POLITECNICO DI TORINO

Master Degree course in Physics of Complex Systems

Master Degree Thesis

HGF.jl: a Julia package for Hierarchical Gaussian Filter fitting and simulation.

Supervisors
Prof. Christopher Mathys
Prof. Andrea Pagnani

Candidate
Jacopo Comoglio

Academic Year 2021-2022

Acknowledgements

I would like to thank my colleague Peter Thestrup Waade who worked with me on the development of this package, my supervisor prof. Chris Mathys who introduced me to the model and gave me the possibility to join him to work on it, and all the staff at Aarhus University interacting mind centre for the amazing work environment.
I would like to also thank all the professors of the master in Physics of Complex Systems both in Paris and Torino for their support and understanding. In particular prof. Alessandro Pelizzola, prof. Jean-Baptiste Fournier and prof. Julien Tailleur without whom I would have never been able to come this far and I would have probably already regretfully left this field.
Last I would like to thank my friends and my family who believed in me even when I had stopped for a long time.

Abstract

The Hierarchical Gaussian Filter has been used for several years now as a good middle ground between Bayesian Inference and Reinforcement Learning models when it comes to estimating how an agent updates its beliefs when presented with new information.
Our Julia package will provide a new environment to run such analysis endorsed with a more user friendly structure allowing both a faster and smoother workflow and the possibility to implement bigger and more complex HGF structures just as easily.
It also makes it finally possible to use sampling techniques to fit the model parameters once given both the inputs and the responses.
After a short introduction to HGF models and Julia, this thesis will provide a list explaining the functions making up the package and a couple of usage examples showing the workflow in action both in a testing case and a real research task.

Contents

Chapter 1

Introduction

This thesis' purpose is to present the newly developed HGF.jl Julia package which was published in September 2022 [1] . This new release is meant to improve the pre-existing MatLab toolbox by making it easier to create and run complex HGF structures and running different samplers without the need to move to a separate program or language (e.g. Stan or R). The package is in fact fully written in native Julia and so are all the used external packages. This allowed us to remove the need to rely on a paid platform like MATLAB while adding new functionalities and, most importantly, a way to fit model parameters.
The first section of this paper will run through the main features of HGF models to give the reader some context on the premises of the model this package aims to run. For a more detailed theoretical presentation on HGF models please refer to [6] and [7] or even [2] for a fully detailed explanation.
A second short introductory section will provide some context and motivation on why the Julia programming language was chosen for this new release. If the reader is not accustomed to this relatively young programming language, please refer to appendix A for a fast overview of some concepts and standards that were used in the package.
After these preliminary sections, the package itself will be presented by introducing first the structure hierarchy on which it is based and then listing and describing all the main functions included in this release.
Please note that as this thesis is being written the package is still a work in progress: the 1.0 version was officially made available to the public in September 2022, but some of its features are still subjected to fixes and expansions in the future releases. The most up to date description of the functionalities for the last releases will be available on the project Github page.
The last part includes two case studies aimed at providing some examples, accompanied by code, for the package usage in different research contexts.
In the example data on the US dollar-Swiss Franc exchange rate are used to exemplify

[1]note: curently the package is published split in two separate packages called "HierarchicalGaussian-Filtering.jl" and "ActionModels.jl"

what kind of analysis our package is capable of performing in terms of fluctuations detection and parameter retrieval.
In the second example instead some analysis is performed on data drawn from an actual behavioural research study "The Alien Task" [2] where participants were required to guess the amount of gold hidden in a cave when presented with the image of an alien whose features noisily correlated with the value of the reward.
To download the current version of the package please refer to the official GitHub repositories `https://github.com/ilabcode/HierarchicalGaussianFiltering.jl` and `https://github.com/ilabcode/ActionModels.jl`.

[2]Soon to be published but not available yet.

Chapter 2

Brief introduction to HGF models

2.1 The HGF structure

The Hierarchical Gaussian Filter (HGF) is a family of models introduced for the first time in [6] to describe how an agent (in its broadest sense, i.e. a human being, an artificial intelligence, the stock market...) learns about the value of a quantity that varies in time. Its basic structure consists of a series of Gaussian random walks hierarchically organized in which the variance of the walk at the lower level is determined by the value assumed by the walk one level up.

Assuming x_1 is any environmental varying quantity the agent wants to know, the model assumes it will vary as a Gaussian random walk around its value at the previous time step. In formulas, calling k the time step:

$$x_1^k \sim \mathcal{N}(x_1^{k-1}, \theta) \tag{2.1}$$

Where here θ is the variance.
In principle however there is no reason why the variance of this random walk would be constant over time, hence we consider it to vary in time as a function $f(x_2)$ of another random walk value x_2 that we consider to be at a higher level on the hierarchy.
The model enriched with a second level can now be written as

$$x_1^k \sim \mathcal{N}(x_1^{k-1}, f(x_2)) \tag{2.2}$$

$$x_2^k \sim \mathcal{N}(x_2^{k-1}, \theta_2) \tag{2.3}$$

Again the assumption that the variance θ_2 of x_2 is constant in time can be relaxed by adding a third layer with variable x_3 just as we did before, with the model now being:

$$x_1^k \sim \mathcal{N}(x_1^{k-1}, f(x_2)) \tag{2.4}$$

$$x_2^k \sim \mathcal{N}(x_2^{k-1}, f(x_3)) \tag{2.5}$$

$$x_3^k \sim \mathcal{N}(x_3^{k-1}, \theta_3) \tag{2.6}$$

In this way, we described a situation where not only the value of x_1 can vary in time but also its volatility may change.

To further clarify why this may be useful, think about an agent who is trying to determine the position of a leaf blowing in the wind. Not only the position of the leaf will vary in time but also the wind strength my change and so how much the position of the leaf fluctuates.
In principle there is no limit to how many levels we could add in this way to our hierarchy, but, as we will see later in the case study section, we will see that if we add more layers than needed, the value for those layers will become approximately constant in time hinting that adding more of them became pointless.

The last point in the basic model definition is the specification of the form for the function $f(x)$ for the variance. The most common choice is

$$f(x) = e^{\kappa x+\omega} \tag{2.7}$$

Where κ and ω are parameters of the model. Notice that since $f(x)$ represents a variance a positive function must be chosen.

2.2 The HGF update equations

In this subsection we will now sketch the procedure used to extract the update equations for an agent belief on the hidden states x_i when being presented with the a series of inputs u_k which value is to be predicted by the first layer x_1.
To run this kind of Bayesian inference we should first of all state the generative model implied by our structure. In our example of a 4-level HGF (3 hidden levels plus one input one) this would amount to

$$\begin{aligned} &p(u^k, x_1^k, x_2^k, x_3^k, x_2^{k-1}, x_3^{k-1}, \omega, \kappa, \theta) \\ &\qquad = p(u^k|x_1^k)p(x_1^k|x_2^k)p(x_2^k|x_2^{k-1}, x_3^k, \kappa, \omega) \\ &\qquad\qquad p(x_3^k|x_3^{k-1}, \theta)p(x_2^{k-1}, x_3^{k-1})p(\omega, \kappa, \theta) \end{aligned} \tag{2.8}$$

where $p(u^k|x_1^k)$ can be a deterministic relationship between sensory input and perceptive state or encode the perceptual uncertainty of the agent. Rewriting the generative density as

$$\begin{aligned} &p(u^k, x_1^k, x_2^k, x_3^k, x_2^{k-1}, x_3^{k-1}, \omega, \kappa, \theta) \\ &\qquad = p(u^k, x_1^k, x_2^k, x_3^k, \omega, \kappa, \theta|x_2^{k-1}, x_3^{k-1})p(x_2^{k-1}, x_3^{k-1}) \end{aligned} \tag{2.9}$$

we can highlight the Markovian nature of this process. (i.e. x_2^{k-1} and x_3^{k-1} are the two variables carrying all the information about the previous timesteps inputs:
$p(x_2^{k-1}, x_3^{k-1}|u^{(1,\dots,k-1)}) = p(x_2^{k-1}, x_3^{k-1})$).
Integrating then over x_2^{k-1} and x_3^{k-1} we obtain $p(x^k, u^k, \chi|u^{(1,\dots,k-1)})$ (where x^k is a shorthand for the set of x at timestep k and $\chi = \{\theta, \omega, \kappa\}$ is the set of parameters.) After u^k is observed it can be plugged in to obtain

$$p(x^k, \chi|u^{(1,\dots,k)}) \tag{2.10}$$

the posterior distribution we were looking for. The last thing needed for the posterior probability over the states x^k is a choice for the parameters' priors.
In the HGF model this choice amount to delta functions priors for all the parameters, meaning that those parameters are considered to be fixed or at least to vary at a much longer timescale compared to the states.
Agent's internal states, in fact, are considered to vary during the learning process (e.g. during an experiment) while parameters represent agent's individual characteristics that may only vary in a time much longer than the duration of an experiment. This priors choice give the model the possibility to represent variability between different agents with the differences in their sets of parameters.
While in principle these equations could be sufficient to invert the model and update it by marginalizing over x_2 and x_3 at the previous timestep to compute the new one, this would involve computationally expensive integrals that would make the model not only hard to compute but also not biologically plausible.
Hence the choice to resort to a Variational Bayesian mean-field approximation.

