Automatically selecting a suitable integration scheme for systems of differential equations in neuron models.

Inga Blundell [1,*], Dimitri Plotnikov [2,3], Jochen Martin Eppler [2] and Abigail Morrison [1,2,4]

[1] *Institute of Neuroscience and Medicine (INM-6), Institute for Advanced Simulation (IAS-6), JARA BRAIN Institute I, Forschungszentrum Jülich, Jülich, Germany*
[2] *Simulation Lab Neuroscience, Institute for Advanced Simulation, JARA, Forschungszentrum Jülich, Jülich, Germany*
[3] *Chair of Software Engineering, Jülich Aachen Research Alliance (JARA), RWTH Aachen University, Aachen, Germany*
[4] *Institute of Cognitive Neuroscience, Faculty of Psychology, Ruhr-University Bochum, Bochum, Germany*

Correspondence*:
Inga Blundell
i.blundell@fz-juelich.de

ABSTRACT

On the level of the spiking activity, the integrate-and-fire neuron is one of the most commonly used descriptions of neural activity. A multitude of variants has been proposed to cope with the huge diversity of behaviors observed in biological nerve cells. The main appeal of this class of model is that it can be defined in terms of a hybrid model, where a set of mathematical equations describes the sub-threshold dynamics of the membrane potential and the generation of action potentials is often only added algorithmically without the shape of spikes being part of the equations. In contrast to more detailed biophysical models, this simple description of neuron models allows the routine simulation of large biological neuronal networks on standard hardware widely available in most laboratories these days.

The time evolution of the relevant state variables is usually defined by a small set of ordinary differential equations (ODEs). A small number of evolution schemes for the corresponding systems of ODEs are commonly used for many neuron models, and form the basis of the neuron model implementations built into commonly used simulators like Brian, NEST and NEURON.

However, an often neglected problem is that the implemented evolution schemes are only rarely selected through a structured process based on numerical criteria. This practice cannot guarantee accurate and stable solutions for the equations and the actual quality of the solution depends largely on the parametrization of the model.

In this article, we give an overview of typical equations and state descriptions for the dynamics of the relevant variables in integrate-and-fire models. We then describe a formal mathematical process to automate the design or selection of a suitable evolution scheme for this large class of models. Finally, we present the reference implementation of our symbolic analysis toolbox for ODEs that can guide modelers

during the implementation of custom neuron models.

Keywords: integrate-and-fire neuron, model dynamics, numerics, integration schemes, ODE, symbolic analysis

1 INTRODUCTION

In common with all body cells, nerve cells (*neurons*) are delimited by a bi-lipid layer (the *cell membrane*) which is largely impermeable for ions and bigger molecules. Active ion pumps and passive channels embedded into the membrane allow the selective passage of certain ions. Through these transporter molecules, neurons maintain a gradient of different ion types across the membrane, which leads to the *membrane potential* (Kandel et al., 2013).

In the absence of input, the membrane potential fluctuates around the *resting potential* E_L (typically at around $-70\,\mathrm{mV}$). Excitatory input depolarizes the membrane, driving the membrane potential closer to zero, while inhibitory input hyperpolarizes the neuron, driving the membrane potential away from zero. If the membrane potential crosses the *spiking threshold* θ (typically at around $-55\,\mathrm{mV}$), the neuron fires an action potential (*spike*), which is transmitted to all downstream (*postsynaptic*) neurons, where it in turn elicits excursions of their membrane potentials.

The basic integrate-and-fire model describes the dynamics of the membrane potential in the following way: the time evolution of the membrane potential V is governed by a differential equation of the type

$$\frac{d}{dt}V(t) = R(V(t), \cdot) \tag{1}$$

where R can be a function of other variables alongside V, whose time evolution is described by another ordinary differential equation which can again contain the membrane potential:

$$\frac{d}{dt}\mathbf{X} = \frac{d}{dt}\begin{pmatrix} V \\ x_1 \\ \vdots \\ x_n \end{pmatrix}(t) = \begin{pmatrix} R_0(\mathbf{X}) \\ R_1(\mathbf{X}) \\ \vdots \\ R_n(\mathbf{X}) \end{pmatrix}$$

Once the membrane potential reaches its threshold θ, a spike is fired and the membrane potential is set back to E_L for a certain amount of time called the *refractory period*. After this time the evolution of Equation 1 starts again. An important simplification in most models compared to biology is that the exact course of the membrane potential during the spike is either completely neglected or only considered partially. Threshold detection is typically added algorithmically on top of the sub-threshold dynamics.

The two most common variants of this type of model are the *current-based* and the *conductance-based* integrate-and-fire model. For the current-based model we have the following general form of the equation:

$$\begin{aligned} \frac{d}{dt}V(t) &= \frac{1}{\tau}(E_\mathrm{L} - V(t)) \\ &\quad + \frac{1}{C}I(t) + F(V(t)). \end{aligned} \tag{2}$$

Here C is the *membrane capacitance*, τ the *membrane time constant* and I the *input current* to the neuron. If we assume that spikes are constrained to a fixed temporal grid, $I(t)$ represents the sum of the currents elicited by all incoming spikes at all grid points for times smaller than t, plus a piecewise constant function I_ext that models additional external input. F, in contrast to the first part of the right-hand-side of Equation 2, is some non-linear function of V that may also be zero.

For the conductance-based integrate-and-fire model we have:

$$\frac{d}{dt}V(t) = \frac{1}{\tau}(E_\mathrm{L} - V(t)) + \frac{1}{C}G(t)(V(t) - E) + F(V(t)). \quad (3)$$

G has the same form as I but models a conductance rather than a current. E is the *reversal potential* at which there is no net flow of ions from one side of the membrane to the other (for details see Kandel et al., 2013). Equation 3 will usually contain several summands $\frac{1}{C}G_i(t)(V(t) - E_i)$ for differing G_i and corresponding E_i, e.g. for inhibitory and excitatory synaptic conductance. For simplicity we assume only one summand. The differential equations for both the current- and conductance-based models are linear when $F \equiv 0$. For the current-based model this means that Equation 2 is a linear *constant coefficient* differential equation.

An example of a neuron model described by a system of differential equations, where $F \not\equiv 0$ is the *adaptive exponential integrate-and-fire model*:

$$\begin{aligned}\frac{d}{dt}V(t) &= \frac{1}{\tau}(E_\mathrm{L} - V(t)) \\ &\quad + \frac{1}{C}G(t)(V(t) - E) \\ &\quad + g \cdot \delta \cdot \exp\left(\frac{V(t) - V_\mathrm{T}}{\delta}\right) - w(t) \\ \frac{d}{dt}w(t) &= \frac{c}{\tau_w}(V(t) - E_\mathrm{L})\end{aligned}$$

For the biophysical meaning of the variables V_T, δ, g, c, τ_ω and w see the original publication by Brette and Gerstner (2005).

Current-based neuron models with $F \not\equiv 0$ are unusual because models from this category are chosen primarily for their simplicity, while conductance-based neuron models are believed to describe neuronal activity in the brain more accurately, albeit at the cost of more complex differential equations.

It should be noted here that although some neuron models are not explicitly referred to or described as *current-based* or *conductance-based* models in the literature their time evolution can still be expressed by differential equations of the mathematical forms shown in Equations 2 and 3.

The choice of an appropriate solver for a given equation is a non-trivial task, as it requires deep knowledge of ordinary differential equations and numerics to assess the type of differential equation and construct an appropriate numeric solver. This choice depends not only on the form of the differential equation but also on the magnitude of the occurring parameters. For example, Rotter and Diesmann (1999) demonstrated that for neuron models that can be expressed as time-invariant linear systems, the analytical solution to the evolution of the dynamics from one time step to the next can be achieved by a matrix multiplication. If applicable, this kind of solution is to be preferred, as it is both exact and computationally efficient.

However, this approach leaves two key steps up to the modeller: firstly, analyzing the dynamics to discern what category of dynamical system it is; secondly, having performed this analysis, to construct the appropriate solver, e.g. the terms of the propagator matrix for such neurons that can be solved in this way (Rotter and Diesmann, 1999) or the configuration of an implicit or explicit numeric solver for all other neuron models. As these steps can be quite challenging to many modellers, it would be of great use to have a framework capable of automatically performing this analysis and solver construction.

In Section 2 we therefore first derive compact canonical representations of the equations and their parts that allow an efficient implementation on a computer system, and then show that the distinction between current- and conductance-based, linear and non-linear, stiff and non-stiff systems of differential equations is important for automatizing the construction or selection of an optimal evolution scheme.

Our reference implementation follows the mathematical observations and is described in Section 3.

Section 4 demonstrates our application of the framework to some commonly used models in computational neuroscience and explains the integration of the framework into the NEST Modeling Language (NESTML; Plotnikov et al., 2016). We close with a presentation of related work in Section 5 and a discussion and outlook in Section 6, where we summarize possible extensions and further applications of our system.

2 MATERIALS AND METHODS

As already pointed out in the previous section, systems of differential equations describing the dynamics in neuron models can be divided into *current-based* and *conductance-based* systems. Additional distinguishing properties are whether the systems are *linear* or *non-linear*, *stiff* or *non-stiff*. We will now describe how these properties influence the choice of an appropriate solver.

For the current-based integrate-and-fire neuron with $F \equiv 0$, we have a first order constant coefficient linear differential equation where I typically satisfies a homogeneous linear differential equation of some order $n \in \mathbb{N}$. Any such ODE or system of ODEs can be solved analytically and efficiently as we will show in Section 2.1.

When evolving systems of ODEs for conductance-based linear or non-linear ODEs, it is necessary to use a numeric integration scheme. Depending on the system at hand, it is advisable to choose either an implicit or an explicit stepping function (Section 2.2).

2.1 Solving linear constant coefficient ODEs analytically

For simplicity we will assume E_L in Equation 2 to be zero or to be included in one of the other terms of the right hand side. As shown by Rotter and Diesmann (1999), if $V : \mathbb{R} \to \mathbb{R}$ satisfies the first order constant coefficient linear differential equation

$$\frac{d}{dt}V(t) = -\frac{1}{\tau}V(t) + \frac{1}{C}I(t) \tag{4}$$

with initial value $V(0) = V_0$, for a function $I : \mathbb{R}^+ \to \mathbb{R}$ and constants C (the capacitance of the membrane) and τ (the membrane time constant), and if I satisfies

$$\left(\frac{d}{dt}\right)^n I = \sum_{i=0}^{n-1} a_i \left(\frac{d}{dt}\right)^i I \tag{5}$$

for some $n \in \mathbb{N}$ and a sequence $(a_i)_{i\in\mathbb{N}} \subset \mathbb{R}$, an analytical solver can be constructed in the form of a propagator matrix.

