User Guide

Section 1 gives an overview of the purpose of the provided files and folders. Key functionality is practically demonstrated using demonstration scripts.

Section 2 explains the design of how simulation conditions are set and what advantages this design offers.

If you have any questions, I will be happy to answer them at jakub.tomek.mff@gmail.com.

# Folders and Files

## Files

The blue-font entries are classes or functions, while the green-font entries are runnable scripts. The scripts are commented to highlight key functionality they demonstrate.

* DataReporter – A Matlab static class containing functionality underlying the automated model evaluation pipeline. Key functions are:
  + DataReporter.runEvaluationSimulations, which simulates a range of protocols that allow assessment of multiple calibration and/or validation criteria.
  + DataReporter.makeReport , which takes output of the previous function, generates plots illustrating how the calibration/validation criteria are tackled, and it produces a HTML report allows a quick and simple inspection of the model’s performance.
* getCurrentsStructure – A function which converts a matrix of model’s state variables over time into a structure of currents. It also extracts other key variables that are not currents, but which are frequently inspected (i.e., time vector, membrane potential, or cytosolic calcium concentration).
* getStartingState – A function which allows the user to select the starting state to be used by the simulated model.
* model\_Torord – The simulation core of the ToR-ORd model, primarily computing the derivative of state variables over time.
  + One of its inputs is a switch determining whether it outputs the derivatives of state variables (as used by an ODE solver), or whether it returns computed currents and other key variables (this is used by getCurrentsStructure).
* model\_Torord\_dynCl – The simulation core of the dynamic-chloride version of ToR-ORd described here: <https://www.biorxiv.org/content/10.1101/2020.06.01.127043v1>
* model\_Torord\_HCM – A variant of the ToR-ORd model with remodeling induced by HCM as described in our paper.
* modelRunner – The interface between user scrips and the simulation core.
* scriptDemonstration\_1\_SingleRun – A script showing how a simple simulation is defined, running the ToR-ORd model for 100 beats and plotting the membrane potential and cytosolic calcium.
* scriptDemonstration\_2\_ParameterComparison – A script showing how to modify a current’s conductance, plotting the simulation outputs.
* scriptDemonstration\_3\_ControlHCMcomparison – This script compares the behavior of the control model versus the HCM one, also showing how they react to a 50% IKr block.
* scriptDemonstration\_4\_exampleMGA – This demonstrates how to run a multicriterial genetic algorithm to re-fit ToR-ORd to different criteria.
  + The script uses a sample fitness function fitnessTesting, which is stored in the folder ‘gaFitness/’. This function aims to create a model with APD of 400 ms, and with calcium transient amplitude of 200 nM. The function
  + *Sidenote: Feel free to explore how ‘maxTimePerBeat’ is used in modelRunner when a genetic algorithm is used. It defines an ODE event which can terminate the simulation which runs too long, allowing premature killing of simulations that would crash anyway, as described in our paper, Appendix 1-15.1.6.*
* scriptDemonstration\_5\_usingEvaluator – A script demonstrating how to run the model evaluation pipeline.
* scriptDemonstration\_6\_testingCl – A script comparing the baseline and dynamic-chloride model.
* scriptDemonstration\_7\_testingUpdatedPhical – A script comparing baseline and dynamic-chloride models with the versions with updated driving force computation as described in the comments along the publication at <https://elifesciences.org/articles/48890> (the comment referring to the observations by Aditi Agrawal)

## Folders

* data/ – that’s where intermediate results are stored. Importantly, the folder also contains the file ‘oliTraces.mat’, which contains aligned experimental traces of human action potentials (this is required for the automated model evaluation). *These were generated by the lab of Prof. Varró in Szeged and were kindly sent to me by Dr. Oliver Britton, hence the file name.*
* gaFitness/ – here is stored the fitness function used by the sample genetic algorithm.
* htmlStuff/ – this contains various helper files used in generation of the HTML report by DataReporter.
* otherModelCodes/ – alternative model functions are here (so that they don’t occlude the main folder)

# Code logic

The standard use of the codes, as shown in demonstration scripts 1-3, consists of three layers:

1. The user script defines parameters of simulation (pacing rate, number of beats simulated, changes to the baseline model, etc.), and passes these parameters to modelRunner. (afterwards, currents would be typically extracted using getCurrentsStructure, and saved or visualized).
2. In the second layer, in modelRunner, which serves as an interface between the user and simulation itself, the structure of parameters is unpacked and undefined parameters are set to default values. Subsequently, all parameters are passed to the simulation core, which is a model file such as modelTorord\_HCM, solved via ode15s.
3. In the third layer, the model file itself, the derivative of state variables of the model is computed.

This structure of the codes was created based upon previous designs of Matlab codes for cardiac models, trying to bring together their advantages, while minimizing their limitations. There are two typical cases of Matlab model codes:

* Monolithic models where the model code is a single massive chunk of code, which is directly called by user script (or sometimes both simulation and model are defined within one file). This generally runs quickly, but on the other hand it usually does not permit easy modification of the model’s behaviour (i.e., changing conductances etc.), or easy swapping of formulations of single currents, and it leads to accumulation of many near-identical model codes, which is chaotic and prone to errors.
* On the other side of the spectrum, there are structured models (mainly the Heijman-Rudy canine model), which are designed with good practise of software carpentry in mind, being separated into distinct parts serving to define simulations and parameters, and into model code which can be extensively parametrised and which is well-structured and readable. One technical limitation of the Heijman-Rudy model implementation is its use of structures to carry parameters to the model code. While this produces well-readable code, it leads to a fairly large increase in runtime because of the way how Matlab structures are implemented (the slowdown is relatively substantial, increasing the computation time several times).

Our model aims to keep the structured nature of the Heijman-Rudy style of models, while eliminating the slowdown following from the use of structures within the model code. We still use a structure to define the parameters, as this is very convenient for the user, but this structure is unpacked in modelRunner into single variables (which is done only once per simulation), and these single variables are then passed to the model codes such as modelTorord. Therefore, there are no structures accessed in the model function (which is called many times by the ODE solver), maintaining the fast runtime. While this approach means that the model code modelTorord has a very high inputs, they are passed automatically in modelRunner, eliminating the possibility of user error.

Another advantage of having an interface function like modelRunner is that one can simulate any model function, as long as it has the same form of inputs and outputs as model\_Torord[[1]](#footnote-1). This is demonstrated in the pre-packaged script scriptDemonstration\_3\_ControlHCMcomparison, which shows that just by changing param.model, either the control or HCM model can be simulated.

One of parameters that the user can specify is param.extraParams – this is an array (or a cell array) of any length, and it can modify basically anything in the model code, as long as the model code knows how to process the extra parameters. E.g., one may create a model code version where extraParams contain parameters of the hERG Markov model, allowing their exploration (either manually, or refitting via the genetic algorithm or other approach)[[2]](#footnote-2).

1. In addition, if modelRunner is updated to branch calls to ode15s, even models with different inputs/outputs can be accommodated. [↑](#footnote-ref-1)
2. The contents of extraParams can be pushed even to contain switches between different formulations of ionic currents in addition to their parameters. It is thus also possible to run a genetic algorithm on a model code with multiple alternative formulations of, e.g., hERG current, letting the evolution find which of the models can fit the criteria the best. [↑](#footnote-ref-2)