Predictive Robot Brain in ROS Simulation

W. Arink (4458702), S. van Ginneken (4356934),
C. Lock (4325117) and S. Römer (4396545)

***Abstract*— This paper contains the first steps towards implementing Active Inference, a neuroscientific theory, in robotics. These steps consist of exploratory research divided into four realms: controlling velocities v_x, v_y and ω_z of a simple robot, the prioritization of the sideways slip velocity, the addition of noise, and a stability analysis of the learning rates. These research topics are tested through simulations of a Jackal robot within the ROS simulation environment. Results in controlling the forward velocity v_x and the addition of noise were most successful, as the desired velocities were reached quickly. Next, the robot was able to control the angular velocity ω_z and prioritize the slip velocity v_y, however, much room is left for optimization. Inconclusive results were obtained for the stability analysis of the system. All things considered, the study showed promising results which propose many further investigation queries in optimizing the implementation of Active Inference in robotics.**

I. Introduction

As artificial intelligence becomes smarter, it starts taking over more jobs, tasks, and routines. At the moment, artificial intelligence is especially used for simple and predictable tasks. Problems arise however, when situations are noisy and become less predictable. Karl Friston, a professor in the field of neuroscience, created a theory on how the biological brain deals with this problem. Friston, the most cited scientist of today, explains with his theory of Active Inference [1] how the biological brain models the unpredictability of the world. This could also be very helpful for developments in artificial intelligence and robotics. Although the theory is well-known, to date it has never been implemented in a robot system.

The goal of this paper is to present a novel implementation of the Active Inference theory on a simple mobile robot, where it will estimate and control the velocities of this robot. A four-wheeled skid-steering robot has been chosen, because the unpredictability of the slip that occurs when skid-steering is exactly the type of noise that Active Inference should handle with exceptional effectiveness. The exploratory research is structured by the following four research questions.

1) Does the system succeed in controlling the velocities v_x, v_y and ω_z of a simple robot vehicle in a simulation using Active Inference?
2) Is it possible to make the algorithm prioritize one of the velocities v_x, v_y and ω_z separately?
3) Does the algorithm filter out noise as the theory promises, or is it limited to a perfect world?
4) How do different learning rates within the algorithm affect the stability of the system?

This study has been conducted as part of the Bachelor End Project for Mechanical Engineering students at the TU Delft in the third year of their Bachelor studies.

II. Theory

The theory used for this study, as mentioned in the introduction, is Active Inference. This theory is based on the Free Energy Principle [2], also composed by Karl Friston. With the Free Energy Principle, Friston tries to create a universal biological reward function based on the notion that all forms of life try to create order in a world that consists of thermodynamical disorder. In other words, Friston states that all life forms try to minimize their Free Energy. He explains that the (human) brain uses the Free Energy Principle to minimize its Free Energy in order to reach a certain goal and how it behaves in order to achieve this goal.

The brain, namely, tries to model the world around itself. In order to make this model it uses two concepts: perception and action. Through perception the brain can approximate the (hidden) states of the world, like estimating its current walking velocity. Through action the brain tries to change these states, e.g. giving a signal to its leg muscles to change its walking velocity. The theory proposes a unifying mechanism, meaning that the brain can change both its perception and action in order to improve its model of the world and ultimately achieve its goal. “An intuitive example of this process [...] would be feeling our way in darkness: we anticipate what we might touch next and then try to confirm those expectations.” [2]

The general neuroscientific theory has been made accessible for engineering purposes through the seminal publications of Friston [1] [2] [3], a more detailed mathematical exposition by Buckly [4] and a translation to Linear Time-Invariant (LTI) state-space systems by Grimbergen [5]. In this formulation Grimbergen explains how the unknown world can be approximated as a state-space model either by predefining it or letting the algorithm approximate it. The algorithm approximates this through a concept called 'expectation maximization' [3]. In this study the world's state-space will be predefined, due to time constraints of this project. Once the world's state-space is defined, the brain uses two formulas: the rate of change of the prediction, based on the perception, called $\dot{\mu}$ and the rate of change of the action, called $\dot{u}$. This way, the brain predicts how the world will change and steers the world towards a desired outcome, called prior ξ. Further details of how these terms are linked together and used in this study are explained in the next sections.

III. Methods

A. Startup decisions

In order to implement Active Inference in robotics, an operating system, a simulation environment and finally a robot needed to be chosen. ROS (Robot Operating System) was chosen as operating system as it has a broad availability of packages relevant for this study. Within ROS, Gazebo was chosen as simulation environment as it is the most common simulation tool used in ROS. Also it is lightweight, realistic and offers easy world manipulation. Next, a robot was chosen suitable for implementation of Active Inference, which needed to contain a strong uncertainty. A vehicle with skid steering was found to be suitable for this uncertainty, as it moves with great slippage. Skid-steering vehicles can only turn by changing the velocity of the wheels, as they are stuck to the frame and cannot be rotated sideways. Consequently, the vehicle will slip sideways when it tries to turn. In this study, this sideways velocity is defined as v_y. The other two velocities in which the kinematics of the vehicle are expressed are the forward velocity v_x and the rotational velocity ω_z, as can be seen in Fig.1. A robot with the above characteristics is the Jackal from Clearpath Robotics. The Jackal's model in Gazebo includes great approximations of its dynamics, including wheel slippage and skid-steering. Which makes the Jackal ideal for this study.

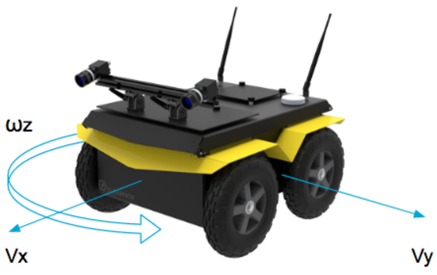

Fig. 1. Clearpath Robotics' Jackal with a representation of v_x, v_y and ω_z.

B. Theoretical setup

The next step, after choosing the ROS simulation environment and the Jackal robot, was to appoint the values of the prior. The goal of the brain was to control the velocities of the Jackal, so the values of the prior contain the following parameters: the forward velocity v_x, the sideways velocity v_y (slip) and the angular velocity ω_z. These values form the prior vector: $\xi = \begin{pmatrix} v_x & v_y & \omega_z \end{pmatrix}^T$.

After that the state-space capturing the basis of the Jackal's kinematics was created. As mentioned above, an LTI state-space formulation was used to implement Active Inference in the simulation. A standard LTI state-space looks as follows:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t) \tag{1}$$

$$\dot{\boldsymbol{y}}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t) \tag{2}$$

In this state-space, $x(t)$ is the state vector, $y(t)$ the output vector and $u(t)$ the input vector. A, B, C and D are matrices. In the following state-spaces only the formulas for the state x are mentioned, as the output vectors y are the same as the state vectors x because the C matrix is an identity matrix and D is equal to zero.

This first state-space (3) captures the properties and kinematics of the Jackal:

$$\begin{aligned}\begin{pmatrix} \dot{v_x} \\ \dot{v_y} \\ \dot{\omega_z} \end{pmatrix} &= \begin{pmatrix} -\frac{4d}{m} & 0 & 0 \\ \omega_{z_0} & -\frac{4d}{m} & -v_{x_0} \\ 0 & 0 & -\frac{(a^2+b^2)d}{I_c} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ \omega_z \end{pmatrix} \\ &+ \begin{pmatrix} \frac{2d}{m} & \frac{2d}{m} \\ 0 & 0 \\ -\frac{2db}{I_c} & \frac{2db}{I_c} \end{pmatrix} \begin{pmatrix} v_L(t) \\ v_R(t) \end{pmatrix}\end{aligned} \tag{3}$$

The states (x) of the Jackal consist of the forward velocity v_x, the sideways velocity v_y (slip velocity), and the rotational velocity ω_z. The inputs (u) of the system consist of the velocities of the two left wheels v_L and the two right wheels v_R. These two inputs only influence the forward velocity v_x and the rotational velocity ω_z directly, because it is not possible to give the Jackal a sideways velocity, as this is the slip velocity (skid steering). The A and B matrices consist of the properties of the Jackal; including the mass m, the moment of inertia I_c, the damping constant d, and the dimensions a (half of the length) and b (half of the width).

Finally, the matrix A contains two more parameters: ω_{z_0} and v_{x_0}. Intuitively, the slip velocity v_y is dependent on both the forward velocity v_x and the angular velocity ω_z, see Eq. 4.

$$\dot{v_y} = \omega_{z_0} * v_x - \frac{4d}{m} * v_y - v_{x_0} * \omega_z \tag{4}$$

In order to make the brain capable of handling the slip velocity v_y, the parameters ω_{z_0} and v_{x_0} are introduced. The values of these parameters were found by exploratory means, rather than a complex dynamical analysis. Calculating the exact and correct values of these parameters, or how v_y is dependent on v_x and ω_z exactly is beyond the scope of this study.

First, experiments were done giving ω_{z_0} and v_{x_0} values of either 0.5 or 1. However, intuitive reasoning gave the insight that ω_{z_0} and v_{x_0} needed to become variable and dependent on the forward and angular velocities. Secondly, the brain needs to influence the slip velocity v_y by changing the v_x and ω_z velocities. An unavoidable issue is that the brain will never be able to reach its prior when changing the slip velocity v_y, because it has to keep altering v_x or ω_z or both. To keep the deviation from the prior as small as possible, a solution was formulated with the following substantiation: if the forward velocity v_x is large, a relatively small angular velocity ω_z is needed in order to give the Jackal a slip velocity v_y. The same applies the other way around. Thus, ω_{z_0} and v_{x_0} need to change the amount v_x and ω_z influence the slip velocity respectively, as seen in matrix A, and Eq. 4. Considering all of the above, it was chosen to make the parameters ω_{z_0} and v_{x_0} represent the prior values of ω_z and v_x respectively.

The reason as to why a negative sign is placed in front of v_{x_0} is because when the Jackal is driving forward, a rotation in the negative z-axis results in a slip velocity in the positive y-axis. Thus, to create a positive slip velocity v_y, either ω_{z_0} or v_{x_0} needs to be negative. Depending on which of the two is negative, the direction of v_x and ω_z is inverted. In order to keep the motion in the x-direction positive the negative sign was added to v_{x_0}.

With this state-space of the Jackal kinematics (3) and some other variables the prediction (μ) and the action (u) are calculated:

$$\dot{\mu} = D\mu - \kappa(-\tilde{C}^\top \Pi_z(y - \tilde{C}\mu)) + (D - \tilde{A})^\top \Pi_w(D\mu - \tilde{A}\mu - \xi)) \tag{5}$$

$$\dot{u} = -\rho \tilde{G}^\top \Pi_z(y - \tilde{C}\mu) \tag{6}$$

These two formulas can also be written in state-space form, simplified as:

$$\dot{\mu} = A_\mu \mu + B_\mu \begin{pmatrix} y \\ \xi \end{pmatrix} \tag{7}$$

$$\dot{u} = A_u u + B_u \begin{pmatrix} y \\ \mu \end{pmatrix} \tag{8}$$

A schematic overview of how these state-spaces interact with each other can be seen in Fig. 2

Below, every parameter which is not already familiar is discussed and explained. The prediction μ (7) consists of the predicted velocities of v_x, v_y and ω_z and their derivatives. The number of derivatives it uses is p, which by default is six. These derivatives are called the generalized coordinates and are used to calculate predictions of future states. The prediction μ has as input the measurements of the velocities y and the prior ξ (the desired velocities). The action u (8) consists of the velocities v_L and v_R which are given to the Jackal and has as input the prediction μ and also the measurements y. The A and B matrices of these state-spaces consist of complicated Active Inference formulas:

$$A_\mu = D - \kappa(D - \tilde{A})^\top \Pi_w (D - \tilde{A}) - \kappa \tilde{C}^\top \Pi_z \tilde{C}^\top \tag{9}$$

$$B_\mu = \begin{pmatrix} \kappa \tilde{C}^\top \Pi_z & \kappa(D - A)^\top \Pi_w \end{pmatrix} \tag{10}$$

$$A_u = 0 \tag{11}$$

$$B_u = \begin{pmatrix} \rho \hat{G}^\top \Pi_z & \tilde{C} \rho \hat{G}^\top \Pi_z \end{pmatrix} \tag{12}$$

D is a divergence matrix, $\tilde{A}$ is the kronecker tensor product of an identity matrix and the A matrix of (3), $\tilde{C}$ is an identity matrix and $\hat{G}$ is the steady state gain matrix. For the details of these matrices and the derivation of these formulas, please look at the paper of Grimbergen [5].

This study focused on the precision matrix Π_z and the learning rates κ and ρ, also found in the above formulas (5) - (12). The precision matrix or inverse covariance matrix influences the perception on the covariance of the noise and the measurements. The learning rates κ and ρ influence how much the prediction and action are adjusted each step respectively. Changing the value of these four variables affects the behaviour and stability of the system.

Since the brain is supposed to control the states of the world, a closed loop system needed to be set up connecting the brain to the world. The whole setup of the closed loop system can be seen in Fig. 3. The output of the brain is the steering signal u (action). This signal, which consists of the velocities v_L and v_R, is then converted to the forward velocity v_x and the rotational velocity ω_z. This is done because the inputs of the state-space of the Jackal (3) are the left and right wheel velocities, but the inputs given to the Jackal are the forward and rotational velocities. The Jackal receives a steering signal and the simulation interface, Gazebo, extracts the states of the world x, in this case the exact measurements of the Jackal velocities. However, the frame of these velocities first have to be converted, as the measurements from Gazebo use a fixed frame, and the velocities in the state-space of the Jackal (3) use the frame of the Jackal. Secondly, in addition to a frame conversion, noise z, Gaussian white noise, has to be added in order to transform signal x into signal y. However, noise is only being implemented in answering the third sub research question; *does the algorithm filter out noise as it promises, or is it limited to a rather perfect world?* In all other segments of this study noise is not added.

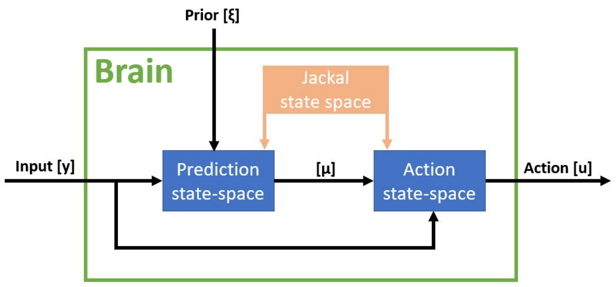

Fig. 2. Schematic overview of the brain (Active Inference controller) and the interaction between the formulas $\dot{\mu}$ and $\dot{u}$.

Now these measurements of the Jackal velocities (y), together with the prior ξ, go into the brain, where a prediction (μ) and a new steering signal (u) are calculated. The brain in Fig. 3 is a simplified version of the brain in Fig. 2.

