

Code Generation: A Strategy for Neural Network Simulators

Dan F. M. Goodman

Published online: 21 September 2010
© Springer Science+Business Media, LLC 2010

Abstract We demonstrate a technique for the design of neural network simulation software, runtime code generation. This technique can be used to give the user complete flexibility in specifying the mathematical model for their simulation in a high level way, along with the speed of code written in a low level language such as C++. It can also be used to write code only once but target different hardware platforms, including inexpensive high performance graphics processing units (GPUs). Code generation can be naturally combined with computer algebra systems to provide further simplification and optimisation of the generated code. The technique is quite general and could be applied to any simulation package. We demonstrate it with the ‘Brian’ simulator (<http://www.briansimulator.org>).

Keywords Code generation · Spiking neural networks · Simulation · Graphics processing units · Computer algebra systems · Numerical integration · Python · C++

Introduction

Since the early days of computational neuroscience, when Huxley computed numerical solutions to differ-

ential equations for the action potential (Hodgkin and Huxley 1952) using a hand cranked mechanical calculator over a period of three weeks, simulations in theoretical and computational neuroscience have continually pushed the boundaries of available computational resources. Given the enormous complexity of the central nervous system, this trend is likely to continue for the foreseeable future. Software for simulations must therefore be both efficient and capable of exploiting the latest hardware, such as general purpose graphics processing units (GPUs, chips consisting of hundreds of specialised processor cores). However, it must also be flexible, and not restrict which models scientists can investigate. These two requirements present a problem: computationally efficient software can be designed by writing code in a low level language such as C++, but this restricts flexibility. Users are typically required to choose from a fixed set of existing neuron models (as in Nest; Gewaltig and Diesmann 2007), or design their own from a combination of pre-defined mechanisms (as in Genesis; Bower and Beeman 1998).

An ideal software package would have the following characteristics. It would allow the user to specify models at a high level. This specification could be, for example, directly specifying the equations in standard mathematical notation, or a CellML or NeuroML document (Garny et al. 2008; Gleeson et al. 2010). It would not restrict what models or equations could be specified based on what had already been implemented. The resulting simulation would run close to as fast or preferably even faster than low level code written by hand. It could be run on different hardware configurations, such as a single desktop PC, a cluster, or a GPU, without the user having to make any changes.

D. F. M. Goodman
Laboratoire Psychologie de la Perception, CNRS,
Université Paris Descartes, Paris, France

D. F. M. Goodman (✉)
Département d’Etudes Cognitives, Ecole Normale
Supérieure, Paris, France
e-mail: dan.goodman@ens.fr

Code generation provides a way to achieve these goals. The basic strategy is as follows: the user works in a high level language such as Python (which features excellent scientific computing packages for data analysis and plotting; Oliphant 2007), or using a graphical interface. On running the simulation, the software package writes code in a low level language such as C++, compiles this code and then runs it (possibly just for key sections of the code). In this way, the advantages of a high level language can be combined with the speed of code written in a low level language, without sacrificing flexibility. In addition, because the model is specified at a high level, the output low level code can be targeted to the chosen hardware configuration without change. Code that is optimal for a CPU for example, is very different to code that is optimal for a GPU.

The first neural simulator package that used this strategy was Neuron (Carnevale and Hines 2006), with the NMODL model specification language (Hines and Carnevale 2000; based on the earlier MODL language of SCoP; Kootsey et al. 1986). Neuron provides an offline code generation tool which reads a model specification file and generates and compiles code for numerical integration and spike propagation for the model. Neuron's approach does require that the user learn some specific syntax and it does not yet feature automatic simplification via symbolic analysis or support for hardware such as GPUs, but it is already an invaluable improvement and may well account for some of Neuron's enormous success.

In this paper we introduce some general principles of code generation in Section “Principles”, including a simple worked example and some discussion of how it could be used in other ways. In Section “Code Generation in Brian” we discuss in detail the code generation strategy used in the Brian neural network simulator package (Goodman and Brette 2008, 2009). Brian uses runtime code generation and compilation, so that code generation is entirely transparent to the user and does not require them to run a separate tool. Finally, in Section “Future Work and Discussion”, we outline plans for future work on Brian (including more support for GPUs and automatic tuning of code at runtime, which may allow for code that is faster than hand written C++) and discuss the role of code generation in neural network simulation. Throughout, we use the technique of symbolic analysis of mathematical expressions, which we consider essential for efficient code generation. We also consider GPU computing in several places, as the necessity of using code generation for the GPU has been one of the major motivations for this work.

Principles

The technique of code generation involves taking a high level representation of a model and generating efficient low level code for various platforms (different CPUs, GPUs, etc.). This code can then be compiled and run behind the scenes, in a way that is entirely transparent to the user. There are two main reasons for doing this: speed and flexibility. For code that is written in a high level language, the primary benefit will typically be speed, because the high level language will already be reasonably flexible (although this is not an automatic property of writing code in a high level language). For code that is written in a low level language, including most existing simulator packages, the primary benefit will be flexibility, as the code will already be highly efficient (although again, this is not an automatic property of writing code in a low level language). A related aspect is that code generation can also be used with multiple target platforms. For example, models can be run on single CPUs, clusters and GPUs. The optimal code may well be different in each case. Using a code generation technique allows us to write templates for each platform once, and then generate code for specific models on any of these platforms without having to duplicate and rewrite them. This will be particularly important in the case of GPUs, as there are many different types of GPUs available and it is difficult to precompile code for all of them.

Worked Example

We start by giving a detailed example of how code generation could work in the case of numerical integration, using Python, C++ and GPU C++ as examples. We consider a leaky integrate-and-fire neuron defined by the following system of differential equations:

$$\begin{aligned}\tau_m \frac{dv}{dt} &= -v + g_e(E_e - v) + g_i(E_i - v) \\ \tau_e \frac{dg_e}{dt} &= -g_e \\ \tau_i \frac{dg_i}{dt} &= -g_i.\end{aligned}\tag{1}$$

This neuron fires a spike when the membrane potential crosses a threshold, $V > V_r$, and subsequently resets to a fixed value, $V \leftarrow V_r$. In this section we consider only the numerical integration of the differential equations.

