Article

Not peer-reviewed version

Introducing ActiveInference.jl: a Julia Library for Simulation and Parameter Estimation with Active Inference Models

Samuel William Nehrer , Jonathan Ehrenreich Laursen , Conor Heins , Karl Friston , Christoph Mathys , Peter Thestrup Waade *

Posted Date: 26 November 2024

doi: 10.20944/preprints202411.1880.v1

Keywords: Active Inference; Free Energy Principle; Predictive Processing; Markov Decision Process; Cognitive Modeling; Julia

Article

Introducing `ActiveInference.jl`: A Julia Library for Simulation and Parameter Estimation with Active Inference Models

Samuel William Nehrer[1,†], Jonathan Ehrenreich Laursen[1,†], Conor Heins[2,3], Karl Friston[3,4], Christoph Mathys[5] and Peter Thestrup Waade [5*]

[1] School of Culture and Communication, Aarhus University
[2] Department of Collective Behaviour, Max Planck Institute of Animal Behavior, Konstanz D-78457, Germany
[3] VERSES Research Lab, Los Angeles, CA 90016, USA
[4] Queen Sq, Institute of Neurology, University College London
[5] Interacting Minds Centre, Aarhus University
[†] Equal contribution.
[*] Correspondence: ptw@cas.au.dk

Abstract: We introduce a new software package for the Julia programming language, the library *ActiveInference.jl*. To make active inference agents with Partially-Observable Markov Decision Process (POMDP) generative models available to the growing research community using Julia, we re-implemented the *pymdp* library for Python. ActiveInference.jl is compatible with cutting-edge Julia libraries designed for cognitive and behavioral modeling as it is used in computational psychiatry, cognitive science, and neuroscience. This means that POMDP active inference models can now easily be fit to empirically observed behavior, using sampling as well as variational methods. In this article, we show how ActiveInference.jl makes building POMDP active inference models straightforward, and how it enables researchers to use them for simulation as well as fitting them to data or to do model comparison.

Keywords: active inference; free energy principle; predictive processing; markov decision process; cognitive modeling; julia

1. Introduction

We introduce a novel software library for Julia, `ActiveInference`, which lets users produce the simulated behaviour of agents and their internal belief states with active inference (AIF) models, as well as to fit such models to empirically observed behaviour. AIF [1–3] is a generally applicable formal framework for understanding and simulating intelligent behaviour, based in neurobiology and first principles from statistical physics [4–8]. AIF treats action and perception as unified under a joint imperative: to minimize the variational free energy (*VFE*), an upper bound on the surprise of sensory observations, under a generative model of the environment. This is closely related to prediction error minimization [9]. Choosing actions that minimise the expected free energy (*EFE*) of their consequences provides a natural balance between exploratory and exploitative behaviour, generalises descriptive approaches to behavioural modelling, like reinforcement learning and expected utility maximisation, and provides a singular approach to adaptive behaviour that can be used across different environments. AIF is argued to be applicable to any self-organising system that actively maintains a stable boundary that defines its integrity [10], a broad category that includes cells and plants [11] as well as humans [2] and even collectives [12]. Owing to its generality, AIF has seen a rise in popularity across multiple fields. It is used for theoretical simulations of the mechanisms underlying various types of behaviour [2], computational phenotyping in computational psychiatry [13,14], and agent-based simulations of population dynamics [15], as well as in engineering and robotics [16]. In AIF, perception and concurrent action is based on performing a variational Bayesian inversion of a generative model of the environment (i.e., a model of how the environment changes and brings about sensory observations). This belief updating includes inferring (hidden) states of the environment, learning parameters of the generative model, as well as learning the structure of the generative model.

Since the requisite inference schemes come pre-specified, the main task in AIF modelling becomes to specify an appropriate generative model. This includes specifying priors over environmental states, as well as what might be called *prior preferences*, *preference priors* or *goal priors*: immutable prior expectations that make up an agents' preferences by furnishing a set of predictions over future states or observations; in fulfilling these predictions, free energy is minimised. The space of possible generative models is vast, and they often have to be handcrafted for a given environment. However, there are some families of generative models that can be considered "universal", in the sense that they can be used for most environments. Currently, the most popular of these is the discrete state-space Partially Observable Markov Decision Process (POMDP) based generative models. Since they are ubiquitous in the literature, we have here focused on making these types of generative models available to researchers. There are, however, other types of universal generative models, like generalised filtering models [17] or Hierarchical Gaussian Filtering-based models [18,19], that will be implemented in the future.

Tools for simulating POMDP-AIF models were originally developed as part of the DEM [20] library for MATLAB [21] (part of the larger SPM library [22]). Since then a modal and flexible software package `pymdp` [23] has been created for Python [24], as well as a performance-oriented package `cpp-AIF` [25] for C [26] which can be used across platforms. Finally, the factor graph library `RxInfer` [27] for Julia [28] has also been used to implement some AIF models on an efficient factor graph back-end [29–31]. The important tools that these packages provide make AIF available for researchers to perform simulation studies and for use in engineering contexts. They do not, however, usually allow for fitting models to empirically observed data, which is a fundamental method used in cognitive modelling [32], often in the context of computational psychiatry [13], to infer mechanisms underlying variations in behaviour, or to investigate differences between (for example clinical) populations. Smith and colleagues [33] provide a guide for manually doing variational Bayesian parameter estimation based on empirical data, but only in MATLAB, and restricted to a particular class of variational parameter estimation methods (variational Laplace) instead of the sampling based methods that currently predominate in the field of cognitive modelling [34,35].

In this paper, we introduce `ActiveInference.jl`, a new software library for Julia [28] that aims to provide easy-to-use tools for model fitting with AIF models and to introduce AIF to the growing community of researchers using Julia for computational psychiatry and cognitive modelling. Julia is a free and open-source high-level programming language, which retains an easy user interface reminiscent of that in MATLAB and Python. Simultaneously, Julia uses its "just-in-time" (JIT) compilations via the LLVM framework to approach the speed of languages like C without relying on external compilers [36]. Julia is also natively auto-differentiable, which means it can solve what is called the two-language problem (i.e., that high level languages often have to rely on lower-level languages, either for performance or for auto-differentiability. This is the case with standard tools for cognitive modelling, where languages like R [37] must rely on external languages like STAN [38] for Bayesian model fitting). This means that `ActiveInference`, in conjunction with `Turing` [39], Julia's powerful library for Bayesian model fitting, and its newly developed extension for behavioural modelling, `ActionModels`, makes it possible to use cutting-edge Markov-Chain Monte Carlo [40] methods as well as variational methods [35] for Bayesian model fitting with AIF. Crucially, this allows researchers to not only simulate AIF in a fast programming language, but to also fit them to empirical behaviour, as is done in cognitive modelling and computational psychiatry. Importantly, this also places AIF models in an ecosystem of other models for computational psychiatry, so that it can easily be compared to models like Hierarchical Gaussian Filters [41] and reinforcement learning models like the classic Rescorla-Wagner model [42]. As part of making `ActiveInference.jl` available to the scientific community, and to the larger software ecosystem within computational psychiatry, it is implemented as part of the Translational Algorithms for Psychiatry-Advancing Science (TAPAS) ecosystem [43].

In the next section, we provide a conceptual and formal introduction to AIF, particularly in the context of using POMDP generative models. In section 3, we demonstrate how to use the package in practice, both for simulation and parameter estimation. In section 4, we give a full worked example of how `ActiveInference` can be used with a concrete simulated dataset. Finally, we discuss potential applications and future directions for developing the package.

2. Active Inference with POMDP's

In this section we will briefly describe the core concepts of AIF and POMDPs. This should familiarise the reader with the vernacular used in the later sections, regarding the functionalities of the package. While various extensions such as structure learning, which enables an agent to learn the structure or shape of its environment through model comparison, [44–47], or hierarchical and temporally deep POMDPs [48,49], are relevant for future work, describing these in detail is beyond the scope of this foundational paper.

At the core of AIF lies a minimisation of a variational free energy upper bound on surprise, for perception as well as action. This is motivated by the Free Energy Principle, [4–8], which states that self-organising systems can be described as minimising the variational free energy of their sensory states. The minimisation of free energy generally takes two quantities as its target: the variational free energy (*VFE*) in the case of perception and the expected free energy (*EFE*) in the case of action.

2.1. POMDPs in Active Inference

In AIF, the Partially Observable Markov Decision Process (POMDP) is one of the most common families of generative models used to do inference about the environment. It is a Markovian discrete state space model - employing it means representing the environment and observations as inhabiting one among a set of possible (possibly multidimensional) states, and that the changes of these states only can depend on the system's previous state and the agent's actions. Environmental states are not directly observable, so they have to be inferred based on incoming sensory observations. In AIF, for POMDP's and other generative models in general, both perception and action are cast as Bayesian inference (see sections 2.2 and 2.3), as well as the learning of parameters of the generative model (see section 2.4). Crucially, an agent's generative model does not *a priori* have to be isomorphic to the true environment (i.e., the data-generating process), although this will generally lead to successful inference, and that the generative model therefore often will come to resemble the environment through learning.

A discrete state-space POMDP in AIF is conventionally defined by five main sets of parameters: **A**, **B**, **C**, **D**, and **E**, [1,33]. Together, these parametrise the agent's prior beliefs about the prior probability of different states in the environment, how states of the environment changes and how they generate observations. Typically, they will be vectors, matrices, or tensors, however, henceforth we will denote them by their corresponding letter in bold. These make up the components needed for the agent to perform AIF.

A, also called the *observation model*, represents the state-to-observation likelihood model. This describes how observations depend on or are generated by states of the environment. It is structured as a matrix with a column for each possible environmental state s, and a row for each possible observation o. Each column is then a categorical probability distribution over the observations that will occur given the environmental state (meaning that each column must contain non-negative values that sum to 1). If the observations are multidimensional (i.e., multiple observations are made at each time point), there is a matrix for each observation modality. If two or more states determine the observation, the likelihood model then becomes a tensor. If **A** is imprecise (i.e., the probabilities are highly entropic and evenly distributed), observations are taken to carry less information about the environment, in many cases leading to more uncertain inference - and *vice versa*.

B, also called the *transition model*, describes the state-to-state transition probabilities of environmental states s. **B** encodes the agent's assumptions about how the environment changes over time, depending on its actions. It has a column and a row for each environmental state s, where each column is a categorical probability distribution over the states the environment will take on the next timestep, given the state it is currently in. If the environment is modelled as multidimensional, there will be a matrix for each environmental state factor. Additionally, there is a separate matrix for each possible action (making each factor in **B** a tensor). This means that for every factor in the model there may be one or more actions that picks out the appropriate slice of the tensor. Action therefore allows the agent to predict that the environment (and the corresponding observations) will change differently depending on the actions that it chooses. If **B** is imprecise (i.e. highly entropic), it means that the transitions of the environment are expected to be uncertain (and therefore often transition to new states). In this sense, volatile and unstable environments will lead to less certain predictions about the future.

C, also called the *preference prior*, is a prior preference over possible observations. It encodes the types of observations that an agent *a priori* expects to encounter; since minimising expected free energy through AIF entails taking actions that make the predicted observations come about, **C** also encodes the agent's preferences. It is a single categorical probability distribution over possible observations; if the observations are multidimensional, there is a separate *preference prior* for each observation modality. If **C** is imprecise (i.e., highly entropic), it's preferences are weak, and it will prioritise collecting information over realising it's preferences; if it has low entropy, the agent will have stronger preferences and instead prioritise preferred outcomes or goals.

D, also called the *state prior*, is the agent's prior belief about the states of the environment. It specifies the agent's belief about the environmental state before receiving any observations. There is a separate *state prior* over environmental states for each factor. With a more precise **A**, the influence of the **D** quickly diminishes, since the likelihood overwhelms the prior in the Bayesian inference.

E, also called the *habit prior*, is the prior over policies or paths. In the AIF vernacular, policies are allowable sequences of actions, with some specified policy length or temporal depth. The **E** encodes the agent's preferences for choosing certain policies in the absence of plans based upon *expected free energy* - sometimes called the agent's 'habits'. It is a single probability distribution over each possible policy.

In addition to the five matrices, there are several hyper-parameters that are not part of the generative model, but are part of the inference algorithm. Here we include two of the most common: the γ and α (inverse) temperature parameters. γ — the precision over policies — is the inverse temperature of a softmax transformation of expected free energies over policies, which will be covered later in this section. After policies have been selected for a given time-step, they are marginalised to calculate the probabilities for taking each possible action on the next time-step. α — the action precision — is the inverse temperature of a softmax transformation on these final action probabilities, with higher values resulting in more stochastic action selection.

As noted, we here focus specifically on the POMDP-based generative models often used in the AIF literature. The basic steps when performing AIF — perception, action and learning — remains the same, however, across generative models. In the remainder of this section, we will describe each of these three steps in turn.