2.3 Variational Bayesian approximation

We will now go through the main steps and features of the approximation used in the HGF model to derive the update equations. For more details on the computational aspect please refer to appendix B and C of [2].
The Variational Bayesian approach allows us to derive an approximated posterior distribution by minimizing the negative surprise on the data, given a model. We will then choose for our approximation a mean-field class of models, allowing us to factorize the distribution at the different levels. We will furthermore choose a single form for the individual levels distributions. This choice will be guided by the maximum entropy principle for it to be the less arbitrary possible. One last simplification comes from the assumption that the distribution will be represented only by its first two moments, not only to simplify the computations but to make them plausible in a biological setting. In the case of nodes where x takes continuous values, this distribution will then be the Gaussian distribution. Solving the variational problem with the mean-field approximation gives as a result the fact that the distribution should be proportional to the exponential of the variational energy $(I(x_i^k))$.

$$\hat{q}((x_i^k)) \propto e^{(I(x_i^k))} \tag{2.11}$$

where $\hat{q}((x_i^k))$ is the distribution at level i in its "unconstrained" form. To enforce the form of q we mentioned beforehand (i.e. Gaussian for "continuous" levels), taking into account the already exponential form of the $\hat{q}$ distribution, it is enough to take a second degree expansion of the variational energies. Concerning the expansion point, after scrapping the idea of using the maximum (since, being it unknown, it would be add computational complexity estimating it), the choice was to expand around the value of x at the previous time step x^{k-1}. The result of this series of approximations is a set of one-step update equations, the form of which can be easily interpreted as a precision weighted prediction error update, reminiscent of Reinforcement Learning theories.

$$\Delta\mu_i^k \propto \kappa_{i-1}\frac{\hat{\pi}_{i-1}^k}{\pi_i^k}\delta_{i-1} \tag{2.12}$$

This is the fundamental equation for the HGF model that is implemented in our package and it can be interpreted as this: the update $\Delta\mu_i^k$ on the mean at level i is proportional to the prediction error δ_{i-1} one level below weighted by the ratio between the prediction precision one level below $\hat{\pi}_{i-1}^k$ and the posterior precision at level i, π_i^k. The coefficient κ_{i-1} can be thought as a parameter representing the coupling between the two levels.

2.4 Action Models

In an experimental setting while we know (and usually even have control) over the input series given to the subject (agent), we cannot access any of their hidden internal states x. What we can observe instead are only the "actions" taken by the agent (even when there is no actual "action" modifying the environment we can take as actions the responses we can observe from the agent). This is what is known as the "observing the observer" framework [3]. To model this we then need to pair our perceptual model (the HGF in our case) with an appropriate action model of our choice (the package provide a good selection of premade ones). This pair HGF-Action Model is what constitutes an "agent" in our model and are the two things to be initialized when creating an agent instance in our package.

2.5 Value and volatility coupling

To allow for more generality and flexibility of use, our package implements a slightly more advanced version of the HGF model with respect to the one introduced before.
This version is known as the "generalized HGF" (for short gHGF) and can be found in detail in chapter 4 of [4].
In this framework nodes who are one level above another one are called his parents (and the lower level node their child).
For the scope of this thesis and for the usage of the package, the only substantial difference with respect to the previous version is the introduction, together with the parent-child relationship we introduced before, called from now on "Volatility Coupling", a new hierarchical relationship called "Value Coupling".
This new relationship is reflected in different ways in the equations depending on the fact that the child node represents an internal state or an input.
Value coupling for input children amount to assuming that those inputs were drawn from a Gaussian distribution with the parent's value as mean.
Value coupling for state children amounts instead to considering a Gaussian distribution whose mean is determined by the sum of the value of the state at the previous time-step and the value of the parent state multiplied by the coupling coefficient. Figure 2.1 presents an example of a structure with both volatility and value parents showing the mathematical form for the assumed distributions.

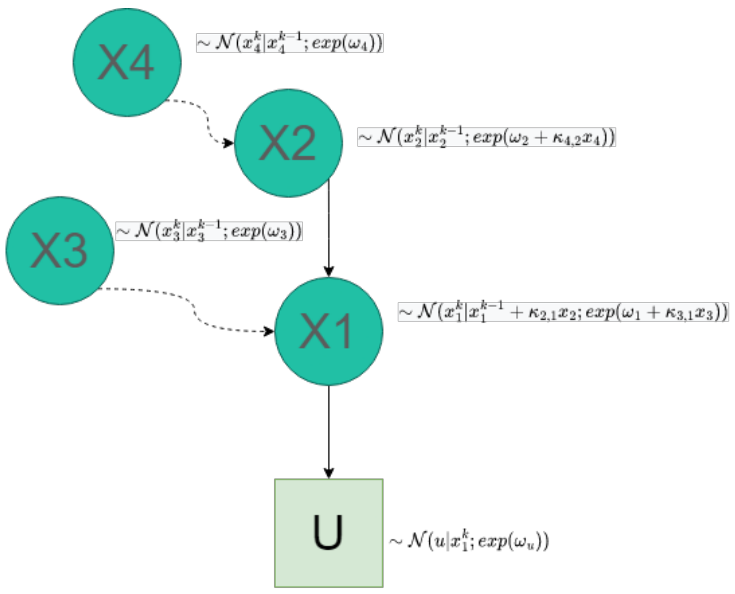
X4
$\sim \mathcal{N}(x_4^k|x_4^{k-1}; exp(\omega_4))$
X2
$\sim \mathcal{N}(x_2^k|x_2^{k-1}; exp(\omega_2 + \kappa_{4,2}x_4))$
X3
$\sim \mathcal{N}(x_3^k|x_3^{k-1}; exp(\omega_3))$
X1
$\sim \mathcal{N}(x_1^k|x_1^{k-1} + \kappa_{2,1}x_2; exp(\omega_1 + \kappa_{3,1}x_3))$
U
$\sim \mathcal{N}(u|x_1^k; exp(\omega_u))$

Figure 2.1. Generalized HGF structure. Circles represent state nodes, squares input nodes. Solid lines connect value parent-child couples and dashed lines variance parent-child couples. Close to every node it is possible to read the distribution implied by the structure.

2.6 HGF main features and motivations summary

The HGF model differs from other models for inference and learning in a series of features that constitutes the main strengths of this framework:

- It has a Reinforcement Learning like set of analytical one-step equations but, at odds with usual RL models, these equations have a strong mathematical basis rooted in Bayesian Statistics.
- At odds with common Bayesian models its update does not involve complex integrals, making it a reasonable candidate to model real-time computations in biological systems.
- Parameters and states used in the model have a clear meaning and so are easy to make sense of.
- The set of parameters is considered fixed or to vary at a much slower rate with respect to the states, allowing using it to model individual differences between agents or differences in value for a single agent at different moments. This can be used e.g. to link abnormal values of the parameters to pathological conditions.

On a strictly practical level, for the package user, the main take-aways from this theoretical introduction on HGF models must be:

- The **agent structure**, being composed by a **perceptual model** (in our case the HGF) with its (internal, inaccessible) **states nodes** representing the way the agent believes on the outside world are filtered and organized and the **input nodes** storing the sensory inputs given by the outside world (remember that sensory uncertainty is modelled by the relationship between the input and the first state level) and an **action model** that may depend on internal state values and gives an output that may influence or modify the external world and that we can observe.

- The fact that all the parameters for the **perceptual model** (evolution rates, initial state values, coupling constants) and the **action model** are used to represent differences between agents or for a same agent in two separate times.

Chapter 3

The Julia programming language

This newly developed Julia [1] package was meant to replace and expand the previously existing MatLab HGF Toolbox. The advantages of writing this new package all in native Julia mainly amount to the following:

- Julia is open source, this will remove the need for a paid software as MatLab was.
- Julia is both fast and high-level, it was designed for performances and, unlike e.g. Python, is a compiled language. Nevertheless being it high level allowed us to code an easy and straightforward framework for the final user.
- Julia "multiple dispatch" features are particularly useful to code a modular structure such as the HGF model allowing the user to code big HGFs in a much easier way than before.
- The Turing [5] Julia package for model sampling is very efficient thanks to Automatic Differentiation and offers a wide variety of state-of-the-art samplers such as the No U-Turns Sampler (NUTS).
- Julia package management systems make it very easy for the final user to install our package once it will be listed in the official General Registry.

All of this factors concurred in making Julia our language of choice to develop a tool that could be at the same time very powerful but easy to use even without a deep knowledge on the underlying mathematics and without the need for advanced programming skills.
We hope in this way to open the path to the usage of more advanced computational and mathematical models in fields where they traditionally struggle to settle.
If the reader is not accustomed to this relatively young programming language we invite them to read the appendix A before moving to the next section since, even if Julia commands and keywords are pretty straightforward and similar to other high-level languages (e.g. Python), there are some standards and peculiarities that is worth knowing in advance for a better understanding.

Chapter 4

Package overview

4.1 General overview

The purpose of this new Julia package is to make it intuitive and easy to implement HGF models and run inference on data also for researchers without a deep understanding of the Mathematics behind it or extensive coding experience.
The package includes both generic low-level functions, allowing the more experienced users to define their own models in full detail, as well as a series of utility functions to speed up the workflow and help even the less experienced users successfully run this tool.
The package backbone is constituted by a structure architecture designed to efficiently implement the HGF structure and take advantage of Julia's multiple dispatch. We will now present an overview of this structure. Please note that the final user will pretty much never interact directly with this since a series of utility functions will do all the work providing a more intuitive front-end.