These could be specified in Python as a multi-line string:

```
equations = '''
dv/dt = (-v+ge*(Ee-v)+gi*(Ei-v))/taum
dge/dt = -ge/taue
dgi/dt = -gi/taui
'''
```

A simple pattern match on each line of the string gives us the expression $f(x)$ in the equation $dx/dt = f(x)$. So for the variable `ge` it is the string `'-ge/taue'`. With this stored in a Python dictionary `expr` (a dictionary in Python is a mapping `m` associating *keys* `k` with *values* `m[k]`), we can now generate C++ code for an Euler integration scheme for a set of N neurons as follows (assuming that variable `v` is stored as an array `v__array`, and so on):

```
code = 'for(int i=0; i<N; i++){\\n'
for var in variables:
    code += '    double &'+var+' = '+var+'__array[i];\\n'
for var in variables:
    code += '    double '+var+'__tmp = '+expr[var]+';\\n'
for var in variables:
    code += '    '+var+' += dt*'+var+'__tmp;\\n'
code += '}'
```

In our case, this gives the following C++ code as output:

```
for(int i=0; i<N; i++){
    double &v = v__array[i];
    double &ge = ge__array[i];
    double &gi = gi__array[i];
    double v__tmp = (-v+ge*(Ee-v)+gi*(Ei-v))/taum;
    double ge__tmp = -ge/taue;
    double gi__tmp = -gi/taue;
    v += dt*v__tmp;
    ge += dt*ge__tmp;
    gi += dt*gi__tmp;
}
```

In a similar way, we could generate Python code as output, assuming that variable `v` is stored in Python array `v` and so on, the generated code would be:

```
v__tmp = (-v+ge*(Ee-v)+gi*(Ei-v))/taum
ge__tmp = -ge/taue
gi__tmp = -gi/taue
v += dt*v__tmp
ge += dt*ge__tmp
gi += dt*gi__tmp
```

The Euler scheme is particularly simple, but starting from the same set of equations defined as a string,

we could define integrators for other schemes such as Runge–Kutta or implicit Euler (see Section “[Code Generation in Brian](#)”). We can also do some symbolic analysis on the equations strings provided by the user, for example substituting constant values, evaluating them and simplifying the resulting expressions. We discuss this further in the next section, but we finish with an example in which divisions by a constant have been replaced by less computationally expensive multiplications using particular values for the constants ($\tau_m = 20$ ms, $\tau_e = 5$ ms, $\tau_i = 10$ ms, $E_e = 60$ mV and $E_i = -20$ mV). This example uses the same equations, but integrated with a second order Runge Kutta scheme, and the target platform is a GPU kernel (the GPU specific code will be explained in more detail in Section “[Code Generation in Brian](#)”):

```
__global__ void integrate(double *v__array,
                          double *ge__array,
                          double *gi__array,
                          double dt, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i>=N) return;
    double &v = v__array[i];
    double &ge = ge__array[i];
    double &gi = gi__array[i];
    double v__buf = -50.0*v+50.0*ge*(0.06-v)
                  -50.0*gi*(0.02+v);
    double ge__buf = -200.0*ge;
    double gi__buf = -100.0*gi;
    double v__half = 0.5*dt*v__buf;
    double ge__half = 0.5*dt*ge__buf;
    double gi__half = 0.5*dt*gi__buf;
    v__half += v;
    ge__half += ge;
    gi__half += gi;
    v__buf = -50.0*v__half+50.0*ge__half
            *(0.06-v__half)
            -50.0*gi__half*(0.02+v__half);
    ge__buf = -200.0*ge__half;
    gi__buf = -100.0*gi__half;
    v += dt*v__buf;
    ge += dt*ge__buf;
    gi += dt*gi__buf;
}
```

This last example is in fact the code generated by Brian, which we will consider in more detail in Section “[Code Generation in Brian](#)”.

Numerical Integration with Existing Libraries

An alternative approach to numerical integration allowing us to straightforwardly make use of existing, sophisticated numerical integration packages is exemplified by Neuron’s NMODL language (Hines and

Carnevale 2000). Here, the high level representation is a Neuron mod file. This file includes various code blocks containing the definition of the model, including a DERIVATIVE block which gives the differential equations of the state variables of the model in standard mathematical form, as in the example above. An example of such a block is:

```
DERIVATIVE states {
  settables(v)
  r' = ((ralpha*(1-r)) - (rbeta*r))
  d' = ((dbeta*(1-s-d)) - (dalpha*d))
  s' = ((salpha*(1-s-d)) - (sbeta*s))
}
```

A tool included as part of Neuron generates C code and compiles it to an executable file which can be loaded by Neuron. Part of this C code is a function which executes this code block. For the example above it is:

```
static int _ode_spec1 (double* _p,
                      Datum* _ppvar,
                      Datum* _thread,
                      _NrnThread* _nt)
{int _reset = 0; {
  settables ( _threadargscomma_v );
  Dr = ( ( ralpha * ( 1.0 - r ) )
        - ( rbeta * r ) );
  Dd = ( ( dbeta * ( 1.0 - s - d ) )
        - ( dalpha * d ) );
  Ds = ( ( salpha * ( 1.0 - s - d ) )
        - ( sbeta * s ) );
}
return _reset;
}
```

The main code of Neuron integrates the differential equations using the CVODES routine of the SUNDIALS package (Hindmarsh et al. 2005), by passing it a pointer to this function. CVODES then uses this function to evaluate the derivatives at various points as part of its numerical integration algorithm, which includes many sophisticated features such as variable time-stepping.

Whether this method or the method described in the previous section is more appropriate depends on several factors. In general, it is preferable to use code that has been well researched and tested, such as CVODES, as it is likely to be more efficient, to have less bugs, and because doing so reduces duplicated development time. There are, however, some reasons to prefer the

simpler, more direct approach of the previous section. For Brian, we chose to do this as we decided on a requirement that our package work in pure Python (with NumPy and SciPy) to maximise portability, and this ruled out the use of libraries written in C/C++. However, the same choice may be made by developers working in C, as they still introduce an additional external dependency, potentially complicating development and distribution. In addition, some desired target platforms may not be supported by the library, or may not run efficiently on them. This is a particular concern for GPU code, which often has to be written in a very different way to CPU code, and the technology is sufficiently new that many existing libraries do not support them well.

Implementing XML Models

Code generation is not restricted to transforming mathematical equations into code. Indeed, an interesting and timely problem is the generation of code from extensible markup language (XML) documents such as NeuroML, CellML and SBML. NeuroML is an XML specification for the definition of models of neural systems (Gleeson et al. 2010), CellML for cellular and subcellular processes (Garny et al. 2008), and SBML for biochemical reaction networks (Hucka et al. 2003). Standardised markup languages such as these can be an invaluable tool for neuroscience, by facilitating the reproducibility and testability of models (Morse 2007). As yet, no neural simulator includes complete support for these languages, however the situation is improving and many simulators now support at least a subset of them.

Simulator support for these languages can be partly satisfied without code generation, by simply implementing the named models included in the specifications. However, these specifications allow for the creation of new models that have not been previously implemented. For example, NeuroML allows for the specification of channel gating mechanisms via state-based kinetic models, and plans for future versions include references to much more general SBML, CellML and MathML documents. Some work in neural simulation has already been realised in this direction, for example the NeuroML Validator (<http://www.neuroml.org/NeuroMLValidator/Validation.jsp>) can be used to generate Neuron mod files from NeuroML documents, and these can in turn be used to generate code by Neuron. However, future versions of NeuroML incorporating MathML (directly or via CellML or SBML) will require code generation, as MathML allows for the specification of arbitrary mathematical expressions. This could be done, for

example, using the MathDOM Python package (<http://mathdom.sourceforge.net/>), which reads MathML documents and can generate output in various formats suitable for use in code generation.

The problem of code generation has already been partially addressed by the systems biology community, although their solutions are not directly applicable to neural simulation. There are several pure CellML simulators that support code generation (Garny et al. 2008). In some cases, these build on the CellML Code Generation Service (CCGS) of the CellML API (Miller et al. 2010), which in turn uses the API’s MathML Language Expression Service (MaLaES) to convert MathML fragments into expressions in several different programming languages. Many SBML tools also use code generation (http://sbml.org/SBML_Software_Guide/SBML_Software_Summary), but there are no common code generation tools equivalent to CCGS and MaLaES.

