Adaptive step ODE algorithms for the 3D simulation of electric heart activity with graphics processing units

V. M. Garcia-Molla[a,*], A. Liberos[b], A. Vidal[a], M. S. Guillem[b], J. Millet[b], Alberto González[c], F. J. Martínez-Zaldívar[c], A. M. Climent[b]

[a]*Department of Information Systems and Computing, Universitat Politècnica de València, Camino de Vera s/n 46022 Valencia SPAIN.*
[b]*ITACA Institute, Universitat Politècnica de València, Camino de Vera s/n 46022 Valencia SPAIN*
[c]*ITEAM Institute, Universitat Politècnica de València, Camino de Vera s/n 46022 Valencia SPAIN.*

Abstract

In this paper we studied the implementation and performance of adaptive step methods for large systems of ordinary differential equations systems in Graphics Processing Units, focusing on the simulation of thre–dimensional electric cardiac activity. The Rush-Larsen method was applied in all of the implemented solvers to improve efficiency. We compared the adaptive methods with the fixed step methods, and we found that the fixed step methods can be faster while the adaptive step methods are better in terms of accuracy and robustness.

Keywords: Adaptive ODE solvers, Embedded Runge–Kutta methods, GPU computing, Cardiac electrophysiology, Rush-Larsen method

1. Introduction

Mathematical models of cardiac electrical activity are important tools for evaluating heart conditions and pathological mechanisms [1, 2, 3]. Electrophysiological models of cardiac tissues describe how ion channels control the transmembrane potential of each cell and how this cellular action potential propagates across the heart. The electrical activity of the heart is usually simulated by modeling the ionic currents of each cardiac cell and the interactions with its neighboring cells. Depending on the model chosen,

*Corresponding author. Phone Number +34 963879723
Email addresses: `vmgarcia@dsic.upv.es` (V. M. Garcia-Molla), `allimas1987@gmail.com` (A. Liberos), `avidal@dsic.upv.es` (A. Vidal), `mariaguillem@gmail.com` (M. S. Guillem), `jmillet@eln.upv.es` (J. Millet), `agonzal@dcom.upv.es` (Alberto González), `fjmartin@dcom.upv.es` (F. J. Martínez-Zaldívar), `acliment@itaca.upv.es` (A. M. Climent)

each cell is modeled with 15-40 ordinary time-dependent differential equations (ODEs). Therefore, a realistic atrial simulation may require thousands or even millions of cells; hence, the final number of coupled ODEs to be solved could be tens or hundreds of millions. The required computing time is significant: an accurate simulation of a few seconds of cardiac activity requires hours or even days of computations in standard computers.

In the last few years, Graphics Processing Units (GPU) have been used as computing platforms to carry out these simulations. These devices, which were initially designed to perform the calculations needed to display graphic content on a computer screen, have evolved to become very powerful and cheap tools for performing heavy computations. Many studies have been devoted to describing electric cardiac simulations using GPUs [4, 5, 6, 7, 8].

These GPU simulations were performed using fixed time step methods. These methods are usually preferred for electric cardiac simulations because fixed time step methods are simple to implement and are easy to couple with other simultaneous simulations. More sophisticated methods are based on variable, adaptive step size. In these formulations, the step size is adjusted to the desired accuracy, and the time step can become very large or very small depending on the changes in the solution. This procedure increases the accuracy during critical steps and reduces the simulation time during non-critical steps [9]. Adaptive step formulations are regularly used in most science fields, and most black box ODE solvers for CPUs rely on adaptive step methods. We are not aware of any publicly available variable step ODE solvers for GPUs. There exist also adaptive spatio-temporal methods [10, 11, 12]. These methods deal with the characteristic wavefronts that appear in cardiac simulations by adapting the spatial discretization in regions with steep gradients, and adapting accordingly the time step only in these regions. These methods can be built on top of an underlying fixed step strategy (like in [10]) or on top of an adaptive step strategy [12]. Spatio-temporal methods are very promising, although at present there are no GPU implementations available.

An important feature of electric cardiac models is that some of the equations have a special form that can be used to improve the performance of the ODE solvers. Rush and Larsen proposed a method in [13] that can be implemented for most cell models. The Rush–Larsen method has been very successful in this field, and several studies [14, 15] have shown that combining this technique with a standard explicit method substantially increases the stability of the method.

The main purpose of this paper is to discuss the implementation and performance of fixed step and adaptive step ODE solvers in GPUs that are adapted for electric cardiac simulation. To completely explore the advantages and disadvantages of the different implementations, we have performed several comparisons related to the electric cardiac simulation problem: multicore CPU solvers vs. GPU solvers, GPU solvers with and without the Rush-Larsen technique, and GPU fixed step solvers vs. GPU adaptive solvers. The methods have been implemented using the CUDA programming environment [16] from NVIDIA Corporation; we will assume some familiarity of the reader with this environment to avoid lengthy descriptions.

As mentioned above, there is no known adaptive ODE software written for GPUs. One of the main goals of this paper is fill this gap by providing a detailed description of our GPU implementations, so that any scientist can reproduce our solvers, for cardiac simulation or for the solution of any other large ODE system.

All of the solvers have been implemented over the Courtemanche atrial model [17] (although the cell model can be changed without great effort). We have selected a relatively large spherical structure with 163842 cells as our main test case.

The organization of this paper is as follows: first, we describe the Courtemanche atrial cell model, and the basic numerical methods considered. Then, we describe our implementations, with special emphasis on the GPU programming of the adaptive solvers. Finally, we describe and discuss the results from several methods in terms of accuracy and execution time.