C. Implementation in Python

After the theoretical setup was finalized, it needed to be coded so it could be interpreted by ROS. The programming language used was Python, as it is both understood by the ROS framework and a common programming language among engineers. As aforementioned, Grimbergen made a state-space formulation which he programmed in the most basic form in MATLAB [5]. The next step was to translate this code to the Python language. The actual code can be

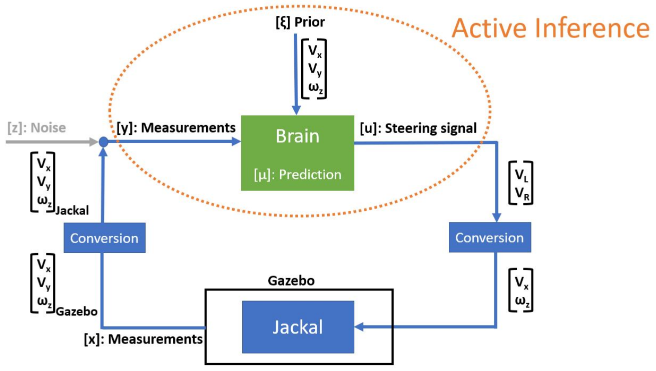

Fig. 3. Schematic overview of the complete setup.

found in Appendix IV. Three simplifications and improvements were made in order to fit the code to this study more closely:

1. The noise was removed. In the MATLAB code there were two forms of noise simulated, a noise z and a noise w. Noise z was added on the measurements to simulate an imperfect noisy sensor that measures the states of the world x. The noise w was added on the steering signal to simulate that the system is not able to perfectly alter the states of the world. Removing the noise is technically cheating according to the theory, since the biological brain is not omnipotent. However, since this is the first time ever Active Inference is tested in a practical setting, the implementation of the theory was done in the most basic form, starting without noise.

2. A multiple input multiple output (MIMO) state-space model was implemented. In the MATLAB files of Grimbergen [5], he simulated the states of the world as a simplified single input single output (SISO) state-space . As explained in the previous section, this project uses a MIMO state-space model (3). This meant that the base code needed to be altered in order to be able to operate with such a system.

3. The function “lsim” was removed. In the MATLAB files of Grimbergen [5], the brain and the world were simulated in a single closed loop system with the function “lsim”. This meant that the world state x, the measurements y, the prediction μ and the action u were all coded in one closed loop system. In this study, the world was not coded but was simulated with ROS in Gazebo. Therefore, the closed loop system was split open into a state-space of the brain and a state-space of the world and the function “lsim” was replaced with individual for loops for the prediction μ, the action u and the measurements y. All python codes can be found in appendix IV.

D. *ROS setup*

As stated earlier, the operating system ROS was used to program the simulation and the Gazebo Graphical User Interface (GUI) was used to visualize the Jackal driving inside the simulation. The friction coefficient in the ground plane in Gazebo was changed to represent the damping coefficient used in the A and B matrix used by the brain, (3). All other parameters remained unchanged during this study.

To understand the way ROS works, ROS nodes and topics need to be understood. Nodes send, receive and process information while topics carry this information between nodes. Every time a node publishes its information (e.g. velocity values) to a topic, other nodes receive this information. If all the nodes and topics are connected, a map as seen in Fig.4 is created.

Notice the following node loop within Fig.4:
Gazebo $\rightarrow$ *Frame Converter* $\rightarrow$ *Brain* $\rightarrow$ *Steering Signal Converter* $\rightarrow$ *Gazebo*
This loop is essentially the same loop as specified in previous section *Theoretical Setup*, see Fig. 3. The code of all these nodes can be found in appendix VII. Later on, when noise will be added, a noise node will be created and put in between the *Frame Converter* and the *Brain*. This node will then alter the information in topic *Measurement Jackal*, thereby creating noise, and turn it into the topic *Measurement Noise*, see Fig. 5. Furthermore, the record nodes are used to record data sent between nodes. This data is then plotted through Matlab into analyzable graphs. This Matlab code can be found in Appendix V. Lastly, three nodes without any connection lines or topics can be observed within Fig. 4. These nodes are active in order to make certain events happen. Node *gazebo gui* spawns the graphical user interface, *robot state publisher* can be used to extract information but is not used in this setup and the node *controller spawner* makes it possible to spawn certain objects into the world, like the Jackal.

Within the ROS files, the for-loop of measurements y was removed from the Python script. It was removed because the state of the world x and the measurements y going into the brain are not calculated in the Python code, but are the values read out of the simulated world within Gazebo. Therefore, the for-loop of the measurements y was removed and replaced with a callback function in ROS, explained in the next section.

The complete implementation from Python to ROS, is captured within the node *Brain*. Apart from slight alterations, all of the Python code is inside this node. For example, the for-loop was translated into a callback function and the “main” in Python was kept in the “main” part of the node. These translations and alterations make the code compatible with ROS. Conveniently, all of the callable functions are stored in the same directory as the *Brain*, so all these functions are available to the *Brain*. The code of the brain node is found in appendix VII.

E. *Measurement and processing of data*

As stated before, data was recorded from certain topics and stored in rosbag files. The concerned topics are *Measurements Jackal*, *cmd vel* and later on also the topic *Measurements Noise*. These rosbag files contain all the data the node publishes to the concerning topics and the publish time-stamps of this data. Afterwards, the rosbag files were imported into MATLAB and turned into useful plots, see

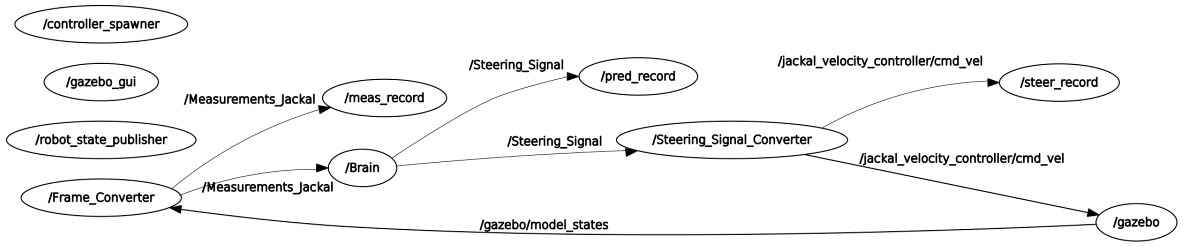

Fig. 4. Complete overview of all active ROS nodes, excluding noise.

appendix V. Every time a setup or test was completed, a rosbag was made, and imported into MATLAB where plots were made.
In order to obtain relevant results and make them comparable, certain parameters were set to fixed values. The most important parameters within this study were: the prior values v_x, v_y and ω_z, and the learning rates ρ and κ. The default values for the prior were set to: $\xi = \begin{pmatrix}0.5 & 0 & 0\end{pmatrix}^T$. A velocity of 0.5 was used, ensuring the Jackal will not make a wheelie or flip over and thereby changing the kinematics of the Jackal's system. The prior values for the v_y and the ω_z were both set to a value of 0, as only giving a value for the prior v_x will be the easiest for the brain to execute. The values for ρ and κ were found using trial and error, and are 50 and 1 respectively. The above values do not completely optimize the algorithm, however they do show stable results. From now on these fixed values will be referred to as the *default values* or the *default setup*.

1) Does the system succeed in controlling the velocities v_x, v_y and ω_z of a simple robot vehicle in a simulation using Active Inference?: The first objective was to verify whether the brain succeeded at controlling each of the prior values separately. This was done by conducting a test using the default setup, followed by simple variations to the prior:

1) Prior $\xi = \begin{pmatrix}0.5 & 0 & 0\end{pmatrix}^T$, with learning rates $\rho = 50$, and $\kappa = 1$.
2) Prior $\xi = \begin{pmatrix}0 & 0 & 0.5\end{pmatrix}^T$, with learning rates $\rho = 50$, and $\kappa = 1$.
3) Prior $\xi = \begin{pmatrix}0 & 0.5 & 0\end{pmatrix}^T$, with learning rates $\rho = 50$, and $\kappa = 1$.

Notice that the priors for v_x and ω_z can be controlled directly, whereas the prior for v_y can only be controlled indirectly. Therefore, the third test was the most difficult for the brain to achieve. It was physically impossible to execute for the Jackal because it cannot move with a sideways velocity v_y without a forward and angular velocity. Therefore, it turned out to be crucial to prioritize v_y in order to achieve its relative prior. Another problem was that the A matrix in (3) on p. 2 is partially dependent on the prior. Namely, since the slippage is linearized, the effect of v_x and ω_z on v_y is represented in the simplified formula in (4) on p. 2. Notice that this formula is derived from (3) on p.2, with v_x and ω_z left out. Here v_{x_0} and ω_{z_0} were set equal to the prior value of the brain as explained in the *Theoretical setup*. Thus, when the prior of $\xi = \begin{pmatrix}0 & 0.5 & 0\end{pmatrix}^T$ was given, the brain did not understand that it could affect the Jackal's slip velocity v_y by changing its other velocities v_x and ω_z, due to $\dot{v_y}$ only being dependent on v_y when ω_{z_0} and v_{x_0} are zero. Both of these problems are addressed and further explained in the next subsection *Prioritizing prior values*.

2) Is it possible to make the algorithm prioritize one of the velocities v_x, v_y and ω_z separately?: To further investigate and optimize the brain, changes to the code have to be made in order to make the brain prioritize certain prior values. Specifically, it would be interesting to test what the brain will do when it is given a prior only for v_y and thinks both the v_x and ω_z directions are unimportant. Hopefully it will stabilize around the correct slip velocity v_y, and choose a stable v_x and ω_z to manage this. This prioritization will be made possible by altering the Π_z matrix.

The objective is to separate the individual priors ξ for v_x, v_y, and ω_z, since in the default setup the brain can only satisfy all the priors at the same time. This results in the brain wanting to reach all desired velocities simultaneously until the prior is reached, which for some priors, including prior $\xi = \begin{pmatrix}0 & 0.5 & 0\end{pmatrix}^T$, is impossible.

In order to resolve this issue, a closer look at the formula for $\dot{u}$ in (6) is necessary. It is desired that the steering signal only tries to satisfy the prior for v_y even though it can not directly do so. In this equation the variable $\hat{G}$ represents the steady state gain, which is related to the effect action has on the measurements based on the Jackal's state-space (3). This $\hat{G}$ is multiplied with the Π_z, which is the precision (or inverse covariance) matrix between the noise z and the measurements y as explained by Grimbergen [5].

In the default setup for this study there is no noise z yet thus the covariance should be infinite. However, it is interesting to change the Π_z matrix in such a way to make the brain think that there is infinite noise z on the irrelevant measurements y in order to make them unimportant for reaching the prior. This means by making the first and third value of the diagonal $\frac{1}{\infty} = 0$, thus:

$$\Pi_z(old) = \begin{pmatrix}0.1 & 0 & 0\\ 0 & 0.1 & 0\\ 0 & 0 & 0.1\end{pmatrix} \tag{13}$$

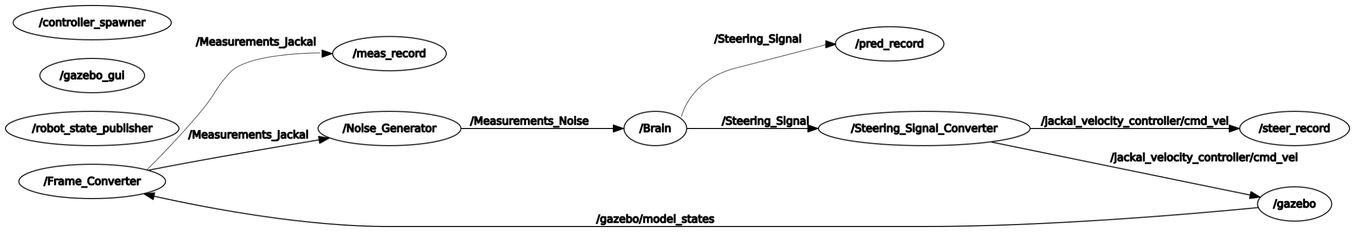

Fig. 5. Complete overview of all active ROS nodes, including noise.

is changed to

$$\Pi_z(new) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{14}$$

With this Π_z the steering signal u will hypothetically only be able to affect the second state v_y.

As the last subsection states the A matrix has to be altered in order for the brain to control the Jackal at all. So, for now a quick fix due to time constraints is that the values in the A matrix from the Jackal are changed manually in the following way, note this is only done for answering the current research question:

$$A(new) = \begin{pmatrix} -\frac{4d}{m} & 0 & 0 \\ 0.5 & -\frac{4d}{m} & -0.5 \\ 0 & 0 & -\frac{(a^2+b^2)d}{I_c} \end{pmatrix} \tag{15}$$

Now, these two concepts (changing the A matrix and changing the Π_z value) need to be used together. The brain should understand both: that it can change the velocities v_x and ω_z in order to alter velocity v_y; and that meeting the prior values v_x and ω_z is not important. Thus, it can optimize only towards its prior for v_y.

A test with the following parameters will be performed in order to see how these changes affect the brain and will be discussed in the next section *Results*:

- Prior $\xi = \begin{pmatrix} 0 & 0.3 & 0 \end{pmatrix}^T$, the learning rates ρ and κ are set to the default values, the Π_z matrix and the A matrix are altered according to (14) and (15) respectively.

The value for v_y was lowered, from 0.5 to 0.3, because exploratory research showed that 0.3 is a lot easier to reach for the Jackal than 0.5.

3) Does the algorithm filter out noise as the theory promises, or is it limited to a perfect world?: The default setup is not a good representation of the real world, as no noise is present in the system. Therefore, noise needs to be added to see if this theory indeed works in unpredictable real world situations, like it is claimed to be.
Gaussian white noise was added to the *Measurements Jackal* topic within ROS by the *Measurements Noise* node, as shown in Fig.5. The code of this node can be found in appendix VII. From here, the measurements including the noise are sent to the brain. The tests were done by adding levels of noise of increasing order. Tests with the following values for the standard deviation σ were conducted:

- $\sigma = 0.005$
- $\sigma = 0.01$
- $\sigma = 0.05$
- $\sigma = 0.1$
- $\sigma = 0.5$
- $\sigma = 1$

Notice the last two tests, with a σ equal to 0.5 and 1, consist of unreasonably high standard deviation, thereby testing whether the brain can handle absurdly large uncertainties. More tests with other values for the standard deviation were done and can be found in appendix I.

4) How do different learning rates within the algorithm affect the stability of the system?: As both learning rates ρ and κ influence how much the prediction and action are adjusted every time step, high values for both are preferred in order to reach the desired prior values in the fastest way. However, such high values for ρ and κ might influence the stability of the Jackal state-space system.

To test if the above is the case for high learning rates the following tests were conducted:

- Default setup with learning rate $\rho = 1 * 10^5$.
- Default setup with learning rate $\kappa = 1 * 10^7$.
- Default setup with learning rates $\rho = 1 * 10^5$ and $\kappa = 1 * 10^7$.

To verify whether the Jackal state-space system is stable, pole-zero plots were made. It is important to notice that the above tests were conducted by the ROS simulation, whereas the following pole-zero plots were made with code from the Python simulation (see appendix IV for these Python codes). For a system to be stable, the region of convergence (ROC) must include the imaginary axis. The ROC is the open region to the right of the pole with the greatest real value of any pole in the system. This means that all poles must lie in the left half of the s-plane for BIBO stability to occur. Also, the system needs to be linear time-invariant and causal. This is the case as the system is linear time-invariant, see (3) on p.2, and causal as it only uses data from previous and current time steps.
Using the default setup with varying learning rate ρ, many different pole-zero plots were plotted into the same figure,

with varying values for ρ. Data obtained from these plots will show whether the system will remain stable for different learning rates and provide knowledge which (high) values of ρ are optimal for the algorithm. In order to compare the stability found from the pole-plots with the test above, the following pole-zero plots have been made:

- Default setup, where ρ is gradually raised from default value of 50 to a value of $5 * 10^7$.
- Default setup, where κ is gradually raised from default value of 1 to a value of $1 * 10^7$.
- Default setup, where both ρ and κ are gradually raised from default value of 50 to a value of $5 * 10^7$ and 1 to 1*107 respectively.