Here, we show how to evaluate the dynamics to discern whether V and I do indeed satisfy the conditions stated above, and how to derive the evolutions scheme for V accordingly. First, we verify that the first order differential equation, $\frac{d}{dt}V = r(V)$, for a right hand side $r : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, is indeed linear with a constant coefficient, i.e. that $\left(\frac{d}{dV}\right)^2 r(V) = 0$ and $\left(\frac{d}{dV}\right) r(V)(t)$ is constant. Second we methodically determine whether I satisfies a linear differential equation of some order n, i.e. we check whether

$$\frac{d}{dt}I = a_0 I \tag{6}$$

for some $a_0 \in \mathbb{R}$ by solving for a_0. If no such a_0 exists we check whether

$$\left(\frac{d}{dt}\right)^2 I = a_0 I + a_1 \frac{d}{dt} I \tag{7}$$

for some $a_0, a_1 \in \mathbb{R}$ using the following procedure: we assume that a_0, a_1 exist such that (7) is satisfied. Then we have for some $t_1, t_2 \in \mathbb{R}$ (for example $t_1 = 1, t_2 = 2$):

$$\mathbf{X}(t_1,t_2) := \begin{pmatrix} I(t_1) & \frac{d}{dt}I(t_1) \\ I(t_2) & \frac{d}{dt}I(t_2) \end{pmatrix},$$

$$\mathbf{X}(t_1,t_2) \cdot \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} \left(\frac{d}{dt}\right)^2 I(t_1) \\ \left(\frac{d}{dt}\right)^2 I(t_2) \end{pmatrix}$$

If $\det(\mathbf{X}(t_1,t_2)) \neq 0$ we therefore know that

$$\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \mathbf{X}^{-1}(t_1, t_2) \cdot \begin{pmatrix} \left(\frac{d}{dt}\right)^2 I(t_1) \\ \left(\frac{d}{dt}\right)^2 I(t_2) \end{pmatrix}.$$

Under the assumption that (7) is satisfied and that $\det(\mathbf{X}(t_1, t_2)) \neq 0$ this gives us a_0 and a_1. If our second assumption is not satisfied we can easily chose t_1 and t_2 so that it is. We can now determine whether the first assumption is correct by inserting the calculated values for a_0 and a_1 and checking if the following equation is true:

$$\left(\frac{d}{dt}\right)^2 I - a_0 I - a_1 \frac{d}{dt} I = 0 \qquad (8)$$

Now, if such a_0 and a_1 exist, they are unique, as I and $\frac{d}{dt}I$ are linearly independent, since there was no $a_0 \in \mathbb{R}$ such that (6) was satisfied. If a_0 and a_1 do not satisfy (8), we check methodically if constants $(a_i)_{i\in\mathbb{N}} \subset \mathbb{R}$ exist, for which (5) is satisfied for $n = 3, 4, \ldots$. Again we assume that $a_0, \ldots, a_n \in \mathbb{R}$ exist such that (5) is satisfied. Then we have for $t = (t_1, \ldots, t_n) \in \mathbb{R}^n$ (for example $t_1 = 1, \ldots, t_n = n$):

$$\mathbf{X}(t) := \begin{pmatrix} I(t_1) & \cdots & \left(\frac{d}{dt}\right)^{n-1} I(t_1) \\ \vdots & \ddots & \vdots \\ I(t_n) & \cdots & \left(\frac{d}{dt}\right)^{n-1} I(t_n) \end{pmatrix}, \qquad (9)$$

$$\mathbf{X}(t) \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \left(\frac{d}{dt}\right)^n I(t_1) \\ \vdots \\ \left(\frac{d}{dt}\right)^n I(t_n) \end{pmatrix}. \qquad (10)$$

If $\det(\mathbf{X}(t)) \neq 0$ we get

$$\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = \mathbf{X}^{-1}(t) \cdot \begin{pmatrix} \left(\frac{d}{dt}\right)^n I(t_1) \\ \vdots \\ \left(\frac{d}{dt}\right)^n I(t_n) \end{pmatrix}. \qquad (11)$$

Again, if $\det(\mathbf{X}(t)) = 0$ we simply use another t, for example $t = (t_1+1, \ldots, t_n+1)$. Then we obtain the values of $a_0, \ldots, a_n$ under the assumption that (5) is satisfied for order n. We check whether the assumption in (5) is true by symbolically evaluating whether

$$\left(\frac{d}{dt}\right)^n I - \sum_{i=0}^{n-1} a_i \left(\frac{d}{dt}\right)^i I = 0.$$

If (5) is not satisfied we go on to check

$$\left(\frac{d}{dt}\right)^{n+1} I = \sum_{i=0}^{n} a_i \left(\frac{d}{dt}\right)^i I$$

for some $a_0, \ldots, a_{n+1}$, and so on. This way, for every I that satisfies (5) for order n we can determine the factors $a_0, \ldots, a_n$. Then we can rephrase (4) as the *homogeneous* differential equation

$$\frac{d}{dt}\mathbf{y}(t) = \mathbf{A}\mathbf{y}(t) \qquad (12)$$

with initial values $\mathbf{y}(0) = \mathbf{y}_0$, $\mathbf{y} = (\frac{d^{n-1}}{dt^{n-1}} I, \frac{d^{n-2}}{dt^{n-2}} I, \ldots, I, V)$ and

$$\mathbf{A} = \begin{pmatrix} a_{n-1} & a_{n-2} & \cdots & \cdots & a_0 & 0 \\ 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 & 0 \\ 0 & 0 & \ddots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & \frac{1}{C} & -\frac{1}{\tau} \end{pmatrix} \qquad (13)$$

Thus for $n = 1$ we have

$$\mathbf{A} = \begin{pmatrix} a_0 & 0 \\ \frac{1}{C} & -\frac{1}{\tau} \end{pmatrix}$$

and for $n = 2$ we have

$$\mathbf{A} = \begin{pmatrix} a_1 & a_0 & 0 \\ 1 & 0 & 0 \\ 0 & \frac{1}{C} & -\frac{1}{\tau} \end{pmatrix}$$

As it can be both more convenient and computationally more efficient when $\mathbf{A}$ is a *lower triangular* matrix we give an alternative choice of $\mathbf{A}$ and $\mathbf{y}$, where $\mathbf{A}$ is a triangular matrix:

$$\mathbf{A} = \begin{pmatrix} a_1 + x & 0 & 0 \\ 1 & -x & 0 \\ 0 & \frac{1}{C} & -\frac{1}{\tau} \end{pmatrix} \tag{14}$$

where

$$x = -\frac{a_1}{2} + \sqrt{\frac{a_1^2}{4} + a_0} \tag{15}$$

and

$$\mathbf{y} = \left(\frac{d}{dt} I + xI, I, V \right). \tag{16}$$

Then we can determine the solution $\mathbf{y}$ at $t \in \mathbb{R}^+$ using the matrix exponential:

$$\mathbf{y}(t) = e^{\mathbf{A}t} \mathbf{y}_0 \tag{17}$$

We can rephrase this to obtain an incremental formulation which allows the evolution of the system by a single calculation of $e^{\mathbf{A}h}$ for a fixed step size $h \in \mathbb{R}^+$:

$$\mathbf{y}(t+h) = e^{\mathbf{A}(t+h)} \cdot \mathbf{y}_0 = e^{\mathbf{A}h} \cdot \mathbf{y}_t.$$

It is important to note here that the exact integration of (2) depends on the exact calculation of $e^{\mathbf{A}h}$. Let $I(t)$ be the sum of currents elicited by all incoming spikes at all grid points for times $t_i \leq t$,

$$I(t) = \sum_{i \in \mathbb{N}, t_i \leq t} \sum_{k \in S_{t_i}} I_k(t),$$

where $I_k(t) = \widehat{\iota}_k \iota(t - t_i)$, for $t \in \mathbb{R}^+$. $\widehat{\iota}_k$ is the *synaptic weight* of synapse k and ι satisfies the differential equation (5) on $\mathbb{R}^+$ for some constants $(a_i)_{i \in \mathbb{N}} \subset \mathbb{R}$ and some $n \in \mathbb{N}$. Then I satisfies the differential equation (5) on $\mathbb{R}^+ \setminus \{t_1, \ldots, t_k\}$. Therefore we can consider I as the solution of the differential equation (5) on the intervals $(0, t_1), (t_1, t_2), \ldots$ with suitable initial values. For $t \in (t_{i-1}, t_i)$ we can calculate

$$\mathbf{y}(t) = e^{\mathbf{A}(t - t_{i-1})} \mathbf{y}_{t_{i-1}}.$$

At time t_i, for $i \in \mathbb{N}$, the differential equation (5) is not satisfied because ι does not satisfy the equation at $t = 0$, but we get $I(t_i)$ by continuous continuation to the boundary of the interval (t, t_i). The derivatives of I contained in $\mathbf{y}$ must be updated by initial values of additional spikes at time t_i, meaning for $\mathbf{P}(h) = e^{\mathbf{A}h}$

$$\mathbf{y}(t_i) = \mathbf{P}(h)\mathbf{y}(t_{i-1}) + \mathbf{x}_{\mathbf{t_i}},$$

where

$$\mathbf{x}_{\mathbf{t_i}} = \mathbf{T} \begin{pmatrix} \left(\frac{d}{dt}\right)^n \iota(0) \\ \vdots \\ \frac{d}{dt} \iota(0) \\ 0 \\ 0 \end{pmatrix} \sum_{k \in S_{t_i + h}} \widehat{\iota}_k.$$

Here $\mathbf{T} \in \mathbb{R}^{n+1} \times \mathbb{R}^{n+1}$ is such that

$$\mathbf{y} = \mathbf{T} \begin{pmatrix} \left(\frac{d}{dt}\right)^{n-1} I \\ \vdots \\ I \\ V \end{pmatrix}$$

$\mathbf{T}$ is the identity matrix when $\mathbf{y}$ is chosen as the vector of derivatives as in Equation 12 and Equation 13 but it may well be non-trivial, e.g. when $\mathbf{y}$ is chosen as in Equation 16.

Now we know an analytical and efficient way to evolve any linear constant coefficient ODE containing the convolution of the solution of a linear homogeneous ODE and a weighted spike train.

