is a prebuilt ROS package. It contains a vector `linear` with the linear velocities along the x-, y- and z-axis (in $\mathrm{m\,s^{-1}}$) and a vector `angular` containing the angular velocities around the x-, y- and z-axis (in $\mathrm{rad\,s^{-1}}$). Therefore, only the first element of the `linear` and the last element of the `angular` vector have to be set to their desired values given in Section III-D. One may notice that this does not directly correspond to the linear model input as defined in equation (11). This command is via the differential drive controller converted to ω_r and ω_l. The differential drive controller is also a prebuilt ROS package being used in the communication between ROS and Gazebo.

2) Simulation world: By default, the Gazebo simulation world (that is installed along with all Jackal simulation packages) is kind of a ‘race’ world defining a circuit the Jackal has to travel. This world contains obstacles representing the circuit borders. However, for this project, the Jackal should not collide with any obstacle, but should rather be free to move around, especially for the circular motion as defined above. Therefore, an ‘empty world’ *world* file is created within the *worlds* directory of the ROS package, causing the simulation to start up with a world containing only free space.

3) Friction model: As given in the Section IV-B, the Gazebo simulation gets as input the *Twist* message with linear and angular velocities from the */jackal_velocity_controller/cmd_vel* topic. This information is processed in a physics engine used to analytically calculate the next positions of all objects and links in the simulation. By having a high enough update frequency, Gazebo is able to properly calculate the new positions thereby resembling reality quite good. However, a difficult modelling aspect in case of the Jackal robot, which is crucial for obtaining realistic values for the rotational velocity, is the contact model of the wheels with the ground. This model often includes parameters that depend on the surface properties of both the wheels and the ground.

Figure 5 shows the evolution of the rotational velocity over time when using different settings of the physics engine Open Dynamics Engine (ODE) in the Gazebo simulation. The reference signal indicates what is published on the */jackal_velocity_controller/cmd_vel* topic regarding the rotational velocity ($\frac{d\phi}{dt}$). Furthermore, the simulation data is shown when either the pyramid friction model or the cone friction model is used in the physics engine in Gazebo. The default physics engine used by Gazebo is ODE and the default friction model uses the pyramid construction, in which the "friction constraints are decoupled into two 1-dimensional constraints that are solved independently" [8]. These 1-dimensional constraints are the longitudinal and lateral directions of the wheel contact points. As the figure shows, this way of modelling results in a quite inaccurate movement with respect to the reference input. The behaviour can be explained using the stick-slip model, in which the force builds up in both constraint directions until a certain threshold (the part where the graph is decreasing) after which the robot slips further (the quite steep rises in the graph). Obviously, this is not a desired way of manoeuvring around, because the signal has weird noise characteristics, which are not useful for the comparison of the AI filter with the Kalman filter.

The cause of this problem was first thought to be the physics engine used. Therefore, physics engine Bullet was used as a second test. Unfortunately, the installed Gazebo 7 did not show the physics engine it was using during its current simulation. Therefore, Gazebo 9 was installed, which is officially not compatible with ROS Kinetic and not supported by Clearpath Robotics Inc. in case of the Jackal robot, which was discovered after a few days. Therefore, Gazebo 7 was re-installed, this time with the updated *libignition-math2* library, causing a newer version (7.16) to install, which actually did show the used physics engine during simulation. However, Bullet did not turn out to give proper results and is therefore not included in this paper.

After this conclusion, a bit of research has lead to the adjustability of the friction model in ODE. The type of model to be used can be defined in the *world* file within the *worlds* directory of the ROS package. The friction model has changed to the cone model, which "solves the coupled dynamic friction constraints by choosing a friction direction parallel to the linear velocity at the contact point" and the forces are calculated in radial directions from the wheel contact point. This better resembles reality and therefore gives more realistic results as shown in Figure 5.

There is still a strange effect present in the rotational velocity simulation data, causing the signal to show regularities every quarter of a circle. Figure 5 shows the movement of the Jackal robot of approximately one circle (actually a little bit more to be sure that one circle is completely travelled). In total five peaks are shown in the cone model data (four corresponding to the current circle and one corresponding to the beginning of the next circle), which seem to coincide with the rising edges of the data obtained using the pyramid model. The exact cause of this phenomenon is unknown yet and will be listed as future work in Section V.

4) Update frequency: Another aspect of the Gazebo simulation is the update frequency defining the discrete moments in time at which the physics engine calculates the next positions of all parts of the Gazebo model under simulation. This frequency defaults to 1 kHz and is taken to be that frequency, because the results seem reasonable and the total

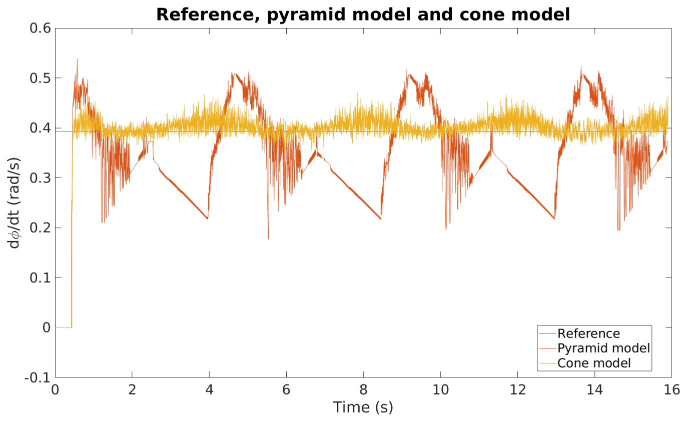

Fig. 5. Rotational velocity using pyramid and cone model

simulation would not take too long. This frequency (together with its corresponding maximum step size and real-time factor) can also be specified in the *world* file in the *worlds* folder of the ROS package.

Searching for 'the best' simulation frequency is also an aspect that needs future research and will thus also be included in Section V.