IV. Results

In this section the results of the four research questions will be discussed. The most important graphs, figures, and plots are included. All other results can be found in appendix I, II and III. In Figs. 6-12 the following parameters are plotted:

- the measurements of the velocities received by the brain (meas $[y]$).
- the prediction of the velocities by the brain (pred $[\mu]$).
- the steering signal received by the Jackal (action $[u]$)

These three parameters are represented by the blue, red and green lines in the plots respectively. On the x-axis the time is plotted in seconds and on the y-axis the velocities are plotted in m/s or rad/s.

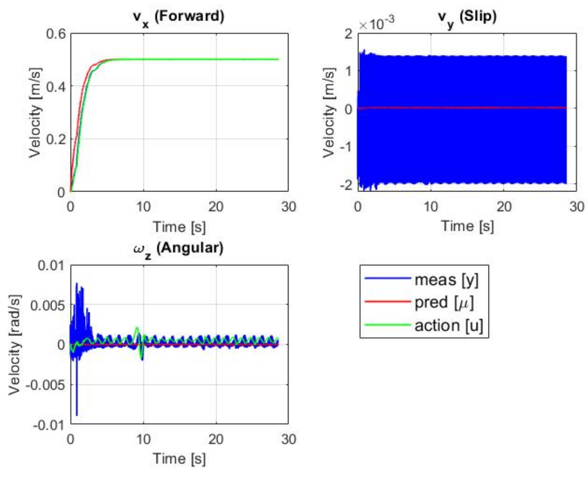

Fig. 6. This figure shows the velocity plots of the default setup (with prior $\begin{pmatrix}0.5 & 0 & 0\end{pmatrix}^T$). It shows that the desired velocity is reached in 5 seconds (top left plot). Note that the y-scale on the other two plots is very small.

1) Does the system succeed in controlling the velocities v_x, v_y and ω_z of a simple robot vehicle in a simulation using Active Inference?: The outcomes of the first series of tests (see p.5) using the default setup with varying priors, are presented respectively in Fig. 6 through 8.

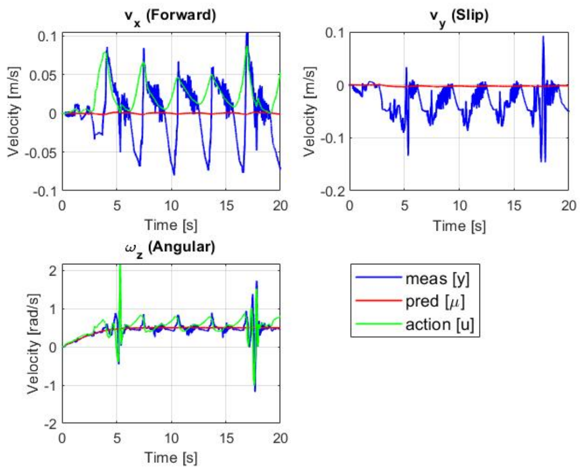

Fig. 7. This figure shows the velocity plot of the default setup with prior $\begin{pmatrix}0 & 0 & 0.5\end{pmatrix}^T$. It shows the desired velocity is reached at 7 seconds (bottom left plot). However, there is still repeating patterns in all plots due to slippage through skid steering and two anomalies at 5 seconds and 17 seconds.

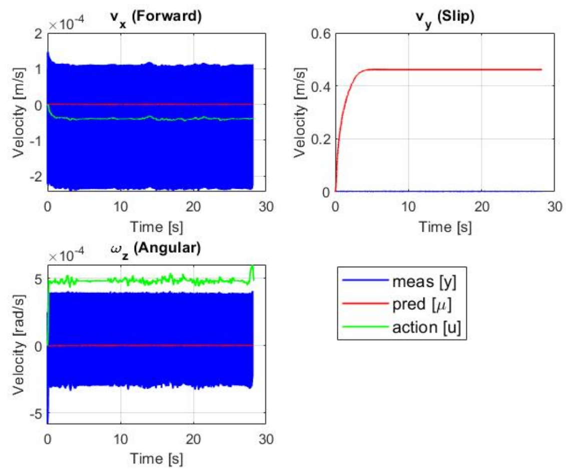

Fig. 8. This figure shows the velocity plot of the default setup with prior $\begin{pmatrix}0 & 0.5 & 0\end{pmatrix}^T$. It shows that the desired velocity is not reached (top right plot) as the measurements remain zero. The prediction (top right plot) does reach the desired velocity and does not follow the measurements. Note that y-scale on the other plots is very small.

Fig. 6 shows the theory was implemented successfully as both the prediction and the measured Jackal velocities converge towards the set prior $\xi = \begin{pmatrix}v_x & v_y & \omega_z\end{pmatrix}^T = \begin{pmatrix}0.5 & 0 & 0\end{pmatrix}^T$. Notice that the prediction lines and measurement lines stay close together, which means that the brain is accurately predicting the actual Jackal velocities and sending correct steering signals to the Jackal in order to reach the prior.

Fig. 7 shows decent behaviour as well. The prediction remains stable and converges towards the prior. The measurement oscillates around the prior in, interestingly enough, repeating patters, but ultimately stays stable as well. Notice how the brain tries to meet the prior $v_y = 0$, but is not able to keep the slip velocity at 0 m/s. Both the repeating pattern and the slip velocity not being zero can be explained by the skid-steering of the robot, because when the robot turns slip (v_y) will be present, as can be seen in fig. 7 as well. The two anomalies in the angular velocity graph at 5 and 17 seconds are definitely also interesting behaviour, but the reason for this behaviour has not been explored during this study.

Fig. 8 shows the test which was theorized to be set up for failure as described in the section *Measurements and processing of data*. The test was still done in order to examine the predictions of the brain. The prediction of the slip velocity v_y as seen in the top right graph converges to the prior while the measured v_y remains zero. This confirms the assumption that this test would fail. Nonetheless, it is still interesting that the prediction of v_y does not coincide with the measurement. However, no further investigation as to why this abnormality occurred has been done due to time constraints.

brain still has difficulty in achieving the prior. This could be due to the fact that the learning rates are far from optimal for this type of prior. Regardless, the measurements for v_x and ω_z seem to be very noisy compared to the smooth steering signal u and the smooth prediction for v_y. This could also be because of the change to Π_z, since the steering signal u is also dependent on the prediction μ. It could also be because the state-space of the Jackal is too much of an oversimplification to approximate slipping behaviour and this problem is simply too difficult to implement in such an early stage of research and development of Active Inference in robotics. Finally, the steering signal starts diverging from the average of the measurements, which could be due to the slip rapidly changing, making the system unable to handle the prior. However, the reason behind the brain's behaviour could be because a plethora of reasons, too many in fact for the scope of this study. Ultimately, this is probably the most interesting test result and would be great to be build upon in a future study.

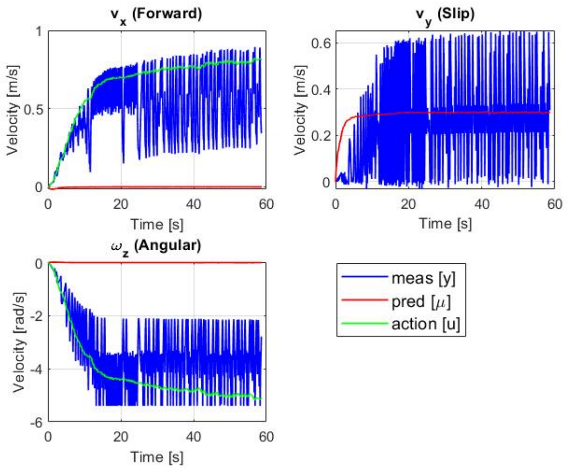

Fig. 9. This figure shows the velocity plot of the default setup with prior $\begin{pmatrix} 0 & 0.3 & 0 \end{pmatrix}^T$ and altered matrices A an Π_z. It shows an attempt is made to reach the desired velocity (top right plot), but the measurements do not necessarily converge towards the desired velocity. Note that the steering signal is not the same as the average of the measurements over time. Note that the run time for these plots is twice as long.

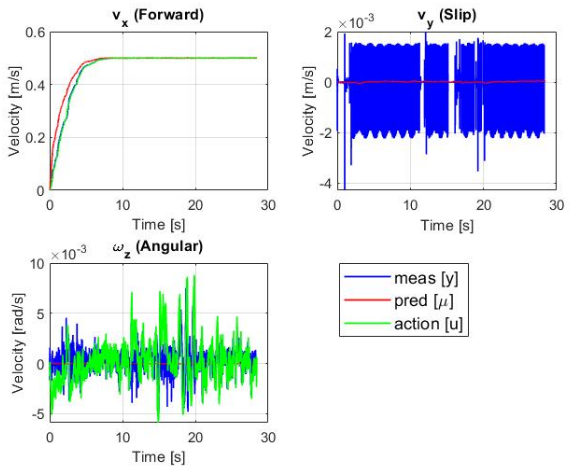

Fig. 10. This figure shows the velocity plot of default setup with noise ($\sigma = 0.01$). It shows the desired velocity is reached at 7 seconds (top left plot). Note that the y-scale of the other two plots is very small and that the measurements are recorded without the added noise.

2) Is it possible to make the algorithm prioritize one of the velocities v_x, v_y and ω_z separately?: The outcome from the test where matrix A in (3) was manually altered and where Π_z was altered in order to prioritize the prior for v_y, can be seen in Fig. 9. In this test the prior ξ was set to $\begin{pmatrix} 0 & 0.3 & 0 \end{pmatrix}^T$ and the learning rates ρ and κ were set to the default values.

The tests show better results in Fig.9 compared to Fig.8. The first notable thing that happens, is that the predictions of v_x and ω_z are zero. This is probably due to the fact that the Π_z parameter affects both the prediction μ (5) and the action u (6) on p.3. The second notable thing is that the brain does make an attempt to reach the desired prior of v_y, however the

3) Does the algorithm filter out noise as it promises, or is it limited to a rather perfect world?: Fig. 10-12 correspond to the tests of the default setup with added noise. Only three tests out of six tests are depicted as this shows enough information to make conclusions. The graphs corresponding to the other tests can be found in appendix I, see Fig. 16 to Fig. 18.
The figures 10-12 clearly show that the brain handles noise in a respectable manner, compared to the graph with the default setup without noise, see Fig. 6 on page 7.
Fig. 10 shows that for little noise the brain is very capable of keeping the Jackal's velocities at the prior values. The measurements, predictions and steering signal all show no odd behaviour. The graphs depicting the ω_z and v_y velocities might look inaccurate but the scales are tiny so differences

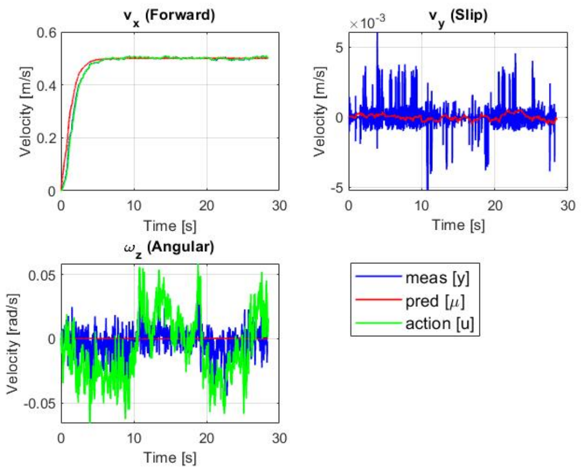

Fig. 11. This figure shows the velocity plot of default setup with noise ($\sigma = 0.1$). It shows the desired velocity is reached at 5 seconds (top left plot). Note that the y-scale of the other two plots is very small and that the measurements are recorded without the added noise.

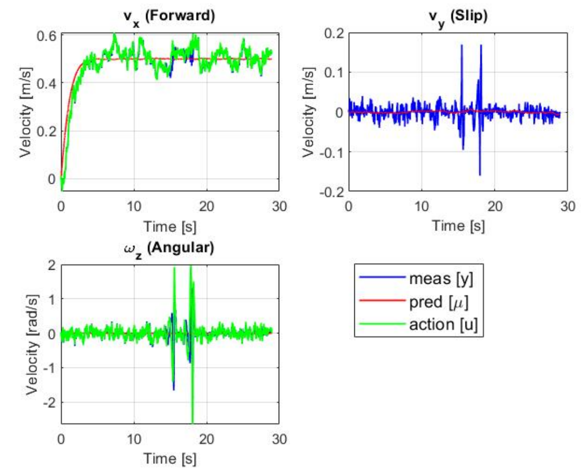

Fig. 12. This figure shows the velocity plot of default setup with noise ($\sigma = 1$). It shows the desired velocity is reached in 5 seconds (top left plot). However the measurement and action lines start to become clearly less stable. Note that there are two anomalies at 16 and 18 seconds and that the measurements are recorded without the added noise.

are negligible. As the standard deviation of the noise is increased, the deviations of the measurements and the steering signals relative to the predictions increase as well, seen in both Fig. 11 and Fig. 12. To give a further indication of how well the brain filters the noise and how big the noise actually is, appendix I can be looked at, see Fig. 19 to Fig. 24, where the measurements with noise can be compared to the outgoing steering signal. In these figures, the measurements with noise and the steering signal are depicted in blue and green respectively. Again, observe how constant the steering signal is compared to the measurements with noise.

These results show the brain filters (Gaussian white) noise extremely well, even when the standard deviation is raised to relatively absurd values.

4) How do different learning rates within the algorithm affect the stability of the system?: Figure 13 gives the insight that increasing the ρ in our default setup results in unstable behaviour. Using trial and error, it was found that the system becomes unstable when ρ reaches a value of approximately 10.000. Within the figure, the prediction converges towards the prior, while the measured velocities, v_x, v_y and ω_z, and the steering signals of v_x and ω_z start oscillating and diverging from the prediction. Notice that steering signal of v_x becomes unstable somewhere between 1 and 2 seconds and goes to minus infinity. A zoomed out picture can be found in Appendix II, see Fig. 25. The measurements and predictions of the velocities do not go to infinity but do not converge either and thus are marginally stable. If the same applies to other priors is not tested. Also, by looking at the difference between the prediction and the measurement, it can be found that the brain does not predict the Jackal behaviour correctly, which is curious, but has not been explored further.

Figure 14 shows that a high kappa does not inherently change the behaviour of the default setup. If this is also the case for other priors is not tested. Notice that the measurements and predictions actually converge quicker than the first test with the default setup.

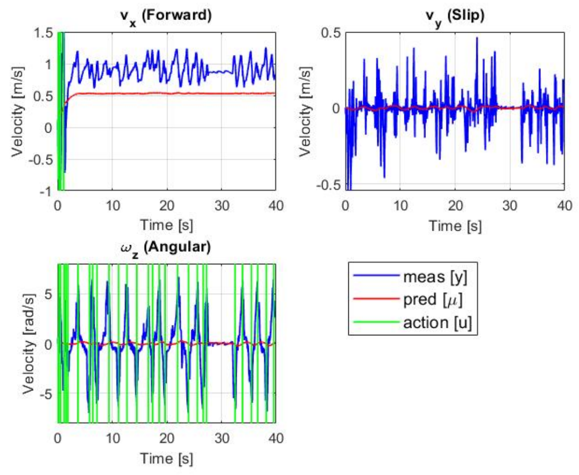

Fig. 13. Velocity plot of default setup with $\rho = 1 * 10^5$, zoomed in. Note that the action line, top left plot, goes to minus infinity and that the measurements do not oscillate around the prior or prediction values for this top left plot.

As followed from the results from the ROS simulation, it seems like ρ impacts the stability more than κ does. Therefore the pole-zero plot of varying values of ρ will be analyzed first.