4.2 The core Structs

Taking a top-down approach at the widest level we found the **AgentStruct**, it represents the agent and its fields are:

- **action model**: the function defining the chosen action model for our agent.
- **params**: a dictionary specifying the action model parameters.
- **state**: a dictionary memorizing the current states given by the action model.
- **history**: a dictionary saving the previous states and action taken by the action model.
- **action**: saving the current value for the action.
- **perception_struct**: the perceptual model assigned to the agent. This is the field where an HGFStruct can be put (but in principle also other perceptual models).

This corresponds to the agent we defined in the theory section, its two main components "action model" and "perceptual model" are passed in the two corresponding fields.
The HGF per se is coded as an **HGFStruct** to be passed to an **AgentStruct** or to be used by itself when no response is needed (this may be the case when we just want to observe the evolution of internal state without the need to fit them against the agent actions).
Its fields are:

- **perception_model**: a function used to update the model at each time-step (for the classic HGF this will be **update_HGF!** but in principle other user-defined functions can be used).
- **all_nodes**: A dictionary containing the all nodes of the HGF model used when there is no need to distinguish by node type.
- **input_nodes**: A dictionary containing the input nodes of the HGF model defined as **InputNode** structures.
- **state_nodes**: A dictionary containing the input nodes of the HGF model defined as **StateNode** structures.
- **ordered_nodes**: A convenience vector of **InputNode** and **StateNode** used to pass the update order when it is nontrivial.

The last building bricks in our structure hierarchy are the **InputNode** and **StateNode** structures both in their continuous and binary version. They represent the states and inputs of the HGF structure as "nodes" connected by parent-child relationships that are summarized inside the nodes themselves. Nodes of the binary type are used to describe inputs that can assume only binary values (e.g. if the light is on or off, if a sound was played or not) or to describe states whose statistic is assumed to be described by a logistic function for the probability of that state to be "on" CHECK.
Binary nodes are defined as a different custom type so that the program, when running the update equations can recognize them and automatically select the correct formulas. In future releases also Categorical nodes are planned to be implemented in a similar fashion to represent quantities that take values in a finite, discrete set. The fields are, for **StateNode**:

- **name**: the name assigned to the node
- **value_parents**: a list of the higher-level nodes coupled to the node by value.
- **volatility_parents**: a list of the higher-level nodes coupled to the node by volatility.
- **value_children**: a list of the lower-level nodes coupled to the node by value.
- **volatility_children**: a list of the lower-level nodes coupled to the node by volatility.

- **params**: a dictionary containing the node parameters such as evolution rate and all the value and volatility couplings to the other nodes (this are the θ, κs and ωs from section 2.
- **state**: a dictionary containing the variables related to the current state of the node, such as prediction and posterior means and precisions and prediction errors.
- **history**: a dictionary saving in a vector the variables related to the state of the node at all time-steps.

For **InputNode** instead:

- **name**: the name assigned to the node
- **value_parents**: a list of the higher-level nodes coupled to the node by value.
- **volatility_parents**: a list of the higher-level nodes coupled to the node by volatility.
- **params**: a dictionary containing the node parameters such as evolution rate and all the value and volatility couplings to the other nodes.
- **state**: a dictionary containing the variables related to the current state of the node, such as input values, prediction errors, and predicted precision.
- **history**: a dictionary saving in a vector the variables related to the state of the node at all time steps.

Note that, at odds with state nodes, the input nodes lack a field for value and volatility children. This is due to the fact that inputs are always placed at the lowest levels in an HGF hierarchy since they represent the perceptual data fed to the agent.

The binary versions **BinaryStateNode** and **BinaryInputNode** have exactly the same fields, they are separate structures purely for multiple dispatch usage.

This structure is diagrammatically summarized in figure 4.1.

4.3 Initialization functions

This subsection includes functions whose purpose is to create HGFStruct and AgentStruct in a more user friendly way together with functions to generate automatically the most used models.

- **init_HGF(node_defaults, input_nodes, state_nodes, edges, update_order = false, verbose = true,)**: this function is used to create an HGF structure with the maximum generality. It also runs a series of checks to ensure the defined structure it's a legitimate HGF.
  In the **node_defaults** argument can be passed a named tuple with the parameters

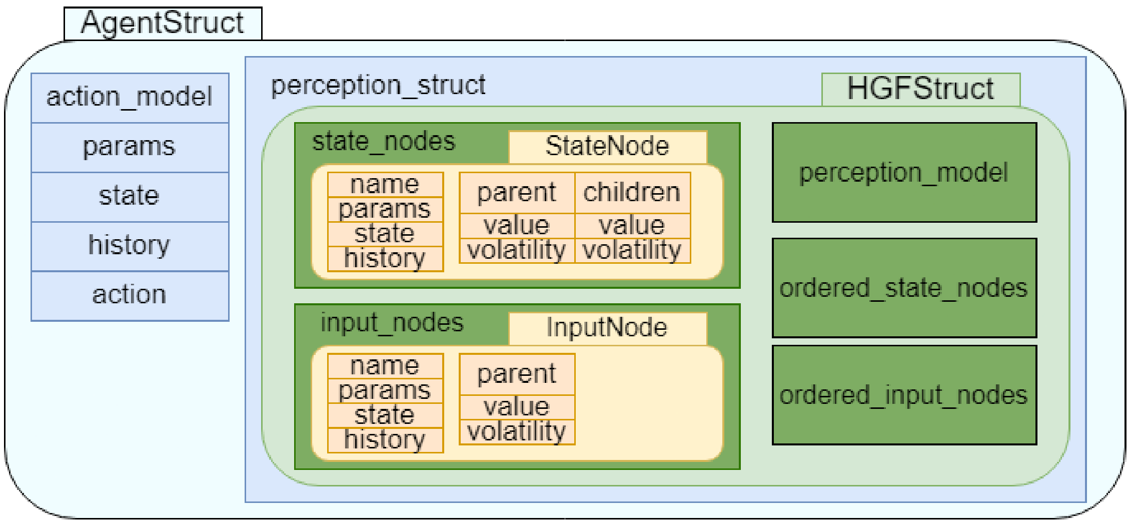

Figure 4.1. Diagram summarizing the main structures of the package. Note that the "state_node" and the "input_node" fields in the HGFStruct may contain multiple StateNode and InputNode structs as well as their binary versions.

to be used for every node if not differently specified.
The **input_nodes** and **state_nodes** arguments are given lists of named tuples stating the node names and, optionally, their parameters.
In the **edges** field are specified, by mean of a list of named tuples, the directed graph edges between parent and child nodes.
The **update_order** keyword can be given a list specifying a particular update order or be set to **false** for it to be determined automatically.
The **verbose** keyword can be changed to false to turn off the warnings.

- **premade_HGF(model_name::String, params_list = (;), starting_state_list = (;))**: this function allows to create the most widely used HGF model in the fastest possible way by simply calling the model name.
  The **model_name** field has to be given one of the available keywords to initialize the corresponding model. Calling the special keyword "**help**" will instead show a list of all the implemented keywords. Currently the working keywords are "**continuous_2level**", "**binary_2level**", "**binary_3level**", "**JGET**".
  The **params_list** and **starting_state_list** optional arguments allows to pass a list of named tuples specifying the parameters and starting states for each node in the preset configuration[1].

- **init_agent(action_model::Function, perception_struct, params, states)**:

[1]Note: in newer versions **params_list** and **starting_state_list** have been unified in a single **params_list** argument

this function creates an AgentStruct based on a user-defined action model in a more user-friendly way. The **action_model** argument takes a function used to compute the action.
In the **perception_structure** field a perceptual model can be passed (if working with HGF this will be an HGFStruct).
The **params** argument receives a named tuple with the **action_model** parameters values. The **states** argument receives a named tuple with the **action_model** states if any.

- **premade_agent(model_name::String, perception_model = (;), params = Dict(), states = Dict(),)**: this function allows to initialize an AgentStruct without having to create a new action model but using the premade ones that can be called by passing the respective keyword to the **model_name** argument. All the other arguments work the same way as in **init_agent**. Calling the special keyword "**help**" will instead show a list of all the implemented keywords. Currently working premade agents keywords are **hgf_gaussian_action**, **hgf_binary_softmax_action**, **hgf_unit_square_sigmoid_action**

4.4 Updating functions

The functions in this subsection are at the core of HGF simulation, we will now present only the main ones since the user will never have to deal with the most low level ones (and also very rarely with all but the **give_inputs!** function).

- **give_inputs!(HGF::HGFStruct, input::Number)**: function to perform one update step on an **HGF** with the provided **input**. This functions calls the **update_HGF!** function which takes case of calling all the hierarchy of update equation in the right order for the created structure.
- **give_inputs!(HGF::HGFStruct, inputs::Array)**: this function calls the previous one multiple times to evolve the **HGF** on the whole array **inputs**.
- **give_inputs!(HGF::HGFStruct, inputs::Dict{String,Vector})**: same as before but this time the **inputs** are provided as vectors in a dictionary (useful when working with multiple input nodes).
- **give_inputs!(agent::HGFStruct, input::Number)**: this is the main function for agent simulation. It extracts the HGF structure from the **agent** and evolves it but it also utilizes the result to generate action/response following the **agent** action_model and returns it.
- **give_inputs!(agent::AgentStruct, inputs::Array)**: this function calls the previous one multiple times to evolve the **agent** on the whole array **inputs** returning actions/responses as an array.
- **give_inputs!(agent::AgentStruct, inputs::Dict{String,Vector})**: same as before but this time the **inputs** are provided as vectors in a dictionary and actions/responses returned as such (useful when working with multiple input nodes).