2. Mathematical Cell Model

The Courtemanche model for the time evolution of a single cell can (like other cardiac models) be written in terms of the transmembrane voltage V, of the vector of ionic concentrations $\mathbf{X} = x_i, i = 1, .., 5$ and the vector of gating variables $\mathbf{W} = w_i, i = 1, ..., 15$ as follows:

$$\frac{dV}{dt} = I\left(t, V, \mathbf{X}, \mathbf{W}\right), \tag{1}$$

$$\frac{dx_j}{dt} = g_j(V, \mathbf{X}, \mathbf{W}),\ j = 1, ..., 5, \tag{2}$$

$$\frac{dw_i}{dt} = a_i(V)w_i + b_i(V),\ i = 1, ..., 15, \tag{3}$$

with the initial conditions $V_0, \mathbf{X_0}, \mathbf{W_0}$. For a single cell, the equation governing the transmembrane voltage can be written as:

$$\frac{dV}{dt} = I\left(t, V, \mathbf{X}, \mathbf{W}\right) = \frac{I_{stim} - I_{ion}}{C_m}, \tag{4}$$

where C_m is the membrane capacity, I_{stim} is an applied stimulus and I_{ion} is the ionic current, which is computed as:

$$I_{ion} = \sum_{i=1}^{12} IC_i\left(t, V, \mathbf{X}, \mathbf{W}\right). \tag{5}$$

where IC_i is the total transmembrane current carried by ion i. Several of these currents are controlled by the gating variables $[w_1, w_2, ..., w_{15}] = \mathbf{W}$. These variables control the opening and closing of the ion channel i through the associated ODE (3). All the expressions are described in [17],and a small description of the variables and equations is given in Appendix A. The whole model has 21 variables (the transmembrane voltage, 15 gating variables and 5 ionic concentrations) and 21 differential equations.

If a system with multiple cells ($NumCells$, numbered from 0 to $NumCells - 1$) is considered, then equation (1) is modified to include the effect of the neighboring cells:

$$\frac{\partial V}{\partial t} = \frac{I_{stim} - I_{ion}}{C_m} + \nabla \cdot \left(D \nabla V\right), \tag{6}$$

where D describes the diffusion of the voltage through the medium. After discretization of the spatial derivatives for an isotropic medium, equation (1) for the k-th cell is described by the following ODE:

$$\frac{dV_k}{dt} = \frac{I_{stim,k} - I_{ion,k}}{C_{m,k}} - D \sum_j \frac{V_k - V_j}{d_{i,k}^2}, \tag{7}$$

where $d_{i,k}$ is the distance between the neighboring cells i and j and the sum is performed over the cells neighboring the k-th cell. We denote the data vector for the

k-th cell as $\mathbf{y_k} = [V_k; \mathbf{X_k}; \mathbf{W_k}]$, and the vector of the transmembrane voltages for all cells as $\mathbf{V}$. Then we wrote the system function f_k for the k-th cell as:

$$f_k(t, \mathbf{Y}) = f_k\left(t, \mathbf{V}, \mathbf{X_k}, \mathbf{W_k}\right) = \begin{cases} \frac{I_{stim,k} - I_{ion,k}}{C_{m,k}} - D \sum_j \frac{V_k - V_j}{d_{i,k}^2}, \\ a_i(V_k) w_{i,k} + b_i(V_k), i = 1, ..., 15, \\ g_j(V_k, \mathbf{X_k}, \mathbf{W_k}), j = 1, ..., 5. \end{cases} \quad (8)$$

Stacking together the data of all cells $\mathbf{Y} = [\mathbf{y_0}; \mathbf{y_1}; ...; \mathbf{y_{NumCells-1}}]$ and the system functions of all cells $F = [f_0; f_1; ...; f_{NumCells-1}]$, the whole system can be written as an initial value problem,

$$\frac{d\mathbf{Y}}{dt} = F(t, \mathbf{Y}). \quad (9)$$

The evaluation of the system function F is crucial in all of the algorithms that are described below. In many cases, the evaluation of the system function must be carried out cell by cell. To emphasize this point, we will switch to the notation $f_k, k = 0, 1, ..., NumCells - 1$ when this happens. Therefore, note that the evaluation of $F(t, \mathbf{Y})$ is the same operation as the evaluation of $f_k(t, \mathbf{Y}), k = 0, 1, ..., NumCells - 1$.

3. Numerical Methods

In this section we detail all of the implemented numerical methods, giving as much details as possible so that the reader can reproduce our results. We proceed in order of increasing complexity, because the optimizations that were devised for low complexity methods were applied to more complex methods.

3.1. The Forward Euler method

The Forward Euler (FE) method is the simplest method for solving initial value ODE systems such as (9). After selection of a time step h, and given the initial solution $\mathbf{Y^0}$ at time $t = t_0$, an approximate solution is obtained at time $t = t_0 + h \cdot NumSteps$ is obtained through the simple loop in Algorithm 1.

Algorithm 1 Forward Euler

```
for $j = 1 \to NumSteps$ do
    $\mathbf{Y^{j+1}} = \mathbf{Y^j} + h \cdot F\left(t_j, \mathbf{Y^j}\right)$
end for
```

This method has well known stability and accuracy limits, and, if CPU implementations are considered, there are better methods. However, in the present situation we are trying to solve a cardiac modeling problem in a GPU. This method is appropriate for parallel GPU implementations and achieves reasonable accuracy and speed when combined with the Rush-Larsen technique (described below). Moreover, the accuracy usually requested for cardiac modeling is quite gross (a 1% or even 5% error is considered acceptable [18]), which makes this method competitive in this particular area.

3.2. Rush-Larsen method

The Rush-Larsen (RL) method, proposed in [13], was devised specifically for cell models, and has been well studied [14, 15, 18, 19]. This method takes advantage of the special form of the equations governing the gating variables, which can be written as:

$$\frac{\partial w_i}{\partial t} = a_i(V)w_i + b_i(V) \tag{10}$$

for a general gating variable w_i. If the transmembrane constant V is constant over a time interval h (and, equivalently, the expressions $a_i(V)$ and $b_i(V)$ are constant over that interval) then the value of the dependent variable after the step can be determined exactly. If the value of w_i at the beginning of the interval is w_i^0, the value after taking a step of length h is:

$$w_i^1 = e^{a_i(V)h}\left(w_i^0 + \frac{b_i(V)}{a_i(V)}\right) - \frac{b_i(V)}{a_i(V)}. \tag{11}$$

Rush and Larsen proposed to split the ODE system, solving the equations for the gating variables through eq(11) and using forward Euler for the rest of the equations. We call this combination the RLFE method. For notation convenience, we split the full data vector $\mathbf{Y}$ into gating and non-gating variables, denoted respectively as $\mathbf{Y_g}$ and $\mathbf{Y_{ng}}$, and split accordingly the system function F into its gating and non-gating parts F_g and F_{ng}. Then, the RLFE can be written as shown in Algorithm 2:

Algorithm 2 Forward Euler Rush Larsen

```
for $j = 1 \to NumSteps$ do
    $\mathbf{Y_{ng}^{j+1}} = \mathbf{Y_{ng}^{j}} + h \cdot F_{ng}\left(t_j, \mathbf{Y^j}\right)$
    Apply (11) to $\mathbf{Y_g^j}$ with step $h$ to obtain $\mathbf{Y_g^{j+1}}$
end for
```

The success of this method depends on the fact that many cell models (including the Courtemanche model) are stiff due to the gating equations, and become non-stiff when the RL method is applied to the equations governing the gating variables. This allows much larger time steps.

