

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, 8000 Aarhus, Denmark; 202204724@post.au.dk (S.W.N.); 202204836@post.au.dk (J.E.L.)

² Department of Collective Behaviour, Max Planck Institute of Animal Behavior, D-78457 Konstanz, Germany

³ VERSES Research Lab, Los Angeles, CA 90016, USA; k.friston@ucl.ac.uk

⁴ Queen Square Institute of Neurology, University College London, London WC1N 3BG, UK

⁵ Interacting Minds Centre, Aarhus University, 8000 Aarhus, Denmark; chmathys@cas.au.dk (C.M.); ptw@cas.au.dk (P.T.W.)

* Correspondence: cheins@ab.mpg.de

† These authors contributed equally to this work.

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 behavioural modelling, as it is used in computational psychiatry, cognitive science and neuroscience. This means that POMDP active inference models can now be easily fit to empirically observed behaviour 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 performing a model comparison.



Academic Editor: Astero Provata

Received: 25 October 2024

Revised: 2 January 2025

Accepted: 7 January 2025

Published: 12 January 2025

Citation: Nehrer, S.W.; Ehrenreich Laursen, J.; Heins, C.; Friston, K.; Mathys, C.; Thestrup Waade, P. Introducing ActiveInference.jl: A Julia Library for Simulation and Parameter Estimation with Active Inference Models. *Entropy* **2025**, *27*, 62. <https://doi.org/10.3390/e27010062>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: active inference; free energy principle; predictive processing; Markov decision process; cognitive modelling; Julia

PACS: 87.15.Aa

MSC: 91-08

JEL Classification: C63

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 fit such models to empirically observed behaviour. AIF [1–3] is a generally applicable formal framework for understanding and simulating intelligent behaviour that is based in neurobiology and first principles from statistical physics [4–8]. AIF treats action and perception as unified under a joint imperative: to minimise the variational free energy (VFE), which quantifies how well the agent's internal generative model explains incoming sensory observations. It is an upper bound on the surprise from sensory observations, making AIF formally related to prediction error

minimisation [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 was 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 are 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 and learning the structure of the generative model. Since the requisite inference schemes come pre-specified, the main task in AIF modelling becomes specifying 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 focus here 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] was created for Python [24], as well as a performance-oriented package cpp-AIF [25] for C++ [26] that 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 the mechanisms underlying variations in behaviour or to investigate the differences between (for example, clinical) populations. Smith and colleagues [33] provided 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 that 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 performed 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 with 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 fully 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.

Core Concepts	
AIF	Active inference is a formal framework for modelling behaviour and cognition. Perception and action are cast as minimising free energy—the <i>VFE</i> and <i>EFE</i> , respectively—given a generative model of the environment.
VFE	The variational free energy \mathcal{F} quantifies how well a generative model explains incoming sensory observations. It can be rewritten as the negative log model evidence (called surprise) upper-bounded by the divergence from the optimal posterior $p(s o)$. Perception as inference is accomplished by selecting the approximate posterior $q(s)$ with the lowest associated <i>VFE</i> .
$\mathcal{F}[q(s), o] \triangleq D_{\text{KL}}[q(s)\ p(o, s)] = \underbrace{D_{\text{KL}}[q(s)\ p(s o)]}_{\text{Divergence}} - \underbrace{\ln p(o)}_{\text{Surprise}}$	
EFE	The expected free energy G quantifies the expected future free energy under an action policy π . It consists of an information gain term and a pragmatic value term that provide a natural balance between exploratory and goal-seeking behaviour. Action as inference is accomplished by selecting the action policy with the lowest associated <i>EFE</i> .
$G_\pi = -\underbrace{\mathbb{E}_{q(\tilde{o}, \tilde{s} \pi)} [\ln q(\tilde{s} \tilde{o}, \pi) - \ln q(\tilde{s} \pi)]}_{\text{Information gain}} - \underbrace{\mathbb{E}_{q(\tilde{o} \pi)} [\ln p(\tilde{o} C)]}_{\text{Pragmatic value}}$	
Generative model	The generative model is an agent's formal assumptions about the structure and dynamics of its environment, based on which perceptual and active inferences are carried out. Many types of generative models exist that are suitable for different environments and tasks.
POMDP	The Partially Observable Markov Decision Process is a type of flexible generative model that is widely used in the AIF literature. In discrete time and usually a discrete state space, this model type is parametrised to fit a given task by a set matrices containing probability distributions.

2. Active Inference with POMDPs

In this section, we 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 the 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. The *VFE* is the free energy associated with a given sensory observation and is resolved perceptually by updating beliefs about the environment. The *EFE* is the free energy that is expected in the future, contingent on a given policy or course of action. Choosing action policies associated with a low *EFE* lead to reducing uncertainty about the environment, as well as making preferred observations more likely.

2.1. POMDPs in Active Inference

In AIF, the POMDP is one of the most common families of generative models used to make inferences about the environment. It is a Markovian discrete state-space model, where employing it means representing the environment and observations as inhabiting one among a set of possible (possibly multidimensional) states, and that the changes in these states can only 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 POMDPs and other generative models in general, both perception and action are cast as Bayesian inferences (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 a successful inference, and that the generative model will therefore often 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], see Figure 1. Together, these parametrise the agent's prior beliefs about the prior probability of different states in the environment, how states of the environment change and how they generate observations. Typically, they will be vectors, matrices or tensors; however, henceforth we 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 inferences, 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 time step, 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 pick 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), its preferences are weak and it will prioritise collecting information over realising its 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. **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 is covered later in this section. After policies have been selected for a given time step, they are marginalised to calculate the probabilities of taking each possible action in 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, here we focus specifically on the POMDP-based generative models often used in the AIF literature. However, the basic steps when performing AIF—perception, action and learning—remain the same across generative models. In the remainder of this section, we describe each of these three steps in turn.