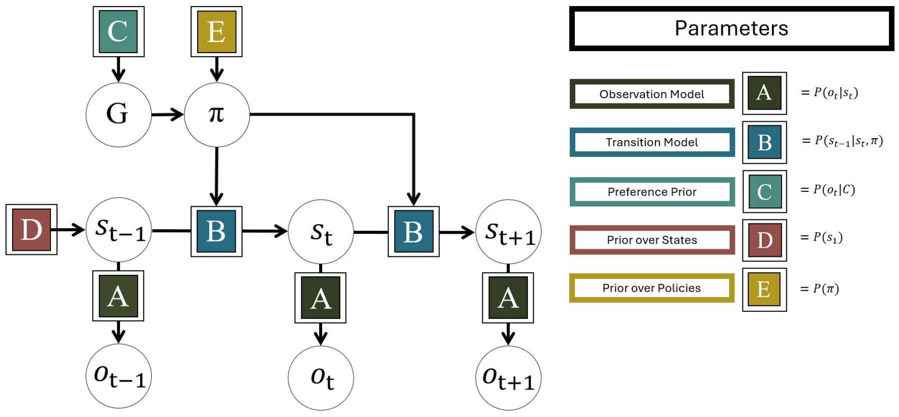


Figure 1. Depiction of a POMDP generative model. This encodes the agent's expectations about how the states s of the environment changes over time t, and how it generates observations o at each time-step. **A**, also called the observation model, describes how environmental states give rise to observations. **B**, also called the transition model, describes how environmental states change over time, depending on actions u (called policies π when structured into sequences). **C** is the preference prior, which encodes the agent's preferences for observations. This shapes the expected free energy G associated with each policy, which is used for policy selection. **D** encodes the agent's prior belief over environmental states, before making any observations, and **E** is the prior over policies that determines the agent's preferences for policies in the absence of other motivation.

2.2. Perception in Active Inference

In AIF, perception is conceptualised as the result of variational (i.e., approximate) Bayesian inference, performed by minimising the *VFE* to optimise parameters of posterior beliefs about the environment. In exact Bayesian inference, we use a parametrised generative model m to make optimal inference about states s of the environment, based on an observation o. This is done by combining a prior belief over states $p(s|m)$, a likelihood model $p(o|s,m)$, and the model evidence $p(o|m)$, a normalisation term encoding the likelihood of receiving the given observations across all possible environmental states, as follows [1]:

$$p(s|o,m) = \frac{p(o|s,m)p(s|m)}{p(o|m)} \tag{1}$$

The posterior distribution over states given observations $p(s|o,m)$ here represent the agent's beliefs about the environment. Forming beliefs in this way is thought to be the process that enables conscious as well as unconscious perception. The product of the likelihood model and prior is also called the joint likelihood $p(o,s|m)$, which fully defines the generative model, and which we will use henceforth. We will also in the following for notational simplicity omit denoting the dependency on the generative model m.

Calculating the model evidence $p(o)$ is often intractable, making exact Bayesian inference unfeasible. The way to circumvent this in AIF is to use a variational approximation to Bayesian inference [23,33,50,51]. This works by transforming the inference into an optimisation problem, specifically a minimisation of the *VFE*. First, an arbitrary probability distribution over environmental states $q(s)$ is introduced, an approximate posterior which will be used to approximate the exact posterior. We then introduce the Kullback-Leibler (KL) divergence between the approximate posterior $q(s)$ and the exact posterior, which is also sometimes called the perceptual divergence:

$$D_{\text{KL}}[q(s)\|p(s|o)] = \sum_s q(s) \ln \frac{q(s)}{p(s|o)} \tag{2}$$

It is a property of the KL divergence that the two distributions are identical when $D_{\mathrm{KL}}[q(s) \| p(s|o)] = 0$. Minimising this divergence then corresponds to approximating the exact posterior $p(s|o)$ with $q(s)$. We cannot evaluate this divergence directly since the exact posterior is still unknown. We therefore replace the expression of the exact posterior with the right hand side of Equation 1. Note that we here use the joint likelihood $p(o,s)$ notation, use the fraction rule $\frac{a}{\frac{b}{c}} = \frac{a}{b} * c$ and logarithmic rule $\ln(a * b) = \ln a + \ln b$:

$$\sum_s q(s) \ln \frac{q(s)}{\frac{p(o,s)}{p(o)}} = \sum_s q(s) \ln \frac{q(s)}{p(o,s)} + \ln p(o) \tag{3}$$

We can now rewrite the first term of the right hand side as the KL divergence of the approximate posterior from the joint likelihood, which is equal to the expression used in Equation 2:

$$D_{\mathrm{KL}}[q(s) \| p(s|o)] = D_{\mathrm{KL}}[q(s) \| p(o,s)] + \ln p(o) \tag{4}$$

We now define the *VFE* ($\mathcal{F}[q(s), o]$) as the KL divergence of the approximate posterior from the joint likelihood. The VFE is only a function of $q(s)$ and o (and the generative model m), and we can therefore calculate it without knowing the model evidence $p(o)$:

$$\mathcal{F} \triangleq D_{\mathrm{KL}}[q(s) \| p(o,s)] = \sum_s q(s) \ln \frac{q(s)}{p(o,s)} \tag{5}$$

The probability-weighted sum can be rewritten as an expectation, and the joint likelihood can be decomposed into a prior and a likelihood:

$$\mathcal{F} \triangleq \mathbb{E}_{q(s)}\left[\ln \frac{q(s)}{p(o,s)}\right] = \mathbb{E}_{q(s)}[\ln q(s) - \ln p(o|s) - \ln p(s)] \tag{6}$$

We can now combine our definition of *VFE* with Equation 4:

$$D_{\mathrm{KL}}[q(s) \| p(s|o)] = \mathcal{F}[q(s), o] + \ln p(o) \tag{7}$$

Finally, we can reorganize the equation, to show that the VFE is the sum of the divergence of the approximation posterior and the exact posterior (if we could do exact inference, this is what we would get), and the surprise $\Im = -\ln(p(o))$ (the negative log model evidence):

$$\mathcal{F}[q(s), o] = \underbrace{D_{\mathrm{KL}}[q(s) \| p(s|o)]}_{\text{Divergence}} \underbrace{- \ln p(o)}_{\text{Surprise}} \tag{8}$$

Since the KL divergence is non-negative, the *VFE* becomes an upper bound on the surprise:

$$\mathcal{F}[q(s), o] \geq -\ln p(o) \tag{9}$$

By rearranging parts of this expression, we can express the *VFE* as a balance between complexity and accuracy, where accuracy is how well the model predicts observation, and complexity is how much the beliefs need to change in order to maintain high accuracy:

$$\mathcal{F}[q(s), o] = \underbrace{D_{\mathrm{KL}}[q(s) \| p(s)]}_{\text{Complexity}} - \underbrace{\mathbb{E}_{q(s)}[\ln p(o|s)]}_{\text{Accuracy}} \tag{10}$$

Since the *VFE* can be calculated (equation 5), it can be used as a target for minimisation. This allows us to choose the approximate posterior $q(s)$ that is associated with the smallest *VFE*; since the surprise does not depend on $q(s)$, minimising the *VFE* this way must necessarily reduce the divergence between the approximate and exact posterior. We now, as is usually done in variational inference, introduce a mean-field approximation [35], so that the joint approximate posterior $q(s_t)$ factorizes across time-steps $t \in T$ and hidden state factors $f \in F$:

$$q(s) = q(s_1, s_2, \ldots, s_T) = \prod_{t=1}^{T} q(s_t) \tag{11}$$

$$q(s_t) = q(s_t^1, s_t^2, \ldots, s_t^F) = \prod_{f=1}^{F} q(s_t^f) \tag{12}$$

The factorization into hidden state factors allows us to calculate the VFE separately for each factor f and sum them to get the total VFE. The factorization in time allows us to calculate the time-specific VFE as in equation 6, using the predictive posterior $\ln p(s_t \mid s_{t-1}, u_{t-1})$ from the last time-step as prior:

$$\mathcal{F}_t = \mathbb{E}_{q(s_t)}[\ln q(s_t) - \ln p(o_t \mid s_t) - \ln p(s_t \mid s_{t-1}, u_{t-1})] \tag{13}$$

Which is the value we intend to minimise. Various methods exist for minimising the *VFE*; the one used in `pymdp`, from which we draw much of our inspiration, is Coordinate Ascent Variational Inference (CAVI) [35], where the fixed points of the *VFE* is solved for with coordinate descent (also known as fixed-point iteration (FPI) [23]). This is also the algorithm currently available in `ActiveInference.jl`. We correspondingly use a coordinate descent update to find the factorised approximate posterior $q(s_t^f)$ that minimise the time-dependent VFE $\mathcal{F}_t$, and therefore optimise for the time-specific variational posterior $q(s_t)$. To get the coordinate descent update, we start by taking the derivative of $\mathcal{F}_t$ with respect to $q(s_t^f)$ and setting the derivative to zero [23]:

$$\frac{\partial \mathcal{F}_t}{\partial q(s_t^f)} = \ln q(s_t^f) + 1 - \mathbb{E}_{q^{i \setminus f}}[\ln P(o_t \mid s_t)] - \ln\left(\mathbb{E}_{P(s_{t-1}^f, u_{t-1}^f)}\left[P(s_t^f \mid s_{t-1}^f, u_{t-1}^f)\right]\right) = 0 \tag{14}$$

Solving for $q(s_t^f)$ yields:

$$\ln q(s_t^f) = \mathbb{E}_{q^{i \setminus f}}[\ln P(o_t \mid s_t)] + \ln\left(\mathbb{E}_{P(s_{t-1}^f, u_{t-1}^f)}\left[P(s_t^f \mid s_{t-1}^f, u_{t-1}^f)\right]\right) - 1 \tag{15}$$

which leads us to the coordinate descent update equation:

$$q^*(s_t^f) = \sigma\left(\mathbb{E}_{q^{i \setminus f}}[\ln P(o_t \mid s_t)] + \ln\left(\mathbb{E}_{P(s_{t-1}^f, u_{t-1}^f)}\left[P(s_t^f \mid s_{t-1}^f, u_{t-1}^f)\right]\right)\right) \tag{16}$$

Where $\mathbb{E}_{q^{i \setminus f}}$ denotes the expectation over $q(s)$ for factor i, where the posterior over states in the other factors f are kept constant. By iteratively solving 16 the FPI scheme will eventually find a local optimum and converge to a solution for the variational posterior. `ActiveInference` by default uses 10 iterations, or until $\partial F_t < 0.001$. This posterior then comprises the AIF agent's belief about the state of the environment - and therefore it's perception.

2.3. *Action in Active Inference*

As with perception, action in AIF is guided by the minimisation of free energy. However, instead of *VFE* being minimized directly, it is the free energy that is expected to occur depending on the actions taken by the agent — the expected free energy or *EFE* — that is minimized. As we will see, choosing actions that minimize the *EFE* leads to a natural balance between exploration and exploitation, ensuring preferences are realized and ambiguity about the environment is minimized. In AIF, policies π are sequences of actions u. The policy length (also called planning horizon or temporal depth) is the length of the policies being considered. The total number of policies therefore depends on the policy length and the number of different actions that can be made at each time-step. An *EFE* is assigned to each policy π (denoted as G_π) - policies associated with a lower *EFE* are then more likely to be chosen.

One can rewrite the *EFE* in different ways to highlight different consequences of optimising it. Below, we show the two most crucial ways to rewrite it, taken from [1,33]. We denote state and observations that are expected future outcomes of actions with (~). Additionally, we introduce a *preference prior* **C** which encodes the agent's preferences:

$$G_\pi = -\underbrace{\mathbb{E}_{q(\tilde{o},\tilde{s}|\pi)}[\ln q(\tilde{s}|\tilde{o},\pi) - \ln q(\tilde{s}|\pi)]}_{\text{Information gain}} - \underbrace{\mathbb{E}_{q(\tilde{o}|\pi)}[\ln p(\tilde{o}|C)]}_{\text{Pragmatic value}} \tag{17}$$

The expression above shows how minimising the *EFE* will lead to a natural balance between information gathering and realising preferences. The first term on the right hand side is the change in belief from prior to posterior under a given policy, called the epistemic value or information gain. Optimising this value is what leads to (notably non-random) exploratory behaviour. The second term is the pragmatic value; minimising this value will ensure that observations are in accordance with the preference prior **C**.

Another way to express the *EFE* is in terms of risk and ambiguity:

$$G_\pi = \underbrace{\mathbb{E}_{q(\tilde{s}|\pi)}[H(p(\tilde{o}|\tilde{s}))]}_{\text{Expected ambiguity}} + \underbrace{D_{KL}[q(\tilde{o}|\pi)\|p(\tilde{o}|C)]}_{\text{Risk (outcomes)}} \tag{18}$$

Here, the first term on the right hand side captures the expected entropy, or uncertainty, of the outcomes given environmental states. Minimising this quantity ensures that the agent will seek states where observations can most clearly be used to distinguish between environmental states. The second term is is the KL divergence of the expected observations from preferred observations, capturing the risk of making unwanted (i.e., *a priori* surprising) observations, which is also minimised by minimising the *EFE*.