3.3. Other fixed-step methods

We implemented and tested other fixed step methods, such as the explicit midpoint method. This is a second order explicit method:

$$\begin{aligned}\hat{\mathbf{Y}} &= \mathbf{Y^j} + \tfrac{h}{2}F\left(t_j, \mathbf{Y^j}\right)\\ \mathbf{Y^{j+1}} &= \mathbf{Y^j} + hF\left(t_j + \tfrac{h}{2}, \hat{\mathbf{Y}}\right)\end{aligned} \tag{12}$$

As mentioned above, the time steps that can be used are much larger if the gating equations are handled with the RL technique. However, the RL technique applied to the full time step h is an $O(h)$ method. To obtain an accuracy closer to that of the selected method, we used the RL technique in the same internal steps that the method employed (this is analogous to the technique proposed in [14]). For example, we combined the explicit midpoint method with the RL technique in the following 4 steps:

1. Compute $\hat{\mathbf{Y}}_{\mathbf{ng}} = \mathbf{Y^j_{ng}} + \frac{h}{2}F_{ng}\left(t_j, \mathbf{Y^j}\right)$.
2. Apply RL with time step $\frac{h}{2}$ to advance from $\mathbf{Y^j_g}$ to $\hat{\mathbf{Y}}_g$.
3. Compute $\mathbf{Y^{j+1}_{ng}} = \mathbf{Y^j_{ng}} + hF_{ng}\left(t_j + \frac{h}{2}, \hat{\mathbf{Y}}\right)$.
4. Apply RL with time step $\frac{h}{2}$ to advance from $\hat{\mathbf{Y}}_{\mathbf{g}}$ to $\mathbf{Y^{j+1}_g}$.

This technique can be applied to other Runge-Kutta methods, but this operation is not appropriate for all Runge-Kutta methods. For example, the trapezoidal method is another second order method, described by the following steps:

$$\begin{aligned}\hat{\mathbf{Y}} &= \mathbf{Y^j} + hF\left(t_j, \mathbf{Y^j}\right),\\ \mathbf{Y^{j+1}} &= \mathbf{Y^j} + \tfrac{h}{2}F\left(t_{j+1}, \hat{\mathbf{Y}}\right) + \tfrac{h}{2}F\left(t_j, \mathbf{Y^j}\right).\end{aligned} \tag{13}$$

In this method, the system function F is evaluated only at the beginning and at the end of the time interval. Therefore, the combination of this method with RL can be accomplished by applying RL to the complete time step h, (which would keep the accuracy of the gating variables as first order) or by performing an extra function evaluation in the middle of the interval (which would slow the whole method). Other methods, (such as the classical 4th order Runge-Kutta method) have similar drawbacks.

Higher order methods require more memory to hold the intermediate results in a step, which can be a crucial issue for very large models.

3.4. Adaptive time step methods

The problem with using a fixed step in algorithms such as FE or RLFE is that the error in each step depends on the step h, and decreases with h. If a given accuracy per step is desired while using a fixed step formulation, then a small step must be used throughout the whole simulation. The key to obtaining a good general accuracy without using tiny steps is to correctly estimate the error made after computing each step of the computed solution.

There are several ways to estimate this error. In this paper, we have used the Embedded Runge-Kutta methods [9, 20]. These are associated pairs of methods of different accuracies, where the low-order method has an accuracy of order P, (when applied over the variable $\mathbf{Y^{old}}$, taking a step h gives the approximate solution $\mathbf{Y^{low}}$), and the high-order method has an accuracy of order $P + 1$ (when applied over the variable $\mathbf{Y^{old}}$, taking a step h gives the approximate solution $\mathbf{Y^{high}}$). Many such pairs exist.

The procedure (slightly simplified) to compute the new time step using Embedded Runge Kutta pairs is described here. Let ϵ be the maximum error that we are ready to admit in each step. Usually, this error is established by setting two parameters, relative tolerance (`rt`) and absolute tolerance (`at`), so that the admissible error must be smaller than $\epsilon = at + rt\left|\mathbf{Y^{old}}\right|$. If both methods are used to take a single step, then the difference between the solutions obtained by the two methods $\left(\|\mathbf{Y^{high}} - \mathbf{Y^{low}}\|\right)$ can be used to estimate the error made by the lower order method. If the error is not acceptable $\left(\|\mathbf{Y^{high}} - \mathbf{Y^{low}}\| > \epsilon\right)$, then the step is rejected and a new step is computed, as shown in [9]. For some constant c the error fulfills $\left(\|\mathbf{Y^{high}} - \mathbf{Y^{low}}\|\right) \approx c \cdot h^{p+1}$ so that the new step $\hat{h}$ can be chosen to satisfy:

$$\left(\frac{\hat{h}}{h}\right)^{P+1} \|\mathbf{Y^{high}} - \mathbf{Y^{low}}\| \approx frac \cdot \epsilon, \tag{14}$$

where $frac$ is a real number smaller than 1, usually taken as 0.9 for safety. Therefore, the new time step $\hat{h}$ is computed as:

$$\hat{h} = h \left(\frac{frac \cdot \epsilon}{\|\mathbf{Y^{high}} - \mathbf{Y^{low}}\|} \right)^{\frac{1}{P+1}}. \tag{15}$$

If the error is acceptable $\left(\|\mathbf{Y^{high}} - \mathbf{Y^{low}}\| \leq \epsilon\right)$ a new time step length (possibly longer than the one taken), is calculated using (15), and a new time step is taken. For an ODE system, the procedure described above to compute the time step length is applied to the largest error computed for all the dependent variables.