- **update_HGF!(HGF::HGFStruct, inputs)**: this function is the core of the HGF evolution and simulation. It takes the **HGF** structure and evolves it on **inputs** by extracting the node update order and running for all nodes (relying on multiple dispatch to distinguish between state and input nodes) the functions implementing the mathematical equations such as:
  - **update_node_prediction!(self::StateNode)**
  - **update_node_posterior!(self::StateNode)**
  - **update_node_prediction_error!(self::StateNode)**
  - **update_node_input!(self::InputNode)**
  - **update_node_prediction!(self::InputNode)**
  - **update_node_prediction_error!(self::InputNode)**

Note that all of this functions are present also with their binary version, taking as inputs **BinaryStateNodes** and **BinaryInputNodes** and implementing the appropriate versions for the equations.

4.5 Fit functions

This section functions are related to parameters estimation given both input and responses to an agent model. The fitting takes advantage of the Turing package for sampling and Automatic Differentiation.

- **fit_model(agent::AgentStruct, inputs::Vector{Float64}, responses::Union{Vector{Float64},Missing}, params_priors_list = (;)::NamedTuple{Distribution}, fixed_params_list = (;)::NamedTuple{String,Real}, sampler = NUTS(), iterations = 1000,)**: this function is the core of the model fitting in the HGF.jl package. It creates a Turing model from an agent structure, run its sampling given input and responses to get the parameters posterior distributions and runs a series of operation in order to preserve the given AgentStruct and make the returned Turing chain object more readable. The function takes the AgentStruct **agent**, stores its parameters, then substitutes the ones listed in the **fixed_params_list** given named tuple with their provided values. It then proceeds to create a Turing **@model** macro where the parameters to be estimated, listed in the named tuple **params_priors_list**, are declared to be distributed accordingly to the given (hyper)priors. It then reset the agent (this is important since during the sampling the agent will go trough a complete cycle of updates multiple times) and then evolves the agent on the given **input** array with the sampled parameters extracting the distribution for the response at each time step and declaring that the given observed **responses** come from that distribution. This declared model is then sampled with the given **sampler** for the given number of **iterations** and the resulting Turing chain is returned (after some clean-up to improve readability). The function also restores the previous parameters at the end and resets the agent history since the Automatic Differentiation used in Turing sampling converts everything to dual numbers, not suitable if the user wants to do more analysis with the same model.

4.6 Plotting functions

This subsection includes custom plotting functions created for the HGF.jl package.
They have been defined as **Plot Recipes** using the **RecipesBase** package. This allowed us to define new plotting functions without the need of adding heavy dependencies on plotting packages and allowing also the user to choose their favourite plotting backend. This unfortunately had a small cost in code readability having to define macros and not functions but for the final user, no substantial difference will occur. We will then present now these plotting recipe macros with the signature they would have if they were functions (since the user would ultimately use them as if they had this signature.) Note that due to Julia's multiple dispatch more functions with the same name but different type signatures in the arguments can be defined.

- **HGF_trajectory_plot(agent::AgentStruct, node_name, property, error_type)**: this function plots the chosen **property** for the node **node_name**. If no **property** is provided it will plot the posterior mean for the selected node with its error ribbon. If in the **property** argument is are passed the keywords "posterior" or "prior" the posterior/prior mean will be plotted for the selected node with its error ribbon.
  Non-continuous quantities will be automatically plotted as scatter plots (with error bars if applicable).
  The **error_type** argument can be used to choose between plotting the standard deviation or a confidence interval in the applicable cases.
  Being this a plot recipe it also accepts all the supported keywords of the chosen plotting backend (such as colours, marker shapes, axis and graph titles, etc.)

- **HGF_trajectory_plot(agent::AgentStruct, action_property)**: this function its used to plot the history for the action model's states and actions. As before being this a plot recipe it accept also all the supported keywords of the chosen plotting backend.

- **HGF_trajectory_plot(HGF::HGFStruct, node_name, property, error_type)**: this function works exactly like the first one but it takes directly an HGFStruct as argument instead of extracting it from the agent.

- **posterior_parameter_plot(chain, params_prior_list, label_list)**: this function takes a **chain** resulting from Turing simulation and the **params_prior_list** containing the estimated parameters that one wants to plot. It creates a plot where each subplot shows the median value, the chosen quantiles, and the distribution shape for both the given priors on parameters and the posterior estimated during the simulation.
  The optional field **label_list** takes as argument a named tuples containing the name to display as titles in the subplots. If nothing is given the default names are

used.
The following special keywords can be used in addition to all the keywords normally supported by the chosen backend (the value after the equal sign is the default.)

  - **prior_offset = 0**: vertical offset for prior x axis.
  - **posterior_offset = 0.01**: vertical offset for posterior x axis. (It is suggested to have slightly different offsets for prior and posterior to avoid poor graph readability if the quantiles bars superimpose.
  - **prior_color = :green**: sets the colour for the prior distribution.
  - **posterior_color = :orange**: sets the colour for the posterior distribution.
  - **interval_1 = 0.5**: sets the quantile to be shown by the thicker error bar. Specify it using decimal number (e.g. 0.5 for 50% quantile.)
  - **interval_2 = 0.8**: sets the quantile to be shown by the thinner error bar.
  - **distributions = true**: changing this flag to false allows to plot only the medians with the quantiles without superimposing the distributions.
  - **plot_width = 900**: sets the width of the single subplots.
  - **plot_height = 300**: sets the height of the single subplots.

- **predictive_simulation_plot(agent::AgentStruct, chain::Chains, state::String, iterations::Int, inputs)**: this function takes as inputs an **AgentStruct** and a **chain** resulting from a Turing fit, draws for the chosen number of **iterations** a set of parameters from the posterior distributions stored in **chain** and plots the evolution of the **agent** on the given **inputs** with a thin grey line with an alpha of 0.1 to make the regions with the most trajectories appear darker.
  It will furthermore draw a colored line for the trajectory corresponding to the median values of the parameters.

- **predictive_simulation_plot(agent::AgentStruct, prior_list::NamedTuple, state::String, iterations::Int, inputs)**: this function takes as inputs an **AgentStruct** and a **NamedTuple** containing a set of prior distributions for the parameters (usually the one used for the fitting with Turing), draws for the chosen number of **iterations** a set of parameters from the give prior distributions and plots the evolution of the **agent** on the given **inputs** with a thin grey line with an alpha of 0.1 to make the regions with the most trajectories appear darker.
  It will furthermore draw a colored line for the trajectory corresponding to the median values of the parameters.

4.7 Utility functions

This section contains a variety of functions designed to make it easier to retrieve information from an HGF or Agent structure or to modify them. Most of them were defined on different classes of inputs by multiple dispatch and exploit this feature to simplify the design of the wider scope ones by calling inside more specific one (e.g. functions accepting

an array of inputs using the version for just a single input inside them.).
These multiple dispatched functions will be listed multiple times with their different possible signature and eventual important differences in the description.

- **get_params(HGF::HGFStruct)**: it returns a named tuple with the value of all the **HGF** parameters.
- **get_params(agent::AgentStruct)**: it returns a named tuple with both the value of all the **agent**'s HGF parameters and its response model.
- **get_params(chain::Chains)**: it returns a named tuple with the median value for all the estimated values on the corresponding simulations.
- **get_states(HGF::HGFStruct, feat::String)**: it returns the current value of the state **feat** of an **HGF**passed as the string representing its name.
- **get_states(HGF::HGFStruct, feats::Array{String})**: same as before but it returns a named tuple with the values of all the requested **feats.**
- **get_states(HGF::HGFStruct)**: same as before but for every available state of the passed **hfg**.
- **get_states(agent::AgentStruct, feat::String)**: same as before but it extracts the **HGF** from the AgentStruct and it is possible to ask as **feat** a state or action from the response model.
- **get_states(agent::AgentStruct, feats::ArrayString)**: same as before but returning the named tuple.
- **get_states(agent::AgentStruct)**: same as before but returning every available state for both the **HGF** and the response model.
- **get_history(HGF::HGFStruct, feat::String)**: it returns a vector with the history of the state **feat** of an **HGF** passed as the string representing its name.
- **get_history(HGF::HGFStruct, feats::Array{String})**: same as before but returns a named tuple with all the requested **feats**.
- **get_history(HGF::HGFStruct)**: same as before but for every available state of the passed **hfg**.
- **get_history(agent::AgentStruct, feat::String)**: same as before but it extracts the **HGF** from the AgentStruct and it is possible to ask as **feat** a state or action from the response model.
- **get_history(agent::AgentStruct, feats::Array{String})**: same as before but returning the named tuple.
- **get_history(agent::AgentStruct)**: same as before but returning every available state for both the **HGF** and the response model.