In the plot in Fig. 15 the default setup is used, but with values for ρ varying between 50 and 5 million. In the left plots the zeros are shown and on right the poles are shown. In order to prevent extreme clustering of the poles and zeros,

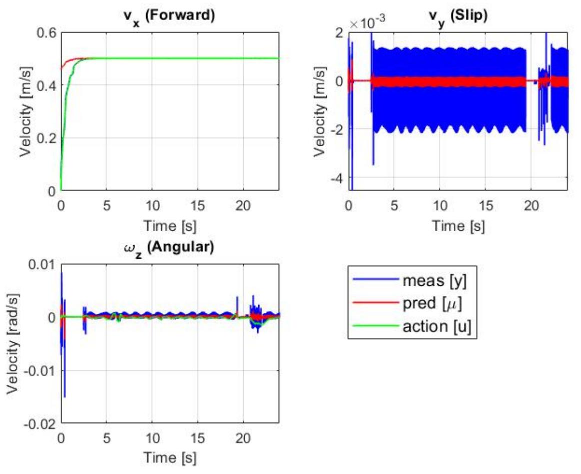

Fig. 14. Velocity plot of default setup with $\kappa = 1{*}10^7$. It shows the desired velocity is reached at 2.5 seconds, note that the prediction line converges even faster towards the prior, top left plot. Note the y-scale on the other two plots is very small.

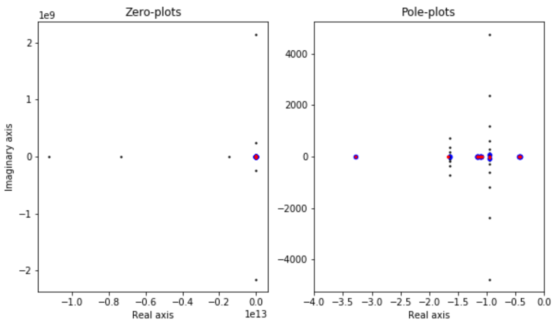

Fig. 15. Pole-zero plot for varying values of ρ. They are plotted with default values with ρ varying between 50 and $5 * 10^7$. The size of ρ is multiplied by 4 for every plot, and are all plotted within the same figure. Red dots represent poles and zeros resulting from default values, blue dots with approximately the first unstable value of $\rho = 1 * 10^5$ and black every other zero and pole. Note that there are no poles or zeroes with positive x-values in these plots.

the vector for ρ is chosen to multiply its previous value 4 times. This means if the first value is equal to 50, the second will be 200, the third 800, etc. Zoomed-in versions of the pole-zero plots can be found in Appendix II, see Fig. 26. The red dots represent the poles and zeros obtained using the default value of ρ. The blue dots represent the poles and zeros obtained using the default setup, but with ρ set to 10.000, which is the point at which the system becomes unstable, as was found in previous findings. The black dots represent all the other poles and zeros.

The plot confirms that the system is indeed stable for the default values, as all red-dotted poles are within the left-half plane. However, the results do not coincide with the findings for high ρ in the ROS simulation done previously. The Jackal state-space system remains stable, even for very high values of ρ, whereas results obtained from the simulation (see Fig. 13) showed the system becomes unstable when a high learning rate of $\rho = 10.000$ is used. The results show it is simply not accurate enough to make a pole-zero plot of the whole state-space system within the Python code and expect the simulation, with the Jackal robot, to behave the same way. In other words, a stability of the whole system within the Python code does not translate to a stability of the system within the ROS simulation (Python code can be found in Appendix IV). While this difference can simply be denoted to the state-space not being an accurate enough representation of the Jackal kinematics, the problem is probably more complex. This conclusion was derived by checking the Jackal's movements; the Jackal was accelerating quickly and thereby constantly making wheelies and driving on two wheels. This caused a drastic change in behaviour and kinematics, resulting in the plots of Fig. 13 and 14 being inadequate. This unwanted behaviour can most probably be denoted to ρ, the learning rate of the action u, being too high and making the action, steering signal u, change too quickly for the Jackal.
The other tests, in which κ was raised to high values and both learning rates were raised to higher values, were not further analyzed in this paper because the previous test already showed this method is not accurate enough and the ROS environment might need some constrictions, considering acceleration. Therefore, these results are not relevant enough to discuss, however plots of these tests can be found in Appendix II, Fig. 27 and 28.

V. Discussion

As the study to further understand the theory of Active Inference is still in its infancy, it is hard to say whether it will actually accelerate artificial intelligence research. However, as the theory shows promising results within the neuroscientific world, it is important to do more research on Active Inference in the field of robotics as well. This study has made the first step of implementing this theory of Active Inference in robotics. Using the results from this study, it can be observed whether some theoretical occurrences also work in a simulated environment. From this, flaws in the theory may be addressed and possibly improved. Also, the study gives a better visualization of the complex theory and helps with further understanding of the theory.

A. *Error analysis*

In order to implement the state-space formulation into a simulated world, multiple simplifications and assumptions were made. The four most important ones are listed below.

Firstly, the values v_{x_0} and ω_{z_0} in the A matrix of (3) on p. 2 were found by exploratory means instead of by a full dynamical analysis. The values used are placeholders in order to represent correlations between the slip velocity and the forward and angular velocity.

Secondly, equations (5) and (6) on p. 3 represent the state-space of the brain. From these state-space equations, it is clear that the whole state-space will change when altering the learning rates, as both ρ and κ are not just gains to the system. Therefore, trying to optimize the learning rates is difficult, as the state-space is constantly changing when altering the learning rates. Also, even if perfect learning rates were obtained, these would only work for certain set priors. When changing the priors to other values, the optimal learning rates change as well.

Thirdly, the prediction μ shows some odd behaviour which this study has not been able to explain. When the default setup is used with a prior for the slip velocity, the prediction converges to the prior while the measurements stay at zero. This could be due to an error in the code, an error in the formulation or maybe even an intrinsic quality corresponding to the theory of Active Inference.

Finally, a maximum value of the Jackal's acceleration was not accounted for. Especially with high learning rates the Jackal occasionally makes a wheelie, disturbing its kinematics. This occurs most likely because it tries to reach the desired velocity too fast and accelerates too fast. The state-space of the Jackal does not take these undesired movements into account, thereby disrupting the algorithm.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. *Summary of results and conclusions*

The goal of this paper was to present a novel implementation of the Active Inference theory on a simple mobile robot, the Jackal. The resulting Active Inference algorithm was used to estimate and control the velocities of this robot. To reach this goal, the following four research questions were answered:

1) Does the system succeed in controlling the velocities v_x, v_y and ω_z of a simple robot vehicle in a simulation using Active Inference?: Research was done whether the brain could succeed in controlling the velocities v_x, and ω_z of the Jackal separately. For the priors $\begin{pmatrix} v_x & v_y & \omega_z \end{pmatrix}^T$ equal to $\begin{pmatrix} 0.5 & 0 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 0 & 0.5 \end{pmatrix}^T$ the theory was implemented successfully as both the prediction as well as the measured Jackal velocities converge towards the priors. Important to observe that the prior of $\begin{pmatrix} 0 & 0 & 0.5 \end{pmatrix}^T$ showed oscillations in v_y direction, proving the Jackal skid steers when having an angular velocity. Results for controlling velocity v_y separately were negative, more on this in the next paragraph.

2) Is it possible to make the algorithm prioritize one of the velocities v_x, v_y and ω_z separately?: Consecutively, research was done to overcome the problem that the Jackal is physically unable to move in the sideways v_y direction when the other priors where given a value of 0. The solution to this problem consist of prioritizing the v_y prior. This was done by changing the Π_z matrix which gave far better results than the default setup. Also successful changes to the A matrix were made to maintain the brains capabilities of controlling v_y whilst v_x and ω_z were zero. However, results are still far from optimal and a large amount of potential research could be done in this field.

3) Does the algorithm filter out noise as the theory promises, or is it limited to a perfect world?: In order to test whether Active Inference can indeed deal with noise as well as is claimed to be, Gaussian white noise was added to the measurements. It can be concluded that the brain can indeed filter noise, with reasonable standard deviations, in a desirable way. The brain is able to correctly achieve the desired priors.

4) How do different learning rates within the algorithm affect the stability of the system?: Lastly, the stability of the system with varying learning rates ρ and κ was studied. The results showed the system becomes unstable when ρ reaches a value of 10.000, whereas κ does not inherently seem to influence the stability. The reason for instability for high ρ is still unclear. Pole-zero plots of the system were also studied from the Python code, but relevant results were not found.

All the sub questions considered, it can be concluded that it is at least possible to use a controller based on Active Inference. Whether it is feasible, can not yet be answered, as this study only explored the capabilities of controlling the velocities of a Jackal in a ROS simulation. More study needs to be done on Active Inference in robotics, to fully discover the benefits of Active Inference in robotics.

B. *Recommendations for future studies*

As this study is only the first to implement Active Inference in robotics, plenty of further future research can be done on this subject. In this section we present the four most interesting topics for further research.

Firstly, more research could be done on the prioritization of prior v_y. This study made some alterations to the matrices A and Π_z in order to try to prioritize this slip velocity and acquired some interesting results, but the behaviour has not been explained yet. It would be interesting to study this behaviour and find a way to accurately prioritize the slip velocity in a way that the Jackal can move with constant slip.

A second study could be done on the addition of different noise types. As Friston [3] claims that Active Inference is able to cope with noise with exceptional effectiveness, it would be interesting to further explore this. This study has made the first step by adding Gaussian white noise to the measurement and the systems seems to be able to deal with it very well. Next research topics can include the addition noise in different ways:

- Instead of just adding noise to the measurements, noise can also be added to the steering signal. This would be a more accurate model of the real world as an action will not perfectly alter the states of the world.
- Another type of noise can be added to the measurements. In this study Gaussian white noise is used, but modelling other types of disturbances, like dependent

noise, could give interesting results on how the system would react to realistic, noisy scenarios.

- An irregular surface terrain can be included. This study uses a flat and homogeneous terrain, but to simulating a more accurate world-like terrain would be more realistic. Hills, bumps and different types of terrain could be implemented to investigate how the system reacts to irregularities in the environment.
- The most realistic way of adding noise is by implementing this setup in a real Jackal and use top view cameras to measure the velocities. It would be interesting to investigate whether the Jackal will still be capable of controlling its velocities in a real environment. Eventually, even more noise could be added by measuring the velocities with GPS or with on board cameras.

Thirdly, more research can be done on the stability of the system. This study made a beginning on investigating the effect of the learning rates κ and ρ on the stability of the system, but no satisfying conclusions have been made yet. More research can be done on how the learning rates influence the zeros and poles and the stability of the system. Besides, investigating which other parameters significantly influence the stability could give more insight in how Active Inference operates and would therefore be very interesting.

Lastly, the A matrix of the state-space of the Jackal needs to be studied thoroughly and improved. Out of curiosity more exploratory research was done by adjusting the values v_{x_0} and ω_{z_0} in the A matrix. This had some major effects on reaching prior of v_y. Using trial and error, improved result of reaching the prior of the slip velocity were found. A graph of this more desired behaviour can be found in Appendix III, Fig. 29. Further improvements of the A matrix could be found through:

- More exploratory research using trial and error can be conducted in order to further optimize the A matrix and the behaviour of the Jackal robot.
- A detailed dynamical analysis, using techniques such as Taylor expansions, can be carried out to find out which values are correct and give optimal results.
- Making the A matrix self-taught instead of predefined. Currently, the most obvious method of doing this is by introducing machine learning through a concept called 'expectation maximization' [3]. This could circumvent the whole issue of having to dive into dynamical theory in order to improve the brain.

Acknowledgements

This project has been commissioned by the department of Cognitive Robotics under supervision of Prof. dr. ir. M. Wisse. We would like to express our appreciation to him for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously has been very much appreciated.

References

[1] K. J. Friston, J. Daunizeau, J. Kilner, and S. J. Kiebel, "Action and behaviour: a free-energy formulation," *Biological cybernetics*, vol. 102, no. 3, pp. 227–260, 2010.

[2] K. J. Friston, "The free-energy principle: a unified brain theory?" *Nature Reviews Neuroscience*, vol. 11, no. 2, pp. 127–138, 2010.

[3] K. J. Friston, N. Trujillo-Barreto, and J. Daunizeau, "Dem: A variational treatment of dynamic systems," *NeuroImage*, vol. 41, no. 3, pp. 127–138, 2008.

[4] C. L. Buckley *et al.*, "The free energy principle for action and perception: A mathematical review," *Journal of Mathematical Psychology*, vol. 81, pp. 55–79, 2017.

[5] S. S. Grimbergen, "A state space formulation of active inference," *in preperation*.

VII. APPENDIX

APPENDIX I
GRAPHS REGARDING NOISE

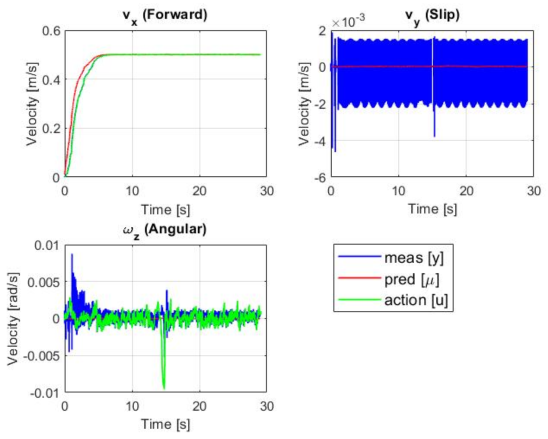

Fig. 16. Velocity plot of default setup with noise ($\sigma = 0.005$). Plotted are the measurements recorded without added noise. It shows the desired velocities are reached at 5 seconds, top left plot. Note the y-scale on the other two plots is very small.

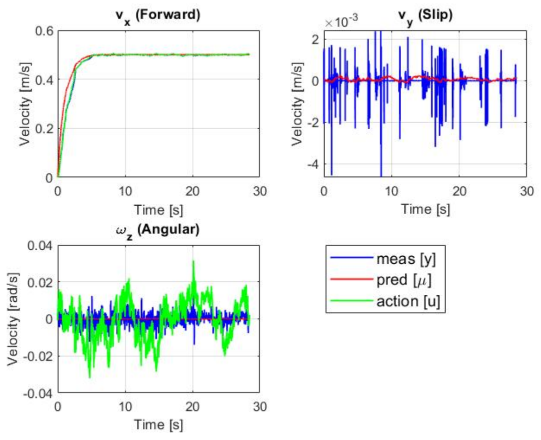

Fig. 17. Velocity plot of default setup with noise ($\sigma = 0.05$). Plotted are the measurements recorded without added noise. It shows the desired velocities are reached at 4 seconds, top left plot. Note the y-scale on the other two plots is very small.

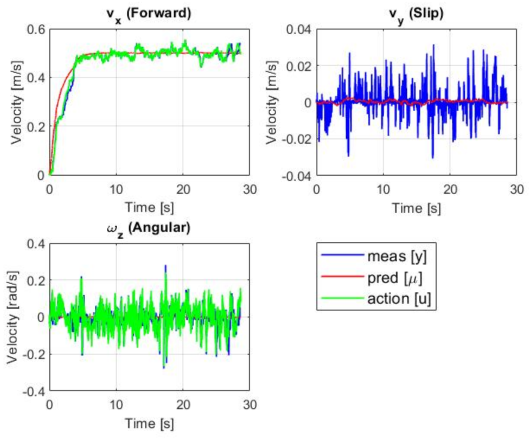

Fig. 18. Velocity plot of default setup with noise ($\sigma = 0.5$). Plotted are the measurements recorded without added noise. It shows the desired velocities are reached at 5 seconds, top left plot, however are not as stable as previous figures. Note the y-scale on the other two plots is very small.

