TVB-AdEx Parameter Sweep Tutorial Document

David Aquilué Llorens - 2022

Abstract

The aim of this document is to provide the reader with a basic understanding of the development and structure of the code written during my internship / master thesis at NeuroPSI under the supervision of Jennifer Goldman and Alain Destexhe. The main goal of this library is to be able to run extensive parameter sweeps of the TVB-AdEx brain-scale cortical model on a supercomputer to have a better understanding of the impact the parameters have on the model dynamics.

I would like to express my gratitude to Kevin Ancourt, Michiel Van der Vlag and Sandra Diaz for their help during the development of this code and the execution of the simulations on the Jülich Supercomputing Centre facilities.

Structure

Problem Definition

The TVB-AdEx model is a biologically informed, brain-scale cortical model built using The Virtual Brain (TVB) platform. Using MRI scans, one can divide a scanned brain into different anatomical regions and estimate the strength of the connections between regions using tractography analyses. From this data one can build the TVB-AdEx model as a network of AdEx mean-field models, each mean-field model describing the neuronal activity of one of the anatomical regions in the scanned brain. The interactions between the mean-field models are described by the connectivity matrix, also called connectome, obtained from tractography.

The TVB-AdEx model contains many parameters that needed to be understood and to have reasonable, physiological ranges determined for them. Most of them had already been chosen via biological or mathematical arguments, but there was a subset of parameters whose impact needed to be studied to have a deeper and more general understanding of the model. The Table below shows the characteristics of the parameters chosen and the reason for their choice.

Parameter	Description	Reason of choice	Range	Units
S	Coupling strength between nodes.	Has to be chosen phenomenologically.	[0, 0.5]	No Units
$E_{L,i}$	Leakage reversal potential of AdEx inhibitory neurons.	Resting membrane potential of a neuron might vary depending on external conditions.	[-80, -60]	mV
$E_{L,e}$	Leakage reversal potential of AdEx excitatory neurons.	Resting membrane potential of a neuron might vary depending on external conditions.	[-80, -60]	mV
T	Timescale of the AdEx mean field model.	Has to be chosen phenomenologically.	[5, 40]	ms
b_e	Adaptation strength of excitatory AdEx neurons.	Models the change in neuromodulation that induces transition between AI and UD.	[0, 120]	pA

For each parameter, 16 evenly spaced values are obtained inside the described range. A simulation would have been run for each of the possible combinations of parameter values, which would result in having to analyze 16^5 differently parametrized TVB-AdEx configurations. Fortunately, preliminary results showed that neuronal activity remains silent when $E_{L,i}$ is significantly greater than $E_{L,e}$ a result of the inhibitory populations being much more active than the excitatory ones. For this reason, a threshold was decided from the preliminary studies and only those combinations where $E_{L,i} < E_{L,e} + 4$ mV have been simulated. In the end, a total number of 675,840 different configurations have been analyzed.

Simulating the TVB-AdEx is a computationally expensive process. It typically takes around a minute to simulate one second of activity. For each parameter combination that was going to be studied, at least five seconds of activity needed to be simulated. Additionally, the analyses might take up to a minute to be executed. Clearly, running 675,840 simulations in a desktop computer would have been an extremely time consuming task (almost eight years if no parallelization had been applied). But performing a parameter sweep is an embarrassingly parallel problem, simulations of different parameter combinations do not interact between them, making it relatively easy to exploit multi-core computing.

For these reasons, access to the JUSUF supercomputer in Jülich Supercomputing Centre was solicited and granted for this project. The JUSUF component consists of 187 nodes, each node having two AMD EPYC 7742 @2.25 GHz processors for a total of 128 cores, 256GB DDR4 of RAM and 1TB NVMe for memory. More information on the JUSUF supercomputer can be found in https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/jusuf .

In order to apply for supercomputer access, a thorough profiling of the simulations and of the feature extraction pipeline were performed. A TVB-AdEx simulation of five seconds with its pertaining feature extraction requires less than 600 MB of RAM at any point in time, less than one GB of static memory for I/O operations, and approximately 6 minutes of execution time. Thus, it was possible to make the most out of the JUSUF nodes, being able to simulate and analyze 128 parameter combinations at the same time in each node.

For more information on the mathematical description of the model and its neurological basis, one can refer to:

Jennifer S. Goldman, N´uria Tort-Colet, Matteo di Volo, Eduarda Susin, Jules Bouté, Melissa Dali, Mallory Carlu, Trang-Anh Nghiem, Tomasz Górski, and Alain Destexhe. Bridging single neuron dynamics to global brain states. Frontiers in Systems Neuroscience, 13, December 2019.

Jennifer S. Goldman, Lionel Kusch, Bahar Hazal Yal¸cinkaya, Damien Depannemaecker, Trang-Anh E. Nghiem, Viktor Jirsa, and Alain Destexhe. A comprehensive neural simulation of slow-wave sleep and highly responsive wakefulness dynamics. bioRxiv, September 2021.