5) Wheel velocities: The last important observation that can be made regarding the Gazebo simulation considers the individual wheel velocities of the Jackal robot. As already mentioned, a discrepancy exists between how the input of the linear model is defined in Section III-D.1 and how the simulation is controlled using the *Twist* message on the */cmd_vel* topic. Therefore, to get the input as used in the linear model (covering the wheel velocities on either side of the robot), the */joint_states* topic is also subscribed to by the Kalman filter. However, the publish frequency of messages to this topic is 50 Hz by default, while the update frequency of the simulation (as described in the previous section) is 1 kHz. The Kalman filter can only be executed when the control input as well as the system output are received on the corresponding topics. The update frequency of the Kalman filter will thus equal 50 Hz. It is desired to raise the publish frequency of messages containing wheel velocity information as this will increase the state estimate update rate of the Kalman filter. Furthermore, as the upper 2 graphs in Figure 6 (representing the rotational velocity of the front wheel on the right-hand side of the robot sampled at different frequencies) show, it will result in more accurate wheel velocity data, thereby improving the reliability of the Kalman filter state estimates.

Increasing the publish rate of the wheel velocities is done by creating a *yaml* file in the *config* folder of the ROS package in which the publish rate of the *jackal_joint_publisher* is adjusted from 50 Hz to 1 kHz.

Another aspect of the wheel velocities to take into account when using the wheel velocity information is displayed by the lower 2 graphs in Figure 6, representing the rotational velocity of the front and rear wheel on the left side of the robot. It clearly shows that the wheel velocities are not

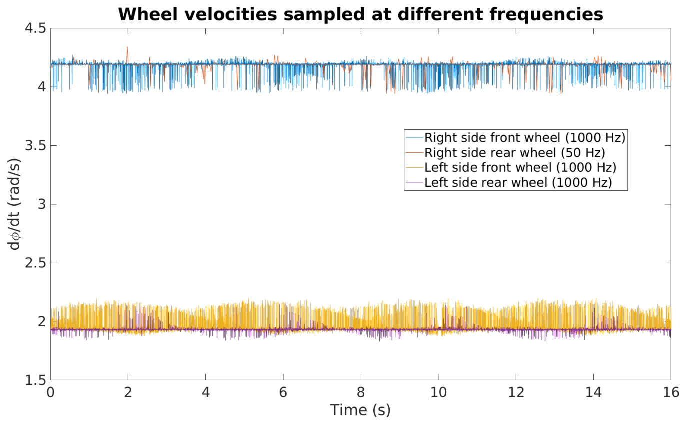

Fig. 6. Wheel rotational velocities with a defined publish rate of 50 Hz for the right side rear wheel and 1000 Hz for the other wheels

exactly equal. This is probably caused by the friction model, because the frictions on both wheels are not equal, due to their physical position on the robot. The same holds for the wheels on the right-hand side of the robot. Given the fact that the robot model assumes equal rotational velocities on each side of the robot, the average wheel velocities on each side will be taken to perform the calculations with.

Finally, by carefully looking at Figure 6, one can conclude that the wheel velocities also show repetitive behaviour four times per circle. This should also be part of the future work as described in Section IV-C.3.

6) Overview: Table II summarises the implemented values and decisions for all adjustable parameters and choices to be made for the simulation environment and its control.

Parameter/implementation choice	**Value/choice**
Publish rate of */cmd_vel*	50 Hz
Content of */cmd_vel*	$\dot{x}_{lin} = 0.3\mathrm{m\,s^{-1}}$ $\dot{\phi}_z = 22.5/180 * \pi\mathrm{rad\,s^{-1}}$
Simulation world	Empty
Friction model	Cone
Simulation frequency	1 kHz (default)
Publish rate of */joint_states*	1 kHz

TABLE II

PARAMETERS AND IMPLEMENTATION CHOICES FOR THE GAZEBO SIMULATION AND ITS CONTROL

D. Linear system model

This section describes how the linear system model is fitted to the Gazebo simulation data according to Algorithm 1 in Section III-D. First of all, the Gazebo simulation data has to be recorded and saved in a usable format. This will be described in Section IV-D.1. Thereafter, the LTI model has to be simulated in MATLAB using the same control inputs as the Gazebo simulation, and the system outputs should be compared with the Gazebo outputs. This implementation is provided in Section IV-D.2. Finally, Section IV-D.3 indicates how the optimum values for d_{long} and d_{lat} are found using the MSE values obtained in the previous step.

1) Obtain and process Gazebo data: In order to be able to fit the model parameters d_{long} and d_{lat} to the Gazebo data, input-output of the Gazebo simulation is required. As given in Section IV-B, the control input can be recorded on the */joint_states* topic. Furthermore, the system output can be obtained from the */gazebo/model_states* topic. The content of these topics over time will thus be saved in a ROS bag. The total simulation time is taken to be 16 s in order to let the Jackal robot drive at least one complete circle in which it is assumed that all robot behaviour needed to fit the LTI model is captured.

Because of the fact that the data on the */joint_states* topic is published independent of the content published to the */gazebo/model_states* topic, the control input data does not arrive at exactly the same time as the system output. To be able to use the control input and system output samples as if they were recorded at exactly the same time, the control input signal is linearly interpolated at the time stamps of the system output. Actually, linear interpolation introduces filter behaviour. However, it is chosen, because it is simply implementable and still gives quite good results. Furthermore, the resulting signals can now be used to simulate the system in MATLAB at the same frequency as Gazebo.

2) MATLAB simulation and comparison: The control input data from Gazebo is used as input in the LTI state space model. First of all, the following factors, with which d_{long} and d_{lat} are multiplied in the system matrices, are pre-calculated in order to prevent from computational overload:

- $\frac{-4}{I}b^2$
- $\frac{-4}{I}a^2$
- $\frac{2br_{wheel}}{I}$
- $-\frac{2br_{wheel}}{I}$

Thereafter the LTI state-space model is constructed in continuous-time for every value of d_{long} and d_{lat} in the predetermined range. These system matrices are discretised using the `c2d` command in MATLAB with sample time 1 ms (corresponding to simulation frequency of 1 kHz) to obtain the matrices Γ and Θ from equation (31). The next step is to calculate the MSE value as provided in (31) for the whole recorded input-output dataset for each discretised system with combination of values for d_{long} and d_{lat}.

3) Find optimum values for d_{long} and d_{lat}: Figure 7 shows the resulting MSE values of this approach. The optimum values for d_{long} and d_{lat} are the values for which the MSE value is the lowest. The figure seems to indicate that minimising the error happens for values of d_{lat} larger than 1500. However, when zooming in it turns out that the minimum is captured in this graph. Table III provides the optimum values found for d_{long} and d_{lat}.

