Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2021

Randomised time-step methods for spiking neural network simulations

Student:	Fabio Deo
CID:	01338063
Course:	EIE4
Project Supervisor:	Dr Daniel Goodman
Second Marker:	Dr Krystian Mikolajszyk

Contents

List of Symbols ii

List of Figures iii

Acknowledgments v

Abstract vi

1 Introduction 1

1.1 Spike-Based Neural Processing 1

1.2 The BRIAN Simulator 2

2 Background 3

2.1 Time-step Analysis 3

2.2 Efficient Time-stepping Numerical Methods 4

2.3 Variable Time-Stepping Methods 5

3 Analysis and Design 6

3.1 Model Architecture 6

3.2 Randomised Time-stepping Methods Design 8

3.2.1 Shared Random Time-stepping Design 9

3.2.2 Subgroup-specific Random Time-stepping Design 9

3.2.3 Neuron-Specific Random Time-stepping Design 10

4 Implementation **12**

4.1 BRIAN2 Model Replica 12

4.2 Shared Random Time-stepping Implementation 16

4.3 Subgroup-Specific Random Time-stepping Implementation 18

4.4 Neuron-Specific Random Time-stepping Implementation 19

5 Testing **21**

5.1 Synchronisation Measure 21

5.2 Testing the Model Replica 22

5.3 Testing the Randomised Time-stepping Methods 24

6 Evaluation and Results **25**

6.1 Mean Squared Error Measure 25

6.2 Fixed Time-stepping Evaluation 25

6.3 Shared Random Time-stepping Evaluation 26

6.3.1 All-to-all Coupling 26

6.3.2 Random Coupling 28

6.4 Subgroup-Specific Random Time-stepping Evaluation 29

6.5 Neuron-Specific Random Time-stepping Results 31

7 Conclusion **34**

8 User guide **35**

9 Appendix **38**

A Python model 38

B Evaluators of synchrony 39

C Fixed random time-stepping implementation 40

D Shared random time-stepping implementation 41

E Neuron-Specific Random Time Stepping Implementation 45

F Additional Syncronisation Analysis 49

List of Symbols

Symbol	Description
dt	Integration time-step
N	Number of neurons
g_l	Conductance of the voltage-independent leak current
V_l	Reversal potential of the voltage-independent leak current
V_{rest}	Resting membrane potential
C	Membrane capacitance
τ	Passive membrane time constant
τ_1	First time constant for the bi-exponential synaptic current
τ_2	Second time constant for the bi-exponential synaptic current
θ	Membrane potential threshold
I_0	Constant external drive
$\bar{I_{syn}}$	Coupling strength
g	Number of subgroups defined in SSRTS
dt_{fixed}	Middle value of the time-step random range in any randomised time-stepping method
dt_{rad}	Radius of the time-step random range in any randomised time-stepping method
ts	Number of time-steps already simulated

List of Figures

1 Leaking Integrate and Fire neurons behaviour from (Kabasov et al. [2]). 1

2 Synchronisation artifacts for large dt (the red and blue curve denote the membrane voltages of two different neurons of the network). 3

3 Comparison of the performance of various numerical methods from (Shelley, et al. [5]). 4

4 High-level design of the shared random time-stepping method. 9

5 High-level design of the subgroup-specific random time-stepping method. 10

6 High-level design of the neuron-specific random time-stepping method. 10

7 Single neuron synaptic bi-exponential current. 13

8 Implementation design for the neuron-specific random time-stepping method using multiple BRIAN2 `NeuronGroup` and `Synapses` objects. 19

9 Original plot of the synchronisation measure Σ over I_{syn}^{-} for various time-steps dt (left) and results replicated in Brian2 (right). 22

10 Original plot of the synchronisation measure Σ over I_{syn}^{-} for various initial degrees of synchrony c (left) and results replicated in Brian2 (right). 23

11 Synchronisation measure for FTS simulations with different time-steps dt (left) and respective MSE scores (right). 26

12 Synchronisation measure for FTS with random 0.5 coupling simulations with different time-steps dt (left) and respective MSE scores (right). 27

13 Synchronisation measure for SRTS simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right). 27

14 Synchronisation measure for SRTS simulations with different random time-step ranges dt_{rad} (left) and respective MSE scores (right). 28

15 Synchronisation measure for SRTS with random 0.5 coupling simulations with different time-steps dt (left) and respective MSE scores (right). 29

16 Synchronisation measure for SSRTS(2, 0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right). 30

17 Synchronisation measure for SSRTS(x, 0.5) simulations with time-step $dt_{fixed} = 0.01$ (left) and MSE scores for SSRTS(x, 0.5) simulations with various time-steps (right). 31

18 Synchronisation measure for SSRTS(2, x) simulations with different random time-step ranges dt_{rad} (left) and respective MSE scores (right). 32

19 Synchronisation measure for NSRTS(0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right). 33

20 Synchronisation measure for SSRTS(4, 0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right). 49

21 Synchronisation measure for SSRTS(8, 0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right). 49

Acknowledgments

I would firstly like to express my deep gratitude towards my supervisor Dr. Daniel Goodman for allowing me to work with him and for the guidance and support provided throughout the project.

Looking back on the people who most helped me kick-start my university career at Imperial, I cannot fail to mention Francesco Franco, whose generous advice led me towards this studies and persuaded me to always move forward.

I would also like to thank my friends here in London for sharing with me every important moment of my life over the past four years, for their comforting words and support through the bad times and their invaluable company in all of the fun bits. A special mention goes to Alessandro Bullitta, whose constant presence in the last three years of my life has shined a light over my experience at Imperial, to Pietro Giraudi for blessing me with his wisdom and thoughtful advice and Marco Selvatici for mentoring me and easing many of the tough challenges our degree threw at us.

There are no words to express my gratitude to Massimo Giampetraglia, whose life-long friendship feels each day more like a brotherhood. You were able to make me feel supported and encouraged every step of the way despite the distance.

Finally I would like to take a moment to express my deepest appreciation towards the people who made all of this possible, my family. I will never find the words to thank my mum for raising me on her own without ever making me feel the lack of a second parent, my brother for putting up with me for the last twenty-two years and my grandparents and uncle for doing just the most outstanding job at helping my mum in raising me and my brother. Mamma, Fulvio, Nonno, Nonna e Zio grazie di aver supportato e appoggiato incondizionatamente ogni mia decisione. Tutti i traguardi che ho raggiunto negli ultimi quattro anni e che raggiungerò in futuro sono anche merito vostro.

Abstract

Spiking Neural Networks often suffer from artificial synchronisation events due to numerical integration neglecting differences in neurons membrane potentials which are smaller than the used simulation time-step. This report presents three new integration methods, based on introducing inhomogeneity in the network integration process by randomising the definition of the time-step dt.

The first method (Shared Random Time-Stepping) consists of a dynamic change of integration time-step for each numerical iteration, which is still shared by all neurons present in the network. The second method (Subgroup-Specific Random Time-Stepping), de-synchronises the integration of neurons by dividing the network into subgroups. Neurons in different subgroups will be integrated using different time-steps, which ultimately mitigates synchronisation events among subgroups. The third method (Neuron-Specific Random Time-Stepping) introduces an additional layer of randomness of the network by instantiating a new random time-step value for each individual neuron at each iteration.

The three novel integration approaches have been validated against a replica of the all-to-all coupled spiking neural network defined in (Hansel, Mato et al. [4]) using the BRIAN2 simulator. The novel methods performance have been finally compared to their fixed time-step version in terms of synchronisation-aversion and computational complexity, returning a 25%, 270% and 890% improvement in performance, respectively.

1 Introduction

1.1 Spike-Based Neural Processing

Spiking Neural Networks (SNN) make use of bio-mimetic algorithms to attempt to replicate more closely the biological learning process undertaken by the human brain. In classical Artificial Neural Networks (ANN) neurons forward information to one another at each iteration, so that they can learn by processing this flow of data and update their weight through back-propagation. However this communication mechanism differs from the one used by biological neurons. The human brain consists of interconnected neurons that communicate through electrical impulses called action potentials, which are triggered only when certain environmental conditions arise. In particular, each neuron has a membrane voltage, which is dynamically modified by incoming electrical impulses. When the voltage exceeds a level named firing threshold, a spike is triggered and the membrane voltage is immediately reset to its initial value. The connections among neurons that allow them to communicate with each other are the synapses. This spiking model is commonly known as Integrate and Fire (IF) model. Leaking IF models are also extremely common, as they also mimic the leakage in membrane voltage present in biological neurons. Their behaviour over time follows a pattern similar to the one visualised in Figure 1.

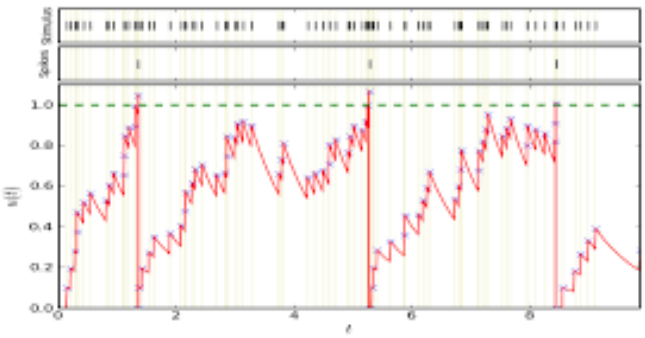

Figure 1: Leaking Integrate and Fire neurons behaviour from (Kabasov et al. [2]).

The more biologically accurate architecture of IF and LIF neural network models comes with a great cost in computational overhead. Providing neurons with internal states to represent their voltage and synaptic current, which evolve over time according to predefined differential equations can be significantly more complex than simple ANN architectures. Furthermore, time must be finely discretised in order not to incur

in artificial synchronisations among the neurons' membrane potentials, as discussed in more detail in Section 2.1.

1.2 The BRIAN Simulator

One of the most widely used simulators for spiking neural network simulations is BRIAN. BRIAN is a free, open source clock driven simulator (Goodman, Brette [3]). It can easily be downloaded, imported and used in the Python programming language. Its clock-driven behaviour implies that a time-step dt is defined to discretise events happening during the simulation. Neurons behaviour is mapped using differential equations, which can be easily defined in BRIAN and are automatically interpreted in the translation to C++ code. BRIAN does not currently support a randomised time-step method to execute neural network simulations, therefore one of the objectives of the project is to integrate such feature in the simulator, if proven to be computationally more efficient than known alternatives. Throughout this project version two of the BRIAN simulator software BRIAN2 has been used.

2 Background

2.1 Time-step Analysis

Defining a suitable integration time-step is paramount, as numerical integrations are considered the main bottleneck in efficient large network simulations. This is because a large dt leads to synchronisation artifacts among neurons throughout the simulation, as observed by (Hansel, Mato et al. [4]), i.e. neurons eventually stabilised to asymptotic states. This behaviour has been identified as a consequence of the inaccuracy in resetting the potential following a spike. Small differences amongst various neurons are discarded, eventually resulting in synchronised membrane potentials. This issue can be mitigated by performing denser simulation, i.e. reducing the time-step magnitude. Nonetheless, this solution is often non-viable due to a significant increase in computational complexity. Figure 2 visualises in a simple diagram the synchronisation issues that could arise due to excessively large time-steps dt in a two-neuron SNN simulation.

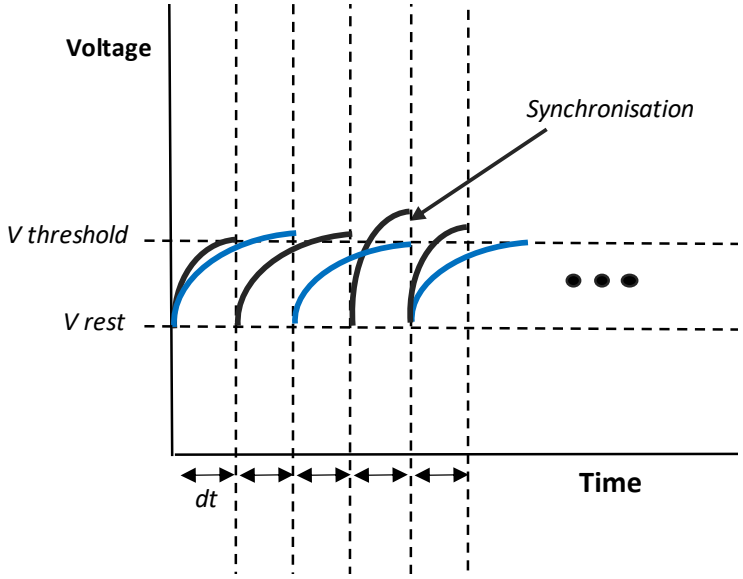

Figure 2: Synchronisation artifacts for large dt (the red and blue curve denote the membrane voltages of two different neurons of the network).