Code Generation in Brian

From the earliest versions, the Brian simulator used code generation for numerical integration of nonlinear differential equations, although only Python code was generated. The latest version features Python, C++ and GPU C++ code generation for numerical integration, thresholding (for example, firing spikes on the condition that $V > V_t$), resetting (after a spike, for ex-

ample $V \leftarrow V_r$), propagation and back-propagation of spikes. In some situations, this can result in substantial speed improvements even on a single CPU, because the Python overheads are eliminated. In the case of large N , we showed in Goodman and Brette (2008) that pure Python performance approaches that of hand written C++ code, because the relative costs of the overheads become negligible. However, in order to extract the maximum advantage from a GPU, C++ code generation is necessary. Figure 1 shows the structure of Brian’s main simulation loop and how generated code is used.

In Python, we can compile and execute code using the built-in `compile` and `eval` functions, and the `exec` keyword. The SciPy package (Jones et al. 2001) includes “Weave”, a package for automatic compilation and execution of C++ code in a way that easily integrates with the high level data structures in Python. For the GPU, we use the PyCUDA package (Klöckner et al. 2009). In each case, we pass a string representation of the code (in Python, C++ or GPU C++) along with a namespace (a mapping of variable names to data), and low-level code is automatically compiled and executed.

To implement code generation, we use a combination of string manipulation using regular expressions, and the SymPy computer algebra system (CAS) package (SymPy Development Team 2009). The overall process is illustrated in Fig. 2 for the case of numerical integration (the other cases are similar). SymPy

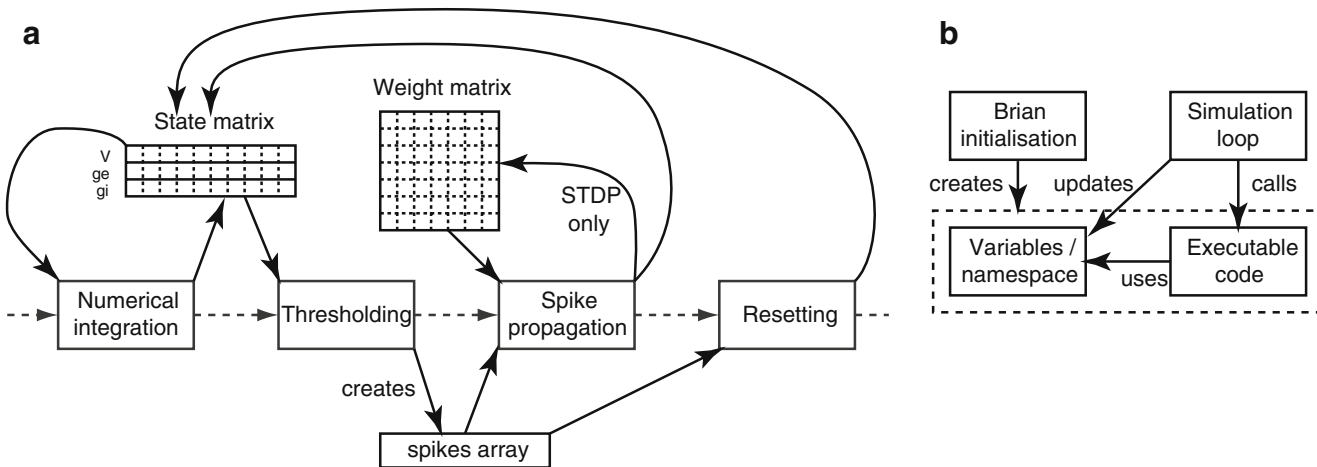


Fig. 1 (a) Main simulation loop. Each time step, the following operations are executed: numerical integration (updating the state matrix); thresholding (producing a list of spiking neurons); spike propagation (updating the state matrix and, in the case of STDP, the weight matrix); and resetting. Each operation involves calling generated code, illustrated in the *right hand panel*. (b) Generated code consists of two objects, a namespace (a mapping

of variable names to data) and an executable code object. The code executes in the namespace, that is, it acts on the objects in the namespace. At initialisation, Brian creates a namespace and executable object, the latter using code generation (see Fig. 2). Each time step, the namespace is updated with the current values of the variables before the executable object is called

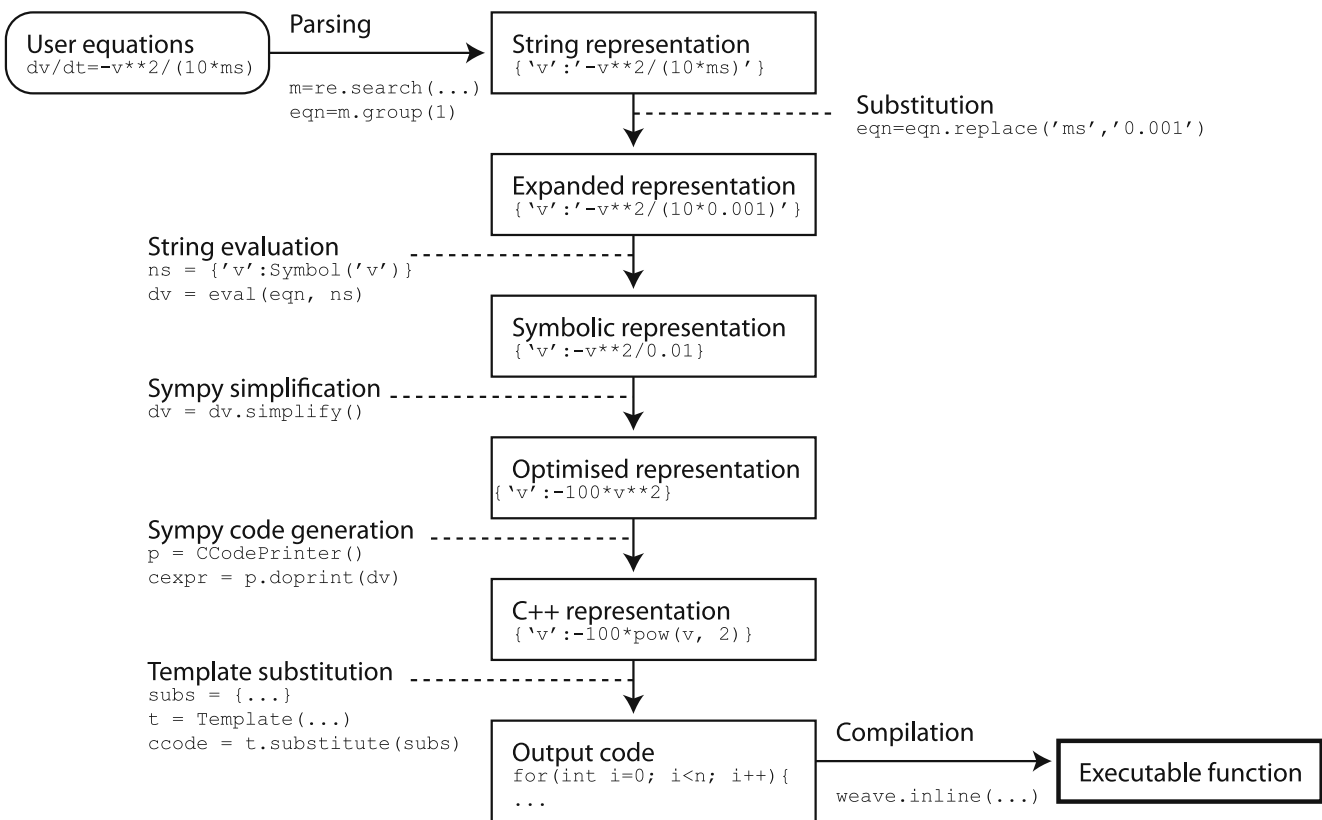


Fig. 2 Code generation process for numerical integration. Starting with the user equations, a string representation is built by parsing the equations using regular expressions. An expanded representation is built by substituting the value of constants. These string representations are then evaluated symbolically using Sympy, creating Sympy symbolic expression objects. These expressions are then simplified using Sympy, for example replac-

