Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering
Signal Processing Systems Group

A Real-World Implementation of Active Inference

Master Thesis

Burak Ergül
b.ergul@student.tue.nl

Supervisor:
Prof.dr.ir. Bert de Vries

Eindhoven, April 2020

Abstract

Moving towards a world with ubiquitous automation, efficient design of intelligent autonomous agents that are capable of adapting to dynamic environments gains traction. In this setting, active inference emerges as a contender that inherently brings together action and perception through the minimization of a single cost function, namely free energy. Being a Bayesian inference method, active inference not only encodes uncertainty for perception but also for action and hence, it provides a strong expediency for building real-world intelligent autonomous agents that have to deal with uncertainties naturally found in the world. Coupled with other methods such as automated generation of inference algorithms and Forney-style factor graphs, active inference offers fast design cycles, adaptability and modularity. Furthermore, in cases where a priori specification of goal priors is prohibitively difficult, Bayesian target modelling opens up active inference to more complex problems and provides a higher-level means of speeding up design cycles through learning desired future observations. In order to assess active inference's capabilities and feasibility for a real-world application, we provide a proof of concept that runs on a ground-based robot in order to navigate in a specified area to find the location chosen by the user through observing user feedback.

Contents

List of Figures

1 INTRODUCTION

Since the industrial revolution, there has been an accelerating tendency towards the automation of our production methods and in this setting, the concept of intelligent autonomous agents gains traction. However, building intelligent autonomous agents involves some challenges that still need to be overcome. These challenges include the development of methods that enable agents to be capable of achieving specific goals in dynamic industrial settings, an approach that enables fast and efficient design cycles and in some cases, the integration of users into the design cycle to tailor for their individual needs (such as in the wearable electronics industry).

This paper proposes a combination of several approaches that contribute to overcoming these challenges in various complementary ways. More specifically, an intelligent autonomous agent needs to be capable of executing three tasks: perception, learning and decision making. The execution of these three tasks falls in the domain of Active Inference (section 2.4) where they are inherently brought together under a generative model. Furthermore, active inference allows the use of a single cost function, namely free energy minimization (section 2.3), which is principally based upon model evidence and being a problem-based approach, it enables ad hoc solutions. Active inference can be executed using Forney-style Factor Graphs (section 2.2) which contribute to the adaptability of models and allow modular/reusable updates through node-local computations. The message update rules needed to do inference on factor graphs can be derived automatically which speeds up the design cycle drastically (section 2.5).

1.1 Research Questions

Active inference and other supporting methods that were mentioned above have been developed to enable engineers to build adaptable agents that can achieve specific goals in dynamic settings but so far these methods have not been implemented for a real-world application. Hence, this paper will try to answer the following questions:

Is Active Inference a feasible option for the efficient design of adaptive controllers/algorithms for real-world applications?

How can Active Inference control be implemented for the efficient design of a real-world robot?

How can users be integrated into the Active Inference controller design cycle to tailor for their specific needs?

To answer these questions, a robot-car is implemented that uses active inference in order to navigate in a previously specified area to find and park in the location chosen by the user. In the beginning, the prior belief of the agent is initialized to an uninformative distribution as to which location the user might choose. This belief is subsequently updated and refined with each iteration through moving to a new location and requesting feedback from the user asking whether the new location is better or worse compared to the previous one. The main contributions of this paper are:

- A proof of concept implementation of active inference for a real-world agent.
- A 2D physical model responsible for navigating the agent by inferring motor velocities.
- A 2D target model responsible for inferring the location chosen by the user by observing continuous user feedback.

2 METHODS OF ACTIVE INFERENCE AND FREE ENERGY MINIMIZATION

2.1 Model-Based Machine Learning

Machine learning has come a long way since its inception with a multitude of different algorithms to address a broad range of problems. However, when a researcher decides to tackle a new problem, he often needs to fit his problem to one of the existing methods or implement a new algorithm. Considering how complex these algorithms can be, Model-based Machine Learning (MBML) provides a new perspective that enables the creation of ad hoc solutions for each specific problem [6]. The MBML approach has several advantages compared to classical approaches which include:

- The ability to specify a wide variety of models, where some well-known traditional machine learning techniques appear as special cases.
- The ease of creating ad hoc solutions to specific problems.
- The segregation of the model (which is application specific) and inference method (which is generic), which enables fast refinements to the model without the need for adapting the inference methods accordingly. Moreover, changes to the inference method also do not require changes to the model.

MBML is implemented through the use of a compact model specification language and the accompanying inference rules to automate the algorithm generation process. Models consist of random variables and their relation is incorporated through specifying their joint probability distribution in the form $p(x_1, ..., x_K)$ where $\{x_1, ..., x_K\}$ includes model parameters as well as observed and latent variables. Bayes theorem rests upon two simple rules:

$$\textbf{Product Rule:} \qquad p(x_a, x_b) = p(x_a)p(x_b|x_a) \tag{1}$$

$$\textbf{Sum Rule:} \qquad p(x_b) = \sum_{x_a} p(x_a, x_b) \tag{2}$$

From which Bayes rule can be derived:

$$p(x_a|x_b) = \frac{p(x_a)p(x_b|x_a)}{p(x_b)} \tag{3}$$

where $p(x_a)$ is known as the *prior*, $p(x_b|x_a)$ as the *likelihood function*, $p(x_a|x_b)$ as the *posterior* and $p(x_b)$ as the *evidence*. In this setting, each new observation of the random variable x_b allows the posterior distribution to be calculated from the prior distribution and the likelihood function. In the next iteration, the previously calculated posterior distribution becomes the prior distribution, which allows for an easy implementation of online learning.

The process of evaluating the posterior distribution $p(x_a|x_b)$ of the latent variable x_a given the observed variable x_b is known as inference. There are two types of inference techniques: exact and approximate inference. Exact inference algorithms calculate the exact posterior distribution of $p(x_a|x_b)$. Some examples of exact inference algorithms include the sum-product algorithm [27], the elimination algorithm [24] and the junction tree algorithm [28]. Next we will focus on the sum-product algorithm and give an example using Forney-style Factor Graphs (FFG).

An FFG is a type of probabilistic graphical model (PGM). PGMs offer a framework that allows the exploitation of structure in complex probability distributions and to describe

them compactly [24]. The specification of a model, in a Bayesian context, corresponds to the specification of a joint probability distribution (also known as a *generative model*) over all the random variables including the parameters in the model as well as latent and observed variables. A PGM is a visual expression of how the joint probability distribution is factorized into the product of distributions over smaller subsets of variables. The PGM framework offers many advantages, first and foremost being that it allows complex distributions to be written down tractably, while also providing a representation that is transparent so that it is easy to understand its semantics and properties. Secondly, it provides a structure for efficient inference, which will be discussed in detail in the next section.

2.2 Forney-style Factor Graphs and Message Passing

A Forney-style factor graph is used to represent a probabilistic model by exploiting factorization in which complex "global" functions can be represented by the product of smaller local functions [15]. An FFG comprises edges that represent variables and nodes that specify the relation between variables. The inference process in this context is performed by passing messages along the edges. Consider the example of a factorization of the joint probability distribution over the variables $x_1, ..., x_5$:

$$f(x_1, ..., x_5) = f_a(x_1, x_3, x_4) f_b(x_2, x_3) f_c(x_4, x_5) \tag{4}$$

Where we are interested in the marginal

$$q(x_4) = \int\!\cdots\!\int f(x_1, \ldots, x_5)\, dx_1 dx_2 dx_3 dx_5 \tag{5}$$

Substituting the factorized version of $f(x_1, ..., x_5)$ we get

$$q(x_4) = \int\!\cdots\!\int f_a(x_1, x_3, x_4) f_b(x_2, x_3) f_c(x_4, x_5)\, dx_1 dx_2 dx_3 dx_5 \tag{6a}$$

$$= \int\!\cdots\!\int f_a(x_1, x_3, x_4) f_b(x_2, x_3) f_c(x_4, x_5)\, dx_1 dx_2 dx_3 dx_5 \tag{6b}$$

$$= \underbrace{\iint f_a(x_1, x_3, x_4) \cdot \underbrace{\int f_b(x_2, x_3)\, dx_2}_{\mu_{f_b}(x_3)}\, dx_3 dx_1}_{\mu_{f_a}(x_4)} \cdot \underbrace{\int f_c(x_4, x_5)\, dx_5}_{\mu_{f_c}(x_4)} \tag{6c}$$

Equation (6c) shows that factorization allows breaking down higher dimensional integrals into smaller ones. The messages in the form $\mu_f(x)$ can be interpreted as summaries propagated from f towards x. This is known as the *sum-product rule* and its generic form is given as follows:

$$\overrightarrow{\mu}(y) = \int\!\cdots\!\int f(y, x_1, \ldots, x_N) \cdot \overrightarrow{\mu}(x_1) \ldots \overrightarrow{\mu}(x_N) \cdot dx_1 \ldots dx_N \tag{7}$$

This means that the message out of a factor node f_k towards edge y is the product of all messages towards f_k along all other edges and factor f_k integrated over all variables except for y. The sum-product algorithm computes two messages in opposite directions for every edge and the multiplication of these two messages results in the marginal distribution. To ensure that every message is computed exactly once, message computation begins from the leaves and proceeds with nodes whose inputs become available [25].

This also ensures that no message has backward dependencies and each message can be calculated from preceding messages [12]. Once the algorithm is executed, exact marginal distributions for all variables become available simultaneously. The FFG of the example given in equation (4) can be inspected in Fig. 1. Finally from equation (6c) we get

$$q(x_4) = \mu_{f_c}(x_4) \cdot \mu_{f_a}(x_4) \tag{8}$$

The process of summarizing (i.e. integrating) parts of an FFG iteratively and forward-

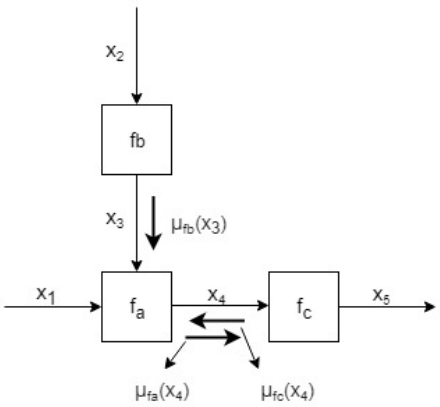


Figure 1: FFG of equation (4) depicting nodes, edges and the messages.

ing messages is known as *message passing* (or summary propagation). An important property of message passing is node-locality which arises from the way in which messages are computed. Specifically, the computation of a message flowing out of factor f_k can be computed from the incoming messages to node f_k and the analytical form of f_k. This means that generic message update rules for specific factor-message combinations can be derived and stored in look-up-tables to be used in the future which allows for the automated generation of inference algorithms (see section 2.5). Thus message passing becomes an efficient and automated process for doing probabilistic inference.

An example of how such a message update rule can be derived is given here for the equality constraint node which is one of the standard building blocks for FFGs (for other building blocks, see [25] Section 4.2). The equality constraint node is defined as the factor $f(x, y, z) = \delta(x - z)\delta(y - z)$ and for the incoming messages on edges x and y, the outgoing message is given by:

$$\mu_3(z) = \iint \mu_1(x)\mu_2(y) f(x, y, z)\, \mathrm{d}x\mathrm{d}y \tag{9a}$$

$$= \iint \mu_1(x)\mu_2(y)\delta(x - z)\delta(y - z)\, \mathrm{d}x\mathrm{d}y \tag{9b}$$

$$= \mu_1(z)\mu_2(z) \tag{9c}$$

From a Bayesian perspective, equality constraint nodes allow the execution of the Bayes rule by combining information from two sources. If message $\mu_1(x)$ is interpreted as the prior and message $\mu_2(y)$ as the likelihood function, then message $\mu_3(z)$ becomes proportional to the posterior distribution over variable z.

2.2.1 Variational Message Passing

Ideally, computing posterior distributions over latent variables would involve performing

exact inference. However, exact inference algorithms like the sum-product algorithm are typically only applied to discrete or linear-Gaussian models and are intractable for all but the simplest models. In most practical cases one must turn to approximate inference methods. Variational message passing (VMP) is such an approximate inference method that is suitable for many applications.

Variational methods have originated in the 18th century with the works of Euler, Lagrange and others on the calculus of variations. Simply put, many problems can be stated as optimization problems where the solution is obtained by exploring all possible input functions to find the one that either maximizes or minimizes the functional [5]. Although variational methods are not intrinsically approximate, they can be successfully applied to find approximate solutions. This is done by restricting the range of functions over which the optimization is performed. In the case of probabilistic inference for instance, this restriction takes the form of factorization assumptions.

In order to find an approximation of the true posterior distribution $p(s|x)$, the goal in variational inference is to find a tractable variational distribution $q(s)$ (also known as the *recognition* distribution) that minimizes a given "distance" measure. An approximate inference solution $q(s) \cong p(s|x)$ for a model $p(x, s)$ with latent variables s and observed variables x can be defined as a variational free energy functional:

$$F[q] \triangleq \underbrace{-\int q(s) \log p(x, s) \mathrm{d}s}_{\text{energy}} + \underbrace{\int q(s) \log q(s) \mathrm{d}s}_{-\text{entropy}} \tag{10a}$$

$$= \underbrace{-\log p(x)}_{-\text{log-evidence}} + \underbrace{\int q(s) \log \frac{q(s)}{p(s|x)} \mathrm{d}s}_{\text{KL-divergence}} \tag{10b}$$

The free energy functional measures the distance between the probability distribution of environmental variables that influence the agent and a recognition distribution encoded by its configuration. Assuming that the agent encodes a probabilistic model of the environment through its states and structure, minimizing free energy corresponds to refining this model to better represent its environment. Free energy can be minimized by changing the agent's configuration to affect the way it samples the environment or by changing the distribution it encodes. These changes correspond to action and perception respectively and thus free energy provides a unified perspective on how to interpret (and design) behavior.