As observed in the figure above, the membrane potentials of the two neurons are not instantaneously reset when reaching the membrane voltage threshold because their state variables are integrated discretely on each time-step. Therefore, a large dt value, as the one illustrated above, causes delays in detecting a spike, eventually leading neurons to synchronise their membrane potential curves. This issue is particularly

significant in homogeneous networks, i.e. networks that are constituted by neurons coupled all-to-all that also share the same synaptic impulse function. This is because in such a setting, once two neurons have synchronised their voltages, it will be impossible for them to receive different synaptic inputs, hence they will never de-synchronise. The model replicated in this project satisfies the homogeneity conditions discussed above, which make it very sensitive to synchronisation artifacts, as discussed in more detail in Section 4.2.

2.2 Efficient Time-stepping Numerical Methods

The easiest approach to define a more efficient time-stepping function is to build more accurate integration schemes, leveraging higher order numerical methods to achieve better approximations. While this course of actions can delay the origin of synchronisation artifacts, it retains an intrinsic error margin, derivable from the synaptic-induced conductance equation, as explained in (Shelley, et al. [5]). Figure 3, shows to what extent the new numerical methods investigated in (Shelley, et al. [5]) compensate for the choice of a larger magnitude of dt.

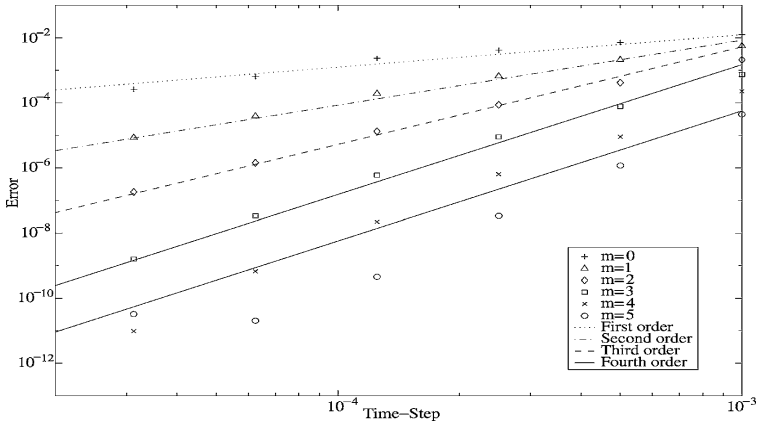

Figure 3: Comparison of the performance of various numerical methods from (Shelley, et al. [5]).

(Hansel, Mato et al. [4]) performed a similar study on the model that was replicated in this project and used as a benchmark to assess the performance of the innovative random time-stepping methods introduced in Section 3.2. In the interest of brevity and clarity, this report will further address this study as the "original model" or "original paper". In the original paper, second order Runge-Kutta with interpolation are proven to be at least a degree of magnitude faster than other numerical methods such as Standard RK2, Standard Euler and Euler with interpolation. Therefore, although higher-order schemes do not provide a full solution

to synchronisation artifacts in neuronal networks, they radically improve the performance of the simulation. Without them, the computational time required to simulate the network would rapidly become unfeasible as the complexity of the model architecture increases.

2.3 Variable Time-Stepping Methods

This project aims to investigate if a variable and randomised time-stepping strategy mitigates the synchronisation activity within the network. This theory has been explored for the first time by (William Lytton, Micheal Hines [6]). They defined a new variable time-stepping integration method called `lvardt`, that leverages the differences in spiking activity to tailor the magnitude of neuron-specific time-steps. In their method, neurons that show an active synaptic behaviour are re-initialised with smaller time-steps upon firing, while the ones that undergo longer idle periods are progressively re-initialised with a larger dt. This dynamic modification of each neuron's individual time-step integrator produces remarkable results in networks with significant synaptic activity, as the computational power spared on idle neurons is used on denser numerical iterations on highly active neurons, improving the overall results and reducing the risk of synchronisation artifacts. However, it must be taken into account the large overhead complexity caused by the re-instantiation of the time-steps based on each neurons recent behavior and the computational cost of having an integrator synchroniser ti make sure that state-variables stay up-to-date throughout the simulation. Nonetheless, the results presented by (William Lytton, Micheal Hines) suggest that this innovative variable time-stepping method outperforms fixed alternatives for reasonably complex and active networks. The network analysed in their research was developed in NEURON, a spiking neural network simulator that provides variable time-step integration functionalities (M. L. Hines, N. T. Carnevale [11]). This project will focus on implementing three simpler versions of the `lvardt` method based on the same varying time-step principle, which are presented in more detail in Section 3.2.

3 Analysis and Design

3.1 Model Architecture

This project is based on a replica of the original model by (Hansel, Mato et al. [4]) in BRIAN2, which will be used to test the efficiency and accuracy of the novel random time-stepping methods designed. The original network encompasses 128 identical excitatory neurons coupled all-to-all. The following differential equation describes the evolution of the neurons' membrane potential over time:

$$C \cdot \frac{dV}{dt} = g_l(V - V_l) + I_{syn}(t) + I_0 \tag{3.1}$$

Furthermore, Table 1 summarises the set of parameters defined in the original model.

Parameter	Description	**Value**
N	Number of neurons	128
g_l	Conductance of the voltage-independent leak current	$0.1\ \frac{mS}{cm^2}$
V_l	Reversal potential of the voltage-independent leak current	$-60\ mV$
V_{rest}	Resting membrane potential	$-60\ mV$
C	Membrane capacitance	$1\ \frac{\mu F}{cm^2}$
τ	Passive membrane time constant	$\frac{C}{g_l} C = 10\ ms$
τ_1	First time constant for the bi-exponential synaptic current	$3\ ms$
τ_2	Second time constant for the bi-exponential synaptic current	$1\ ms$
θ	Membrane potential threshold	$-40\ mV$
I_0	Constant external drive	$2.3\ \frac{\mu A}{cm^2}$
$\bar{I_{syn}}$	Synaptic current coupling strength	range $[0, 1]\ \frac{\mu A}{cm^2}$

Table 1: Network parameters defined in the original model.

A neuron triggers an impulse when its membrane voltage V reaches the membrane potential threshold θ. After the spike, the membrane voltage of the spiking neuron is instantaneously reset to the resting voltage

V_{rest} according to the following equation:

$$V(t_0^+) = V_{rest} \quad if \quad V(T_0^-) = \theta \tag{3.2}$$

No refractory period is considered in the original model definition. This design choice will be retained in the BRIAN2 simulator model replica. Furthermore, the synaptic current $I_{syn}(t)$ evolves over time as modeled by Equation (3.3):

$$I_{syn}(t) = \frac{\bar{I_{syn}}}{N} \sum_{neurons} \sum_{spikes} f(t - t_{spike}) \ , \tag{3.3}$$

where:

$$f(t) = \frac{1}{\tau_1 - \tau_2}(e^{-\frac{t}{\tau_1}} - e^{-\frac{t}{\tau_2}}) \tag{3.4}$$

Section 4.1 explains in more detail how to translate the model architecture described above in BRIAN2 compatible differential equations that can be fed as input to the `NeuronGroup` and `Synapses` classes to ultimately reproduce the model. The original model also specifies how to initialise neurons' internal states. At $t = 0$ all synaptic currents I_{syn} are zero and the neurons voltages are chosen in the range $[-60, -40]mV$ according to the equation

$$V_i(0) = \tau I_0 + (V_{rest} - \tau I_0)e^{-c\frac{(i-1)T}{N\tau}} \ , \tag{3.5}$$

where $i = 1, ..., N$, $0 \leq c \leq 1$ and T is defined as

$$T = -\tau * ln(\frac{\tau I_0 - \theta}{\tau I_0 - V_{rest}}) \tag{3.6}$$

The value of T denotes how long a neuron takes to fire without interactions, while the coefficient c is the degree of initial synchrony. If c is set to 1, the membrane voltages of the neurons will spread out following a uniform distribution in the range $[-60, -40]mV$. Inversely, for $c = 0$ all neurons will be initialised with the same membrane voltage, which will be equal to V_{rest}.

3.2 Randomised Time-stepping Methods Design

Fixed time-stepping (FTS) methods, as the name suggests, consist of using a global static time-step, that does not change throughout the course of the network simulation. Fixed time-steps are trivial to implement in BRIAN2, since the internal variable `defaultclock.dt` that regulates the integration time-step used in the simulation can be overridden to any `float` value. For this reason, using a fixed dt does not add any computational overhead to the network simulation. However, due to their lack of flexibility, FTS methods are also more prone to incur in the synchronisation events discussed in Section 2.2. Therefore, a more complex variable approach has been investigated by (William Lytton, Micheal Hines [6]). As introduced in Section 2.3, its significant performance improvements have been attributed to the lack of unnecessary numerical integration computations for largely inactive neurons. The computational power saved is dynamically redistributed to actively spiking neurons via the usage of smaller time-steps, which increases the integration accuracy of their internal states.

This project will not address such a complex methodology in choosing the time-stepping size, instead focusing on researching the effects of allocating random time-steps on the network's performance. This design choice comes from the assumption that fixed time-step methods greatly suffer from synchronisation artifacts primarily due to the fact that all neurons are updated in a predictable and synchronous fashion. Therefore, introducing just a small degrees of randomness allows to retain a simple network architecture, while still potentially resulting in significant improvements in the synchrony-aversion capabilities of the network. Three randomised time-stepping methods have been embedded in the original model replica. Consequently, their effects on the model performance have been thoroughly analysed and discussed in terms of execution time and synchronisation activity. The three novel time-stepping method designs are presented individually in the following sections.

3.2.1 Shared Random Time-stepping Design

The first method that has been designed and evaluated in this project is shared random time-stepping (SRTS). As the name suggests, SRTS retains the concept of a unique, global integration time-step shared by all neurons. However, on every iteration the magnitude of *dt* is re-initialised to a random value within a predefined range. As shown in the high-level design of the SRTS method in Figure 5, this new approach introduces a low degree of randomness inside the network, as the neurons' internal states are still updated in a synchronous fashion throughout the simulation.

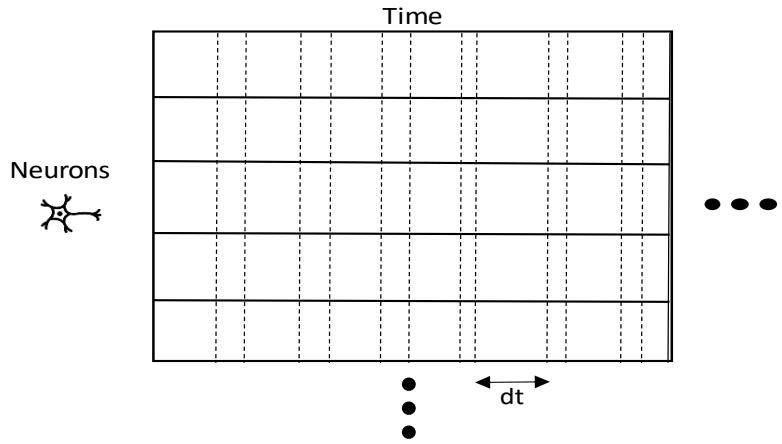

Figure 4: High-level design of the shared random time-stepping method.

Due to its simple design, SRTS does not carry significant computational overheads, since it only consists of one extra calculation of complexity $O(1)$, i.e. the computation of a new random time-step on each iteration. Further details on the integration of this method in the BRIAN2 simulator code-base and the analysis of its performance with respect to FTS are provided in Section 4 and 6 respectively.

3.2.2 Subgroup-specific Random Time-stepping Design

Subgroup-specific random time-stepping (SSRTS) tries to overcome the negative effects caused by synchronous updates still present in SRTS by dividing the network into subgroups of neurons. Only the neurons within the same subgroup share the same randomly initialised time-step, that remains constant for the full length of the simulation. In doing so, SSRTS reduces the likelihood of synchronisation activity between

neurons from different subgroups by introducing disparity in their updating process.