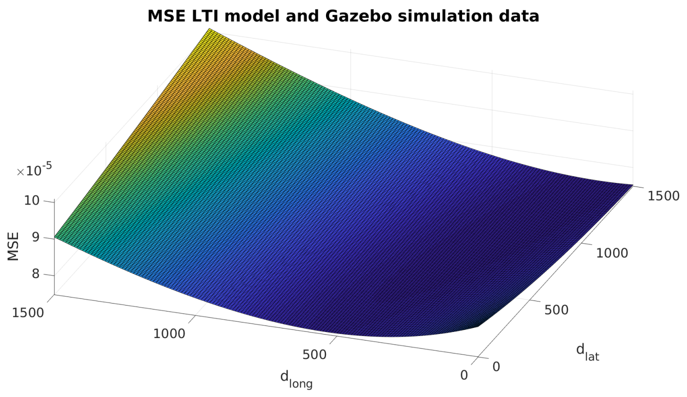

Fig. 7. MSE values for $d_{long} \in [1, 1500]$ and $d_{lat} \in [1, 1500]$ with stepsize 10

Parameter	Value
d_{long}	227.89
d_{lat}	1102.03

TABLE III

OPTIMUM VALUES FOR d_{long} AND d_{lat} AFTER FITTING THE LTI MODEL TO THE GAZEBO DATA

It has to be noted that the method for calculating the MSE values for every combination of d_{long} and d_{lat} suffers from the curse of dimensionality with decreasing stepsize ($\mathcal{O}(n^2)$, where n represents the number of possible values for d_{long} and d_{lat}). Therefore, the MATLAB function `fminsearch` is used to calculate the exact optimum as provided in Table III. The function starts at a certain point (d_{long0}, d_{lat0}) and tries to find the coordinates (d_{long}, d_{lat}) for which the MSE value has a (local) minimum. Fortunately, as Figure 7 shows, the MSE values have a convex shape. Therefore, `fminsearch` will always find the global optimum up to a certain accuracy for d_{long} and d_{lat}, which certainly meets the requirements in this project.

E. Noise model

Now that the LTI model is implemented, the noises in Figure 2 still need to be implemented. As described in Section III-E, a distinction is made between the process and output noise. Both implementations are provided in Section IV-E.1 and IV-E.2, respectively. Section IV-E.3 gives an overview of the final noise values used in the system. Given the fact that the *Data_processing* node implementation only depends on the noise properties, its implementation will be discussed in Section IV-E.4.

1) Process noise: The process noise is given by $\boldsymbol{w}$ in equation (11). In discrete-time, it is thus the difference between the next state and the predicted next state. Actually, its mean value is already displayed in Figure 7 and approximately equal to zero. What is further needed to characterise the process noise in the AI framework, is the standard deviation σ_w.

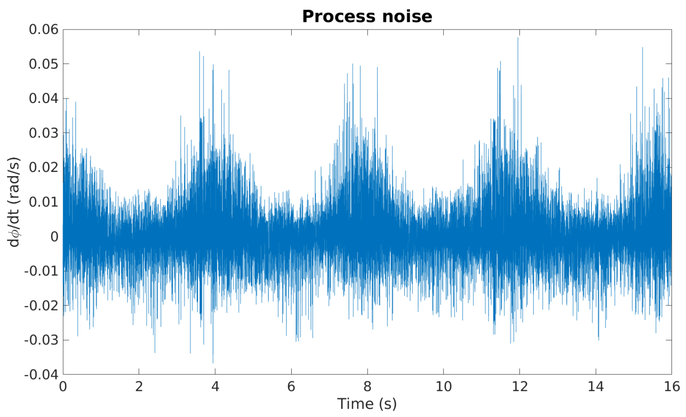

Fig. 8. Process noise when driving at least one complete circle

The LTI model can account for small variations around the operating point, but does not exactly resemble the Gazebo simulation results. The remaining variations can be regarded as process noise. Therefore, the same data is used as for the construction of the LTI model in order to calculate σ_w. Figure 8 shows the process noise capturing a time period in which the Jackal drives a little bit more than one circle, for which it is assumed that all possible variations in the process noise are captured.

The variance of this signal equals $7.49 \times 10^{-5} rad^2/s^2$, resulting in a standard deviation of $\sigma_w = 8.65 \times 10^{-3} rad/s$.

When carefully looking at Figure 8, one may observe the same interval of regularities as described before. This is caused by the fact that the LTI model does not expect these regular deviations from the desired movement and this will thus be regarded as process noise. This problem can hopefully be solved in future research projects.

Figure 2 also shows a Gaussian filter $H_w(s_w)$, where s_w represents the kernel width. This filter should be fitted to the noise in case it is coloured. Given the fact that the noise is assumed to be white, the kernel width is taken to be zero ($s_w = 0$).

2) Output noise: Similar to the process noise, the output noise is given by $\boldsymbol{z}$ in equation (12), which represents the difference between the system output and the predicted system output in discrete-time. As explained in Section III-E.2, the output noise can be manually generated. The value for σ_z was explained to be $5\sigma_w$, resulting in $\sigma_z = 5 \times 8.65 \times 10^{-3} = 4.33 \times 10^{-2} rad/s$.

According to Section III-E.2, the kernel width s_z of the Gaussian filter used to create coloured output noise should be at least 10 times higher than the simulation sampling time (being 1 ms) and at least 100 times lower than than the total simulation time (being 16 s). To satisfy both criteria, $s_z = 0.1$ is chosen to be the filter kernel width.