2.3 The Free Energy Principle

The free energy principle (FEP) arose from a desire to explain how the brain works in a unified framework and it was first introduced by Karl Friston [19]. It states that any

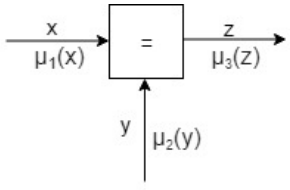


Figure 2: The equality constraint node allows the execution of the Bayes rule by combining information from two sources.

agent (be it a single-celled organism, a complex multi-cellular organism or even a social network of organisms) that strives to resist disorder, needs to minimize free energy [11]. This minimization can be over long periods of time as is the case in natural selection or in milliseconds during perceptual synthesis [16]. Simply put, all agents incorporate a model of their environment and the process of free energy minimization is the process of improving this model to better represent the environment.

Looking at (10b), it can be seen that the free energy functional consists of two terms: the KL divergence between the true posterior and the recognition distribution and the negative log-evidence. The negative log evidence in this setting is also known as *surprise*. To evaluate surprise, the hidden state variables of the environment need to be marginalized. However, this is in general intractable especially since for non-trivial models the posterior distributions over latent variables do not have an analytical form. Thus, free energy comes into play by providing an approximation of surprise that is much easier to work with. Since KL-divergence is always non-negative, it follows that free energy constitutes an upper bound on surprise and by minimizing free energy, one indirectly minimizes surprise. The KL-divergence term measures the distance between the recognition distribution and the conditional density of the environmental causes given the sensory observations, and hence minimizing free energy corresponds to making the recognition distribution get closer to the conditional density and the recognition distribution encoded by the agent's state becomes an approximation of the posterior probability of the causes of its sensory observations. Following this line of thought, agents that match their internal structure to the external causal structure of the environment are better at finding local of minima of F.

From one perspective, agents try to minimize free energy in order to occupy a limited number of desired states (e.g. upholding homeostasis in biological entities) and since the only channel through which information can be gained from the environment is sensory inputs, occupying desired states corresponds to making desired observations which are constrained by *target priors*. From this perspective, changing the sensory inputs requires acting upon the environment. This highlights the important interplay between action and perception in minimizing free energy: through perception agents can make better predictions about the environment and thus act upon the environment accordingly.

2.4 Active Inference

Active inference brings together perception and action under a unifying theory as a corollary of the Free Energy Principle. It tries to explain how self-organizing organisms interact with their environment and argues that living entities resist disorder (i.e. survive) by minimizing a variational free energy functional under a model of their environment. For biological entities, the specification of this model corresponds to minute refinements on the model over generations under natural selection and other evolutionary processes. On the other hand, for synthetic agents, a generative model can be specified by an engineer. Model specification constitutes a physical and statistical separation of the agent from the environment [37]. This separation can be achieved by a *Markov Blanket* that comprises the sensory and action variables and it defines the interface between the agent and its environment.

In Fig. 3, the environmental process constitutes action a_t, a latent state variable z_t, output y_t and it is specified as $(y_t, z_t) = R_t(z_{t-1}, a_t)$. It should be noted that the environmental process needs to be explicitly specified in case of a simulation, whereas it is unnecessary in real-world applications. On the other hand, the probabilistic generative model of the environment does need to be specified. The generative model is given by $p_t(x, s, u)$ where x, s and u are sequences of observations, internal states and controls

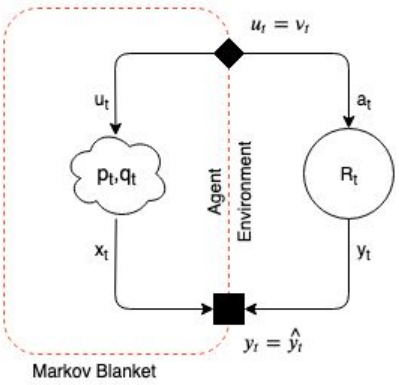


Figure 3: The interface between the agent and its environment [36]

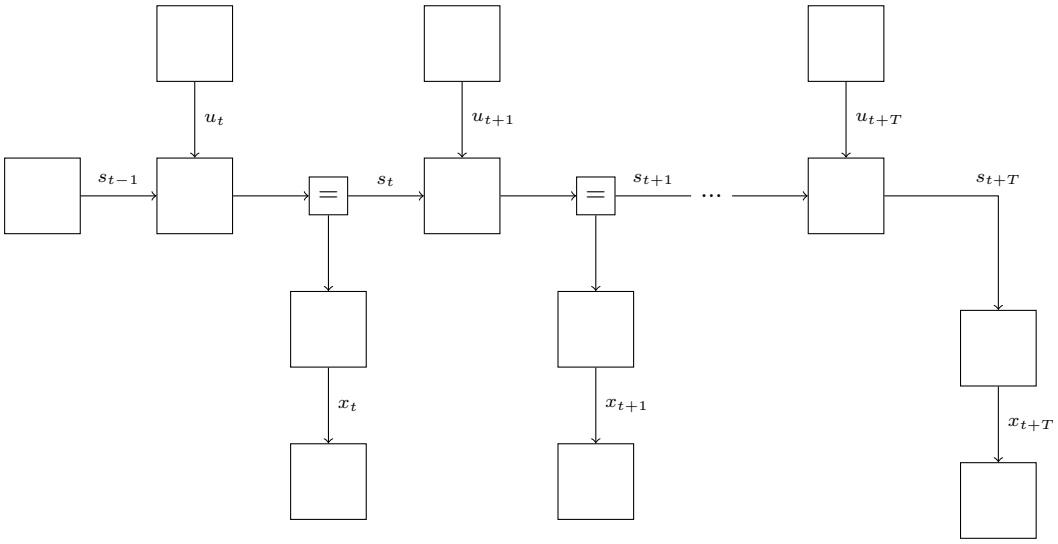


Figure 4: FFG of the generative model of eq. (11). Edges represent random variables and the nodes specify the relation between variables.

respectively. By providing a distinction between action as a state of the real-world and *beliefs* about future action given as control states, active inference changes the problem fundamentally from selecting optimal action to making optimal inferences about control [18]. In this context, minimization of free energy corresponds to minimization of prediction errors for the observations x and it leads to posterior distributions over both states s and controls u which in turn are observed by the environmental process as actions and indirectly leads to changes in the observations. Thus, inference for the states and controls both lead to free energy minimization.

Since the synthetic agents we strive to realize are built to fulfill specific goals, they need to be endowed with a sense of goal-directedness through extending the internal model over future states and incorporating counter-factual beliefs about future outcomes known as *target priors*. Target priors lead to high surprisal for observations that are deemed undesirable and free energy minimization consequently leads to posterior beliefs over controls that are believed by the agent to avoid these undesired observations.

For simple tasks, the target priors can be set in a relatively simple manner by hand (see the Bayesian Thermostat example in [36]). However, as the complexity of the task increases, a priori specification of target priors can become difficult. An approach that was pursued in [26] allows the model to *learn* desired goal priors on future observations by augmenting the agent with a separate model. Through this approach, a simple interface between the user and the agent can be constructed where the agent learns the goal states by receiving feedback from the user.

The agent's internal model is a reflection of the beliefs of the agent about how the environmental process generates observations from actions (hence the name *generative model*) and it is given by

$$p_t(x, s, u) \propto p(s_{t-1}) \prod_{k=t}^{t+T} \underbrace{p(x_k|s_k)}_{\text{observation}} \underbrace{p(s_k|s_{k-1}, u_k)}_{\text{state transition}} \underbrace{p(u_k)}_{\text{control}} \underbrace{p'(x_k)}_{\text{target}} \tag{11}$$

where the subscript t indicates that the model is time-varying and the context-based target priors for observations $p'(x_k)$ are distinguished from the predictions for observations $p(x_k|s_k)$ with a prime symbol. It should be noted that the model includes time steps $(t \dots t+T)$, effectively reflecting the ability of the model to be run forward to gather assumptions about future observations. Extending the model in this way with future time steps allows the model to reason about not only the consequences of action in the next time step but plan a series of actions, also known as a *policy*, to attain goals that are distant in time. Considering that there may be many different policies that may be undertaken, free energy is calculated for each policy at each time step and the one that has the minimum cumulative free energy in the future is picked.

The agent has access to a recognition distribution $q_t(x, s, u)$, next to its internal model. The recognition distribution comprises the prior beliefs of the agent over latent variables, including the future observation variables which are by definition unobserved. Initially the recognition distribution is set to an uninformative distribution and subsequent observation and inference steps allow the information encoded in the recognition distribution to increase gradually and become more precise. This process can be divided into three phases: (1) act-execute-observe, (2) inference and (3) slide.

In the act-execute-observe step, the internal model $p_t(x, s, u|u_{\leq t}, x_{\leq t})$ is updated to $p_t(x, s, u|u_{\leq t}, x_{\leq t}) \propto p_t(x, s, u|u_{\leq t-1}, x_{\leq t-1})\delta(u_t - \upsilon_t)\delta(x_t - \hat{y}_t)$ where $p_t(x, s, u|u_{\leq t-1}, x_{\leq t-1})$ and $p_t(x, s, u|u_{\leq t}, x_{\leq t})$ reflect the state of the generative model just before and after the t^{th} act-execute-observe step. Control parameters υ_t are determined by free energy minimization in the previous step and $\hat{y}_t$ is the output of the environmental process observed

by the agent. Note that the execute step is processed by the environmental process after it receives an action from the agent.

Next, in the inference phase, the consequences of updating the internal model need to be processed for the model's latent variables through free energy minimization. Here, the recognition distribution is parameterized by sufficient statistics μ as $q_t(x, s, u) = q(x, s, u|\mu_t)$ and μ_t is computed by free energy minimization:

$$\mu_t = \arg\min_{\mu} \int q(x, s, u|\mu) \log \frac{q(x, s, u|\mu)}{p_t(x, s, u)} \mathrm{d}x\mathrm{d}s\mathrm{d}u \tag{12}$$

Finally, in the slide phase, the first time slices of the internal and recognition models are marginalized and a new time slice is added to the horizon. After the slide step, the algorithm loops back to another act-execute-observe step.

2.5 Automated Generation of Inference Algorithms

Next to the advantages discussed earlier in section 2.2, message passing also provides a convenient paradigm for automated generation of inference algorithms. The main advantage of this approach is that by automating the inference and performance evaluation (which is also a Bayesian inference task) steps, it allows the designer to quickly loop through the design cycle by proposing alternative models until the performance evaluation criteria is satisfied. Generation of inference algorithms can be automated through constructing appropriate message passing schedules and by constructing a look-up table for pre-derived message update rules. More specifically, inference algorithm generation involves: message update scheduling, update rule selection and message type inference (based on the factor node, inbound message types and the required form of the outbound messages) and lastly code generation where the sequence of message updates is compiled to source code [12]. Note that since the final result is source code, the user has flexibility in manually modifying the algorithm at any level, which comes in handy for building problem specific algorithms.

ForneyLab is a software tool that facilitates automatic generation of inference algorithms. It is written in a high-level language called Julia [4], which offers MATLAB-like syntax and speed comparable to compiled C code. The design cycle in ForneyLab comprises three phases.

In the first phase, the **build** phase, the user specifies a probabilistic model using a domain-specific syntax that resembles other probabilistic programming languages. This involves using macros and one of the main advantages is that even complex models can be specified in less than a page while the generated inference algorithm may be thousands of lines long. ForneyLab builds an FFG of the specified model which provides modularity and enables the re-use of computational inference primitives.

The next phase is the **schedule** phase and this is where the user specifies an inference problem. ForneyLab subsequently proceeds to automatically deriving a message passing algorithm that computes the marginals for desired variables.

Finally, in the **infer** phase, ForneyLab parses and executes the generated inference algorithm [12].

3 APPLICATION TO AUTONOMOUS AGENTS

The purpose of this project is to implement a robot-car that uses active inference in order to navigate in a previously specified area to find and park in the location chosen by the

user. In the beginning, the prior belief of the agent is initialized to an uninformative distribution as to which location the user might choose. This belief is subsequently updated and refined with each iteration through moving to a new location and requesting feedback from the user asking whether the new location is better or worse compared to the previous one.

An intelligent autonomous agent that is capable of achieving the goal stated above may comprise three main functional blocks:

- The pre-processing block acts as the interface between the agent and the environment. It gathers data from various sensors, converts them to the appropriate forms to be fed in as observations and receives actions from the physical model to actuate the motors.
- The physical model is responsible for inferring the necessary actions to move the robot from initial position a to final position b, given the observations received from the pre-processing block and the target priors from the target model.
- Finally, the target model is responsible for receiving user feedback to infer beliefs about the location chosen by the user. These beliefs are subsequently used as goal priors in the physical model to achieve goal-directed behavior.

