Improving simulation methods in the SpineCreator tool

George Powell 2014 SURE placement, supervised by James Marshall and Alex Cope

1 Abstract

The SpineCreator software is a cross-platform tool for creating and running neural simulations, it includes a powerful graphical tool for designing large-scale parallel dynamical systems without requiring programming. Simulations and networks are saved in a standardised and open XML format called SpineML. The temporal behaviour of components within SpineML systems are specified with systems of Ordinary Differential Equations; simulating SpineML systems therefore requires accurately and efficiently solving ODEs forward in time in a general manner. Until now SpineCreator exclusively used the Forward Euler method, which although extremely simple has a number of severe problems with accuracy and stability. This project investigates alternatives to the Forward Euler method for solving systems of ODEs, with a specific focus on the ODEs used in common biological neural models. A series of methods and improvements have been implemented and tested in a C++ program isolated from SpineML, and a number of general recommendations for further improvements are given.

2 Initial Value Problems

The dynamical systems that SpineML simulates are **Initial Value Problems**. This means that the state of the system at any point is represented by a finite number of real number variables that start at given values and change over time at a rate depending only on the current state of the system. The initial value problems are therefore represented by a set of variable names, along with their respective initial values and time-derivative functions (these are functions that take the state of all the variables as input and return the rate at which a specific one is changing at that point).

Numerical simulations work by starting the variables at their initial values and using the time-derivative functions to estimate the state of the system as time goes on, usually progressing the system forward in time again and again by a chosen **time-step** *dt*. There is a general trade-off between simulation accuracy and computational efficiency; a smaller value for *dt* means a more accurate but more computationally intensive simulation.

Analytic/Algebraic solutions inspect the form of the time-derivative functions and use mathematical deduction to derive closed-form functions that can give the value of the variables at any point in time without simulating it iteratively to get there. These solutions are the ideal as they are very accurate and efficient, but most interesting or complex systems of ODEs will not have solutions of this type.

3 FORWARD EULER [1]

3.1 DESCRIPTION

Forward Euler is the simplest and most obvious numerical simulation method. The method can be specified by the following time-step equation:

$$v_{n+1} = v_n + dt * \frac{dv}{dt}(v_n)$$

This means that each time we step the system forward by an amount *dt*, we add an amount to each variable proportional to its time-derivative at that point. This makes sense given the definition of a time-derivative, and will result in an estimation of the true function:

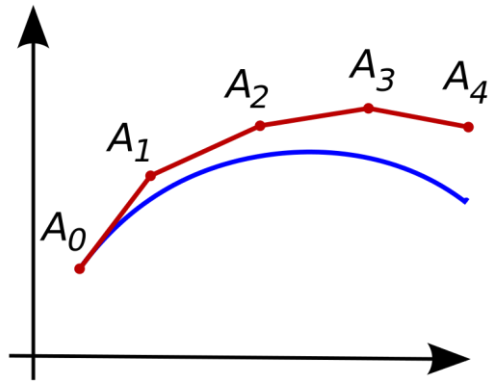

Fig. 1: Illustration of the Euler method. The unknown curve is in blue, and its forward Euler approximation is in red.

Forward Euler can produce arbitrarily accurate solutions by using smaller and smaller values for *dt*, however as a general method for solving systems of ODEs it has some severe problems. These problems have specific undesirable effects within biological simulations of neurons.

3.2 PROBLEM 1: UNDERSTEER

When using Forward Euler to estimate a curve, the result will **always understeer** around bends in that curve, as seen in **Fig. 1.** This is due to the forward Euler method assuming the rate of change of a variable stays constant within any particular time-step. This results in all curves being underestimated in sharpness.

3.2.1 In the context of neural simulations

In neuron models where the membrane voltage increases exponentially until reaching a threshold, using forward Euler will result in the voltage-gain being slower than in an exact solution. This may result in unrealistically slow spiking rates and slower spike response times.

3.3 PROBLEM 2: INSTABILITY

There are a number of stable systems for which forward Euler will produce a wild and unstable solution, particularly when using larger values for *dt*.

Using forward Euler to solve the differential equations for **Sine** and **Cosine**, for example, does not result in the expected stable oscillation between -1.0 and 1.0, instead the oscillation increases in

magnitude exponentially over time, returning values in the tens of thousands after a short period of time. This instability exists for any choice of value for *dt*, however is less severe for small values.