- **set_params!(agent::AgentStruct, params_list = (;)::NamedTuple)**: it changes the **agent**'s parameters that are keys in the named tuple **params_list** to the corresponding assigned values.
  The syntax of the names in the give named tuple is the same one as the one given by the **get_params** function and it consist (for parameters involving just one node) in **nodename** + _ + **featname** where **nodename** is the name chosen for the node whose feature is relative to and **featname** is one of the possible parameters or starting states such as (**evolution_rate**, **posterior_mean**, **posterior_precision**). For parameters involving two nodes (e.g. **coupling_strength**) instead the syntax must be **childnodename** + _ + **parentnodename** + _ + **featname**.
- **reset!(HGF::HGFStruct)**: it resets the **HGF** to its starting values.
- **reset!(agent::AgentStruct)**: same as above but for the whole **agent**.
- **lognormal_params(μ, σ)**: it takes as arguments the wanted mean μ and standard deviation σ and returns a named tuple with the **mean** and standard deviation **std** to be put as parameters in the normal distribution in the lognormal to have the requested mean and standard deviation in the overall distribution.
  This utility becomes pretty useful in the context since the lognormal distribution is often used as prior for positive only parameters (e.g. initial precisions).[2]

[2]This function will be probably moved to a separate package for statistical utilities in future releases.

Chapter 5

The CHF-USD exchange rate

The first use example we will now present is an analysis regarding the exchange rates data from Swiss franc to US dollar in the the two years 2010 and 2011. This task has always played the role of a "test" for any HGF computer implementation up to now so it was an obvious choice for us to check if the package was working as expected.
This analysis will also provide a first tutorial for the future users and a first basic example of the HGF workflow.
In this example, we consider an agent (that in this case may represent a single ForEx trader as well as the currency market as a whole) who is trying to guess the value of the exchange rate by updating their internal HGF states given as inputs the historical time series. This "guess" will then be modeled as a Gaussian response model with the mean given by the predicted value for x_1 and a parameter that may vary from agent to agent as standard deviation.

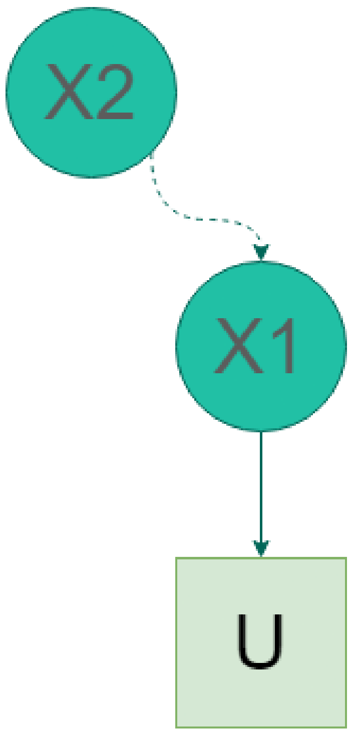

Figure 5.1. 2 level HGF structure. Circles represent state nodes, squares input nodes. Solid lines connect value parent-child couples and dashed lines variance parent-child couples.

5.1 Simulating the agent

In this first part of our example/sanity check, we will initialize our agent with a set of chosen parameters and evolve it on the given inputs so to have them generate a vector of responses we will use in the second part to try and estimate back the correct parameters.

We start our code by importing the HGF package and creating a continuous two level HGF structure using the **premade_HGF** command.
This structure consist of one input node u representing the given exchange rate value which is connected to a value parent x_1 who has itself a volatility parent x_2 estimating its variance as shown in figure 5.1.
Note that the same result could have been obtained by manually creating the structure with the following code. The **premade_HGF** function saves the user a lot of time and lines of code but it is still possible to create more structure in complete freedom.

```
node_defaults = (
    params = (; evolution_rate = 3),
    starting_state = (; posterior_precision = 1),
    coupling_strengths = (; value_coupling_strength = 1),
)
```

```
#List of input nodes to create
input_nodes = [(name = "u", params = (; evolution_rate = 2))]

#List of state nodes to create
state_nodes = [
    (name = "x1", params = (; evolution_rate = 2)),
    (
        name = "x2",
        params = (; evolution_rate = 2),
        starting_state = (; posterior_mean = 1, posterior_precision = 2),
    ),
]

#List of child-parent relations
edges = [
    (child_node = "u", value_parents = "x1"),
    (child_node = "x1", volatility_parents = "x2"),
]

#Initialize an HGF
myHGF = HGF.init_HGF(node_defaults, input_nodes, state_nodes, edges);
```

The next step is to initialize the agent by mean of the function **premade_agent**. We pass it the just created HGF and chose the Gaussian response as the response model, selecting as the mean for the response the posterior mean of state x_1.

```
my_agent = HGF.premade_agent(
    "HGF_gaussian_response",
    my_HGF,
    Dict("action_noise" => 1),
    Dict(),
    (; node = "x1", state = "posterior_mean"),
);
```

Now to give the agent the chosen parameters and starting states we just need to create a named tuple with the desired values and call the function **set_params** with the agent and the tuple as arguments. We also call the **reset!** function in order to have the new starting values correctly placed in the structure. Note: the parameters value could have been passed already during the creation of the HGF and agent but this procedure makes the code tidier and easier to read and interpret step by step.

```
params_list = (
    u_x1_coupling_strenght = 1.0,
    x1_x2_coupling_strenght = 1.0,
    u_evolution_rate = -log(1e4),
    x1_evolution_rate = -13,
    x2_evolution_rate = -2,
    x1_posterior_mean = 1.04,
    x1_posterior_precision = 1 / (0.0001),
    x2_posterior_mean = 1.0,
    x2_posterior_precision = 1 / 0.1,
    action_noise = 0.01,
)
```

```
HGF.set_params(my_agent, params_list)

HGF.reset!(my_agent)
```

Now it is finally possible to give the agent the inputs and have it evolve while saving the generated responses in a vector to be used in the second part.

```
inputs = Float64[]
open("data//canonical_input_trajectory.dat") do f
    for ln in eachline(f)
        push!(inputs, parse(Float64, ln))
    end
end
responses = HGF.give_inputs!(my_agent, inputs)
```

The evolution of the states and responses can now be output with the command **get_history** or directly plotted.
Note that we can use freely any normal plotting option thanks to the implementation of plots as recipes. Also using an exclamation mark after the command will have it plotted on the same graph as per Julia standards.
The results of the plotting are shown in images 5.2, 5.3 and 5.4.

```
using Plots
using LaTeXStrings

HGF_trajectory_plot(my_agent, "u",
size=(1300,500),
xlims = (0,615),
markerstrokecolor = :auto,
markersize=3,
markercolor = "green2",
title ="Agent simulation",
ylabel="CHF-USD exchange rate"
)

HGF_trajectory_plot!(my_agent, "x1", "posterior_mean",
color="red",
linewidth=1.5)
HGF_trajectory_plot!(my_agent, "action",
size=(1300,500),
xlims = (0,614),
markerstrokecolor = :auto,
markersize=3,
markercolor = "orange",
)

HGF_trajectory_plot(
    my_agent,
    "x2",
    color = "blue",
    size = (1300, 500),
    xlims = (0, 615),
    title = L"Posterior\:expectation\,of\,x_{2}",
)
```