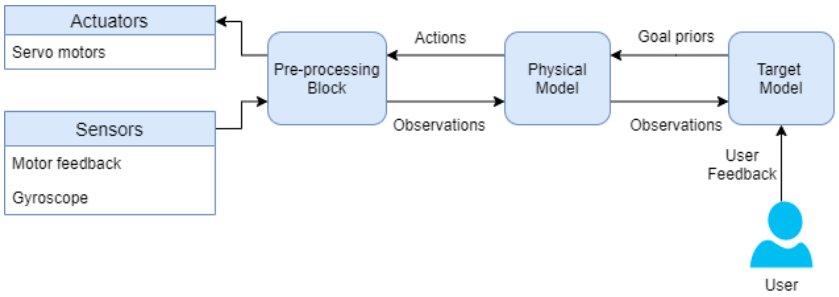


Figure 5: Functional blocks of the active inference-based parking agent. Functional blocks allow a modular approach.

Figure 5 highlights an important advantage of using an MBML approach: modularity. Since the blocks are functionally separate and the interfaces between them are clearly defined, each block can be implemented separately without having to worry about the other blocks. The following subsections will focus on the implementation of each of these blocks in detail. However an overarching theme is the design and iterative improvement of model architectures in a simulation environment first and subsequently porting them to the robot to make the necessary adjustments on model parameters to design specific behavior.

Box's loop provides an established approach to model development and it summarizes the constituting phases in a design cycle. As mentioned earlier in section 1, this paper uses a combination of complementary methods that work together to enable one of the main advantages of active inference which is fast design cycles . A constituent part of this is the rigorous application of Box's loop which has been followed throughout the implementation phase. Although it seems that the main focus of Box's loop is the iterative improvement of model architectures, as we shall see, design cycles can be extended to iterations after applying the model. Having designed a flexible model in the initial iterations, specific behavior on the agent's part can further be enforced without changing the model architecture. This process can be seen in detail in section 3.3.1.

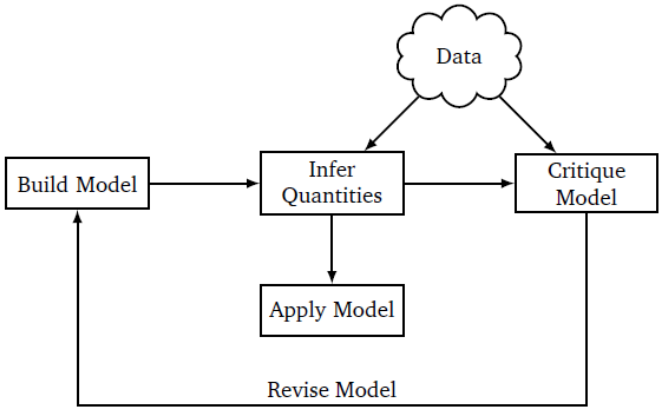


Figure 6: Box's loop summarizes the design cycle [36] [7].

In the following sections, we will first introduce the physical setup of the agent in section 3.1 where the implementation of the pre-processing block is also discussed. Next comes the Bayesian sunflower section 3.2, a small project that was undertaken in order to gain hands on experience with active inference on a relatively simple project and to identify problems with the robot setup. The following two sections focus on the implementation of the physical model, section 3.3.1, and the target model, section 3.3.2.

3.1 Robot Setup and the Pre-Processing Block

For an active inference agent, having a good model of the environment does not necessarily translate to successfully achieving a specified goal in practice. Real-world active inference agents are also constrained by how they interact with their environment and this essentially comes down to the range and accuracy of their sensory modalities and the capabilities of their actuators. Clearly, before undertaking an active inference implementation for a real-world application, an agent needs to be designed that has the necessary physical attributes.

In order to identify which physical attributes an agent should possess, its goal and how this goal will be attained need to be analyzed.In this project, the agent needs to be capable of:

- Keeping track of its position and orientation accurately and
- Inferring actions reasonably fast.

These two points have to be satisfied in order for the agent to be capable of achieving its goal. Let us now look at how they were satisfied for this project.

The first thing to do was to choose a chassis. There are a large number of chassis with different shapes and sizes to choose from on the market. However, it comes down to three main form factors: robots with wheels, tracks and spider-bots. Spider-bots can immediately be crossed off since they present unnecessary complexity. Then, the choice comes down to either a robot with wheels or one with tracks. The difference is that although tracked robots perform well on rugged terrain, they are more prone to slipping on smooth surfaces. Thus, a robot with wheels was chosen. Parallax's robot shield for Arduino has two wheels on the front, a dummy wheel on the back and it provides enough space for prototyping [22]. Parallax also offers a variety of components that are

suitable for this robot shield. This robot shield comes with continuous rotation servo motors. Next, we'll look at which micro-controllers were used in this project.

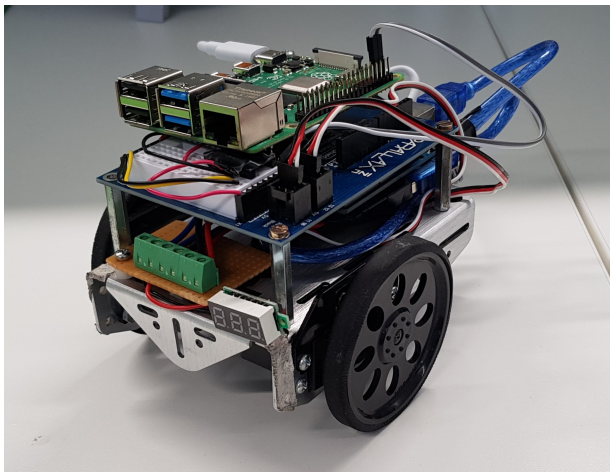


Figure 7: Image of the robot that was used in the current study.

Processing information

Considering this is a proof-of-concept project, we have opted to go for one of the most popular and powerful embedded devices on the market: a Raspberry Pi 4 [20]. The Raspberry Pi 4 has a 64-bit quad core Cortex-A72 microprocessor that has a clock speed of 1.5 GHz and it comes with a 4 GB SDRAM. While these specs ensure sufficient performance for our purposes, Raspberry Pi 4 also runs a free operating system based on Debian which allows easy installation of Julia [4], a high-level programming language for high performance numerical analysis and computational science which is required for running ForneyLab [12].

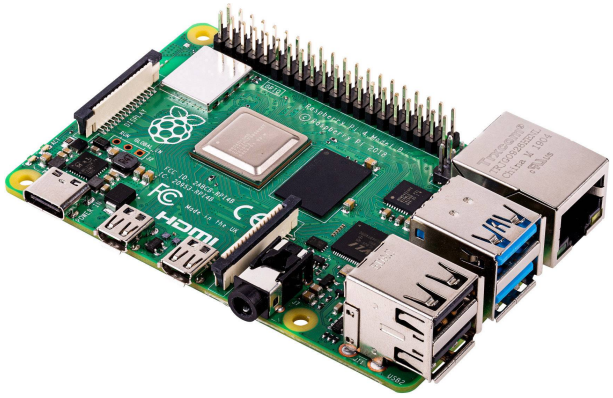


Figure 8: The Raspberry Pi 4 (RPi4). A RPi4 was used as a platform for executing free energy miniminization (coded in Julia, running on Raspberry Pi's Linux variant).

Alongside the Raspberry Pi, an Arduino Uno is used [2]. Arduino Uno is an MCU board that is based on the ATmega328P. It has 16 digital I/O pins, 6 analog inputs and it has a clock speed of 16 MHz. The Arduino Uno provides ease in controlling motors (through its various libraries and dedicated hardware) and gathering sensor readings, so

it has become very popular in robotics projects over the years. In our implementation, it also provides a simple form of parallelism so that inference can take place without being interrupted while the previous action is being executed.

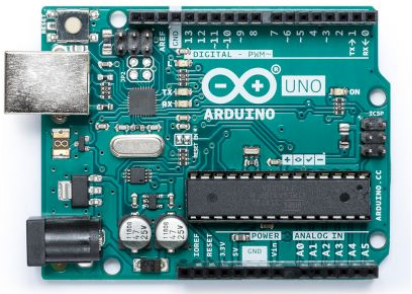


Figure 9: An Arduino Uno was used to gather sensor readings, actuate the motors and sample analog voltages.

Accurate localization

Although the Global Positioning System (GPS) has revolutionized outdoor localization, indoor localization is still an unsolved problem . Both academia and the industry have been spending resources and effort on trying to solve the problem for well over a decade now and even though hundreds of different approaches have been proposed, the community has still not converged on a single solution that satisfies the required localization accuracy at low cost [30].

Indoor localization approaches can be divided into infrastructure-based and infrastructure-free approaches. The infrastructure-based approach requires the deployment of custom hardware such as Bluetooth beacons, ultrasound speakers, cameras and others. Infrastructure-free approaches, on the other hand, do not require custom hardware to be deployed in the working area. During the execution of this project, both approaches have been studied and several variations of them have been considered. However, since an in-depth comparison of these approaches and their variations is out of the scope of this paper, we will focus on several variations that seemed the most promising here. For an in-depth comparison of 22 different indoor localization systems, see [30].

Perhaps the most commonly used infrastructure-free approach is dead reckoning. In dead reckoning, the current position of a mobile entity is calculated by advancing a previously known position using estimated speed over time and course [14]. Dead reckoning systems commonly combine readings from several sensors in order to get more accurate location estimates. These sensors include wheel encoders, gyroscopes and accelerometers among others. Although dead reckoning has many advantages such as being easier to implement, computationally inexpensive and low cost, its main drawback is the accumulating error over long periods of time. These accumulating errors are mostly due to there being no external frame of reference to bound the error and they arise from wheels slipping and drift in the inertial sensors. However, if these errors are kept to a minimum, reasonable localization accuracies can be achieved depending on the application.

In most dead reckoning implementations, encoders are used to provide odometric information. However, encoders have several disadvantages including (relatively) low resolution (usually dependent on how many slits there are on the wheel) and the need to constantly count every pulse generated. Note that when a pulse from the encoder is

missed, an error is introduced proportional to the size of the wheel and inversely proportional to the resolution of the encoder. In our implementation, we have decided to use digital angular position feedback [32]. This feedback is generated by a hall effect sensor inside the motor and its output is a pulse wave of fixed frequency and varying duty cycle. Angular position information can easily be calculated from this duty cycle. The resolution of this feedback is higher than most encoders used in robotics applications and it can be measured at any point in the algorithm as long as the wheels have not completed a full rotation, thus minimizing errors and decreasing computational cost.

Alongside the angular position feedback, a gyroscope was used to provide orientation readings. One of the most commonly used sensors in the market is MPU-6050 [23]. It is a 6 axis Inertial Measurement Unit (IMU) that provides high resolution readings with low power consumption ratings. One of MPU-6050's most important features is that it also has a Digital Motion Processor (DMP) which decreases the computational load on the Raspberry Pi. Communication with MPU-6050 is achieved through I2C.

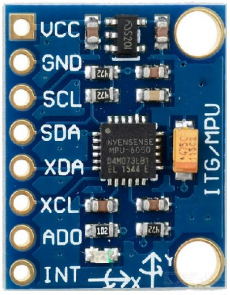


Figure 10: MPU-6050 is a 6 axis Inertial Measurement Unit (IMU). It was used to get readings on the robot's orientation.

As seen in Fig. 5, the pre-processing block is responsible for receiving sensor readings and calculating the position and orientation updates to be fed in as observations to the physical model in the active inference-based parking robot. The orientation of the agent ϕ_k, is initialized to $\pi/2$ radians through adjusting the gyroscope readings with an initial offset reading and subsequently, angular displacement $\Delta\phi_k$ (radians) is integrated to this value in every iteration. Next, the current position of the agent (x_k, y_k) is calculated from its previous position (x_{k-1}, y_{k-1}) by first calculating the displacement of the robot, Δ_{pos} (cm), given the wheel radius r (cm), angular displacements of the left and right wheels, Δ_{left} (degrees) and Δ_{right} (degrees), using the following equations:

$$\Delta_{\text{pos}} = \frac{(\Delta_{\text{left}} + \Delta_{\text{right}})}{2} \cdot \frac{2\pi r}{360} \tag{13a}$$

$$x_k = x_{k-1} + \Delta_{\text{pos}} \cdot cos(\phi_k) \tag{13b}$$

$$y_k = y_{k-1} + \Delta_{\text{pos}} \cdot sin(\phi_k) \tag{13c}$$

Note that these equations treat a curved trajectory as a combination of straight lines, so they provide an approximation of the real case, see Figure 11. However, as long as they

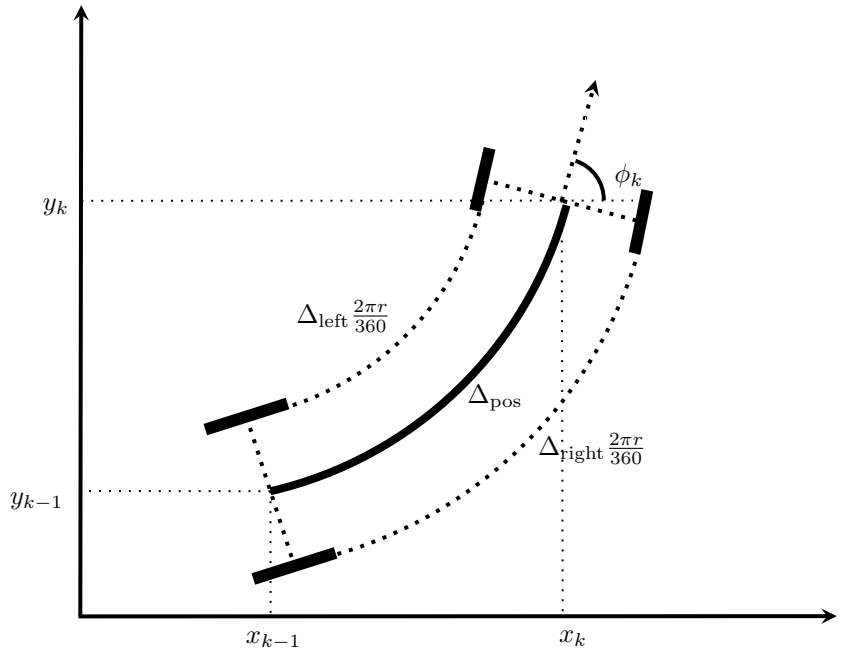


Figure 11: In dead reckoning, the current position of a mobile entity is calculated by advancing a previously known position using estimated speed over time and course.

are executed frequently enough (in our case every 2-3 ms), they prove to be reasonably accurate [29] [9].