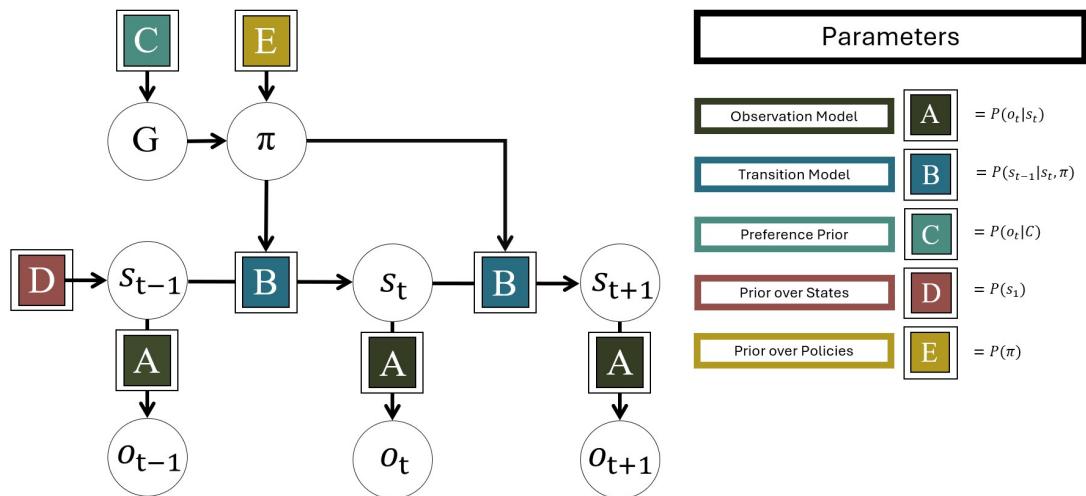


Figure 1. Depiction of a POMDP generative model. This encodes the agent’s expectations about how the state s of the environment changes over time t , and how it generates observation o at each time step. **A**, also called the observation model, describes how environmental states give rise to observations. **B**, also called the transition model, describes how environmental states change over time, depending on action u (called policy 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 an optimal inference about state s of the environment based on observation o . This is performed 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]:

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

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 use henceforth. In the following, for notational simplicity, we also 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 the minimisation of the *VFE*. First, an arbitrary probability distribution over environmental states $q(s)$, an approximate posterior that is used to approximate the exact posterior, is introduced. 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:

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

It is a property of the KL divergence that the two distributions are identical when $D_{\text{KL}}[q(s) \| p(s|o)] = 0$. 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 here we use the joint likelihood $p(o,s)$ notation, fraction rule $\frac{a}{b} = \frac{a}{b} * c$ and logarithmic rule $\ln(a * b) = \ln a + \ln b$:

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

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):

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

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 $p(o)$:

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

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

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

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

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

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

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

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

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

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

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

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 performed in variational inference, introduce a mean-field approximation [35] so that the joint approximate posterior $q(s_t)$ factorises across time steps $t \in T$ and hidden state factors $f \in F$:

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

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

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

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

which is the value we intend to minimise. Various methods exist for minimising the VFE; the one used in `pymdp`, from which we drew much of our inspiration, is Coordinate Ascent Variational Inference (CAVI) [35], where the fixed points of the VFE are 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 minimises the time-dependent VFE \mathcal{F}_t , and therefore optimises for the time-specific variational posterior $q(s_t)$. To obtain 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]:

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

Solving for $q(s_t^f)$ yields

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

which leads us to the coordinate descent update equation:

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

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. By default, `ActiveInference` uses 10 iterations or stops when $\partial \mathcal{F}_t < 0.001$. This posterior then comprises the AIF agent's belief about the state of the environment, and therefore, its 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 minimised 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 minimised. As stated below, choosing actions that minimise the *EFE* leads to a natural balance between exploration and exploitation, ensuring preferences are realised and ambiguity about the environment is minimised. In AIF, policies π are sequences of actions u . The policy length (also called the 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), where 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 the states and observations that are expected future outcomes of actions with (\sim) . Additionally, we introduce a *preference prior* C that encodes the agent's preferences:

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

The expression above shows how minimising the *EFE* leads to a natural balance between information gathering and realising preferences. The first term on the right-hand side is the change in belief from the prior to the 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 ensures that observations are in accordance with the preference prior C .

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

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

Here, the first term on the right-hand side captures the expected entropy, or uncertainty, of the outcomes given the 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 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 performed 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 POMDPs.

The parameters that are subject to learning in POMDPs 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 parametrised by a set of concentration parameters θ :

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

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:

$$\theta_{t+1} = \omega * \theta_t + \eta * \chi_t \quad (20)$$

The updated value for the concentration parameter (θ_{t+1}) is found by adding the previous concentration parameter θ_t multiplied by 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 by a learning rate η . With this relatively simple update equation—which, in essence, amounts to just counting the 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 POMDPs, 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 `pymdp` [23], a Python library for implementing simulations of AIF in discrete state spaces. 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 `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::Vector{Array{T, N}},          # A-matrices
    B::Vector{Array{T, N}},          # B-matrices
    C::Vector{Array{Real}},          # C-matrices (optional)
    D::Vector{Vector{Real}},         # D-matrices (optional)
    E::Vector{T},                  # E-vector (optional)
    pA::Union{Vector{Array{T, N}}},  # Dirichlet priors for A-matrices (optional)
    pB::Union{Vector{Array{T, N}}},  # Dirichlet priors for B-matrices (optional)
    pD::Union{Vector{Array{Real}}},  # 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 default to uniform priors. **A**, **B**, **C**, **D** and **E** and their corresponding Dirichlet priors, in the cases 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 six environmental states, five possible observations and two possible actions. Here, we 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 5×6 matrix (six states and five observations). The **B** object is a one-dimensional vector containing a uniform $6 \times 6 \times 2$ array (six states and two 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 performed 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 at 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. Here, we 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. Here, we 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 about 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 parameters with the `update_parameters!` function. **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 parameters based on provided priors, pA, pB, and pD.
update_parameters!(aif)

```

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 performed by marginalising the policy probabilities to obtain 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 that implements 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 various behavioural models to data. Here, we show how to use it in conjunction with ActiveInference to make the simulation of AIF models easy and in a fully generalised 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 that 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! provides the agent with an observation, updates its beliefs and returns 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 the 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 the 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 the behaviour and belief updating of agents, ActionModels also makes it possible to fit models to data and perform parameter estimation. This is used in general to form better models and theories of mental processes, as well as to find 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 performed in fields like cognitive modelling 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 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 the 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 achieve this 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 and predictive checks. In this section, we outline how to use ActiveInference for simulation and model fitting in conjunction with ActionModels. In the following section, we show how to achieve 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 provide the necessary code to run this simulation. All code required to reproduce the example simulation can be found in an open source OSF repository osf.io/j3k5q/. This example was performed with the current version of ActiveInference.jl (0.1.1); 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 is also suitably represented in a discrete state space. Together, this makes it easily compatible with a POMDP-based AIF approach.

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 that indicates which of the locations is the reward location. Generally, the cue may be more or less informative; in this example, it always accurately reflected the reward conditions state (reward in the right or left arm). The reward location only provides a 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 cue 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 the agent to realise 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].

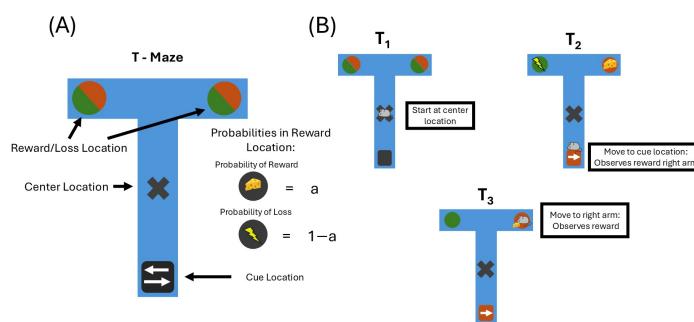


Figure 2. A depiction of the T-maze. (A) The full layout of the T-maze task, with the centre location, the cue location and the two reward conditions. (B) A three-step example of a T-maze trial. The agent (in this case, a mouse) starts at the centre location. In order to reduce the uncertainty regarding 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 the reward with some probability.

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

State Factor 1 (Location):	State Factor 2 (Reward Condition):
1. Centre location;	1. Reward condition right;
2. Right arm;	2. Reward condition left.
3. Left arm;	
4. Cue location.	

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

Modality 1 (Location):	Modality 2 (Reward):	Modality 3 (Cue):
1. Centre location;	1. No reward;	1. Cue right;
2. Right arm;	2. Reward;	2. Cue left.
3. Left arm;	3. Loss.	
4. Cue location.		

The first observation modality had an observation for each of the positions in the maze, and generally reflected the agent's actual position perfectly. The second modality was the agent's observation of a reward or loss, which depended on the reward condition and the position. The “no reward” observation was received whenever the agent was not occupying either of the arm locations, where it did not observe any reward or loss. The third modality was the observation of the cue, which depended on the reward modality so that the agent could use it to infer the correct reward condition. The action was one-dimensional with four options and was 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 now provide an example of using the package with a T-maze environment. First, we installed and loaded 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 loaded 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 proceeded to set up **A**, **B**, **C**, **D** and **E**. For this we used the `create_matrix_templates` helper function to set up the correct structure and then populate it. To start, we defined what went into the helper function. It took five arguments: the numbers 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 are made clear below when discussing populating the parameters.

The first three arguments should be specified as vectors of integers, containing the numbers of states and observations for each factor or modality. In our case, we had two factors: a location factor and a reward condition factor, which had four and two states, respectively. There were three modalities: one location modality with four observations, one reward modality with three observations and one cue modality with two observations. Finally, there were four possible actions for controlling the location factor, and only one possible action for the reward state factor. The policy length was specified as an integer, in our case 2, and we populated the template with zeros.

```

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

```

We started by defining **A**, or the observation model. In this example, since we allowed for **A** learning, **A** was not used directly, but we still defined 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 was the location observation. Here, there were four possible observations, mapped to the four location states and two reward condition states. This resulted in an **A** that was four location observations by four location states by two reward conditions, i.e., a $4 \times 4 \times 2$ 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 was the reward modality, which mapped the observations “no reward”, “reward” and “loss” onto the location states and reward conditions. For the second modality, we therefore had a tensor that was three reward observations by four location states by two reward conditions, i.e., a $3 \times 4 \times 2$ tensor. When the agent was at the centre and cue locations, we let the modality accurately expect the observation of “no reward” with certainty. For the two arm locations, we let the agent be agnostic regarding whether they provided rewards or losses. This was different from the true reward probabilities (see Figure 3), which the agent needed to learn over time. This was 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 ]
```

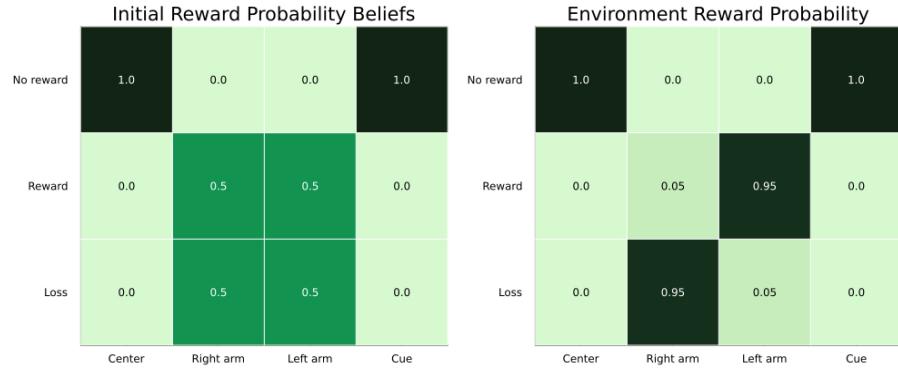


Figure 3. Reward probabilities for the four locations. The centre (column 1) and cue (column 4) locations always resulted in the “no reward” observation (row 1). The two arms (columns 3 and 4) resulted in either rewards (row 2) or losses (row 3), with some probability. Left: the agent’s agnostic starting beliefs about reward probabilities. Right: the true reward probabilities for the reward condition left arm, which the agent needed to learn over time. The amount of saturation of the green color represent the likelihood of a specific observation in a give state.