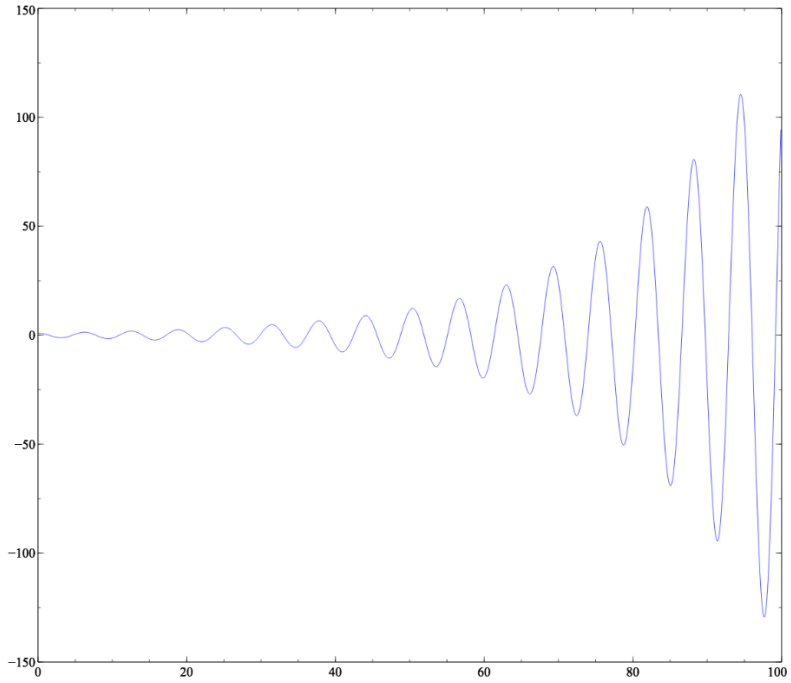

Fig. 2: The result of using Forward Euler to estimate a cosine wave. The result is unstable and increases exponentially in size.

3.3.1 In the context of neural simulations

A simple system which forward Euler fails to simulate stably is **exponential decay.** This formula is commonly found within the biological simulation of synapses; a neuron spike will often be sent out to other neurons as an exponentially decaying voltage.

Using forward Euler to simulate an exponential decay can result in an oscillating system which never settles to zero, as would be expected of an accurate solution. Choosing a small value for *dt* will solve the problem in this case, but choosing a smaller *dt* will have an impact on computational efficiency and will not solve the instability issue for all systems.

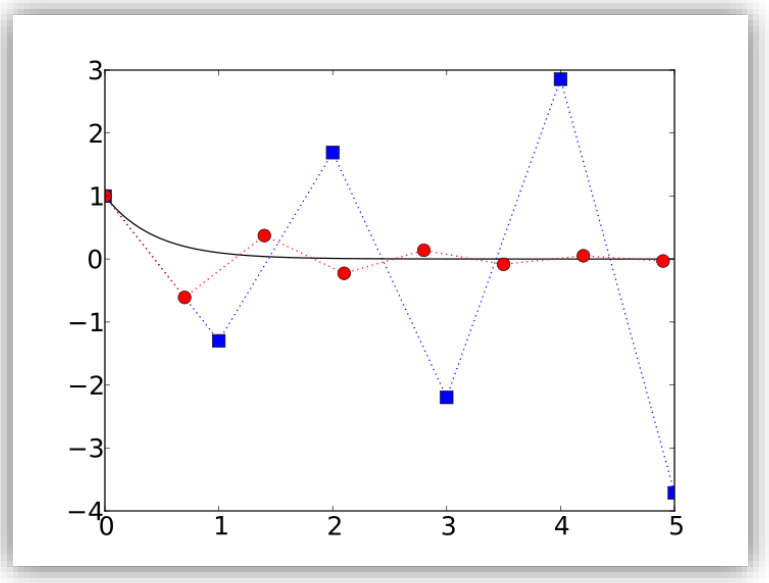

Fig. 3: In black is the exact solution, in red is a stable but oscillating forward Euler solution, and in blue is an unstable oscillating forward Euler solution with a larger time-step.