ing division by a constant with multiplication. Sympy is then used to output C++ code from these optimised representations (which involves, for example, replacing $x**y$ with $\text{pow}(x, y)$). Finally, this C++ code is inserted into templates defined by the numerical integration scheme to form the final output code. This code is then compiled and executed using the SciPy Weave package

can be used for algebraic manipulation of expressions, including automatically simplifying expressions such as $x/(10*ms)$ to $x*100$. It uses Python syntax to define mathematical expressions, so that anything that is valid Python will be valid for SymPy. It also includes a module for generating C++ expressions, which involves some non-trivial syntactic transformations. For example, in Python there is a $**$ exponentiation operator, whereas in C++ you need to use the pow function. So the Python expression $x**2$ would need to be rendered in C++ as $\text{pow}(x, 2)$. Beyond these syntactic differences at the level of expressions, there are differences in vectorisation strategies. For example, a reset operation that sets the value of variable V to V_r for every neuron that has spiked could be coded in Python as:

```
for i in spikes:
    V[i] = Vr
```

However, this incurs a performance penalty of $O(n)$ Python operations, for n the number of neurons that have spiked. This penalty comes about as follows. Executing this code in C++ requires Cn machine code instructions, whereas executing it in Python requires the same Cn machine code instructions to actually perform the computation plus Pn additional instructions to interpret the Python expression $V[i] = Vr$ for each iteration of the loop, where $P \gg C$. Writing it as follows reduces that penalty to $O(1)$, that is, requiring only $\text{const.} + Cn$ instructions:

```
V[spikes] = Vr
```

We do not attempt to address the general case of optimal vectorization strategies, which is an ongoing research area. Projects attempting to address this for Python include Psyco and PyPy (Rigo 2004), and Theano (<http://deeplearning.net/software/theano/>).

In the sections below, we cover how Brian handles code generation for numerical integration and spike propagation, including STDP. We do not discuss other algorithmic components of model specification, such as the thresholding and reset mechanisms, as the same techniques apply fairly straightforwardly to these cases. By default, Brian handles thresholding by finding the indices of all neurons satisfying $V > V_t$, but any mathematical expression can be given and code can be generated from these. Resetting is again typically handled by setting $V = V_r$ for all neurons that have spiked, and, in the case of refractoriness, is held at that value by repeatedly executing that statement for the duration of the refractory period. Again, any statement or series of statements can be given and code can be generated from them, including more complicated schemes having, for example, one statement to be evaluated at the initial reset time, and another statement for the refractory period. In Brian, spike times are constrained to the grid, which can cause numerical problems in some cases, although these could be ameliorated by introducing spike time interpolation (Hansel et al. 1998; Morrison et al. 2007).

There are various possible sources of error in the process outlined above that need to be handled. First of all, users may write equations with syntactic errors: these will be handled at parsing time, as the attempt to convert them into SymPy expressions will raise an exception. Secondly, users may write syntactically correct but semantically incorrect equations. These cannot be detected automatically, although in Brian they will raise an error if they are dimensionally inconsistent, as Brian has a system for defining variables with physical units. Finally, some errors may occur if the user writes an expression that is well defined in Python but not in C++, for example using a Python function they have defined. This will also raise an error when attempting to compile the C++ code, but in this case a fall-back strategy can be used: revert to using Python code generation if C++ code generation fails.

Numerical Integration

The most important aspect of code generation in Brian is the generation of numerical integration code. Brian currently features four integration schemes: exact solution of linear differential equations; Euler method; 2nd order Runge–Kutta method; exponential Euler method (MacGregor 1987). We do not use code generation for linear differential equations as the exact solution for these can be computed as a matrix-vector product ($\mathbf{x}(t + dt) = M\mathbf{x}(t) + \mathbf{a}$ for a fixed matrix M and vector \mathbf{a} ; Rotter and Diesmann 1999) which can be com-

puted efficiently with the NumPy package (Oliphant 2006).

The integration schemes are expressed as a list of code blocks. Each code block is a string, possibly consisting of multiple lines. Output code is constructed by evaluating the first code block for each variable, then the second code block, and so on. The following substitutions are made: $\{\text{var}\}$ is replaced by the name of the variable (extracted by pattern matching from the user defined equations); $\{\text{var_expr}\}$ by the expression for the right hand side of the differential equation for that variable (also extracted by pattern matching); $\{\text{vartype}\}$ by the data type of the variable (none in Python, and either `float` or `double` in C++). In this framework, the Euler scheme is expressed very simply as:

```
euler = ['${vartype} ${var}__tmp = ${var_expr}',
        '${var} += ${var}__tmp*dt']
```

The way Python code is generated from this scheme is then essentially the same as was demonstrated in Section “[Worked Example](#)” for C++ code, only more general. First of all, a list of differential equations defined by the name of the variable (`var`) and the expression defining its derivative (`var_expr`) is extracted by pattern matching on the equations string. Then, for each variable defined in the user equations, each of the code block templates are filled out with the appropriate values and joined together. The output of this scheme is valid Python code, all that is added to it is to define the variable names of the differential equations. In Brian, the variables are stored in a single two-dimensional array S of dimensions (M, N) for M variables and N neurons. The final Python code for the integration scheme is joined to code that loads the variable names of the differential equations with pointers to this array (“views” in NumPy terminology). So, for the differential equations of Section “[Worked Example](#)” the complete Python code would be:

```
v = S[0, :]
ge = S[1, :]
gi = S[2, :]
v__tmp = -50.0*v + 50.0*ge*(0.06 - v)
        - 50.0*gi*(0.02 + v)
ge__tmp = -200.0*ge
gi__tmp = -100.0*gi
v += dt*v__tmp
ge += dt*ge__tmp
gi += dt*gi__tmp
```

If this code is stored as a string `pycode`, and the variables `S` and `dt` are stored in a namespace `ns`, then

the code above can be executed with the Python statement `exec pycode in ns`. For C++ code output, we have to loop over all neurons, and we use references to load the variable names, as follows:

```
double *v_ptr = S+0*num_neurons;
double *ge_ptr = S+1*num_neurons;
double *gi_ptr = S+2*num_neurons;
for(int i=0; i<num_neurons; i++){
    double &v = *v_ptr++;
    double &ge = *ge_ptr++;
    double &gi = *gi_ptr++;
    ...
}
```