The third and last modality was the cue modality, which mapped the cue observations onto the location and reward condition states. This resulted in an **A** that was two cue observations by four location states by two reward conditions, i.e., a $2 \times 4 \times 2$ tensor. The observations in this modality were correctly assumed by the agent to truthfully reveal the current reward condition—i.e., whether the right or left arm was better—when standing at the cue location. We implemented this by giving each cue observation equal probabilities at all locations except the cue location, where there was 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 created all three modalities of **A**, we could continue to **B**, or the transition model. Each of the two state factors of **B**—the location factor and reward condition factor—needed to be defined separately. We started with the location factor, which contained the transition to and from four possible location states under four different actions: a $4 \times 4 \times 4$ tensor. The agent could control these states perfectly with its four movement actions, independently

of its current position. This was 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 was rather simple, as we let the agent correctly assume that the reward condition never changed (although one could have enabled **B** learning to let the agent learn the transition probabilities of the environment). There were two possible states for these state factors, so we implemented this as a 2×2 identity matrix:

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

Then, we built **C**, the prior preference over observations. **C** encoded preferences for each observation in each modality: the location, reward and cue. The relative value of the entry corresponding to a specific observation defined the preference; here, we set all values with neutral preferences as 0, and used positive values for preferences and negative values for aversions. The only observations over which the agent had preferences were in the reward modality, where the rewards (index 2) were preferred and losses (index 3) were disliked. The strength of the preference determined the explore–exploit balance of the agent; here, we use 3 as the 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
```

Then, we built **D**, which was the prior or initial belief over the environmental states. There was a 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 was at the centre location (index 1), and gave uniform priors for the reward condition:

```
# For the location state factor
D[1] = [1.0, 0.0, 0.0, 0.0] # Certain belief 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 established **E**, the prior over policies. This was a vector over all possible policies; with four possible actions and a policy length of 2, there were $4^2 = 16$ policies in total. Here, we gave 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)
```

Thus, the five matrices were defined (we note that the **C**, **D** and **E** matrices were set to agnostic by default if not defined by the user). All that was left was to set the Dirichlet prior for **A** learning, and set the 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. Here, we used 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 defined the settings of the agent (other characteristics of the agent that were not estimable parameters). Here, we set the policy length to 2, which let the agent use expected information gain for both the state and parameter info gain for its actions, and specify that it was specifically the second modality of **A** that was 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 could then 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

Since the environment and agent were set up, we could proceed to simulate the behaviour of the agent in the environment. We created a for loop, where the agent received an observation, made inferences about the environment, updated A , inferred policies and sampled 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_parameters!(aif_agent)

    # 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 started by moving to the cue location, and then proceeded to move to the left arm repeatedly. The main objects of learning here were the reward condition state and the A parameters for rewards under the two reward conditions. After observing the cue, the agent updated its belief (correctly) to be certain of being in the left reward condition (Figure 4). Over time, the agent learned the correct probability of receiving rewards in the left arm (0.94 versus a correct 0.95). It did not learn the probabilities for the right arm; this was because it never moved to that location, having already learnt that the left arm was 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 .

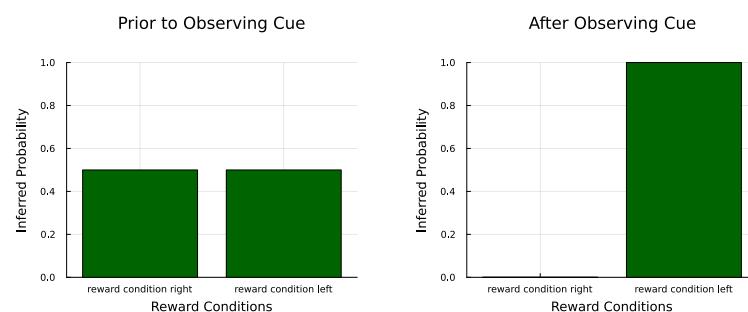


Figure 4. State inference for the reward condition. The agent's beliefs about the reward condition changed from agnostic (**left**) to certain that it was the left reward condition (**right**) after observing the cue.

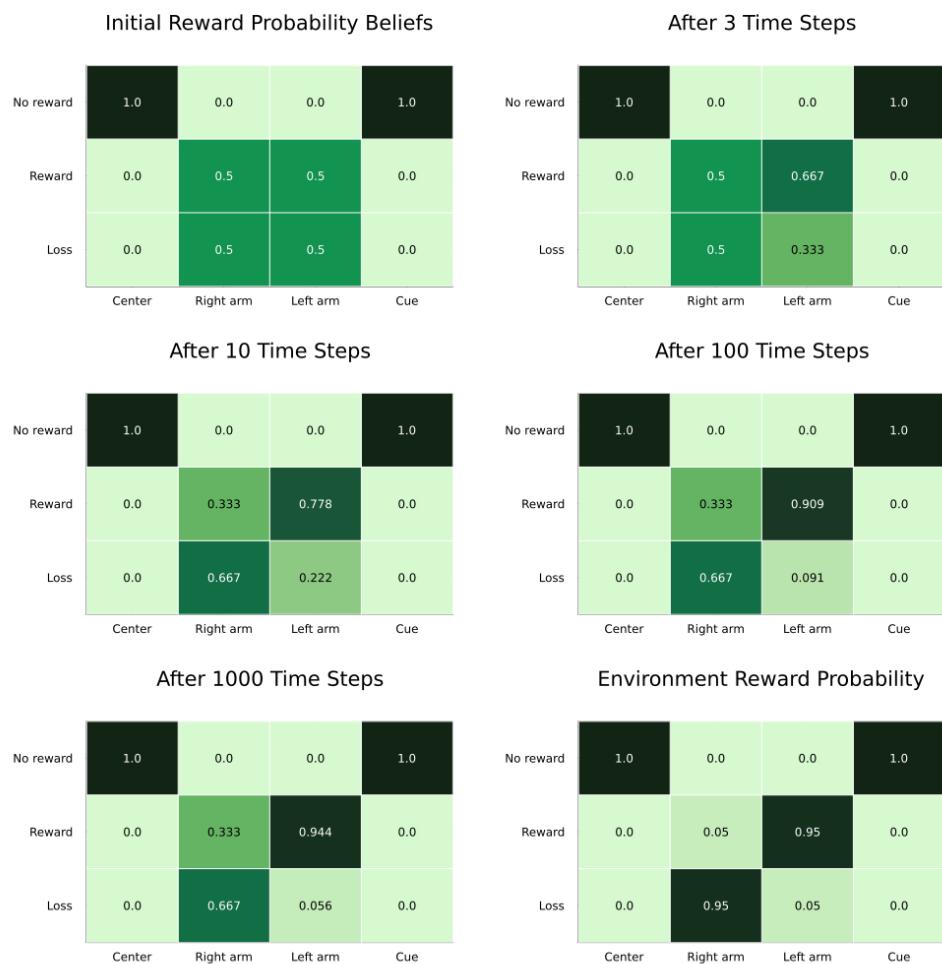


Figure 5. A learning for the actual reward condition (reward condition left). The agent correctly learned the probability of receiving rewards in the rewarding arm. It did not learn the probabilities of the non-rewarding arm since it did not explore that option. The color grading signifies the likelihood of an observation being generated by a specific state. The more saturated the color, the higher the likelihood.

4.3. Fitting the Model to the Data

Simulations are useful for a variety of purposes, like exploring the consequences of different priors and parameters and establishing the face validity of hypothetical mechanisms underlying behavioural phenomena. However, we often want to use models to make inferences about specific observed phenomena, like the 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 the most likely given some observed behaviour of a participant. This is often performed 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 inferences (i.e., comparing parameters of the same model fitted to different data, like two groups of subjects). When performing a 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].

Here, we demonstrated 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 used the simulation method shown in the previous section to produce a synthetic dataset with known parameter values for each agent (in practice, these are often participants in an experiment), here with a focus on estimating the α parameter. We then used MCMC methods to estimate the parameters for each agent and compared the estimated values with the correct values. Here, we simulated two groups of five synthetic subjects agents with different α values (the parameters for the first group were sampled from a Gaussian distribution with mean = 8 and SD = 2, and the second group with with mean = 24 and SD = 2). Each agent interacted with the T-maze environment for 300 time steps. We produced the following data frame, containing the data of each of the agents: their observations, actions and an identifier, a format suitable for cognitive and behavioural 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 used ActionModels to fit the AIF model created above to each of the agents in the dataset. We began by initialising 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 wanted to estimate: the α action precision. As an example, we chose 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 instantiated 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 used the `fit_model` function to perform (parallelised) 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 contained the `chains` object with 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 could plot the posteriors and chains, which is often performed 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, :])