More detailed descriptions can be found in [9, 20]. All high quality CPU codes for solving ODE systems use adaptive time step, because the codes provide accuracy and efficiency with this mechanism [21].

It is generally acknowledged in the numerical ODE solving literature that low order methods are more efficient than high order methods when the tolerances required for the solution are not too tight [22, 23]. However, cardiac simulations usually require accuracies of approximately 1 to 5% (quite gross for ODE solution standards). Therefore, we chose to use low order methods. We implemented three adaptive explicit methods:

1. The lowest order Runge-Kutta pair, a 2-1 order method formed by the trapezoidal method and the forward Euler method. [9]
2. The Bogacki-Shampine pair, a 3-2 order method with three stages implemented in the `ode23` MATLAB ODE solver [22, 23].
3. The well–known Runge-Kutta-Fehlberg a 5-4 order method with 6 stages [9, 20].

The error estimate $\left(\|\mathbf{Y^{high}} - \mathbf{Y^{low}}\|\right)$ is asymptotically correct for the lower order method. However, it is common to select the high order method as solution. This is known as *local extrapolation* and we have applied this approach. The basic adaptive step size algorithm is shown in Algorithm 3.

The line $\texttt{swap}\left(\mathbf{Y^{high}}, \mathbf{Y^{old}}\right)$ can be implemented by copying the contents of each vector on the other but it is much more convenient to exchange the pointers if both vectors are accessed through pointers.

Algorithm 3 Generic Embedded Runge-Kutta adaptive step algorithm

```
Input $t_{simul}, t_{end}, h, \mathbf{Y^{old}}$
while $t_{simul} \leq t_{end}$ do
    Compute $\mathbf{Y^{high}}$ and $\mathbf{Y^{low}}$ from state $\mathbf{Y^{old}}$ using time step $h$
    Estimate maximum error $maxerr = \|\mathbf{Y^{high}} - \mathbf{Y^{low}}\|$
    if $maxerr < \epsilon$ then
        $t_{simul} = t_{simul} + h$
        Compute new h using (15)
        swap$(\mathbf{Y^{high}}, \mathbf{Y^{old}})$
    else
        Compute new h using (15)
    end if
end while
```

4. Implementations

4.1. CPU implementation of ODE solvers

The programming of ODE solvers is a relatively simple matter in any programming language. There are hundreds of available implementations that have been described in many papers and books (see [9] for details). A subroutine that computes the system function F is needed. This routine must evaluate the function systems f_k for all cells.

We are interested in high performance architectures, such as multicore CPUs and GPUs, and we want to highlight how this parallelism is introduced, to obtain efficient implementations. Any method for initial value problems has a time-advancing loop similar to Algorithm 1. It is clear that the iterations of such loop cannot be parallelized, because the result of each iteration depends on the previous iteration. Instead, we can obtain parallelism by evaluating the system function F for many cells at the same time. Simply rewriting the loop in Algorithm 1 as

```
for $j = 1 \to NumSteps$ do
    for $k = 0 \to NumCells - 1$ do
        $\mathbf{y_k^{j+1}} = \mathbf{y_k^j} + h \cdot f_k\left(t_j, \mathbf{y_k^j}\right)$
    end for
end for
```

clarifies that the k loop can be parallelized, because the system function f_k can be evaluated for multiple cells at the same time. In a shared memory computer with multiple cores, using a simple OpenMP pragma [24] is enough to parallelize the k loop.

4.2. Forward Euler: GPU implementation

Here, we describe some details of the CUDA implementations of the fixed step explicit Euler method. Most of these implementation details were also used for the adaptive versions described below. All of our GPU implementations have been written using single precision.

A CUDA program is constructed by pieces of code, called kernels [16]. Each kernel is invoked from the main program, to assign a task to the GPU. When a kernel is sent to the GPU, many instances of the kernel are executed in parallel in the GPU. Each of these instances is executed by a thread, and each thread runs in one of the many microprocessors or cores of the GPU. These threads are organized into blocks of threads, which can be visualized as "teams" of threads that are executed together. In the call to the kernel, the programmer must specify as parameter of the kernel how many blocks ($NumBlocks$) and how many threads per block ($NumThreadsperBlock$) must be used to execute this kernel.

All of our CUDA solvers have been structured so that, like in the CPU version, the parallelism is obtained by simultaneously performing the computations for many cells. The computations for a given cell are carried out by a single CUDA thread. If the number of threads is larger than the number of cells, then each thread will take care of a single cell. If the number of cells is larger than the number of threads, which is usually the case, then several cells are assigned to each thread as follows: Suppose that there are $NumCells$ cells and $NumThreads$ threads ($NumThreads = NumBlocks \cdot NumThreadsperBlock$), with $NumCells > NumThreads$. Each thread is identified by an integer number Tid, $0 \leq Tid < NumThreads$. Then, in a given step, thread Tid will start processing cell Tid. When finished with cell Tid, thread Tid will process cell $NumThreads + Tid$, then cell $2 \cdot NumThreads + Tid$ and so on, until all the cells have been processed.

The structure of the main loop (executed on the CPU) would be like the structure in Algorithm 4, and each kernel (that is executed on the GPU) would resemble those in Algorithm 5.