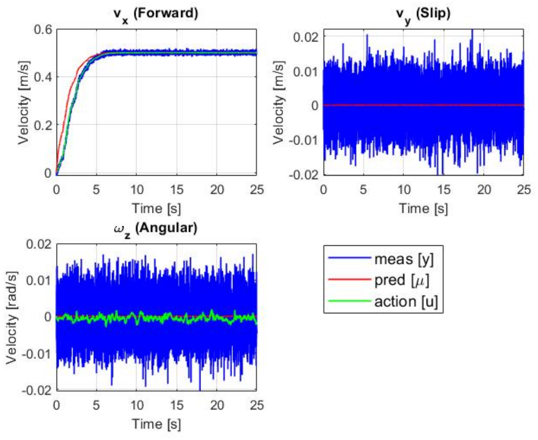

Fig. 19. Velocity plot of default setup with noise ($\sigma = 0.005$). Note these are the exact same tests as the figures above, and in section*Results*, with corresponding σ. Only the measurements are recorded with the added noise.

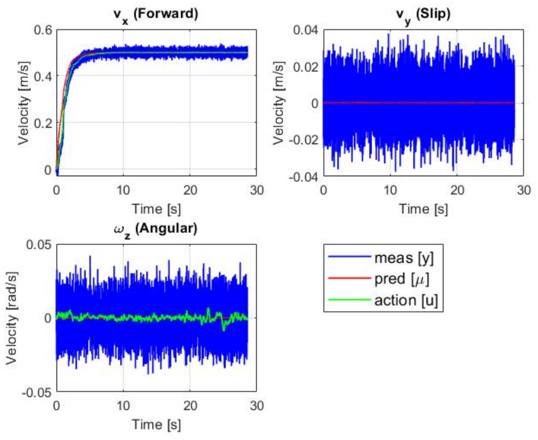

Fig. 20. Velocity plot of default setup with noise ($\sigma = 0.01$). Note these are the exact same tests as the figures above, and in section*Results*, with corresponding σ. Only the measurements are recorded with the added noise.

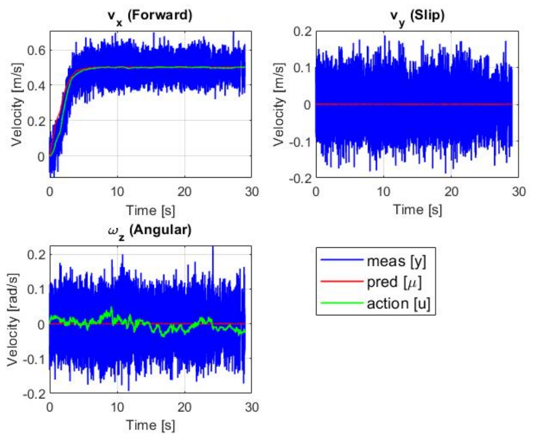

Fig. 21. Velocity plot of default setup with noise ($\sigma = 0.05$). Note these are the exact same tests as the figures above, and in section*Results*, with corresponding σ. Only the measurements are recorded with the added noise.

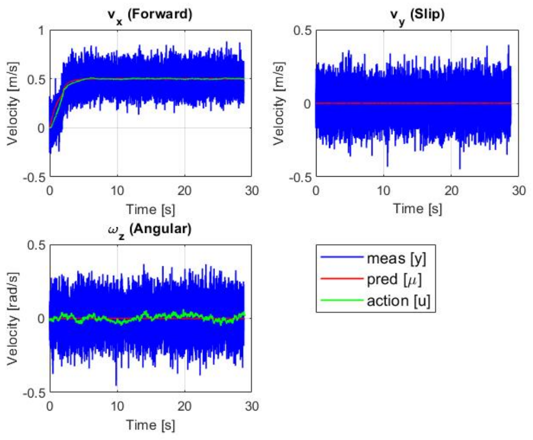

Fig. 22. Velocity plot of default setup with noise ($\sigma = 0.1$). Note these are the exact same tests as the figures above, and in section*Results*, with corresponding σ. Only the measurements are recorded with the added noise.

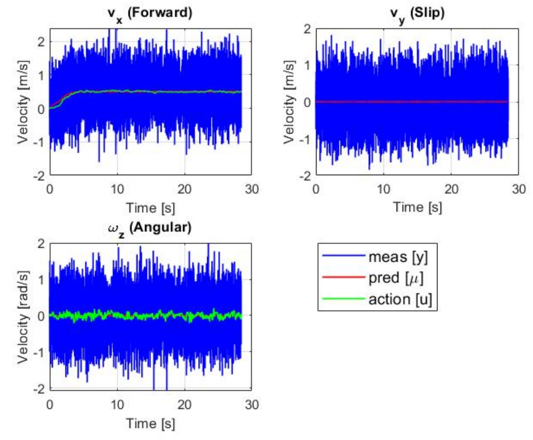

Fig. 23. Velocity plot of default setup with noise ($\sigma = 0.5$). Note these are the exact same tests as the figures above, and in section*Results*, with corresponding σ. Only the measurements are recorded with the added noise.

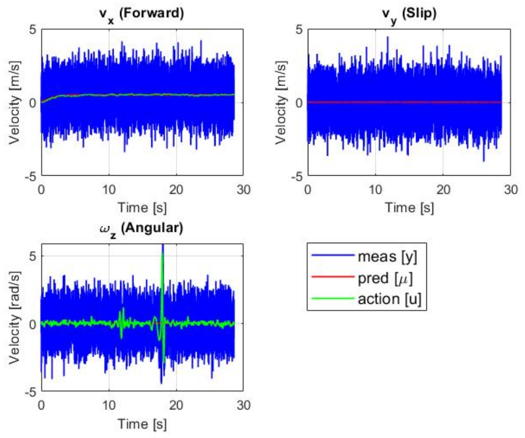

Fig. 24. Velocity plot of default setup with noise ($\sigma = 1$). Note these are the exact same tests as the figures above, and in section*Results*, with corresponding σ. Only the measurements are recorded with the added noise.

Appendix II
Graphs Regarding Stability

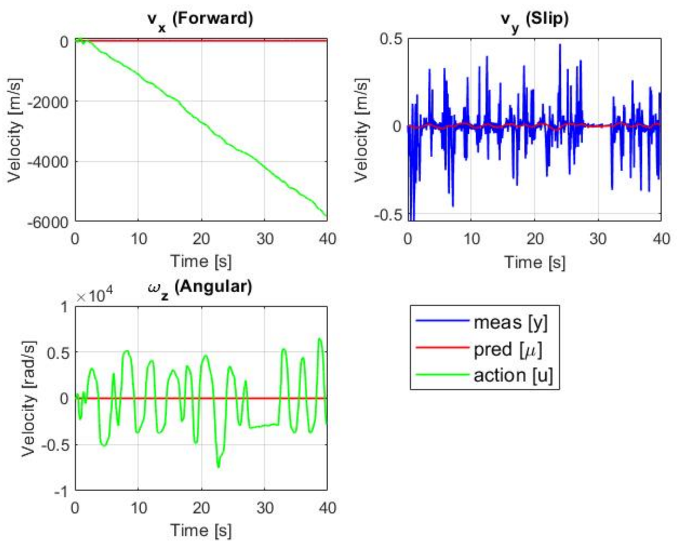

Fig. 25. Velocity plot of default setup with $\rho = 10000$, zoomed out. Note that the action line, top left plot, goes to minus infinity and the values of the angular action are very big, bottom left plot.

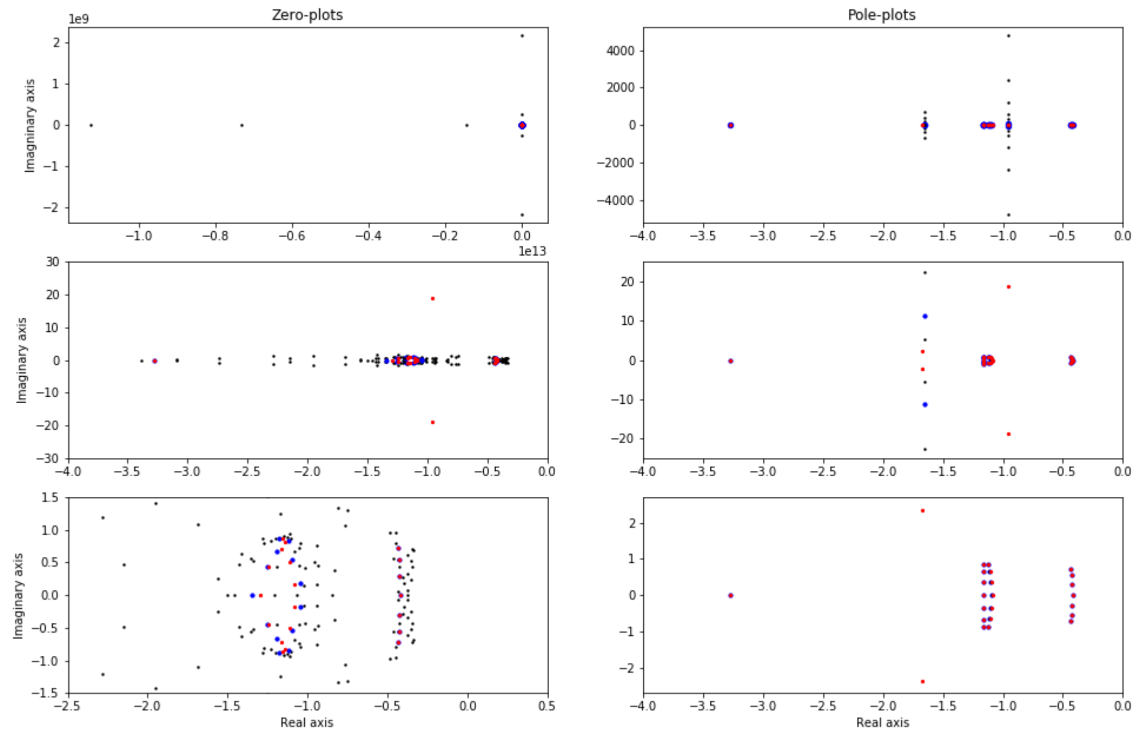

Fig. 26. Zoomed-in versions of zero plots can be viewed on the left-hand side, and pole plots on the right-hand side. It is plotted with default values with ρ varying between 50 and 5.000.000. The size of ρ is multiplied by 4 every plot, and are all plotted within the same figure. Red dots represent poles and zeros resulting from default values, blue dots with approximately the first unstable value of $\rho = 10.000$ and black every other zero and pole.

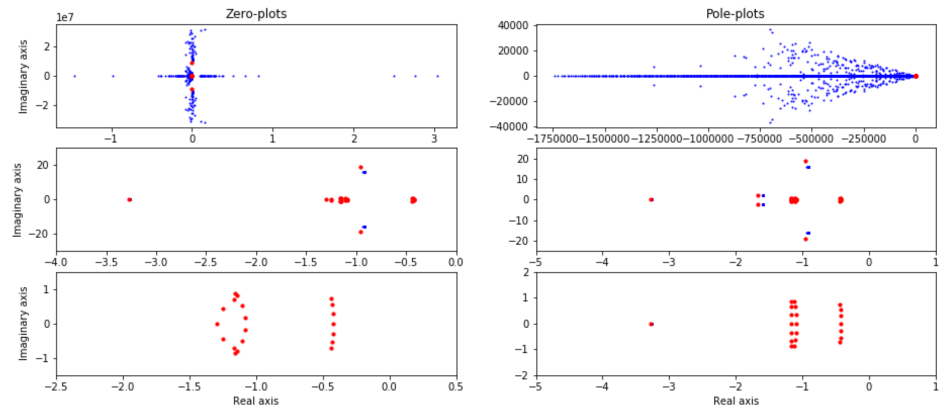

Fig. 27. Zoomed-in versions of zero plots can be viewed on the left-hand side, and pole plots on the right-hand side. It is plotted with default values with κ varying between 1 and 1.000.000. Step size is constant and equal to 50.000. Red dots represent default values. Note all poles remain in the left half plane.

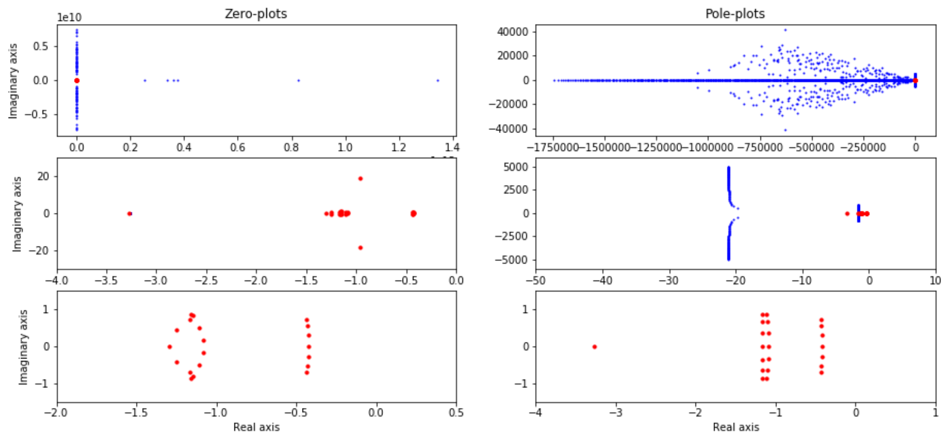

Fig. 28. Zoomed-in versions of zero plots can be viewed on the left-hand side, and pole plots on the right-hand side. It is plotted with default values with ρ and κ having varying values. Step size of both learning rates are constant. Red dots represent default values. Note all poles remain in the left half plane.

Appendix III
Graphs Regarding Prioritization

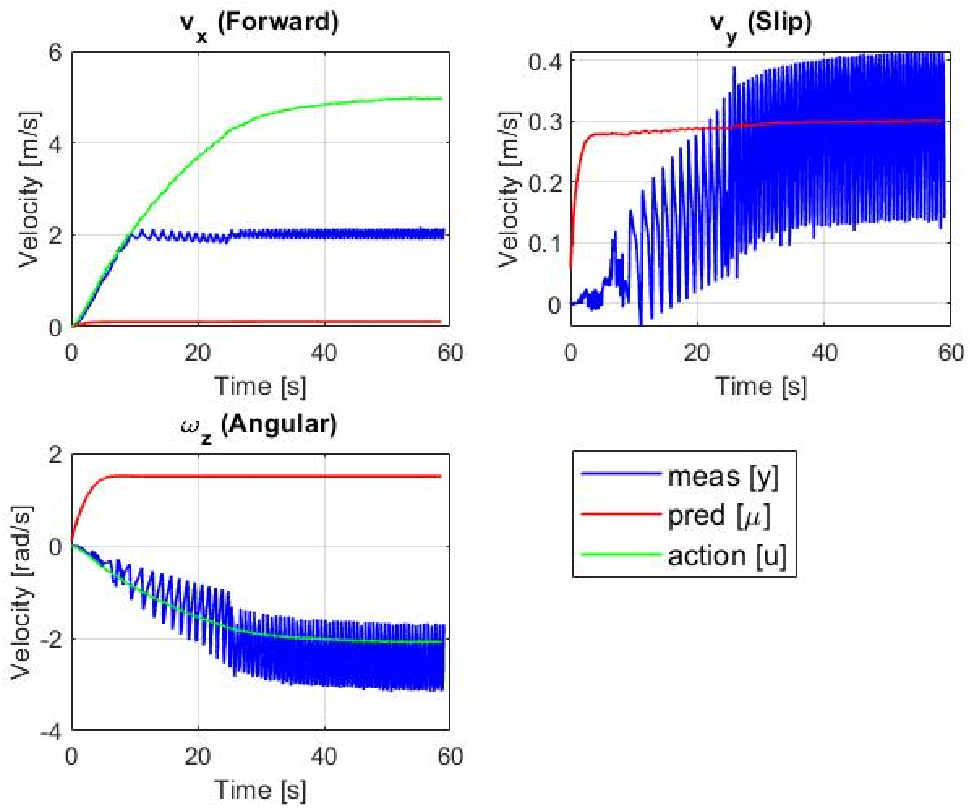

Fig. 29. Default setup where spots [1,2] and [1,0] in the A matrix are filled with 0.1 and 1.5 respectively, changing the impact of v_x and ω_z on v_y, resulting in improved slip behaviour.