Figure 9 shows the system output decomposition including the Gazebo data and the added coloured output noise. It shows that the value of $\sigma_z = 5\sigma_w$ is reasonable, because a clearly coloured noise signal is generated, but it stays

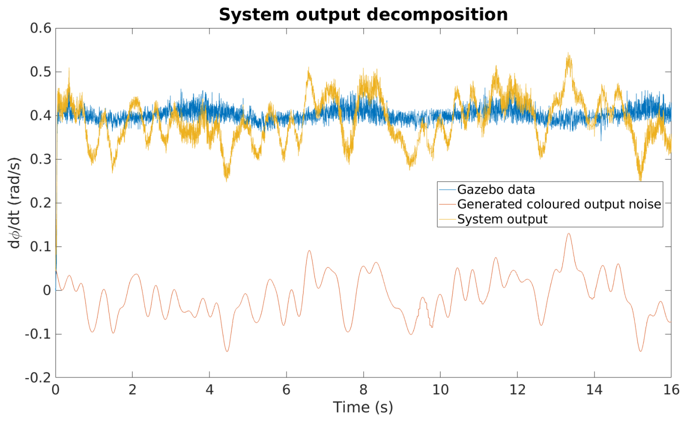

Fig. 9. System output decomposition

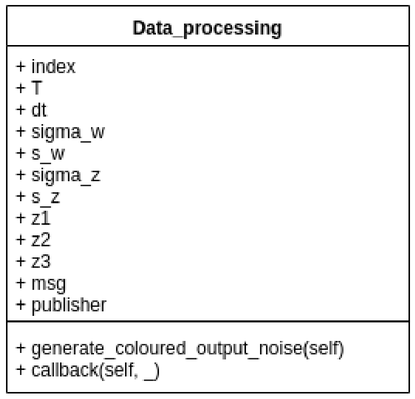

Fig. 10. Data processing UML class diagram

within reasonable bounds.

3) Overview: Table IV shows the resulting noise model properties. To summarise: the process noise is assumed to be white, while the output noise is created to be coloured, thereby only giving s_w a value of zero.

Noise property	Value
σ_w	8.65×10^{-3} rad/s
s_w	0
σ_z	4.33×10^{-2} rad/s
s_z	0.1

TABLE IV
NOISE PROPERTY VALUES

4) Data processing: As mentioned in Section IV-B, the *Data_processing* node is responsible for adding coloured output noise to the Gazebo data. In order to produce the coloured output data, a signal convolution (with argument `valid`) between a white output noise signal and the Gaussian filter (given in equation (17)) is calculated for a prespecified amount of simulation time. The coloured output noise should be added to the data coming from the */gazebo/model_states* topic. It is easier programming to first calculate the whole coloured output signal and add one sample of this data to the current Gazebo data sample that is coming in. Therefore, the coloured output noise is generated in the node's initialisation phase. However, signal convolution costs some time. Therefore, the coloured simulation data will only be published once the whole output noise signal has been generated. It will thus take some time to have the coloured simulation data available when running this node using the *launch* file.

The simulation time used to construct the noise signal is taken to be 17 s to be sure that it can be added to the Gazebo data for one complete circle.

Figure 10 provides the Unified Modelling Language (UML) class diagram of the *Data_processing* node. The node consists of one instance of the class `Data_processing` performing all operations needed to retrieve Gazebo data from the */gazebo/model_states* topic and publish the coloured data on the */filter/y_coloured_noise* topic. In its initialisation phase, the coloured output noise array is calculated using the `generate_coloured_output_noise` function. This function uses the variables `T`, `dt`, `sigma_z` and `s_z` to create the coloured noise for three states in total (in order to also be applicable if all three states of equation (28) will be used in the future), represented by three arrays `z1`, `z2` and `z3`. In this project, only `z3` is used. Furthermore, `msg` is the instance of the message *Gazebo_model_states_noise* (as described in Section IV-B) to be published by `publisher` to the */filter/y_coloured_noise* topic.

F. AI filter

Figure 11 shows the UML class diagram of the *Active_Inference* node. It shows an association between the `Subscriber` and the `AI` class. As the names suggest, the `Subscriber` class subscribes to the topic */filter/y_coloured_noise*, on which the simulation data is published and the `AI` class performs the state estimation using the AI filter equations as defined in Section III-A.

The `Subscriber` class first initialises the operating point by defining `mean_u` and `mean_y` in its `__init__` function, which are determined when linearising the system. The `debug` variable, as the name indicates, is used for debugging purposes. It prints valuable information on the screen while the algorithm is running. Furthermore, the number of states (`n_states`), the embedding order (`p`) and the reference signal (`x_ref`) are defined and used when creating an instance of the `AI` class, called `ai`. The `Subscriber` class also declares the message type to be published (`msg`) and initialises the ROS publisher (`publisher`) to publish the state estimate result of the AI filter. Finally, the `Subscriber` class contains a callback function (`callback`) which is automatically called once a new message on the */filter/y_coloured_noise* topic is encountered. This data can immediately be used to call the `compute_mu` function inside the `AI` class, which performs the state estimation.

When a new instance of the `AI` class is created, it first initialises all variables needed to perform the state estimation, which are given by its attributes in Fig-

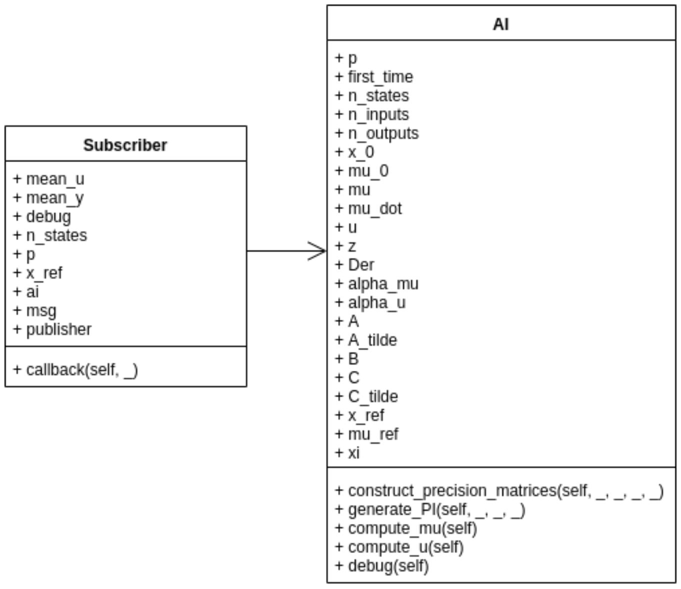

Fig. 11. Active Inference UML class diagram

ure 11. As mentioned, the most important function of this class is the `compute_mu` function, which performs the state estimation. However, in order to do so, the precision matrices need to be known. They are created using the `construct_precision_matrices` and `generate_PI` functions once the first message is published on the */filter/y_coloured_noise* topic, which contains the parameters necessary to construct these matrices. Furthermore, a `debug` function is present to print important variables on the screen for debugging the state estimation algorithm. Finally, the `compute_u` function is also listed, because this function will be used in the future to close the control loop by calculating the next necessary control input.