All the data needed for the computations and the results are kept in the global memory of the GPU so that all of the threads can access the necessary data. An important feature of this problem is that the subroutine that evaluates the system function f_k for each cell is quite complex and uses many variables; for efficiency, these variables must

Algorithm 4 Main Loop for Forward Euler

```
for $j = 1 \to NumSteps$ do
    $kernel\_FE <<< NumThreadsperBlock, NumBlocks >>> (t, \mathbf{Y^{in}}, \mathbf{Y^{out}})$
    swap $\left(\mathbf{Y^{in}}, \mathbf{Y^{out}}\right)$
end for
```

Algorithm 5 kernel_FE

```
$kernel\_FE(t, \mathbf{Y^{in}}, \mathbf{Y^{out}})$
$k = Tid$
while $k < NumCells$ do
    $\mathbf{y_k^{out}} = \mathbf{y_k^{in}} + h \cdot f_k\left(t, \mathbf{y_k^{in}}\right)$
    $k = k + NumThreads$
end while
```

be stored in "registers". Generally speaking, CUDA kernels should optimally utilize "shared memory", which is a fast memory that is available to all the threads in a block [16]. However, the amount of available shared memory depends on the number of used registers. Because every thread needs many registers, there is a strong limit on the available shared memory. Because of this limitation, we did not use shared memory in our kernels. Instead, we focused on achieving good patterns of memory access and limiting the number of registers used by our code.

If a given computation requires that a previous computation has finished with all of the cells, then the two computations must be written in different kernels because (as long as a single stream is used, which is our case) it is a safe way to obtain global synchronization of all of the threads.

The data structure also has a big influence on the performance. A simpler way to organize the data would be to create a data structure for each cell that contains the 21 variables for each cell, and then create an array of this new data type, with `NumCells` elements. Using a C-like syntax, the data structure can be written as:

$$\textit{struct celldata} \left\{ \begin{array}{l} \text{float } V; \\ \text{float } w_1; \\ \text{float } w_2; \\ \vdots \\ \text{float } x_5; \end{array} \right\} ; \textit{struct celldata bigdata}\left[NumCells\right]; \qquad (16)$$

This is an "Array of Structures" data layout and is the way the data has been organized in the CPU version.

However, memory access in GPUs is more efficient if all of the threads access the data in a regular form, called “coalesced access” [16]. To obtain this type of access, it is more efficient to use a “Structure of Arrays” data layout [25].

$$struct \quad celldata \left\{ \begin{array}{l} \text{float } V\left[NumCells\right]; \\ \text{float } w_1\left[NumCells\right] \\ \text{float } w_2\left[NumCells\right]; \\ \vdots \\ \text{float } x_5\left[NumCells\right]; \end{array} \right\} \tag{17}$$

The access to memory is further improved if the size of the arrays in the structure is a multiple of 32. Therefore, the first multiple of 32 greater than `NumCells` is selected(`NumCelAux`) and the data structure is then

$$struct \quad celldata \left\{ \begin{array}{l} \text{float } V\left[NumCelAux\right]; \\ \text{float } w_1\left[NumCelAux\right] \\ \text{float } w_2\left[NumCelAux\right]; \\ \vdots \\ \text{float } x_5\left[NumCelAux\right]; \end{array} \right\} \tag{18}$$

The extra cells are not used, which is a slight waste of memory, very small compared with the overall size of the system. Using the NVIDIA CUDA profiler [26], we confirmed that the only unaligned memory accesses are those related to the voltages of the neighboring cells, and the effect of this unaligned memory access is small.

The results (the cell voltages at the different times) must be sent back from the GPU to the CPU. These transfers are slow, and data transfer in every step would be inefficient. Instead, the results are sent every `Nsave` steps, where `Nsave` is a predefined parameter with a value that can be adjusted depending on the problem.

It would be conceptually simple to obtain a multi-GPU version by splitting the cells among the available GPUs, and programming the communications between the GPUs.

Finally, we remark that the choice of the step length for the fixed step algorithm can only be safely made through a trial and error procedure, or by using previous experience.

4.3. Implementations of RLFE method

The CPU and GPU implementations of the RLFE method are similar to the CPU and GPU FE implementations. The only difference is that there must be a routine, called `advance_gating`, which advances the value of the gating variables over a time step h, using formula (11). The routine that computes the system functions for each cell now only evaluates the system function for the transmembrane voltage V and for the ionic concentrations $\mathbf{X}$. The main loop would look like Algorithm 4, and the kernel now look like Algorithm 6.

Algorithm 6 kernel RLFE

```
  $kernel\_RLFE(t, \mathbf{Y^{in}}, \mathbf{Y^{out}})$
  $k = Tid$
  while $k < NumCells$ do
      $\mathbf{y^{out}_{k,ng}} = \mathbf{y^{in}_{k,ng}} + h \cdot f_{k,ng}\left(t, \mathbf{y^{in}_{k}}\right)$
      $\mathbf{y^{out}_{k,g}} = advance\_gating(\mathbf{y^{in}_{k}}, h)$
      $k = k + NumThreads$
  end while
```

All of the implementation details described above for the FE method apply to the RLFE method.

4.4. Implementations of higher order methods

The implementation for the CPU of the fixed step methods does not present any problem and is a simple programming exercise. The implementation for GPUs is not much more complex, but we have provided the main guidelines for the sake of reproducibility and completeness.

In any Runge-Kutta ODE method with an order ≥ 2 the system functions must be evaluated two or more times. Such evaluations are called stages. Usually, each function evaluation depends on the results of the previous stages. In our case, the computation of a stage must not start until the computation of the previous stage has finished *for all cells.* This creates a synchronization point after each stage. We have chosen to use different kernels for each stage, which guarantees the desired synchronization when a single stream is used.

Therefore, Algorithm 7 illustrates the main loop in the explicit midpoint method.

Algorithm 7 Main loop for MidPoint method

```
for $j = 1 \to NumSteps$ do
    $kernel\_midpoint1 <<< NumThreadsperBlock, NumBlocks >>> (t, \mathbf{Y^{in}}, \hat{\mathbf{Y}})$
    $kernel\_midpoint2 <<< NumThreadsperBlock, NumBlocks >>> (t, \hat{\mathbf{Y}}, \mathbf{Y^{out}})$
    swap $(\mathbf{Y^{in}}, \mathbf{Y^{out}})$
end for
```

If RL is not used, the kernel `kernel_midpoint1` and the kernel `kernel_midpoint2` would be similar to Algorithm 5; if RL is used, each one of these kernels would resemble Algorithm 6.