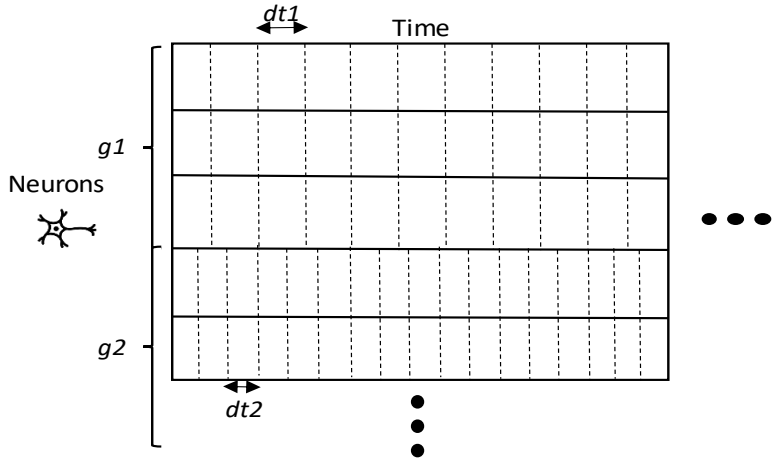

Figure 5: High-level design of the subgroup-specific random time-stepping method.

In terms of complexity overhead, SSRTS introduces only g extra operations performed during the initialisation of the network, where $g$is the number of subgroups of neurons. These account for a one-off additional complexity of $O(g)$, that can be approximated to $O(1)$ since the number of subgroups g is generally very constricted.

3.2.3 Neuron-Specific Random Time-stepping Design

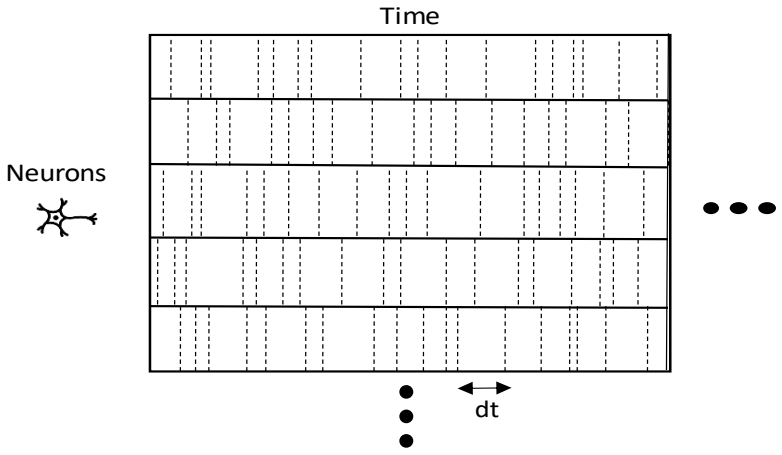

Figure 6: High-level design of the neuron-specific random time-stepping method.

The highest degree of randomness was introduced in the original model by neuron-specific time-stepping

(NSRTS). NSRTS is based on computing a different randomly initialised time-step on every iteration and for each neuron individually. Figure 6 visualises an example of a neuron-specific random time-stepping network architecture.

Although NSRTS achieves higher inhomogeneity in the network, it also carries a complexity overhead for each iteration of $O(N)$, where N denotes the number of neurons in the network, since the time-step is re-initialised for each neuron individually. Therefore, for NSRTS to outperform other methods, the improvement in synchrony-aversion achieved must be significant enough to outperform FTS despite using slightly larger time-step magnitudes on average. This is because the performances of two methods are comparable only when they carry similar computational costs.

4 Implementation

This section first presents the practical work performed in order to reproduce the original model in Python using the BRIAN2 simulator library, followed by the adjustments made to integrate the three randomised time-stepping methods proposed in Section 3.2. The latter encompasses the introduction of new features in the C++ auto-generated project as well as new Python simulation code developed from scratch. For the sake of clarity, terminal line commands have been highlighted in bold (e.g. **make**), variables, functions and data-types in typewriter (e.g. `np.array`) and folders or file names in Italic (e.g. *output*).

4.1 BRIAN2 Model Replica

In order to replicate the model architecture defined in Section 3.1 in BRIAN2 compatible code, standard differential equations to describe both the membrane potential and synaptic current of the neurons must be derived from the equations provided in the original paper. The differential equation for the membrane voltage as defined in the original paper is reported in Equation 3.1. Since it was initially given in ODE form, it needs very little manipulation to meet BRIAN2 standards. Dividing both sides by the membrane capacitance C yields the standard form:

$$\frac{dV}{dt} = \frac{1}{C}\left[g_l(V - V_l) + I_{syn}(t) + I_0\right] \tag{4.1}$$

The standard equation is included in the code as part of a string as shown in Listing 1. The model parameters used in the equations must be defined by explicitly specifying the units, as shown in Listing 3 in Appendix A.

```
eqs = ''' dV/dt = (-gl*(V - Vl) + I_syn + Io) / C : volt '''
```

Listing 1: Definition of the voltage differential equation in BRIAN2.

Inversely, the synaptic current is provided in the original paper in integral form, as reported in Equation (3.3). Hence, it must be transformed into the standard ODE form accepted by BRIAN2, before it can be properly interpreted by the Python model. The synaptic current used in the original model is bi-exponential,

thus it can be re-written in differential form by defining a system of two differential equation. Specifically, the system of ODEs that translates the synaptic current of the original model is of the form

$$\begin{aligned} \frac{dI_{syn}}{dt} &= \frac{(k \cdot s - I_{syn})}{\tau_1} \\ \frac{ds}{dt} &= \frac{-s}{\tau_2} \ , \end{aligned} \tag{4.2}$$

where the constant k is used to re-scale the curve to match the magnitude of a single synaptic impulse shown in Figure 7.

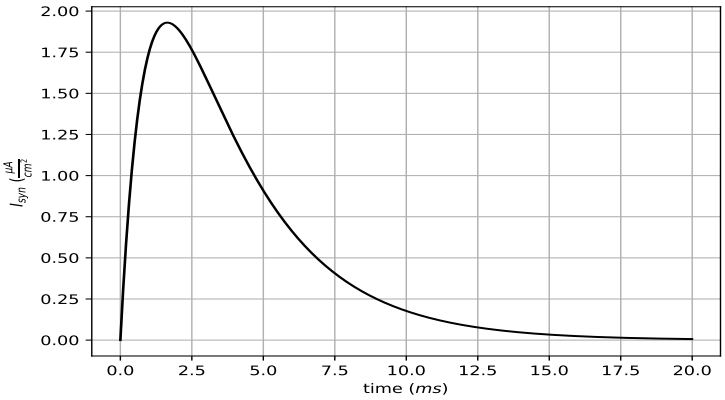

Figure 7: Single neuron synaptic bi-exponential current.

To find the value of k for which the pair of differential equations in Equation (4.2) equates the amplitude of a single synaptic impulse, the ODE system is solved for the initial conditions specified in the original paper, as shown in Equation 4.9.

$$\begin{aligned} \frac{df}{dt} &= \frac{(k \cdot s - f)}{\tau_1} && \text{with} \quad f(0) = 0 \\ \frac{ds}{dt} &= \frac{-s}{\tau_2} && \text{with} \quad s(0) = 1 \end{aligned} \tag{4.3}$$

The variable s changes according to a divisible ordinary differential equation, hence can be easily solved by

integrating each variable separately. The computation of the function $s(t)$ in integral form is included in Equation 4.4

$$\begin{aligned} \frac{1}{s}ds &= \frac{-1}{\tau_2}dt \\ \int \frac{1}{s}ds &= \int \frac{-1}{\tau_2}dt \\ \ln s &= \frac{-t}{\tau_2} \\ \ln s &= \frac{-t}{\tau_2} + C_1 \\ s &= e^{\frac{-t}{\tau_2}} + K_1 \quad , \end{aligned} \tag{4.4}$$

where C_1 denotes an arbitrary constant and $K_1 = e^{C_1}$. The initial condition provided in the original paper $s(0) = 1$ yields $K_1 = 0$. The value computed for K_1 is substituted back into the second equation in (4.4) to get the final integral form of $s(t)$ as shown in Equation 4.5.

$$s(t) = e^{\frac{-t}{\tau_2}} \tag{4.5}$$

As previously mentioned, the differential equation for $f(t)$ in (4.9) is bi-exponential and therefore matches the general form shown in Equation 4.6

$$f(t) = Ae^{\frac{-t}{\tau_1}} + Be^{\frac{-t}{\tau_2}} \quad , \tag{4.6}$$

where A and B denote two arbitrary constants. The first derivative of the generic bi-exponential form is computed in Equation 4.7.

$$\frac{df}{dt} = -\frac{A}{\tau_1}e^{\frac{-t}{\tau_1}} - \frac{B}{\tau_1}e^{\frac{-t}{\tau_2}} \tag{4.7}$$

Equations (4.5), (4.6) and (4.7) are substituted back in the initial differential equations system in (4.9), yielding:

$$\tau_1(-\frac{A}{\tau_1}e^{\frac{-t}{\tau_1}} - \frac{B}{\tau_1}e^{\frac{-t}{\tau_2}}) = ke^{\frac{-t}{\tau_2}} - (Ae^{\frac{-t}{\tau_1}} + Be^{\frac{-t}{\tau_2}}) \tag{4.8}$$

Equation 4.8 is then used to determine the values of the constants A and B in the generic bi-exponential equation by equating the coefficients on the left-hand side and right-hand side for both exponent terms as shown in Equation (4.9).

$$\begin{aligned} \text{for } e^{\frac{-t}{\tau_1}}: \quad & -A = -A \\ \text{for } e^{\frac{-t}{\tau_2}}: \quad & -\frac{\tau_1}{\tau_2} = k - B \\ & (1 - \frac{\tau_1}{\tau_2})B = k \\ & B = \frac{k\tau_2}{\tau_2 - \tau_1} \end{aligned} \tag{4.9}$$

To infer the value of the constant A, the initial condition $f(0) = 0$ is substituted into Equation (4.6) yielding:

$$\begin{aligned} f(0) =& Ae^0 + Be^0 \\ A =& -B = -\frac{k\tau_2}{\tau_2 - \tau_1} \end{aligned} \tag{4.10}$$

The integral form for $f(t)$ is then computed by substituting the values found for A and B back into the generic form of the synaptic current given in Equation (4.6), yielding 4.11.

$$\begin{aligned} f(t) &= -\frac{k\tau_2}{\tau_2 - \tau_1}e^{\frac{-t}{\tau_1}} + \frac{k\tau_2}{\tau_2 - \tau_1}e^{\frac{-t}{\tau_2}} \\ f(t) &= -\frac{k\tau_2}{\tau_2 - \tau_1}(e^{\frac{-t}{\tau_1}} - e^{\frac{-t}{\tau_2}}) \end{aligned} \tag{4.11}$$

The final step is to compute the constant k by matching the integral form of $f(t)$ computed in (4.11) with the equation for $f(t)$ given in the original model and reported in Equation (3.4). An exact match between the two function is obtained for

$$k = -\frac{1}{\tau_2} \tag{4.12}$$

The system of differential equations computed to map the single synaptic impulse $f(t)$ can be ultimately appended to the network's equations defined in Listing 1 as shown Listing 2.

```
eqs = '''
    dV/dt = (-gl*(V - Vl) + I_syn + Io) / C   : volt
    I_syn = (I_syn_bar / N) * f * tau         : amp / metre**2
    df/dt = (k * s - f) / tau1                : 1 / second
    ds/dt = -s / tau2                         : 1
'''
```

Listing 2: Definition of the synaptic current differential equations in BRIAN2.

The complete code for the model replica is shown in Appendix A.

4.2 Shared Random Time-stepping Implementation

As mentioned in Section 3.2.1, SRTS consists of a single random re-initialisation of the integration time-step that happens on each iteration of the simulation. This functionality is easily achievable in BRIAN2, as the simulator allows to dynamically override the internal variable `defaultclock.dt`, which regulates the simulation's dt directly from the Python high-level model. However, this trivial implementation conveys two main limitations. Firstly, BRIAN2 automatically performs several internal checks on the `defaultclock.dt` throughout the simulation execution, to ensure that the executed simulation time is always a multiple of the newly defined time-step. This rather strict constraint arises from the way in which BRIAN2 computes the time t has been already simulated, as explained in further detail in the BRIAN2 simulator documentation [16]. The time t is computed on each iteration as $dt \cdot ts$, where ts is the count of time-steps already executed:

this is also shown in the original update operation, which has been commented in Listing 7. Each time the `defaultclock.dt` variable is overridden throughout the simulation, BRIAN2 recomputes the amount of time-steps executed ts as an integer value. Hence, the new time-step magnitude must be a multiple of the simulation time t that has already passed. This constraint on the time-step magnitudes makes the usage of a completely randomised time-step method impossible without modifying the internal C++ code-base. Additionally, the internal dynamics of BRIAN2 discourage the usage of multiple successive simulations using the `net.run()`, as the program suffers from massive overheads due to multiple overrides of parts of the C++ project code performed.