```

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

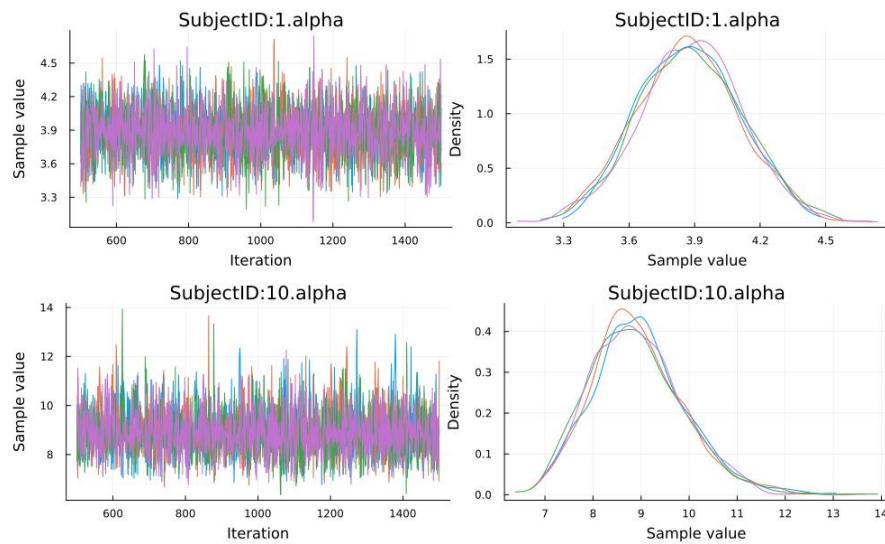


Figure 6. Chain traces. Each color signifies an individual chain.

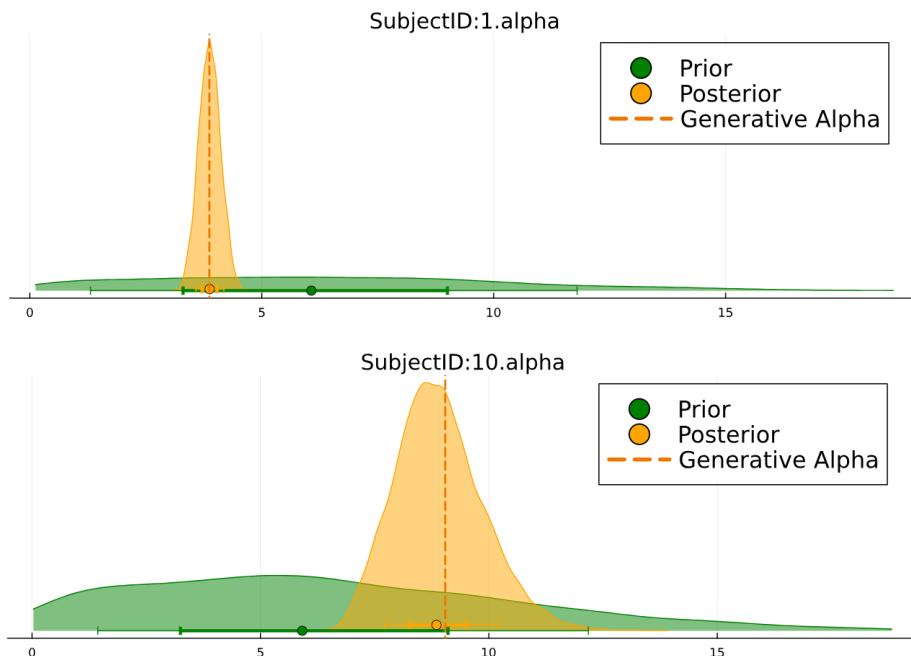


Figure 7. Posterior estimates of the α parameter plotted against the prior for two synthetic subjects, one from each group.

```

# 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")

```

We then, as is often the case in computational psychiatry, wanted to compare the distributions of parameter values between the two groups. We extracted the median of the estimated posteriors for each subject and plotted them against the value used to generate

the behaviour (Figure 8). We saw that the estimation successfully captured the difference between the two groups, and that the α parameter recovered fairly well. Note that the ability to recover parameters depends on the specific model and task, as well as on the specific values of the parameters (when α is very high, for example, the behaviour becomes essentially deterministic; further increases in α would then not have any effect on the 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 (also known as the model evidence); because the model evidence includes a complexity term, the parameter recovery will often recover parameters that provide a simpler explanation for the data relative to the parameters used to generate these data.