Appendix IV
Python Codes

Main Active Inference code.

```
# -*- coding: utf-8 -*-
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from scipy import linalg
from setupBrain import setupBrain
from staticPrior import staticPrior
from setupCL import setupCL
from IFE import IFE
## State Space Active Inference: One Dimensional S stem
#
# The user can specify a desired state space model in the indicated section.
# Also the hyperparameters of the algorithm can be tuned in the indicated
# section. Don't make any other changes unless you want to adapt the inner
# workings of the algorithm.
#
# Several assumptions/simplifications have been adopted in this first
# version to make simulation more convenient or for other reasons. These are:
#
# 1) The prior variable 'xi' is constant (static reference)
# 2) The model parameters are known exactly, no learning

## State space model
m = np.matrix([17])           # mass in kg
d= 0.8*m                      # damping coefficient in x direction
length = np.matrix([0.420])    # length of the Jackal in m
a = 0.5*length                 # half length for multiplication
width = np.matrix([0.27])      # width of the Jackal in m
b = 0.5*width                  # half width for multiplication
Ic = 0.4485                    # Moment of Inertia around center of mass
R = np.matrix([0.098])           # wheel radius

x_eq = np.r_[0.5,0,0]

A = np.zeros((3,3))
A[0,0] = -4*d/m
A[1,1] = -4*d/m
A[2,2] = -(a**2+b**2)*d/Ic
A[1,2] = -x_eq[0]
A[1,0] = x_eq[2]

B = np.zeros((3,2))
B[0,:] = 2*d/m
B[2,0] = -2*d*b/Ic
B[2,1] = 2*d*b/Ic

C = np.identity(3)



## Tuning parameters
    # -------------------------------------------------------------------------
rho   = 50    # control learning rate (default: 2e8)
kappa = 1    # estimation learning rate (default: 1e2)
p     = 0                  # number of generalized states (default: 6)
varw  = np.matrix([10.0])       # uncertainty in states (variance)
varz  = np.matrix([10.0])       # uncertainty in outputs (variance)
s     = 0.1     # smoothness of (all) noises
    # -------------------------------------------------------------------------

## Preprocessing:
   # Time:
T_end = 10
dt    = 0.001
t     = np.arange(0, dt+T_end, dt)
N     = np.size(t,0)

    # Dimensions:
n = np.size(A,1)        # state eval
m = np.size(B,1)        # input
```

```
q = np.size(C,0)       # output

    # Put matrices in state space structure:
process = signal.StateSpace(A,B,C)

    # Setup AI variables:
brain = setupBrain(process,p,kappa,rho,varw,varz,s)

## Construct prior signal:
    # The prior is now assumed to be a desired (static) state.
brain.xi = staticPrior(brain.A,x_eq,p)

## Retrieve closed loop state space representation:
    # Note: the state of this system is [x mu u]
[Yss,MUss,AIss] = setupCL(process,brain)

## Simulate Active Inference
    # Construct input signals:
xi = brain.xi              # retrieve prior (constant in this version)

    # Simulations:
x0 = np.hstack([np.zeros(n),np.zeros(n*(p+1)+m)])  # Initial closed loop state [x mu u];

    # Constructing exponential matrixes
n_states = AIss.A.shape[0]
n_inputs = AIss.B.shape[1]
M = np.vstack([np.hstack([AIss.A * dt, AIss.B * dt]),
               np.zeros((n_inputs, n_states + n_inputs))])
expMT = linalg.expm(M.transpose())
Ad = expMT[:n_states, :n_states]
Bd = expMT[n_states:, :n_states]

n_states = MUss.A.shape[0]
n_inputs = MUss.B.shape[1]
M = np.vstack([np.hstack([MUss.A * dt, MUss.B * dt]),
               np.zeros((n_inputs, n_states + n_inputs))])

expMT = linalg.expm(M.transpose())
Abrain = expMT[:n_states, :n_states]
Bbrain = expMT[n_states:, :n_states]

n_states = Yss.A.shape[0]
n_inputs = Yss.B.shape[1]
M = np.vstack([np.hstack([Yss.A * dt, Yss.B * dt]),
               np.zeros((n_inputs, n_states + n_inputs))])
expMT = linalg.expm(M.transpose())
Aworld = expMT[:n_states, :n_states]
Bworld = expMT[n_states:, :n_states]

    # For-loop Simulation
yAI=np.zeros((N,AIss.A.shape[0]))
mu=np.zeros((N,MUss.A.shape[0]))
y=np.zeros((N,Yss.A.shape[0]))
yAI[0,:]=x0
y[0] = x0[0:n]

for i in range(0,N-1):
    yAI[i+1] = np.dot(yAI[i], Ad) + np.dot(xi, Bd)

    mu[i+1] = np.dot(mu[i], Abrain) + np.dot(np.hstack([np.matrix([y[i]]),np.matrix([xi])]), Bbrain)
    y[i+1] = np.dot(y[i],Aworld) + np.dot(mu[i,n*(p+1):mu.shape[1]],Bworld)


    # Extract states from the results:
xAI  = yAI[:,0:n].transpose()
muAI = yAI[:,n:n*(p+2)].transpose()
uAI  = yAI[:,n*(p+2):yAI.shape[1]].transpose()
     # Calculate IFE for whole trajectory:
F  = IFE(muAI,np.dot(C,xAI),brain)
FE = IFE(mu[:,0:n*(p+1)].transpose(),np.dot(C,y.transpose()),brain)
## Plot results:
plt.close("all")
plt.figure(1)
plt.subplot(221)
plt.grid(True)
plt.plot(t,x_eq[0]*np.ones(np.size(t)),'r:',label='$x_{eq}$')
plt.plot(t,xAI[0,:].transpose(),'--',label='$x$')
plt.plot(t,muAI[0,:],label='$\mu_x$')
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)

plt.xlabel('Time $[s]$')
plt.ylabel('State $[m/s]$')

plt.subplot(222)
plt.plot(t,uAI[0,:])
plt.grid(True)

plt.xlabel('Time $[s]$')
plt.ylabel('Action / Force $[N]$')

plt.subplot(223)

plt.plot(t,F[0]);
plt.grid(True)
plt.xlabel('Time $[s]$')
plt.ylabel('Free Energy')

plt.figure(2)
plt.subplot(221)

plt.grid(True)
plt.plot(t,x_eq[0]*np.ones(np.size(t)),'r:',label='$x_{eq}$')
plt.plot(t,y[:,0],'--',label='$y$')
plt.plot(t,mu[:,0],label='$\mu_x$')
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)

plt.xlabel('Time $[s]$')
plt.ylabel('State $[rad/s]$')

plt.subplot(222)
plt.plot(t,mu[:,n*(p+1):mu.shape[1]])
plt.grid(True)

plt.xlabel('Time $[s]$')
plt.ylabel('Action / Force $[N]$')

plt.subplot(223)

plt.plot(t,FE[0]);
plt.grid(True)
plt.xlabel('Time $[s]$')
plt.ylabel('Free Energy')

# %matplotlib qt
```

Forward model code.

```
# -*- coding: utf-8 -*-
def forwardModel(process):
    import numpy as np
## Construct the forward model for given state space process
#
# INPUTS:
# process   =   State space model (continuous or discrete)

## Unpack the process:
    A = process.A
    B = process.B
    C = process.C
    D = process.D

    n = np.size(A,1)

## Define forward model (adopting steady state gain assumption):
    if process.dt is None:
        Ghat= np.dot(np.dot(C,(-np.linalg.inv(A))),B)+D
    else:    # if discrete system
        Ghat= np.dot(np.dot(C,np.linalg.inv(np.identity(n)-A)),B) + D

    return Ghat
```

Generalized statespace code.

```
# -*- coding: utf-8 -*-
def generalizeStateSpace(process,p):
    import numpy as np
## Construct a generalized state space model
# INPUTS:
# process    =   State space model of generative process. Can be either
#                continuous or discrete time model.
# p          =   Desired number of generalized coordinates (>=0)
# genmeas    =   Optional boolean, indicating availability of generalized
#                measurements.
#
# OUTPUTS:
# Atilde     =   A matrix of generatized model
# Btilde     =   B matrix of generalized model
# Ctilde     =   C matrix of generalized model
# Dtilde     =   D matrix of generalized model

## Pre-processing:
    A = process.A
    B = process.B
    C = process.C
    D = process.D

    n = np.size(A,1) # state dimension
    m = np.size(B,1)
    q = np.size(C,0) # output dimension

## Generalize the model:
    # p = 0 -> no generalization desired.
    # p = 1 -> adds one derivative, etc. etc.

    Ip = np.identity((p+1))

    Atilde = np.kron(Ip,A)
    Btilde = np.kron(Ip,B)
    Ctilde = np.hstack([C,np.zeros((n,n*p))])
    Dtilde = np.hstack([D,np.zeros((n,n*p))])

    return [Atilde,Btilde,Ctilde,Dtilde]
```

Informational Free Energy code.

```
# -*- coding: utf-8 -*-

def IFE(muAI,xAI,brain):
    import numpy as np
## Evaluate current Informational Free Energy (IFE)
#
# INPUTS:
# mu         = Current generalized state estimation mu(k)
# y          = Current measurement, this is the new data
# brain      = Structure containing all variables of the robot brain
#
# OUTPUTS:
# F          = Current value of the free energy

## Extract required variables

    n = np.size(xAI,0)
    N = np.size(xAI,1) # number of timepoints
    #M = np.size(y,0)
    F=np.zeros((n,N))  # initialize vector

    for j in range(0,n):
        Atilde = brain.A[j::n,j::n]
        Ctilde = brain.C[0,0::n]
        D      = brain.D[0,0]
        Piz    = brain.Piz[0,0]
        Piw    = brain.Piw[j::n,j::n]
        xi     = brain.xi[j::n]
        mu     = muAI[j::n,:]
        y      = np.matrix([np.dot(Ctilde[0],xAI[j,:])])
## Calculate the Free Energy

        for i in range(0,N):
            F[j,i] =  0.5*(
                    (y[:,i]-np.dot(Ctilde,mu[:,i]))*Piz*(y[:,i]
                    - np.dot(Ctilde,mu[:,i])).transpose()
                    + np.dot((np.dot(D,mu[:,i])-np.dot(Atilde,mu[:,i])-xi),np.dot(Piw,(np.dot(D,mu[:,i])
                    - np.dot(Atilde,mu[:,i])-xi)).transpose())
                    )
    return F
```

Setup brain code.

```
# -*- coding: utf-8 -*-
def setupBrain(process,p1,kappa,rho,varw,varz,s):
    #from generalizeMeasurement import generalizeMeasurement
    from generalizeStateSpace import generalizeStateSpace
    from forwardModel import forwardModel

    from scipy.linalg import toeplitz
    import numpy as np
    from scipy import signal
##Construct a structure containing the generative model variables
# INPUTS:
# process    =   State space model of generative process (assuming agent
#                knows the exact process dynamics). Can be either continuous
#                or discrete time model.
# p          =   Desired number or generalized coordinates (>=0)
# kappa      =   Learning rate for perception cycle (>0)
# rho        =   Learning rate for action cycle (>0)
# varw       =   Variance of the state noises
# varz       =   Variance of the output noises
# s          =   Smoothness of the noises (same for all)

# OUTPUTS:
# brain.A       =   A matrix of generative model
# brian.B       =   B matrix of generative model
# brain.C       =   C matrix of generative model
# brain.G       =   Forward Model of agent
# brain.D       =   Derivative operator matrix
# brain.p       =   Number of generalized states
# brain.Piz     =   Precision matrix output noise
# brain.Piw     =   Precision matrix state noise
# brain.kappa   =   Perception learning rate
# brain.rho     =   Action learning rate
# brain.s       =   Smoothness of the noises


## Brain model options:
    A = process.A
    B = process.B
    C = process.C

## Pre-processing:
    kappa1 = kappa
    rho1 = rho
    s1 = s

    # Dimension variables:
    n = np.size(A,1); # state dimension

    # Identity matrices for quick construction:
# ==============================================================================
#     x=[]
#     for i in range(p1+1):
```

```
#          x.append(s*0*(1/(i+1)))
#          x.append(s*1/(i+1))
#          x.append(s*0/(i+1))
#      Ip = np.diag(x)
# ================================================================================
    Ip = np.identity(n*(p1+1))

    In = np.identity(n)

## Model Construction:
    # Set up generative model (assuming process is known):
    Ahat = A
    Bhat = B
    Chat = C
    model = signal.StateSpace(Ahat,Bhat,Chat)

    # Generalize the model:
    [Atilde,Btilde,Ctilde,Dtilde] = generalizeStateSpace(model,p1)

    # Construct forward model:
    Ghat = forwardModel(model)

    # Construct D-matrix (derivative operator!):
    if p1 is 0:
        D1 = np.zeros((n,n))
    else:
        T = toeplitz(np.zeros(p1+1),np.r_[0, 1, np.zeros(p1-1)])
        D1 = np.kron(T,In)


## Set up generalized precision (inverse covariance) matrices:
    Piz_tilde = np.kron(Ip,np.diag(1/varz))

    Piw_tilde = np.kron(Ip,np.diag(1/varw))

## Generate the output structure:
    class brain:
        A = Atilde
        B = Btilde
        C = Ctilde
        D = Dtilde
        G = Ghat
        Do = D1
        p = p1
        Piz = Piz_tilde
        Piw = Piw_tilde
        kappa = kappa1
        rho = rho1
        s = s1

    return brain
```