Note that the update rule for the state estimation as given in equation (20) is constructed using continuous-time matrices. However, this filter is run every time a new message is published on the */filter/y_coloured_noise* topic, which is when a new message of the */gazebo/model_states* topic is received at 1 kHz. Therefore, the AI filter runs at discrete times having intervals of approximately 1 ms. In order to discretise the state estimation, the following equation is implemented, which linearly extrapolates the slope at the current timestep ($\dot{\boldsymbol{\mu}}_k$), starting from the previous state estimate ($\boldsymbol{\mu}_{k-1}$):

$$\boldsymbol{\mu}_k = \boldsymbol{\mu}_{k-1} + \dot{\boldsymbol{\mu}}_k * \Delta t \tag{32}$$

where $\Delta t = 1\text{ms}$.

Note also that generalised output $\tilde{\boldsymbol{y}}$ is used in the AI filter (given by equation (20)), due the presence of coloured output noise. However, in a physical system, in this case the Jackal robot simulation in Gazebo, only the real system output (not the higher-order derivatives of the generalised output) can be measured. Therefore, the higher-order derivative elements in the output vector are taken to be zero.

The first tests of this filter have been run by recording the simulation data (*/filter/y_coloured_noise* topic) in ROS using ROS bags, loading the data, executing the AI filter equations and plotting the results in MATLAB. This way plots were easily generated and the filter equations could easily be tuned or adjusted in case of a small mistake, without having to restart the whole simulation in Gazebo.

Figure 12 shows the results of the first valuable AI filter test. The figure clearly shows that the state estimate is exploding. In order to create this graph, the learning rate was manually adjusted to a relatively low value of $\alpha_\mu = 3.408 \times 10^{-6}$. For larger values the state estimate explodes even more. When diving into the filter details, it was found that this is probably caused by the combination of relatively large values of the learning rate α_μ and the process error term $(\mathcal{D} - \tilde{A})^T \Pi_w (\mathcal{D}\boldsymbol{\mu} - \tilde{A}\boldsymbol{\mu} - \boldsymbol{\xi})$ from equation (20).

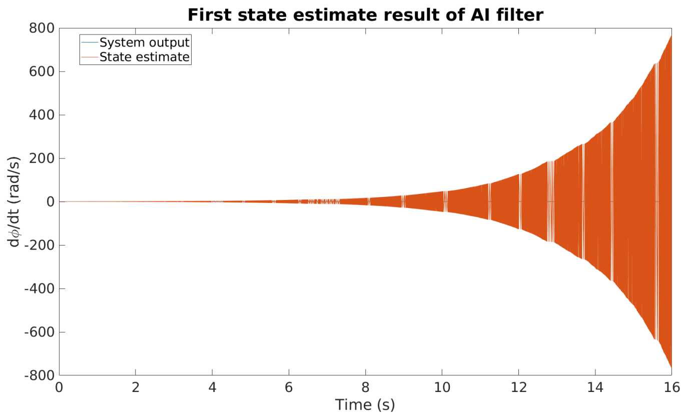

Fig. 12. First result of running AI filter

When setting the learning rate somewhat smaller (i.e. $\alpha_\mu = 3.4 \times 10^{-6}$), the result in Figure 13 is obtained.

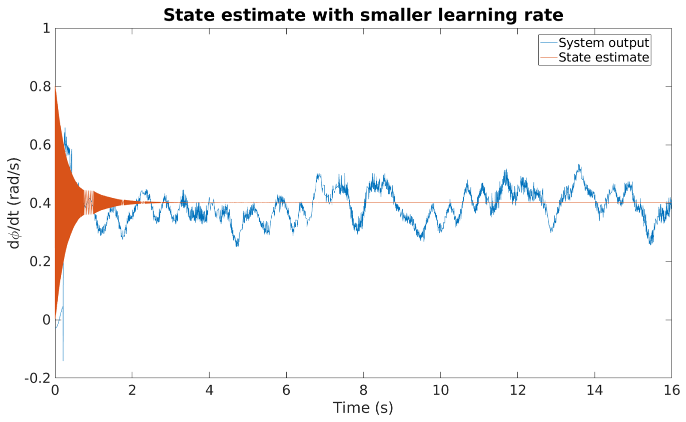

Fig. 13. AI state estimate for $\alpha_\mu = 3.4 \times 10^{-6}$

This result seems to indicate that the filter strongly converges towards zero. Take care that the linear system can only account for changes around the operating point, so the operating point is subtracted from the control input data and added to the filter output. The resulting state estimate thus quickly converges towards the operating point.

However, this does not seem to be reasonable, because the filter cannot exactly predict what the noise on the system output will be to compensate for it. Therefore, the conclusion can be drawn that the AI filter now strongly converges towards the prior $\boldsymbol{\xi}$, which was implicitly assumed to have zero elements when constructing Figures 12 and 13. The prior can be calculates as follows:

$$\boldsymbol{\xi} = \mathcal{D}\boldsymbol{\mu}_{ref} - \tilde{A}\boldsymbol{\mu}_{ref} \tag{33}$$

where $\boldsymbol{\mu}_{ref}$ can be considered the reference signal going into the filter. To see the effect of the prior on the state estimate, only the first (non-generalised) element of $\boldsymbol{\mu}_{ref}$, μ_{ref}, is adjusted to 0.6 (which is an arbitrarily chosen reference signal in the same order of the predefined rotational velocity). The result of this change in prior and keeping the learning rate at $\alpha_\mu = 3.4 \times 10^{-6}$ is shown in Figure 14.