On the GPU, we do not loop over neurons, instead we launch a GPU kernel process with one thread per neuron. A GPU consists of a large number of processor cores which can execute code in parallel (512 cores in the current state of the art chips). There are restrictions on what code can be executed on these cores, which we do not go into in detail here except to say that roughly speaking, each core has to execute the same code on a different piece of data. Code running on the GPU is called a “kernel”. A GPU kernel launch involves multiple blocks and threads that can be divided into a one-, two- or three-dimensional structure. In our case, we launch a one-dimensional kernel but with multiple blocks, each block consisting of multiple threads. The neuron index can be computed from the block index `blockIdx.x`, the block dimension `blockDim.x` and the thread index `threadIdx.x` as `blockIdx.x*blockDim.x+threadIdx.x`. If the number of neurons does not divide perfectly into the block dimension, then too many threads will be launched. The GPU kernel has to check for this condition and return without doing anything in that case. The neuron model state variables `v_array`, `ge_array` and `gi_array` are stored in the global memory of the GPU, that is, memory that is accessible to every thread. The GPU kernel code for our example is then:

```
__global__ void integrate(double *v_array,
                        double *ge_array,
                        double *gi_array,
                        double dt, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i>=N) return;
    double &v = v_array[i];
    ...
}
```

The code written in this way is optimal for the GPU since if two threads are adjacent (say i and $i + 1$) they read and write to global memory at adjacent locations, meaning that the memory access can be “coalesced” into a single operation for several adjacent threads. Ensuring coalesced global memory access patterns is the key to optimal GPU code efficiency, as global memory access can take hundreds of clock cycles to complete (NVIDIA 2009), in many cases dwarfing the arithmetical operations involved in the numerical integration itself.

Moving on to more complicated numerical integration schemes, we require one addition to the framework developed above. We use the syntax `@substitute (expr, subs)` to make substitutions of different names or values into an expression. The `substitute` function evaluates the Python code `subs` which should be a dictionary of key-value pairs (`v`, `e`). This dictionary is then used to make substitutions into `expr`, replacing variable `v` with expression `e`. With this, we can define the 2nd order Runge–Kutta method. For the differential equations $dx/dt = f(x)$ (for vector x) we define $x_{half} = x + (dt/2) f(x)$ and the numerical integration step is then $x \leftarrow x + dt \cdot f(x_{half})$. In our framework, this can be written as follows:

```
rk2 = [
    '''
    ${vartype} ${var}__buf = ${var_expr}
    ${vartype} ${var}__half = (.5*dt)
        *${var}__buf
    ${var}__half += ${var}
    '''
    ,
    '''
    ${var}__buf = @substitute(var_expr,
        dict((v, v+'__half')for v in variables))

    ${var} += dt*${var}__buf
    '''
]
```

We require the `@substitute` to replace each variable with its half timestep value. In this case, the expression (for example) `'x*y'` would be replaced by `'x_half*y_half'` after applying the substitution dictionary `{'x':'x_half', 'y':'y_half'}`. The Python expression `dict(...)` creates a dictionary consisting of all key-value pairs (`v`, `v+'__half'`) for each variable `v` (a string) in the list `variables`.

Finally we have the exponential Euler method for stiff differential equations such as those of Hodgkin–Huxley type. This method can be applied to differential equations which are conditionally linear, that is, where the right hand side of each differential equation is linear with respect to the variable being differentiated while holding the other variables constant. For each variable x , the differential equation is $dx/dt = Ax + B$

for A and B depending on the other variables. We compute the solution to this differential equation under the assumption that A and B are constant as $x(t + dt) = (x(t) + B/A)e^{-A dt} - B/A$. We implement this in our framework as follows, using `@substitute` to compute the A and B values for each variable:

```
exp_euler = [
    '''
    $ {vartype} $ {var}__B = @substitute(var_expr,
                                       {var:0})
    $ {vartype} $ {var}__A = @substitute(var_expr,
                                       {var:1})

    $ {var}__A -= $ {var}__B
    $ {var}__B /= $ {var}__A
    $ {var}__A *= dt
    ''' ,
    '''
    $ {var} += $ {var}__B
    $ {var} *= exp($ {var}__A)
    $ {var} -= $ {var}__B
    ''' ]
```

The expression `{var:0}` is a Python dictionary consisting of a single key-value pair (`var, 0`). The effect of this in the code above is to substitute the value `var=0` in `var_expr`. If $dx/dt = f(x) = Ax + B$ then substituting $x = 0$ gives us $f(0) = B$. Substituting $x = 1$ gives us $f(1) = A + B$ so we compute A as $A = f(1) - f(0)$.

Propagation and Back-propagation

Simulation of spike-timing dependent plasticity (STDP) of a general nature (defined by user-specified equations) requires code-generation integrated with propagation and back-propagation. We implement STDP by defining a set of synaptic differential equations, and code that is executed for each synapse after a pre- or post-synaptic spike. This code is provided by the user in a high level way, and we generate Python or C++ from it. As an example, the standard model of STDP is defined by a weight change ΔW depending on a pair of pre- and post-synaptic spikes at relative times $\Delta t = t_{\text{post}} - t_{\text{pre}}$:

$$\Delta W = \begin{cases} A_- e^{\Delta t/\tau_-} & \text{if } \Delta t < 0 \\ A_+ e^{-\Delta t/\tau_+} & \text{if } \Delta t > 0 \end{cases} \quad (2)$$

Typically, $A_+ > 0$ for potentiation and $A_- < 0$ for depression. This can be implemented using our STDP mechanism using the technique of (Song et al. 2000).

Define pre- and post-synaptic traces a_- and a_+ which evolve according to the differential equations:

$$\begin{aligned} \tau_- \frac{da_-}{dt} &= -a_- \\ \tau_+ \frac{da_+}{dt} &= -a_+ \end{aligned} \quad (3)$$

When a pre-synaptic spike arrives, we update $a_+ \leftarrow a_+ + A_+$ and $W \leftarrow W + a_-$. When a post-synaptic spike arrives, we update $a_- \leftarrow a_- + A_-$ and $W \leftarrow W + a_+$. You can see that if a pre-synaptic spike arrives time Δt before a post-synaptic spike then at the time of the pre-synaptic spike a_+ will be set to A_+ and W will be increased by $a_- = 0$. At the time of the post-synaptic spike, a_+ will have decayed to $A_+ e^{-\Delta t/\tau_+}$ and W will be increased by this amount, as required. In a similar fashion, if the post-synaptic spike arrives $-\Delta t$ before the pre-synaptic spike, then initially a_- will be set to A_- and W will be increased by $a_+ = 0$, and then at the time of the pre-synaptic spike a_- will have decayed to $A_- e^{-\Delta t/\tau_-}$ and W will be increased by this amount, again as required. These equations for STDP are specified in Brian by the differential equations string:

```
equations = '''
dam/dt = -am/taum
dap/dt = -ap/taup
'''
```

Here `m` and `p` are used in place of `-` and `+`. The pre-synaptic code is specified as:

```
pre = '''
ap += Ap
W += am
'''
```

And the post-synaptic code as:

```
post = '''
am += Am
W += ap
'''
```

The user can also optionally specify limits for the weight. Note that the variable `ap` only depends on pre-synaptic spikes and the variable `am` only depends on post-synaptic spikes, so we do not store one copy of these per synapse, but only per neuron. We then use numerical integration as in the previous section to solve the differential equations. In Python, the pre-synaptic code generated for these equations is as follows, where

`spikes` is an array of the indices of the pre-synaptic neurons that have spiked:

```
ap[spikes] += Ap
for i in spikes:
    W[i, :] += am
    W[i, :] = clip(W[i, :], 0, wmax)
```

The line `ap += Ap` in the original string has been vectorized over the pre-synaptic neurons and replaced by `ap[spikes] += Ap`. The line `W += am` has been vectorized once for each spike `i` over the post-synaptic neurons as `W[i, :] += am`. The expression `W[i, :]` refers to row `i` of the matrix `W`. Finally, a line has been added to clip the values of `W` between 0 and `wmax`. For the code run on a post-synaptic spike, we do the same thing but replace `W[i, :]` with `W[:, i]` to refer to column `i` of `W` instead of row `i`.