2.1.1 Adding a constant external input current

A common requirement in neuroscientific modeling is to add a bias current to neurons. We will now show how to solve the differential equation when we have an additional constant external input current I_{E}:

$$\frac{d}{dt}V(t) = -\frac{V(t)}{\tau} + \frac{1}{C}(I(t) + I_{\mathrm{E}}),\ V(0) = V_0$$

As shown above, we can solve

$$\frac{d}{dt}V_1 = -\frac{V_1(t)}{\tau} + \frac{I(t)}{C},\ V_1(0) = V_{1_0}. \qquad (18)$$

Consider the following differential equation,

$$\frac{d}{dt}V_2 = -\frac{V_2(t)}{\tau} + \frac{I_{\mathrm{E}}}{C},\ V_2(0) = V_{2_0}, \qquad (19)$$

where τ, C and I_{E} are constants. By *variation of constants* (Walter, 2000) we have a solution of (19):

$$\begin{aligned} V_2(t) &= \left(\frac{I_{\mathrm{E}}\tau}{C}e^{t/\tau} + V_{2_0}\right)e^{-t/\tau} \\ &= \frac{I_{\mathrm{E}}\tau}{C} + V_{2_0}e^{-t/\tau}, \end{aligned}$$

$$\begin{aligned} V_2(t+h) &= \frac{I_{\mathrm{E}}\tau}{C} + V_{2_0}e^{-t/\tau}e^{-h/\tau} \\ &= V_2(t)e^{-h/\tau} + \frac{I_{\mathrm{E}}\tau}{C}(1 - e^{-h/\tau}). \end{aligned}$$

Now we know solutions V_1 and V_2 of (18) and (19). Therefore $V := V_1 + V_2$ solves

$$\begin{aligned} \frac{d}{dt}V = \frac{d}{dt}(V_1 + V_2) &= -\frac{V_1(t) + V_2(t)}{\tau} \\ &+ \frac{1}{C}(I(t) + I_{\mathrm{E}}) \\ &= \frac{V(t)}{\tau} + \frac{1}{C}I(t) + \frac{I_{\mathrm{E}}}{C}. \end{aligned}$$

and for $\mathbf{P} := \mathbf{P}(h) = e^{\mathbf{A}h}$ the following holds

$$\begin{aligned} V(t+h) = &\mathbf{P}_{n+1,1}\mathbf{y}_1(t) + \cdots \\ &+ \mathbf{P}_{n+1,n+1}V_1(t) + V_2(t)e^{-h/\tau} \\ &+ \frac{I_{\mathrm{E}}}{C}(1 - e^{-h/\tau}). \end{aligned}$$

As the last column a in $\mathbf{A}$ has only one entry $a_{n+1} = \frac{-1}{\tau}$ and $\mathbf{P} = e^{\mathbf{A}h} = \sum_{k=0}^{\infty}\frac{(\mathbf{A}h)^k}{k!}$,

$$\begin{aligned} \mathbf{P}_{n+1,n+1} &= \left(\sum_{k=0}^{\infty}\frac{(\mathbf{A}h)^k}{k!}\right)_{n+1,n+1} \\ &= \sum_{k=0}^{\infty}\frac{(\frac{-h}{\tau})^k}{k!} = e^{-h/\tau}. \end{aligned}$$

We get:

$$\begin{aligned} V(t+h) = &\mathbf{P}_{n+1,1}\mathbf{y}_1(t) + \cdots \\ &+ \mathbf{P}_{n+1,n}\mathbf{y}_n(t) \\ &+ V(t)e^{-h/\tau} + \frac{I_{\mathrm{E}}\tau}{C}(1 - e^{-h/\tau}). \end{aligned}$$

This method is also applicable when we have a piece-wise constant function $\widehat{y}_0$ instead of a constant I_{E}:

$$\frac{d}{dt}V_2 = -\frac{V_2(t)}{\tau} + \frac{\widehat{y}_0}{C},\ V_2(0) = V_{2_0}.$$

where for all $i \in \mathbb{N}$ there is a $c_i \in \mathbb{R}$ such that $\widehat{y}_0(t) = c_i$ for all $t \in [t_i, t_i + h)$. We rephrase the problem as:

$$\frac{d}{dt}V_{2_i} = -\frac{V_{2_i}(t)}{\tau} + \frac{c_i}{C},\ V_{2_i}(0) = V_{2_{i0}}$$

on $t \in [t_i, t_i + h)$ for all $i \in \mathbb{N}$ and get

$$V_2(t_i) = \frac{c_i\tau}{C} + V_2(t_{i-1})e^{-h/\tau}$$

and

$$V(t_i) = V(t_{i-1})e^{-h/\tau} + \frac{c_i\tau}{C}(1 - e^{-h/\tau}).$$

Now we have an exact description for how to handle the evolution of linear constant coefficient ODEs containing the convolution of the solution of a linear homogeneous ODE and a weighted spike train with an additional constant external input, that is still analytical and efficient.

2.1.2 Handling sums

The approximation of postsynaptic currents observed in real brain experiments is sometimes best modeled by different functions for different synapses. We can handle the case when I is the sum of functions I_1, I_2 which satisfy a homogeneous differential equation of arbitrary order m and n in the following way. As seen above if V_1 is a solution of

$$\frac{d}{dt}V_1(t) = -\frac{V_1(t)}{\tau} + \frac{1}{C}I_1(t)$$

and V_2 is a solution of

$$\frac{d}{dt}V_2(t) = -\frac{V_2(t)}{\tau} + \frac{1}{C}I_2(t)$$

then $V = V_1 + V_2$ is a solution of

$$\frac{d}{dt}V(t) = -\frac{V(t)}{\tau} + \frac{1}{C}(I_1(t) + I_2(t)).$$

If, furthermore, I_1 satisfies (5) for $n \in \mathbb{N}$

$$\begin{aligned} V_1(t+h) &= \mathbf{P}^1_{n+1,1}\mathbf{y}_{1_1}(t) + \cdots \\ &\quad + \mathbf{P}^1_{n+1,n}\mathbf{y}_{1_n}(t) + V_1(t)e^{-h/\tau}. \end{aligned}$$

where $\mathbf{P}^1$ is the corresponding propagator matrix and I_2 satisfies (5) for some $m \in \mathbb{N}$

$$\begin{aligned} V_2(t+h) &= \mathbf{P}^2_{m+1,1}\mathbf{y}_{2_1}(t) + \cdots \\ &\quad + \mathbf{P}^2_{m+1,m}\mathbf{y}_{2_m}(t) + V_2(t)e^{-h/\tau} \end{aligned}$$

where $\mathbf{P}^2$ is the corresponding propagator matrix, then

$$\begin{aligned} V(t+h) &= \mathbf{P}^1_{n+1,1}\mathbf{y}_1(t) + \cdots \\ &\quad + \mathbf{P}^1_{n+1,n}\mathbf{y}_1(t) \\ &\quad + \mathbf{P}^2_{m+1,1}\mathbf{y}_{2_1}(t) + \cdots \\ &\quad + \mathbf{P}^2_{m+1,m}\mathbf{y}_{2_m}(t) + V(t)e^{-h/\tau}. \end{aligned}$$

Therefore we just need to compute two propagator matrices to handle the sum.

2.2 Choice of a suitable numeric integration scheme

Explicit methods for solving differential equations are methods that only use already known values of the function at earlier grid points to determine the value at the next grid point. The efficiency and accuracy of explicit methods is typically sufficient for systems of ODEs used to model neuronal behavior. Popular examples of such methods are the explicit 4th order classical Runge-Kutta or the explicit embedded Runge-Kutta-Fehlberg method (Dahmen and Reusken, 2005) for the approximative solution of ODEs. Most neuron model implementations currently use explicit stepping algorithms and still achieve satisfactory results in terms of accuracy and simulation time (Morrison et al., 2007; Hanuschkin et al., 2010). However, some published models involve possibly *stiff* differential equations (e.g. Brette and Gerstner, 2005), which potentially require a different class of solvers.

Lambert (1992) defines stiffness as follows:

> If a numerical method [...] applied to a system with any initial conditions, is forced to use in a certain interval of integration a steplength which is excessively small in relation to the smoothness of the exact solution in that interval, then the system is said to be stiff in that interval.

A typical case of stiffness is for example, when different parts of the solution of a system of equations decays on different time scales.

This usually comes from very different scales inherent to the ODE. These scales will reflect in the parameters of the equations, i.e. the range of constants occuring in the equations of the systems. Therefore the stiffness of a system always depends not only on the mathematical form of the equations but heavily on the magnitude of the constants occuring in them.

In principle it is possible to solve stiff equations with explicit methods, but this comes at the expense of a very small step size when using an adaptive step size algorithm and trying to achieve a certain accuracy. This in turn leads to high computational costs. For non-adaptive step size algorithms it leads to plain wrong results without the user knowing, since the algorithm still terminates, but with large error. Moreover, as the limited machine precision on a digital computer constitutes a lower bound for the step size, explicit methods usually become unstable when applied to stiff problems.

Implicit methods, on the other hand, do not use previous values to calculate the solution at the next grid point, but only employ them implicitly in the form of the solution of a system of equations. This makes implicit methods computationally much more costly, but usually allows a larger step size to be chosen, thus avoiding stability problems (Strehmel and Weiner, 1995).

In order to detect whether an explicit or implicit method is better suited for a given ODE we devise the following testing strategy.

First, we choose representative spike trains (drawn from a Poisson distribution) and compute approximate solutions for the given system of ODEs using an explicit and implicit method of the same order:

1. an explicit 4th order Runge-Kutta method
2. an implicit Bulirsch-Stoer method of Bader and Deuflhard (Strehmel and Weiner, 1995)

both with adaptive step size. We can then compare them with respect to the required *average step size* and *minimal step size*. In cases where the implicit method performs better than the explicit method, we have reason to believe that the ODE is stiff and that the use of an implicit method is advisable.

Although ODEs may be stiff only for very specific initial conditions, usually stiffness should be observable for a wide range of initial values, or in this case for a number of incoming spike trains (Strehmel and Weiner, 1995). By choosing many spike trains, evaluating the required step sizes for the implicit and explicit method for each of them, and comparing that to the machine precision ε, it is thus possible to detect whether the problem at hand is stiff or not. We propose the following rules for choosing an implicit algorithm:

- if the minimal step size of runs using the explicit method is close to machine precision (i.e. less than $10 \cdot \varepsilon$) and this is not the case for the minimal step size of runs using the implicit method (i.e. greater than or equal to $10 \cdot \varepsilon$) this is a hint that the system of ODEs is possibly stiff. In this case an explicit stepping function could become unstable or even abort, so we suggest the use of an implicit algorithm.

- if the minimal step size of runs using the explicit method is reasonably large (i.e. greater than or equal to $10 \cdot \varepsilon$) we have to test two cases:
  - if the minimal step size of runs of the implicit method is very small (i.e. less than $10 \cdot \varepsilon$), we suggest using an explicit method.
  - if the minimal step size of runs of the implicit method is large (i.e. greater than or equal to $10 \cdot \varepsilon$), we go on to check if the average step size of runs using the implicit algorithm is much larger than the average step size of runs using the explicit algorithm. If this is the case, this again indicates that the system of ODEs is stiff and therefore choosing an implicit evolution method is advisable.

For a non-stiff system of ODEs, the computation time of an explicit algorithm should be lower, as it does not require the solution of a system of equations (Dahmen and Reusken, 2005). Therefore the choice of an explicit evolution method is sensible in cases where none of the above conditions are met. The algorithm that follows from these rules is depicted in Figure 2.

3 REFERENCE IMPLEMENTATION

In order to automate the process of finding the most appropriate solver for a given system of ODEs on a computer, we have designed and implemented an analysis toolbox in Python (`http://github.com/nest/ode-toolbox`). It builds on the formal mathematical foundations introduced in the previous sections and uses SymPy (Meurer et al., 2017) to carry out symbolic mathematical tests and transformations. To achieve a high degree of portability and re-usability, the input to the algorithm is given either in the form of JSON files or Python dictionaries, which specify equations, parameters and additional properties (for an example, see Section 3.4). These two means of input allow an easy embedding of the toolkit into third-party tool chains and enable us to leverage the Python and SymPy parsers, which delegates all syntax checking and exception handling to well established and tested tools.