For these reasons the SRTS method has been implemented by directly modifying the internal C++ code-base, to allow for a greater flexibility in changing the time-step dynamically and to ultimately achieve a complete randomised time-stepping method. Listings 6, 8 and 9 in Appendix D show the Python-based simulation loop, as well as the automatic adjustments introduced to the auto-generated project. The custom function `srts_simulation` defined in Listing 6 outputs an array of the synchronisation measure scores Σ computed for different values of the synaptic current coupling strength $\bar{I_{syn}}$. The process is fully automatised by using the Python library `subprocess` as well as Python's built-in file reading-writing features, in order to redefine the internal variable that stores the value of $\bar{I_{syn}}$ at the beginning of each run. The updated project code is later compiled through the `make all` command and ultimately executed using the compiled *main* executable file. The results of the latest simulation are stored in several binary files in the `results` folder. The paths of the binary files containing the neuron potentials monitored throughout the simulation and the time-steps used are automatically inferred and their content is loaded into `np.arrays` data structures. In order to remain consistent with the voltage sampling technique used in the original FTS model, it has been decided to select the time samples such that they are the largest values smaller than each one-millisecond step in the $[5, 10]s$ range.

Ultimately, the list of synchronisation measure scores for the sampled potentials for all values of $\bar{I_{syn}}$ are computed and returned. The automatic testing achieved through Python scripting works on a fixed C++ template project folder renamed to *SRTS* which is generated when launching the initial setup script *setup.py*. This setting allows for SRTS testing to be fully automated as the modified C++ code-base will not be overridden by the execution of other unrelated simulations, that will instead modify the content of the default *output* folder. Finally, Listing 9 shows the adjustments automatically performed by the setup script on the C++ main file in order to support SRTS. In FTS the simulation loop consists of a simple call to

the `magicnetwork.run` function. This must be replaced with an explicit simulation while-loop in order to dynamically modify the time-step magnitude in SRTS. Inside the while-loop the internal variables can be manually updated, using the custom function `fRand`, which is defined in Listings 9, to generate a new random dt value within the given range.

4.3 Subgroup-Specific Random Time-stepping Implementation

Subgroup-specific random time-stepping consists in dividing the network into smaller neuron groups, which are then integrated in time at different rates, i.e. using different time-steps. Conveniently, the built-in BRIAN2 class `NeuronGroup` accepts a dt argument that sets the integration time-step of all neurons within the group. The g subgroups created must be interlinked by all-to-all synapses across all the neurons inside the network in order to correctly replicate the original architecture. Therefore, each group must be initialised with g `Synapses` class objects, that will connect each neuron with all other neurons in their same group as well as all neurons in the other groups.

Hence the number of `Synapses` class objects generated for a generic SSRTS network is 2^g where g is the number of subgroups defined. Each neuron group must be monitored using `SpikeMonitor` and `StateMonitor` objects in order to record the membrane potentials and the spiking activity of the neurons within that specific subgroup. Ultimately, all the membrane potentials recorded must be vertically concatenated to reconstruct a single matrix that can be analysed by using the same synchronisation measure function used for the other methods, as further discussed in Section 5.1. The complete code for SSRTS simulations is provided in Listing 10 in Appendix E.

This simulation design, however, suffers from large overheads in the initial network construction due to the exponential increase in `Synapses` class objects as the number of neuron subgroups g grows larger: this ultimately results in unfeasible execution times for large values of g. Therefore, a neuron-specific time-stepping behaviour was not achievable when using a SSRT($g = 128$) architecture, as the computational cost of interconnecting 2^{128} `Synapses` class objects is prohibitively large.

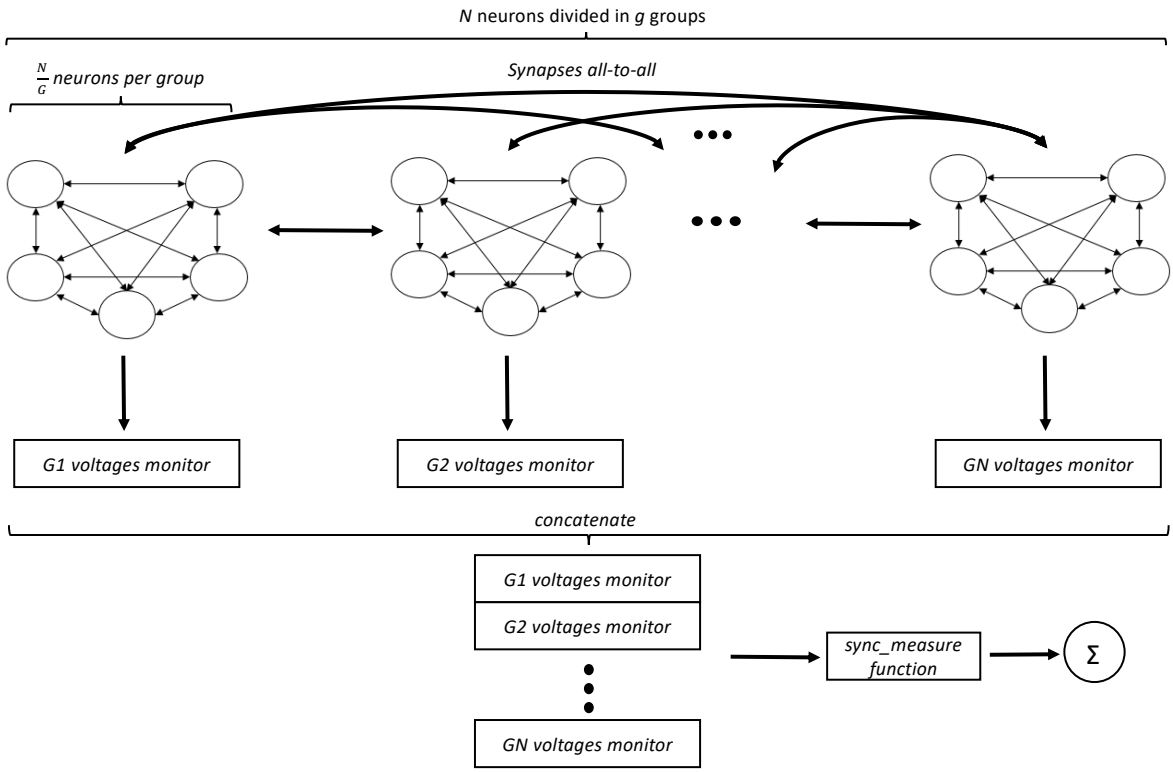

Figure 8: Implementation design for the neuron-specific random time-stepping method using multiple BRIAN2 `NeuronGroup` and `Synapses` objects.

4.4 Neuron-Specific Random Time-stepping Implementation

Contrary to the first two novel methods discussed earlier, neuron-specific random time-stepping is not easily embeddable in the BRIAN2 C++ project workflow, since the simulator tracks the integration time-step only on global levels, i.e. as a globally shared parameter and as an internal shared variable of `NeuronGroup` objects. The lack of any time-step variable definition on the individual neuron level determines the impossibility of a simple integration of NSRTS in BRIAN2 code-base that would not involve a significant refactoring of the auto-generated project. Therefore, the design choice of re-implementing the SNN simulation code from scratch in Python was made. Due to the architectural simplicity of the original model and the enormous optimisation capabilities of the `Numba` library in Python programming language, the simulation code was easily replicated and accelerated to a level where NSRTS results would be comparable with other methods. The complete simulation code for NSRTS is presented in Appendix E, with the dimensionless network parameters defined in Listing 12 and the main simulation function shown in Listing 11. The NSRTS simulation initialises and

tracks the three internal states of each neuron `v`, `f` and `s` in separate `np.array` data structures. Starting from the very first neuron of the network, each iteration of the simulation loop updates the state variables of one single neuron by generating a random time-step, resetting the voltage if a spike occurred and ultimately integrating the new states using the Euler method update formula

$$x(t + dt) = x(t) + dt \cdot x(t) \tag{4.13}$$

Consistently with other methods, the potentials are monitored and sampled with a frequency of $1ms$ in the range $[5, 10]s$ by selecting the discrete time instant closest to each sampling time. A list t of the cumulative times that have been already integrated for each neuron respectively is updated throughout the simulation and used at the end of each iteration to select the neuron i with the lowest $t[i]$ to integrate next. This allows the full integration process to be sequential, ensuring that synaptic impulses are not propagated back in time.

5 Testing

5.1 Synchronisation Measure

This section aims to discuss the methodologies and measures used to validate the models and evaluate the performance of the novel random time-stepping methods. First, the correctness of the model replica was tested against the results included in the original model by comparing the synchrony-aversion capabilities of the network observed for the two architectures. The plots included in Figure 1 and 2 in Section 3 of the original paper have been replicated in BRIAN2 by using the custom function `sync_measure`. The `sync_measure` function, which is included in Listing 5 in Appendix B, computes the measure of synchrony Σ as defined in the original paper in Section 2.3. The function leverages the parallelisation capabilities on vector operations of the Python `Numpy` library to optimise the computation. The input of the function `V` is the matrix of neuron membrane potentials recorded each millisecond in the $[5, 10]s$ range of the simulation. The measure of synchrony Σ is based on the analysis of the temporal fluctuations of network activity, as outlined in (Hansel, Mato et al. [4]) and taking inspiration from (Hansel and Sompolinsky [13]), (Golomb and Rinzel [14]), and (Ginzburg and Sompolinsky [15]).

In order to compute it, the average membrane potential must be first evaluated at time t as

$$A_N(t) = \frac{1}{N} \sum_{i=1}^{N} V_i(t) \tag{5.1}$$

Then, its time fluctuations are estimated by computing the variance Δ_N of the average membrane potentials:

$$\Delta_N = \langle A_N(t)^2 \rangle_t - \langle A_N(t) \rangle_t^2 \tag{5.2}$$

Since the single cell activity Δ must be taken into consideration, the variance in the equation above is normalised over the neuron population as shown below.

$$\Delta = \frac{1}{N} \sum_{i=1}^{N} \left(\langle V_i(t)^2 \rangle_t - \langle V_i(t) \rangle_t^2 \right) \tag{5.3}$$

Finally, the coherence measure Σ is defined as the ratio of Equation (5.2) over (5.3), i.e. the variance over the population-averaged variance:

$$\Sigma_N = \frac{\Delta_N}{\Delta} \tag{5.4}$$

The size of all networks defined in this experiment is fixed to $N = 128$, therefore the simplified identifier Σ in place of Σ_N will be further used to denote the synchronisation measure of the networks presented. This notation is also consistent with the one used in the original paper.

5.2 Testing the Model Replica

The synchronisation measure described in Section 5.1 is used to analyse the changes in the network performance for different values of the synaptic current coupling strength I_{Syn}^{-}. The results of this analysis as presented in the original method are included in the left plot in Figure 9 and visually compared with the ones obtained for the BRIAN2 model replica shown on the right.

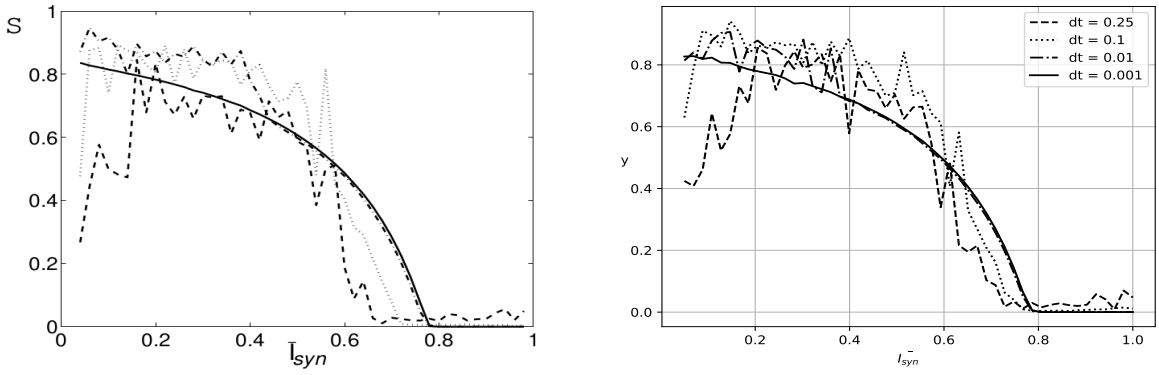

Figure 9: Original plot of the synchronisation measure Σ over I_{syn}^{-} for various time-steps dt (left) and results replicated in Brian2 (right).