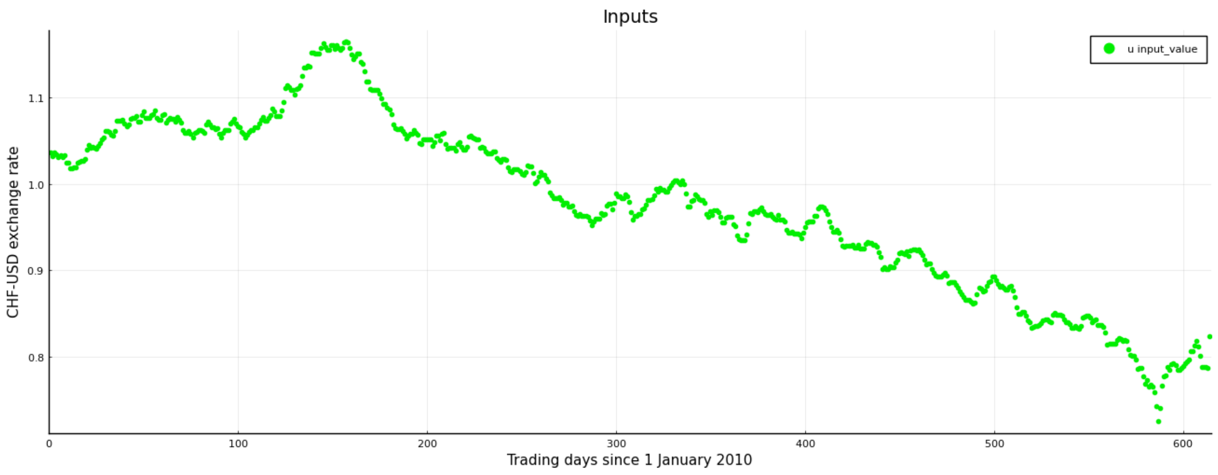

Figure 5.2. The values of the CHF-USF exchange rate given as inputs to the HGF

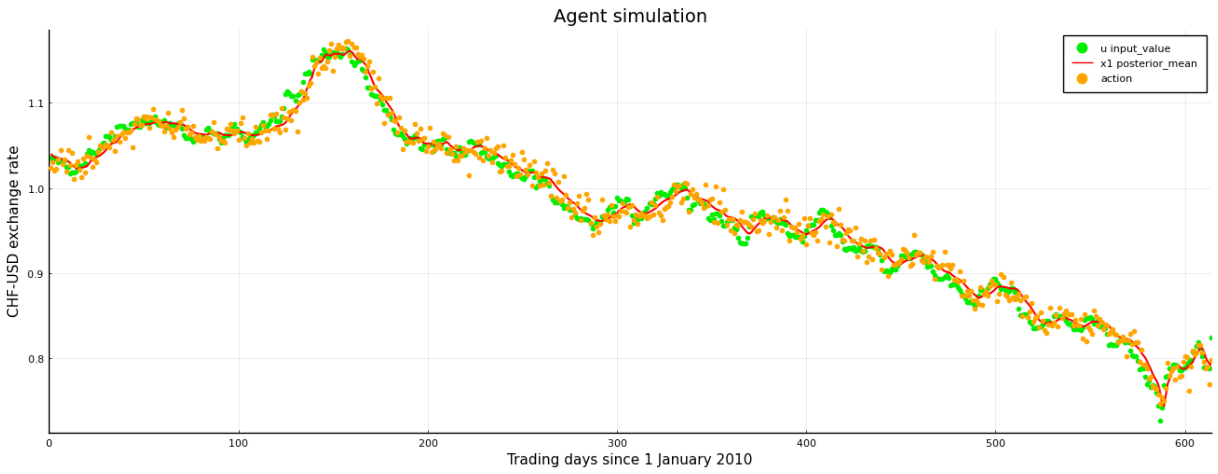

Figure 5.3. The inputs (green) together with the mean of the state x_1 (red) and the generated responses (orange).

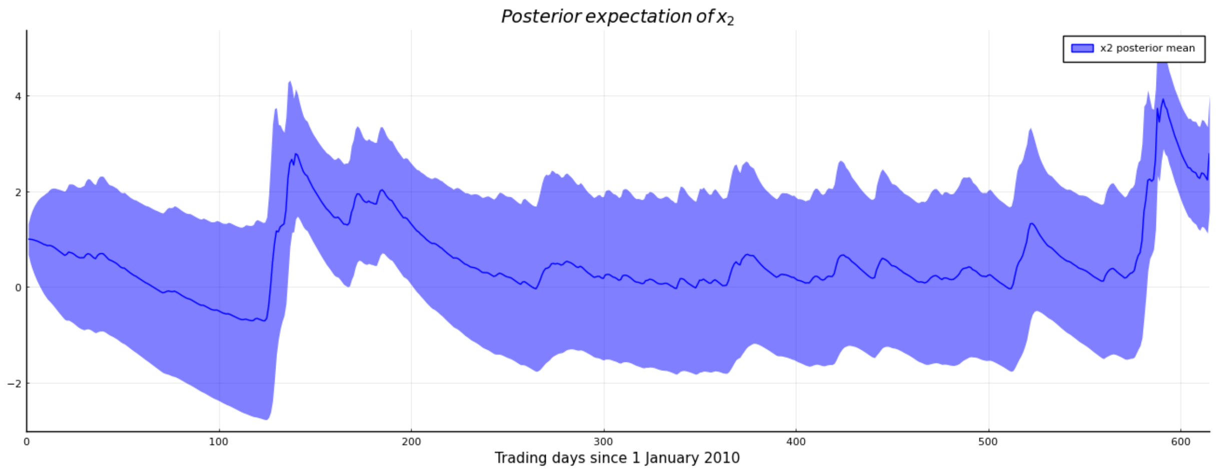

Figure 5.4. The value of the mean of x_2 with its inverse precision as error ribbon. Note the two spikes corresponding to April 2010 and September 2011.

We can observe how the posterior mean of x_1 follows closely the given inputs in figure 5.3 and it is also interesting to note how in figure 5.4 the posterior mean of x_2 presents two sharp peaks around April 2010 and September 2011. In the first period, in fact, the Greek default happened while the second one correspond to the decision of the Swiss government to put a limit on how much the value of the Euro could drop with respect to the Franc. These two events caused a sharp increase in the uncertainty of the future values of the exchange rates and this is clearly reflected in the hidden states of our agent that perceives the increased volatility and changes their way to adapt beliefs to keep estimating the correct values.

5.2 Fitting the agent parameters

The second part of this example consists in estimating via the Turing sampling package the values of some of the agent parameters trying to recover the ones used in the first part.
First of all, we import the Turing package and we create a named tuple with the parameter we want to fix to an educated guess.

```
using Turing

first_input = inputs[1]
first20_variance = Turing.Statistics.var(inputs[1:20])

fixed_params_list = (
u_x1_coupling_strenght = 1.0,
x1_x2_coupling_strenght = 1.0,
action_noise =0.01,
x2_posterior_mean = 1.,
x1_posterior_precision = 1/first20_variance
)
```

Then we create a second named tuples containing the hyperpriors on the parameters we would like to recover.
We use normal priors for the parameters that can take both positive and negative values and lognormal priors for the positive only parameters. The mean and standard deviations for the priors are again educated guesses developed by frequent users of the model.

```
params_prior_list = (
u_evolution_rate = Normal(log(first20_variance),2),
x1_evolution_rate = Normal(log(first20_variance),4),
x2_evolution_rate = Normal(-4,4),
x1_posterior_mean = Normal(first_input,sqrt(first20_variance)),
x2_posterior_precision = LogNormal(HGF.lognormal_params(10,1).mean,HGF.
    lognormal_params(10,1).std),
)
```

Now to start the sampling it is enough to call the function **fit_model** and pass it the inputs, the responses computed in the first part, the two named tuples for the parameters, and the agent itself. We store the resulting Turing chain object in a variable for future analysis. The last two arguments let us choose the sampler and the number of iterations (in this case No U Turn Sampler and 1000 iterations).

```
chain = HGF.fit_model(
    my_agent,
    inputs,
    responses,
    params_prior_list,
    fixed_params_list,
    NUTS(),
    1000,
)
```

We can now extract the median for the fitted parameters with the function **get_params** and compare their values with the ones chosen at the beginning (this is reported in table 5.1.)
We can also compare the prior distribution for the parameters with the posterior one by calling **posterior_parameter_plot** on the chain and the prior list. The result of this plot is shown in figure 5.5.
Note: what is now called "**x1_posterior_mean**" is the parameter representing the initial value for the mean of the node x1 and in future releases will be then called "**x1_initial_value**" (the old name is due to the fact that the initial value is stored as the posterior value during the first update).

```
fitted_params = HGF.get_params(chain)

posterior_parameter_plot(chain,params_prior_list)
```

	Real Value	Fit Value
u evolution rate	-9.21	-9.21
x1 evolution rate	-13	-12.62
x2 evolution rate	-2	-2.53
x1 posterior mean	1.04	1.035

Table 5.1. Real model parameters to fitted ones comparison.

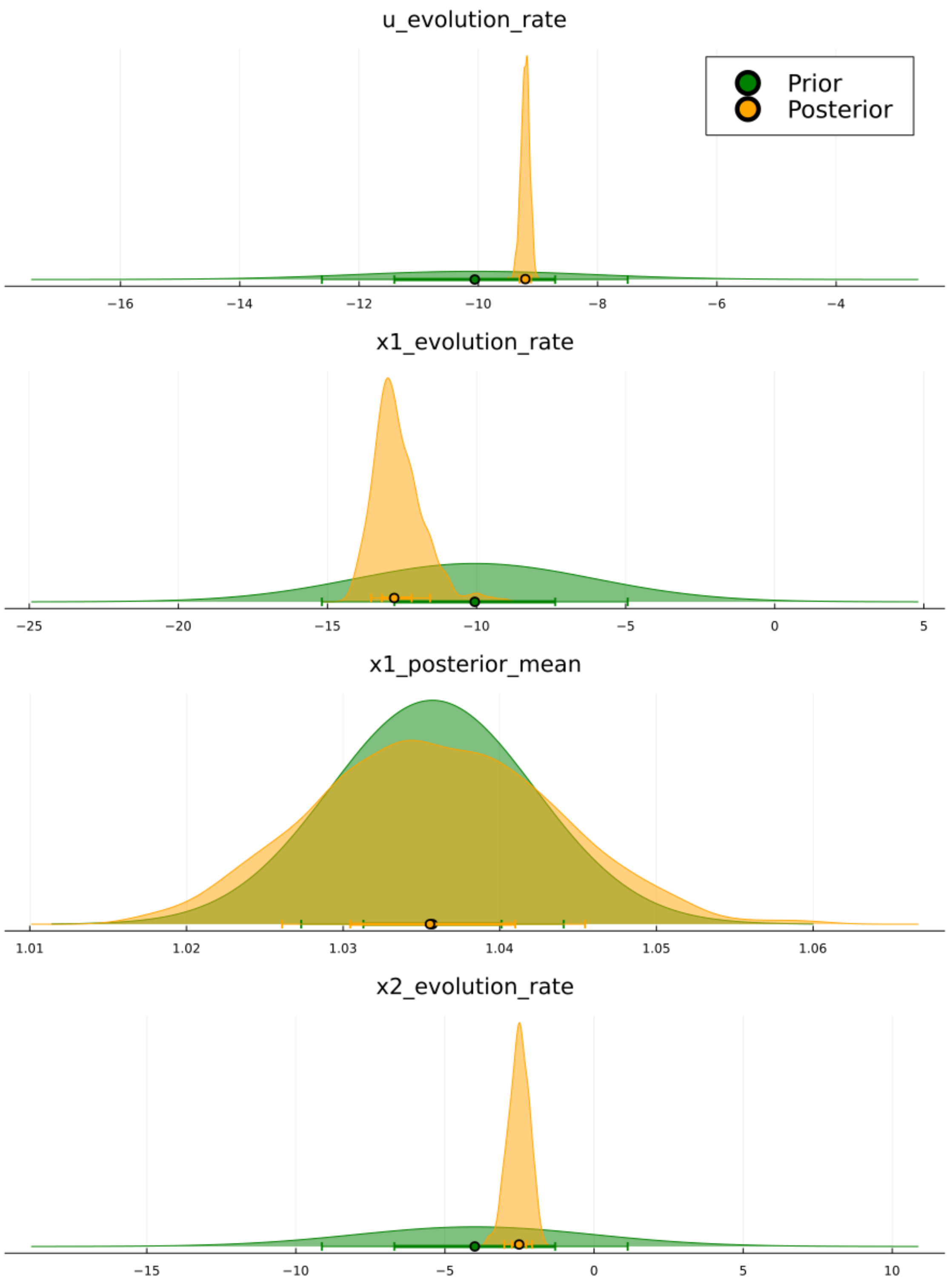

Figure 5.5. Comparison between the prior distributions used for the sampling and the posterior distributions for the parameters.