2.4. Learning in Active Inference

In AIF, the parameters of the generative model can also be updated via Bayesian belief updating methods - a process called 'parameter learning' or sometimes just 'learning' [2]. In general, this is done by introducing belief distributions over the possible values of the parameters that are subject to learning, and updating this distribution for each observation using Bayesian belief updating. This additionally implies introducing priors on the belief distributions. Depending on the type of generative model used, the belief distributions and their priors will take different forms, and so will their update equations. In the following, we demonstrate parameter learning specifically in the context of POMDP's.

The parameters that are subject to learning in POMDP's are usually the entries in the five matrices. Since the matrices consist of categorical probability distributions, it is natural to use Dirichlet distributions — distributions over categorical probability distributions — as belief distributions over their values [33,52]. Beliefs about each probability distribution Θ is then described by a Dirichlet distribution parametrized by a set of concentration parameters θ.

$$p(\Theta) = Dir(\Theta|\theta) \tag{19}$$

The concentration parameter of a Dirichlet distribution is essentially a non-negative count of how many times the given category (be it a type of observation or state transition) has occurred. The distribution of concentration parameter counts will determine the shape of the estimated categorical probability distribution, while the scale of the concentration parameters will determine the certainty per precision of the belief. Updating beliefs about Θ (the parameters in the matrices) then corresponds to updating these concentration parameters θ, with the following update equation:

$$\theta_{t+1} = \omega * \theta_t + \eta * \chi_t \tag{20}$$

The updated value for the concentration parameter (θ_{t+1}) is found by adding the previous concentration parameter θ_t, multiplied with a forgetting rate ω, to the observed data count χ (either the observation in the case of **A**learning, or the inferred state or state transition for other matrices) multiplied with a learning rate η. With this relatively simple update equation - which in essence amounts to just counting occurrences of categories - an AIF agent can update its beliefs about the various matrices it uses to make inferences about environmental states. For more details on parameter learning with POMDP's, see [23,33,52].

3. Using `ActiveInference.jl`

In this section we provide an overview of the various functions a user will need to operate `ActiveInference`. This includes functionalities for creating POMDP agents, for simulating behaviour, and for fitting the models to data. In the next section, we demonstrate how to use the package on a concrete worked example. `ActiveInference` is under continual development, and the newest version of the package, including documentation for how to use it, can be found at github.com/ilabcode/ActiveInference.jl.

3.1. Creating and using a POMDP

The general structure of `ActiveInference.jl` is heavily inspired by the Python library `pymdp` [23]. Those already acquainted with `pymdp` should find the syntax here familiar. `ActiveInference` can be installed as normal from the official Julia General Registry using the Julia's native package manager Pkg:

```
using Pkg
Pkg.add(ActiveInference)
```

It can then be loaded into the current project environment:

```
using ActiveInference
```

Central to the package is the is the AIF object. This is a structure containing all the components of the generative model, as well as the dynamic belief states and the various settings needed to perform AIF, and is used in conjunction with most of the high level functions of the package. An AIF object can be created with the `init_aif` function, which takes as arguments the components of the generative model and a dictionary of various settings and parameters.

```
# Create AIF object
aif = init_aif(
         A::Array{Any},  # A-matrices
         B::Array{Any};  # B-matrices
         C::Array{Any},  # C-matrices (optional)
         D::Array{Any},  # D-matrices (optional)
         E::Vector{Any}, # E-vector   (optional)
         pA::Array{Any}, # Dirichlet priors for A-matrices (optional)
         pB::Array{Any}, # Dirichlet priors for B-matrices (optional)
         pD::Array{Any}, # Dirichlet priors for D-vectors  (optional)
         parameters::Dict{String, Real}, # Dictionary containing other parameters (optional)
         settings::Dict{String, Any}     # Dictionary containing settings (optional)
         )
```

A and **B** are the only mandatory arguments to the `init_aif` function - the other arguments are keyword arguments that defaults to uniform priors. **A**, **B**, **C**, **D** and **E** and their corresponding Dirichlet priors, in the case of **A**, **B**, and **D** should be formatted as standard `Array` objects. All but **E** can have multiple modalities/factors (see section 4), so they should be formatted as vectors of Arrays with one

Array per modality/factor. These Arrays can be hand-specified by the user, or be generated with some of the helper functions supplied by `ActiveInference`. Here, we create an AIF agent equipped with a generative model with 6 environmental states, 5 possible observations, and 2 possible actions. We here use helper functions to create matrices and vectors with the correct dimensions; in section 4 we create them manually. First, we define the number of states, observations, controls, and the length of policies:

```
# Information about number of states, observations, actions and policy length
states        = [6] # Six states, single factor
observations = [5] # Five observations, single modality
controls      = [2] # Two actions, single factor
policy_length = 1  # Length of policies

# Generate uniform templates for matrices and vectors of the generative model
A, B, C, D, E = create_matrix_templates(states, observations, controls, policy_length)
```

The **A** object generated here is a one-dimensional vector containing a uniform 5x6 Matrix (6 states and 5 observations). The **B** object is a one-dimensional vector containing a uniform 6x6x2 Array (6 states and 2 actions). The **C**, **D**, and **E** objects are one-dimensional vectors each containing uniform vectors with their corresponding sizes. We can now modify these to supply the agent with more informative priors over observations, initial states, and policies. Here, we do this using the `onehot` function:

```
# We make C take the following form: [0, 0, 0, 0, 1]
C[1] = onehot(5,5)  # Initialize the single element of the C object with a one-hot vector

# D will be: [1, 0, 0, 0, 0, 0]
D[1] = onehot(1,6)  # Initialize the single element of the D object with a one-hot vector

# To make the agent prefer policy 2
E = onehot(2,2) # Initialize as a one-hot encoded vector: [0,1]
```

We now create the Dirichlet priors for **A**, **B** and **D**. When we use parameter learning, these are used to define **A**, **B**, and **D** defined above, and are updated on every time-step. One way to construct Dirichlet priors is to simply multiply the matrices below with a scaling factor - a higher scaling leads to more precise priors that require stronger evidence to update. We here use a scaling parameter of 2. In the current version, parameter learning is only implemented for the **A**, **B** and **D**.

```
scaling_parameter = 2.0

pA = deepcopy(A) * scaling_parameter # Create pA as a scaled copy of A

pB = deepcopy(B) * scaling_parameter # Create pB as a scaled copy of B

pD = deepcopy(D) * scaling_parameter # Create pD as a scaled copy of D
```

Finally, we define the hyper parameters and other settings in two dictionaries. We here display the most common parameters and settings, and set them to their default values which are used if they are not set by the user.

```
parameters = Dict(
                  "alpha" => 16.0, # Action precision alpha
                  "gamma" => 16.0, # Policy precision gamma
                  "lr_pA" => 1.0,  # Learning rate of A
                  "fr_pA" => 1.0,  # Forgetting rate of A
                  "lr_pB" => 1.0,  # Learning rate of B
                  "fr_pB" => 1.0,  # Forgetting rate of B
                  "lr_pD" => 1.0,  # Learning rate of D
                  "fr_pD" => 1.0   # Forgetting rate of D
```

```
                   )

settings = Dict(
                # Length of policies to consider
                "policy_len"           => 1,
                # Whether to use state information gain for action selection
                "use_states_info_gain" => true,
                # Whether to use parameter information gain for action selection
                "use_param_info_gain"  => false,
                # Whether to use pragmatic gain for action selection
                "use_utility"          => true,
                )
```

We can now input the above arguments into `init_aif` and create an AIF object. This allows us to present it with observations and let it choose actions. The following displays a few functions in the order they are usually used, first inferring environmental states, then updating matrices, and then selecting a new action. First, we make inferences on the environment with `infer_states!`, which returns the updated posterior belief $q(s)$ about environmental states given an observation. The observation is presented as a vector with an entry for each observation modality.

```
observation = [1] # Define an observation for each observation modality
qs = infer_states!(aif, observation) # Produce a posterior belief about states
```

We can now also update the Dirichlet beliefs about the various matrices, with the `update_A!`, `update_B!` and `update_D!` functions. **A** is updated based on the last observation and the posterior belief about the state that generated it. **B** is updated based on the posteriors about the previous state transition given the previous action. **D** is updated based on the posterior over states at the first time-step.

```
# Update A
update_A!(aif, observation)

# Update B
update_B!(aif, qs_previous) # qs_previous is the posterior over states at t-1

# Update D
update_D!(aif, qs_t1) # where qs_t1 is the posterior over states at the first time step
```

The `infer_policies!` function then calculates the expected free energy for each possible policy (the number of policies varies depending on the amount of possible actions per time step, and the length of policies considered), as well as the corresponding posterior probability over these policies. This calculation depends on the settings specified in the `init_aif` function, including the policy length and which parts of the information gain to use, as well as the policy precision γ.

```
# Infer policies
infer_policies!(aif)
```

Finally, `sample_action!` then samples the next action from the agent. This is done by marginalizing the policy probabilities to get the probabilities for the action on the next time-step, and then softmax transforming it with the α action precision parameter.

```
# Sample Action
sample_action!(aif)
```

These functions can be combined by users in various ways, depending on their purpose. Often, however, users will want to combine them in a single function implementing the full action-perception loop that receives an observation and returns an action. This is implemented with the `ActionModels` sister package for behavioural modelling.

3.2. *Simulation with* `ActionModels`

`ActionModels` is a library for implementing, simulating and fitting to data various behavioural models. We here show how to use it in conjunction with `ActiveInference` to make simulation of AIF models easy, and in a fully generalized framework that is compatible with other types of cognitive and behavioural models as well. `ActiveInference` provides a full 'action model' - a full model of the action-generating process in an agent - for using AIF, called `action_pomdp!`. In this case, all this information is contained in the AIF object. `action_pomdp!` then takes the AIF object and a single-time-step observation as arguments, and then runs state inference, parameter learning and policy inference, and returns probability distributions over the possible actions of the agent.

```
observation = [1] # observation with one modality
# Run the action model for a single observation
action_distributions = action_pomdp!(aif::AIF, observation)
```

This can conveniently be used in conjunction with an `ActionModels Agent` - a more abstract structure which is used for running behavioural models in general, and which is used when fitting models to data. We therefore begin with initialising an `Agent` that contains the AIF object.

```
# Initialize ActionModels Agent with active inference agent as a substruct.
agent = init_agent(
                    action_model = action_pomdp!, # The active inference action model
                    substruct = aif,              # The AIF object
                   )
```

The `Agent` object can be used with a set of standard functions. `single_input!` will provide the agent with an observation, update it is beliefs and return a sampled action; for non-action-dependent observations, `give_inputs!` provides a series of observations across time-steps, and returns actions for each. These can be easily used in an agent-based simulation to have AIF agents evolve and act over time.

```
# Give single observation to the agent
observation = [1]
action = single_input!(agent, observation)

# Give multiple observations to the agent and simulate actions
observations = [1, 2, 1, 2, 3]
actions = give_inputs!(agent, observations)
```

Additionally, a set of convenience functions can extract and set parameters and (histories of) beliefs. We briefly show how to extract current or histories of past states:

```
# Get all current belief states
states = get_states(agent)
# Get a specific state, like the expected free energies only
efe = get_states(agent, "expected_free_energies")

# Get history for all states
history = get_history(agent)
# Get history of expected free energies only
history_efe = get_history(agent, "expected_free_energies")
```

And how to change parameters of a created agent:

```
# Set individual parameter, alpha to 1.0
set_parameters!(agent, "alpha", 1.0)

# Set multiple parameters by passing a dictionary
new_parameters = Dict(
                      "alpha" => 1.0, # Set alpha to 1.0
                      "lr_pA" => 0.5  # Set learning rate of A to 0.5
                      )
set_parameters!(agent, new_parameters) # Set new parameters to agent
```

3.3. *Model Fitting with* `ActionModels`

In addition to simulating behaviour and belief updating of agents, `ActionModels` also makes it possible to fit models to data and do parameter estimation. This is used in general to form better models and theories of mental processes, as well as for finding mechanistic differences (usually prior beliefs in AIF) between, for example, clinical populations - or investigating how computational constructs like Bayesian beliefs relate to for example neuronal dynamics. This is done in fields like cognitive modeling and mathematical psychology [34], as well as computational psychiatry [14,53]. In the following, we briefly describe the high-level functions needed to fit AIF models to empirical data with `ActionModels`.