Each curve denotes a set of simulations performed with a different integration time-step. For the sake of

clarity, the curves formatting was kept consistent with the one used in the original pictures. Equivalent curves are observed to follow similar trends, which suggests that the network architecture has been correctly replicated and confirms the argument supported in the original model that the largest time-step able to suppress most synchronisation activity is $dt = 0.001ms$. Small differences in the two visualisations are expected, as the original paper does not specify the sampling frequency used for the synaptic current coupling strength I_{syn}^{-} nor its lower bound. It has been decided to use fifty linearly spaced values of I_{syn}^{-} in the range $[0.05, 1]$. The sampling frequency of fifty allows to capture a detailed study of the network synchronisation, while still resulting in a reasonable execution time. The lowest boundary of I_{syn}^{-} has been set to 0.05 upon visual inspection of the $dt = 0.001$ curve, that for lower synaptic current coupling strengths deviates from the expected value. For the purpose of evaluating the performance of the new random time-stepping methods, the information on synchronisation activity observed in Figure 9 must be compressed and quantified by a single score, that represents the robustness of the model against synchronisation artifacts. In order to do so, the curve $dt = 0.001$ obtained by simulating the model replica with a fixed time-stepping method was considered as ground truth. Then, the distance of other simulations from the ground truth was quantified using the Mean Squared Error of the synchronisation measures, as discussed in more detail in Section 6.1.

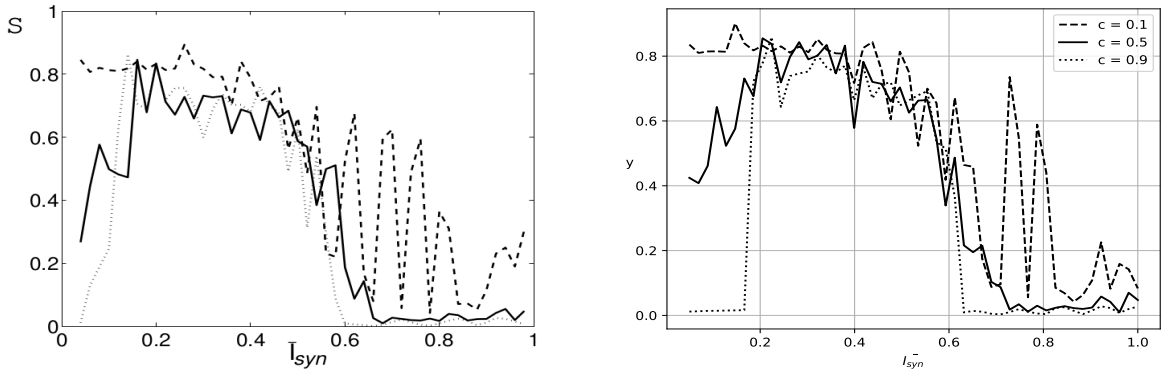

Figure 10: Original plot of the synchronisation measure Σ over I_{syn}^{-} for various initial degrees of synchrony c (left) and results replicated in Brian2 (right).

The original paper also presents a study on the different synchronisation measures obtained when equivalent models are initialised with different initial degrees of synchrony c. The accurate replication of these results gives a further intuition on the correctness of the model architecture. As mentioned in Section 3.1, the parameter c takes a value in the range $[0, 1]$, where the maximum value results in a uniform distribution of the initial neuron voltages in the range $[-60, -40]mV$, while the minimum denotes the scenario in which

all neurons share the initial membrane voltage $V_i = V_{rest}$. Figure 10 shows on the right the plot of the synchronisation measures recorded for the three different values of c, compared to the original model results shown on the left. Consistently with the results obtained in the first test, the two visualisations appear very similar, hinting at the correctness of the model replica. This further analysis also confirms the argument supported in the original paper that the optimal initial potential synchronisation is $c = 0.5$. Therefore, this initial setting of $c = 0.5$ has been used in all additional tests carried on the randomised time-stepping methods and presented in Section 6.

5.3 Testing the Randomised Time-stepping Methods

The correctness of all novel randomised time-stepping methods implemented in this project ought to be tested in comparison with the fixed time-stepping method. This is simply achieved by forcing the random range radius dt_{rad} in randomised time-stepping methods to zero. Since all randomised methods compute the next integration time-step dt as a random value with a certain range, which is specified by a middle value (dt_{fixed}) and a radius (dt_{rad}) as

$$dt_{random} = [dt_{fixed} \cdot (1 - dt_{rad}),\ \ dt_{fixed} \cdot (1 + dt_{rad})] \tag{5.5}$$

initialising random time-stepping models with a random range radius of zero results in the randomised method behaving in the same exact way as fixed time-stepping.

The results obtained by testing all the newly proposed methods using the aforementioned initial settings, resulted in plots equivalent to the one observed in Figure 9. Moreover, the extremely accurate results discussed in Section 6 for most randomised time-stepping method are a further indicator that the additional logic embedded in the model to implement the novel methods is flawless.

6 Evaluation and Results

6.1 Mean Squared Error Measure

This section aims to discuss the additional evaluation methods used alongside the original synchronisation measure defined in Section 5.1. In order to compare different time-stepping methods, the information on the network synchronisation activity provided as a `np.array` data structure output by the `sync_measure` function is condensed in a single Mean Squared Error (MSE) score. The MSE represents how distant the results of a specific method m are from the ground truth gt, by computing the average squared Euclidean distance of the synchronisation measures recorded over a range of synaptic current coupling strengths as specified in Equation (6.1).

$$mse(m) = \sum_{i}^{\bar{I_{syn}}} (gt[i] - m[i])^2 \tag{6.1}$$

The Python implementation of a custom mean squared error function is shown in Section B Listing 4. The results presented in this section often feature a visualisation of the synchronisation measure evolution over the synaptic current coupling strengths on the left and the consequent MSE scores displayed in a scatter plot on the right. This type of visualisation will be further referred to as "synchronisation analysis".

It should also be mentioned that the full extent of results presented in this section have been achieved on a 8 GiB RAM laptop mounting a Intel Core i5-8250U CPU at 1.60GHZ x 8. Furthermore, the execution time recorded for simulations using BRIAN2 high-level Python modeling are computed as the cumulative run-time of the network objects, in order to discard initialisation overheads.

6.2 Fixed Time-stepping Evaluation

The model replica performance has been evaluated on a set of several time-step magnitudes to allow for a comprehensive comparison in the synchronisation activity observed in FTS against the new randomised methods. The set of integration time-step chosen is $dt = [1, 0.5, 0.25, 0.1, 0.05, 0.025, 0.01]\ ms$. The left plot

included in the synchronisation analysis will display only the subset $dt = [0.25, 0.1, 0.025, 0.01]$, whereas the full information is summarised in the adjacent MSE plot.

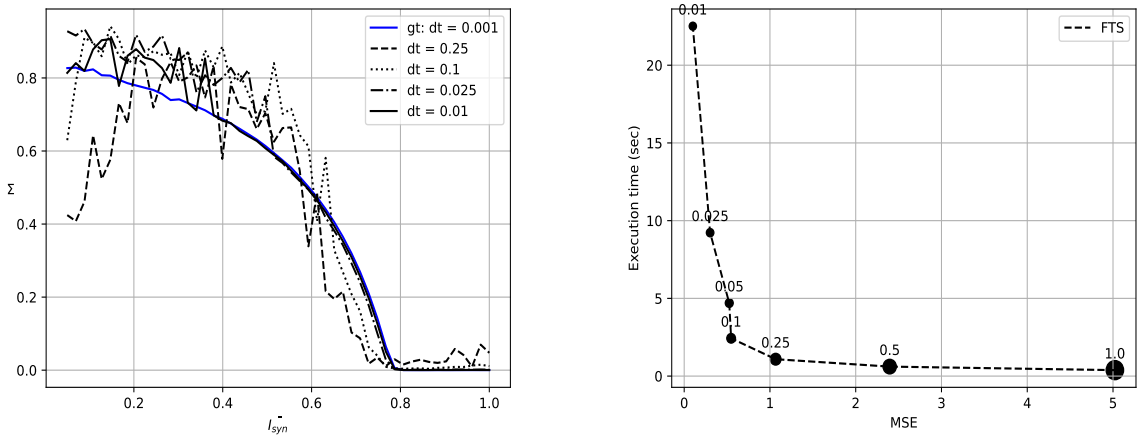

Figure 11: Synchronisation measure for FTS simulations with different time-steps dt (left) and respective MSE scores (right).

Figure 11 shows the synchronisation analysis for the FTS method. Each marker on the MSE plot corresponds to a simulation performed with a certain fixed time-step dt that is both specified next to the marker and hinted by the size of the marker used, which is directly proportional to the dt. As expected, the model steadily improves as the time-step magnitude decreases. Moreover, the execution time grows linearly with respect to the decrease in dt. The inverse proportion relationship among the two can be appreciated by looking at two consecutive markers, i.e. as the time-step magnitude between two consecutive markers is halved, the execution time doubles.

6.3 Shared Random Time-stepping Evaluation

6.3.1 All-to-all Coupling

Compared to FTS, shared random time-stepping did not result in a significant performance improvement, as shown in the synchronisation analysis in Figure 13.

Although for larger time-steps SRTS(0.5), i.e. the shared random time-stepping model with a time-step range radius of $dt_{rad} = 0.5$, significantly outperforms the fixed counterpart, for smaller time-steps it converges more

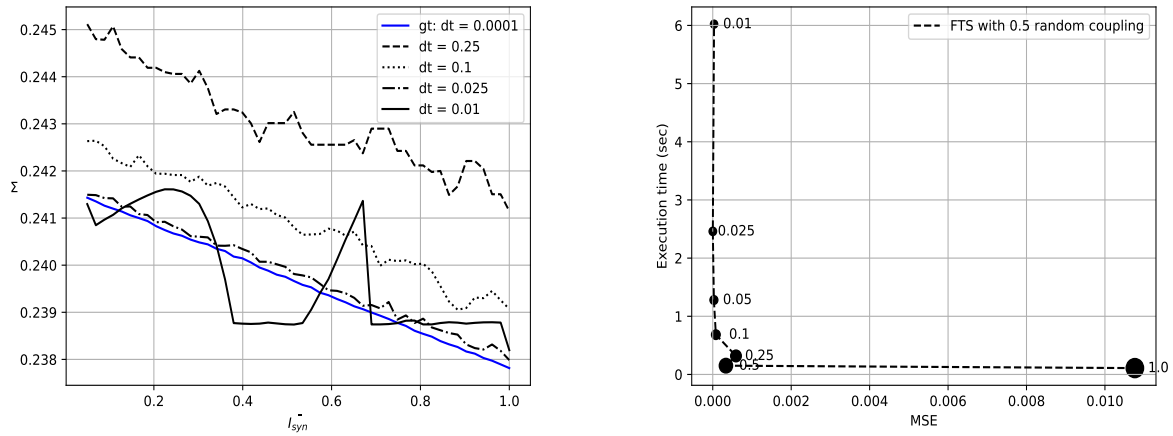

Figure 12: Synchronisation measure for FTS with random 0.5 coupling simulations with different time-steps *dt* (left) and respective MSE scores (right).

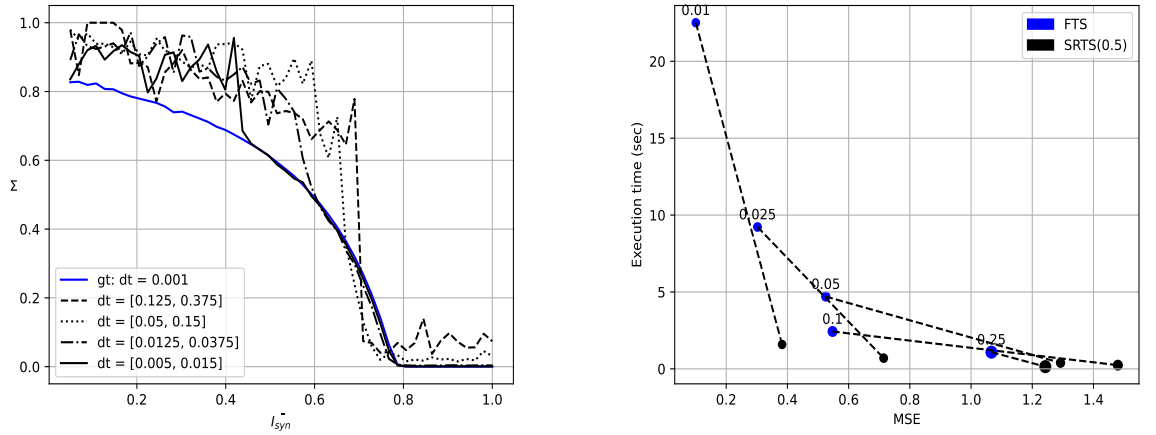

Figure 13: Synchronisation measure for SRTS simulations with different time-steps *dt* (left) and its MSE performance comparison with FTS (right).

slowly towards the ground truth.