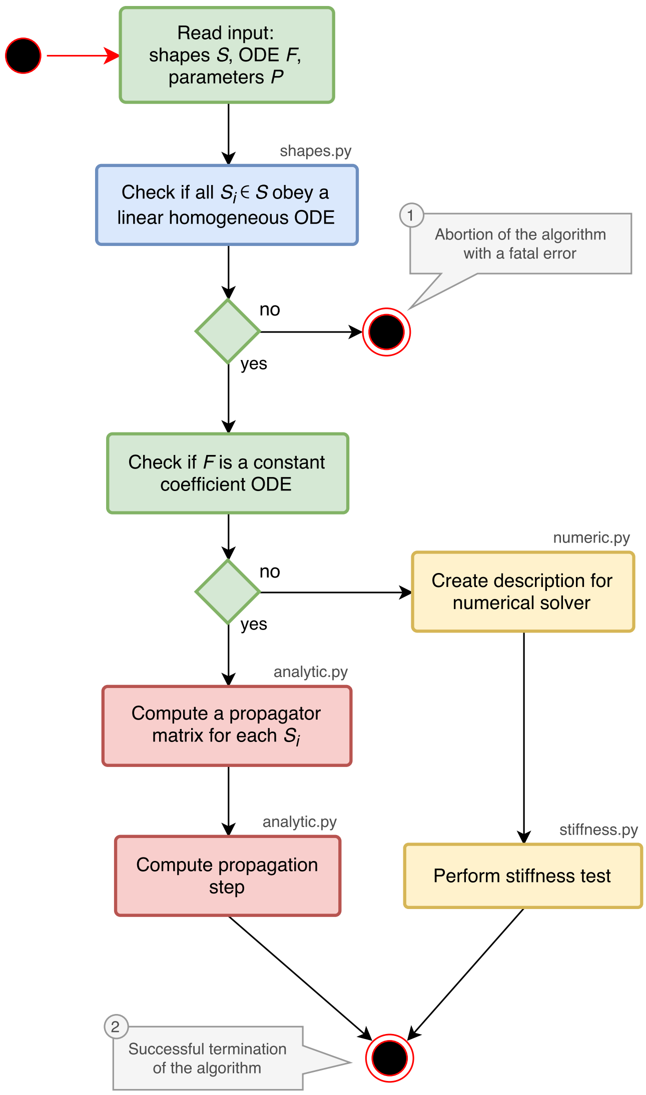

Figure 1. Activity diagram summarizing all steps of the ODE analysis algorithm. Steps executed in the main script of the toolbox are shown in green. The analysis of postsynaptic shapes (blue box) is detailed in Section 3.1. Parts shown in red represent the generation of an analytical solver, which is described in Section 3.2. The selection of a numerical stepper function is carried out by the yellow actions and explained in Section 3.3.

The algorithm expects three components in the input: i) an ODE describing the time evolution of a state variable (e.g. V), ii) a list of postsynaptic shapes (e.g. I) used within this ODE and specified either as functions of time or as ODEs with initial conditions and iii) a set of parameters with default values for the equations. Fundamentally, the analysis algorithm checks the given system of ODEs for membership of the following two major categories and generates or selects an appropriate solver accordingly:

1. First order linear constant coefficient ODEs for the dynamics of a state variable (see Equation 4) whose inhomogeneous part is a postsynaptic shape (i.e. satisfies Equation 5) can be solved exactly using an analytical stepping scheme (Section 2.1).
2. All other systems of ODEs have to be solved by a numerical solver. ODEs in this category are, for example, non-linear ODEs describing the time evolution of a state variables, or linear ODEs with an inhomogeneous part which is not a postsynaptic shape, i.e. not satisfying Equation 5.

The implementation of the analysis toolbox consists of different Python components which are introduced in the activity diagram in Figure 1. The main script orchestrates the execution of the analysis and uses the functions and classes of the different submodules:

`shapes.py` contains classes and functions for analyzing and storing postsynaptic shapes either given as functions of time or ODEs with initial values (blue parts in Figure 1). The main algorithm in this module is explained in section Section 3.1.

`analytic.py` provides the functionality to generate propagator matrices and compute a specification for the update step (red parts in Figure 1). A detailed description can be found in Section 3.2.

`numeric.py` contains the code for creating a description of the update step for further processing by the stiffness tester or a numerical stepper function (upper yellow box in Figure 1).

`stiffness.py` implements the stiffness tester (lower yellow box in Figure 1). This module can either be used as a module within the analysis toolbox or a third-party tool, or run in a standalone fashion. It is explained in Section 3.3 together with the preparatory steps carried out in `numeric.py`.

The main script starts by reading and validating the input from a JSON file or a Python dictionary. It expects the keys `shapes`, `odes` and `parameters` to be present in the input. For each postsynaptic shape in the `shapes` section, it runs the algorithm described in Section 3.1, which checks if the given postsynaptic shape obeys a linear homogeneous ODE and transforms it into a canonical representation suitable for further processing. If one of the postsynaptic shapes fails the test for linearity and homogeneity, the script terminates with an error (1 in Figure 1), because this class of ODEs cannot be solved easily with traditional methods as explained in Section 6.

After processing the postsynaptic shapes, the script checks whether all equations in the `odes` section of the input are linear constant coefficient ODEs: the ODE is linear if the right hand side of the ODE differentiated twice by its symbol is zero, the coefficient of the symbol is constant if the right hand side of the ODE differentiated by its symbol is constant. If these two tests succeed, the system can be solved analytically (see Section 3.2). If one of them fails, a numerical stepper has to be chosen (Section 3.3). The output of the main script is again a Python dictionary or a JSON file, which contains a specification of the most appropriate solver for the given input (2 in Figure 1). The remainder of this section explains the different algorithms in the submodules of the analysis toolbox.

3.1 Analysis of postsynaptic shapes

In the neuroscience literature, postsynaptic shapes are described either as functions of time or as ODEs with initial values. To provide users with maximum flexibility, both specifications are supported by our toolbox. Regardless of the form of the specification, each of the given postsynaptic shapes has to satisfy a linear, homogeneous ODE (Equation 5) to be solved either analytically or numerically.

In case the postsynaptic shape is given as an ODE with initial values, the check for linearity an homogeneity is straightforward. For each occurring derivative of the postsynaptic shape in the shape's definition, we simply have to iteratively subtract the product of the derivative and its factor from the original definition of the postsynaptic shape and check if the final difference is zero. This check fails if the postsynaptic shape is non-linear (i.e. at least one of the derivatives occurs as a power term) or not homogeneous (i.e. not all terms of the postsynaptic shape definition are products containing a derivative of the shape). This check is implemented in the function `shape_from_ode()` in the `shape` module of the toolbox.

In case the postsynaptic shape is given as a function of time, we check whether the function obeys a linear homogeneous ODE by trying to construct such an equation together with the initial values of all relevant derivatives. This procedure is implemented in the function `shape_from_function()` of the `shape` module. We start the evaluation by checking if the postsynaptic shape function obeys a linear homogeneous ODE of order 1.

```
1  t_value = None
2  ds = [shape, diff(shape, t)]
3  for t_ in range(1, max_t):
4    if ds[0].subs(t, t_) != 0:
5      t_value = t_
6      break
7
8  found_ode = False
9  if t_value is not None:
10   a0 = (1/ds[0] * ds[1]).subs(t, t_value)
11   diff_lhs_rhs = ds[1] - a0 * ds[0]
12   found_ode = diff_rhs_lhs == 0
```

In line 10 we calculate the factor a_0 from Equation 6 by dividing the first derivative of the postsynaptic shape by the shape at an arbitrary point t. To avoid a division by zero, we have to find a t so that the postsynaptic shape function is not zero at this t (lines 3-6). Line 11 calculates the difference between the left and the right hand side of Equation 6. If this difference is zero (line 12) we know that the postsynaptic shape satisfies a linear homogeneous ODE of order 1. We also know the ODE itself by calculating its initial value in line 40 below.

If the postsynaptic shape does not obey a linear homogeneous ODE of order 1, we check if the postsynaptic shape function satisfies a linear homogeneous ODE of a higher order. This test is run in a loop (line 15) that increments the order to check for each time Equation 5 is not satisfied. The loop terminates if either an ODE is found or `max_order` iterations are exceeded. The latter check prevents expensive tests of unlikely high orders.

```
13 order = 1
14 factors = [a0]
15 while not found_ode and order < max_order:
16   order += 1
17   ds.append(diff(ds[-1], t))
18   X = zeros(order)
19   Y = zeros(order, 1)
```

We start the loop by setting the next potential `order` (line 16), appending the next higher derivative of postsynaptic shape to the list of derivatives (line 17) and initializing the matrix $\mathbf{X}$ with size `order`$\times$`order` (Equation 9, line 18) and the vector $\mathbf{Y}$ with length `order` (right hand side of Equation 10, line 19).

```
20   invertible = False
21   for t_ in range(max_t):
22     for i in range(order):
23       substitute = i + t_ + 1
24       Y[i] = ds[order].subs(t, substitute)
25       for j in range(order):
26         X[i, j] = ds[j].subs(t, substitute)
27
28     if det(X) != 0:
29       invertible = True
30       break
```

$\mathbf{X}$ and $\mathbf{Y}$ are assigned values according to Equations 9 and 10 (line 24 and 26) for varying $t = (t_1, \ldots, t_n)$ (line 21) in order to find a t such that the matrix $\mathbf{X}$ is invertible, i.e $\det(\mathbf{X}) \neq 0$ (line 28). In the inner loop (line 22-26), t_i is substituted so that we first try $t = (1, \ldots, n)$, second $t = (2, \ldots, n+1)$ and so on (line 23).

If we find an invertible $\mathbf{X}$, we calculate the potential factors a_i from Equation 5 according to Equation 11 for the current order we are checking for (`factors`, line 32).

```
31   if invertible:
32     factors = X.inv() * Y
33     diff_rhs_lhs = 0
34     for k in range(order):
35       diff_rhs_lhs -= factors[k] * ds[k]
36     diff_rhs_lhs += ds[order]
37     if diff_rhs_lhs == 0:
```

```
38        found_ode = True
39        break
```

Lines 33-36 calculate the difference between the left and the right hand side of Equation 5. If this difference is zero (line 37) we know that the postsynaptic shape satisfies an linear homogeneous ODE of order `order`.

If we do not find an ODE during the execution of the `while` loop, we terminate the algorithm with an error (1 in Figure 1). If we do, we can go on to calculate the initial values of the postsynaptic shape equation by substituting t by 0 for all derivatives of the postsynaptic shape, which fully defines the found ODE.

```
40  iv = [x.subs(t, 0) for x in ds[:-1]]
```

In the case of successful termination, the functions `shape_from_ode()` and `shape_from_function()` both return a `Shape` object to the main script of the toolbox, which encapsulates all attributes of the postsynaptic shape required for further processing.

3.2 Generation of an analytical evolution scheme

If the ODE describing the update of a state variable was found to be a constant coefficient ODE and all postsynaptic shapes obey linear homogeneous ODEs, we can solve the system of ODEs analytically according to Section 2.1. To this end, the module `analytic` provides a class `Propagator`, which has two member functions corresponding to the two steps required for the generation of an analytical evolution scheme.