4.5. Implementations of adaptive methods

Again, the CPU implementation of adaptive methods has been thoroughly described elsewhere [9, 22]. We have described here only our GPU implementation. Adaptive solvers based on embedded Runge-Kutta methods rely necessarily on multiple stage methods, so that the GPU implementation would be similar to Algorithm 7, meaning that there should be a kernel for each stage. The only important details that still must be discussed are 1) the computation of the error between the two approximations, and 2) the decision about the validity of the step taken, with the computation of the new time step.

We structured these computations into two kernels. The first kernel (`kernel_error`) computes the maximum error for each block of threads, and stores this error in the array `ErrGlobal`, which has a position for each block. The second kernel(`kernel_reduction`) uses the errors in `ErrGlobal` to compute the maximum overall error, decides whether the step taken can be accepted and computes the new time step.

The computation of the maximum error between both approximations means that the errors of all of the variables must be compared to obtain a single real number result. In parallel computing this is called a reduction, and the final stages of the reduction must be carried out by a few (or only one) thread, or by the CPU. We followed the guidelines about CUDA reduction by Mark Harris [27].

We describe first the kernel `kernel_error`. We assume that the low order approximation $\mathbf{Y^{low}}$ and the high order approximation $\mathbf{Y^{high}}$ have been previously computed. In Algorithm 8, `NumBlocks` is the number of blocks with which this kernel is executed; `BlockDim1` is the number of threads in the block, in the kernel `kernel_error`;

`LocalTid` is the integer identifying the present thread in this block (so, $0 \leq LocalTid < BlockDim1$); `BlockId` is the integer identifying the block and `Tid` is the integer identifying the thread among all of the threads ($0 \leq Tid < numThreads$).

Algorithm 8 kernel_error

```
$kernel\_error(\mathbf{Y^{low}}, \mathbf{Y^{high}}, \mathbf{ErrGlobal})$
shared float $error[BlockDim1]$
$k = Tid$
$error[LocalTid] = 0.0$
while $k < NumCells$ do                         ▷ compute error of all cells in the Block
    $err = \|\mathbf{y_k^{low}} - \mathbf{y_k^{high}}\|$
    $error[LocalTid] = max(error[Tid], err)$
    $k = k + NumThreads$
end while
s=$\frac{BlockDim1}{2}$
while $s > 0$ do                                 ▷ compute max error in the Block
    if $LocalTid < s$ then
        $error[LocalTid] = max(error[LocalTid], error[LocalTid + s])$
    end if
    syncthreads()
    s=s/2
end while
if $LocalTid == 0$ then
    $\mathbf{ErrGlobal}[BlockId] = error[0]$
end if
```

First, the array `error` is declared, with as many positions as threads in each block (`BlockDim1`). Next, in the first while loop, each thread computes the maximum error for all its cells and stores the maximum error in the local position `error[LocalTid]`. Then, the maximum value in the array `error`, which will be the maximum error for this block of threads, is computed, using a technique described in [27].

Finally, the maximum error for this block is stored in the array `ErrGlobal`. When all the blocks have executed this kernel, the array `ErrGlobal`, which has `NumBlocks` positions, must keep the maximum errors of all of the blocks.

Here, we discuss the kernel `kernel_reduction`, described in Algorithm 9. This kernel is called with a single block and with number of threads `BlockDim2` equal to the number of blocks with which the kernel `kernel_reduction` was called, that is, `BlockDim2 = NumBlocks`.

First, the same technique used in `kernel_error` is used again to compute the overall maximum error. When that loop finishes, the maximum error for all of the variables is stored in `error[0]`. This is the desired error estimate used to check the validity of the step and to compute the time step.

It is unnecessary to use more than one thread to perform this computation, so that it is made by the thread 0. Alternatively, this single thread calculation could be carried out by the CPU. The CPU would make this computation faster, but some data traffic between the GPU and the CPU would be necessary. In our simulations, we ensured that the time spent in this kernel was less than 1% of the whole execution time.

Algorithm 9 kernel_reduction

```
$kernel\_reduction$(**ErrGlobal**, int $StepOk$, int CorrectSteps,float timestep)
shared float $error[blockDim2]$
$error[LocalTid] = ErrGlobal[LocalTid;$
s=$\frac{BlockDim2}{2}$
**while** $s > 0$; **do**
    **if** $LocalTid < s$ **then**
        $error[LocalTid] = max(error[LocalTid], error[LocalTid + s])$
    **end if**
    syncthreads()
    s=s/2;
**end while**
**if** $LocalTid == 0$ **then**
    **if** $error[0] > \epsilon$ **then**
        $StepOk$=0
        Compute new timestep using $error[0]$ and formula (15)
    **else**
        $StepOk$=1
        Compute new timestep using $error[0]$ and formula (15)
        CorrectSteps=CorrectSteps+1
    **end if**
**end if**
```

The conclusion of the main loop is carried out in the CPU. Because the time steps are variable, we do not know in advance how many time steps are necessary to reach the final time. This can be implemented in different forms. We have chosen to use the integer parameter `StepOk` to inform the CPU whether the taken step has been successful. If the step is successful, then the CPU interchanges the pointers associated to the "old" and "new" solutions. The computed time step is sent to the CPU, so that the main loop can compute the present simulation time, and verify whether the algorithm has reached the desired final time. This is shown in Algorithm 10.

Algorithm 10 Main loop for a generic k-stage Adaptive Runge-Kutta method

```
while $t_{simul} \leq t_{end}$ do
    $kernel\_stage1 <<< NumThreadsperBlock, NumBlocks >>> (t, \mathbf{Y^{in}}, ...)$
    $kernel\_stage2 <<< NumThreadsperBlock, NumBlocks >>> (...)$
    ...
    $kernel\_stagek <<< NumThreadsperBlock, NumBlocks >>> (..., \mathbf{Y^{low}}, \mathbf{Y^{high}})$
    $kernel\_error <<< NumThreadsperBlock, NumBlocks >>> (t, \mathbf{Y^{low}}, \mathbf{Y^{high}})$
    $kernel\_reduction <<< NumBlocks, 1 >>> (StepOk, CorrectSteps, h)$
    Copy $StepOk$ to host
    if $StepOk$==1 then
        swap $(\mathbf{Y^{in}}, \mathbf{Y^{high}})$
        Copy $h$ to host
        $t_{simul} = t_{simul} + h$
    end if
end while
```

5. Results

We have tested our solvers over a variety of one, two and three dimensional cases, with up to 500000 cells. Because the conclusions that can be drawn from these experiments are similar for all of them, we have presented the numerical results only for a representative case.