Setup closed loop code.

```
# -*- coding: utf-8 -*-
def setupCL(GenProc, brain):
    from scipy import signal
    import numpy as np
## FUNCTION DESCRIPTION:
# This function constructs the closed loop state space model
# INPUTS:
# GenProc    = Generative Process (State Space Model)
# brain      = Generative Model   (Structure):

# GenMod.A       =   A matrix of generative model
# GenMod.C       =   C matrix of generative model
# GenMod.G       =   Forward Model of agent
# GenMod.xi      =   Prior functional (static for now)
# GenMod.Piz     =   Precision matrix output noise
# GenMod.Piw     =   Precision matrix state noise
# GenMod.kappa   =   Perception learning rate
# GenMod.rho     =   Action learning rate

# OUTPUTS:
```

```
# AIss           =   State Space Model of the closed loop system

## Pre-processing
    # Extract process matrices:
    A = GenProc.A
    B = GenProc.B
    C = GenProc.C

# Unpack generative model:
    Atilde = brain.A
    Btilde = brain.B
    Ctilde = brain.C
    Dtilde = brain.D
    D      = brain.Do
    Ghat   = brain.G
    p      = brain.p
    rho    = brain.rho
    kappa  = brain.kappa
    Piz    = brain.Piz
    Piw    = brain.Piw

# Dimensions:
    n = np.shape(A)  # state
    m = np.shape(B)  # input
    q = np.shape(Ctilde)  # output

## Processing:
    M = D - kappa*np.dot(np.dot((D-Atilde).transpose(),Piw),(D-Atilde)) - kappa*np.dot(Ctilde.transpose(),Ctilde)
    l=np.shape(M)

    Q= -rho*np.dot(np.dot(Ghat.transpose(),Piz[0:n[0],0:n[1]]),C)
    R= np.dot(np.dot(rho*Ghat.transpose(),Piz[0:n[0]]),Ctilde.transpose())

# The state of the model is [x mu u], hence the 3x3 partitioning of Acl:
    Amu = np.zeros((l[0] + R.shape[0], l[1] + m[1]))
    x=0;y=0
    Amu[x:x+l[0],y:y+l[1]] = M
    x=x+l[0]
    Amu[x:x+R.shape[0],y:y+R.shape[1]] = R

    Acl = np.zeros(((n[0]+l[0]+Q.shape[0]),(n[1]+l[1]+m[1])))
    r=np.shape(Acl)
    x=0;y=0;
    Acl[x:x+n[0],y:y+n[1]]=A
    Acl[x:x+m[0],r[1]-m[1]:r[1]]=B
    x=x+n[0]
    Acl[x:x+q[1],y:y+q[0]]= np.dot(np.dot(kappa*Ctilde,Piz).transpose(),C)
    y=y+n[1]
    Acl[x:x+l[0],y:y+l[1]]=M
    x=x+q[1]; y=0
    Acl[x:r[0],y:y+n[1]]=Q
    y=y+Q.shape[1]
    Acl[x:r[0],y:y+n[1]]=R

    """
    Acl = [A                       zeros(n,n*(p+1))        B;...
           kappa*Ctilde.'*Piz*C  M                   zeros(n*(p+1),m);...
           -rho*Ghat*Piz*C         rho*Ghat*Piz*Ctilde zeros(m,m)];
    """

# The inputs are [xi w z], hence the 3x3 partitioning of Bcl:
    Bmu = np.zeros((l[0] + R.shape[0], l[1] + R.shape[1]))
    x=0;y=0
    Bmu[x:x+l[0],y:y+n[1]] = kappa*np.dot(Ctilde,Piz).transpose()
    x=x+l[0]
    Bmu[x:x+Q.shape[0],y:y+n[1]]= -rho*np.dot(Ghat.transpose(),Piz[0:n[0],0:n[1]])
    x=0;y=y+n[1]
    Bmu[x:x+l[0],y:y+l[1]] = kappa*np.dot((D-Atilde).transpose(),Piw)

    Bcl = np.zeros(((n[0]+l[0]+R.shape[0]),(l[1])))
    x=0+n[0];y=0
    Bcl[x:x+D.shape[0],y:y+D.shape[1]]= kappa*np.dot((D-Atilde).transpose(),Piw)

    """
    Bcl = [zeros(n,n*(p+1))            eye(n)        zeros(n,q);...
```

```
            kappa*(D-Atilde).'*Piw  zeros(n*(p+1),n)  kappa*Ctilde.'*Piz;...
            zeros(m,n*(p+1))            zeros(m,n)    -rho*Ghat*Piz];
        """

    # For inspection purposes, we choose to measure all signals:
        Cmu = np.zeros((np.size(B,1),np.size(Amu,1)))
        x=0;y=0+l[1]
        Cmu[x:x+B.shape[1],y:y+B.shape[1]] = np.identity(B.shape[1])
        #Cmu = np.eye(Amu.shape[0],Amu.shape[1])

        Ccl = np.eye(1,Acl.shape[1])

    # There is no direct feedthrough of any signal:
        Dmu = np.zeros((Cmu.shape[0],Bmu.shape[1]))

        Dcl = np.zeros((Ccl.shape[0],Bcl.shape[1]))

    # Put the four matrices in a state space sturcture (continuous):
        Yss  = signal.StateSpace(A,B,C)
        MUss = signal.StateSpace(Amu,Bmu,Cmu)
        AIss = signal.StateSpace(Acl,Bcl,Ccl)

        return (Yss,MUss,AIss)
```

Static prior code.

```
# -*- coding: utf-8 -*-
def staticPrior(Atilde,x_eq,p):
    import numpy as np
## Construct the static part of a prior
# INPUTS:
# Atilde     =   Belief of the agent about the generative process
# x_eq       =   Desired equilibrium state of generative process
# p          =   Number of generalized states
#
# OUTPUTS:
# xi         =   Reference value that will steer the estimation dynamics
#
## Pre-processing:
    n = np.size(x_eq,0)

## Construct the prior signal:
    x_gen_eq = np.hstack([x_eq,np.zeros(n*p)])    # Equilibrium in generalized coordinates

    Ades = 0                            # Desired process dynamics (not functional yet)

    xi = -np.dot((Atilde+Ades),x_gen_eq)      # static prior variable
    return xi
```

Pole-zero plot code for varying values of ρ

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 20 17:27:41 2018

@author: Stijn
"""
# -*- coding: utf-8 -*-
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from scipy import linalg
from control import *
import sys

from setupBrain import setupBrain
from staticPrior import staticPrior
from setupCL import setupCL
from IFE import IFE

rho = []
x = 50
```

```python
for i in range(100):
    rho.append(x)
    x *= 2
    if x == 51200:
        x = 50000
    if x >= 5e6:
        break

rho.reverse()
kappa = 1
desired_rho = 50000

for i in range(len(rho)):

    '''
    Put in code from main_simple.py and remove kappa and rho variables
    '''

    plt.figure(1)
    #plt.suptitle('Pole-Zero Plot for Varying Rho Values', fontsize=25)
    if i == len(rho)-1:
        plt.subplot(321)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')

        plt.subplot(322)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlim(-4, 0)


        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 5, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')
        plt.xlim(-4,0)
        plt.ylim(-30,30)

        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 5, 'r')
        plt.xlabel('Poles, real')
        plt.xlim(-4, 0)
        plt.ylim(-25,25)

        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 5, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')
        plt.xlim(-0.25e1,0.05e1)
        plt.ylim(-1.5,1.5)

        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 5, 'r')
        plt.xlabel('Poles, real')
        plt.xlim(-4, 0)
        plt.ylim(-2.7,2.7)

    elif i == 6:
        plt.subplot(321)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'b')

        plt.subplot(322)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'b')

        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'b')

        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'b')

        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'b')

        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'b')
    else:
        plt.subplot(321)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 2, 'y')
        plt.subplot(322)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 2, 'y')

        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 2, 'y')
        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 2, 'y')
        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 2, 'y')
        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 2, 'y')
```

Pole-zero plot code for varying values of κ

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 21 10:06:18 2018

@author: Stijn
"""

kappa = []
for i in range(0,100):
    x = 1 + 1e4*i
    kappa.append(x)

kappa.reverse()
rho = 50

for i in range(len(kappa)):

    '''
    Put in code from main_simple.py and remove kappa and rho variables
    '''


    fig = plt.figure(2)
    #plt.suptitle('Pole-Zero Plot for Varying Kappa Values', fontsize=25)


    if i == len(kappa)-1:

        zero = plt.subplot(321)
        zero.set_title("Zero-plots")
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')


        pole = plt.subplot(322)
        pole.set_title("Pole-plots")
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlabel('Poles, real')

        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')
        plt.xlim(-4,0)
        plt.ylim(-30,30)
        #plt.ylim(-30,30)

        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlabel('Poles, real')
        plt.xlim(-5, 1)
        plt.ylim(-25,25)
        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')
        plt.xlim(-0.25e1,0.05e1)
        plt.ylim(-1.5,1.5)

        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlabel('Poles, real')
        plt.xlim(-5,1)
        plt.ylim(-2,2)


    else:
        plt.subplot(321)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 1, 'b')
        plt.subplot(322)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 1, 'b')

        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 1, 'b')
        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 1, 'b')
        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 1, 'b')
        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 1, 'b')
```

Pole-zero plot code for varying values of ρ and κ

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Dec 20 17:30:10 2018

@author: Stijn
"""

kappa = []
rho = []

for i in range(0,100):
    x = 1 + 1e4*i
    kappa.append(x)

for i in range(0,100):
    x = 50 + 50000*i
    rho.append(x)

rho.reverse()
kappa.reverse()

for i in range(len(kappa)):

    '''
    Put in code from main_simple.py and remove kappa and rho variables
    '''

    if i == len(kappa)-1:

        zero = plt.subplot(321)
        zero.set_title("Zero-plots")
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')

        pole = plt.subplot(322)
        pole.set_title("Pole-plots")
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlabel('Poles, real')


        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')
        plt.xlim(-4,0)
        plt.ylim(-30,30)
        #plt.ylim(-30,30)

        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlabel('Poles, real')
        plt.xlim(-50, 10)
        plt.ylim(-6000,6000)

        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 10, 'r')
        plt.xlabel('Zeros, real')
        plt.ylabel('imag')
        plt.xlim(-2,0.05e1)
        plt.ylim(-1.5,1.5)

        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 10, 'r')
        plt.xlabel('Poles, real')
        plt.xlim(-4,1)
        plt.ylim(-1.5,1.5)


    else:
        plt.subplot(321)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 1, 'b')
        plt.subplot(322)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 1, 'b')

        plt.subplot(323)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 1, 'b')
        plt.subplot(324)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 1, 'b')
        plt.subplot(325)
        plt.scatter(AIzpg.zeros.real,AIzpg.zeros.imag, 1, 'b')
        plt.subplot(326)
        plt.scatter(AIzpg.poles.real,AIzpg.poles.imag, 1, 'b')
```

Appendix V
Matlab Codes

Code to generate plots from rosbag files

```
%script for plotting results from rosbag files
clear all
close all

%bag names
prediction_bag = "pr10000.bag";
measurement_bag= "mr10000.bag";
action_bag     = "sr10000.bag";

%Open the rosbag log files and get content information
bagpred = rosbag(prediction_bag);
bagmeas = rosbag(measurement_bag);
bagact  = rosbag(action_bag);

%data measurements
datameas = readMessages(bagmeas,"DataFormat",'struct');

%data prediction
datapred = readMessages(bagpred,"DataFormat",'struct');

%data prediction
dataact = readMessages(bagact,"DataFormat",'struct');


%velocties measurements
xmeas = cellfun(@(m) double(m.XVelocity), datameas);
ymeas = cellfun(@(m) double(m.YVelocity), datameas);
angmeas = cellfun(@(m) double(m.AngularVelocity), datameas);

%velocities prediction
MuX = cellfun(@(m) double(m.MuX), datapred);
MuY = cellfun(@(m) double(m.MuY), datapred);
MuW = cellfun(@(m) double(m.MuW), datapred);

%velocities action
xact = cellfun(@(m) double(m.Linear.X), dataact);
angact = cellfun(@(m) double(m.Angular.Z), dataact);

%Time vectors
Timemeas = bagmeas.MessageList.Time;
Timemeas = Timemeas-min(Timemeas);

Timepred = bagpred.MessageList.Time;
Timepred = Timepred-min(Timepred);

Timeact = bagact.MessageList.Time;
Timeact = Timeact-min(Timeact);

%plots
subplot(2,2,1)
plot(Timemeas,xmeas,"b","LineWidth",1)
hold on
plot(Timepred,MuX,"r",'LineWidth',1)
hold on
plot(Timeact,xact,"g",'LineWidth',1)
title("v_x (Forward)")
xlabel("Time [s]")
ylabel("Velocity [m/s]")
grid on


subplot(2,2,2)
plot(Timemeas,ymeas,"b",'LineWidth',1)
hold on
plot(Timepred,MuY,"r",'LineWidth',1)
title("v_y (Slip)")
xlabel("Time [s]")
ylabel("Velocity [m/s]")
grid on
```

```
subplot(2,2,3)
plot(Timemeas,angmeas,"b",'LineWidth',1)
hold on
plot(Timepred,MuW,"r",'LineWidth',1)
hold on
plot(Timeact,angact,"g",'LineWidth',1)
title("\omega_z (Angular)")
xlabel("Time [s]")
ylabel("Velocity [rad/s]")
grid on
legend({'meas [y]','pred [\mu]','action [u]'},'Location','southeast','FontSize',11)
```

Appendix VI
ROS Node Codes

Brain node

```
#! /usr/bin/env python
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from scipy import linalg
from scipy.linalg import toeplitz
from jackal_msgs.msg import JackalCoords
from jackal_msgs.msg import Measurements
from jackal_msgs.msg import Brain
from setupBrain import setupBrain
from staticPrior import staticPrior
from setupCL import setupCL
from IFE import IFE
import rospy

def AICallback (JackalCoords, pub):

    x_vel = JackalCoords.x_velocity
    y_vel = JackalCoords.y_velocity
    ang_vel = JackalCoords.angular_velocity

    y = np.hstack([x_vel, y_vel, ang_vel])

    global mu_old
    mu_new = np.dot(mu_old, Abrain) + np.dot(np.hstack([np.matrix([y]),np.matrix([xi])]), Bbrain)

    #print "new", mu_new

    Steering_Signal = Brain()
    Steering_Signal.left = mu_old[0,-2]
    Steering_Signal.right = mu_old[0,-1]
    Steering_Signal.angular = (Steering_Signal.right*0.098-Steering_Signal.left*0.098)/0.27
    Steering_Signal.forward = (Steering_Signal.left+Steering_Signal.right)/2

    Steering_Signal.mu_x = mu_new[0,0]
    Steering_Signal.mu_y = mu_new[0,1]
    Steering_Signal.mu_w = mu_new[0,2]

    mu_old = mu_new

    pub.publish(Steering_Signal)


    #OPEN SPACE FOR FREE ENERGY


if __name__ == '__main__':
    rospy.init_node('AI_controller')

    #AI MAIN

    m = np.matrix([17])             # mass in kg
    d= 0.8*m                        # damping coefficient in x direction
    length = np.matrix([0.420])     # length of the Jackal in m
    a = 0.5*length                  # half length for multiplication
    width = np.matrix([0.27])      # width of the Jackal in m
    b = 0.5*width                   # half width for multiplication
    Ic = 0.4485                     # Moment of Inertia around center of mass
    R = np.matrix([0.098])          # wheel radius

    # Desired equilibrium (x = 40m/s)
    x_eq1 = np.r_[0.5,0,0]
    x_eq = np.ma.masked_array(x_eq1, x_eq1 == 1)

    A = np.zeros((3,3))
    A[0,0] = -4*d/m
    A[1,1] = -4*d/m
    A[2,2] = -(a**2+b**2)*d/Ic
    A[1,2] = -x_eq1[0]
    A[1,0] = x_eq1[2]
```

```
#A[0,1] = -0.5
#A[1,0] = 0.5

B = np.zeros((3,2))
B[0,:] = 2*d/m  #forward
B[2,0] = -2*d*b/Ic
B[2,1] = 2*d*b/Ic

C = np.identity(3)        # measure the state directly


## Tuning parameters
# CHOOSE YOUR OWN HYPERPARAMETERS HERE
# -----------------------------------------------------------------------
rho   = 50   # control learning rate (default: 2e8)
kappa = 1   # estimation learning rate (default: 1e2)
p     = 6               # number of generalized states (default: 6)
varw  = np.matrix([10.0])      # uncertainty in states (variance)
varz  = np.matrix([10.0])      # uncertainty in outputs (variance)
s     = np.matrix([0.1])     # smoothness of (all) noises
# -----------------------------------------------------------------------

## Preprocessing:
# Time:
T_end = 20
dt    = 0.001
t     = np.arange(0, dt+T_end, dt)
N     = np.size(t,0)

# Dimensions:
n = np.size(A,1)       # state eval
m = np.size(B,1)       # input
q = np.size(C,0)       # output

# Put matrices in state space structure:
process = signal.StateSpace(A,B,C)

# Setup AI variables:
brain = setupBrain(process,p,kappa,rho,varw,varz,s)

## Construct prior signal:
# The prior is now assumed to be a desired (static) state.
brain.xi = staticPrior(brain.A,x_eq,p)