The function `compute_propagator_matrices()` takes an ODE and a list of `Shape` objects and computes a propagator matrix (Equation 17) for each postsynaptic shape. These matrices can be used to evolve the system from one point to the next. The basic idea here is to populate the matrix $\mathbf{A}$ using the factors of the derivatives (`factors`, computed in lines 12 and 31 of the code in Section 3.1), the factor of the postsynaptic shape used in the ODE for the state variable (`ode_shape_factor`) and the factor of the symbol of the ODE (`ode_sym_factor`). For the equation

$$\frac{d}{dt}V = \frac{1}{\tau} \cdot V + \frac{1}{C_1} \cdot I_1 + \frac{1}{C_2} \cdot I_2$$

`ode_sym_factor` would thus be $\frac{1}{\tau}$. It is calculated using the following line of code:

```
1  ode_sym_factor = diff(ode_def, ode_symbol)
```

`ode_shape_factor` would be $\frac{1}{C_1}$ for postsynaptic shape I_1 in the example equation and $\frac{1}{C_2}$ for I_2. As these factors and other parameters depend on the postsynaptic shape, we run the following code in a loop (omitted for better readability), each iteration assigning the current `Shape` object to the variable `shape`:

```
2  ode_shape_factor = diff(ode_def, shape.symbol)
3
4  if shape.order == 1:
5    A = Matrix([
6      [shape.factors[0], 0],
7      [ode_shape_factor, ode_sym_factor]])
8  elif shape.order == 2:
9    pq = -shape.factors[1] / 2 +
   ↪   sqrt(shape.factors[1]**2 / 4 +
   ↪   shape._factors[0])
10   A = Matrix([
11     [shape.factors[1] + pq, 0, 0 ],
12     [1, -pq, 0 ],
13     [0, shape_factor, ode_sym_factor]])
14 else:
15   order = shape.order
16   A = zeros(order + 1)
17   A[order, order] = ode_sym_factor
18   A[order, order - 1] = shape_factor
19   for j in range(0, order):
20     A[0, j] = shape.factors[order - j - 1]
21   for i in range(1, order):
22     A[i, i - 1] = 1
```

Line 2 computes the `ode_shape_factor` for the current postsynaptic shape. In order to make the calculation of the solution more efficient (i.e. using fewer arithmetic operations on a computer), `compute_propagator_matrices()` creates a lower triangular matrix for postsynaptic shapes of order 1 and 2 (lines 5-7 and 9-13, respectively) as explained in Equation 14 and a generic matrix for all higher orders according to Equation 13 (lines 15-22). The variable `pq` in line 9 corresponds to Equation 15.

The propagator matrix for each postsynaptic shape can now be computed by taking the matrix exponential of the matrix $\mathbf{A}$ multiplied by the update step size h:.

```
propagator_matrices.append(exp(A * h))
```

The second function of the `Propagator` class, `compute_propagation_step()`, takes the list of propagator matrices and postsynaptic shapes and computes a calculation specification that can be executed to actually perform the system update. As this function merely runs a loop over all propagator matrices and generates the update instructions as a list of strings, the code is omitted here.

3.3 Finding an appropriate numerical solver

In case the differential equation describing the dynamics of a state variable was not found to be a linear constant coefficient ODE, the system must be evolved using a numerical stepping scheme as explained in Section 2. Instead of a full calculation specification, as produced for the analytical solution in Section 3.2, the `numeric` module of the toolbox just passes the specification of ODEs from the input and the `Shape` objects created by the algorithm in Section 3.1 on to the stiffness tester, which is implemented in the `stiffness` module.

The stiffness tester uses the standard Python modules SymPy and NumPy for symbolic and numeric calculations. For evolving the ODEs during the test procedure, it currently uses PyGSL, a Python wrapper around the GNU Scientific Library (GSL; Gough, 2009). This library was chosen over more pythonic alternatives such as SciPy due to its more comprehensive selection of ODE solvers.

The stiffness tester executes the algorithm described in Section 2.2 and gives a recommendation as to whether the use of an explicit or an implicit evolution scheme is appropriate. The steps performed by the algorithm are shown in Figure 2. The choice of the factor 6 for comparing average step sizes of the explicit and the implicit schemes is motivated in Section 3.3.1. For the evolution of the system of ODEs, the equations receive representative spike trains drawn from a Poisson distribution with a rate of $\nu = 0.1\,\mathrm{s}^{-1}$ and inter-spike intervals distributed around $\frac{1}{\nu}$ (Connors and Gutnick, 1990).

3.3.1 Comparison of average step sizes

When comparing average step sizes of the implicit and explicit method applied to a certain set of ODEs, we assume that the set of ODEs is stiff when the average step size of the implicit method is considerably larger than the average step size of the explicit method, see Section 2.2, i.e. when $s_{\text{implicit}} > \beta \cdot s_{\text{explicit}}$ for some β.

To determine an appropriate factor β, we developed a testing strategy using a well known example of a set of stiff ODEs: with $a = -100$ and initial values $y_1(0) = y_2(0) = 1$,

$$\begin{aligned}\frac{dy_1}{dt} &= ay_1 \\ \frac{dy_2}{dt} &= -2y_2 + y_1\end{aligned} \tag{20}$$

is a typical stiff ODE system (example taken from Dahmen and Reusken, 2005). The solution $y_1(t) = e^{-100t}$ decays very quickly, whereas the solution $y_2(t) = -\frac{1}{98}e^{-100t} + \frac{99}{98}e^{-2t}$ decreases a lot more slowly, which causes the stiffness of this system.

y_1 is already reduced by four decimal places at $t = 0.1$ and y_1 is practically negligible for even larger t. Nevertheless, it plays a major role in the calculation of y_2 when using an explicit integration method. Using a simple explicit Euler method and a resolution h for the approximation $\tilde{y}_1$ of y_1, we have the following recursive specification:

$$\tilde{y}_1(t+h) = \tilde{y}_1(t) - 100h\tilde{y}_1(t) = (1-100h)\tilde{y}_1(t).$$

For $h = \frac{1}{200}$ and $t = \frac{1}{10}$ we get

$$\tilde{y}_1(1/10) = 2^{-20} < 10^{-6}.$$

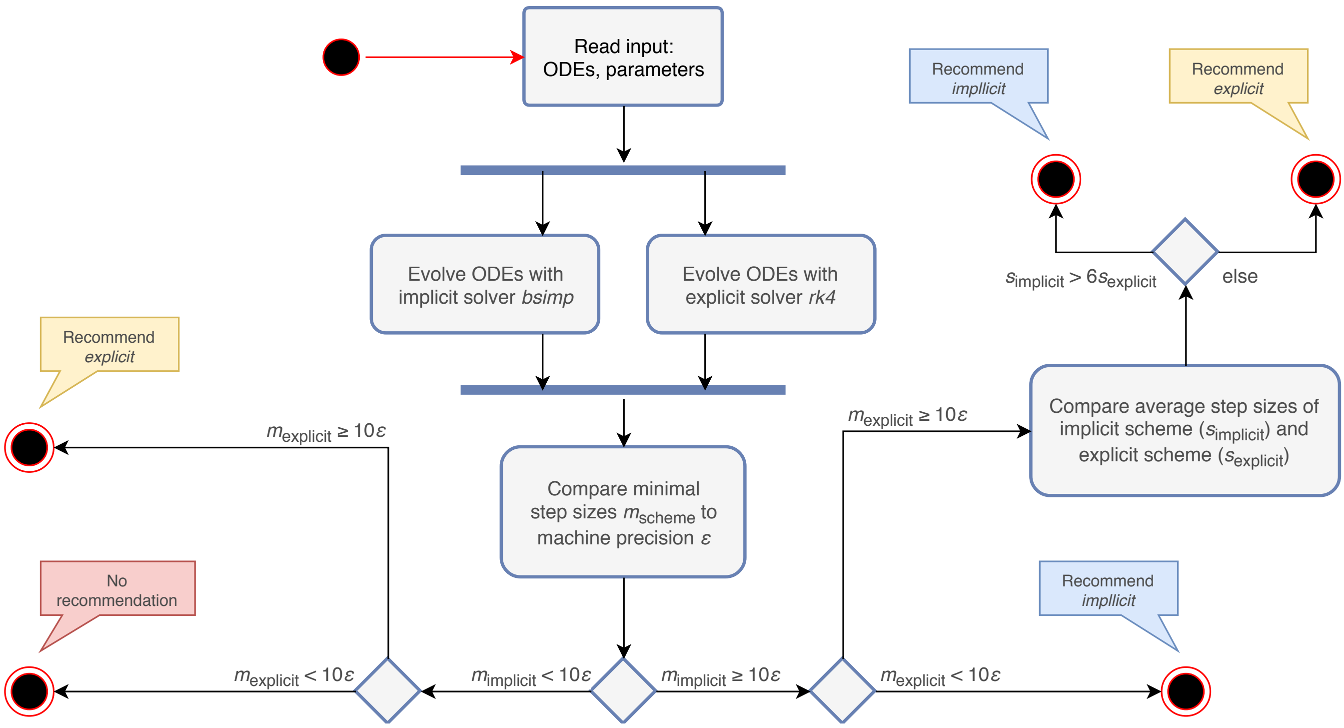

Figure 2. Activity diagram summarizing the steps taken to recommend an appropriate numerical stepping scheme. The input to the algorithm are the ODEs and their parameters. After evolving the system of ODEs in parallel with an implicit and an explicit solver, it compares the minimal step sizes (m_{scheme}) of each scheme with the machine precision (ε). Depending on the outcome of the comparison, it recommends an appropriate stepping scheme (explicit or implicit) or compares the average step sizes (s_{scheme}) of the tested schemes. In the case that both the step size of the explicit and implicit solver are close to ε, the algorithm does not give a recommendation, but terminates with a warning instead.

For computational efficiency, we would like to choose a larger step size for y_2 since the solution decays a lot slower than y_1. If we therefore choose $h = \frac{1}{2}$ to integrate y_2, we get

$$\tilde{y}_1(t+h) = -49\tilde{y}_1(t),$$

causing an explosive growth in the course of the calculations.

A stiff set of ODEs will always result in the average step size of an implicit method exceeding by far the average step size of a comparable explicit method. Hence the runtime of the implicit method should be less than the explicit method's runtime. However, runtime is not solely affected by the grade of stiffness, so the stiffness of a given set of ODEs is evaluated more accurately by comparing average step sizes.

To isolate stiffness from other factors, we chose Equation 20 for its simplicity. This problem is clearly stiff, as described above, and the grade of stiffness relates directly to the size of the factor a. Therefore it can be used as a controlled stiff problem where other effects coming from the complexity of the system do not play a role.

We measure the runtimes of the implicit and the explicit methods (using the corresponding GSL-solvers) for five runs over 20 milliseconds each, whilst systematically varying the stiffness controlling parameters a and the resolution h. The quotient of the average implicit and explicit runtimes is shown in Figure 3.