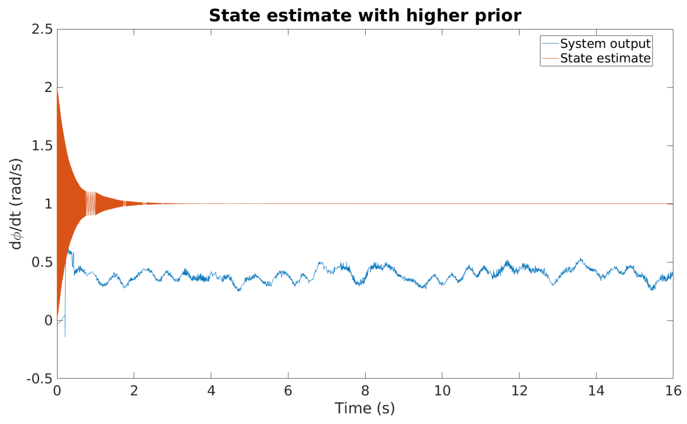

Fig. 14. State estimate of AI filter with $\mu_{ref} = 0.6$

This figure confirms the conclusion that the AI filter strongly converges towards the prior, thereby completely forgetting the system output. When looking at equation (20), it can be concluded that the output error term is weighted by precision matrix Π_z, while the process error term involves the precision matrix Π_w. Given equation (18) and the results in Table IV, it can be concluded that Π_w is a matrix with a relatively large element at position (1,1)[1] with a relatively small value for σ_w and $s_w = 0$. Π_z, on the other hand, contains a larger value for σ_z than σ_w and has a non-zero value for s_z. Therefore, the elements in Π_z are relatively small, causing the influence of the system output to disappear in the filter state estimate results.

The effect of manually decreasing the element at (1,1) in Π_w to a value of 1 and 0.01 with a higher learning rate of $\alpha_\mu = 0.01$ and the same prior ($\mu_{ref} = 0.6$) is shown in Figure 15.

From this result can be concluded that the ratio between the values in Π_w and Π_z plays a role in the trade-off of the AI filter between the prior and the system output. The filter expects the output to resemble its prior (which can be interpreted as some kind of intelligence learned by previous experiences), but it should also work well on different system outputs with the same noise properties. The higher the values

[1]Note that MATLAB matrix indexing is used here

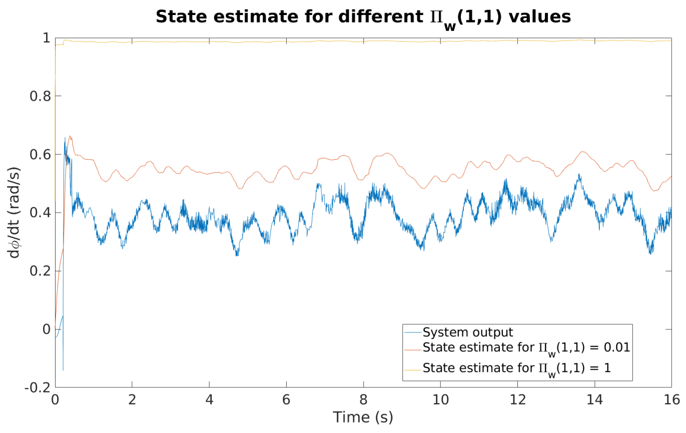

Fig. 15. Effect of varying values for (1,1) element in Π_w

in Π_w with respect to Π_z, the more the filter will converge to its prior, instead of the system output.

The final result that can be drawn from the AI filter results is displayed in Figure 16. The AI filter state estimate is shown for two different values of the learning rate. Furthermore, $\Pi_w(1,1) = 0.01$ and $\mu_{ref} = 0.6$. From this figure can be concluded that the higher the learning rate, the quicker the AI filter reacts to the ratio between prior and system output (determined by the ratio of Π_w values and Π_z values), but the more sensitive it will be to the noise present on the system output. The last effect can be explained by the fact that the state estimate will converge faster towards the aforementioned ratio between prior and system output, which slightly depends on the noise properties on the system output.

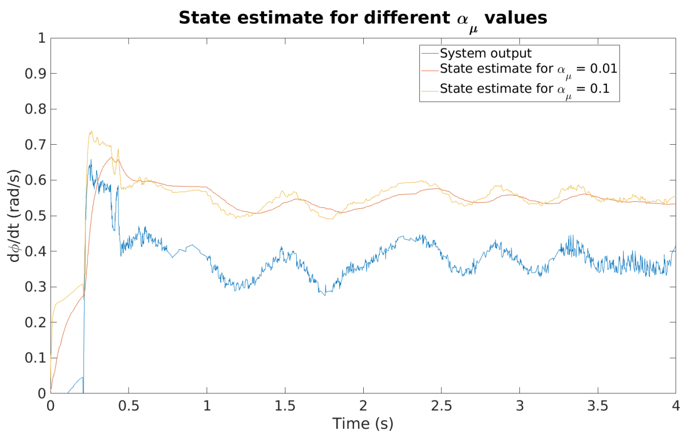

Fig. 16. Effect of varying values for α_μ

From the analysis above, the following conclusions can be drawn regarding the AI filter under the presence of white process noise and coloured output noise:

1) Depending on the learning rate, the state estimate will either explode or strongly converge to the prior when using the precision matrices as they are defined in equation (18)

2) The state estimate depends on the ratio between the values of the elements in Π_w and Π_z, which determine the position between reference (captured in the prior) and system output
3) The higher the learning rate, the quicker the state estimate will converge to the value described in conclusion 2 and the more sensitive it will be to noise in the system output

Further research into the exact contributions of all variables in the state estimate update rule, including the noise properties (and thereby automatically Π_w and Π_z), learning rate and prior, is thus needed to create a properly functioning AI filter.

G. Kalman filter

Figure 17 shows the UML class diagram of the *Kalman_filter* node. This class diagram shows the same structure as provided in Figure 11 for the *Active_Inference* node. Also in this case, the `Subscriber` class first initialises the variables `mean_u`, `mean_y`, `debug`, `msg`, `publisher`, which have the same meaning as in the AI filter. Only the `publisher` differs by publishing filter results to the */filter/kalman/output*, instead of the */filter/ai/output* topic. Moreover, an instance `kalman` of the `Kalman` class is created, which also initialises all variables needed to run the prediction (`predict` and update `update` functions of the Kalman filter given in Figure 17. Furthermore, the `Subscriber` class subscribes to the */filter/y_coloured_noise*. Once the system output is received the callback function `callback_y` is run translating the system output from operating point to origin and setting the variable `y_present` to `True`. Furthermore, the `Subscriber` subscribes to the */joint_states* topic, which provides the control input. Upon detection of this data, the callback function `callback_u` is called, thereby translating the control input from operating point to origin and setting variable `u_present` to `True`. Both `callback_y` and `callback_u` check whether the Kalman filter prediction and update rules can already be executed by looking up the value for `u_present` and `y_present`, respectively. If both control input and system output are received, the function `run_kalman` is called, resulting in the execution of the `predict` and `update` functions and publishing the results.