Although there's no way of preventing the wheels from slipping in a practical setup, its effect on localization accuracy can theoretically be decreased through augmenting the agent with optical mouse sensors. The intuition behind this approach is that an optical mouse sensor provides a secondary source of information on the displacement of the agent that is independent of the wheels. Several articles in the literature report high localization accuracies where information from several optical mouse sensors (at least two) are coupled with information from encoders [33] [8]. However, this approach has several drawbacks. First of all, the resolution of an optical mouse sensor highly depends on the surface it's being used on and it needs to be calibrated for each new surface. Secondly, these sensors are designed to work a couple of millimeters from the surface and anything beyond that renders them unusable. The height sensitivity can be reduced by augmenting the sensor with a lens as explained in [3], however, this would be too time consuming.

Several infrastructure-based off-the-shelf modules were also investigated including Bluetooth and WiFi-based implementations. However, at the time none of the readily available modules were capable of producing the accuracy that is required in this project. After dead reckoning was implemented and was working reasonably accurately, an indoor localization system based on ultrasound and RF communication became available on the market that would have suited this project well [34]. Using (at least) three beacons that are deployed in the indoor working environment, it provides a reported localization accuracy of ± 2 cm within a 50 m range. Using this indoor localization system would have probably reduced the time spent in development.

Light Direction Sensor

In the Bayesian sunflower project (section 3.2), an active inference agent is built that continuously rotates towards the light source. Hence, the agent needs to have access to a sensor that reports the direction of the incident light. Although the market is rife with light presence sensors, a light direction sensor could not be found. However, building one out of two Light Dependent Resistors (LDR) is relatively straightforward. An LDR, also known as a photo resistor, is a type of resistor whose resistance depends on the amount of light falling on its surface. Typically, the resistance of an LDR is inversely proportional to illuminance.

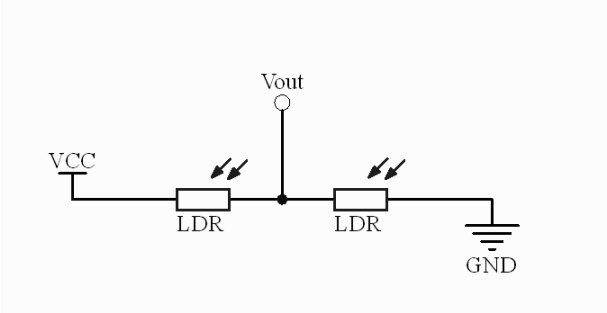
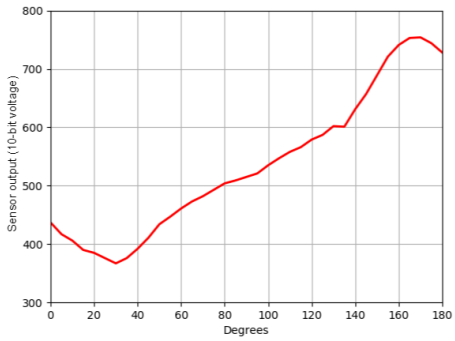


Figure 12: Schematic of the light direction sensor. The light direction sensor was used in the Bayesian sunflower project to get readings on the direction of the light source.

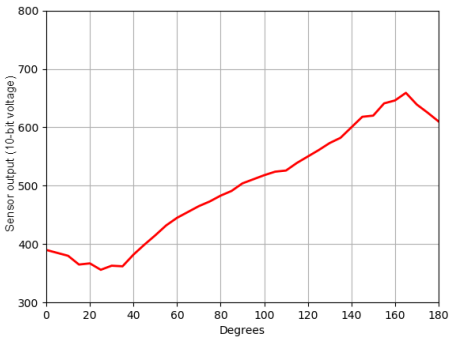
Figure 12 shows the schematic of the light direction sensor. The idea behind this sensor is that when the light source is directly in front of the sensor, the resistances of the two LDR's are equal and hence the voltage sampled is equal to 2.5 V. On the other hand, when the light is falling on the sensor at an angle, depending on which LDR the light source is closer to, the output is greater or less than 2.5 V. However, initial experiments with the sensor have shown that the voltage sampled when the light source is directly in front of the sensor is not exactly equal to 2.5 V since the response characteristics of the two LDR's do not match each other exactly. This is due to slight differences in the manufacturing process and ambient light conditions. These effects can be alleviated in practice by simply taking an initial offset reading and adjusting subsequent readings with this offset to get 2.5 V when the light source is in front of the sensor.

As we shall see in section 3.2, building the generative model requires information about the response characteristics of the light direction sensor. In order to find this, an experimental setup was prepared where initially an offset reading was taken with the flashlight directly in front of the robot and subsequently, starting at 0 degrees, the flashlight was moved around the robot at a certain distance in 5 degree steps for a total of 180 degrees. At each step the response of the sensor was recorded and adjusted using the offset reading. This process was repeated under different ambient light conditions and at different distances from the sensor. The results can be seen in Fig. 13.

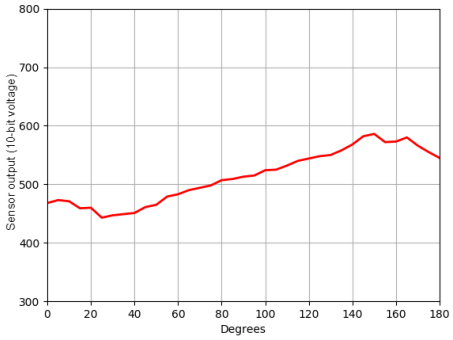
There are several things to note in this figure. First of all, since the light direction sensor is an analog sensor and the Arduino's analog-to-digital converter (ADC) has a resolution of 10 bits, the sensor reports values in the range [0, 1023]. Second, the values reported at 90 degrees is around 512 (corresponding to 2.5 V) even under different conditions. This means that the goal prior for the Bayesian sunflower can directly be set to 512, without worrying about varying environmental conditions. Third, the response can be assumed to be linear from 20 degrees to 160 degrees. Finally, the distance of the light source is inversely proportional to the slope of the response. As we'll see, this effect will



(a) Ambient light coming from the left, light source distance: 50 cm.



(b) Ambient light coming from the back, light source distance: 50 cm.



(c) Ambient light coming from the back, light source distance: 75 cm.

Figure 13: Response characteristics of the light direction sensor under different conditions. The light direction sensor provides accurate readings when the light source is directly facing the sensor even under different environmental conditions.

result in smaller controls (i.e. slower rotation) when the light source is farther from the agent.

Communication

Various components that make up the agent need to have access to a robust communication protocol. In our implementation we have decided to use I2C since the Arduino Uno, MPU-6050 and the Raspberry Pi 4 all have dedicated hardware for I2C. I2C is a synchronous, serial computer bus that was developed by Philips Semiconductor. It is widely used for attaching peripheral ICs to microcontrollers for short distance communication.

Initially, the idea was to execute the pre-processing block solely on the Arduino, so it would gather readings from both the angular position feedback sensor and the MPU-6050 to calculate position and orientation and subsequently send them as observations to the physical model on the Raspberry Pi. However, this proved not to be a particularly robust approach since it required concurrent communication or withholding communication until the current one was finished on Arduino's part. For example, when the Arduino was communicating with the MPU-6050, the Raspberry Pi could request observations from the Arduino which resulted in interference on the channel and loss of packages. On

account of this, a more strict hierarchy was implemented next where the Raspberry Pi became the master and the Arduino and MPU-6050 became slaves. In this framework, only the Raspberry Pi can initiate communication sessions and hence there cannot be interference on the channel due to concurrent access to the channel. Note that in this implementation, the Arduino does not receive any orientation information and so it cannot calculate position updates. Thus, some processes of the pre-processing block were distributed among the Arduino and the Raspberry Pi. In this scheme, the Arduino keeps track of the angular position of the wheels and MPU-6050 keeps track of the orientation of the agent and Eq. 13 is calculated on the Raspberry Pi.

The paragraph above describes an initial iteration on the design cycle, specifically concerning the pre-processing block in this case. In the following sections, many more iterations on the design cycle are introduced. As a general approach to building real-world active inference agents, initial iterations on the design cycle are executed in simulation environments where model architectures are built. Once satisfactory results are obtained in simulations, models are ported to the robot and later iterations on the design cycle are carried out specifically focusing on real-world performance. While initial iterations usually concern refinements to model architectures, later iterations may also include updates to functions facilitating the interface between the agent and the environment in order to improve real-world performance.

3.2 The Bayesian Sunflower

The Bayesian sunflower is based on the Bayesian thermostat, a classical active inference application [17] [10] where the agent is placed in a nonlinear temperature gradient and where it subsequently positions itself according to the desired goal temperature. The Bayesian thermostat was explored extensively in [36] where a simulation was also provided. The Bayesian sunflower on the other hand, describes an agent that actively rotates towards the direction of the light source much like a sunflower, hence the name. In the Bayesian sunflower, the agent receives light direction values as observations which are subsequently used to infer controls in terms of rotation velocities in the agent's generative model. The goal prior is set to 512 (i.e. the agent is directly facing the light source).

The Bayesian sunflower project was designed primarily to gain hands on experience with active inference on a real-world agent and on a relatively simple setup where possible problems with hardware could be caught early on. The advantage of the Bayesian sunflower over the Bayesian thermostat is that setting up the physical experiment is easier since only a flashlight is required instead of a heat source and it's easier to observe the agent's dynamic behavior (e.g. imagine walking around the robot with a flashlight and seeing the robot immediately responding to it as opposed to changing the temperature of the heat source).

Environmental Model Specification

The environmental process was specified using the response characteristics of the light direction sensor. The response of the light direction sensor $\mathcal{L}(z)$, given the orientation of the agent with respect to the light source z (degrees) is given as:

$$\mathcal{L}(z) = 2.76 \cdot (z - 90) + 512 \tag{14}$$

where the coeffecient 2.76 is the slope of the response characteristic when the light source is 50 cm away from the agent. Actions a_t, given in terms of rotation velocity, affect the environment as follows:

$$z_t = z_{t-1} + a_t \tag{15}$$

and the agent is assumed to have access to noisy observations

$$y_t \sim \mathcal{N}(\mathcal{L}(z_t), 10^{-2}) \tag{16}$$

Generative Model Specification

Moving on to the generative model, the goal prior directs the agent's actions to achieve a desired observation which is specified as $x^* = 512$ (corresponding to 90 degrees, or a reading of 2.5 V from the light direction sensor).

$$p'(x_k) = \mathcal{N}(x_k|x^*, 10^{-2}) \tag{17}$$

The state transition model is given as

$$p(s_k|s_{k-1}, u_k) = \mathcal{N}(s_k|s_{k-1} + \alpha \cdot u_k, 10^{-2}) \tag{18}$$

where the coefficient α provides a simple way of adjusting the mean of the control u_t and hence the velocity of the robot. It will be useful especially once the model is ported to the robot.

The observation model reflects the assumption that the response characteristic is linear and it is given as follows:

$$p(x_k|s_k) = \mathcal{N}(x_k|2.76 \cdot s_k, 10^{-2}) \tag{19}$$

Finally, priors for controls and the initial state are specified as:

$$p(u_k) = \mathcal{N}(u_k|0, 10^{-2}) \tag{20a}$$

$$p(s_0) = \mathcal{N}(s_0|0, 10^{12}) \tag{20b}$$

The generative model was implemented in ForneyLab, which can be used to automatically generate a message passing algorithm for free energy minimization.

Simulation Results

The simulation protocol follows the (1) act-execute-observe, (2) infer, (3) slide loop as explained in 2.4 and it was run for 100 time steps with a horizon of $T = 2$. Actions were allowed after the 25th iteration. It can be seen in Fig. 14 that immediately after the agent is allowed to act, it quickly rotates towards the light source and it displays very stable behavior once the goal is reached. Notice that as the agent gets closer to the goal, its actions become smaller. This prevents overshooting and in general results in stable behavior once the goal is reached. In the 50th iteration the light source is moved 20 degrees to simulate a mobile light source. The agent again rotates towards the light source and then stops.