## Retrieve closed loop state space representation:
# Note: the state of this system is [x mu u]
[Yss,MUss,AIss] = setupCL(process,brain)

## Simulate Active Inference
# Construct input signals:
xi = brain.xi               # retrieve prior (constant in this version)
#u = np.repeat(xi, N, 0)      # prior signal

# Simulations:
#x0 = np.hstack([np.zeros(n),np.zeros(n*(p+1)+m)])  # Initial closed loop state [x mu u];

# Constructing exponential matrixes
n_states = MUss.A.shape[0]
n_inputs = MUss.B.shape[1]
M = np.vstack([np.hstack([MUss.A * dt, MUss.B * dt]), np.zeros((n_inputs, n_states + n_inputs))])

expMT = linalg.expm(M.transpose())
Abrain = expMT[:n_states, :n_states]
Bbrain = expMT[n_states:, :n_states]

# For-loop Simulation
mu_old = np.zeros((1,MUss.A.shape[0]))
#y = np.zeros(Yss.A.shape[0])


pub = rospy.Publisher('Steering_Signal', Brain, queue_size = 1)
rospy.Subscriber('Measurements_Jackal', Measurements, AICallback, pub)
rospy.spin()
```

Brain node with noise

```
#! /usr/bin/env python
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from scipy import linalg
from scipy.linalg import toeplitz
from jackal_msgs.msg import JackalCoords
from jackal_msgs.msg import Measurements
from jackal_msgs.msg import Brain
from setupBrain import setupBrain
from staticPrior import staticPrior
from setupCL import setupCL
from IFE import IFE
import rospy

def AICallback (JackalCoords, pub):

    x_vel = JackalCoords.x_velocity
    y_vel = JackalCoords.y_velocity
    ang_vel = JackalCoords.angular_velocity

    y = np.hstack([x_vel, y_vel, ang_vel])

    global mu_old
    mu_new = np.dot(mu_old, Abrain) + np.dot(np.hstack([np.matrix([y]),np.matrix([xi])]), Bbrain)

    #print "new", mu_new

    Steering_Signal = Brain()
    Steering_Signal.left = mu_old[0,-2]
    Steering_Signal.right = mu_old[0,-1]
    Steering_Signal.angular = (Steering_Signal.right*0.098-Steering_Signal.left*0.098)/0.27
    Steering_Signal.forward = (Steering_Signal.left+Steering_Signal.right)/2

    Steering_Signal.mu_x = mu_new[0,0]
    Steering_Signal.mu_y = mu_new[0,1]
    Steering_Signal.mu_w = mu_new[0,2]

    mu_old = mu_new

    pub.publish(Steering_Signal)


    #OPEN SPACE FOR FREE ENERGY


if __name__ == '__main__':
    rospy.init_node('AI_controller')

    #AI MAIN

    m = np.matrix([17])            # mass in kg
    d= 0.8*m                       # damping coefficient in x direction
    length = np.matrix([0.420])    # length of the Jackal in m
    a = 0.5*length                 # half length for multiplication
    width = np.matrix([0.27])     # width of the Jackal in m
    b = 0.5*width                  # half width for multiplication
    Ic = 0.4485                    # Moment of Inertia around center of mass
    R = np.matrix([0.098])         # wheel radius

    # Desired equilibrium (x = 40m/s)
    x_eq1 = np.r_[0.5,0,0]
    x_eq = np.ma.masked_array(x_eq1, x_eq1 == 1)

    A = np.zeros((3,3))
    A[0,0] = -4*d/m
    A[1,1] = -4*d/m
    A[2,2] = -(a**2+b**2)*d/Ic
    A[1,2] = -x_eq1[0]
    A[1,0] = x_eq1[2]
```

```
#A[0,1] = -0.5
#A[1,0] = 0.5

B = np.zeros((3,2))
B[0,:] = 2*d/m  #forward
B[2,0] = -2*d*b/Ic
B[2,1] = 2*d*b/Ic

C = np.identity(3)        # measure the state directly


## Tuning parameters
# CHOOSE YOUR OWN HYPERPARAMETERS HERE
# -------------------------------------------------------------------------
rho   = 50   # control learning rate (default: 2e8)
kappa = 1     # estimation learning rate (default: 1e2)
p     = 6                # number of generalized states (default: 6)
varw  = np.matrix([10.0])      # uncertainty in states (variance)
varz  = np.matrix([10.0])      # uncertainty in outputs (variance)
s     = np.matrix([0.1])     # smoothness of (all) noises
# -------------------------------------------------------------------------

## Preprocessing:
# Time:
T_end = 20
dt    = 0.001
t     = np.arange(0, dt+T_end, dt)
N     = np.size(t,0)

# Dimensions:
n = np.size(A,1)       # state eval
m = np.size(B,1)       # input
q = np.size(C,0)       # output

# Put matrices in state space structure:
process = signal.StateSpace(A,B,C)

# Setup AI variables:
brain = setupBrain(process,p,kappa,rho,varw,varz,s)

## Construct prior signal:
# The prior is now assumed to be a desired (static) state.
brain.xi = staticPrior(brain.A,x_eq,p)

## Retrieve closed loop state space representation:
# Note: the state of this system is [x mu u]
[Yss,MUss,AIss] = setupCL(process,brain)

## Simulate Active Inference
# Construct input signals:
xi = brain.xi                # retrieve prior (constant in this version)
#u = np.repeat(xi, N, 0)       # prior signal

# Simulations:
#x0 = np.hstack([np.zeros(n),np.zeros(n*(p+1)+m)])  # Initial closed loop state [x mu u];

# Constructing exponential matrixes
n_states = MUss.A.shape[0]
n_inputs = MUss.B.shape[1]
M = np.vstack([np.hstack([MUss.A * dt, MUss.B * dt]), np.zeros((n_inputs, n_states + n_inputs))])

expMT = linalg.expm(M.transpose())
Abrain = expMT[:n_states, :n_states]
Bbrain = expMT[n_states:, :n_states]

# For-loop Simulation
mu_old = np.zeros((1,MUss.A.shape[0]))
#y = np.zeros(Yss.A.shape[0])


pub = rospy.Publisher('Steering_Signal', Brain, queue_size = 1)
rospy.Subscriber('Measurements_Noise', Measurements, AICallback, pub)
rospy.spin()
```

Frame converter node

```
#! /usr/bin/env python
import rospy
import math
from gazebo_msgs.msg import ModelStates
from tf.transformations import euler_from_quaternion
from jackal_msgs.msg import JackalCoords

def callback(ModelStates, pub):
    #Quaternion coordinates in world frame
    x_ori = ModelStates.pose[-1].orientation.x
    y_ori = ModelStates.pose[-1].orientation.y
    z_ori = ModelStates.pose[-1].orientation.z
    w_ori = ModelStates.pose[-1].orientation.w
    quaternion = (x_ori, y_ori, z_ori, w_ori)

    #Euler coordinates from quaternion coordinates in world frame
    (roll, pitch, yaw) = euler_from_quaternion(quaternion)
    theta = yaw #yaw in radians

    ang_vel = ModelStates.twist[-1].angular.z
    x_vel_world = ModelStates.twist[-1].linear.x
    y_vel_world = ModelStates.twist[-1].linear.y
    abs_vel = math.sqrt(x_vel_world**2 + y_vel_world**2)

    #Euler coordinates in jackal frame
    x_vel_jackal = y_vel_world * math.cos(math.pi * 0.5 - theta) + x_vel_world * math.cos(theta)
    y_vel_jackal = y_vel_world * math.cos(theta) - x_vel_world * math.cos(math.pi*0.5 - theta)

    print "-----------------------------"
    print "X velocity is:", x_vel_jackal
    print "Y velocity is:", y_vel_jackal
    print "Angular velocity is:", ang_vel
    print "-----------------------------"
    print ""

    coordinates = JackalCoords()
    coordinates.x_velocity = x_vel_jackal
    coordinates.y_velocity = y_vel_jackal
    coordinates.angular_velocity = ang_vel

    pub.publish(coordinates)


if __name__ == '__main__':
    rospy.init_node('Frame_Converter')
    pub = rospy.Publisher('Measurements_Jackal', JackalCoords, queue_size = 1)
    rospy.Subscriber('/gazebo/model_states', ModelStates, callback, pub)
    rospy.loginfo("The real-time velocities from the Jackal are:")
    rospy.spin()
```

Frame converter node with noise

```
#! /usr/bin/env python
import rospy
import math
from gazebo_msgs.msg import ModelStates
from tf.transformations import euler_from_quaternion
from jackal_msgs.msg import JackalCoords

def callback(ModelStates, pub):
    #Quaternion coordinates in world frame
    x_ori = ModelStates.pose[-1].orientation.x
    y_ori = ModelStates.pose[-1].orientation.y
    z_ori = ModelStates.pose[-1].orientation.z
    w_ori = ModelStates.pose[-1].orientation.w
    quaternion = (x_ori, y_ori, z_ori, w_ori)

    #Euler coordinates from quaternion coordinates in world frame
    (roll, pitch, yaw) = euler_from_quaternion(quaternion)
    theta = yaw #yaw in radians

    ang_vel = ModelStates.twist[-1].angular.z
    x_vel_world = ModelStates.twist[-1].linear.x
    y_vel_world = ModelStates.twist[-1].linear.y
    abs_vel = math.sqrt(x_vel_world**2 + y_vel_world**2)

    #Euler coordinates in jackal frame
    x_vel_jackal = y_vel_world * math.cos(math.pi * 0.5 - theta) + x_vel_world * math.cos(theta)
    y_vel_jackal = y_vel_world * math.cos(theta) - x_vel_world * math.cos(math.pi*0.5 - theta)

    print "-----------------------------"
    print "X velocity is:", x_vel_jackal
    print "Y velocity is:", y_vel_jackal
    print "Angular velocity is:", ang_vel
    print "-----------------------------"
    print ""

    coordinates = JackalCoords()
    coordinates.x_velocity = x_vel_jackal
    coordinates.y_velocity = y_vel_jackal
    coordinates.angular_velocity = ang_vel

    pub.publish(coordinates)


if __name__ == '__main__':
    rospy.init_node('Frame_Converter')
    pub = rospy.Publisher('Measurements_Jackal', JackalCoords, queue_size = 1)
    rospy.Subscriber('/gazebo/model_states', ModelStates, callback, pub)
    rospy.loginfo("The real-time velocities from the Jackal are:")
    rospy.spin()
```

Noise generator node

```
#! /usr/bin/env python
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from scipy import linalg
from scipy.linalg import toeplitz
from jackal_msgs.msg import JackalCoords
from jackal_msgs.msg import Measurements
from jackal_msgs.msg import Brain
from setupBrain import setupBrain
from staticPrior import staticPrior
from setupCL import setupCL
from IFE import IFE
import rospy

def NoiseCallback(JackalCoords, pub):

    std = 1
    num_samples = 1

    mean_X = JackalCoords.x_velocity
    mean_Y = JackalCoords.y_velocity
    mean_Angular = JackalCoords.angular_velocity

    Noise_X = np.random.normal(mean_X, std, size=num_samples)
    Noise_Y = np.random.normal(mean_Y, std, size=num_samples)
    Noise_Angular = np.random.normal(mean_Angular, std, size=num_samples)

    Measurements_Noise = Measurements()
    Measurements_Noise.x_velocity = Noise_X
    Measurements_Noise.y_velocity = Noise_Y
    Measurements_Noise.angular_velocity = Noise_Angular

    pub.publish(Measurements_Noise)

if __name__ == '__main__':
    rospy.init_node('Noise_Generator')
    pub=rospy.Publisher('Measurements_Noise', Measurements, queue_size = 1)
    rospy.Subscriber('Measurements_Jackal',JackalCoords, NoiseCallback, pub)
    rospy.spin()
```

Steering signal node

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from rosgraph_msgs.msg import Clock
from jackal_msgs.msg import Brain

def callback(Brain, pub):
    seconds = rospy.get_time()
    print Brain.forward
    move.linear.x = Brain.forward
    move.angular.z = Brain.angular
    '''
    if int(seconds) % 6 == 0:
        move.linear.x = 0.5
        move.angular.z = 0.5
    elif int(seconds+2) % 6 == 0:
        move.linear.x = 0
        move.angular.z = 0.5
    elif int(seconds+4) % 6 == 0:
        move.linear.x = 0
        move.angular.z = 0.5
    '''
    pub.publish(move)

if __name__ == '__main__':
    rospy.init_node('Steering_Signal_Converter')
    pub = rospy.Publisher('/jackal_velocity_controller/cmd_vel', Twist, queue_size=1)
    rospy.Subscriber('/Steering_Signal', Brain, callback, pub)
    move = Twist()
    rospy.spin()
```

Steering signal node with noise

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from rosgraph_msgs.msg import Clock
from jackal_msgs.msg import Brain

def callback(Brain, pub):
    seconds = rospy.get_time()
    print Brain.forward
    move.linear.x = Brain.forward
    move.angular.z = Brain.angular
    '''
    if int(seconds) % 6 == 0:
        move.linear.x = 0.5
        move.angular.z = 0.5
    elif int(seconds+2) % 6 == 0:
        move.linear.x = 0
        move.angular.z = 0.5
    elif int(seconds+4) % 6 == 0:
        move.linear.x = 0
        move.angular.z = 0.5
    '''
    pub.publish(move)

if __name__ == '__main__':
    rospy.init_node('Steering_Signal_Converter')
    pub = rospy.Publisher('/jackal_velocity_controller/cmd_vel', Twist, queue_size=1)
    rospy.Subscriber('/Steering_Signal', Brain, callback, pub)
    move = Twist()
    rospy.spin()
```

APPENDIX VII
ROS LAUNCH CODES

Launch file of all nodes without noise

```
<launch>

    <node pkg ="bep" type="Steering_Signal_Converter.py" name="Steering_Signal_Converter">
    </node>

    <node pkg ="bep" type="Frame_Converter.py" name="Frame_Converter">
    </node>

    <node pkg ="bep" type="Brain.py" name="Brain" output="screen">
    </node>

    <node pkg ="rosbag" type="record" name="meas_record" args="record -O /home/sebastiaan/meas --duration=60
/Measurements_Jackal">
    </node>

    <node pkg="rosbag" type="record" name="pred_record" args="record -O /home/sebastiaan/pred --duration=60
/Steering_Signal">
    </node>

    <node pkg="rosbag" type="record" name="steer_record" args="record -O /home/sebastiaan/steer --duration=60
/jackal_velocity_controller/cmd_vel">
    </node>
</launch>
```

Launch file of all nodes with noise

```
<launch>

    <node pkg ="bep" type="Steering_Signal_Converter_N.py" name="Steering_Signal_Converter">
    </node>

    <node pkg ="bep" type="Frame_Converter_N.py" name="Frame_Converter">
    </node>

    <node pkg ="bep" type="Brain_N.py" name="Brain" output="screen">
    </node>

    <node pkg ="bep" type="Noise_Generator_N.py" name="Noise_Generator" output="screen">
    </node>

    <node pkg ="rosbag" type="record" name="meas_record" args="record -O /home/sebastiaan/measN --duration=30
/Measurements_Jackal">
    </node>

    <node pkg="rosbag" type="record" name="pred_record" args="record -O /home/sebastiaan/predN --duration=30
/Steering_Signal">
    </node>

    <node pkg="rosbag" type="record" name="steer_record" args="record -O /home/sebastiaan/steerN --duration=30
/jackal_velocity_controller/cmd_vel">
    </node>
</launch>
```