This code is generated by analyzing the equations and the pre- and post-synaptic code strings. The first step is to divide the equations into pre- and post-synaptic equations. This is done by creating a dependency graph on the differential equations (with nodes the variables of the equations, and edges indicating a dependency between the two variables), and splitting it into two connected components. Variables which are modified in the pre-synaptic code are considered pre-synaptic and those which are modified in post-synaptic code are considered post-synaptic. The consistency of all this is checked and an error raised if there is a problem. To generate the pre- and post-synaptic Python code, the strings are split into lines that do and lines that do not modify the weight matrix. For lines that do not modify it, each variable `x` is simply replaced by `x[spikes]`. Lines that do modify the weight are wrapped in a loop `for i in spikes:` and the symbol `W` is replaced by `W[i, :]` for propagation or `W[:, i]` for back-propagation.

Code generation for customised propagation and back-propagation is complicated by several factors. In Brian there are at least 12 types of propagation that can occur: there can be homogeneous or heterogeneous neuronal delays; the connection matrix structure can be dense, sparse or ‘dynamic’ (and more structures are planned); there can be synaptic weight modulation or not. There is also the difference between propagation (which involves iterating over rows of the connection matrix) and back-propagation (which involves iterating over columns). The difference between rows and columns is not entirely trivial in sparse matrix data structures. Normally, sparse matrices are optimised either for efficient row access (CSR type) or efficient column access (CSC type). We use our own sparse matrix data structure which allows for efficient row and

column access at the cost of slightly higher memory requirements. With STDP, further complications are added by the fact that we may use the value of a variable at an earlier point in time rather than the current time if neuronal delays are being used (although we will not cover this case here).

In Python, we solve these problems by requiring that connection matrix structures define row and column based vector operations, and then we work on a spike-by-spike basis as in the Python code for STDP above. This means that (back-)propagation involves $O(n)$ Python instructions, for n the number of spikes per time step.

Using C++ we can reduce this to $O(1)$ Python instructions, at the cost of slightly more complexity.

Code is specified in an iterative scheme similar to Python, with iteration allowed over the array of spike indices `spikes`, or over a row or column of a matrix. Neuron model variables can be loaded with the `load` keyword. For example, standard propagation of spikes is defined by the following scheme:

```
for j in spikes:
    for k in row j of W:
        load V[k]
        V += W
```

That is, we iterate over each neuron index `j` in the array `spikes`. Now we iterate over row `j` of the matrix `W`, so that for each iteration step the target neuron index is `k`. Finally, we load the target neuron state variable `V` corresponding to neuron index `k`, and execute the code `V += W`.

In the case of a sparse matrix `W`, this generates the following C++ code (annotated with the line of the scheme that generated it):

```
// for j in spikes:
for(int spike_index=0; spike_index<spikes_len;
    spike_index++)
{
    int j = spikes[spike_index];
    // for k in row j of W:
    for(int p=rowindex[j]; p<rowindex[j+1]; p++)
    {
        int k = allj[p];
        double &W = alldata[p];
        // load V[k]
        double &V = V__array[k];
        // V += W
        V += W;
    }
}
```

The `rowindex`, `allj` and `alldata` variables belong to the sparse matrix data structure. The weight

values for row i are stored in `alldata` with indices from `rowindex[i]` up to but not including `rowindex[i+1]`. The corresponding column index are stored in array `allj`.

For STDP, the post-synaptic code is generated with the following scheme:

```
for j in spikes:
    for var in post_vars:
        load var[j]
    per_neuron_post
    for i in row j of W:
        for var in pre_vars:
            load var[i]
        per_synapse_post
    clip
```

The variables `pre_vars` and `post_vars` are dictionaries of pairs (`var`, `array`) associating variable name `var` with its array of values `array`, for the pre- and post-synaptic variables. The variables `per_neuron_post` and `per_synapse_post` are the lines of code corresponding to the per-neuron or per-synapse code (determined by whether it modifies the weights or not). The variable `clip` is code to clip the value of `W` depending on whether upper or lower limits have been given. The pre-synaptic code is similar but uses row i of `W` instead of col i of `W`. Here is the generated post-synaptic code for a dense matrix (annotated as before):

```
// for j in spikes:
for(int spike_index=0; spike_index<spikes_len;
    spike_index++)
{
    int j = spikes[spike_index];
    // for var in post_vars:
    //     load var[j]
    double &am = am_array[j];
    // per_neuron_post ('am += Am')
    am += -0.000105;
    // for i in row j of W:
    for(int i=0; i<num_source_neurons; i++)
    {
        double &W = W_array[i+j*num_target_neurons];
        // for var in pre_vars:
        //     load var[i]
        double &ap = ap_array[i];
        // per_synapse_post ('W += ap')
        W += ap;
        // clip
        if(W<0.0) W = 0.0;
        if(W>0.01) W = 0.01;
    }
}
```

In the case of STDP with delays, the scheme is slightly more complicated. We do not present further details here. In addition, Brian does not yet support propaga-

tion and back-propagation on the GPU. This is possible but technically challenging due to the difficulty of devising algorithms that involve coalesced memory access patterns (Nageswaran et al. 2009). GPU propagation is planned for release 2.0 of Brian. Finally, we should note that in Brian we do not use the exact syntax above for propagation and back-propagation, but it is equivalent.

Results

We demonstrate the results of using code generation in Brian in Fig. 3. These results are designed to show that code generation allows us to bridge the gap between speed and flexibility in a simulator written in a high level language (with the interpretation costs to overcome). A simulator already designed in a low level language will not see any speed improvements from using code generation, but rather gain an improvement in flexibility. We compared performance in numerical integration using the exponential Euler method for Hodgkin–Huxley neurons (with no synapses), and in propagation and back-propagation using STDP (using a null neuron model with no numerical integration, and spikes introduced by hand). For the Hodgkin–Huxley neurons we used the model from Brette et al. (2007). For the numerical integration, we tested a standard version of Brian (1.2.0) without code generation, against Brian using Python code generation, Brian using C++ code generation, and the same numerical integration using hand-coded C++.