Experiments on the robot

The simulation results act as a proof of concept and the next step is porting the model to the robot itself in order to see its performance in a real-world setup. The main difference between a simulation protocol and an experimental protocol is that in a simulation protocol, the environment is simulated whereas in the experimental protocol the environment is the world itself. Since the act-execute-observe step creates a nice separation between the agent and the environment, porting the model to the robot is rather straightforward and only requires converting the act() and observe() functions to communication steps. More specifically, in the act step the agent sends the mean of

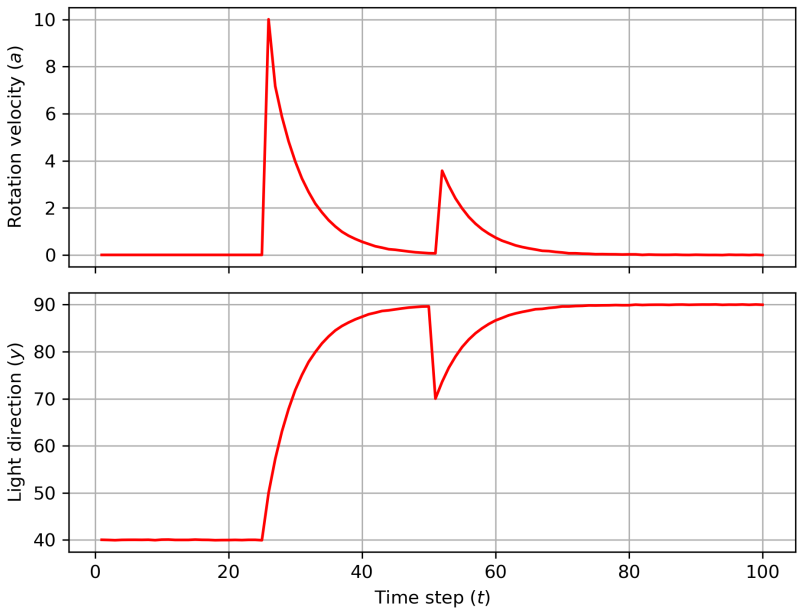


Figure 14: Simulation results of the Bayesian sunflower. After the agent is allowed to act, it quickly rotates towards the light source. In order to simulate a dynamic light source, in iteration 50 the light source is moved 20 degrees.

u_t (which specifies rotation velocity) to the Arduino, and upon receiving this value, the Arduino produces the necessary electrical signals to actuate the motors. In the observe step on the other hand, the agent requests a light direction reading from the Arduino.

Although porting the model to the robot sounds simple in theory, it should be noted that a working model requires all the underlying processes to work as well. These processes are mainly related to the pre-processing block and include getting correct readings from the sensors, being able to actuate the motors at the desired velocities and achieving a robust communication protocol. Many of these processes need to run in parallel (e.g. electrical signals with correct duty cycles need to be sent to the motors at all times, even when the motors are not turning and communication needs to take place alongside this), so it is necessary to make sure that these processes do not interfere with each other. The implementation of the pre-processing block was discussed in detail in 3.1.

Once the model is up and running on the robot, the behavior of the agent can be fine-tuned. An implicit assumption in the simulation protocol is that each iteration takes one second. Although this makes implementing the simulation protocol easier, it does not reflect the actual case where each iteration takes much less than a second (2-3 ms). Hence, the controls generated in the simulation protocol (which were given without a unit) need to be multiplied by a certain factor to adjust the rotation velocity of the robot in the experimental protocol. This can be achieved by making use of the α coefficient mentioned in the generative model specification subsection. The coefficient α is selected so that the robot is neither too fast that it's overshooting nor is it too slow that its response is too slow.

A video of the Bayesian sunflower can be found here `https://youtu.be/P4yB9L8LHeo`.

3.3 An Active Inference-based Parking Agent

Having mostly set up the robot and the accompanying functions related to the preprocessing block in the previous phase, it's time to move on to the implementation of the active inference-based parking agent. This project offers a task of appropriate complexity for a proof of concept while also allowing the exhibition of active inference's capabilities.

In this project, the agent's goal is to consecutively navigate to new target locations (inferred from user feedback) and eventually settle in the location chosen by the user. The parking agent comprises two generative models: a physical model and a target model. Let us first look at the physical model.

3.3.1 The Physical Model

The physical model is responsible for inferring the controls necessary for navigating the agent from any position a to position b. The observations it receives from the preprocessing block are in terms of position and orientation. As in the Bayesian sunflower, the physical model in the parking agent follows the (1) act-execute-observe, (2) infer, (3) slide loop. The inferred controls are in terms of translation and rotation velocities and they are used in implementing a differential steering scheme [9]. Differential steering applies to ground-based vehicles where more or less torque is applied to one side of the vehicle than the other. Differential steering can produce curved as well as straight trajectories [9].

In the initial iteration of the design cycle for the physical model, a model was considered where the agent would first rotate towards the target and subsequently start moving towards it. Differential steering would still apply in order to make sure the agent did not stray from the path. This behavior would be achieved by constructing two generative models. The first model would be responsible for inferring translation velocities by observing the Euclidean distance to the target and the second generative model would be responsible for inferring rotation velocities by observing the angle between the agent's orientation and the target. However, this idea was later dropped in favor of the following model which has one generative model (and hence one inference step). While the following model is capable of displaying the same behavior as the model initially proposed, it can also display a wider range of behaviors.

Generative Model Specification

The generative model specification starts by specifying the transition model which describes the relationship between the current state, the previous state and the controls. The states of the generative model are given by $s_k = (x_k, y_k, \phi_k)$ where (x_k, y_k) specify the position of the agent and ϕ_k the orientation. Controls are given by $u_k = (\Delta\phi_k, r_k)$ where $\Delta\phi_k$ specifies rotation velocity and r_k specifies translation velocity. The transition model is given as follows:

$$s_k = g(s_{k-1}, u_k) \tag{21}$$

where $g(s_{k-1}, u_k)$ is given by

$$\phi_k = \phi_{k-1} + \Delta\phi_k \tag{22a}$$
$$x_k = x_{k-1} + r_k cos(\phi_k) \tag{22b}$$
$$y_k = y_{k-1} + r_k sin(\phi_k) \tag{22c}$$

$$p(s_k|s_{k-1}, u_k) = \mathcal{N}(s_k|g(s_{k-1}, u_k), 10^{-1} \cdot I_3) \tag{23}$$

To couple the observations with internal states, the observation model is given by

$$p(x_k|s_k) = \mathcal{N}(x_k|s_k, 10^{-1} \cdot I_3) \tag{24}$$

Finally, the goal priors are specified as follows

$$p'(x_k) = \mathcal{N}(x^*, 10^{-2}) \tag{25a}$$
$$p'(y_k) = \mathcal{N}(y^*, 10^{-2}) \tag{25b}$$
$$p'(\phi_k) = \mathcal{N}(\phi_k^*, 10^{-2}) \tag{25c}$$

The goal priors for the target location (x^*, y^*) are entered manually at this stage. However as we shall see, the target model (section 3.3.2) will be providing the goal priors for the target location later on. The goal prior for orientation is calculated at each iteration using the following equation:

$$\phi_k^* = \arctan\left(\frac{y_k - y^*}{x_k - x^*}\right) \tag{26}$$

where ϕ_k^* specifies the angle between the agent's current position and the target position. Specifying the goal prior for orientation this way ensures that the agent actively rotates towards the target position in each iteration.

Simulation Results

The simulation is run for 30 time steps with a horizon T=2. To make the simulation closer to the actual case, the actions of the agent are constrained to $|\Delta\phi_k| < 0.2\pi$ and $|r_k| < 10$. The results can be seen in Fig. 15. The results show that the physical model is capable of navigating the agent to reach its target. It can also be seen that as the agent gets closer to the target its actions become smaller.

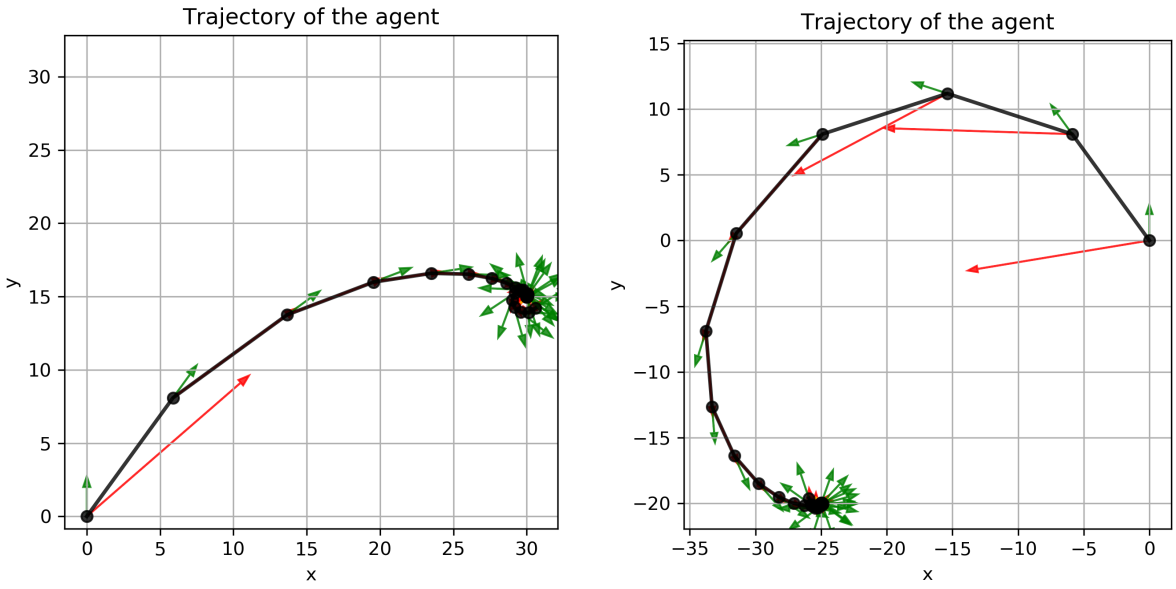
Experiments on the robot

Having achieved promising results in a simulation environment, the next step is porting the model to the Raspberry Pi. The process is similar to the one explained in section 3.2 meaning that the act and observe steps become communication steps. However note that some functions related to the pre-processing block are also executed in the act and observe steps. Eqn. 13 is executed in the observe step after sensor readings are received and in the act step, inferred controls are converted to a more appropriate form to be sent to the Arduino. The following equations show how the velocity for each motor is calculated.

$$\nu_{left_k} = \alpha \cdot r_k + \beta \cdot \Delta\phi_k \tag{27a}$$
$$\nu_{right_k} = \alpha \cdot r_k - \beta \cdot \Delta\phi_k \tag{27b}$$

As it can be seen, while r_k acts as a base velocity for both motors, $\Delta\phi_k$ creates a difference between them to enforce differential steering. Coefficients α and β are used to fine-tune the translation and rotation velocities respectively.

For experiments with the robot, a setup was prepared with a large sheet of paper as a working surface to decrease slippage where several target locations were marked. The experiments done were similar to the simulations given previously. Initial experiments showed a noticeable decrease in localization accuracy as the robot moved especially in cases where the robot had to rotate more (like in Figure 15b). Further experimentation was conducted to see if this problem would persist (1) when the robot had to move



(a) Target position: (30,15)

(b) Target position: (-25,-20)

Figure 15: Simulation results of the physical model. Green arrows show the orientation of the agent and the red arrows show the proposed motion for the next iteration.

in a straight line and (2) when it only had to rotate to achieve a specified orientation. In both cases, no discernible decrease in localization accuracy was seen even when the experiments were consecutively repeated without resetting the agent's location to its initial state (i.e. $x_0 = 0$, $y_0 = 0$, $\phi_0 = \pi/2$). So the decrease in localization accuracy seemed to be occurring when translation and rotation were happening simultaneously and if the robot needed to rotate a considerable amount. This problem is probably due to the limitations of Eqn. 13 combined with slippage. However, it was not investigated further since preventing it is rather simple as will be explained next, and it lets us highlight an important feature of active inference.

In order to overcome the decrease in localization accuracy, no changes to the model architecture are required. Since the physical model describes two-wheeled robot motion in 2D so well, it provides a degree of flexibility in enforcing specific behavior. By suppressing the control for translation velocity when the angle between the agent's orientation and the target orientation is greater than a certain amount ($|\phi_k - \phi_k^*| > \psi$), the amount the agent needs to rotate while translating can be decreased. This effectively results in the robot *pivoting* where it stands until its orientation is a specific amount away from the target. In our implementation this value was chosen as $\psi = pi/6$ radians which results in observable curved trajectories while also bringing the decrease in localization accuracy down to an acceptable level. Note that if ψ is set to zero, the agent only moves in straight lines.

The way the decrease in localization accuracy is brought down to an acceptable level, as explained above, lets us see how the design cycle can be extended to iterations after the model is applied. It also highlights that design cycles should not only consider changes to the model architecture, but a designer should also be aware of what is possible with the model at hand. Having designed a flexible model in the initial iterations of Box's loop, it is possible to enforce specific behavior without changing the model architecture.

The current implementation was also put to the test to see if its localization accuracy was acceptable. In this project, an "acceptable localization accuracy" is not strictly defined. However, as we shall see in the target model section, the agent needs to be capable of consecutively navigating to new target locations around 35-40 times inside a 3x3 m^2 area before it settles in the location chosen by the user. The agent's localization accuracy was tested keeping this constraint in mind and it was confirmed that the agent was capable of navigating accurately with further experimentation. The physical model's robustness was also tested by forcefully changing the agent's trajectory en route. These tests showed that the agent was capable of adapting to the situation by adjusting its actions where necessary, attesting to active inference's robustness under dynamic environmental conditions. A video of these tests can be found here: `https://youtu.be/AJevoOmKMO8`.