Chapter 6

The alien task

This second example utilizes data from a real cognitive task whose results are soon to be published.
Participants in the task were presented with the image of an alien in front of a cave and were asked to guess the amount of gold in it by moving a slider. The amount of gold was drawn from a Gaussian distribution whose mean would jump to a different value from time to time and also its variance varied between two values. These values also noisily correlated with the presented alien features.
In this analysis, we will reproduce some of the results obtained in the original paper using the MatLab toolbox using our new package. In particular, we will fit the agent parameters for one subject in one session using the real values of the gold amount as inputs and the subject guesses as responses.
The agent will be modeled with a so-called JGET (Jumping Gaussian Estimation Task) HGF structure characterized by one input node u with one value parent x_1 having itself one volatility parent x_2 on one side and one volatility parent x_3 which has itself a higher level volatility parent x_4 (the structure is schematized in figure 6.1).
The idea behind the choice of this structure is that the two branches generating from the input nodes try to estimate respectively the actual value of the input and its variance but since both these quantities vary in time they both have a volatility parent trying to model this variation.
As a response model instead it will again be used as above a Gaussian response with the x_1 mean as the mean value.

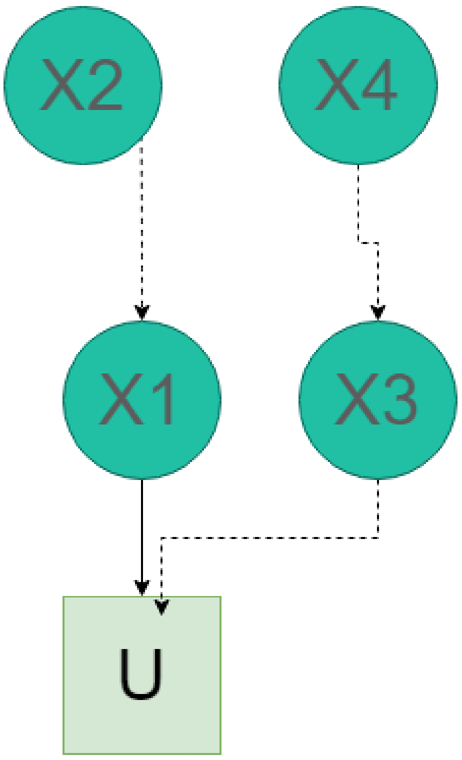

Figure 6.1. The JGET model. As usual circles represent state nodes, squares input nodes. Solid lines connect value parent-child couples and dashed lines variance parent-child couples.

6.1 model fitting and simulations

Just as before the first thing we need to do is to import the package and initialize the HGF model. Since, even if this is a more complex model, it is still a pretty standard one we can then just create it with the **premade_model** function. We can then initialize the agent with the created HGF just as before.

```
using HGF

my_HGF = HGF.premade_HGF("JGET");

my_agent = HGF.premade_agent(
    "HGF_gaussian_response",
    my_HGF,
    Dict("action_noise" => 1),
    Dict(),
    (; node = "x1", state = "posterior_mean"),
);
```

Notice that also this more complex model could have been defined by the user if it would have not been preemptively added to the function. For example with a code like this.

```
#List of input nodes to create
input_nodes = [(name = "u1", params = (; evolution_rate = 2))]

#List of state nodes to create
state_nodes = [
    "x1",
    "x2",
    "x3",
    "x4",
]

#List of child-parent relations
edges = [
    (child_node = "u1", value_parents = "x1", volatility_parents = "x3"),
    (child_node = "x1", volatility_parents = "x2"),
    (child_node = "x3", volatility_parents = "x4"),
]

#Initialize an HGF
myHGF = HGF.init_HGF(node_defaults, input_nodes, state_nodes, edges);
```

Now, without any need to initialize our parameters since were are gonna fix or estimate them in the fitting, we declare the two lists of fixed values and priors for the fitting and run the **fit__model** function on the **inputs** and **responses** vectors for one session of one subject that we extracted from the experiment data frame.

```
inputs = subject_20_inputs["1"]
responses = subject_20_responses["1"]
firstinput = inputs[1]

fixed_params_list = (u_x1_coupling_strenght = 1.0,
u_x3_coupling_strenght = 1.0, x1_posterior_mean = firstinput,
x1_posterior_precision = exp(-1.0986), x1_x2_coupling_strenght = 1.0,
x4_posterior_mean = 1.0, x4_posterior_precision = exp(2.306),
x2_posterior_mean = 3., x2_posterior_precision = exp(2.306),
x4_evolution_rate = -10.0, x3_posterior_mean = 3.2189,
x3_posterior_precision = exp(-1.0986), x3_x4_coupling_strenght = 1.0,
u_evolution_rate = 1.0,
)

prior_params_list = (
    action_noise = Truncated(Normal(100,20), 0, Inf),
    x1_evolution_rate = Normal(-3,5),
    x2_evolution_rate = Normal(-7,5),
    x3_evolution_rate = Normal(-3,5),
)

chain = HGF.fit_model(my_agent,inputs,responses,prior_params_list,
    fixed_params_list,NUTS(0.65),1000)
```

Now we can visualize the results of the fitting by plotting the prior and posterior distributions for all the parameters using the function **posterior__parameter__plot**. The result is displayed in figure 6.2.

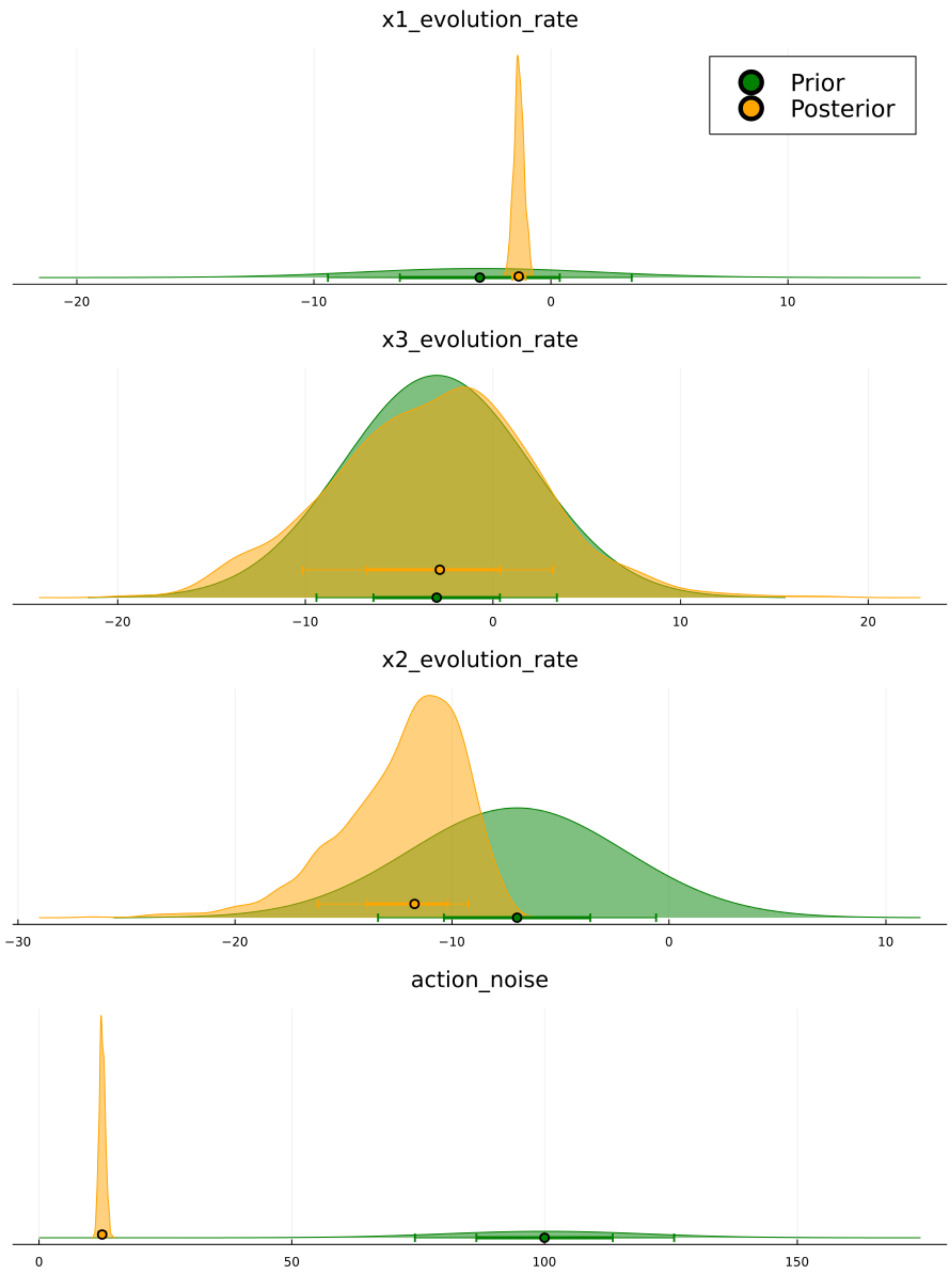

Figure 6.2. Comparison between the prior distributions used for the sampling and the posterior distributions for the parameters.