The Kalman filter is tested in the same way as the AI filter: by recording a ROS bag containing the necessary data to run the filter. The filter equations are implemented in MATLAB in order to easily tune the filter using the ROS bag data. After tuning the filter, the filter is implemented in Python and the results are compared.

However, before applying the Kalman filter to the coloured system output data resulting from the Gazebo simulation, the filter is first verified in MATLAB using a simulation with constant input, the linear system model as defined in Section IV-D and white noise signals (having amplitude σ_w and σ_z) added to the system state and output. The result is shown in Figure 18 and indicates that the Kalman filter is able to accurately predict the correct state value under the presence of the white noise signals.

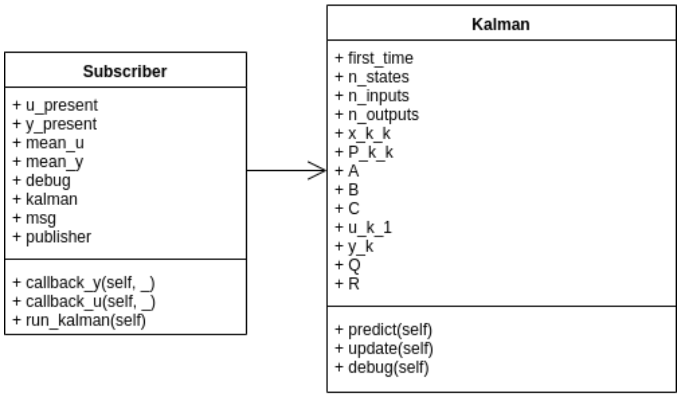

Fig. 17. Kalman filter UML class diagram

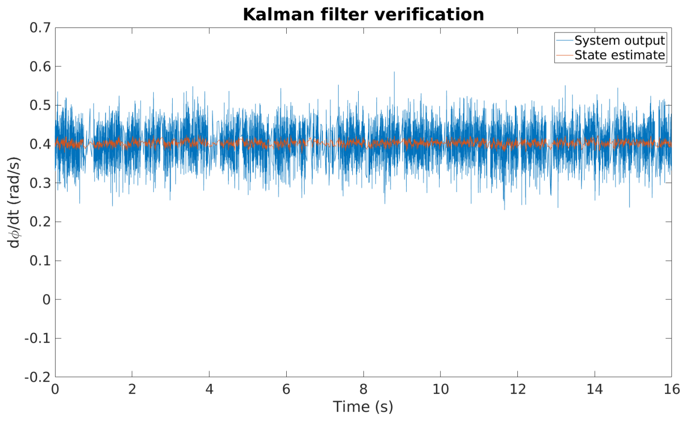

Fig. 18. Kalman filter verification

Note that the Kalman filter uses discrete-time matrices, instead of continuous-time matrices in case of the AI filter, in order to make correct estimates of the state in the prediction stage. These discrete-time matrices are discretised with a sampling time of 1 ms (corresponding to the 1 kHz simulation frequency of Gazebo and thus the publish rate of the */filter/y_coloured_noise* topic).

Figure 19 shows the results of the Kalman filter in the *Kalman_filter* ROS node and in MATLAB.

From the figure can be concluded that the Kalman filter behaves as expected, which means that it tries to compensate for the white process and output noise, but it cannot compensate for the coloured output noise. The figure also shows a small difference in state estimate obtained by running the filter in a Python script and in MATLAB. This effect is probably caused by the numerical differences between the two programming languages.

Finally, the filter performance can be calculated using the MSE between the state estimate and the data from the */gazebo/model_states* topic (representing the real system state in this case), which is given by the following equation:

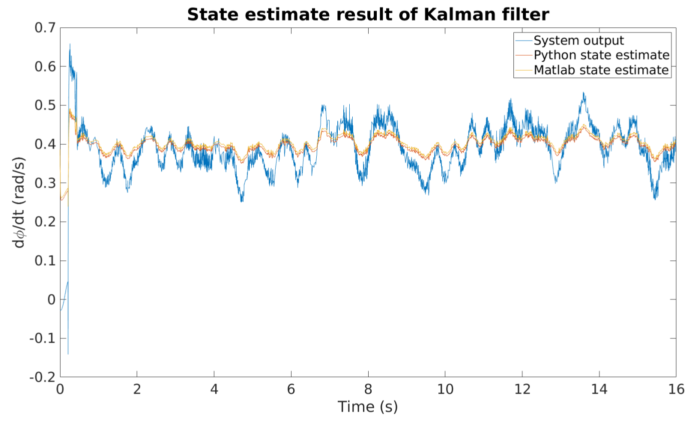

Fig. 19. Kalman filter state estimate in Python and MATLAB

$$MSE = \frac{1}{N}\sum_{k=1}^{N}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k)^2 \tag{34}$$

Figure 20 shows the difference between the Gazebo model state and the Kalman filter state estimate, which can be seen as a visual interpretation of the error described above. The main difference between the two signals is caused by the coloured output noise, which the Kalman filter cannot compensate for.

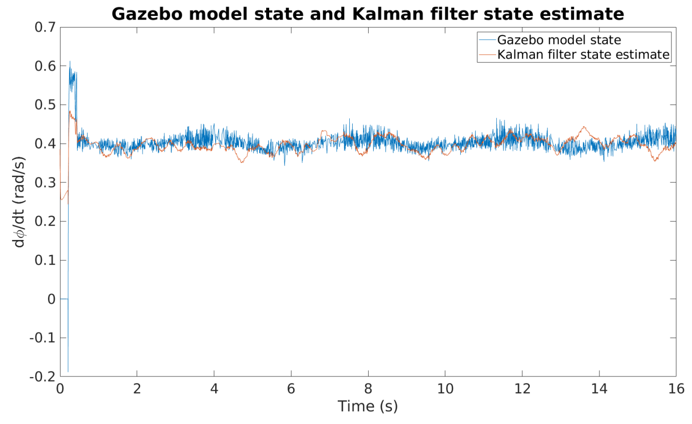

Fig. 20. Gazebo model state and Kalman filter state estimate

The MSE value for the Kalman filter equals 0.0025. This value can be used in the future to compare with the AI filter performance.