One of the plausible reasons for this behaviour has been identified in the high degree of homogeneity of the original model, i.e all neurons share the same synaptic current equations and constants and are coupled all-to-all. This highly homogeneous architecture is not only very susceptible to synchronisation artifacts, but also makes them irreversible for methods such as SRTS in which there is no difference in the individual neurons integration time-steps. Once two neurons synchronise their potentials they cannot diverge again, since the two units receive the same synaptic inputs from all other spiking events happening in the network

and their state variables are updated synchronously. Therefore, sporadic integrations with larger time-steps in SRTS are likely acting as a catalyst for synchronisation events that occur slower in fixed time-stepping simulations. In order to confirm this hypothesis a thorough hyper-parameter search has been performed on the network parameter dt_{rad} for a SRTS model with $dt_{fixed} = 0.01$. Figure 14 features the synchronisation analysis obtained for the following set of dt_{rad} magnitudes: $dt_{rad} = [0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5]$.

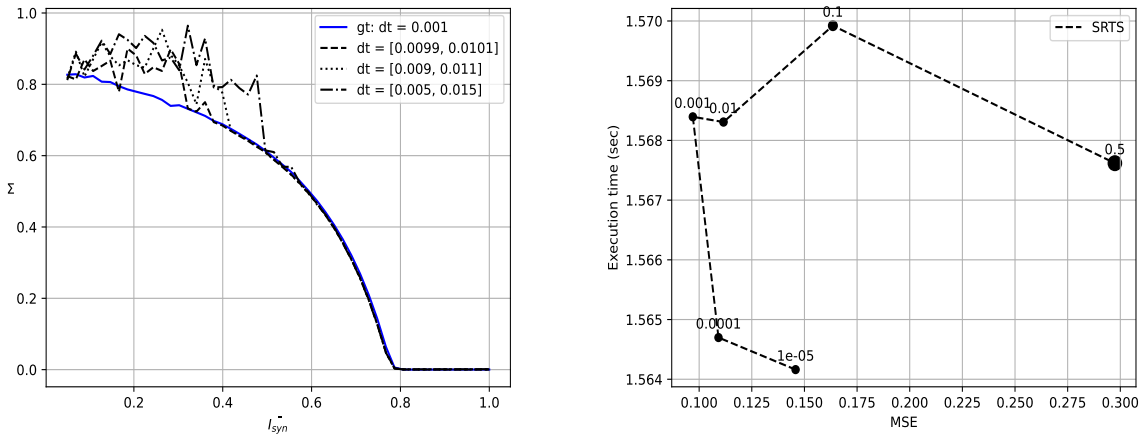

Figure 14: Synchronisation measure for SRTS simulations with different random time-step ranges dt_{rad} (left) and respective MSE scores (right).

As theorised, networks initialised with wider ranges suffer from a significantly higher synchronisation activity. Moreover, simulations initialised with extremely small ranges seem to perform similarly to FTS on average, with optimal results obtained for values close to $dt_{rad} = 0.001$. The optimal model SRTS(0.001) achieves an MSE score 25% lower than the one recorded for FTS. In this setting, the disadvantages observed for sporadically large dt values are limited as the random range is extremely tight, so the beneficial effects brought by randomness result in an overall improvement in performance.

6.3.2 Random Coupling

In order to mitigate the network homogeneity the network architecture was partially redesigned to account for more diversity in the synaptic impulse patterns. Namely, the all-to-all coupling amongst neurons was modified to random coupling with probability $p = 0.5$. Implementing this architectural change on BRIAN2 C++ auto-generated project is as easy as changing the coupling condition initialisation, as shown in Appendix D in Listing 7. However, such a change in the core architecture of the model implies completely different

synchronisation activity for all time-stepping methods. Therefore, a new ground truth was computed using the synchronisation analysis results for FTS simulations with random coupling, which are shown in Figure 12. It was found that for the random coupling experiment the ground truth had to be computed using a time-step of one degree of magnitude smaller than the dt used in all-to all coupling in order to achieve a smooth synchronisation curve.

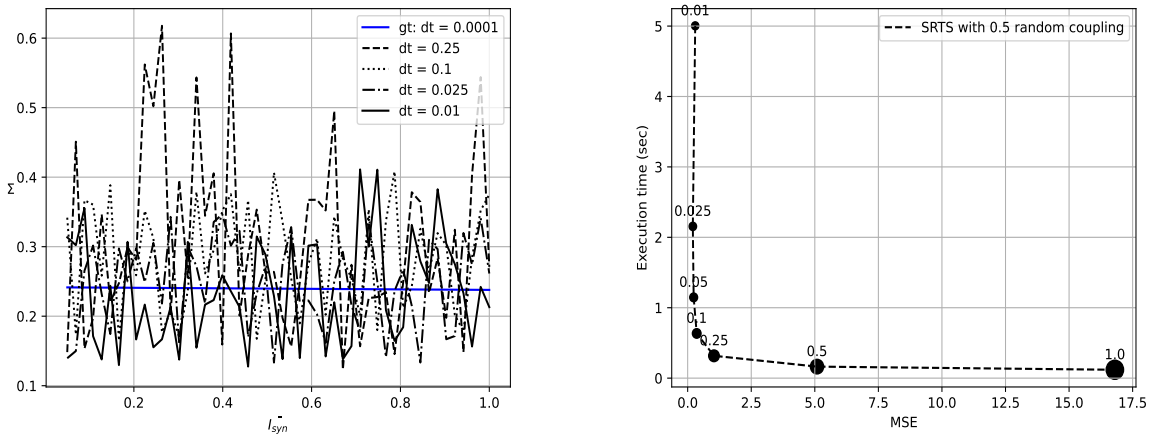

Figure 15: Synchronisation measure for SRTS with random 0.5 coupling simulations with different time-steps dt (left) and respective MSE scores (right).

From the results displayed in Figure 12 it is clear that lowering the degree of homogeneity in the network has a beneficial effect on synchrony-aversion. The synchronisation measures observed in the left plot are extremely close to the ground truth, which implies remarkably low MSE scores. Inversely, the synchronisation analysis for SRTS(0.5) with random coupling included in Figure 15 shows a higher and rather oscillatory neuron synchronisation activity. This ultimately means that the higher inhomogeneity of a random coupled network does not balance the disadvantages of using a large radius.

6.4 Subgroup-Specific Random Time-stepping Evaluation

Subgroup-specific random time-stepping as introduced in Section 3.2.2, consists in dividing the network in smaller subgroups of neurons which share the same randomised time-step. In SSRTS models, neurons within the same subgroup g are integrated synchronously using a fixed time-step randomly initialised during the network initialisation. Therefore, their synchronisation activity with one another should not differ with respect to FTS. Inversely, the synchronisation activity among neurons in different subgroups should reduce

due to the asynchronicity of neurons' state updates. Figure 21 shows the synchronisation analysis for a SSRTS(2, 0.5) model, i.e. a subgroup specific random time-stepping model with two subgroups g and time-step range radius $dt_{rad} = 0.5$.

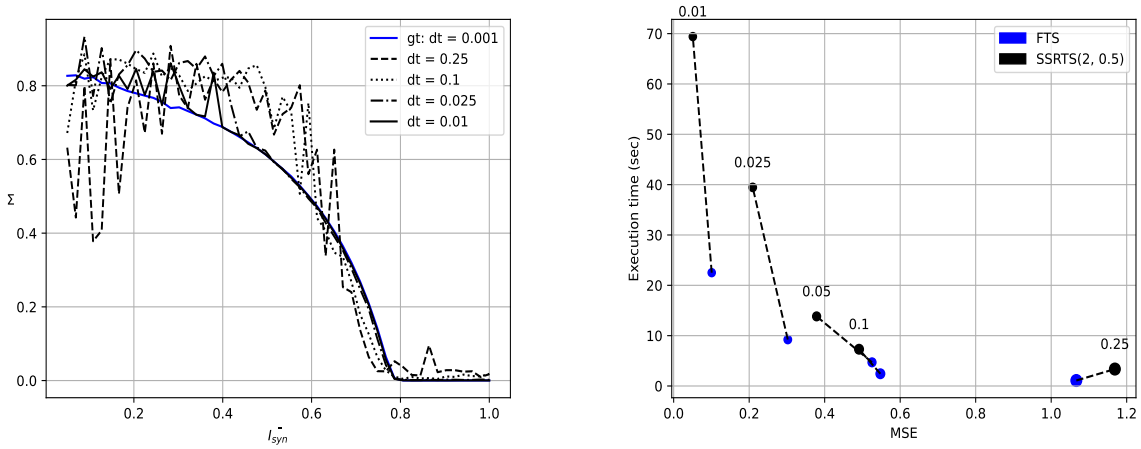

Figure 16: Synchronisation measure for SSRTS(2, 0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right).

It should be mentioned that for SSRTS the hyperparameter search on the optimal random time-step range did not produce the same results shown in Section 6.3. Figure 18 shows the results obtained by simulating SSRTS(2, x) models with x as a random range radius in the following set: $dt_{rad} = [0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5]$ and with $dt_{fixed} = 0.01$. As opposed to SRTS, the MSE plot shows that wider random ranges do not severely impact the performance of SSRTS in a negative way. The higher robustness of SSRTS against wide time-step ranges is likely due to the fact that they introduce larger discrepancies in the integration process inter-subgroups, which has a positive effect on performance that partially balances the higher likelihood of synchronisation activity intra-subgroup. Similarly to what observed in Figure 14 for SRTS, the performance of the model increases as the random range shrinks, hitting an optimal radius magnitude that minimises the overall MSE score of the method.

The optimal value for dt_{rad} in SSRTS methods lies within the $[0.005, 0.01]$ range, i.e. approximately one degree of magnitude higher than the one computed in Section 6.3 for SRTS. In this optimal region the MSE score recorded is approximately $mse(SSRTS) = 0.037$, which accounts for a 270% improvement on the performance of the FTS method using the same time-stepping settings.

However, it must be noted that the process of dividing the network into subgroups through BRIAN2 simulator

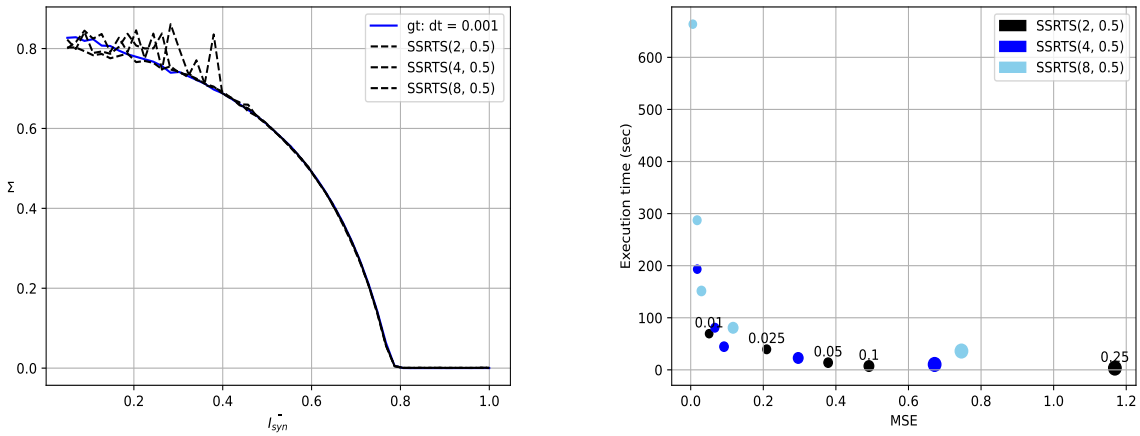

Figure 17: Synchronisation measure for SSRTS(x, 0.5) simulations with time-step $dt_{fixed} = 0.01$ (left) and MSE scores for SSRTS(x, 0.5) simulations with various time-steps (right).

is extremely computationally expensive as the number of `Synapses` class objects grows exponentially with respect to the number of subgroups g. Hence, the execution time of SSRTS(2, x) recorded in this section is approximately three times larger than what observed in equivalent FTS simulations. The exponential increase in execution time as the number of subgroups g grows larger can be appreciated in Figure 17. The plot compares the performance of SSRTS models initialised with an increasing number of subgroups both in terms of MSE and execution time. Further synchronisation analysis for specific SSRTS methods are also provided in Appendix F.

It should be mentioned that this efficiency bottleneck is largely due to an internal overhead in the BRIAN2 auto-generated C++ code-base, therefore building a SSRTS simulation loop from scratch equivalently to what it was done for NNSRTS should result in execution times similar to the ones observed in fixed time-stepping. Therefore, the performance improvement obtained for SSRTS is not to be discarded, since the exponential increase in execution time is a mere artifact of the implementation methodology used.