3.3.2 The Target Model

In an active inference setting, goal-directed behavior is elicited through the specification of desired future observations in the generative model. However, in some cases, these desired future observations, also known as goal priors, are difficult to specify a priori. Thus, implementing active inference becomes a challenge in more complex problems. This problem can be alleviated through extending the notion of a goal prior to include a full probabilistic model which can infer desired future observations through a simple scheme of observing user feedback. This model that extends the physical model is also known as the target model and it is responsible of inferring beliefs about the location chosen by the user through iteratively updating its internal model parameters given the user feedback. This problem was explored extensively in the literature in [26] and in [13].

Perhaps the effects of fast design cycles can most clearly be seen in the design of the target model. In this section, four iterations on the design cycle will be introduced where each iteration is motivated by an inadequate performance. The approach taken in implementing the target model was to first design 1D models and then, using the insights gained, move on to 2D models. Models were compared to each other using specified performance metrics.

The target models were implemented using Turing.jl which is a general purpose probabilistic programming language for robust, efficient Bayesian inference and decision making [21]. The target models we designed include some nonlinear functions that may be difficult to implement tractably in a message passing based inference library like ForneyLab. Turing.jl provides a wide variety of sampling based inference methods. It should be noted that sampling based methods are generally computationally more intensive compared to message passing based methods. However, our experiments have shown that the increase in execution time does not affect the process significantly (even though the user is part of the process) and overall model performance is not effected by this increase in execution time in any way.

First Major Iteration on the Design Cycle: Target Models in 1D

In order to integrate the user into the loop, a protocol that specifies how the user interacts with the agent needs to be established such that the process is not too complicated and more importantly, such that the user does not need to understand the inner workings of the agent. This protocol is given as follows: first, the agent samples a goal position from the belief about the target position and subsequently it moves to this sampled position. Next, the user compares the new position to the previous position and provides feedback specifying whether the agent has come closer to the target or not. Upon receiving this feedback, the agent infers a posterior belief over the desired target position and the process repeats itself until the user deems that the target position has been found. In

order to infer beliefs about the target position given the user feedback, a target model is required.

In [26], a cart parking task in 1D is defined to provide a proof of concept for the notion of extending the goal prior to include a full probabilistic model and the approach was validated using a simulation. This was chosen as our starting point to gain some intuition about implementing a target model in 1D similar to the one described in [26] as follows where y_t and y_{t-1} specify the current and previous positions respectively, b_t and b_{t-1} the noisy beliefs about the positions, x^* the belief about the real target position and r_t user feedback.

$$\begin{aligned} p(r_t, b_t, b_{t-1}, x^*, \lambda | y_t, y_{t-1}) = \\ p(\lambda)p(x^*)p(b_t|y_t)p(b_{t-1}|y_{t-1})p(r_t|x^*, b_t, b_{t-1}, \lambda) \end{aligned} \tag{28}$$

$$p(b_t|y_t) = \mathcal{N}(b_t|y_t, 1) \tag{29a}$$
$$p(b_{t-1}|y_{t-1}) = \mathcal{N}(b_{t-1}|y_{t-1}, 1) \tag{29b}$$
$$p(\lambda) = \mathcal{N}(\lambda|20, 7) \tag{29c}$$
$$p(x^*) = \mathcal{N}(x^*|x_0, 50) \tag{29d}$$
$$p(r_t|x^*, b_t, b_{t-1}, \lambda) = Ber(r_t|\sigma(U(b_t, b_{t-1}, x^*, \lambda))) \tag{29e}$$

The utility function $U(b_t, b_{t-1}, x^*, \lambda)$ is given as

$$U(b_t, b_{t-1}, x^*, \lambda) = f(y_t, x^*, \lambda) - f(y_{t-1}, x^*, \lambda) \tag{30}$$

where the objective function $f(y, x^*, \lambda)$ is given as

$$f(y, x^*, \lambda) = -e^{\lambda/2}|x - x^*| \tag{31}$$

The utility function is used to compare the current position y_t to the previous position y_{t-1} given the target position x^*. The parameter λ is the precision parameter and it determines the width of the objective function. Passed through a sigmoid function σ, the utility function is used to parameterize a Bernoulli distribution from which the user feedback r_t is sampled. Notice that since the user feedback is sampled from a Bernoulli distribution, it is a binary value. In our simulations, user feedback was generated artificially by passing the result of the utility function (where the real target is supplied) through a sigmoid function.

The result of the simulation can be seen in Fig. 16. One thing to note in this figure is that x^* and the current position y_t are plotted separately. This reflects a design choice made to enable the agent to explore more and it was achieved by sampling the position of the agent for the next iteration from x^* as $y_t \sim \mathcal{N}(x^*|m_{x^*}, v_{x^*})$, the alternative choice being the current position directly being equal to the mean of x^*. Increasing exploratory behavior in this manner results in the agent converging faster to the real target. It should be noted that the variance of x^* (depicted as the blue area around x^*) decreases as the agent incorporates more and more information, and since y_t is sampled from $\mathcal{N}(m_{x^*}, v_{x^*})$, the actions become smaller as the agent becomes more certain of the real target.

The target model implementation using the Bernoulli distribution accepts binary user feedback as mentioned above. Although it is relatively simple to implement, user feedback in the form of 1's and 0's does not yield much information. Iterating once on the design cycle, a significant improvement can be achieved by using the Beta distribution instead. Beta distribution allows the user feedback to be continuous in the range (0,1).

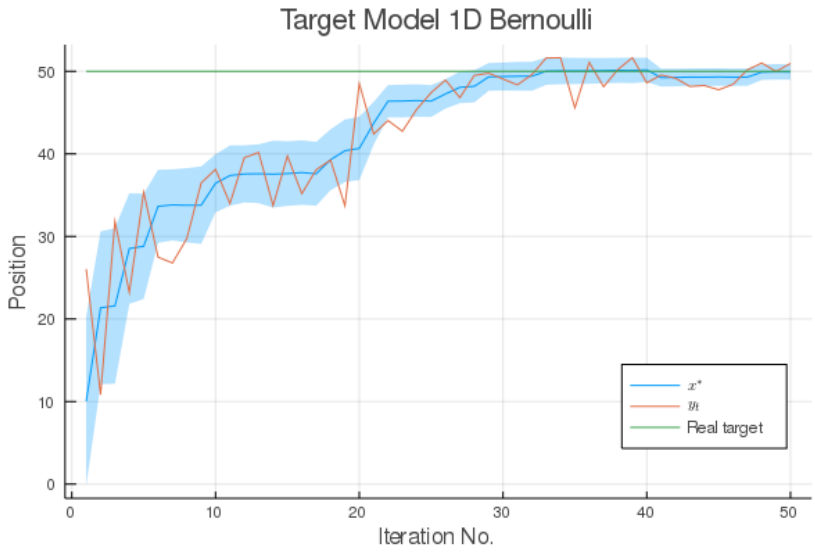


Figure 16: Simulation results of Target Model 1D Bernoulli. The agent infers the location chosen by the user on a 1D line by observing binary user feedback.

This means that the user feedback now incorporates information about the *distance* of the real target relative to the action executed in the previous cycle. Another advantage of a target model implementation using the Beta distribution is that it allows *neutral* feedback when a rating of 0.5 is provided. At first glance, it might seem like a rating of 0.5 does not yield any information, however, it actually tells the agent it has not moved closer to nor farther away from the real target. The advantage of this becomes more clear in 2D models.

The target model using the Beta distribution is similar to Eq. 29, the exception being the way the user feedback is sampled.

$$p(r_t|x^*, b_t, b_{t-1}, \lambda) = Beta(r_t|\alpha, \beta) \tag{32a}$$

$$\alpha = \tau \cdot \sigma(U(b_t, b_{t-1}, x^*, \lambda)) \tag{32b}$$

$$\beta = \tau \cdot (1 - \sigma(U(b_t, b_{t-1}, x^*, \lambda))) \tag{32c}$$

Note that the shape parameters, conventionally indicated by α and β, are multiplied by the coefficient τ. The reason for this is to decrease the variance of the Beta distribution so that the model has less uncertainty about the user feedback. This makes sense since the user feedback is assumed to be deterministic and multiplying α and β by τ can be thought of as receiving the same feedback for the same action τ times. In this case, τ was set to 50, which is a sufficiently large number. The result of the simulation for the Beta implementation can be seen in Fig. 17.

Receiving continuous user feedback as opposed to binary feedback improves the performance of the target model significantly as expected. x^* converges to the real target faster and its variance is much smaller once the target has been found.

Second Major Iteration on the Design Cycle: Target Models in 2D

Now that we have working 1D target models, target model implementations in 2D do not have to start from scratch, the respective implementations in 1D can be used as starting points. The main difference in 2D models is that multivariate distributions are

Figure 17: Simulation results of Target Model 1D Beta. The agent infers the location chosen by the user on a 1D line by observing continuous user feedback in the range (0,1).

used instead of univariate distributions, the accompanying priors become vectors and the form of the objective function is changed slightly to allow operations on matrices.

The target model implementation in 2D using the Bernoulli distribution is given as follows:

$$p(b_t|y_t) = \mathcal{N}(b_t|y_t, [10, 10]) \tag{33a}$$
$$p(b_{t-1}|y_{t-1}) = \mathcal{N}(b_{t-1}|y_{t-1}, [10, 10]) \tag{33b}$$
$$p(\lambda) = \mathcal{N}(\lambda|[2, 2], [5, 5]) \tag{33c}$$
$$p(x^*) = \mathcal{N}(x^*|x_0, [100, 100]) \tag{33d}$$
$$p(r_t|x^*, b_t, b_{t-1}, \lambda) = Ber(r_t|\sigma(U(b_t, b_{t-1}, x^*, \lambda))) \tag{33e}$$

and the objective function $f(y, x^*, \lambda)$ is given as:

$$f(y, x^*, \lambda) = -\sqrt{(y - x^*)^T e^{\lambda}(y - x^*)} \tag{34}$$

Fig. 18 shows the result of the simulation where the plot on the top shows the mean of x^* on a 2D plane as it is iteratively updated. The color of the line changes according to a specified gradient with each iteration so that x^* can be tracked more easily. The plot on the bottom shows the standard deviations of the random variables (that make up x^*) which are holding the beliefs about the real target's x and y coordinates. It can be seen that both standard deviations decrease over time, indicating that the agent becomes more certain about the real target. The standard deviation of the random variable holding the belief about the y axis is larger until iteration 20, the effects of which can be seen as larger steps taken on the y axis.

In this simulation the coordinates of the real target were specified as (15,30) and it took the agent around 65 iterations to find it. Although 65 iterations might not sound like much, once the target model is ported to the robot, each iteration is going to take a certain amount of time including the execution of the target model to produce new goal

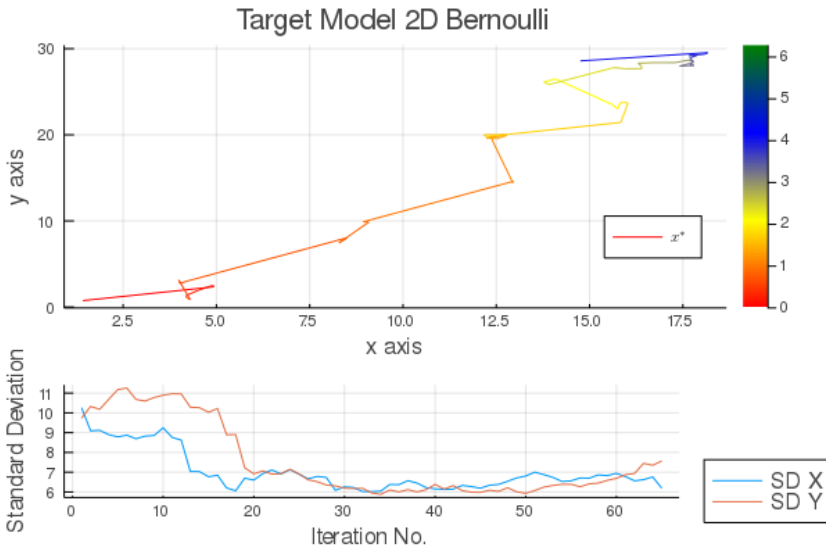


Figure 18: Simulation results of Target Model 2D Bernoulli depicting how the agent converges to the location chosen by the user on a 2D plane by observing binary user feedback.

priors (in terms of coordinates) for the physical model and the execution of the physical model that physically takes the robot from point a to b. It should also be noted that since a user will be providing the feedback, it might become tiresome to go through 65 iterations.

As in the 1D case, a significant improvement can be expected when the Beta distribution is used instead of the Bernoulli distribution. Looking at Fig. 19, it can be seen that the target is indeed found much faster. Fig. 19 also shows improved exploratory behavior which is due to the variance of x^* being initialized to a higher value compared to the Bernoulli implementation. In implementations with the Beta distribution, the variance of x^* tends to decrease faster and in some cases where the target is far from the initial position of the agent, the target is never reached due to the low variance at later iterations. One might think that for the Bernoulli implementation, initializing the variance of x^* with a higher value would also improve the performance, however, even if the mean of x^* is equal to the real target in any given iteration, since the variance is usually still high, in later iterations the mean of x^* tends to get farther from the real target. Overall, increasing the initial variance of x^* (compared to what is given here) did not provide any improvement on the performance in our experiments.