We have our `Agent` object defined as above. We then need to specify priors for the parameters we want to estimate. Here we will estimate the α parameter, and use a Gamma distribution as prior:

```
# Load package for specifying distributions
using Distributions
# Initialize priors
priors = Dict("alpha" => Gamma(4,4))
```

We can now use the `create_model` function to instantiate a probabilistic model object with data. This takes the agent object, the priors, and a set of observations and actions as arguments:

```
# Initialize model for single subject
single_subject_model = create_model(
  agent,  # ActionModels agent object
  priors, # Dictionary with parameter priors
  observations, # Vector of observations
  actions,      # Vector of actions
)
```

If we have multiple subjects whose parameters we wish to estimate, we can do so by passing a DataFrame object to the same function. Here, we specify which columns are actions and inputs, as well as which column to use for grouping the specific time series.

```
# Initialize model for multiple subjects
multiple_subjects_model = create_model(
  agent,  # ActionModels agent
  priors, # Dictionary with parameter priors
  data;   # Dataframe with data from multiple subjects
  grouping_cols = ["ID"], # Column to split dataframe
  input_cols = ["observations"],  # Columns with observations
  action_cols = ["actions"],      # Columns with actions
)
```

This model can be used as a normal `Turing` model object. `ActionModels` provides a convenience function for doing this with appropriate defaults:

```
results = fit_model(
  single_subject_model; # Model object
  n_iterations = 1000,  # Number of iterations
  n_chains = 1          # Number of chains
)
```

The output of the `fit_model` function is an object containing the standard `Turing` chains, which we can use to access the chain statistics:

```
# Extract the Chains object
chains = results.chains

#Rename the chains for interpretable parameter names
rename_chains(chains, model)
```

`ActionModels` provides a range of convenience functions for behavioural modelling. We can extract the posterior parameter estimates for each participant, and extract it in a convenient data frame structure for later processing:

```
# Extract quantities from the chain
agent_parameters = extract_quantities(single_subject_model, chains)
# Get posterior median
estimates = get_estimates(agent_parameters)
```

We can also sample parameter values from the prior, and plot the posteriors against the priors:

```
#Sample from the prior
prior_chains = sample(single_subject_model, Prior(), 1000)

# Plot parameter distribution.
plot_parameter_distribution(chains, prior_chains)
```

See the documentation for `ActionModels` at github.com/ilabcode/ActionModels.jl for various other functionalities, including modelling how parameters vary across a population, parameter recovery, predictive checks, and more. We have in this section outlined how to use `ActiveInference` for simulation and model fitting, in conjunction with `ActionModels`. In the following section, we show how to do this on a concrete worked example.

4. Usage Example

In this section, we demonstrate a full usage example of how to create an AIF agent, simulate behaviour in a classic T-maze environment, and fit the AIF agent to a simulated example dataset. We will provide the necessary code to run. All code required to reproduce the example simulation can be found on an open source OSF repository osf.io/j3k5q/. This example is done with the current version of `ActiveInference.jl` (0.0.2) - the newest version can be found at github.com/ilabcode/ActiveInference.jl.

4.1. Setting up Environment and Agent

A T-maze is a simple task commonly employed in the behavioural sciences, as well as in the AIF literature [14,54–57]. It is a minimal type of task that requires balancing exploration and exploitation, or epistemic and pragmatic value, respectively. It also is suitably represented in a discrete state space. Together, this makes it easily compatible with a POMDP-based AIF approach.

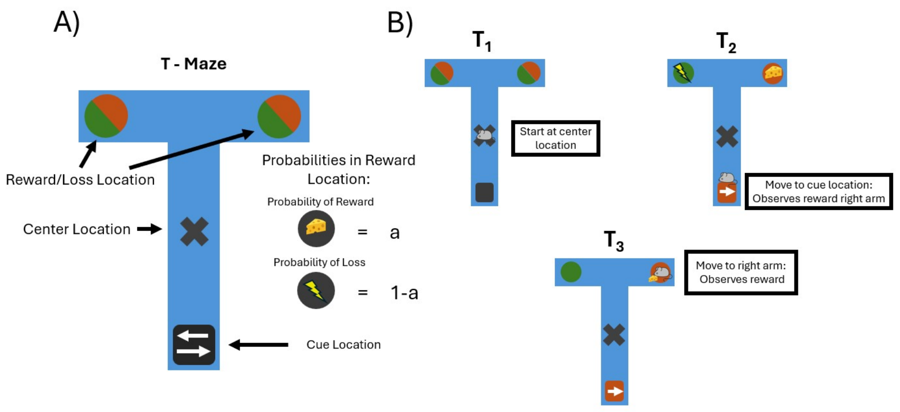


Figure 2. A depiction of the T-maze. A) The full layout of the T-maze task, with the centre location, the cue location, and the two reward conditions. B) A three-step example of a T-maze trial. The agent (in this case, a mouse) starts at the centre location. In order to reduce the uncertainty regarding which arm the reward is located in, the agent moves to the cue location. The cue location reveals the right arm to be the reward location, and in the subsequent time step it goes to the right arm and observes reward with some probability.

The structure of the T-maze is, as the name suggests, a T-shaped maze, consisting of a centre location, a cue location (bottom of the T), and reward and loss locations (one in each arm of the T) (Figure 2). On every trial, the agent can move to one of the two arms of the T to receive a reward; one, called the reward location, will yield rewards with a higher probability than the other side. At the cue location, which the agent can move to, the agent receives a cue which indicates which of the locations is the reward location. Generally, the cue may be more or less informative - in this example, it always accurately reflects the reward conditions state (reward in the right or left arm). The reward location only provides reward probabilistically. This means the agent can either take a chance and go directly to one of the two upper arms, or spend its first move seeking information about where the reward is before moving to the reward location. Since the clue location is not preferred, the second option comes with a cost in terms of pragmatic value, which has to be outweighed by the epistemic value in resolving uncertainty about the reward location state. Note that, for that the agent to realize that this uncertainty reduction will aid it in its subsequent choice of arm, it would have to be able to anticipate the effect of its actions on its own future beliefs - a process called "sophisticated inference" [58].

In the following, we proceed to describe the generative model used by the agent, which is here identically structured to the *generative process* - the actual environmental process generating the data. In the maze task, the environmental states and concurrent observations are multidimensional, while actions are unidimensional. We here refer to the different dimensions of the environment as *state factors*, and the different dimensions of the observations as *observation modalities*. There are two state factors, the location of the agent, and the reward condition:

State Factor 1 (Location):

1. Centre Location
2. Right Arm
3. Left Arm
4. Cue Location

State Factor 2 (Reward Condition):

1. Reward Condition Right
2. Reward Condition Left

The first factor describes which of the four locations in the maze the agent is inhabiting - the centre, cue location, and left and right arms - and is controlled by the action. The second factor is the reward condition factor, and has two possible states: the reward location being on the right or on the left. The agent's actions directly control the first factor, while it has to infer the second based on the cue in order to complete the task. There are three observation modalities: one for observing the agent's own location, one for observing rewards, and one for observing the cue:

Modality 1 (Location):

1. Centre Location
2. Right Arm
3. Left Arm
4. Cue Location

Modality 2 (Reward):

1. No Reward
2. Reward
3. Loss

Modality 3 (Cue):

1. Cue Right
2. Cue Left

The first observation modality has an observation for each of the positions in the maze, and will generally reflect the agent's actual position perfectly. The second modality is the agent's observation of a reward or loss, which will depend on the reward condition and the position. The "no reward" observation is received whenever the agent is not occupying either of the arm locations, where it does not observe any reward or loss. The third modality is the observation of the cue, which will depend on the reward modality so that the agent can use it to infer the correct reward condition. The action is one-dimensional, with four options, used to move between the different locations:

Actions:

1. Move to Centre Location
2. Move to Right Arm
3. Move to Left Arm
4. Move to Cue Location

We will now provide an example of using the package with a T-maze environment. First, we will install and load the package:

```
using Pkg
Pkg.add(ActiveInference)

using ActiveInference
```

We first set up the environment or generative process, the T-maze. The T-maze is included as a pre-made environment in `ActiveInference`, so we simply load it. We set the reward probability for the reward condition to 95 percent.

```
using ActiveInference.Environments

# setting the probability of reward to 0.95
Env = TMazeEnv(0.95)
initialize_gp(Env)
```

We then proceed to set up **A**, **B**, **C**, **D**, and **E**. For this we use the `create_matrix_templates` helper function to set up the correct structure, and then populate it. To start, we will define what goes into the helper function. It takes five arguments; the number of states and observations in each factor and modality, the policy length, the number of controls, and lastly; what to initially populate them with. The specific states and observations will be made clear as we populate the parameters.

The first three argument should be specified as vectors of integers, containing the number of state and observations for each factors or modality. In our case we have two factors; a location factor, and a reward condition factor, which respectively have 4 and 2 states. There are 3 modalities; one location modality with 4 observations, one reward modality with 3 observations, and one cue modality with two observations. Finally, there are four possible actions for controlling the location factor, and only one possible action for the reward state factor. The policy length is specified as an integer, in our case 2, and we will populate the template with zeros.

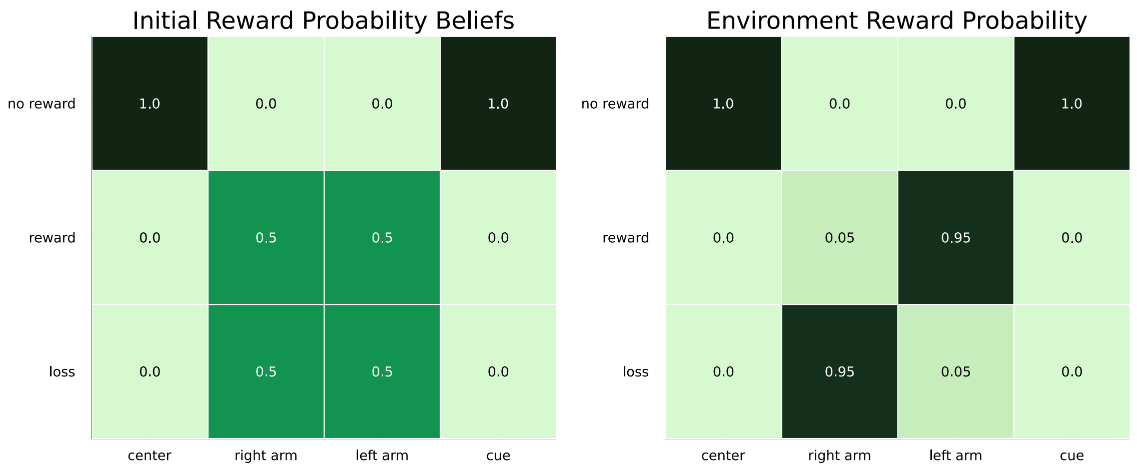
```
A, B, C, D, E = create_matrix_templates([4, 2], [4, 3, 2], [4, 1], 2, "zeros")
```

We start by defining **A**, or observation model. Since we in this example allow for **A** learning, **A** will not be used directly, but we still define it in order to construct the Dirichlet prior over it (and it could be used directly if **A** learning was not required). Of the three observation modalities, the first is the location observation. Here there are 4 possible observations, mapping to the 4 location states and 2 reward condition states. This results in an **A** that is 4 (location observations) by 4 (location states) by 2 (reward condition), an 4x4x2 A tensor. We let the agent correctly assume perfect observations of the location by specifying an identity matrix for **A** in each reward condition.

```
# Use identity matrices between locations and observations.
# For reward condition right
A[1][:,:,1] = [ 1.0  0.0  0.0  0.0
                0.0  1.0  0.0  0.0
                0.0  0.0  1.0  0.0
                0.0  0.0  0.0  1.0 ]

# For reward condition left
A[1][:,:,2] = [ 1.0  0.0  0.0  0.0
                0.0  1.0  0.0  0.0
                0.0  0.0  1.0  0.0
                0.0  0.0  0.0  1.0 ]
```

The second modality is the reward modality, and maps the observations "no reward", "reward" and "loss" onto the location states and reward conditions. For the second modality we therefore have a tensor that is 3 (reward observations) by 4 (location states) by 2 (reward condition), a 3x4x2 tensor. When the agent is at the centre and cue locations, we let it accurately expect with certainty the observation of "no reward". For the two arm locations, we let the agent be agnostic regarding whether they will provide rewards or losses. This is different from the true reward probabilities (see figure 3), which the agent will have to learn over time. This is the case for both reward conditions.

```
# Set center (column 1) and cue (column 4) to give no reward observation (row 1)
# Set reward and loss probabilities to 0.5 for the arm locations

# For reward condition right
A[2][:,:,1] = [ 1.0  0.0  0.0  1.0
                0.0  0.5  0.5  0.0
                0.0  0.5  0.5  0.0 ]

# For reward condition left
A[2][:,:,2] = [ 1.0  0.0  0.0  1.0
                0.0  0.5  0.5  0.0
                0.0  0.5  0.5  0.0 ]
```



	center	right arm	left arm	cue
no reward	1.0	0.0	0.0	1.0
reward	0.0	0.5	0.5	0.0
loss	0.0	0.5	0.5	0.0

	center	right arm	left arm	cue
no reward	1.0	0.0	0.0	1.0
reward	0.0	0.05	0.95	0.0
loss	0.0	0.95	0.05	0.0

Figure 3. Reward probabilities for the four locations. The centre (column 1) and cue (column 4) locations always result in the "no reward" observation (row 1). The two arms (columns 3 and 4) result in either rewards (row 2) or losses (row 3), with some probability. Left: The agent's agnostic starting beliefs about reward probabilities. Right: The true reward probabilities for reward condition left arm, which the agent must learn over time.