6.5 Neuron-Specific Random Time-stepping Results

The highest degree of randomisation was introduced in the model by using neuron-specific random time-stepping. This novel approach outperformed all other time-stepping methodologies, showing significantly smoother and faster convergence patterns towards the ground truth synchronisation results. Figure 19

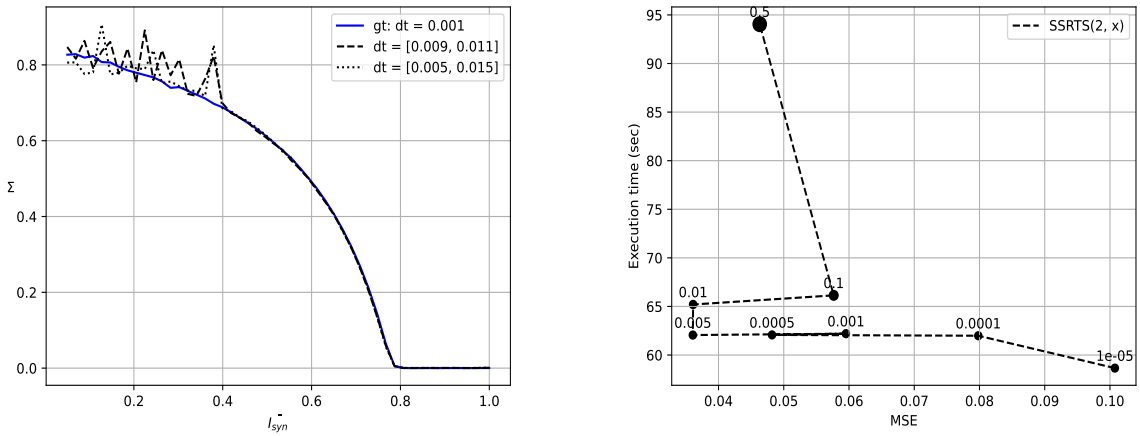

Figure 18: Synchronisation measure for SSRTS(2, x) simulations with different random time-step ranges dt_{rad} (left) and respective MSE scores (right).

shows the synchronisation analysis for a NSRTS(0.5) model. If compared with other methods, NSRTS shows a significantly more robust architecture against oscillatory synchronisation activity. The non-oscillatory pattern observed in NSRTS synchronisation analysis interestingly results in worse MSE evaluations when compared to FTS for larger time-steps. This may be caused by the fact that highly oscillatory synchronisation activity in the network has a beneficial effect on the MSE evaluator, since the evaluation will register sporadic accurate results closely to points where the synchronisation measure overlaps with the ground truth curve. Therefore, despite the NSRTS actual synchronisation activity resembles more closely the ground truth, it is evaluated as less accurate according to the MSE indicator. Under this assumption, the usage of a more robust evaluator against noisy signals, such as the median of the squared deviations, would likely represent more truthfully the actual performance of the model.

Inversely, the mean squared error score for lower time-steps is significantly better for the neuron-specific time-stepping method. For the smallest value of dt tested ($dt = 0.01$), which is an order of magnitude larger than the one used as ground truth, the MSE recorded for NSRTS is $mse(SRTS(0.5)) = 0.01164$, which compared to the $mse(FTS) = 0.1008$ registered for FTS yields a 860% improvement in accuracy. Ultimately, NSRTS scored significantly better than the other alternative methods discussed in this project retaining a similar execution time. These results imply that the execution time of the network can be massively optimised by utilising this novel method as opposed to FTS. In the original model scenario, for a target Mean Squared Error tolerance of approximately 0.1, the model would take approximately 23 seconds

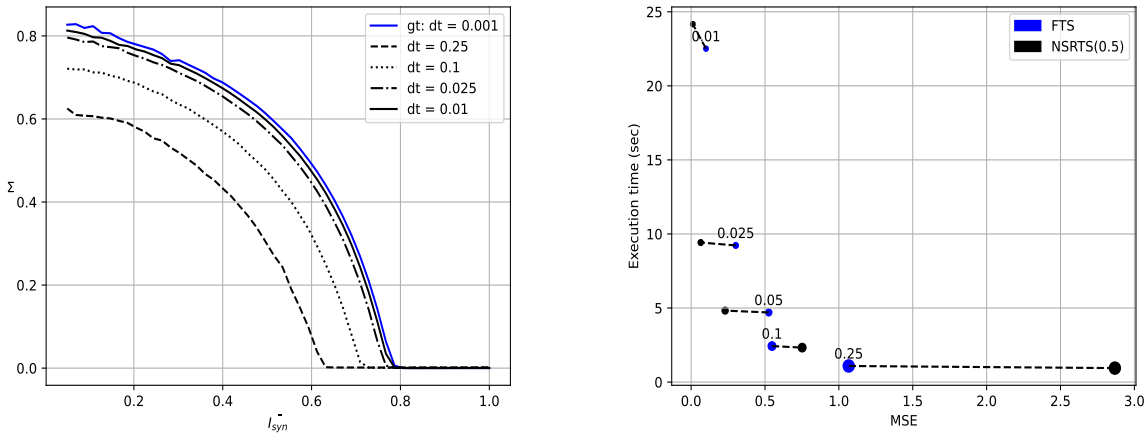

Figure 19: Synchronisation measure for NSRTS(0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right).

to execute one simulation using FTS, which is 2.56 times larger than the 9 seconds it would take to simulate an equivalently accurate NSRTS model. This is because the former method needs to integrate using a time-step no greater than $dt = 0.01$, while the latter achieves the same results using a time-step as large as $dt = 0.025$.

7 Conclusion

Building on the assumption of the beneficial effect of randomness on the synchrony-aversion capabilities of spiking neural networks, the analysis proposed in this project offers an overview of the effectiveness of three innovative randomised time-stepping integration method. These methods were found to be significantly more efficient if compared to the generic fixed time-stepping method originally implemented by (Hansel, Mato et al. [4]) . In particular, introducing some degree of randomness within the neuron integration process was proven to achieve the highest performance returns, as observed for NSRTS and, on a smaller scale, for SSRTS.

The outstanding results achieved by both the aforementioned methods pave the way for the analysis of several additional randomised time-stepping methods for further work. As an example, a hybrid method that consists of a neuron-specific randomised approach which does not recompute the time-step for each iteration is a less computationally expensive version of NSRTS that could potentially retain similar performance improvements. Similarly, a subgroup-specific approach that recomputes new time-steps at each iteration could be a matter of further evaluation.

Regarding the BRIAN2 Simulator software, a natural extension of the project would be to integrate all novel methods assessed in this project as features of BRIAN2 simulations. Finally, the considerable speed-up observed for the new randomised time-step methods should be documented with further testing, as to understand to what extent the homogeneity characteristics of the model impact the results observed in this project. Ultimately, the results discussed in this report are expected to be summarised in a scientific publication.

8 User guide

The complete code-base for this project is available open-source at the following Github link:

https://github.com/Fabio752/Randomised-time-stepping-methods-for-SNN-simulations

The project can be easily imported by using cloning the github repository from the terminal/command prompt through the `git clone` command as follows:

```
git clone <repository_path>
```

Once cloned locally, the project must be set up using a few simple commands:

1. If Python 3.x is not yet downloaded on your local machine, download it from
   https://www.python.org/downloads/.
2. If the Python version installed does not include `pip`, install it following the documentation at:
   https://pip.pypa.io/en/stable/installing/.
3. From the root repository of the project, execute the following command to install all the Python libraries needed.
   `pip install -r requirements.txt`.
4. Execute the *setup.py* script by running the following command to automatically build the folder output folder structure as well as generating a fixed C++ project in the *SRTS* folder, which is used to perform SRTS simulations.
   `python setup.py`

The additional scripts provided in the root repository perform the various simulations discussed in this report and save the results in unique binary files in the *bin* folder. Some simulations take very long time to execute, so in the interest of usability the *bin* folder has been uploaded with the whole set of results needed to reproduce all visualisations included in this report. In this regard, the two scripts *plot_analysis* and *plot_single_current*, also included in the root repository, can be used to reproduce all plots, which will be saved in the appropriate subfolders inside the *img* folder.

References

[1] Filip Ponulak, Andrzej Kasiński (2011) Introduction to spiking neural networks: Information processing, learning and applications.

[2] Nikola Kasabov, FIEEE, FRSNZ (2012) Evolving spiking neural networks and neurogenetic systems for spatio-and spectro-temporal data modelling and pattern recognition.

[3] Daniel Goodman, Romain Brette (2008) Brian: a simulator for spiking neural networks in Python

[4] D.Hansel, G.Mato, et al. (1998) On numerical simulations of integrate-and-fire neural networks

[5] Micheal J. Shelley, Louis Tao (2001) Efficient and Accurate Time-Stepping Schemes for Integrate-and-Fire Neuronal Networks

[6] William Lytton, Micheal Hines (2005) Independent variable timestep integration of individual neurons for network simulations

[7] Romain Brette et al. (2007) Simulation of networks of spiking neurons: A review of tools and strategies

[8] Michiel D'Haene, Benjamin Schrauwen et al. (2009) Accelerating Event-Driven Simulation of Spiking Neurons with Multiple Synaptic Time Constants

[9] Maurizio Mattia, Paolo Del Giudice (2000) Efficient Event-Driven Simulation of Large Networks of Spiking Neurons and Dynamical Synapses

[10] Jan Reutimann, Michele Giugliano, Stefano Fusi (2003) Event-Driven Simulation of Spiking Neurons with Stochastic Dynamics

[11] M. L. Hines, N. T. Carnevale (2001) NEURON: A Tool for Neuroscientists

[12] S.D. Cohen, A.C. Hindmarsh, P.F.Dubois (2000) CVODE, A Stiff/Nonstiff ODE Solver in C

[13] Hansel, D., Sompolinsky, H. (1992) Synchronization and computation in a chaotic neural network.

[14] Golomb, D., Rinzel, J. (1993). Dynamics of globally coupled inhibitory neurons with heterogeneity.

[15] Ginzburg, I., Sompolinsky, H. (1994). Theory of correlations in stochastic neu- ral networks.

[16] BRIAN Authors (2020) https://brian2.readthedocs.io/en/stable/user/running.html#changing-the-simulation-time-step

9 Appendix

A Python model

```
# Parameters.
neurons   =     128
gl        =     0.1   *    mS / cm**2
Vl        =     -60   *       mV
C         =       1   *  ufarad / cm**2
tau       =      10   *       ms
theta     =     -40   *       mV
tau1      =       3   *       ms
tau2      =       1   *       ms
I0        =     2.3   *     uA / cm**2
T = -tau * log(1 - gl/I0 * (theta - Vl))
k = 1 / tau2
rt = 10000
duration = rt / 1000
# Equations.
eqs = '''
dV/dt = (-gl * (V - Vl) + I_syn + I0) / C : volt
I_syn = (I_syn_bar / N) * f * tau         : amp / metre**2
df/dt = (k * s - f) / tau1                : 1 / second
ds/dt = -s / tau2                         : 1
'''
```

Listing 3: Original model's parameters definition in Brian2 code (*model.py*).

B Evaluators of synchrony

```python
import numpy as np


def mse(coherences, ground_truth):
    return np.sum(np.square(coherences - ground_truth))


def sync_measure(V):
    # Initialise lists.
    v_i_squared = np.square(V)
    avg_potentials = np.mean(V, axis = 0)

    # Compute delta_n
    squared_potentials = np.square(avg_potentials)
    delta_n = np.mean(squared_potentials) - np.square(np.mean(avg_potentials))

    # Compute delta
    mean_v_i_squared = np.square(np.mean(V, axis = 1))
    delta = np.mean(np.mean(v_i_squared, axis = 1) - mean_v_i_squared)

    # Compute sigma_n
    sigma_n = delta_n / delta
    return sigma_n
```

Listing 4: Synchronisation measure and MSE custom functions (*measures.py*).

C Fixed random time-stepping implementation

```
def fixed_simulation(dt, rt, I_syn_bar_magnitude, method = 'euler', c = 0.5,
    profile = True):
    I_syn_bar = I_syn_bar_magnitude * uA / cm**2
    # Initialise fixed dt and runtime.
    defaultclock.dt = dt * ms
    runtime = rt * ms
    # Initialise neuron group.
    G = NeuronGroup(N, eqs, threshold='V > theta', reset='V = Vl',
                    method = method, name = 'G')
    G.V = ''' Vl + I0 / gl * (1 - exp(-c * (i * T) / (N * tau))) '''
    G.s = 1
    # Initialise synapses and connect them all-to-all.
    S = Synapses(G, G, on_pre= "s += 1", method = method, name = 'S')
    S.connect(condition = 'i != j')
    # Initialise spike and state monitor.
    sp_m = b2.SpikeMonitor(G, name = 'sp_m')
    st_m = b2.StateMonitor(G,["V"], record = True, dt = 1 * ms, name = 'st_m')
    # Build and run network.
    net = b2.Network(G)
    net.add(S)
    net.add(sp_m)
    net.add(st_m)
    net.run(runtime, profile = profile)
```

Listing 5: Fixed time-stepping simulation code in Brian2 (*simulations.py*).