4 Improved Fixed Time-Step Methods

Forward Euler is called a fixed time-step method because as the simulation progresses, the size of *dt* doesn't change dynamically. There are alternative fixed time-step methods which solve some of the instability issues that forward Euler suffers from.

4.1 Modified Euler [1]

The simplest improvement to Forward Euler is called **modified Euler**. Like forward Euler, modified Euler estimates the value for a variable after a time *dt* by using the time-derivative at the beginning of the time-step, but then uses this new value to calculate an estimate of the time-derivative at the *end* of the time-step. The final estimate for the value of the variable is the average of using the time-derivative at the beginning and end of the time-step.

Modified Euler produces a stable solution for both the **sin-cosine** system and for **exponential decay**. The drawback is that modified Euler requires two evaluations of the time-derivative function per time-step. Runge-Kutta methods are another improvement to modified Euler, and require more time-derivative evaluations per time-step.

4.2 Runge-Kutta 4th order Method [1]

The Runge-Kutta 4th order method uses 4 time-derivative evaluations per time-step, but produces accurate and stable solutions for much larger time steps than modified Euler is able to. The 4th order Runge-Kutta method is the most commonly used Runge-Kutta method and is generally a good option for a fixed time-step method.

4.2.1 Note on implementation

It is important to note that when applying the modified Euler or Runge-Kutta methods to systems of Ordinary Differential Equations with *more than one variable*, all the variables (i.e. the whole system)

should be moved forward together rather than each variable individually. For example in modified Euler all the variables should be moved forward using the time-derivatives at the start of the time step to calculate the time derivatives at the end of the time-step. In the current implementation of these methods within SpineCreator the variables are moved forward individually and the solutions are not as accurate as they could be with the same number function evaluations.

5 ADAPTIVE TIME-STEP METHODS

It is a common case with dynamical systems that at one period a very small time-step is needed, and at another a much larger time step is sufficient. This is the motivation behind numerical integration methods that adapt their step-size dynamically. At every time step in these methods, an estimate is made for the error introduced within that time-step, and the step-size (*dt*) is modified based on this estimated error. In systems that have chaotic, non-linear periods followed by periods of calm or linear behaviour, an algorithm that dynamically uses a much larger step-size during the linear periods could result in very accurate solutions with an order of magnitude less computations than a fixed time-step method would use.

5.1 DORMAND-PRINCE METHOD [1]

The Dormand-Prince method is a Runge-Kutta method that calculates both a 4^{th} and 5^{th} order solution, the difference between which can be used as an estimation of the local error introduced during that time-step. This estimation of error can then be used to increase or decrease the step-size, depending on if it is smaller or larger than we require. This method also has the advantage of being configurable using a more meaningful value for *error tolerance* rather than a specific value for *dt*.