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

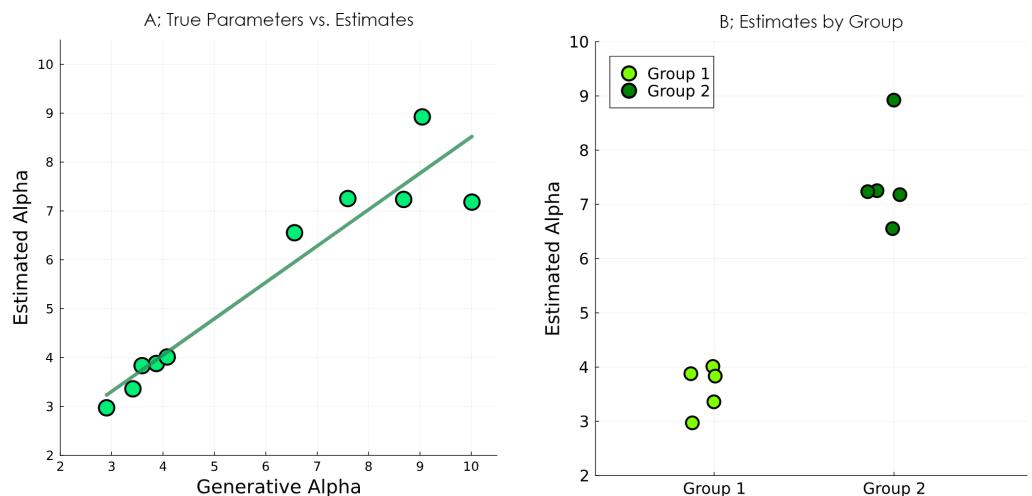


Figure 8. Results of the parameter recovery study. (A) Estimated parameter values plotted against the values used to generate the data. (B) Parameter estimates split by the two groups from which the parameter values of synthetic subjects were sampled.

Finally, there are various metrics for model comparison that might be calculated, as implemented by various software packages. Here for demonstration, we calculated 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 introduce `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 an 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 investigat-

ing the temporal dynamics of belief changes in experimental participants. Dynamic belief trajectories can then be related to other (for example, physiological) measures, as is 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.

Since they implement full approximate Bayesian inferences, AIF models are computationally more demanding than many approaches traditionally used in cognitive and agent-based modelling, in particular when the dimensionality of the generative model is large. This means that models with highly multidimensional or complex behaviour and large numbers of agents can be computationally infeasible to implement, especially given the additional computational demands introduced by fitting these models to empirical data. Avenues for addressing this implicit scaling problem were proposed in the context of machine learning applications [68,69], and with the use of simplifying assumptions—the use of which are ubiquitous in computational modelling—AIF has been used to model multi-agent phenomena, such as opinion dynamics [15,70], coordinated foraging [71] and fish school movements [12]. It remains to be explored how AIF models can be applied to highly complex natural phenomena, such as a concrete election, which underscores the need for efficient but flexible and accessible software tools in the field.

There are many ways in which ActiveInference can be improved. It would be useful to extend the set of dynamic belief 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 \mathbf{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 with 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 POMDPs. 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 POMDPs, including other universal models, like generalised filtering [17] and Hierarchical Gaussian Filter models [41], as well as custom

generative models, or even (deep learning-based) amortised inference models. These various extensions could provide valuable tools for using AIF models in both theoretical and applied research.

Author Contributions: Conceptualisation, 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.; visualisation, S.W.N. and J.E.L.; supervision, C.M. and P.T.W.; project administration, P.T.W. All authors read and agreed to the published version of this manuscript.

Funding: C.M. acknowledges funding from Aarhus Universitets Forskningsfonds (grant no. AUUFF-E-2019-7-10) and from the Carlsberg Foundation (grant no. CF21-0439). K.F. is supported by funding from the Wellcome Trust (Ref: 226793/Z/22/Z).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

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

Conflicts of Interest: Authors Conor Heins and Karl Friston were employed by the company VERSES Research Lab. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. The funders had no role in the design of this study; in the collection, analyses or interpretation of data; in the writing of this 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
FEF	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: Cambridge, MA, USA, 2022. [[CrossRef](#)]
2. Friston, K.; FitzGerald, T.; Rigoli, F.; Schwartenbeck, P.; O'Doherty, J.; Pezzulo, G. Active inference and learning. *Neurosci. Biobehav. Rev.* **2016**, *68*, 862–879. [[CrossRef](#)]
3. Friston, K.; FitzGerald, T.; Rigoli, F.; Schwartenbeck, P.; Pezzulo, G. Active inference: A process theory. *Neural Comput.* **2017**, *29*, 1–49. [[CrossRef](#)]
4. Friston, K.J.; Stephan, K.E. Free-energy and the brain. *Synthese* **2007**, *159*, 417–458. [[CrossRef](#)] [[PubMed](#)]
5. Friston, K. The free-energy principle: A unified brain theory? *Nat. Rev. Neurosci.* **2010**, *11*, 127–138. [[CrossRef](#)] [[PubMed](#)]
6. Friston, K. The free-energy principle: A rough guide to the brain? *Trends Cogn. Sci.* **2009**, *13*, 293–301. [[CrossRef](#)] [[PubMed](#)]
7. Friston, K. A free energy principle for a particular physics. *arXiv* **2019**, arXiv:1906.10184. [[CrossRef](#)]
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. *Phys. Rep.* **2023**, *1024*, 1–29. [[CrossRef](#)]
9. Friston, K.; Kiebel, S. Predictive coding under the free-energy principle. *Philos. Trans. R. Soc. B Biol. Sci.* **2009**, *364*, 1211–1221. [[CrossRef](#)] [[PubMed](#)]
10. Karl, F. A Free Energy Principle for Biological Systems. *Entropy* **2012**, *14*, 2100–2121. [[CrossRef](#)]
11. Corcoran, A.W.; Pezzulo, G.; Hohwy, J. From allostatic agents to counterfactual cognisers: Active inference, biological regulation, and the origins of cognition. *Biol. Philos.* **2020**, *35*, 32. [[CrossRef](#)]
12. Heins, C.; Millidge, B.; Da Costa, L.; Mann, R.P.; Friston, K.J.; Couzin, I.D. Collective behavior from surprise minimization. *Proc. Natl. Acad. Sci. USA* **2024**, *121*, e2320239121. [[CrossRef](#)] [[PubMed](#)]