The third and last modality is the cue modality, which maps the cue observations onto the location and reward condition states. This results in an **A** which is 2 (cue observation) by 4 (location state) by 2 (reward condition), a 2x4x2 tensor. Observations in this modality is correctly assumed by the agent to truthfully reveal the current reward condition - i.e., whether the right or left arm is better - when standing on the cue location. We implement this by giving each cue observation equal probabilities at all locations except the cue location, where there is a perfect correspondence between the reward condition and the observation.

```
# Set cue observation probabilities to be equal at all locations except the cue location (column 4)
# Let cue locations correspond to reward conditions.

# For reward condition right
A[3][:,:,1] = [ 0.5   0.5   0.5   1.0
                0.5   0.5   0.5   0.0 ]

# For reward condition left
A[3][:,:,2] = [ 0.5   0.5   0.5   0.0
                0.5   0.5   0.5   1.0 ]
```

Having now created all three modalities of **A**, we can continue to **B**, or the transition model. Each of the two state factors of **B** - the location factor and reward condition factor - needs to be defined separately. We start with the location factor, which contains transition to and from four possible location states, under four different actions - a 4x4x4 tensor. The agent can control these states perfectly with it's four movement actions, independently of it's current position. This is implemented by letting the probability of transitioning to a location be 1 for the location corresponding to the action chosen:

```
# Set transition probabilities to 1 for the chosen location
# For action "Move to Center Location"
B[1][:,:,1] = [ 1.0   1.0   1.0   1.0
                0.0   0.0   0.0   0.0
                0.0   0.0   0.0   0.0
                0.0   0.0   0.0   0.0 ]

# For action "Move to Right Arm"
B[1][:,:,2] = [ 0.0   0.0   0.0   0.0
                1.0   1.0   1.0   1.0
                0.0   0.0   0.0   0.0
                0.0   0.0   0.0   0.0 ]
```

```
# For action "Move to Left Arm"
B[1][:,:,3] = [ 0.0   0.0   0.0   0.0
                0.0   0.0   0.0   0.0
                1.0   1.0   1.0   1.0
                0.0   0.0   0.0   0.0 ]

# For action "Move to Cue Location"
B[1][:,:,4] = [ 0.0   0.0   0.0   0.0
                0.0   0.0   0.0   0.0
                0.0   0.0   0.0   0.0
                1.0   1.0   1.0   1.0 ]
```

The reward condition factor in **B** here is rather simple, as we let the agent correctly assume that the reward condition never changes (although one could have enabled **B** learning to let the agent learn the transition probabilities of the environment). There are two possible states for this state factors, so we implement this as a 2x2 identity matrix.

```
# Identity matrix representing agent belief that reward condition does not change
B[2][:,:,1] = [ 1.0   0.0
                0.0   1.0 ]
```

We now build **C**, the prior preference over observations.**C** is encoding preferences for each observation each modality: the location, reward and cue. The relative value of the entry corresponding to a specific observation defines the preference; we here set all values with neutral preferences as 0, and use positive values for preferences and negative values for aversions. The only observations over which the agent has preferences are in reward modality, with rewards (index 2) being preferred and losses (index 3) being disliked. The strength of the preference will determine the explore-exploit balance of the agent; we here use 3 as value.

```
# Preference over locations modality
C[1] = [0.0, 0.0, 0.0, 0.0] #No preference

# Preference over reward modality
C[2] = [0.0, 3.0, -3.0] #Rewards preferred, losses not preferred

# Preference over cue modality
C[3] = [0.0, 0.0] #No preference
```

We now build **D**, which is the prior or initial belief over environmental states. There is vector in **D** for each of the two state factors, the location state and the reward condition. We let the agent start with a belief that it is at the centre location (index 1), and give uniform priors for the reward condition.

```
# For the location state factor
D[1] = [1.0, 0.0, 0.0, 0.0] # Certain beleif of being at center location

# For the reward condition state factor
D[2] = [0.5, 0.5] # Agnostic prior over reward condition state
```

Finally, we establish **E**, the prior over policies. This is a vector over all possible policies - with four possible actions and a policy length of 2, there are $4^2 = 16$ policies in total. We here give the agent an agnostic prior over policies.

```
# Setting uniform E vector to not prefer any policies by equally preferring all
E .= 1.0/length(E)
```

Now the five matrices have been defined (we note that C, D and E matrices are set to agnostic by default if not defined by the user). All that is left is to set the Dirichlet prior for **A** learning, and set parameters and settings. In the following, we set the the Dirichlet prior for **A** to a scaled copy of the original **A**, using a weak scaling parameter of 2.0.

```
# Use a weak scaling parameter
scale_concentration_parameter = 2.0

# Make a scaled copy of A
pA = deepcopy(A) * scale_concentration_parameter
```

We then set the various hyper-parameters for the agent's inference algorithm. We here use standard default values: an **A** learning rate of 1, and γ and α values of 16.

```
parameters = Dict(
    "lr_pA" => 1.0,
    "alpha" => 16.0,
    "gamma" => 16.0,
)
```

Finally, we define the settings of the agent (other characteristics of the agent that are not estimable parameters). Here, we set the policy length to 2, which lets the agent use expected information gain for both state and parameter info gain for it's actions, and specify that it is specifically the second modality of **A** that is to be learned.

```
settings = Dict(
    # A policy length of two
    "policy_len" => 2,

    # Use parameter information gain in action selection
    "use_param_info_gain" => true,

    # Use state information gain in action selection
    "use_states_info_gain" => true,

    # Only do A learning for the reward modality.
    "modalities_to_learn" => [2]
)
```

Having built all the necessary components of the generative model, we can now instantiate an AIF agent.

```
aif_agent = init_aif(
    A, B, C = C, D = D, E = E, pA = pA, settings = settings, parameters = parameters
)
```

4.2. Simulating Behaviour

Now that the environment and agent have been set up, we can proceed to simulate the behaviour of the agent in the environment. We create a `for` loop, where the agent receives an observation, makes inference on the environment, updates **A**, infers policies, and samples actions.

```
# Run 1000 time steps
T = 1000

# Sample initial observation
```

```
obs = reset_TMaze!(Env)

# For loop over every time step
for t = 1:T

    # Infer states based on the current observation
    infer_states!(aif_agent, obs)

    # Infer policies and calculate expected free energy
    infer_policies!(aif_agent)

    # Updating A. This is the learning part,
    # that includes the counting of the pA Dirichlet.
    update_A!(aif_agent, obs)

    # Sample an action based on the inferred policies
    chosen_action = sample_action!(aif_agent)

    # Feed the action into the environment and get new observation.
    obs = step_TMaze!(Env, chosen_action)
end
```

The agent here starts by moving to the cue location, and then proceeds to move to the left arm repeatedly. The main objects of learning here are the reward condition state and the **A** parameters for rewards under the two reward conditions. After observing the cue, the agent updates its belief (correctly) to be certain of being in the left reward condition (figure 4). Over time, the agent learns the correct probability of receiving rewards at the left arm (.94, versus a correct .95). It does not learn the probabilities for the right arm - this is because it never moves to that location, having already learnt that the left arm is more likely to produce rewards (figure 5). This would be less likely to be the case with lower γ and α values, as well as a more entropic **C**.

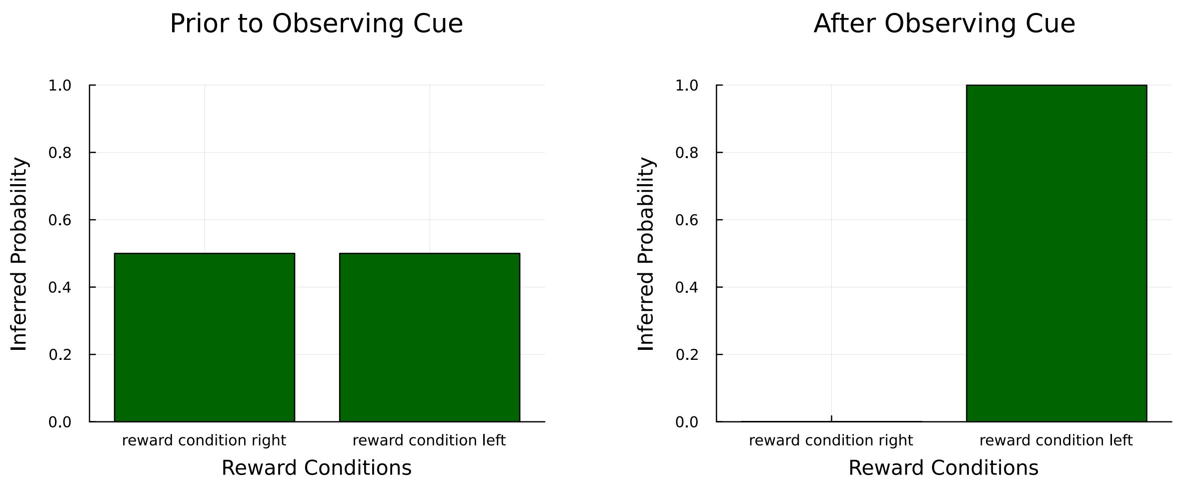


Figure 4. State inference for the reward condition. The agent's beliefs about the reward condition changes from agnostic (left) to certain that it is the left reward condition (right) after observing the cue.

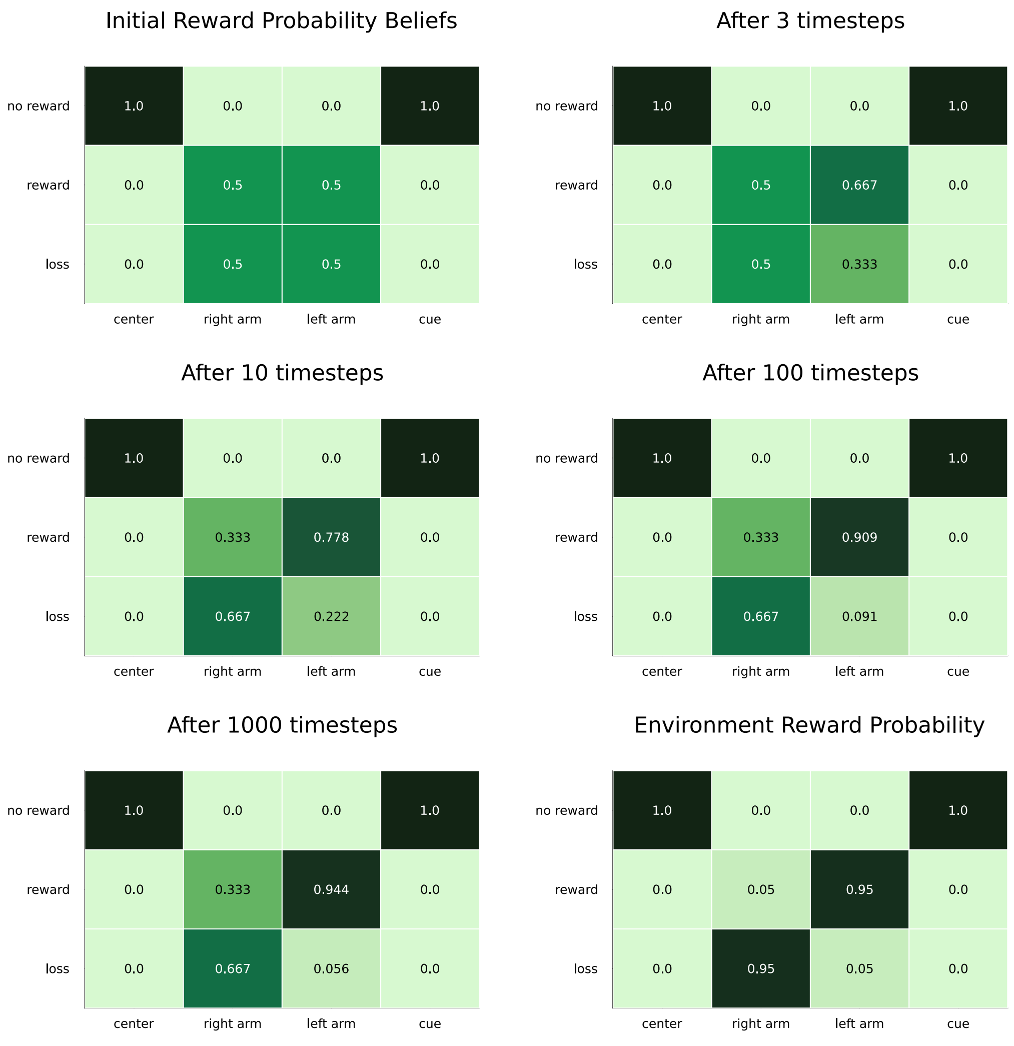


Figure 5. **A** learning for the actual reward condition (reward condition left). The agent correctly learns the probability of receiving rewards in the rewarding arm. It does not learn the probabilities of the non-rewarding arm, since it does not explore that option.