Consistent with our previous work (Goodman and Brette 2008), we found that in the case of a large number N of neurons, Brian performance with or without code generation approached that of hand-coded C++. This follows because the Python interpretation overheads are constant, and so for large N they become tiny relatively. In fact, it appears that both Python and C++ code generation, and hand-coded C++ tend towards a constant factor faster than Brian. This is probably because Brian without code generation does not include the simplification via symbolic algebra, and so it is actually performing a more complicated calculation. In the case of small N , each step from Brian to Python code generation to C++ code generation to hand-coded C++ gave a considerable speed improvement. This appears very strongly in the case of relative timings, but in the absolute timings it appears that there was virtually no difference between the speed of Brian with C++ code generation and hand-coded C++.

These speed improvements will be useful in the case of relatively small networks running for a long period of time (giving speed increases of up 40 times),

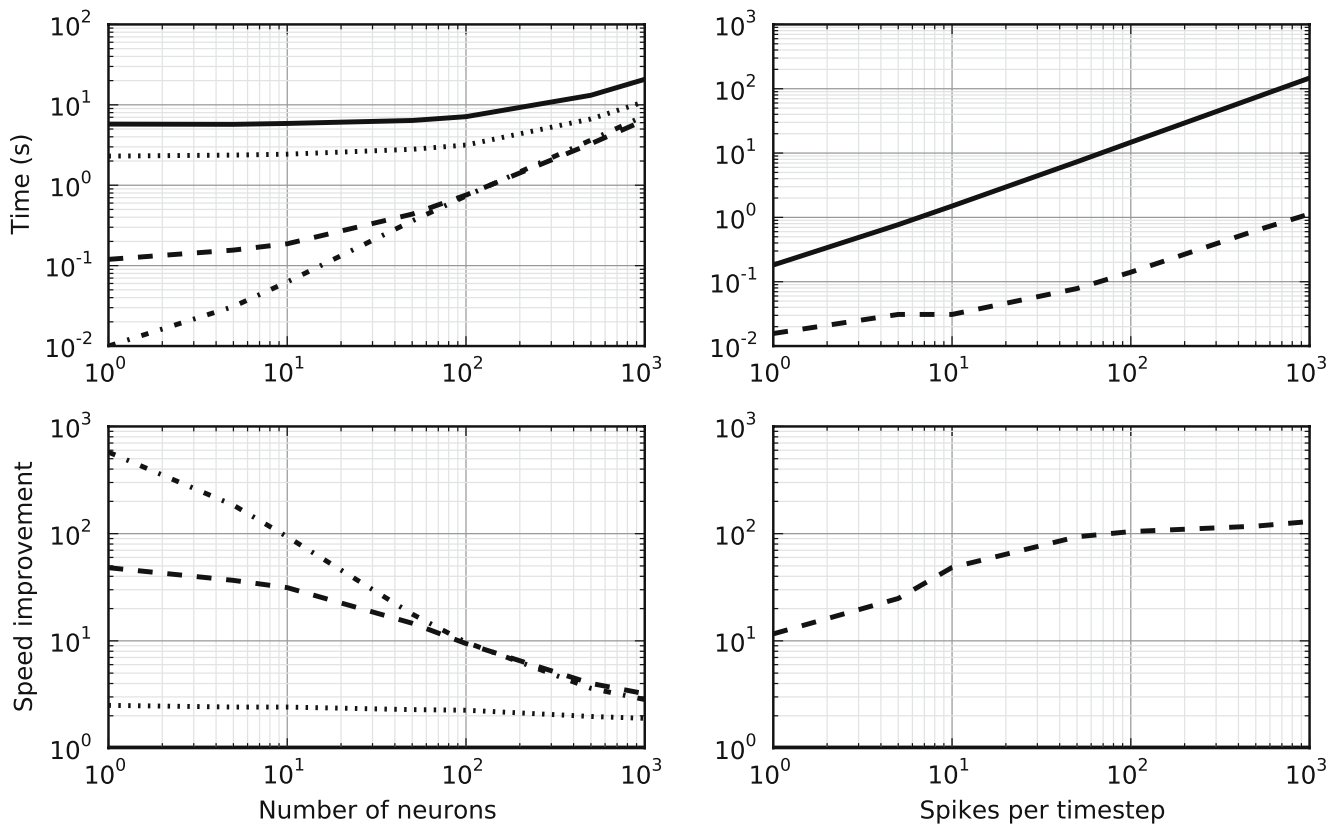


Fig. 3 *Left panels:* Performance for numerical integration in Brian without code generation (*thick line*), Brian with Python code generation (*dotted*), Brian with C++ code generation (*dashed*) and hand-coded C++ (*dash-dot*). The *upper plot* is the absolute time for a simulation of a number of Hodgkin–Huxley neurons. The *lower plot* is the speed improvement compared to a standard Brian run. The neurons were integrated with the

exponential Euler method, but the graphs are qualitatively similar for other integration schemes. *Right panels:* Performance for STDP in Brian without code generation (*thick line*), and with code generation (*dashed*). The *upper figure* is absolute times and the *lower figure* is the speed improvement of code generation compared to standard Brian

and in the case of large networks with complicated differential equations such as Hodgkin–Huxley type neurons thanks to the automatic simplification (around twice as fast). More important than that though will be the ability to run code on the GPU. In an earlier paper, using the Euler method on relatively simple differential equations, we showed that Brian using a simpler form of code generation on the GPU (no symbolic algebra) could give speed improvements of 60–80 times (Rossant et al. 2010).

We also compared the performance in the case of STDP. In this case, we compared a standard version of Brian against a version using C++ code generation. The standard version of Brian uses some static C++ code, but requires $O(n)$ Python operations for n spikes. The code generation version only requires $O(1)$ Python operations. The results show that the improvement increases as the number of spikes per timestep increases. To give an idea of a reasonable number of spikes per

timestep, a neuron firing at 50 Hz simulated with a dt of 0.1 ms would fire around 0.005 spikes per timestep. So for a small network of 1,000 such neurons, we would expect around 5 spikes per timestep, and for a large network of 100,000 such neurons, we would expect around 500 spikes per timestep. In this case, we get a speed improvement of around 100 times.

Future Work and Discussion

The most important extension of this work for the Brian simulator is the development of GPU propagation data structures and algorithms, and integration with the code generation framework. This will allow Brian to run all user defined models on the GPU as simply as on the CPU, providing potentially enormous speed improvements for users. Indeed, we demonstrated a 60–80 times speed improvement in the case of

simulations of models without propagation in Rossant et al. (2010), and Nageswaran et al. (2009) showed a 26 times speed improvement over CPU code in simulating a network with delays and STDP. GPU support for Brian is currently planned for the 2.0 release.

We are also working to extend the role and scope of symbolic analysis in Brian. At the moment, equations have to be specified in the form $d\mathbf{x}/dt = f(\mathbf{x})$, and not, for example, as $\tau dV/dt = -V$, and differential equations are restricted to being first order. More general forms could be supported with suitable symbolic analysis to reduce them to this form. In addition, at the moment Brian automatically detects if equations are linear and uses an exact solver in that case, but for nonlinear equations the user has to specify the solver by hand. In certain cases, the best solver could be determined by symbolic analysis (for example, if the differential equation was of Hodgkin–Huxley type the exponential Euler solver could be selected automatically). Finally, although most biophysically plausible properties can be reduced to systems of first order differential equations, it can be more natural to express them in other ways. For example, an α -synapse gives a current or conductance of the form $g(t) = (t/\tau)e^{1-t/\tau}$. This can be reduced to the system of ODEs $dx/dt = (y - x)/\tau$ and $dy/dt = -y/\tau$. Symbolic analysis could be used to perform these reductions automatically. In general, symbolic analysis can be used to further the goal of having the user specify their model at the highest level possible in standard mathematical notation (so that there is no particular syntax to learn), but allow for efficient simulation.