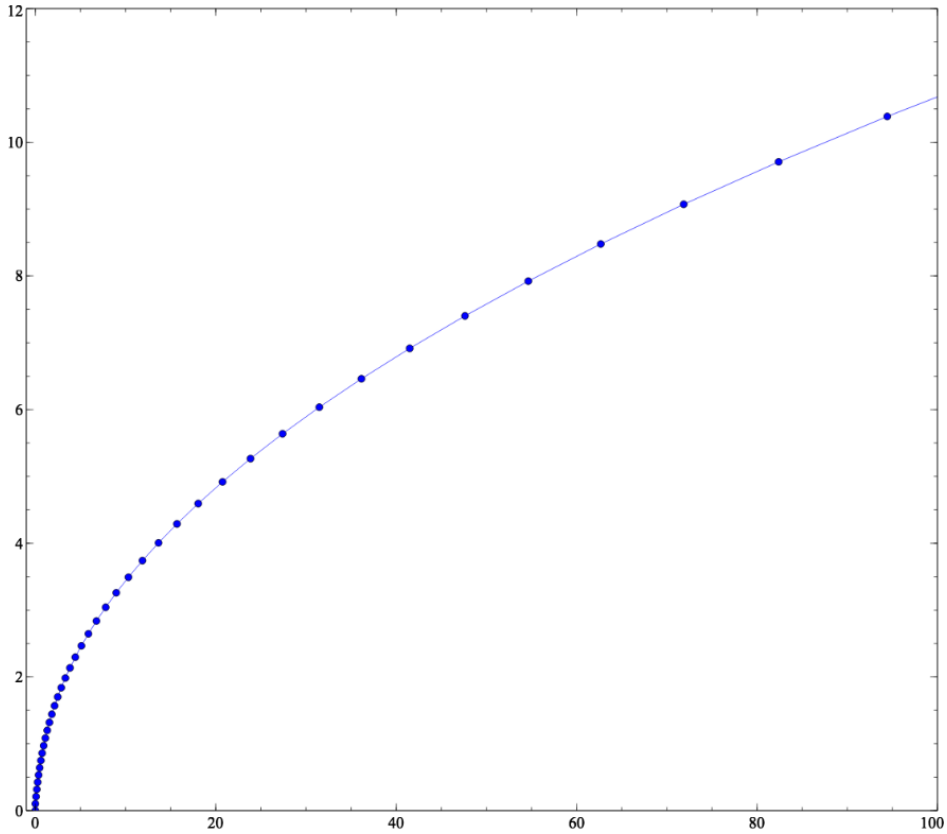

Fig. 4: The Dormand Prince method being used to solve a simple ODE with an adaptive time-step. The dots are the points at which the value of the function is estimated. It demonstrates that as the system moves forward in time, a larger step-size is used (dots are further apart on the horizontal time axis) because the method has calculated it will be accurate enough. This produces a very accurate solution while using a small fraction of the calculations that a comparable fixed time-step method would have used.

5.1.1 Dormand-Prince method for biological neuron simulations

The Dormand-Prince method can be used effectively to simulate biological neurons with two minor modifications:

1. Biological neuron simulations often need to detect when a variable crosses a certain threshold, for example a neuron will 'spike' when a voltage passes an upper bound. If the Dormand-Prince method has adapted to use a very large value of *dt* at the point that it crosses the threshold, it may be necessary to estimate the exact timing of the spike using linear interpolation within the time-step. A "root finding" algorithm may be used for extra accuracy but will probably be redundant if the Dormand-Prince method is already configured to find an accurate solution.
2. After a threshold has been reached and the system has been reset according to some rules (for example the voltage might be reset to a resting state) the previous calculated value for dt is no longer relevant for finding an accurate solution. This is because the accuracy obtained by using a particular time-step is dependent entirely on the state of the system's variables, so when they get reset so must the value for *dt*. Without modification, the Dormand-Prince method will detect any sudden accuracy changes but it may be worth discarding the old value for *dt* each time a threshold is passed, or when any non-continuous state-change occurs.

6 SPECIFIC TAILORED METHODS

In practice, the SpineCreator tool may be used predominantly to simulate only a small selection of neuron-types. This makes it worthwhile to seek specific tailored solution methods for some of these neuron types, and detect when these systems are being simulated so the tailored solution can be used instead of a general one.

6.1 IZHIKEVICH MODEL ZERO-ORDER HOLD SOLUTION [2]

The Izhikevich model of the neuron is a very popular model that can be solved in an efficient manner using a Zero-Order Hold solution. The Izhikevich model uses two state-variables **u** and **v**, and their time-derivatives result in a dynamical system that cannot be solved analytically, but if either **u** or **v** is assumed to be stationary for some short time period, the other variable can be solved analytically and accurately progressed in time. [see paper]

6.2 EXPONENTIAL DECAY AND OTHER SIMPLE SYSTEMS

In a similar vein to above, analytic solutions may easily be detected for very simple systems that have exponential or polynomial solutions etc. These could be detected and solved easily without using advanced algebraic techniques, and given that biological neural models commonly use functions such as exponential decay, detecting and using an analytic solution for these limited cases may result in significant efficiency gains, particularly in the case where there is an exponential decay component for each neuron in the model.

7 GENERAL ALGEBRAIC SOLVING METHODS [3]

There exist powerful and general algebraic methods for solving large classes of ODEs analytically. These methods take as input the algebraic form of all the differential equations and automatically classify and solve them to produce a closed form solution if one exists. The Wolfram Language, for example, includes a function **DSOLVE** which automatically finds efficient closed form solutions to given systems of ODEs. **DSOLVE** takes advantage of numerous advanced mathematical methods to create a very general and versatile algebraic solver. Implementing or investigating the specific methods were out of scope of this project, however this route may be worthwhile in the case that SpineCreator is being used to simulate a large range of analytically solvable systems. Given that this route may be time consuming to implement, or large third party libraries must be included, it might not be worthwhile until its advantages are fully understood for typical SpineML simulations.