For each measurement series, we can determine a^*, the value of a for which the runtimes of the explicit and the implicit evolution scheme are the same. We then calculate the ratio of the step sizes employed by the implicit and explicit schemes at a^*:

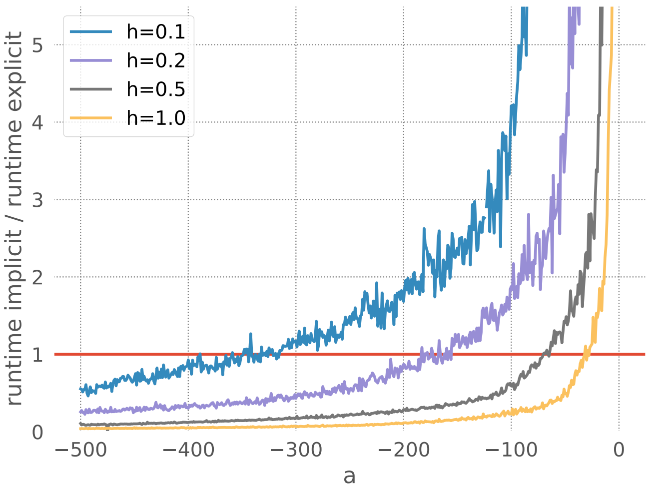

Figure 3. Comparison of implicit and explicit methods for a stiff ODE. Ratio of runtimes for the implicit and explicit method as a function of the factor a in Equation 20, for varying resolutions h and a desired accuracy of 10^{-3}. Curves averaged over 5 runs of 20 ms each. The red bar indicates when the explicit and implicit methods require the same amount of time to evolve the ODE system. Where a curve is below the red bar, the implicit method is faster than the corresponding explicit method.

$r^* = \frac{s_{\text{implicit}}(a^*)}{s_{\text{explicit}}(a^*)}$. Because in this problem the runtime, stiffness and step size are solely influenced by the factor a, we can consider r to be the borderline factor, i.e. problems with $s_{\text{implicit}} > r^* \cdot s_{\text{explicit}}$ are sufficiently stiff to make the implicit method faster.

For all the curves in Figure 3, we determine a value for r^* between 6 and 7. As some input scenarios may result in a somewhat stiffer system than that brought about by the representative spike train chosen in the stiffness tester, we choose $\beta = 6$ conservatively on the low side of the range of r^*, to ensure that the implicit scheme is used in all stiff cases.

3.4 Example

The use of the toolbox as a Python module is explained in detail in the `README.md` file of the git repository at `http://github.com/nest/ode-toolbox`. Here, we demonstrate the use of the analysis toolbox by executing the script file `ode_analyzer.py` in a stand-alone fashion for generating a solver specification for a conductance-based integrate-and-fire neuron with alpha-shaped postsynaptic conductances. The script expects the name of a JSON file as its only command line argument:

```
python ode_analyzer.py iaf_cond_alpha.json
```

The file `iaf_cond_alpha.json` is shown in Listing 1. It contains the specification of one differential equation for the membrane potential `V_m` in the `odes` section in lines 3-7. This section is a list and can potentially contain multiple ODEs. The `shapes` section defines two postsynaptic shapes, one of which is specified as a function of time (`g_in`, lines 10-14), the other as an ODE with initial conditions (`g_ex`, lines 15-20). The parameters and their default values are given in the `parameters` dictionary in lines 22-33. This dictionary maps default values to parameter names and has to contain an entry for each free variable occurring in the equations given in the `odes` or `shapes` sections.

Depending on the complexity of the ODEs and postsynaptic shapes contained in the input, the analysis may take some time. During its execution, the analysis tool prints diagnostic messages about the current processing steps. If all steps succeed, it writes the result again to a JSON file, which can be read by the next tool in the model generation pipeline to create the a complete model implementation.

For the input shown in Listing 1, the analysis toolbox produces the following output:

```
{
  "solver": "numeric-explicit"
  "shape_ode_definitions": [
    "-1/tau_syn_in**2 * g_in + -2/tau_syn_in *
    ↪  g_in__d",
    "-1/tau_syn_ex**2 * g_ex + -2/tau_syn_ex *
    ↪  g_ex__d"
  ],
  "shape_state_variables": [
    "g_in__d",
    "g_in",
    "g_ex__d",
    "g_ex"
  ],
  "shape_initial_values": [
    "0",
    "e/tau_syn_in",
    "0",
    "e/tau_syn_ex"
```

```
{
  "odes": [
    {
      "symbol": "V_m",
      "definition": "(-(g_L*(V_m-E_L))-(g_ex*(V_m-E_ex))-(g_in*(V_m-E_in))+I_stim+I_e)/C_m",
      "initial_values": ["E_L"]
    }
  ],
  "shapes": [
    {
      "type": "function",
      "symbol": "g_in",
      "definition": "(e/tau_syn_in)*t*exp((-1)/tau_syn_in*t)"
    },
    {
      "type": "ode",
      "symbol": "g_ex",
      "definition": "(-1)/(tau_syn_ex)**(2)*g_ex+(-2)/tau_syn_ex*g_ex'",
      "initial_values": ["0",  "e / tau_syn_ex"]
    }
  ],
  "parameters": {
    "V_th": -55.0,
    "g_L": 16.6667,
    "C_m": 250.0,
    "E_ex": 0,
    "E_in": -85.0,
    "E_L": -70.0,
    "tau_syn_ex": 0.2,
    "tau_syn_in": 2.0,
    "I_e": 0,
    "I_stim": 0
  }
}
```

Listing 1. Example JSON file as input to the analysis toolbox. The file contains three entries: `odes` describing the ODEs of the system, `shapes` containing the postsynaptic shapes used in the ODEs and `parameters` specifying the parameters and default values for the differential equations in the `shapes` and `odes` sections.

```
  ],
}
```

The meaning of the fields is explained in detail in the `README.md` of the toolbox.

4 RESULTS

To evaluate the proposed framework for the semantic analysis of a system of ODEs and assessment of its stiffness we have chosen two approaches. One was to apply the stiffness tester to the neuron models currently implemented in the NEST Modeling Language (NESTML; Plotnikov et al., 2016), the other was to compare runtimes of explicit and implicit evolution schemes applied to two commonly used simplified versions of the Hodgkin-Huxley model.

The stiffness tester was integrated and successfully used in the tooling for NESTML, a domain specific language for the definition of neuron models for the neuronal simulator NEST (Gewaltig and Diesmann, 2007; Kunkel et al., 2017). NESTML is built using MontiCore (e.g. Krahn, 2010; Grönniger et al., 2008). MontiCore is a language workbench (Erdweg et al., 2013) that enables an agile and incremental implementation of lightweight DSLs including the symbol table functionality (Mir Seyed Nazari, 2017), code generation facilities (e.g. Schindler, 2012; Rumpe, 2017) and support for editors in Eclipse IDE (e.g. Völkel, 2011; Krahn et al., 2007). NEST's focus is on the simulation of the dynamics of large networks of spiking neurons (e.g. Potjans and Diesmann, 2012; van Albada et al., 2015; Kunkel et al., 2010). Neuron models in NEST are usually rather simple point neurons or models with a few electrical compartments instead of rich compartmental neurons built from morphologically detailed reconstructions. The simulator is capable of running on a large range of computer architectures ranging from laptops over standard workstations to the largest supercomputers available today (Kunkel et al., 2014).

Within NESTML, the analysis toolbox developed in Sections 2 and 3 is used for the numerical analysis of neuron models defined as systems of ODEs and provides either the implementation of an efficient and accurate analytical integration scheme or recommends a good numerical solver. Therefore it allows the simulation of a large variety of biological neuron models in NEST.

As a simple yet meaningful validation of the stability checks introduced in Section 2.2, we applied the stiffness tester to all neuron models currently implemented in NESTML (see `https://github.com/nest/nestml/tree/master/models`). The result of this evaluation is that with default parametrization, the systems of ODEs of all neuron models are non-stiff and can thus be safely integrated using an explicit numerical integration scheme without any detrimental effects on efficiency and accuracy. This is a reassuring finding, as it indicates that previous studies using these neuron models are unlikely to contain distorted results due to numeric instabilities in the integration, for a counter-example see Pauli et al. (2018).

However, when the default parametrization is slightly altered, the stiffness test finds that some systems of ODEs are now evaluated as being stiff, which suggests that the choice of an implicit evolution scheme would be more advisable than the default choice. Figure 4 summarizes these observations for a selection of six commonly used neuron models and shows how a systematic change of one parameter in these models results in an evaluation as stiff or non-stiff.

As a second test, we apply the stiffness tester to the Fitzhugh-Nagumo and Morris-Lecar models (FitzHugh, 1961; Nagumo et al., 1962; Morris and Lecar, 1981), non-linear oscillators that include the generation of an action potential as part of the dynamics, rather than applying an artificial threshold as many point neuron models do. To assess the comparative performance of the two approaches, we vary both the stiffness controlling parameter of the model equations and the resolution h, as a parameter of the stiffness tester (`stiffness.py`; see Section 3). For small values of h, the explicit approach is expected to exhibit a better performance, as it is relatively easy to find the solution, and the explicit approach is computationally less expensive. As h increases, it becomes harder to determine the

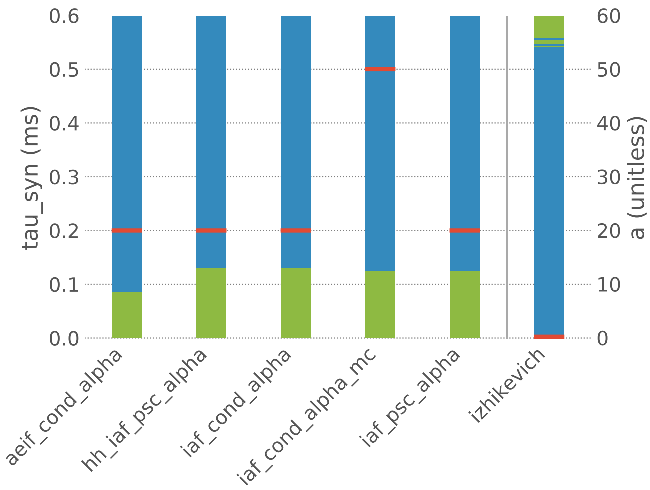

Figure 4. Results of the stiffness test for six neuron models from NEST. Red bars indicate the default value of the selected parameter in NEST, blue indicates the value range in which the system of ODEs evaluates as non-stiff, green indicates the range in which it evaluates as stiff. `aeif_cond_alpha` is a conductance-based adaptive exponential integrate-and-fire model with alpha-shaped postsynaptic conductances, `hh_psc_alpha` a Hodgkin-Huxley type model with alpha-shaped postsynaptic currents, `iaf_cond_alpha` a conductance-based integrate-and-fire neuron with alpha-shaped postsynaptic conductances, `iaf_cond_alpha_mc` a conductance-based integrate-and-fire neuron with alpha-shaped postsynaptic conductances and multiple compartments, `iaf_psc_alpha` a current-based integrate-and-fire neuron with alpha-shaped postsynaptic currents and `izhikevich` the model dynamics proposed by Izhikevich (2003). The test was applied to the ODE systems for varying values of the parameter `tau_syn` of the first five models and for the parameter `a` of the last model.

correct solution, so that the more expensive, but more reliable, implicit method becomes advantageous. Alternatively, a systematic variation of the desired accuracy would yield the same insight (data not shown).