We have chosen a three-dimensional spherical case with 163842 cells as main test. With this number of cells, the total dimension of the ODE system is $21 \cdot 163842 = 3440682$. The period simulated is 300 ms, and two stimulations take place over selected cells in this period, one starting at 1 ms and other at 250 ms. To test the accuracy of the methods, we have randomly selected 100 cells, solved the case with each method and recorded the time evolution of the voltage for these 100 cells. The obtained trajectories are compared with a reference solution, which was computed using the double precision Runge-Kutta Cash-Karp subroutines described in [20], modified to include the RL technique. The relative tolerance used was 10^{-6} and the absolute tolerance was 10^{-3}.

Figure 1 depicts the transmembrane voltage evolution of several of these cells. The stimulations propagate through the structure, reaching the cells at different times.

The CPU tests were performed in a computer with two Intel Xeon X5680 hexacore processors at 3.33 GHz, while the GPU tests were performed in a machine with an Intel Core i-7 quadcore equipped with a NVIDIA Geforce GTX 580 GPU.

5.1. Accuracy evaluation

In this section, we discuss the accuracy evaluation of the different methods for this problem. In [18, 19, 28] the accuracy of different methods was studied for single cell models. We considered that the multiple cell problem has special features and must be tackled differently.

A previous remark is that the results have been obtained with single precision, with eight digits at most. This means that a relative tolerance of 10^{-6} is very close to the accuracy limit.

The standard procedure for this type of problem is to compare the results of different methods against a reference solution that is obtained with the most accurate method available. All of our solutions are composed of the trajectories of our selected 100 cells, and the error estimates given are computed as the maximum error for all the 100 cells.

The standard method to measure error in this type of simulations is the relative root-mean square (*RRMS*). If our reference solution for a given cell k is $\mathbf{y_k^{ref}}$ and our new calculated solution is $\mathbf{y_k}$, then the *RRMS* is computed as:

$$RRMS = \sqrt{\frac{\sum_{i=1}^{end}\left(\mathbf{y_{k,i}^{ref}} - \mathbf{y_{k,i}}\right)^2}{\sum_{i=1}^{end}\mathbf{y_{k,i}^2}}} = \frac{\|\mathbf{y_k^{ref}} - \mathbf{y_k}\|_2}{\|\mathbf{y_k}\|_2}, \tag{19}$$

where $\mathbf{y_{k,i}^{ref}}$ and $\mathbf{y_{k,i}}$ are the reference and new solution, both computed at the same time instant t_i Because we want to compute the *RRMS* for adaptive and fixed step solutions, we need to interpolate the solutions into the same time steps. We chose a step of 0.05 and interpolate all the solutions (using cubic splines) to obtain approximate values at time instants multiple of 0.05. Then, the *RRMS* was computed using the obtained interpolated solutions.

However, we could determine that the *RRMS* estimator is not too appropriate for this problem. The reason can be seen examining Figure 2. For all the time points considered, the RRMS uses the difference of the solution computed with the reference solution.

In Figure 2(a) we can see the trajectories of a cell computed with different methods. We can see that the trajectories are not identical, but are quite close, and the error should not be too large. The difference between the trajectories has caused that the cells have been stimulated in slightly different moments; the trajectories are very similar but for a small time (horizontal) displacement. However, in Figure 2(b), we see that in the zones where the solution has strong changes, this time displacement causes that the *RRMS* estimator includes large local errors.

These large local errors are smoothed by the *RRMS* estimator; however, this is because the *RRMS* estimator is not sensitive enough, and in some cases gives relatively large errors for quite good approximations.

Figure 2(b) shows that the computation only of "vertical" errors is not enough for this problem; Therefore, we have devised a simple procedure (which surely has been used elsewhere) that tries to estimate more accurately the real maximum error. For all the points in the considered solution, we have interpolated twice the new solution onto the reference solution; first, we use standard linear interpolation and we can compute a "vertical error", and second, we use inverse linear interpolation to obtain a "horizontal" error (See Figure 3).

We take as the *local interpolated absolute error* for this point as the minimum of the two distances, between each point and its two points interpolated onto the reference solution. Then, the *interpolated absolute error*, or *I_Abs_Error* for this trajectory is the largest of the errors for all the points of the trajectory; the *interpolated relative error* or *I_Rel_Error* for this trajectory is the *I_Abs_Error* divided by the largest absolute value of the voltage for this trajectory. Accuracy results are given in subsection 5.2.3 in terms of these two error estimates and of *RRMS*.

5.2. Comparisons

5.2.1. Methods with and without Rush-Larsen technique

We have tested several methods with and without the Rush-Larsen technique to evaluate its effect. In [18, 19], it was already reported that RL allows much faster solution for one cell models. Our results just confirm that fact. The fixed step forward Euler method implemented without RL could use in our case a maximum time step of $5 \cdot 10^{-5}$ seconds (for larger time steps the problem becomes unstable and fails), while the implementation with Rush-Larsen technique attained a maximum step of $6 \cdot 10^{-4}$ seconds. Because the computational cost of an iteration of FE and an iteration of RLFE are similar, the overall simulation time using RLFE is usually 10 times faster than using FE.

Adaptive methods not implemented with RL usually fail to obtain a solution in our test problem due to stagnation (the time step becomes tiny, taking too long to advance). This has happened with our GPU adaptive solvers and also with black-box explicit CPU solvers such as CVODE [29, 30] or the Intel ODE library [31]. It would be interesting to combine the RL technique with these state-of-the-art solvers. However, this is not easy to do because the RL technique requires explicitly the time step to be taken, while the programming interfaces of these solvers do not offer direct access to the time step.

It is interesting also that the stability limit for RLFE (that requires one system function evaluation per step) is approximately the same for higher order fixed step methods, such as explicit midpoint with RL or trapezoidal method with RL, which require two system function evaluations per step. This causes that the faster RLFE version executes in half the time required to execute the faster versions of these second order methods.