13. Patzelt, E.H.; Hartley, C.A.; Gershman, S.J. Computational Phenotyping: Using Models to Understand Individual Differences in Personality, Development, and Mental Illness. *Personal. Neurosci.* **2018**, *1*, e18. [[CrossRef](#)] [[PubMed](#)]
14. Schwartenbeck, P.; Friston, K. Computational Phenotyping in Psychiatry: A Worked Example. *eNeuro* **2016**, *3*, ENEURO.0049-16.2016. [[CrossRef](#)]
15. Albaracín, M.; Demekas, D.; Ramstead, M.J.D.; Heins, C. Epistemic Communities under Active Inference. *Entropy* **2022**, *24*, 476. [[CrossRef](#)] [[PubMed](#)]
16. Lanillos, P.; Meo, C.; Pezzato, C.; Meera, A.A.; Baioumy, M.; Ohata, W.; Tschantz, A.; Millidge, B.; Wisse, M.; Buckley, C.L.; et al. Active Inference in Robotics and Artificial Agents: Survey and Challenges. *arXiv* **2021**, arXiv:2112.01871. [[CrossRef](#)]
17. Friston, K.; Stephan, K.; Li, B.; Daunizeau, J. Generalised Filtering. *Math. Probl. Eng.* **2010**, *2010*, 621670. [[CrossRef](#)]
18. Waade, P.T.; Mikus, N.; Mathys, C. Inferring in Circles: Active Inference in Continuous State Space Using Hierarchical Gaussian Filtering of Sufficient Statistics. In *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., et al., Eds.; Communications in Computer and Information Science; Springer: Cham, Switzerland, 2021; pp. 810–818. [[CrossRef](#)]
19. Weber, L.A.; Waade, P.T.; Legrand, N.; Møller, A.H.; Stephan, K.E.; Mathys, C. The generalized Hierarchical Gaussian Filter. *arXiv* **2023**, arXiv:2305.10937. [[CrossRef](#)]
20. Friston, K.J.; Trujillo-Barreto, N.; Daunizeau, J. DEM: A variational treatment of dynamic systems. *NeuroImage* **2008**, *41*, 849–885. [[CrossRef](#)]
21. MATLAB, Inc. MATLAB, version: 9.13.0 (R2022b); MATLAB, Inc.: Natick, MA, USA, 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: Amsterdam, The Netherlands, 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. *J. Open Source Softw.* **2022**, *7*, 4098. [[CrossRef](#)]
24. Rossum, G.v.; Drake, F.L. *The Python Language Reference*, Release 3.0.1 [repr.] ed.; Number Pt. 2 in Python Documentation Manual; van Rossum, G., Drake, F.L., Eds.; Python Software Foundation: Hampton, NH, USA, 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. [[CrossRef](#)]
26. Josuttis, N.M. *The C++ Standard Library: A Tutorial and Reference*; Addison-Wesley: Boston, MA, USA, 2012; Google-Books-ID: 9DEJKhasp7gC.
27. Bagaev, D.; Podusenko, A.; Vries, B.d. RxInfer: A Julia package for reactive real-time Bayesian inference. *J. Open Source Softw.* **2023**, *8*, 5161. [[CrossRef](#)]
28. Bezanson, J.; Karpinski, S.; Shah, V.; Edelman, A. Julia Language Documentation. 2016. Available online: <https://readthedocs.org/projects/julia-wf/downloads/pdf/stable/> (accessed on 26 May 2024).
29. van de Laar, T.W.; de Vries, B. Simulating Active Inference Processes by Message Passing. *Front. Robot. AI* **2019**, *6*, 20. [[CrossRef](#)]
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. *Stud. Undergrad. Res. E-J.* **2019**, *5*, 1–5. [[CrossRef](#)]
31. van de Laar, T.; Şenöz, İ.; Özçelikkale, A.; Wyneersch, H. Chance-Constrained Active Inference. *Neural Comput.* **2021**, *33*, 2710–2735. [[CrossRef](#)]
32. Busemeyer, J.R.; Diederich, A. *Cognitive Modeling*; SAGE: Thousand Oaks, CA, USA, 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. *J. Math. Psychol.* **2022**, *107*, 102632. [[CrossRef](#)] [[PubMed](#)]
34. Lee, M.D.; Wagenmakers, E.J. *Bayesian Cognitive Modeling: A Practical Course*, 1st ed.; Cambridge University Press: Cambridge, UK, 2014. [[CrossRef](#)]
35. Blei, D.M.; Kucukelbir, A.; McAuliffe, J.D. Variational Inference: A Review for Statisticians. *J. Am. Stat. Assoc.* **2017**, *112*, 859–877. [[CrossRef](#)]
36. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization, 2004, CGO 2004, Palo Alto, CA, USA, 20–24 March 2004; pp. 75–86. [[CrossRef](#)]
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. *J. Stat. Softw.* **2017**, *76*, 1–32. [[CrossRef](#)] [[PubMed](#)]
39. Ge, H.; Xu, K.; Ghahramani, Z. Turing: A Language for Flexible Probabilistic Inference. In Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (PMLR), Playa Blanca, Lanzarote, 9–11 April 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. Available online: <https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://jmlr.org/papers/volume15/hoffman14a/hoffman14a.pdf&ved=2ahUKEwjPzP32seCKAxXLENAFHRziJ-0QFnoECBcQAQ&usg=AQvVaw3adSH1tmvDJ1-6CEuitKet> (accessed on 17 May 2024).
41. Mathys, C.; Weber, L. Hierarchical Gaussian Filtering of Sufficient Statistic Time Series for Active Inference. In *Active Inference*; Verbelen, T., Lanillos, P., Buckley, C.L., De Boom, C., Eds.; Communications in Computer and Information Science; Springer: Cham, Switzerland, 2020; pp. 52–58. [CrossRef]
42. Rescorla, R.A. A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and non-reinforcement. *Class. Cond. Curr. Res. Theory* **1972**, *2*, 64–69.