8 EXACT SPIKE TIMINGS AND LOWER FREQUENCY COMPONENT-TO-COMPONENT COMMUNICATION [4]

This document so far has been concerned with methods for solving ordinary differential equations. This section discusses a larger structural change to SpineCreator's simulation techniques, involving increasing the fidelity of spike-timings while lowering the frequency of data communication between components.

The most significant hurdle to parallelising the computational simulation of a highly connected neural network is the constant passage of data between neurons. This is a bottleneck because the read and write commands on global shared memory in graphics cards are expensive and non-parallel. It is therefore advantageous to know beforehand which sections of the simulated system are isolated from other events within the system, and how long they will be isolated from those events. This information is analogous to knowing the "event horizon" in a physical system, i.e. the boundary in space that the effects of a particular event could have reached by a certain time. In physics the event horizon expands at a maximum of the speed of light.

In a biological neural network however, events are propagated to other neurons only after a **synaptic delay**. The crucial observation is that when a spike occurs at a particular point in time, its effects to other neurons will not be heard until the synaptic delay period has passed. This means that the simulation of a component can be completely isolated from other components for a period equal to the minimum synaptic delay within the network.

In its current implementation, the SpineCreator program accepts a single parameter *'dt'* which is used in three distinct ways:

1. The time-step used in the forward Euler integration.
2. The fidelity at which spike timings are represented; i.e. if a spike occurs within a time step, its timing is represented as the time at the end of the time step.
3. The rate at which neurons communicate events with each other. This is so events are propagated as soon as they occur.

There are three ways that this can be improved:

1. Use a more meaningful error tolerance parameter instead of a fixed *dt*, this will be handed directly to an adaptive step-size integration method.
2. Use either linear interpolation or a root-finding algorithm to calculate exact spike-timings not limited to a discrete time grid.
3. Communicate spiking events to other neurons at a rate equal to the minimum synaptic delay within the network. This is typically 1-2ms for chemical synapses, which is significantly larger than a typical step-size meaning that restricting communication between components to this rate may facilitate significantly more parallelisation on a GPU.

9 IMPLEMENTATION [5]

The following techniques were implemented in a stand-alone C++ application:

- Forward Euler
- Modified Euler
- Runge-Kutta 4th order
- Dormand Prince Adaptive step-size

The functions take as input an array of initial values and a corresponding array of pointers to their time-derivative functions. This enables the methods to be tested and compared quickly with a range of multi-variable ODE systems. Results are outputted to a text file in CSV format that can be visualised with third-part applications like Excel or Veusz.

A recent version of the code is available on GitHub at https://github.com/georgepowell/numerical-methods. Please refer to this for specific implementation details.

10 Figures & References

10.1 Figures

1. From Wikimedia Commons: http://en.wikipedia.org/wiki/File:Euler_method.svg
2. Visualisation of CSV output from project's C++ program in Veusz
3. From Wikimedia Commons: http://en.wikipedia.org/wiki/File:Instability_of_Euler%27s_method.svg
4. Visualisation of CSV output from project's C++ program in Veusz

10.2 References

1. Full descriptions of all numerical methods used in this document are available in **Numerical Recipes 3rd edition** - *William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery*
2. Full description, and analysis of zero-order hold solution of Izhikevich Neuron from *Humphries, M. D., & Gurney, K. (2007)* - **Solution methods for a new class of simple model neurons. *Neural Comp*, *19*, 3216–3225.** Available at http://www.abrg.group.shef.ac.uk/!DATA/attachment/0085.Humphries_simple_model_solutions_preprint.pdf
3. **DSOLVE** method in the Wolfram Language
    a. Documentation: http://reference.wolfram.com/language/ref/DSolve.html
    b. Implementation notes, including detailed list of mathematical techniques: http://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html#10466
4. Section inspired by *Morrison, A., Straube, S., Plesser, H. E., & Diesmann, M. (2007a).* **- Exact subthreshold integration with continuous spike times in discrete time neural network simulations. Neural Computation, 19, 1437–1467.**
5. Code available from http://Github.com