To look at the evolution of the agent using the median value for the fitted parameters we can give it the parameters by calling twice **set_parameters** once with the fixed one and once with the medians of the fit results extracted with **get_params** and then resetting the agent to load the correct starting values. Then with **give_inputs!** we evolve our agent.

```
fitted_params = HGF.get_params(chain)

HGF.set_params(my_agent,fitted_params)
HGF.set_params(my_agent,fixed_params_list)
HGF.reset!(my_agent)

HGF.give_inputs!(my_agent, inputs)
```

Now, using the plotting function of the package, we can show how the internal state of the agent evolved given the inputs in figures 6.3.

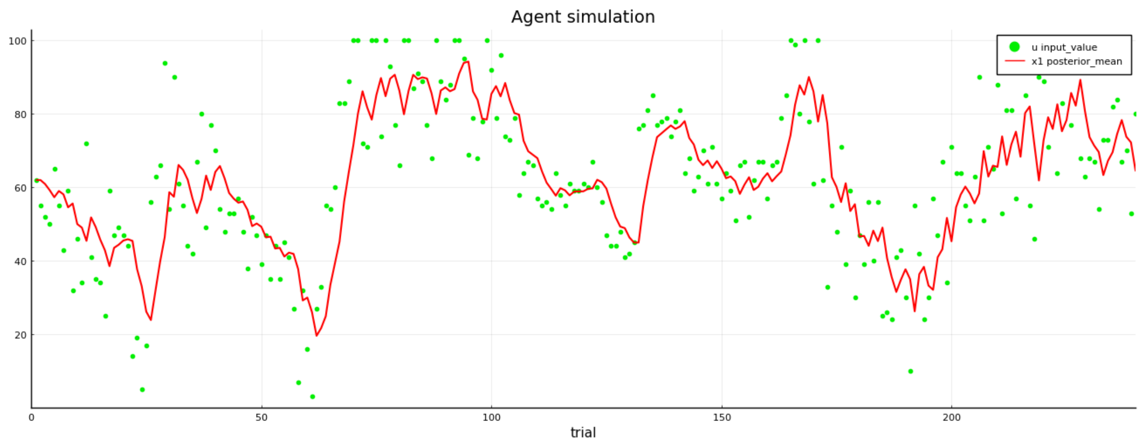

Figure 6.3. Input values given to the subject (green), plotted together the mean value for the state x_1 (red) estimating them.

6.2 multiple datasets analysis

With our new package, it is easy to analyze and fit all datasets for all subjects sessions and store all the information o the posterior distributions in a dictionary that could be then used to extract and save median values and all the needed quantiles. It amounts just to a couple of nested for loops.

```
for ID in keys(inputs_dict)
    for session in keys(inputs_dict[ID])
        inputs = inputs_dict[ID][session]
        responses = inputs_dict[ID][session]
        firstinput = inputs[1]
        fixed_params_list = (u_x1_coupling_strenght = 1.0,
            u_x3_coupling_strenght = 1.0, x1_posterior_mean = firstinput,
            x1_posterior_precision = exp(-1.0986), x1_x2_coupling_strenght = 1.0
                ,
            x4_posterior_mean = 1.0, x4_posterior_precision = exp(2.306),
            x2_posterior_mean = 3., x2_posterior_precision = exp(2.306),
            x4_evolution_rate = -10.0, x3_posterior_mean = 3.2189,
            x3_posterior_precision = exp(-1.0986), x3_x4_coupling_strenght = 1.0
                ,
            u_evolution_rate = 1.0,
        )
        prior_params_list = (
            action_noise = Truncated(Normal(100,20), 0, Inf),
            x1_evolution_rate = Normal(-3,5),
            x2_evolution_rate = Normal(-7,5),
            x3_evolution_rate = Normal(-3,5),
        )

        chain = HGF.fit_model(my_agent,inputs,responses,prior_params_list,
            fixed_params_list,NUTS(0.65),1000)
        chains_dict[ID][session] = chain
    end
end
```

This shows, even more, the convenience of our package since before any of this analysis would have involved multiple files and passing through multiple programs (MatLab, Stan, R).

Chapter 7

Discussion and Future updates

As stated in the beginning, this project is still a work in progress even if we made the first version available to the public in September 2022 during the presentation in Zurich. At its current status, it is possible to use it to create a wide range of HGF structures with just a few lines of code and modify them at pleasure. Simulating and fitting this kind of agents was made possible in a much simpler way than before and it requires much less technical expertise from the user.
We hope this will encourage researchers already working in this framework to experiment more with the possibilities given by bigger and more complex networks while new researchers who may have been discouraged before by the mathematical complexity of this model could now start using it together with (or in substitution to) purely Bayesian or Reinforcement Learning based models.
While the main structure of the package is defined, there is still room left for new features and improvement. New releases will include functions to create new types of nodes together with their set of update equations (e.g. categorical nodes), but also new features and utilities such as functions to add and remove nodes, new premade HGF structures and action models, as well as a new section of functions for model comparison.
We are also working on optimizing the fitting functions as much as possible and planning on adding support for parallelization.
We are currently working to have our project approved as an official package and have it registered in the Julia Registry as soon as possible so to make as easy as possible for whoever would like to try it out to install it on their computers.
After this first release we are currently waiting for researcher to put at test our package in order to collect suggestions on new requested features, improvements on the already present ones and possible bugs that need to be fixed.

Bibliography

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

[2] Mathys D, Christoph. *Hierarchical Gaussian filtering: Construction and variational inversion of a generic Bayesian model of individual learning under uncertainty.* PhD thesis, ETH Zurich, 2012. Artwork Size: 213 p. Medium: application/pdf Pages: 213 p.

[3] Jean Daunizeau, Hanneke E. M. den Ouden, Matthias Pessiglione, Stefan J. Kiebel, Klaas E. Stephan, and Karl J. Friston. Observing the observer (i): Meta-bayesian models of learning and decision-making. *PLOS ONE*, 5(12):1–10, 12 2010.

[4] Weber E, Lilian A. *Perception as Hierarchical Bayesian Inference - Toward non-invasive readouts of exteroceptive and interoceptive processing.* PhD thesis, ETH Zurich, 2020. Artwork Size: 291 p. Medium: application/pdf Pages: 291 p.

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

[6] Christoph Mathys, Jean Daunizeau, Karl J. Friston, and Klaas E. Stephan. A Bayesian foundation for individual learning under uncertainty. *Frontiers in Human Neuroscience*, 5, 2011.

[7] Christoph D. Mathys, Ekaterina I. Lomakina, Jean Daunizeau, Sandra Iglesias, Kay H. Brodersen, Karl J. Friston, and Klaas E. Stephan. Uncertainty in perception and the Hierarchical Gaussian Filter. *Frontiers in Human Neuroscience*, 8, November 2014.

Appendix A

Minimal Julia glossary

This aim of this appendix is not to give a complete description of Julia programming language but only to highlight some concepts and standards peculiar of this language and that were used extensively while writing the package.
Please refer to Julia official website `https://julialang.org/` if you would like to learn more about this language.

- **::** : the double colon is used in Julia to specify the type requirement for a specific argument of a function (e.g. sum(x::Float64, y::Float64) would return an error if the two argument are for example integers).
- **struct**: Julia function to define new types, partially analogous to a class definition in other programming language. "Methods" for this "class" are defined using the type signature and multiple dispatch.
- **Multiple dispatch**: by multiple dispatch we mean the way in which Julia dispatch different calls of a function based on the type of their arguments. This allows to redefine a function with the same name but different type signature. It is then possible to wrap up a variable in a new user defined type using **struct** so that a function will act in a different way on it.
- **!** : by Julia standards an exclamation mark is added after the name of a function if its arguments may be modified.
- **Plots backends**: In Julia multiple plotting packages are available and are grouped by the **Plots** package as backends. Most common backends are **GR**, **Plotly** and **Pyplot**.
- **(;)** : a way to initialize an empty named tuple.
- **Named Tuple**: Data structure similar in concept to a dictionary but immutable and more efficient. It performs better than dictionaries for quantities that do not change or change rarely. As a dictionary it pairs a key of type **Symbol** to a value.