Matteo di Volo, Alberto Romagnoni, Cristiano Capone, and Alain Destexhe. Biologically realistic mean-field models of conductance-based networks of spiking neurons with daptation. Neural Computation, 31(4):653–680, April 2019.

Yann Zerlaut, Sandrine Chemla, Frederic Chavane, and Alain Destexhe. Modeling mesoscopic cortical dynamics using a mean-field model of conductance-based networks of adaptive exponential integrate-and-fire neurons. Journal of Computational Neuroscience, 44(1):45–61, November 2017.

JUSUF – The supercomputer and script execution

In order to be able to use the JUSUF supercomputer, the user needs to be granted access through SSH.

The parallelization applied to tackle this problem is fairly simple, since we are dealing with an embarrassingly parallel task. We will apply the same way of working constantly, for different tasks.

The JUSUF supercomputer makes use of an MPI software that is executed when calling the *sbatch* command. For more information one can refer to the JUSUF documentation (https://apps.fz-juelich.de/jsc/hps/jusuf/cluster/batchsystem.html#writing-a-batch-script).

Partitions and Directories

In order to tackle the parameter sweep of the TVB-AdEx, the two main partitions of the JUSUF supercomputer have been used: *Scratch* and *Project*. The user will have a designed folder in each one of the partitions that only they (and the people in the same project) will be able to access.

Scratch: The working directory. It is a temporary storage partition to be used during the execution of the scripts. For instance, in our case, each five second simulation of the TVB-AdEx needs to store the results in a ~500 MB folder. If we have many simultaneous simulations, this will require a considerable amount of storage, thus, we use S*cratch* for that. It also allows us to store many files produced by our scripts. Backups of the data in this partition are not done regularly.

Project: Where we store our codes, final results and small data. This directory is regularly backed up. From here we write, modify and submit our scripts.

In the *JUSUFlike* folder in this code repository you will see two main folders, *Project* and *Scratch*. They correspond to the files that we would have in our project's directories in the two JUSUF partitions.

Batching scripts

The shell scripts that have been submitted using *sbatch* have all a similar structure to the one in the code below. In it, one can see the different settings that have to be selected to run a parallel execution of a python script.

```
#!/bin/bash -x
#SBATCH --account=icei-hbp-2022-0005
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=128
#SBATCH --time=00:30:00
#SBATCH --output=output_%j.out
#SBATCH --error=error_%j.er
#SBATCH --mail-user=daquilue99@gmail.com
#SBATCH --mail-type=ALL
#SBATCH --partition=batch
#SBATCH --job-name=parsweep_newconnectome

source activateTVB
srun python3 HPC_check.py
```

Once the shell script that contains this information is submitted using *sbatch*, JUSUF's scheduler will decide when to execute it and run it in its due time.

In this case, we would use one node of the supercomputer, executing the same *HPC_check.py* script in each one of its 128 cores (1 process = 1 execution of the *HPC_check.py* script -> on 1 core) for a maximum of 30 minutes, or until the script's execution is finished.

Thus, we run, on each core, the same identical python script. However, we will see that there is one variable inside the script that will change for each process, allowing us to easily tackle embarrassingly parallel tasks.

MPI execution of Python scripts

The *HPC_check.py* file will be a Python script that exploits the *mpi4py* library to tackle the parallelization of the problem. The script will contain the following lines at its top (together with the imports of all the libraries and functions):

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

The key concept here is the **rank** variable. Thanks to *mpi4py*, the *rank* variable will have a different number for each core that is executing the script. That is, we run on one node this same script 128 times, each core executes the same lines of code, but the *rank* variable is different for all the cores, going from 0 to 127.

This is the basis of the parallelization of this project, the rank variable can be used to index a list of sub-lists for instance, where each sub-list contains a different combination of parameters of the TVB-AdEx model, allowing us to run different parameter combinations at the same time.

Preparing the data

As previously mentioned, our objective is to simulate 675,840 different TVB-AdEx parameter combinations of 5 parameters. We represent this data as a (675840x5) matrix (***M***) where each row represents a different parameter combination and each column holds the values of a parameter.

We will also split this data in multiple chunks (by rows), to ease the parallelization process and to be able to simulate a subset of the whole dataset at a time. It might be hard to find errors, correct mistakes, etc. when we run everything at once during long simulation times. Instead, with smaller chunks of the dataset, everything is more manageable.

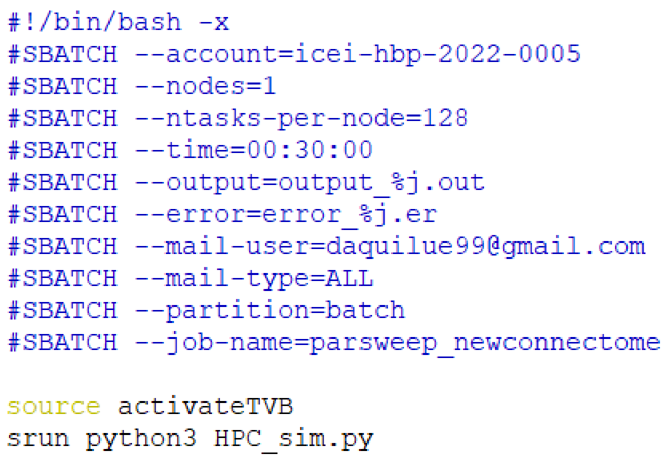
Thus, when executing the *generate_data_chunks.py* (as a simple python script, no need to batch it or anything) we will store the chunks of the ***M*** in *JUSUFlike/Project/Data/data_chunks*. In *JUSUFlike/Project/Data/* we also store the *assignment.npy* file. This array is used by the script that simulates the parameter sweep (*HPC_sim*.py) to know which core has to run which parameter combinations. To obtain this matrix we will need to declare the computational resources we want to use for the simulation (how many nodes, cores per node, how many parameter combinations per core, etc.) in the *generate_data_chunks.py*.