4.3. Fitting the Model to Data

Simulations are useful for a variety of purposes, like exploring the consequences of different priors and parameters and for establishing the face validity of hypothetical mechanisms underlying behavioural phenomena. However, we often want to use models to make inference about specific observed phenomena, like differences in behaviour between various populations, as in computational psychiatry [14]. One standard method here is model fitting, where we estimate the parameter values (e.g. prior beliefs) of an AIF model that are most likely given some observed behaviour of a participant. This is often done with approximate Bayesian methods. In the cognitive and behavioural sciences, the predominant method is Markov-Chain Monte Carlo (MCMC) methods [34], which are slower but in the limit can estimate parameter posteriors without making assumptions about their functional form. An alternative, which is more often used in other fields and also available in `ActiveInference`, is variational methods, which are faster, but require making assumptions about the functional form of the posterior. In general, MCMC methods are favourable when making parameter inference (i.e., comparing parameters of the same model fitted to different data, like two groups of subjects). When doing Bayesian model comparison (i.e. comparing different models fitted to the same data), the different approaches rely on different approximations of the model evidence, with the variational free energy having some claims to being a better approximation than the information criteria classically used with MCMC methods (although see other approximations like the Pareto-Smoothed Importance Sampling [59] or Thermodynamic Integration methods [60] - see [35] for a further review). Note that, independently of which of these approaches one might take, the process involves inverting a generative model of the mental processes underlying the behaviour of a given subject - a generative model which itself is an inversion of the subject's generative model of the environment. We can call the generative model that the agent has of its environment the *subjective* generative model, and the model we have of the agent the *objective* generative model, in what has been called a meta-Bayesian approach or 'observing the observer' [1,61].

We here demonstrate model fitting by fitting the POMDP model to the synthetic behaviour that it generated - this is called a parameter recovery study, since we can then compare the estimated parameters to the generative values used for creating the simulated data [62,63]. Here, we use the simulation method shown in the previous section to produce a synthetic dataset with known parameter values for each agent (in practice often participants in an experiment), here focusing on estimating the α parameter. We then use MCMC methods to estimate the parameters for each agents, and compare the estimated values with the correct values. We here simulate two groups of 5 synthetic subjects agents with differing α values (the parameters for the first group are sampled from a Gaussian distribution with mean = 8 and SD = 2, the second group with with mean = 24 and SD = 2). Each agent interacts with the T-maze environment for 300 time steps each. We produce the following data frame, containing the data of each of the agents: their observations, actions, and an identifier - a format suitable for cognitive and behavioral modelling.

```
3000×5 DataFrame
 Row │  Location    Reward     Cue    Action_Location   Action_Reward   SubjectID
     │     Int64     Int64    Int64       Int64              Int64         Int64
─────┼─────────────────────────────────────────────────────────────────────────────
   1 │     1           1        1           4                  1             1
   2 │     4           1        2           3                  1             1
   3 │     3           3        2           2                  1             1
   . │     .           .        .           .                  .             .
   . │     .           .        .           .                  .             .
3000 │     2           2        2           2                  1            10
```

We will use `ActionModels` to fit the AIF model created above to each of the agents in the dataset. We begin by initializing an `ActionModels` agent:

```
using ActionModels

# Initialize ActionModels Agent with the action model and created active inference agent
agent = init_agent(
    action_model = action_pomdp!, # Action model function
    substruct = aif, # Active inference agent as a substruct
)
```

We then set the prior for the parameter we want to estimate - the α action precision. We choose as example a wide, weakly informative prior: a Gaussian distribution with mean 5 and standard deviation 5, truncated at 0 and 20.

```
using Distributions

# Set priors
parameter_priors = Dict(
    "alpha" => Truncated(Normal(5,5), 0, 20)
)
```

Next, we instantiate the probabilistic model with data and parameter priors:

```
 model = create_model(
    agent, # ActionModels Agent containing the model
    parameter_priors, # Dictionary with parameter priors
    data; # Dataframe containing data from all subjects
    grouping_cols = [:SubjectID], # Identifier column
    input_cols = ["Location", "Reward", "Cue"], # Observation columns
    action_cols = ["Action_Location", "Action_Reward"] # Columns containing actions
)
```

Finally, we use the `fit_model` function to do (parallelized) parameter estimation for each of the agents:

```
results = fit_model(
    model; # Model object
    parallelization = MCMCDistributed() # Parallelize over chains
    n_iterations = 1000, # Number of iterations
    n_chains = 4 # Number of chains
)
```

The output contains the `Chains` object containing the resulting posterior samples:

```
# Take out chains from the results object and rename parameters
renamed_posterior_chains = rename_chains(results.chains, model)
```

```
    Summary Statistics
parameters          mean      std       mcse      ess_bulk    ess_tail    rhat
SubjectID:1.alpha   3.8785    0.2350    0.0034    4826.0114   2512.8061   1.0016
SubjectID:2.alpha   2.9718    0.1945    0.0029    4523.9650   2781.4532   1.0033
SubjectID:3.alpha   3.3598    0.2147    0.0031    4816.1661   3054.7500   1.0016
        .              .         .         .          .           .          .
        .              .         .         .          .           .          .
SubjectID:10.alpha  8.9233    0.9660    0.0170    3551.1005   2126.9604   1.0013
```

We can plot the posteriors and chains, often done to diagnose whether the sampling was successful (Figure 6):

```
using StatsPlots

# Plot chain traces for the first and the last subject
plot(renamed_posterior_chains[:,1:1,:])
plot(renamed_posterior_chains[:,10:10,:])
```

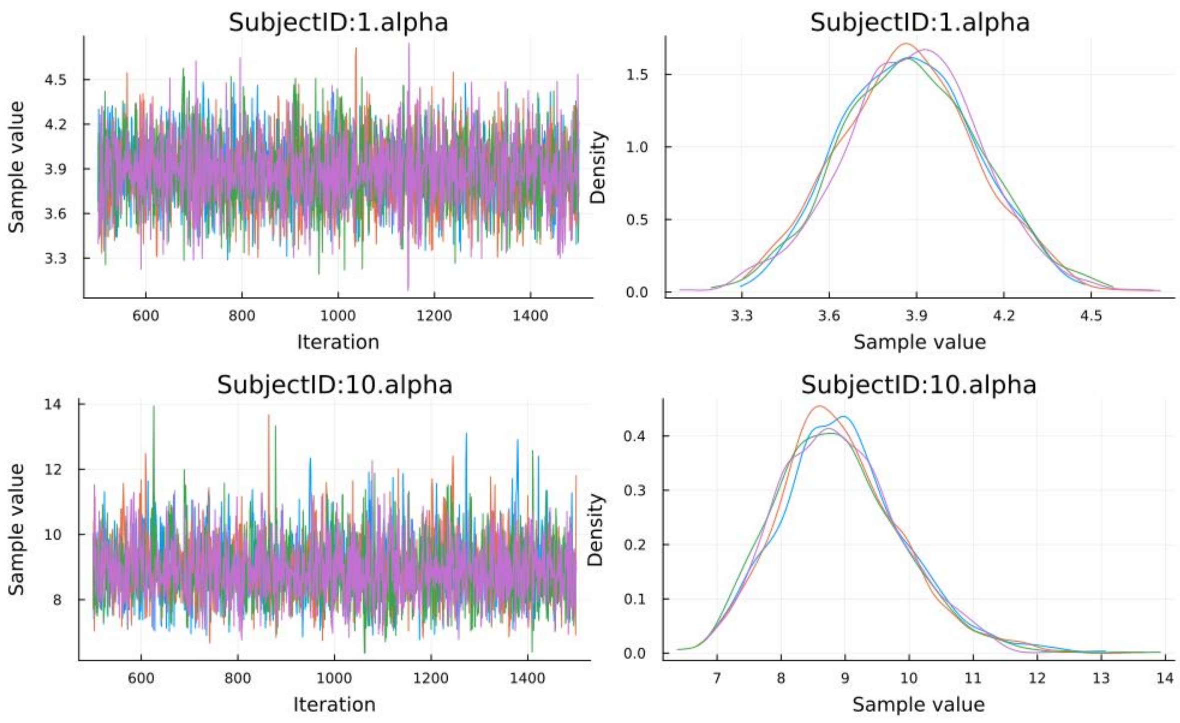


Figure 6. Chain Traces

Further, we use `plot_parameters` from `ActionModels` to plot the posterior estimates against their priors. Here we do this for one agent from each group, also highlighting the parameter value used to generate the behaviour (Figure 7). We see that the posterior are correctly centred around the generative value.

```
# Sample from the model prior
prior_chains = sample(model, Prior(), 1000)
# Rename parameters from the prior chains to match the posterior chains
renamed_prior_chains = rename_chains(prior_chains, model)

# Plot the posterior and prior for the first subject
plot_parameters(renamed_prior_chains[:,1:1,:], renamed_posterior_chains[:,1:1,:])
# Visualize the true alpha value
vline!([data[1,:Alpha]], line=:dash, color = :darkorange2, label = "Generative Alpha")

# Plot the posterior and prior for the last subject
plot_parameters(renamed_prior_chains[:,10:10,:], renamed_posterior_chains[:,10:10,:])
# Visualize the true alpha value
vline!([data[3000,:Alpha]], line=:dash, color = :darkorange2, label = "Generative Alpha")
```

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 November 2024 doi:10.20944/preprints202411.1880.v1

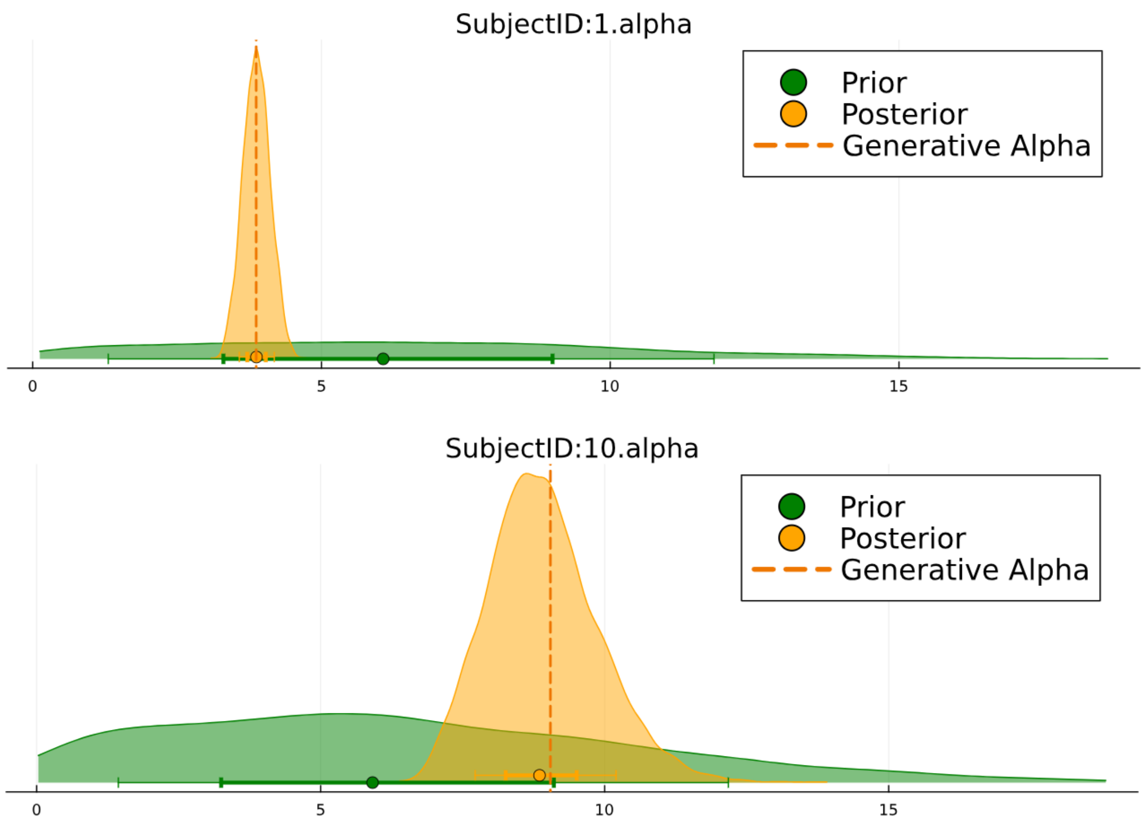


Figure 7. Posterior estimates of the α parameter plotted against the prior for two synthetic subjects, one from each group.