Although figures 18 and 19 are suitable for inspecting the behavior of the agent for one simulation, being a stochastic process that displays a somewhat varying behavior in each simulation, they do not provide the full means of comparing two models. Comparing two models necessitates the explicit definition of suitable metrics that define the performance of a model. An obvious metric is the Euclidean distance of the target belief to the real target over time which encaptures how fast a model converges to the real target on average. However, a model that performs well on average does not necessarily perform as well in each simulation and may display widely varying results. So another useful metric would encapture the varying behavior of the model between several simulations. The combination of these two metrics would thus give an idea about the accuracy and the precision of a model.

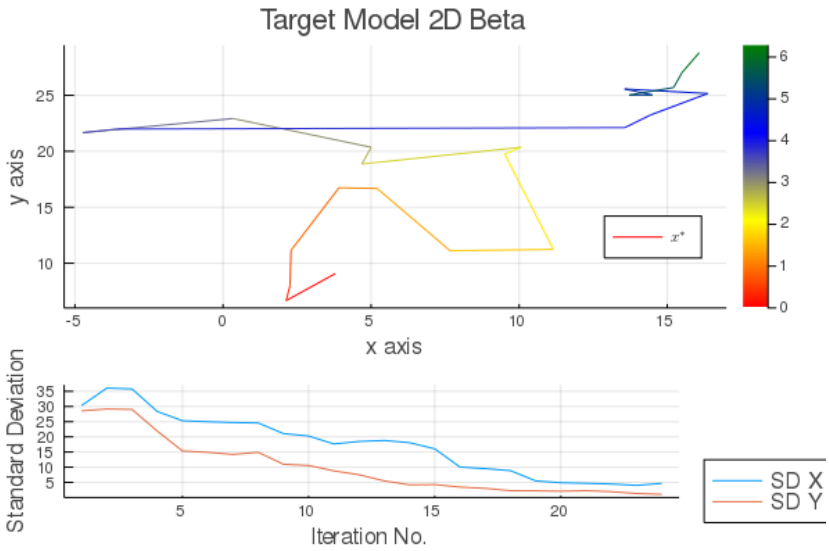


Figure 19: Simulation results of Target Model 2D Beta depicting how the agent converges to the location chosen by the user on a 2D plane by observing continuous user feedback in the range (0,1).

Focusing on the metrics stated above, a simulation protocol was designed and the algorithm is given as follows: 10 simulations are run one after the other and before each simulation, the priors are initialized to the same starting values. Each simulation consists of 50 time steps and for each time step, the Euclidean distance between the mean of x^* and the real target is calculated. At the end of the protocol, the average distance for each simulation at each time step is calculated. The real target was set to (45,30) and the initial position to (0,0).

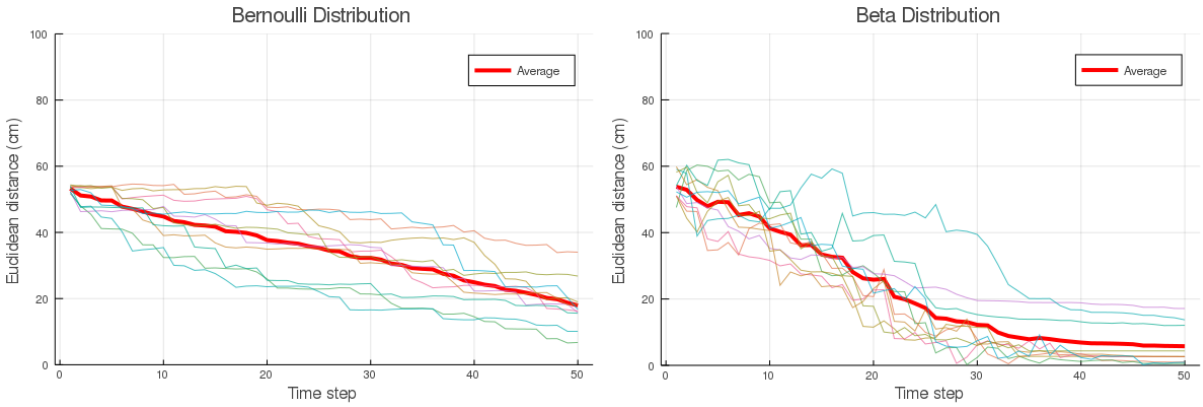


Figure 20: Bernoulli and Beta implementation benchmarks. The figures show the Euclidean distance between the location chosen by the user (i.e. the real target) and the belief about the location chosen by the user (x^*). Results of 10 simulations are given here, including their average.

Fig. 20 clearly shows that the Beta distribution implementation converges faster on average, although it can also be seen that in some simulations, the variance of x^* decreases too fast and the model does not converge to the real target. Clearly, another iteration

on the design cycle is necessary to increase the target model's precision and accuracy, which will be treated in the next subsection.

3.3.3 Selection of Priors with Thompson Sampling

All the target models explored so far use the mean of the posterior distribution of x^* as the prior for the next iteration. Although this provides a simple means of doing online learning, there's some information available that is not being utilized that can further improve the performance. This information can be utilized by choosing the prior for the mean of x^* in a smarter way.

Thompson sampling is an algorithm that is widely used in online decision making problems where actions are sequentially executed to maximize some reward function in the long run. One of the reasons it is so widely used is because of how it addresses the exploration-exploitation trade-off and it is usually explained using the multi-armed bandits problem. Simply put, in a multi-armed bandits setting there are alternative choices where their properties (e.g. a reward of 1 with probability θ_k in the Bernoulli Multi-armed bandits problem) are initially unknown or only partially known. An example of a naive approach to this problem is the Greedy algorithm [35] where for each choice, a probability distribution is initialized indicating ignorance (e.g. Beta(1,1)) and subsequently, decisions are made based on which probability distribution has the highest mean (i.e. which choice is assumed to have the highest probability of returning a reward) where ties are broken randomly. After each decision made, the corresponding probability distribution is updated depending on whether it provided a reward or not. The problem with Greedy algorithms is that they have a tendency towards exploitation for immediate gain. For example, if one of the choices' probability distribution has a higher mean than the rest, in subsequent iterations the Greedy algorithm will always choose this one even though another choice might have a higher variance and if explored, it might have a higher probability of returning a reward. In contrast, in Thompson sampling, a reward probability is *sampled* from the corresponding probability distribution and thus, choices that have been explored less so far (which have a higher variance) have a chance of being chosen. Thompson sampling has been shown to be close to optimal [1].

The Thompson sampling framework explained above, considers a discrete set of actions with accompanying probability distributions representing beliefs about the mean reward. However, in our case there are an infinite number of possible actions and implementing Thompson sampling as explained above would require constraining the agent to an explicitly specified, finite set of actions. Our experiments have shown that even though this is possible, the performance of the target model suffers significantly especially in 2D models. In order to properly harness the power of Thompson sampling in our case, a different perspective is required. Instead of constraining the set of possible actions to a finite set, a sufficiently large number of proposals can be sampled from the posterior distribution of x^* and the utility of each proposal can be calculated given the previous action. Hence, another iteration on the design cycle is introduced.

Fig. 21 shows the values of the utility function calculated for all possible x^* coordinates, given the previous action depicted as the black arrow. The real target is depicted as the red cross. Note that this figure does not depict the actual case where only a certain number of proposals are sampled from the posterior but shows all possible coordinates in the 2D plane to better visualize the information embedded in the system. In the real case the sampled prior proposals are scattered according to the posterior distribution of x^*. The prior of x^* for the next iteration can subsequently be chosen as the proposal that provides the highest utility, which will be a proposal closer to the area depicted white in this figure.

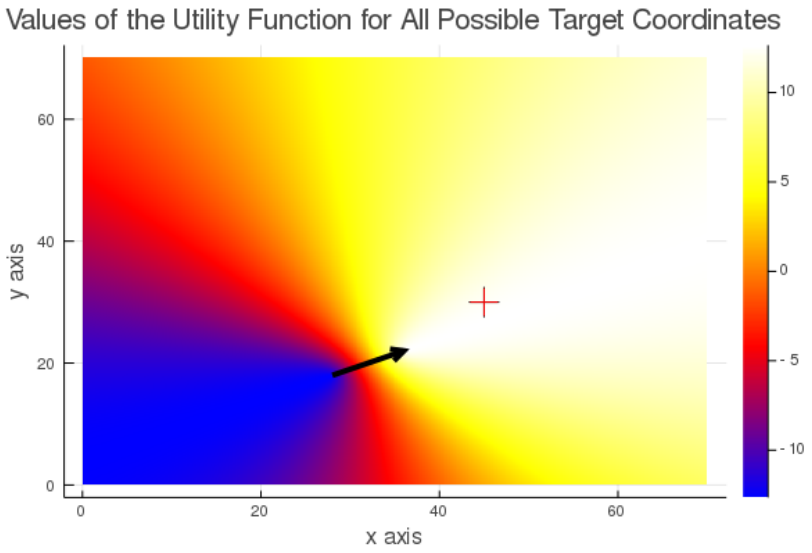


Figure 21: Values of the utility function for all possible target coordinates, one time step.

This strategy seems to be applicable only when a user feedback that is higher than 0.5 is received at first sight since an action that receives a rating less than 0.5 indicates that the agent has moved further from the real target and consequently the proposal chosen would actually be further from the target. However, to fully harness the power of this strategy so that the information provided by each action, no matter if it takes the agent further away from the target, is utilized to a higher extent, the direction of the actions that take the agent further from the target can simply be reversed when calculating the utility values. The intuition behind this comes from the fact that $\sigma(U(y_t, y_{t-1}, x^*, \lambda)) = 1 - \sigma(U(y_{t-1}, y_t, x^*, \lambda))$. In layman's terms, when an action receives a feedback that is smaller than 0.5, the action in the opposite direction would receive a feedback higher than 0.5 and could then be used to sample proposals. Thompson sampling was first implemented for the Bernoulli distribution. The results

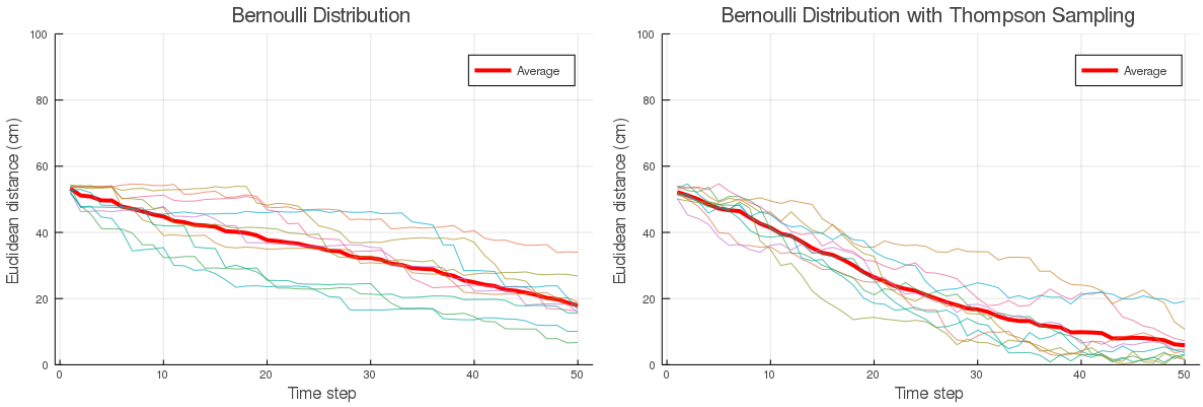


Figure 22: Bernoulli and Bernoulli with Thompson sampling comparison. Thompson sampling improves the performance of the model by selecting better priors for x^*.

can be seen in Fig. 22. For easier comparison, both the naive Bernoulli implementa-

tion and the Bernoulli implementation with Thompson sampling have been visualized. Thompson sampling clearly improves the performance of the model. Where the naive implementation fails to converge after 50 iterations, the implementation with Thompson sampling is within 10 cm of the target at around 35 iterations on average. It's also interesting to see that the Bernoulli implementation with Thompson sampling produces a result similar to the naive Beta implementation. The differences seem to be a slightly slower convergence and a slightly improved precision.

Next, Thompson sampling was implemented for Beta distribution. The results can be seen in Fig. 23. Again a clear improvement can be seen.

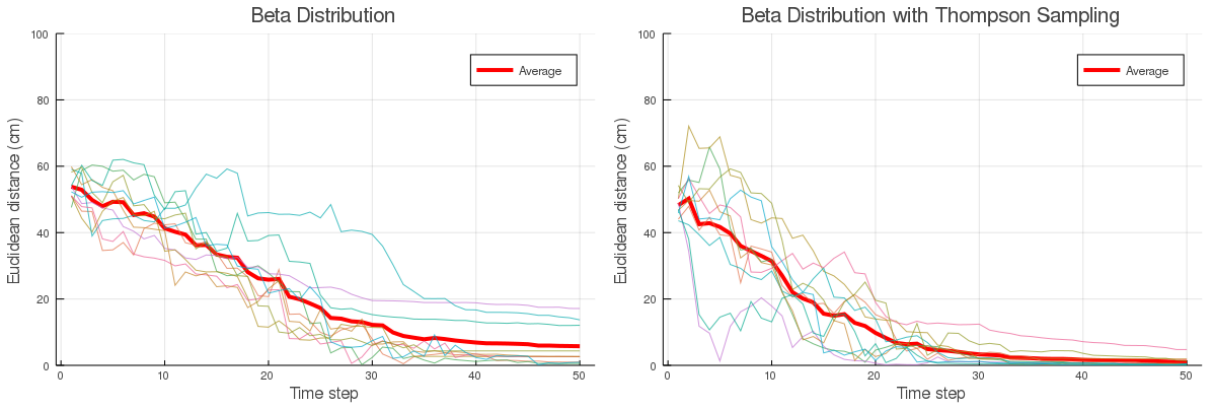


Figure 23: Beta and Beta with Thompson sampling comparison. Thompson sampling improves the performance of the model by selecting better priors for x^*.