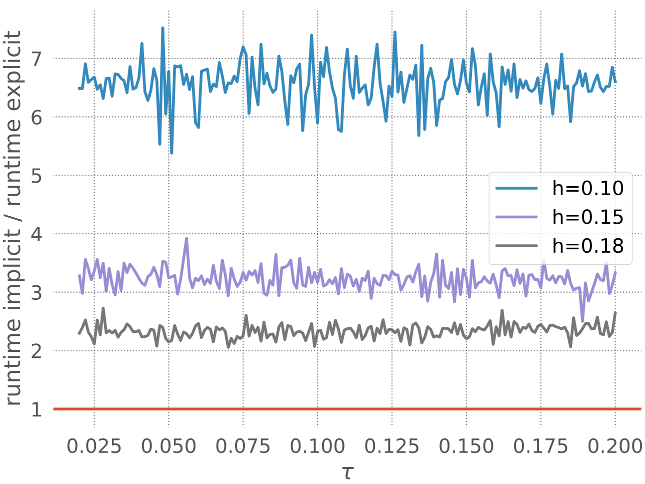

Figure 5. Application of the stiffness tester to the Fitzhugh-Nagumo model. Ratio of runtimes for the implicit and explicit method as a function of the factor τ in Equation 21, for varying resolution h and a desired accuracy of 10^{-5}. Curves averaged over 5 runs of 20 ms each. Red bar as in Figure 3.

Figure 5 demonstrates a comparison of the implicit and explicit methods applied to the FitzHugh-Nagumo model. The model comprises two variables, one for the membrane potential V and a recovery variable W. The dynamics are given by:

$$\begin{aligned} V' &= V - \frac{1}{3}V^3 - W + 0.25 \\ W' &= \tau(V + 0.7 - 0.8W). \end{aligned} \tag{21}$$

The figure shows the quotient of the time that the corresponding GSL-solvers for the explicit and implicit methods spent on integrating the ODE system for 20 milliseconds with a desired accuracy of 10^{-5}. For all resolutions shown in Figure 5, the explicit scheme is faster, and is also the approach recommended by our toolbox. As the resolution becomes coarser (increased values of h), the curves shift down towards the point at which the implicit method would be faster. For $h > 0.185$, our toolbox recommends an implicit approach, and indeed in such cases the explicit scheme, as implemented by the GSL, exits with an error. This is due to the variable V becoming so large in one of the internal steps that it can no longer be represented by a `double`. For a higher required accuracy of 10^{-10}, all curves shift to below the red line (data not shown), and the toolbox recommends an implicit solver for all tested resolutions.

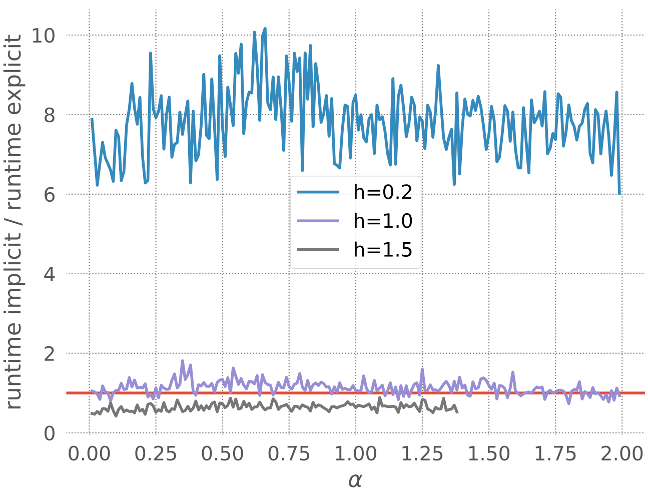

Figure 6. Application of the stiffness tester to the Morris-Lecar model. Ratio of runtimes for the implicit and explicit method as a function of the factor ε in Equation 22, for varying resolution h and a desired accuracy of 10^{-5}. Curves averaged over 5 runs of 20 ms each. Red bar as in Figure 3.

We apply the same approach to the Morris-Lecar model (Morris and Lecar, 1981):

$$
\begin{aligned}
V' &= I + 2W(-0.7 - V) + 0.5(-0.5 - V) \\
&\quad + 1.1m(V)(1 - V) \\
W' &= \alpha\lambda(V)(w(V) - W) \qquad (22) \\
m(V) &= \frac{1}{2}\left(1 + \tanh\left(\frac{V + 0.01}{0.15}\right)\right) \\
w(V) &= \frac{1}{2}\left(1 + \tanh\left(\frac{V + 0.12}{0.3}\right)\right) \\
\lambda(V) &= \cosh\left(\frac{V - 0.22}{2 \cdot 0.3}\right),
\end{aligned}
$$

where I represents injected current. Figure 6 shows that for a resolution of $h = 0.2$, the explicit solver is faster, but for larger values of h the implicit solver becomes more efficient. Accordingly, our toolbox recommends explicit for the former and implicit for the latter. Note also that the explicit solver exits with an overflow error for $h = 1.5$ with values of α above 1.4. Again, the toolbox catches this risk of numerical instability and recommends the implicit scheme.

These results show that the toolbox can correctly assess where it is safe and efficient to use an explicit scheme, and where an implicit scheme would be appropriate, either for reasons of speed or for numerical stability.

5 RELATED WORK

In this section we compare our proposed framework for choosing evolution schemes for systems of ODEs in neural models with the corresponding approaches implemented in the simulators Brian (Goodman and Brette, 2009; Stimberg et al., 2014) and NEURON (Hines and Carnevale, 2000; Carnevale and Hines, 2006). These two simulators were chosen as they are in wide-spread use in the community. We will further consider the application of software for symbolic computation (for *exact* mathematical calculations) or scientific computing (for numerical calculations) to our setting in language modelling for neural simulators.

5.1 Brian

Similar to our framework, the implementation of the Brian simulator also makes a distinction between systems of ODEs that can be solved analytically and systems that can only be solved efficiently in a numeric manner. In addition to simple integrate-and-fire neurons, Brian also supports multi-compartmental neurons and neurons described by stochastic ODEs. As these types of models cannot be currently analyzed by our ODE analysis toolbox, we will not take them into account here. Instead we focus on single-compartmental deterministic neuron models as we can only draw a meaningful comparison for this group of neuron models.

In Brian, neuron dynamics can be described by a system consisting of ODEs and time-dependent functions. They are either classified as *linear*, meaning they can be solved analytically, or as *non-linear*, meaning they cannot be solved analytically and must be solved numerically using the *forward Euler method* (if not stated otherwise by the author of the model). In theory, linear constant coefficient ODEs can be solved analytically by Brian. However, if the dynamics of a neuron are described using a non-constant function of time rather than an ODE defining this function they are always solved numerically. This could be improved by using our proposed framework, which allows an analytical solver to be generated even for a system consisting of time-dependent functions that satisfy a linear homogeneous ODE and feed into a linear constant coefficient ODE. Our framework thus allows an analytical evolution for a larger class of neuron dynamics. In particular, our framework seems to be more robust with respect to the use of several different postsynaptic shapes, as they are treated seperately in contrast to Brian's approach, where the system is analyzed by SymPy as a whole.

All systems of ODEs in Brian that are not evolved by an analytical evolution scheme are by default evolved using the simple Euler method. To circumvent this, it is possible to choose a numerical evolution scheme from a list of other methods. This approach works well for users who are aware of the numerical consequences of their choice of solver but can be problematic for scientists who lack the ability to weigh up the advantages and disadvantages of different numerical evolution schemes for their particular system of ODEs. Moreover, as demonstrated in Figure 3, the choice of an appropriate evolution scheme might depend on the exact parameters for the ODEs and thus not be obvious even for an advanced user.

5.2 NMODL

NMODL is the model specification language of the NEURON simulator. NEURON was created for describing large multi-compartmental neuron models and thus also supports a wider range of models than our proposed framework currently does. We will again only contrast those types of models for which a comparison is meaningful.

For linear systems of ODEs, NMODL chooses an evolution method that propagates the system by evolving each variable under the assumption that all other variables are constant during one time step. In many cases this approach approximates the true solution well, but it is still less accurate than an actual analytical solution. For all other systems of ODEs, i.e. all non-linear ODEs, an implicit method is chosen, regardless of the exact properties of the equations to guarantee an evolution of stiff ODEs without causing numeric instabilities. This is a robust solution but may lead to excessively large simulation run times in cases where the choice of an explicit evolution scheme for non-stiff ODE systems would be sufficient.

5.3 Software for symbolic computation and scientific computing

There are a number of high quality and widely used applications available for symbolic computation, most notably *Wolfram Mathematica* (Benker, 2016), *Modelica* (Tiller, 2001) and *Maple* (Westermann, 2010). All three provide frameworks for solving ordinary differential equations both symbolically and numerically. Here, we will briefly describe their capabilities and limitations for both symbolic and numeric integration of systems of ODEs.

5.3.1 Symbolic integrators

At first appearance the integration schemes provided by the programming languages (or in the case of Modelica, modelling language) seem appropriate for the task addressed in our study. As discussed in Section 1, the ordinary differential equations used to define neuron models and to describe their dynamical behaviour are typically linear (though not homogeneous and not linear with a constant coefficient) and can in several cases be solved analytically by any of the programs above. However, for the specific requirements related to neural simulations, there are several reasons why they are not entirely well suited.

Firstly, neurons receive input that generally changes in every integration step due to the arrival of incoming spikes, thus changing the differential equations to be solved. Although each of these differential equations can be integrated easily using, e.g. Wolfram Mathematica, none of these frameworks provide a general, exact solution for each integration step, that takes a run-time generated varying input into account. The next two points are related to the size of neural systems commonly investigated. Spiking neuronal network models often contain of the order of 10^3 - 10^5 neurons, and sometimes substantially more (Kunkel et al., 2014). Calling external software for symbolic computation of ordinary differential equations during run time for each neuron is therefore often too costly. Moreover, for large models, the simulation software is likely to be deployed on a large cluster or supercomputer. The aforementioned applications are typically not installed on such architectures, whereas Python is a standard installation, providing the package SymPy, which is sufficient for symbolic computation in this context.

5.3.2 Numerical integrators

There are a number of approaches to automatically select numeric integrators depending on whether the problem is stiff or non-stiff (Shampine, 1983, 1991; Petzold, 1983). These approaches are typically designed to switch integration schemes during runtime when the problem changes its properties. All of them rely in one way or another on the behaviour of the Jacobian matrix evaluated at the point of integration. Typically, the methods try to approximate the dominant eigenvalue of the Jacobian with a low cost compared to that of the stepping algorithm. However, for a spiking neural network simulation, the determination of the stiffness of the system, and thus the solver, should occur before the simulation starts, as to minimize runtime costs.