We then, as is often the case in computational psychiatry, want to compare the distributions of parameter values between the two groups. We extract the median of the estimated posteriors for each subject, and plot them against the value used to generate the behaviour (figure 8). We see that the estimation successfully captures the difference between the two groups, and that the α parameter recovers fairly well. Note that the ability to recover parameters depend on the specific model and task, as well as on the specific values of the parameters (when α is very high, for example, behaviour becomes essentially deterministic. Further increases in α would then not have any effect on behaviour - and therefore not be estimable). A subtle issue here is that the parameters that best explain some data are not necessarily the parameters used to generate those data. This is because the best parameters are those that maximise the marginal likelihood of the data (a.k.a., model evidence): because model evidence includes a complexity term, parameter recovery will often recover parameters that provide a simpler explanation for the data; relative to the parameters used to generate those data.

```
# Extract quantities from the fitted model
agent_parameters = extract_quantities(model, renamed_posterior_chains)
# Extract posterior estimates
posterior_estimates = get_estimates(agent_parameters)
```

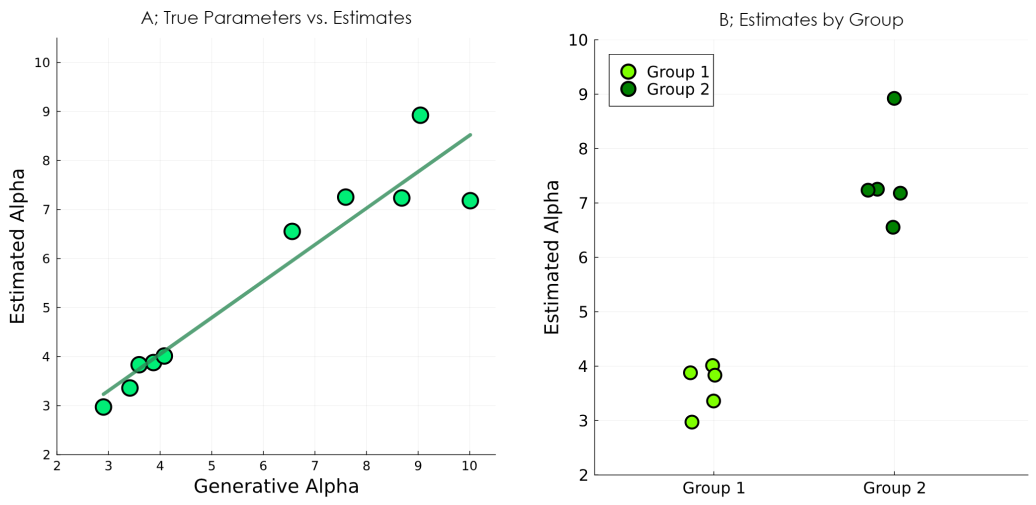


Figure 8. Results of the parameter recovery study. A) Estimated parameter values plotted against the values used to generate data. B) Parameter estimates split by the two groups from which parameter values of synthetic subjects were sampled.

Finally, there are various metrics for model comparison that might be calculated, implemented by various software packages. We here for demonstration calculate the Pareto-Smoothed Importance Sampling approximation to Leave One Out cross validation (PSIS-LOO) [59], as implemented by `ParetoSmooth.jl` [64].

```
using ParetoSmooth: psis_loo
# Calculate the PSIS LOO
PSIS_loo = psis_loo(model,results.chains)
```

5. Discussion

We have introduced `ActiveInference.jl`, a novel Julia software package for creating and using POMDP-based AIF models for simulation and fitting to empirical data, demonstrating its ease of use on a small parameter study with simulated agents. `ActiveInference.jl` makes AIF modelling available in a fast language, equipped with an interface and situated in a ecosystem oriented specifically towards cognitive and behavioural modelling.

Importantly, the ability to fit models to empirical data with sampling-based methods provides value to researchers within cognitive modelling and computational psychiatry: it allows for comparing estimated parameter values between population groups, or investigating the temporal dynamics of belief changes in experimental participants. Dynamic belief trajectories can then be related to other (for example, physiological) measures, as usual in model-based neuroscience [65]. This method can also in principle be used for fitting models to other types of experimentally observable systems, like animals, organoids [66] and simulated or emergent systems [67]. The package can also be used for agent-based modelling in general, for repeating earlier analyses with sampling based model-fitting, and for comparing POMDP-based AIF models directly to other types of models.

There are many ways in which `ActiveInference` can be improved. It would be useful to extend the set of dynamic beliefs states to include prediction errors, since they are often used for model-based neuroscience. This would entail departing from discrete state space (i.e., POMDP) models to consider continuous state space models apt for Bayesian filtering or predictive coding (see below). An alternative would be to generate prediction errors from belief updating under discrete models, where prediction errors can be read as the (KL) divergence between posterior and prior beliefs (i.e., complexity or information gain). A simple interface could be added for creating custom parametrisations of the requisite parameters—that could be parametrised with Boltzmann or Gibbs distributions, as opposed to Dirichlet distributions. Parameter learning could be extended to all generative model parameters, as well as in parametrised forms (e.g., so that the Boltzmann parameter or temperature of the parameters that are learned); similarly for the precision over expected free energies γ. Preference priors should also be implementable for environmental states, in addition to observations, and **A** can be made action dependent.

A library of pre-made canonical POMDP models could be created so that users can easily implement them directly. Alternatives to the fixed point iteration method for updating posteriors over environmental states could be included, like the marginal message passing algorithm. There are various ways in which the package can be made more computationally efficient - and it could be compared to other software implementations. There are plenty of utility and plotting functions that could be added to the package to make it easier to use, and to facilitate integration with the model fitting packages it relies on; for example, to allow for combining the models with linear regressions to compare parameters values of different populations in a single model. More complex types of POMDP models can also be added, like hierarchical and temporally deep POMDP's. Model structure learning could be considered, where different model structures are compared and chosen between by evaluating their free energies. Sophisticated inference, where predictions are also made about changes in one's own beliefs — depending on expected action-dependent observations in the future — could also be implemented [58]. Finally, the package could be extended to other types of generative models than the POMDP's, including other universal models like generalized filtering [17] and Hierarchical Gaussian Filter models [41], as well as custom generative models, or even (deep-learning based) amortized inference models. These various extensions could provide valuable tools for using AIF models in both theoretical and applied research.

Author Contributions: Conceptualization, S.W.N., J.E.L and P.T.W.; methodology, S.W.N., J.E.L and P.T.W.; software, S.W.N., J.E.L and P.T.W.; formal analysis, S.W.N. and J.E.L.; writing—original draft preparation, S.W.N. and J.E.L; writing—review and editing, C.H., K.F., C.M. and P.T.W.; visualization, S.W.N. and J.E.L; supervision, C.M. and P.T.W.; project administration, P.T.W.; All authors have read and agreed to the published version of the manuscript."

Funding: C.M. acknowledges funding from Aarhus Universitets Forskningsfonds (grant no. AUFF-E-2019-7-10) and from the Carlsberg Foundation (grant no. CF21-0439).

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: The original data presented in the study are openly available in ActiveInferenceJuliaPaper at URL: https://osf.io/j3k5q/

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AIF	Active Inference
FEP	Free Energy Principle
VFE	Variational free energy
EFE	Expected free energy
MCMC	Markov Chain Monte Carlo
POMDP	Partially-Observed Markov decision Process

References

1. Parr, T.; Pezzulo, G.; Friston, K.J. *Active Inference: The Free Energy Principle in Mind, Brain, and Behavior*; The MIT Press, 2022. doi:10.7551/mitpress/12441.001.0001.
2. Friston, K.; FitzGerald, T.; Rigoli, F.; Schwartenbeck, P.; O'Doherty, J.; Pezzulo, G. Active inference and learning. *Neuroscience & Biobehavioral Reviews* **2016**, *68*, 862–879. doi:10.1016/j.neubiorev.2016.06.022.
3. Friston, K.; FitzGerald, T.; Rigoli, F.; Schwartenbeck, P.; Pezzulo, G. Active inference: A process theory. *Neural Computation* **2017**. doi:10.1162/NECO_a_00912.
4. Friston, K.J.; Stephan, K.E. Free-energy and the brain. *Synthese* **2007**, *159*, 417–458. doi:10.1007/s11229-007-9237-y.
5. Friston, K. The free-energy principle: a unified brain theory? *Nature Reviews Neuroscience* **2010**, *11*, 127–138. Number: 2 Publisher: Nature Publishing Group, doi:10.1038/nrn2787.
6. Friston, K. The free-energy principle: a rough guide to the brain? *Trends in Cognitive Sciences* **2009**, *13*, 293–301. doi:10.1016/j.tics.2009.04.005.
7. Friston, K. A free energy principle for a particular physics, 2019. arXiv doi:10.48550/arXiv.1906.10184.
8. Friston, K.; Da Costa, L.; Sajid, N.; Heins, C.; Ueltzhöffer, K.; Pavliotis, G.A.; Parr, T. The free energy principle made simpler but not too simple. *Physics Reports* **2023**, *1024*, 1–29. doi:10.1016/j.physrep.2023.07.001.
9. Friston, K.; Kiebel, S. Predictive coding under the free-energy principle. *Philosophical Transactions of the Royal Society B: Biological Sciences* **2009**, *364*, 1211–1221. Publisher: Royal Society, doi:10.1098/rstb.2008.0300.
10. Karl, F. A Free Energy Principle for Biological Systems. *Entropy (Basel, Switzerland)* **2012**, *14*, 2100–2121. doi:10.3390/e14112100.
11. Corcoran, A.W.; Pezzulo, G.; Hohwy, J. From allostatic agents to counterfactual cognisers: active inference, biological regulation, and the origins of cognition. *Biology & Philosophy* **2020**, *35*, 32. doi:10.1007/s10539-020-09746-2.
12. Heins, C.; Millidge, B.; Da Costa, L.; Mann, R.P.; Friston, K.J.; Couzin, I.D. Collective behavior from surprise minimization. *Proceedings of the National Academy of Sciences* **2024**, *121*, e2320239121. Publisher: Proceedings of the National Academy of Sciences, doi:10.1073/pnas.2320239121.
13. Patzelt, E.H.; Hartley, C.A.; Gershman, S.J. Computational Phenotyping: Using Models to Understand Individual Differences in Personality, Development, and Mental Illness. *Personality Neuroscience* **2018**, *1*, e18. doi:10.1017/pen.2018.14.
14. Schwartenbeck, P.; Friston, K. Computational Phenotyping in Psychiatry: A Worked Example. *eNeuro* **2016**, *3*, ENEURO.0049–16.2016. doi:10.1523/ENEURO.0049-16.2016.
15. Albarracin, M.; Demekas, D.; Ramstead, M.J.D.; Heins, C. Epistemic Communities under Active Inference. *Entropy* **2022**, *24*, 476. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, doi:10.3390/e24040476.
16. Lanillos, P.; Meo, C.; Pezzato, C.; Meera, A.A.; Baioumy, M.; Ohata, W.; Tschantz, A.; Millidge, B.; Wisse, M.; Buckley, C.L.; Tani, J. Active Inference in Robotics and Artificial Agents: Survey and Challenges, 2021. arXiv:2112.01871 [cs], doi:10.48550/arXiv.2112.01871.
17. Friston, K.; Stephan, K.; Li, B.; Daunizeau, J. Generalised Filtering. *Mathematical Problems in Engineering* **2010**, *2010*, 1–34. doi:10.1155/2010/621670.
18. Waade, P.T.; Mikus, N.; Mathys, C. Inferring in Circles: Active Inference in Continuous State Space Using Hierarchical Gaussian Filtering of Sufficient Statistics. Machine Learning and Principles and Practice of Knowledge Discovery in Databases; Kamp, M.; Koprinska, I.; Bibal, A.; Bouadi, T.; Frénay, B.; Galárraga, L.; Oramas, J.; Adilova, L.; Krishnamurthy, Y.; Kang, B.; Largeron, C.; Lijffijt, J.; Viard, T.; Welke, P.; Ruocco, M.; Aune, E.; Gallicchio, C.; Schiele, G.; Pernkopf, F.; Blott, M.; Fröning, H.; Schindler, G.; Guidotti, R.; Monreale, A.; Rinzivillo, S.; Biecek, P.; Ntoutsi, E.; Pechenizkiy, M.; Rosenhahn, B.; Buckley, C.; Cialfi, D.; Lanillos, P.; Ramstead, M.; Verbelen, T.; Ferreira, P.M.; Andresini, G.; Malerba, D.; Medeiros, I.; Fournier-Viger, P.; Nawaz, M.S.; Ventura, S.; Sun, M.; Zhou, M.; Bitetta, V.; Bordino, I.; Ferretti, A.; Gullo, F.; Ponti, G.; Severini, L.; Ribeiro, R.; Gama, J.; Gavaldà, R.; Cooper, L.; Ghazaleh, N.; Richiardi, J.; Roqueiro, D.; Saldana Miranda, D.; Sechidis, K.; Graça, G., Eds.; Springer International Publishing: Cham, 2021; Communications in Computer and Information Science, pp. 810–818. doi:10.1007/978-3-030-93736-2_57.
19. Weber, L.A.; Waade, P.T.; Legrand, N.; Møller, A.H.; Stephan, K.E.; Mathys, C. The generalized Hierarchical Gaussian Filter, 2023. arXiv:2305.10937 [cs, q-bio], doi:10.48550/arXiv.2305.10937.