D Shared random time-stepping implementation

```
def srts_simulation(dt_fixed, dt_rad = 0.5, up_rad = None, low_rad = None,
    prj_folder = "SRTS", t_bin = "auto", v_bin = "auto"):
    if up_rad is None:
        up_rad = dt_rad
    if low_rad is None:
        low_rad = dt_rad
    dt = dt_fixed * 10**-3
    upp_lim = dt * (1 + up_rad)
    low_lim = dt * (1 - low_rad)
    rand_dt = "\t\t\tdt = fRand("+ str(low_lim) + ", " + str(upp_lim) + ");\n"
    main = './SRTS/main.cpp'
    utils.write_file(main, 88, rand_dt)
    # use same I_syn_bars of fixeddt simulation for consistency.
    I_syn_bars = np.fromfile("bin/fixeddt_I_syn_bars", dtype=np.float64) *
        10**-2
    coherences = []
    exec_time = 0
    # Find useful files with glob.
    for file in glob.glob('./SRTS/code_objects/G_stateupdater_*.cpp'):
        state_updater = file
    if t_bin == "auto":
        for file in glob.glob('./SRTS/results/_dynamic_array_st_m_t_*'):
            t_bin = file
    if v_bin == "auto":
        for file in glob.glob('./SRTS/results/_dynamic_array_st_m_V_*'):
            v_bin = file
    for I_syn_bar in tqdm.tqdm(I_syn_bars):
        new_I_syn_bar = "\tconst double _lio_2 = 1.0f*(" + str(I_syn_bar) + "
            * 0.01)/128; \n"
        utils.write_file(state_updater, 98, new_I_syn_bar)
```

```
        # Compile (make all) and run (./main) simulation
        cwd = os.path.abspath(prj_folder)
        process = subprocess.Popen(["make", "all"], stdout=subprocess.PIPE,
            cwd = cwd)
        process.wait()
        start_time = time.time()
        process = subprocess.Popen(["./main"], stdout=subprocess.PIPE, cwd =
            cwd)
        process.wait()
        exec_time += time.time() - start_time
        Ts = np.fromfile(t_bin, dtype=np.float64)
        Vs = np.reshape(np.fromfile(v_bin,  dtype=np.float64), (N, -1), order
            = 'F')
        # Find indexes closest to sampling points of 1ms multiples
        idxs = []
        instant_to_record = 5.0
        while instant_to_record < (duration - 0.001):
            exp_loc = int(Ts.shape[0] * instant_to_record / duration)
            idx = utils.find_idx(Ts, exp_loc, instant_to_record)
            idxs.append(idx)
            instant_to_record += 0.001
        # filter voltages based on sampling indexes, compute and append
            coherence.
        Vs = Vs[:, idxs]
        sigma = measures.sync_measure(Vs)
        coherences.append(sigma)
    return coherences, exec_time / len(I_syn_bars)
```

Listing 6: SRTS simulations using Python to modify and run C++ auto–generated code (*simulations.py*).

```
...
// const char _cond = (i != _k)
   const char _cond = (i != _k) && (((double) std::rand() / (RAND_MAX)) <
      0.5
...
```

Listing 7: Modification made to the auto-generated C++ file *SRTS/code_objects/S_synapses_create_generator_codeobject.cpp*.

```
...
inline void tick(){
    timestep[0] += 1;
    // t[0] = timestep[0] * dt[0];
    t[0] += dt[0];
    }
...
```

Listing 8: Modification made to the *SRTS/brianlib/clocks.h* auto-generated C++ header file.

```
#include <random>
double fRand(double fMin, double fMax)
{
    double f = (double)std::rand() / RAND_MAX;
    return fMin + f * (fMax - fMin);
}

int main(int argc, char **argv)
{
    ...
    // magicnetwork.run(10.0, NULL, 10.0);
    double duration = 10.0;
    double t = 0.0;
    double dt;

    srand (time(NULL));
    while (t < duration){
        dt = fRand(9.5e-06, 1e-05);
        _array_defaultclock_dt[0] = dt;
        _array_defaultclock_t[0] = t;
        magicnetwork.run(dt, NULL, 10.0);
        t += dt;
    }
    ...
}
```

Listing 9: Modification made to the *SRTS/main.cpp* auto-generated C++ script to allow variable random dts.

E Neuron-Specific Random Time Stepping Implementation

```
def nsrts_syngroup_simulation(rt, I_syn_bar_magnitude, method = 'euler', c =
    0.5, dt = 0.25, rad = 0.5, g = 2):
    # SUPPRESS WARNINGS
    BrianLogger.suppress_hierarchy('brian2.synapses.synapses.
        synapses_dt_mismatch')
    start_scope()
    device.reinit()
    device.activate()
    dt = dt * ms
    I_syn_bar = I_syn_bar_magnitude * uA / cm**2
    net = b2.Network()
    group_size = neurons / g

    for p in range(g):
        rand_dt = rd.uniform(dt * (1 - rad), dt * (1 + rad))
        G = NeuronGroup(group_size, eqs, threshold='V > theta', reset='V = Vl'
            , method = method, name = 'G' + str(p+1), dt = rand_dt)

        G.V = '''Vl + I0 / gl * (1 - exp(-c * (p * group_size + i) * T / (g *
            N * tau)))'''
        G.s = 1
        net.add(G)

    idx = 1
    for p in range(g):
        for q in range(g):
            ng1 = net.__getitem__('G' + str(p + 1))
            ng2 = net.__getitem__('G' + str(q + 1))
            S = Synapses(ng1, ng2, on_pre= "s += 1", method = method, name = '
                S' + str(idx))
```

```
            if p == q:
                S.connect(condition='i!=j')
            else :
                S.connect()
            net.add(S)
            idx += 1

    for p in range(g):
        G = net.__getitem__('G' + str(p + 1))
        sp_m = b2.SpikeMonitor(G, name='sp_m' + str(p + 1)) # SPIKE
        st_m = b2.StateMonitor(G, ["V", "I_syn", "s"], record = True, dt = 1 *
            ms, name = 'st_m' + str(p + 1)) # STATE

        net.add(sp_m)
        net.add(st_m)

    net.run(rt * ms, profile = True)
    return net
```

Listing 10: Function to run sub-group random time-stepping simulations (*simulation.py*).

```
from numba import jit
@jit(nopython=True)
def nsrts_python_simulation(I_syn_bar_, dt_fixed = 0.25, c = 0.5, dt_rad =
    0.5):
     c = 0.5
     t = np.zeros(neurons)
     I_syn_bar_ *= 10**-6 / 10**-4
     i = 0
     t_rec = np.zeros(neurons)
     v = np.zeros(neurons)
     f = np.zeros(neurons)
     s = np.zeros(neurons)
     Vs = np.zeros((neurons, 5000))
     # Initialisations
     for n in range(neurons):
         v[n] =  Vl_ + I0_ / gl_ * (1 - np_.exp(-c * (n * T_) / (neurons * tau_
             )))
         s[n] = 1
     while t[i] < rt:
         # Generate random dt
         dt = rd.uniform(dt_fixed * (1 - dt_rad), dt_fixed * (1 + dt_rad))
         if  v[i] > theta_:
             v[i] = Vl_
             for j in range(neurons):
                 if i != j:
                     s[j] += 1
         curr_neuron_ts = int(t_rec[i])
         if t[i] >= 5000 + curr_neuron_ts:
             Vs[i, curr_neuron_ts] = v[i]
             t_rec[i] += 1
```

```
        t[i] += dt
        I_syn = (I_syn_bar_ / neurons * f[i] * tau_)
        v[i] += (dt * 10**-3) * (-gl_ * (v[i] - Vl_) + I_syn + I0_) / C_
        f[i] += (dt * 10**-3) * (k_ * s[i] - f[i]) / tau1_
        s[i] += (dt * 10**-3) * (- s[i] / tau2_)

        i = argmin(t)
    return Vs
```

Listing 11: Individual random time-stepping simulation in Python accelerated with Numba (*simulations.py*).

```
gl_ = 0.1 * 10 **-3 / 10 ** -4
Vl_ =-60 * 10 **-3
C_ = 1 * 10 **-6 / 10 **-4
tau_ = 10 * 10 **-3
theta_ = -40 * 10 **-3
tau1_ = 3 * 10 **-3
tau2_ = 1 * 10 **-3
I0_ = 2.3 * 10 **-6 / 10 **-4
T_ = -tau_ * log(1 - gl_/I0_ * (theta_ - Vl_))
k_ = 1 / tau2_
```

Listing 12: Dimensionless model parameters for Python-Numba simulation (*model.py*).

F Additional Syncronisation Analysis

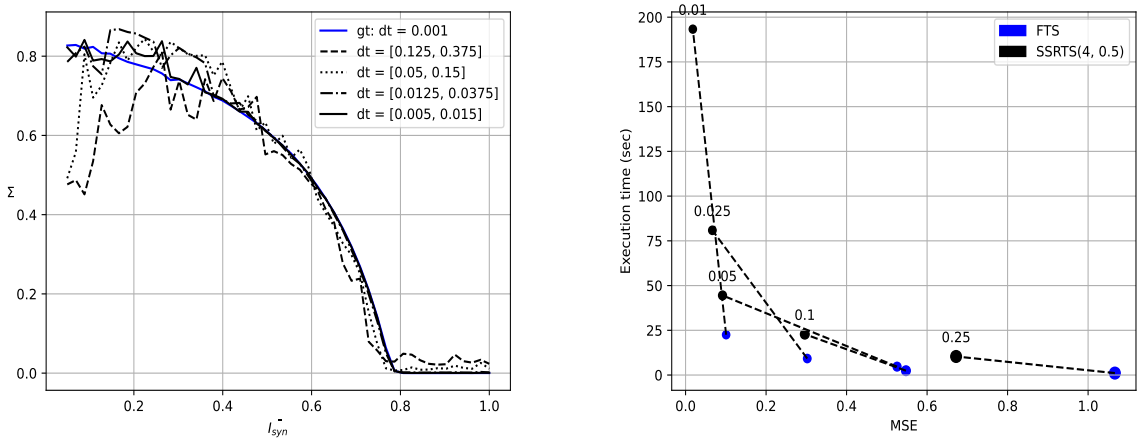

Figure 20: Synchronisation measure for SSRTS(4, 0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right).

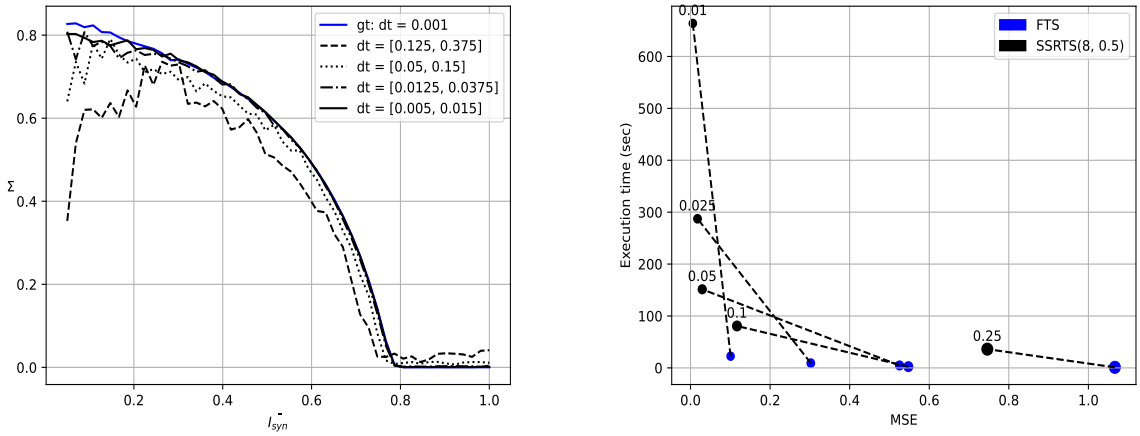

Figure 21: Synchronisation measure for SSRTS(8, 0.5) simulations with different time-steps dt (left) and its MSE performance comparison with FTS (right).