43. Frässle, S.; Aponte, E.A.; Bollmann, S.; Brodersen, K.H.; Do, C.T.; Harrison, O.K.; Harrison, S.J.; Heinze, J.; Iglesias, S.; Kasper, L.; et al. TAPAS: An Open-Source Software Package for Translational Neuromodeling and Computational Psychiatry. *Front. Psychiatry* **2021**, *12*, 680811. [CrossRef]
44. Smith, R.; Schwartenbeck, P.; Parr, T.; Friston, K.J. An Active Inference Approach to Modeling Structure Learning: Concept Learning as an Example Case. *Front. Comput. Neurosci.* **2020**, *14*, 41. [CrossRef] [PubMed]
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*, e0277199. [CrossRef] [PubMed]
46. Xia, R.; Zhang, Y.; Liu, X.; Yang, B. A survey of sum-product networks structural learning. *Neural Netw.* **2023**, *164*, 645–666. [CrossRef] [PubMed]
47. Friston, K.J.; Da Costa, L.; Tschantz, A.; Kiefer, A.; Salvatori, T.; Neacsu, V.; Koudahl, M.; Heins, C.; Sajid, N.; Markovic, D.; et al. Supervised structure learning. *arXiv* **2023**, arXiv:2311.10300. [CrossRef] [PubMed]
48. Serrano, S.A.; Santiago, E.; Martinez-Carranza, J.; Morales, E.F.; Sucar, L.E. Knowledge-Based Hierarchical POMDPs for Task Planning. *J. Intell. Robot. Syst.* **2021**, *101*, 82. [CrossRef]
49. Friston, K.J.; Rosch, R.; Parr, T.; Price, C.; Bowman, H. Deep temporal models and active inference. *Neurosci. Biobehav. Rev.* **2017**, *77*, 388–402. [CrossRef] [PubMed]
50. Friston, K.; Mattout, J.; Trujillo-Barreto, N.; Ashburner, J.; Penny, W. Variational free energy and the Laplace approximation. *NeuroImage* **2007**, *34*, 220–234. [CrossRef] [PubMed]
51. Friston, K.J. Variational filtering. *NeuroImage* **2008**, *41*, 747–766. [CrossRef]
52. Tu, S. The Dirichlet-Multinomial and Dirichlet-Categorical Models for Bayesian Inference. 2014. Available online: <https://people.csail.mit.edu/stephentu/writeups/dirichlet-conjugate-prior.pdf> (accessed on 21 June 2024).
53. Smith, R.; Badcock, P.; Friston, K.J. Recent advances in the application of predictive coding and active inference models within clinical neuroscience. *Psychiatry Clin. Neurosci.* **2021**, *75*, 3–13. [CrossRef]
54. Deacon, R.M.J.; Rawlins, J.N.P. T-maze alternation in the rodent. *Nat. Protoc.* **2006**, *1*, 7–12. [CrossRef]
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. *bioRxiv* **2021**, bioRxiv:2021.10.05.463245. [CrossRef]
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 GABA_A receptor complexes containing $\alpha 1$ and $\alpha 5$ subunits is paralleling a multiple T-maze learning task in mice. *Brain Struct. Funct.* **2017**, *222*, 549–561. [CrossRef]
57. Sharma, S.; Rakoczy, S.; Brown-Borg, H. Assessment of spatial memory in mice. *Life Sci.* **2010**, *87*, 521–536. [CrossRef]
58. Friston, K.; Da Costa, L.; Hafner, D.; Hesp, C.; Parr, T. Sophisticated Inference. *arXiv* **2020**, arXiv:2006.04120. [CrossRef] [PubMed]
59. Vehtari, A.; Simpson, D.; Gelman, A.; Yao, Y.; Gabry, J. Pareto Smoothed Importance Sampling. *arXiv* **2024**, arXiv:1507.02646. [CrossRef]
60. Aponte, E.A.; Yao, Y.; Raman, S.; Frässle, S.; Heinze, J.; Penny, W.D.; Stephan, K.E. An introduction to thermodynamic integration and application to dynamic causal models. *Cogn. Neurodyn.* **2021**, *16*, 1. [CrossRef] [PubMed]
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. [CrossRef] [PubMed]
62. Wilson, R.C.; Collins, A.G. Ten simple rules for the computational modeling of behavioral data. *eLife* **2019**, *8*, e49547. [CrossRef]
63. Hess, A.J.; Iglesias, S.; Köchli, L.; Marino, S.; Müller-Schrader, M.; Rigoux, L.; Mathys, C.; Harrison, O.K.; Heinze, J.; Frässle, S.; et al. Bayesian Workflow for Generative Modeling in Computational Psychiatry. *bioRxiv* **2024**, bioRxiv:2024.02.19.581001. [CrossRef]
64. TuringLang/ParetoSmooth.jl. 2024. Available online: <https://github.com/TuringLang/ParetoSmooth.jl> (accessed on 25 October 2024).
65. Palmeri, T.J.; Love, B.C.; Turner, B.M. Model-based cognitive neuroscience. *J. Math. Psychol.* **2017**, *76*, 59–64. [CrossRef] [PubMed]
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. [CrossRef]
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. *Preprints* **2024**, 2024101895. [CrossRef]

68. Tschantz, A.; Baltieri, M.; Seth, A.K.; Buckley, C.L. Scaling active inference. *arXiv* **2019**, arXiv:1911.10601. [[CrossRef](#)]
69. Friston, K.; Heins, C.; Verbelen, T.; Costa, L.D.; Salvatori, T.; Markovic, D.; Tschantz, A.; Koudahl, M.; Buckley, C.; Parr, T. From pixels to planning: Scale-free active inference. *arXiv* **2024**, arXiv:2407.20292. [[CrossRef](#)]
70. Kastel, N.; Hesp, C.; Ridderinkhof, K.R.; Friston, K.J. Small steps for mankind: Modeling the emergence of cumulative culture from joint active inference communication. *Front. Neurorobot.* **2023**, *16*, 944986. [[CrossRef](#)]
71. Friedman, D.A.; Tschantz, A.; Ramstead, M.J.D.; Friston, K.; Constant, A. Active Inferants: An Active Inference Framework for Ant Colony Behavior. *Front. Behav. Neurosci.* **2021**, *15*, 647732. [[CrossRef](#)] [[PubMed](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.