Thus the question remains whether it would be possible to carry out these kind of tests during generation of the neuron model. Applying the test to a large number of randomly selected values of the state variables, or carrying out a number of test runs using representative spike trains would allow to work around the fact that the solution up to a given point is not yet known. However, as these tests rely on determining the stiffness through the properties of the Jacobian, they would still not be completely precise. As we have the advantage of effectively no computational constraints during generation of the neuron model, there is thus no advantage by using such a low-cost strategy. In our approach we compute the solution using both explicit and implicit schemes and compare their behavior a posteriori, thus obtaining an accurate assessment of the appropriate solver for a given set of parameters.

In addition, as for symbolic integration, the packages that provide such stiffness testing capability for numeric integration do not provide a framework for handling a run-time determined variable input due to incoming spikes. Thus we conclude that the specific problem addressed by our toolbox lies outside the scope of general purpose symbolic and numeric integration packages.

6 DISCUSSION

We have presented a novel simulator-independent framework for the analysis of systems of ODEs in the context of neuronal modeling and provided a reference implementation for the selection and generation of appropriate integration schemes as open source software.

In this section we will summarize the restrictions of our framework, discuss alternative ideas for the implementation and describe possible future additions.

The framework we propose is currently limited to the analysis of equations for non-stochastic single-compartmental integrate-and-fire neuron models. The reason for this is that the analysis toolbox was developed in the context of the NESTML project, in which we put our main focus on the class of neurons presently available in the NEST simulator. The extension of the framework to other classes of neurons is one of our current research objectives. In particular, this work includes support for systems of stochastic ODEs. The symbolic analysis of neuron ODEs enables generation of the sophisticated C++ neuron implementation that switches between implicit and explicit solvers at run-time of the neurons depending on the runtime performance of the particular solver. This functionality will be integrated in upcoming releases of NESTML.

Another restriction of the framework is that it can only analyze systems of ODEs with postsynaptic shapes that obey a linear homogeneous ODE. This is due to the fact that evolving a system including postsynaptic shapes as functions of time rather than functions defined as ODEs would result in a very long sum of multiple linear combinations of shifts of this function for each incoming spike. Evaluating such a sum would make the evolution of the system containing it computationally very costly. Finding a more efficient solution for this problem is of high priority in our current work.

As noted in Section 2, the calculation of e^{Ah} may become difficult to compute analytically rather than numerically if the matrix A becomes very large. In this case, i.e. when e^{Ah} is computed as a numerical approximation, the integration scheme is, strictly speaking, not analytical. Here it might be sensible to look into other numerical methods, e.g integrating the system of ODEs using a quadrature formula of order 5 and thereby obtaining an accuracy of 10^{-8} despite the use of a numerical scheme.

When comparing implicit and explicit integration schemes, we compare the *average step size* and the *minimal step size* of the respective schemes. An alternative possibility would be to use fixed step sizes instead and compare the results of the explicit and implicit schemes using the results of the implicit scheme as a reference. This could be implemented alongside our current stiffness tester to provide a higher degree of certainty.

As pointed out in Section 4, the stiffness of a system of ODEs depends greatly on its parametrization. Therefore it might be a useful extension to run the stiffness test not only during the generation of the model code, but also when instantiating the model in a simulator, and when model parameters are changed. This would, however, require a call to the analysis toolbox at run time, which might not be easily possible on all machines a particular simulator may run on. For example, in a supercomputer environment, job allocations are usually fixed, and not all libraries required by the toolbox may be available. An alternative solution to the problem could be to run the stiffness test for varying parameters during the generation phase of the model. This way the analysis toolbox could create a lookup table, mapping parameter values to the most appropriate integration scheme.

Another possible extension of the current framework could be to implement implicit and explicit integration schemes for evolving the systems of ODEs during the stiffness analysis, and thereby gain independence of PyGSL, which can be challenging to install. These custom implementations could be tailored to our specific requirements and give us more control over the integration scheme and the exact methodology for adaptive step size control.

The current implementation of the framework only supports fixed thresholds for the detection of spikes and evaluates the spiking criterion on a fixed temporal grid. A part of our current work is to evaluate more realistic scenarios, such as adaptive thresholds or precise detection of spike times in between the grid points. For a general discussion on the topic, see Hanuschkin et al. (2010).

Our presented framework is re-usable independently of NESTML and NEST. The source code is available under the terms of the GNU General Public License version 2 or later on GitHub at `https://github.com/nest/ode-toolbox/` and we hope that the code can serve both as a useful tool for neuroscientists today, and as a basis for a future community effort in developing a simulator-independent system for the analysis of neuronal model equations.

CONFLICT OF INTEREST STATEMENT

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

AUTHOR CONTRIBUTIONS

IB developed the mathematical derivations of the solver selection system and devised the algorithms. The reference implementation was conceived and created by IB and DP. DP integrated the framework into the NESTML system. JME and AM supervised and guided the work. The article was written jointly by all authors.

FUNDING

This work was supported by the JARA-HPC Seed Fund *NESTML - A modeling language for spiking neuron and synapse models for NEST* and the *Initiative and Networking Fund* of the Helmholtz Association and the Hemholtz Portfolio Theme *Simulation and Modeling for the Human Brain*. The current work on NESTML is partly funded by the

European Union's Horizon 2020 research and innovation programme under grant agreement No. 720270.

ACKNOWLEDGMENTS

We gratefully acknowledge the fruitful discussions with the users of NESTML, who provided use cases and guided the work through their critical questions and thoughts. We would especially like to thank Arnold Reusken, Markus Diesmann, Hans Ekkehard Plesser, Guido Trensch, Bernhard Rumpe and Tanguy Fardet for their ongoing support and interest in the NESTML project.

REFERENCES

Benker, H. (2016). *MATHEMATICA kompakt: Mathematische Problemlösungen für Ingenieure, Mathematiker und Naturwissenschaftler* (Springer)

Brette, R. and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiol* 94, 3637–3642. doi:10.1152/jn.00686.2005

Carnevale, N. T. and Hines, M. L. (2006). *The NEURON Book* (New York, NY, USA: Cambridge University Press)

Connors, B. W. and Gutnick, M. J. (1990). Intrinsic firing patterns of diverse neocortical neurons. *Trends in neurosciences* 13, 99–104

Dahmen, W. and Reusken, A. (2005). *Numerik für Naturwissenschaftler* (Springer)

Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., et al. (2013). The state of the art in language workbenches. In *Software Language Engineering*, eds. M. Erwig, R. F. Paige, and E. Van Wyk (Cham: Springer International Publishing), 197–217

FitzHugh, R. (1961). Impulses and physiological states in theoretical models of nerve membrane. *Biophysical J.*

Gewaltig, M.-O. and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia*

Goodman, D. and Brette, R. (2009). The Brian simulator. *Frontiers in Neuroscience*

Gough, B. (2009). *GNU scientific library reference manual* (Network Theory Ltd.)

Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2008). Monticore: a framework for the development of textual domain specific languages. In *Companion of the 30th international conference on Software engineering* (ACM), 925–926

Hanuschkin, A., Kunkel, S., Helias, M., Morrison, A., and Diesmann, M. (2010). A general and efficient method for incorporating precise spike times in globally time-driven simulations. In *Frontiers in Neuroinform.* doi:10.3389/fninf.2010.00113

Hines, M. and Carnevale, N. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation*

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks* 14, 1569–1572

Kandel, E. R., Schwartz, J. H., Jessell, T. M., Siegelbaum, S. A., and Hudspeth, A. (2013). *Principles of Neural Science*. Principles of Neural Science (McGraw-Hill Education), fifth edn.

Krahn, H. (2010). *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. No. 1 in Aachener Informatik-Berichte, Software Engineering (Shaker Verlag)

Krahn, H., Rumpe, B., and Völkel, S. (2007). Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)* (Jyväskylä University, Finland)

Kunkel, S., Diesmann, M., and Morrison, A. (2010). Limits to the development of feed-forward structures in large recurrent neuronal networks. *Frontiers in Computational Neuroscience* 4, 1–15. doi:10.3389/fncom.2010.00160

Kunkel, S., Morrison, A., Weidel, P., Eppler, J. M., Sinha, A., Schenck, W., et al. (2017). NEST 2.12.0

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., et al. (2014).

Spiking network simulation code for petascale computers. *Frontiers in Neuroinformatics* 8

Lambert, J. D. (1992). *Numerical Methods for Ordinary Differential Systems* (Wiley)

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., et al. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science* 3, e103. doi:10.7717/peerj-cs.103

Mir Seyed Nazari, P. (2017). *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29 (Shaker Verlag)

Morris, C. and Lecar, H. (1981). Voltage oscillations in the barnacle giant muscle fiber. *Biophys. J.*

Morrison, A., Straube, S., Plesser, H. P., and Diesmann, M. (2007). Exact Subthreshold Integration with Continuous Spike Times in Discrete-Time Neural Network Simulations. *Neural Computation*

Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). An active pulse transmission line simulating nerve axon. *Proc. IRE.*

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing polychronization: a guide to maximizing the reproducibility of spiking network models. *Frontiers in Neuroinformatics* (in press)

Petzold, L. (1983). Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM Journal on Scientific and Statistical Computing* 4, 136–148. doi:10.1137/0904010

Plotnikov, D., Blundell, I., Ippen, T., Eppler, J. M., Morrison, A., and Rumpe, B. (2016). NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference* (Bonner Köllen Verlag), vol. 254 of *LNI*, 93–108

Potjans, T. and Diesmann, M. (2012). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cerebral Cortex* 24

Rotter, S. and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological Cybernetics* 81, 381–402. doi:10.1007/s004220050570

Rumpe, B. (2017). *Agile Modeling with UML: Code Generation, Testing, Refactoring* (Springer International)

Schindler, M. (2012). *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11 (Shaker Verlag)

Shampine, L. (1983). Type-insensitive ode codes based on extrapolation methods. *SIAM Journal on Scientific and Statistical Computing* 4, 635–644. doi:10.1137/0904044

Shampine, L. (1991). Diagnosing stiffness for Runge–Kutta methods. *SIAM Journal on Scientific and Statistical Computing* 12, 260–272. doi:10.1137/0912015

Stimberg, M., Goodman, Benichoux, V., and Brette, R. (2014). Equation-oriented specification of neural models for simulations. *Frontiers in Neuroinformatics*

Strehmel, K. and Weiner, R. (1995). *Numerik gewöhnlicher Differentialgleichungen* (B.G. Teubner)

Tiller, M. (2001). *Introduction to physical modeling with Modelica* (Kluwer Academic Publishers)

van Albada, S. J., Helias, M., and Diesmann, M. (2015). Scalability of asynchronous networks is limited by one-to-one mapping between effective connectivity and correlations. *PLOS Computational Biology* 11, 1–37. doi:10.1371/journal.pcbi.1004490

Völkel, S. (2011). *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9 (Shaker Verlag)

Walter, W. (2000). *Gewöhnliche Differentialgleichungen* (Springer)

Westermann, T. (2010). *Mathematische Probleme lösen mit Maple* (Springer)