20. Friston, K.J.; Trujillo-Barreto, N.; Daunizeau, J. DEM: A variational treatment of dynamic systems. *NeuroImage* **2008**, *41*, 849–885. doi:10.1016/j.neuroimage.2008.02.054.
21. Inc, T.M. MATLAB version: 9.13.0 (R2022b), 2022.
22. Penny, W.D.; Friston, K.J.; Ashburner, J.T.; Kiebel, S.J.; Nichols, T.E. *Statistical Parametric Mapping: The Analysis of Functional Brain Images*; Elsevier, 2011. Google-Books-ID: G_qdEsDlkp0C.
23. Heins, C.; Millidge, B.; Demekas, D.; Klein, B.; Friston, K.; Couzin, I.D.; Tschantz, A. pymdp: A Python library for active inference in discrete state spaces. *Journal of Open Source Software* **2022**, *7*, 4098. doi:10.21105/joss.04098.
24. Rossum, G.v.; Drake, F.L. *The Python language reference*, release 3.0.1 [repr.] ed.; Number Pt. 2 in Python documentation manual / Guido van Rossum; Fred L. Drake [ed.], Python Software Foundation: Hampton, NH, 2010.
25. Gregoretti, F.; Pezzulo, G.; Maisto, D. cpp-AIF: A multi-core C++ implementation of Active Inference for Partially Observable Markov Decision Processes. *Neurocomputing* **2024**, *568*, 127065. doi:10.1016/j.neucom.2023.127065.
26. Josuttis, N.M. *The C++ Standard Library: A Tutorial and Reference*; Addison-Wesley, 2012. Google-Books-ID: 9DEJKhasp7gC.
27. Bagaev, D.; Podusenko, A.; Vries, B.d. RxInfer: A Julia package for reactive real-time Bayesian inference. *Journal of Open Source Software* **2023**, *8*, 5161. doi:10.21105/joss.05161.
28. Bezanson, J.; Karpinski, S.; Shah, V.; Edelman, A. Julia Language Documentation **2016**.
29. van de Laar, T.W.; de Vries, B. Simulating Active Inference Processes by Message Passing. *Frontiers in Robotics and AI* **2019**, *6*. Publisher: Frontiers, doi:10.3389/frobt.2019.00020.
30. Vanderbroeck, M.; Baioumy, M.; Lans, D.v.d.; Rooij, R.d.; Werf, T.v.d. Active inference for Robot control: A Factor Graph Approach. *Student Undergraduate Research E-journal!* **2019**, *5*, 1–5. doi:10.25609/sure.v5.4181.
31. van de Laar, T.; Şenöz; Özçelikkale, A.; Wymeersch, H. Chance-Constrained Active Inference. *Neural Computation* **2021**, *33*, 2710–2735. doi:10.1162/neco_a_01427.
32. Busemeyer, J.R.; Diederich, A. *Cognitive Modeling*; SAGE, 2010. Google-Books-ID: R7KDF35g5LQC.
33. Smith, R.; Friston, K.J.; Whyte, C.J. A step-by-step tutorial on active inference and its application to empirical data. *Journal of Mathematical Psychology* **2022**, *107*, 102632. doi:10.1016/j.jmp.2021.102632.
34. Lee, M.D.; Wagenmakers, E.J. *Bayesian Cognitive Modeling: A Practical Course*, 1 ed.; Cambridge University Press, 2014. doi:10.1017/CBO9781139087759.
35. Blei, D.M.; Kucukelbir, A.; McAuliffe, J.D. Variational Inference: A Review for Statisticians. *Journal of the American Statistical Association* **2017**, *112*, 859–877. arXiv:1601.00670 [cs, stat], doi:10.1080/01621459.2017.1285773.
36. Lattner, C.; Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86. doi:10.1109/CGO.2004.1281665.
37. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2021.
38. Carpenter, B.; Gelman, A.; Hoffman, M.D.; Lee, D.; Goodrich, B.; Betancourt, M.; Brubaker, M.; Guo, J.; Li, P.; Riddell, A. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* **2017**, *76*, 1–32. doi:10.18637/jss.v076.i01.
39. Ge, H.; Xu, K.; Ghahramani, Z. Turing: A Language for Flexible Probabilistic Inference. Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. PMLR, 2018, pp. 1682–1690. ISSN: 2640-3498.
40. Hoffman, M.D.; Gelman, A. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo **2014**.
41. Mathys, C.; Weber, L. Hierarchical Gaussian Filtering of Sufficient Statistic Time Series for Active Inference. Active Inference; Verbelen, T.; Lanillos, P.; Buckley, C.L.; De Boom, C., Eds.; Springer International Publishing: Cham, 2020; Communications in Computer and Information Science, pp. 52–58. doi:10.1007/978-3-030-64919-7_7.
42. Rescorla, R.A. A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and non-reinforcement. *Classical conditioning, Current research and theory* **1972**, *2*, 64–69. Publisher: Appleton-Century-Crofts.
43. Frässle, S.; Aponte, E.A.; Bollmann, S.; Brodersen, K.H.; Do, C.T.; Harrison, O.K.; Harrison, S.J.; Heinzle, J.; Iglesias, S.; Kasper, L.; Lomakina, E.I.; Mathys, C.; Müller-Schrader, M.; Pereira, I.; Petzschner, F.H.; Raman, S.; Schöbi, D.; Toussaint, B.; Weber, L.A.; Yao, Y.; Stephan, K.E. TAPAS: An Open-Source Software Package

for Translational Neuromodeling and Computational Psychiatry. *Frontiers in Psychiatry* **2021**, *12*. Publisher: Frontiers, doi:10.3389/fpsyt.2021.680811.

44. Smith, R.; Schwartenbeck, P.; Parr, T.; Friston, K.J. An Active Inference Approach to Modeling Structure Learning: Concept Learning as an Example Case. *Frontiers in Computational Neuroscience* **2020**, *14*. Publisher: Frontiers, doi:10.3389/fncom.2020.00041.
45. Neacsu, V.; Mirza, M.B.; Adams, R.A.; Friston, K.J. Structure learning enhances concept formation in synthetic Active Inference agents. *PLOS ONE* **2022**, *17*. Publisher: PLOS, doi:10.1371/journal.pone.0277199.
46. Xia, R.; Zhang, Y.; Liu, X.; Yang, B. A survey of sum–product networks structural learning. *Neural Networks* **2023**, *164*, 645–666. doi:10.1016/j.neunet.2023.05.010.
47. Friston, K.J.; Da Costa, L.; Tschantz, A.; Kiefer, A.; Salvatori, T.; Neacsu, V.; Koudahl, M.; Heins, C.; Sajid, N.; Markovic, D.; Parr, T.; Verbelen, T.; Buckley, C.L. Supervised structure learning, 2023. arXiv:2311.10300 [cs], doi:10.48550/arXiv.2311.10300.
48. Serrano, S.A.; Santiago, E.; Martinez-Carranza, J.; Morales, E.F.; Sucar, L.E. Knowledge-Based Hierarchical POMDPs for Task Planning. *Journal of Intelligent & Robotic Systems* **2021**, *101*, 82. doi:10.1007/s10846-021-01348-8.
49. Friston, K.J.; Rosch, R.; Parr, T.; Price, C.; Bowman, H. Deep temporal models and active inference. *Neuroscience & Biobehavioral Reviews* **2017**, *77*, 388–402. doi:10.1016/j.neubiorev.2017.04.009.
50. Friston, K.; Mattout, J.; Trujillo-Barreto, N.; Ashburner, J.; Penny, W. Variational free energy and the Laplace approximation. *NeuroImage* **2007**, *34*, 220–234. doi:10.1016/j.neuroimage.2006.08.035.
51. Friston, K.J. Variational filtering. *NeuroImage* **2008**, *41*, 747–766. doi:10.1016/j.neuroimage.2008.03.017.
52. Tu, S. The Dirichlet-Multinomial and Dirichlet-Categorical models for Bayesian inference **2014**.
53. Smith, R.; Badcock, P.; Friston, K.J. Recent advances in the application of predictive coding and active inference models within clinical neuroscience. *Psychiatry and Clinical Neurosciences* **2021**, *75*, 3–13. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/pcn.13138, doi:10.1111/pcn.13138.
54. Deacon, R.M.J.; Rawlins, J.N.P. T-maze alternation in the rodent. *Nature Protocols* **2006**, *1*, 7–12. Publisher: Nature Publishing Group, doi:10.1038/nprot.2006.2.
55. Lin, M.H.; Liran, O.; Bauer, N.; Baker, T.E. Power dynamics of theta oscillations during goal-directed navigation in freely moving humans: A mobile EEG-virtual reality T-maze study, 2021. Pages: 2021.10.05.463245 Section: New Results, doi:10.1101/2021.10.05.463245.
56. Ghafari, M.; Falsafi, S.K.; Szodorai, E.; Kim, E.J.; Li, L.; Höger, H.; Berger, J.; Fuchs, K.; Sieghart, W.; Lubec, G. Formation of GABAA receptor complexes containing 1 and 5 subunits is paralleling a multiple T-maze learning task in mice. *Brain Structure and Function* **2017**, *222*, 549–561. doi:10.1007/s00429-016-1233-x.
57. Sharma, S.; Rakoczy, S.; Brown-Borg, H. Assessment of spatial memory in mice. *Life Sciences* **2010**, *87*, 521–536. doi:10.1016/j.lfs.2010.09.004.
58. Friston, K.; Da Costa, L.; Hafner, D.; Hesp, C.; Parr, T. Sophisticated Inference. *arXiv:2006.04120 [cs, q-bio]* **2020**. arXiv: 2006.04120.
59. Vehtari, A.; Simpson, D.; Gelman, A.; Yao, Y.; Gabry, J. Pareto Smoothed Importance Sampling, 2024. arXiv:1507.02646, doi:10.48550/arXiv.1507.02646.
60. Aponte, E.A.; Yao, Y.; Raman, S.; Frässle, S.; Heinzle, J.; Penny, W.D.; Stephan, K.E. An introduction to thermodynamic integration and application to dynamic causal models. *Cognitive Neurodynamics* **2021**, *16*, 1. doi:10.1007/s11571-021-09696-9.
61. Daunizeau, J.; Ouden, H.E.M.d.; Pessiglione, M.; Kiebel, S.J.; Stephan, K.E.; Friston, K.J. Observing the Observer (I): Meta-Bayesian Models of Learning and Decision-Making. *PLOS ONE* **2010**, *5*, e15554. Publisher: Public Library of Science, doi:10.1371/journal.pone.0015554.
62. Wilson, R.C.; Collins, A.G. Ten simple rules for the computational modeling of behavioral data. *eLife* **2019**, *8*, e49547. Publisher: eLife Sciences Publications, Ltd, doi:10.7554/eLife.49547.
63. Hess, A.J.; Iglesias, S.; Köchli, L.; Marino, S.; Müller-Schrader, M.; Rigoux, L.; Mathys, C.; Harrison, O.K.; Heinzle, J.; Frässle, S.; Stephan, K.E. Bayesian Workflow for Generative Modeling in Computational Psychiatry, 2024. Pages: 2024.02.19.581001 Section: New Results, doi:10.1101/2024.02.19.581001.
64. TuringLang/ParetoSmooth.jl, 2024. original-date: 2021-06-04T22:08:47Z.
65. Palmeri, T.J.; Love, B.C.; Turner, B.M. Model-based cognitive neuroscience. *Journal of mathematical psychology* **2017**, *76*, 59–64. doi:10.1016/j.jmp.2016.10.010.

66. Kagan, B.J.; Kitchen, A.C.; Tran, N.T.; Habibollahi, F.; Khajehnejad, M.; Parker, B.J.; Bhat, A.; Rollo, B.; Razi, A.; Friston, K.J. In vitro neurons learn and exhibit sentience when embodied in a simulated game-world. *Neuron* **2022**, *110*, 3952–3969.e8. doi:10.1016/j.neuron.2022.09.001.
67. Waade, P.T.; Olesen, C.L.; Laursen, J.E.; Nehrer, S.W.; Heins, C.; Friston, K.; Mathys, C. As One and Many: Relating Individual and Emergent Group-Level Generative Models in Active Inference, 2024. doi:10.20944/preprints202410.1895.v1.