More speculatively, it may be possible using this approach to have models specified at a high level running faster than code written in a low level language. This is possible because the simulator could analyse the performance of the simulation as it runs, and adapt the code dynamically in response. These techniques have already been used for making high-level interpreted languages run as fast or even faster than low-level compiled languages. Java Just-In-Time (JIT) interpreters can analyse the code as it runs and generate machine code at run-time for key portions of the code that are run repeatedly (typically inner loops). Consequently, Java code can now run close to as fast, and in some cases actually faster, than generic C++ using the standard template library (Bull et al. 2001). The Psyco package (Rigo 2004) applies JIT techniques to the Python language. Speed improvements can be as large as 100 times in some cases, although code typically still runs considerably slower than equivalent C++ code. Development has now shifted to the PyPy project (Ancona et al. 2007). Runtime analysis and tuning of code is likely to be particularly important in the case

of GPU code, as different GPUs have very different characteristics (number of cores, memory bandwidth, etc.) and minor changes in the code can lead to much larger performance differences than in the case of CPU code (Klöckner et al. 2009).

Discussion

Scientists working in the field of computational neuroscience require the use of cutting edge computational hardware, but they should not be required to be experts in this technology. In principle, neural network simulation software should solve this problem. In practice, because they require more flexibility than the software easily allows, many scientists choose to write their own code by hand, usually in either C/C++, which is fast but time consuming to write, or in Matlab, which is slower but simpler. This has some unfortunate side effects. Firstly, writing code by hand is time consuming, and a drain on scientists' time. Secondly, it restricts the practice of computational neuroscience research to those with the technical skills to write their own software by hand. Thirdly, hand written code is more difficult for other scientists to analyse, is more likely to contain subtle bugs compared to simulations written with dedicated software, and may lead to results which are not reproducible.

We have shown that the techniques of symbolic analysis combined with run-time code generation can be used to write neural network simulation software that is both user-friendly, making it possible to write model definitions at a high level, and computationally efficient. This high level representation can be in many different forms, including standardised declarative languages such as CellML and NeuroML (discussed in Section “Implementing XML Models”), or explicit mathematical equations in standard form (as in Brian). The appropriate high level description may depend on the context. In biophysical models, declarative languages specifying the model in terms of ion channels, reaction networks, and so forth are probably the most appropriate, whereas if reduced models of integrate-and-fire type are being used, a specification based on equations would be preferable. In both cases, a high level specification of the model improves the reproducibility and facilitates the verification of the model (although in neither case does it entirely solve the problem).

We demonstrated these techniques primarily with the Brian simulator, however they are quite general and could be used in other software. Indeed, we make use of freely available Python packages such as Weave, PyCUDA and SymPy.

Information Sharing Statement

Brian is an open source software package that can be downloaded from <http://www.briansimulator.org>. The complete source code is available online at <http://neuralensemble.org/trac/brian>.

Acknowledgements The author would like to thank Romain Brette, Cyrille Rossant and Bertrand Fontaine for their work on Brian, testing of code generation, and helpful comments on the manuscript. This work was partially supported by the European Research Council (ERC StG 240132).

References

- Ancona, D., Ancona, M., Cuni, A., & Matsakis, N. D. (2007). RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages* (pp. 53–64). Montreal, Quebec, Canada: ACM.
- Bower, J. M., & Beeman, D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the General NEural Simulation System* (2nd ed.). New York: Springer-Verlag.
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23, 349–98.
- Bull, J. M., Smith, L. A., Pottage, L., & Freeman, R. (2001). Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande* (pp. 97–105). Palo Alto, California: ACM.
- Carnevale, N. T., & Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press.
- Garny, A., Nickerson, D. P., Cooper, J., dos Santos, R. W., Miller, A. K., McKeever, S., et al. (2008). CellML and associated tools and techniques. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences*, 366(1878), 3017–3043. PMID: 18579471.
- Gewaltig, O., & Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia*, 2(4), 1430.
- Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput Biol*, 6(6), e1000815.
- Goodman, D., & Brette, R. (2008). Brian: A simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2, 5.
- Goodman, D. F. M., & Brette, R. (2009). The Brian simulator. *Frontiers in Neuroscience*, 3(2), 192–197.
- Hansel, D., Mato, G., Meunier, C., & Neltner, L. (1998). On numerical simulations of Integrate-and-Fire neural networks. *Neural Computation*, 10(2), 467–483.
- Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., et al. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM transactions on mathematical software*, 31(3), 363–396.
- Hines, M. L., & Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation* 12(5), 995–1007.
- Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4), 500–544. PMID: 12991237.
- Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., et al. (2003). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4), 524–531.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001–2005). SciPy: Open source scientific tools for Python. <http://www.scipy.org/>.
- Klößner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2009). PyCUDA: GPU Run-Time code generation for High-Performance computing. *0911.3456*.
- Kootsey, J. M., Kohn, M. C., Feezor, M. D., Mitchell, G. R., & Fletcher, P. R. (1986). SCoP: An interactive simulation control program for micro- and minicomputers. *Bulletin of Mathematical Biology*, 48(3–4), 427–441.
- MacGregor, R. J. (1987). *Neural and Brain Modeling*. Academic Press.
- Miller, A., Marsh, J., Reeve, A., Garny, A., Britten, R., Halstead, M., et al. (2010). An overview of the CellML API and its implementation. *BMC Bioinformatics*, 11(1), 178.
- Morrison, A., Straube, S., Plesser, H. E., & Diesmann, M. (2007). Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural Computation*, 19(1), 47–79. PMID: 17134317.
- Morse, T. (2007). Model sharing in computational neuroscience. *Scholarpedia*, 2(4), 3036.
- Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., & Veidenbaum, A. (2009). Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *Proceedings of the 2009 international joint conference on neural networks* (pp. 3201–3208). Atlanta, USA: IEEE.
- NVIDIA (2009). CUDA programming guide 2.3.
- Oliphant, T. (2006). *Guide to NumPy*. USA: Trelgol Publishing.
- Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science and Engineering*, 9(3), 10–20.
- Rigo, A. (2004). Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation* (pp. 15–26). Verona, Italy: ACM.
- Rossant, C., Goodman, D. F. M., Platkiewicz, J., & Brette, R. (2010). Automatic fitting of spiking neuron models to electrophysiological recordings. *Frontiers in Neuroinformatics*. doi:10.3389/neuro.11.002.2010.
- Rotter, S., & M. Diesmann (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological Cybernetics*, 81(5–6), 381–402. PMID: 10592015.
- Song, S., Miller, K. D., & Abbott, L. F. (2000). Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3, 919–26.
- SymPy Development Team (2009). *SymPy: Python library for symbolic mathematics*.