V. CONCLUSIONS

This paper described the implementation of a simulation and filter framework in ROS with as ultimate goal to compare the performance of the AI filter with the Kalman filter. The idea behind the AI filter is that it should perform better than existing filters when applied to a system under the presence of coloured process and coloured output noise. The Jackal robot is a skid-steering robot having slippery behaviour when driving in a circle, as shown in [7]. This slippery behaviour is an advantage when trying to simulate coloured noises. Therefore, this paper uses the Jackal robot in the Gazebo simulation.

The Gazebo simulation uses a friction model in its solver to model the contact point between wheel and ground. The model (chosen to be the cone model) is an important parameter when analysing the rotational velocity of the robot and causes non-linear robot behaviour. However, the system model of the Jackal robot is a linear model [7]. Therefore, the tunable parameters in this model are fitted using a least-squares algorithm to the simulation data. Furthermore, the process and output noise properties are fitted to the simulation data. The process noise is assumed to be white and the coloured output noise can be manually added to the simulation data, because Gazebo directly outputs the robot's rotational velocity, as described in Section III-E. However, the gazebo model rotational velocity shows unexpected regularities occurring four times per circle. The cause of this effect is yet unknown and needs further research. It may be related to the update frequency of the Gazebo simulation. Optimising the simulation frequency is thus also an aspect to consider in future work. Moreover, the process noise needs further investigation to see whether it contains structure and can thus be regarded as coloured process noise or not.

After adding the coloured output noise, the simulation data is used as input for the AI and Kalman filter. The Kalman filter performs as expected. However, the AI filter either shows unstable behaviour or converges to its prior very quickly. This is probably caused by the combination of precision matrices (constructed using the process and output noise properties), learning rate and prior. Further research is needed to develop an exact framework for the state estimate to be stable, involving the mentioned variables. Questions need to be answered like: do these results only appear in case of white process noise? Should the values in Π_w and Π_z be scaled with respect to the measurement values in order to prevent unstable behaviour? Is the Gaussian filter a good approximation to represent the coloured noise present on the system output?

Moreover, an important aspect to consider is timing. In this paper it has been assumed that every signal is received at the simulation frequency of 1 kHz and continuous-time matrices were discretised using the sampling time belonging to that frequency. However, this may not be the case. Furthermore, control input and system output signals are very likely to appear at different times on their respective topics. Therefore, investigation of the consequences of asynchronous inputs to the filters is also needed.

Finally, it can be concluded that this paper has provided a practical framework involving the Jackal robot to perform tests with the AI framework. The system is easily extendable to, for instance, perform state estimation on all three robot states, perform experiments with the real Jackal robot and to close the loop by calculating the control input using AI. Furthermore, it has provided valuable insights in the filtering part of AI needed to do future research into this area and has implemented a Kalman filter which can be used as benchmark filter.

VI. PROJECT ACHIEVEMENTS

This section describes the personal value of having done the project described in this paper.

First of all, to finish a project successfully, careful project planning is needed. The supervisor often gives an indication of what a potential research direction is for the project. It is important to stay in contact with the supervisor in order to not deviate too much from the final project goal. Deviation is needed, because sometimes it opens up theoretical gaps that need to be filled, but the project should also be finished in time. Therefore, weekly meetings with the supervisor were organised to discuss all achievements of the previous week. Before each meeting, all subjects to discuss and pictures confirming the current knowledge were sent to the supervisor in order to have an efficient meeting.

Secondly, this project has brought much knowledge regarding the AI framework and the Kalman filter. Some literature has been read regarding these topics to understand the concepts, but emphasis was put on the implementation of these algorithms. The implementation of these frameworks gave a lot of valuable insights and resulted in new theoretical gaps to be filled in the literature.

Thirdly, this project needed a careful experiment design to compare the filtering part of the AI algorithm with the Kalman filter. Although no conclusions could be drawn regarding this research question, it has helped in thinking of ways to properly set up an experiment and make design decisions in order to reach the final project goal. During the experiment development a lot of unexpected problems popped up that needed to be solved and sometimes the results indicated further research that needed to be done before being able to draw conclusions regarding the big picture. This suddenly opened up new research branches and required an iterative way of working.

Finally, this project provided a lot of practical experience in developing re-usable software, together with writing proper documentation. The very first thing to do was to get an operating system working on an external disk. New knowledge regarding this operating system was obtained, as well as experience with software development in ROS and programming in Python.

It can be concluded that this project was a very valuable part of the master program. The obtained knowledge, theoretical as well as practical, will certainly be needed after graduation.

REFERENCES

[1] S. Grimbergen, C. van Hoof, P. M. Esfahani, and M. Wisse, "Active inference for state space models: A tutorial," 2019, unpublished.

[2] C. R. Inc. Jackal ugv - small weatherproof robot - clearpath. [Online]. Available: https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/

[3] K. J. Friston, N. Trujillo-Barreto, and J. Daunizeau, "Dem: a variational treatment of dynamic systems," *Neuroimage*, vol. 41, no. 3, pp. 849–885, 2008.

[4] I. Hijne, "Computation of the generalized precision matrix," 2019, unpublished.

[5] M. Wisse, "Derivation of generalized covariance matrix," 2019, unpublished.

[6] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 196.

[7] A. van Doeveren, "The noisy jackal: Measurement and analysis of the coloured noise on the longitudinal, lateral and rotational velocities and identifcation of its characteristics due to the unmodeled dynamics of a skid steer mobile robot during a steady-state turning manoeuvre," Master's thesis, Delft University of Technology, Delft, 2019.

[8] O. S. R. Foundation. Friction parameters. [Online]. Available: http://gazebosim.org/tutorials?tut=physics_params&cat=physics#Frictionparameters