Additionally, executing this script will generate the necessary folders in the *Results* and *Indicators* folders in the *Scratch* partition.

Running the scripts

Once the data has been prepared and stored in its corresponding directory, we can proceed to run the parameter sweep. It is best to not batch the entire parameter sweep in a single run. Instead, it is best to submit multiple jobs, each job simulating and analyzing a batch of chunks. That way, if something goes wrong, it will be much easier to track and fix the error and the amount of computational resources lost won't be as high.

The shell script that we will use to submit the batch contains the following lines:



```
#!/bin/bash -x
#SBATCH --account=icei-hbp-2022-0005
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=128
#SBATCH --time=00:30:00
#SBATCH --output=output_%j.out
#SBATCH --error=error_%j.er
#SBATCH --mail-user=daquilue99@gmail.com
#SBATCH --mail-type=ALL
#SBATCH --partition=batch
#SBATCH --job-name=parsweep_newconnectome

source activateTVB
srun python3 HPC_sim.py
```

Bear in mind that the settings shown here serve simply as an example, they will need to be modified to fit the requirements of the simulations (for 675840 parameter combinations this is clearly very little time for just one node).

The *HPC_sim.py* is the python script that, together with the data already prepared, the *assignment.npy* matrix, and the MPI rank, will 1) choose what parameter combinations to simulate, 2) will run a TVB-AdEx simulation with those parameter combinations, 3) will apply the analyses stored in the *analysis.py* script, and 4) save the results in a file.

The main goal now is to store the results in a 675840x(5+ R) matrix (***R***), where R is the number of different values that we have obtained as results (such as mean excitatory firing rate, mean Functional Connectivity in the network, relative power of frequency bands, etc.) The first five columns contain the values of the parameters, that way, one row contains both the parameter combination used for a simulation and the results of that simulation.

However, to obtain this matrix, we still have to go through two more steps.

Each time a parameter combination is simulated and analyzed by the *HPC_sim.py,* we obtain a row vector corresponding to one row of the ***R*** matrix. This vector is stored in a .npy file in the *Scratch* partition, in the *Scratch/Results/chunk_id/* directory, and its name will contain the values of the parameter, so that we are able to distinguish all the different combinations. It is important to note that using this method, we will generate many small files. That's why we use the *Scratch* partition, since it allows us to write and store many files.

Additionally, for each parameter combination that has been correctly simulated, we generate an empty .txt file in *Scratch/Indicators/chunk_id/* titled *name_of_parameter_combination_file_COMPLETED.txt* that we will use to know that there has not been any problem with that parameter combination.

Checking the results and obtaining the final results

To check if all the simulations have been completed correctly, we can make use of the *HPC_check.py* script. Batching it in a simple job such as the one in Figure \ref, we can check each chunk at the same time, in parallel. This script will check that each chunk folder in *Scratch/Indicators/* contains all the corresponding .txt files. In */Project/Data/rem_chunks/* a .npy file will be generated for each chunk that does not contain all their corresponding *_COMPLETED.txt* files. The .npy file contains a matrix where each row is the parameter combination that has not been correctly executed and analyzed. That way, we can re-run the *HPC_sim.py* script using the data from the */Project/Data/rem_chunks/* folder and finally obtain our completed parameter sweep.

The *HPC_check.py* script will also merge all the .npy files of the parameter combinations and merge them into bigger matrices. With the *batch_files* function from the *processing_results.py* script, we can select how large we want those matrix to be (number of rows, number of parameter combinations per matrix). The main rationale behind storing the final results in different matrices, except of one big ***R*** matrix is that it might become too large to load directly on memory when performing the analyses on a regular computer. Instead, if we fraction this matrix in different parts and only load those results that are interesting to us, we will run into less memory problems.

Processing and analyzing the results

Finally, there is an extensive collection of python functions that can be used to perform analyses on the results. They can be found in the */Project/Codes/processing_results.py* script, as well as inside the multiple jupyter notebooks in the repository.

DISCLAIMER: In no way this repository is intended to be a module, library or pipeline that is ready to use. It is a very specific set of scripts, designed for the context and problem that I needed to solve and, therefore, is hardly generalizable. However, for similar problems, I hope that it can provide the reader with some useful ideas and snippets of code.

If you have any question or doubt, don’t hesitate to contact me: daquilue99@gmail.com