3.3.4 Combining the Target Model and the Physical Model

Target model 2D Beta with Thompson sampling implementation comes within 5 cm radius of the real target in around 25 iterations and it also achieves high precision meaning that after iteration 30, most simulations are very close to finding the target. This performance was deemed high enough and we moved on to porting the target model to the robot. Remember that, the physical model was capable of processing manually entered goal priors (in terms of coordinates) where we left off. Now, this provides a rather simple interface between the physical model and the target model such that positions sampled from x^* (indicated by y_t in the target model) can be fed to the physical model as goal priors. Upon receiving the goal prior, the physical model is run until the robot reaches that position and subsequently, feedback is requested from the user after which the process repeats itself.

At this stage, before any experiments with the robot, no big surprises with the performance of the overall implementation was expected since each block was tested and verified. However, experiments on the robot with actual user feedback resulted in a performance worse than in the simulations. Since the physical model was performing within acceptable bounds (i.e. the robot was capable of moving to within 2 cm of the goal position proposed by the target), the problem had to lie with the user feedback because this was the only difference with the simulations. Upon closer inspection of the user feedback generated in the simulations, it was found that in the majority of cases, a feedback of either 1 or 0 was being generated and in some cases where it was difficult to tell whether the agent had moved closer to the target or not, a feedback close to 0.5 was being generated. This, although unexpected, can be attributed to the initial prior

for the precision parameter λ which was set to a low value. Initializing λ with a low value had proven to improve the performance of the model previously. So, it seems that the power of the Beta distribution comes from the ability to provide neutral feedback, although the argument for the feedback being relative to the distance to the target still holds as well. This discovery actually improves the model from the user's perspective since it's easier to provide a feedback of 1, 0 or 0.5. Repeating the experiments have confirmed this discovery, performances similar to the ones reported in the simulations were achieved with real user feedback.

4 DISCUSSION

Since the industrial revolution, there has been an accelerating tendency towards the automation of our production methods and in our age, this growing tendency corresponds to the anticipation of a fourth industrial revolution that differs from the first three in its velocity, scope and systems impact. The vision for a fourth industrial revolution includes (but is not limited to) concepts such as fully automated factories (a.k.a. dark factories), unprecedented improvements in connectivity and a greater integration of "smart" technology into our lives. In this setting, the concept of intelligent autonomous agents gains traction as an actor that may bring about this revolution.

Active inference provides a unified framework for designing intelligent agents where action and perception inherently arise from the minimization of a single cost function, namely free energy minimization. Furthermore, since generative models specify stochastic processes and encode uncertainty for action and perception, active inference enables intelligent agents to be robust and adaptable under dynamic, unpredictable conditions. In section 3.3.1, an experiment is provided where the robustness and adaptability of the agent is empirically validated. This experiment depicts how the agent adjusts its actions in order to reach a desirable observation when the environment produces adverse effects. Further evidence for active inference's robustness for a real-world application is provided in [31] where a body perception and action model is presented for a humanoid robot. Active inference can also be augmented with supporting methods in order to address the challenges designing intelligent agents in industrial settings bring forth. Although these challenges may somewhat vary from sector to sector, two challenges are faced throughout the industry.

The first of these challenges is to enable fast design cycles. Fast design cycles enable rapid proposal and critique of models and they enable fast development as well as early identification of the problem. In this paper, fast design cycles are mainly facilitated by automated generation of inference algorithms, the impact of which can be seen throughout the paper. Furthermore, since generative models offer a degree of flexibility in enforcing specific behavior, iterations on the design cycle need not only consider changes to the model architecture but can also be facilitated by other means (including but not limited to updates to the act function for example). Flexibility of models in this sense may offer feasible workarounds and further increase the speed of design cycles. The secondary level in which fast design cycles impact system development is enabled through providing a means to the end user of personalizing the agent. This level is facilitated by the augmentation of the agent with a target model. In the wearable electronics industry, algorithms are usually developed offline and an iteration on the design cycle constitutes receiving user feedback, making improvements to the algorithm and deploying the changes. However this is a long process and the changes may not appeal to each user. In this setting, the target model effectively removes the engineer from the design cycle and gives flexibility to the user to adjust the device to her liking (without

requiring the user to understand how the system works).

The second challenge in the industry concerns modularity. Modularity makes complex systems more manageable by dividing the system into modules that perform logically discrete functions interacting through well-defined interfaces. The MBML perspective encourages modular design through the specification of distinct generative models that are assigned specific functions. Figure 5 attests the modular nature of this project. For example by separating the pre-processing block from the physical model, an abstraction is created that hides implementation details related to hardware functions. While this provides ease in implementation, it also enables adaptability since each block can easily be replaced with a block that conforms to the specified interface.

In this project, implementing active inference for a real-world agent did not highlight any shortcomings of active inference that are not readily identifiable through simulations. Although the results we present in this paper are promising, the feasibility of active inference for building intelligent autonomous agents needs to be explored further. The following may be of interest to explore further:

- Implementation of active inference for a task with real-time constraints.
- Exploring sensor fusion with active inference. Currently the go-to method for implementing sensor fusion is Kalman filtering. Active inference may prove comparable and even advantageous since actions are incorporated to the model alongside perception.
- Exploring hierarchical models. Part of model design is trying to find suitable initial priors. A higher level model that iteratively proposes new priors may significantly speed up this process.

5 CONCLUSIONS

In order to assess active inference's capabilities and feasibility for a real-world application, a proof of concept was implemented showing that active inference is indeed a viable method for building real-world intelligent autonomous agents. This involved the implementation of an active inference-based parking agent constituting a physical model and a target model that ran on a ground-based robot. The following research questions were answered:

Is active inference a feasible option for the efficient design of adaptive controllers/algorithms for real-world applications?

The premise of active inference allows an intelligent autonomous agent to execute all tasks, namely online tracking of states (perception), parameters (learning) and actions (decision making) by running inference on a generative model of its environment. Generative models contribute to the robustness of intelligent agents by encoding uncertainty for action and perception. Furthermore, using a factor graph approach combined with automated generation of inference algorithms drastically accelerates the design cycles, allowing rapid proposal and critique of models.

How can active inference control be implemented for the efficient design of a real-world robot?

In this paper, an iterative design process specified by Box's Loop was followed which involves consecutive proposal and critique of models. Generative models were initially built, iteratively refined and verified in simulations and later ported to the robot. Experiments on the robot specifically focused on real-world performance, depending on which

further iterations on the design cycle were carried out. Experiments on the robot showed that in some cases, enforcing specific behavior could be facilitated through updates to functions specifying the interface between the agent and the environment, without the need to change the model architecture. It's also worth mentioning that the agent was designed in a modular manner. This modular approach provided an abstraction between functional blocks, hiding low-level implementation details.

How can users be integrated into the active inference controller design cycle to tailor for their specific needs?

A priori specification of user preferences as goal priors may be difficult. Through extending the notion of a goal prior to include a full probabilistic model which can infer desired future observations, users can be integrated into the design cycle. This allows active inference agents to be customizable and reduces the amount of knowledge required from users to be capable of interacting with an agent.

ACKNOWLEDGEMENT

We would like to thank Bert de Vries, Thijs van de Laar, Magnus Koudahl and Martin Roa Villescas for their invaluable input and Sander Stuijk for being a member of the assessment committee and providing a working space.

Bibliography

[1] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In Shie Mannor, Nathan Srebro, and Robert C. Williamson, editors, *Proceedings of the 25th Annual Conference on Learning Theory*, volume 23 of *Proceedings of Machine Learning Research*, pages 39.1–39.26, Edinburgh, Scotland, 25–27 Jun 2012. PMLR.

[2] Arduino.cc. Arduino uno rev3. `https://store.arduino.cc/arduino-uno-rev3`, 2020. Accessed: 2020-04-08.

[3] Steven Bell. High-precision robot odometry using an array of optical mice. 2011.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[5] Christopher M Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, NY, 2006.

[6] Christopher M. Bishop. Model-based machine learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371, 2013.

[7] David M. Blei. Build, compute, critique, repeat: Data analysis with latent variable models. *Annual Review of Statistics and Its Application*, 1(1):203–232, 2014.

[8] Andrea Bonarini, Matteo Matteucci, and Marcello Restelli. Dead reckoning for mobile robots using two optical mice. pages 87–94, 01 2004.

[9] Johann Borenstein, Hobart R. Everett, and Liqiang Feng. "where am i?" sensors and methods for mobile robot positioning. 1996.

[10] Christopher L. Buckley, Chang Sub Kim, Simon McGregor, and Anil K. Seth. The free energy principle for action and perception: A mathematical review. *Journal of Mathematical Psychology*, 81:55 – 79, 2017.

[11] Matteo Colombo and Cory Wright. First principles in the life sciences: the free-energy principle, organicism, and mechanism. *Synthese*, Sep 2018.

[12] Marco Cox, Thijs van de Laar, and Bert de Vries. A factor graph approach to automated design of bayesian signal processing algorithms. *CoRR*, abs/1811.03407, 2018.

[13] M.G.H. Cox and A. de Vries. A parametric approach to bayesian optimization with pairwise comparisons. In *NIPS 2017*, 2017.

[14] Stefan Edelkamp, Stefan Schroedl, and Sven Koenig. *Heuristic Search: Theory and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[15] G. D. Forney. Codes on graphs: normal realizations. *IEEE Transactions on Information Theory*, 47(2):520–548, Feb 2001.

[16] Karl Friston. The free-energy principle: A rough guide to the brain? *Trends in Cognitive Sciences*, 13(7):293–301, 2009.

[17] Karl Friston, Spyridon Samothrakis, and Read Montague. Active inference and agency: optimal control without cost functions. *Biological Cybernetics*, 106(8):523–541, 2012.

[18] Karl Friston, Philipp Schwartenbeck, Thomas Fitzgerald, Michael Moutoussis, Tim Behrens, and Raymond Dolan. The anatomy of choice: active inference and agency. *Frontiers in Human Neuroscience*, 7:598, 2013.

[19] Karl J. Friston, James Morvan Kilner, and Lee M. Harrison. A free energy principle for the brain. *Journal of Physiology-Paris*, 100:70–87, 2006.

[20] Warren Gay. *Raspberry Pi Hardware Reference*. 01 2014.

[21] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, pages 1682–1690, 2018.

[22] Parallax Inc. Robot shield with arduino. `https://www.parallax.com/product/32335`, 2020. Accessed: 2020-04-08.

[23] InvenSense. Mpu-6000 and mpu-6050 product specification revision 3.4. `https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf`, 2013. Accessed: 2020-04-08.

[24] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[25] S. Korl. A factor graph approach to signal modelling, system identification and filtering. *Series in signal and information processing*, 2005.

[26] Magnus T. Koudahl and Bert de Vries. Batman: Bayesian target modelling for active inference. *ICASSP 2020*, 2020.

[27] F. R. Kschischang, B. J. Frey, and H. . Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, Feb 2001.

[28] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2):157–224, 1988.

[29] G.W. Lucas. A tutorial and elementary trajectory model for the differential steering system of robot wheel actuators. `http://rossum.sourceforge.net/papers/DiffSteer/`, 2001. Accessed: 2010-09-30.

[30] Dimitrios Lymberopoulos, Jie Liu, Xue Yang, Romit Choudhury, Vlado Handziski, Souvik Sen, Filip Lemic, Jasper Buesch, Zhiping Jiang, Han Zou, Hao Jiang, Chi Zhang, Ashwin Ashok, Chenren Xu, Patrick Lazik, Niranjini Rajagopal, Anthony Rowe, Avik Ghose, Nasim Ahmed, and Peter Hevesi. A realistic evaluation and comparison of indoor location technologies: Experiences and lessons learned. 04 2015.

[31] Guillermo Oliver, Pablo Lanillos, and Gordon Cheng. Active inference body perception and action for humanoid robots, 2019.

[32] Parallax Inc. *Parallax Feedback 360° High-Speed Servo*, 7 2017.

[33] S.-H Park and Soo-Yeong Yi. Mobile robot localization using optical mouse sensor and encoder based on kalman filter algorithm. *International Journal of Control and Automation*, 10:61–70, 06 2017.

[34] Marvelmind robotics. Precise indoor 'gps', starter set hw v4.9-nia. `https://marvelmind.com/product/starter-set-hw-v4-9/`, 2020. Accessed: 2020-03-18.

[35] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A tutorial on thompson sampling, 2017.

[36] Thijs van de Laar. *Automated design of Bayesian signal processing algorithms*. PhD thesis, Technische Universiteit Eindhoven, Department of Electrical Engineering, 6 2019. Proefschrift.

[37] Thijs W. van de Laar and Bert de Vries. Simulating active inference processes by message passing. *Frontiers in Robotics and AI*, 6:20, 2019.