5.2.2. CPU vs. GPU

We have selected the RLFE method, implemented as described above, and compared it with a version optimized to execute in a modern CPU with 12 cores. We used the twelve cores available using OpenMP, as outlined in 4.1. The data structure was like the one shown in (16), which is more appropriate for memory access in CPUs, and finally, the Intel Compiler `icc` was used to compile the code, applying the optimization flags *-fast -openmp.*

Even with all these optimizations, the CPU solvers could not compete with the GPU solvers. The fastest version (with single precision, 24 threads and time step $5 \cdot 10^{-4}$ seconds) needed 107.4 seconds to complete the case, 10 times slower than its GPU counterpart with the same time step (see Table 4). The reason for this difference is clearly that the evaluation of the system functions for all cells is an "embarrassingly parallel" problem, completely adequate for the manycore GPU architecture.

The performance of adaptive step solvers in CPU is similarly low compared with adaptive solvers in GPU.

5.2.3. Adaptive vs fixed step in GPU

The tables 1, 2, 3, 4, and 5 show the results in terms of execution time and accuracy, using the RRMS estimate and the estimates discussed in section 5.1. The results are summarized graphically in Figure 4, as a scatter plot of computing time vs. relative accuracy (computed with the *I_Rel_Error* estimator), using a log scale for the y-axis. All of the methods have been implemented with the Rush-Larsen technique, adapted in the higher order methods as discussed in 3.3.

Adaptive methods have been tested with different sets of tolerances. The tolerances used are quite gross, but these are the most efficient for the required accuracies. The Bogacki-Shampine method stagnates if at is smaller than 0.001, while the trapezoidal Euler method suffers the same problem for absolute tolerances smaller than 0.01. The latest pair of tolerances tested is a pure absolute error test, with a really gross at value. However, with this absolute error tolerance, the adaptive methods were quite fast in all cases, and the final accuracy was not as bad as might be expected.

The largest time step that could be used for the fixed step explicit Euler Rush-Larsen method was 0.06. The same limit was found for the fixed steps implementations of the trapezoidal method and of the explicit midpoint method.

The results show that simpler methods are faster, but the accuracies obtained are significantly worse. The Runge-Kutta-Fehlberg method requires too much time and, moreover, the accuracy is not impressive (this may be due to the application of the RL technique in the internal points). For this problem, the pairs Bogacki-Shampine and Trapezoidal-Euler perform better. However, the fixed step versions perform quite well. If the minimum accuracy required is a 5%, that is, a relative error less than $5 \cdot 10^{-2}$, then Bogacki-Shampine or Trapezoidal-Euler with $at = 1, rt = 0$ could be chosen. Also fixed step RLFE with a time step 0.025 (or even a bit larger) and the explicit midpoint with time step 0.06 would give enough accuracy.

It can also be noticed that, for solutions with a small relative error (solutions with good accuracy) the relative interpolated error estimator *I_Rel_Error* is more sensitive than the *RRMS* estimator. However, the *RRMS* estimator is more sensitive for low accuracy solutions (see for example the results in table 4 for time step 0.05 or 0.06).

6. Discussion

The results presented show that the fastest results are obtained with fixed time step. The advantage for fixed time step is related to the features of cardiac tissue simulation. When re-entrant phenomena are simulated, there will be cells undergoing strong changes of potential, and this will happen during the whole simulation. These abrupt changes force a reduction of the time step. The degree of this limitation depends on the accuracy needed, large fixed steps need around half the computing time than the fastest adaptive methods. The error obtained with these large steps is greater than the 5% limit suggested in [18], although this error probably is still acceptable, taking into account the accuracy of the biological models considered.

A different treatment may be required when long simulations must be carried out, without previous information about the solution or about the optimal time step. The user of a fixed step code must define the time step, which should be used through the whole simulation. A time step too large may result in the method becoming unstable (and, typically, failure by overflow) or, maybe worse, in the code giving a qualitatively wrong solution. In absence of a reference solution, this is a real possibility. The error estimation built in adaptive solvers detects automatically strong changes, and adapts the time step accordingly. Therefore, this mechanism gives an extra degree of confidence in the computed solution, and reduces to a minimum the chances of the method becoming unstable.

A rule of thumb for use of fixed or adaptive methods, in electrocardiac simulations, might be formulated as follows: if the user has good information about the appropriate fixed time step, knowing that it is accurate enough, or if experimentation with the code is relatively "cheap", allowing the determination of the best fixed time step, then the user should select fixed step methods. If the user needs to perform long simulations of a new problem (new discretization, changes in parameters, ...) then adaptive step methods are more appropriate.

Another important practical matter is the coupling of other equations with the ODE system. In our case, simulation of electric cardiac activity should ultimately be coupled with other equations governing fluid flow, heart deformation and other phenomena. When fixed step methods are used for the different methods, the coupling is quite trivial to handle. If adaptive ODE solving is used, different phenomena may be computed at different times, which is more troublesome. However, this problem can be handled easily through free interpolants [22, 23]. These are interpolation formulae, devised for some ODE integration methods, that do not require further computations (hence the word free) and allow to compute approximate solutions at any time point, with an accuracy close to that of the associated ODE integration method.

All the solvers have been implemented over the Courtemanche atrial model [17]. The solvers and algorithms have been written to be as general as possible, encapsulating the ODE system in a system function. The only feature that makes some of the algorithms specific for electrocardiac simulation is the need of two different system functions, one for the gating variables and a different one for the rest of the variables. In particular, it should be an easy matter to use different cell models. It was shown in [18, 19] that different cell models can have very different stability properties. Therefore, the performance of the resulting codes would depend on the stability properties of the chosen model.

7. Conclusions

In this paper we have addressed the solution of large ODE systems in GPUs through adaptive step methods, setting the focus on the simulation of electric cardiac activity.

Next, we summarize the main contributions of this work:

1. We have designed and described in this paper the programming of adaptive step size methods for ODEs in GPUs. The proposed technique can be easily adapted to any embedded Runge-Kutta pair (even to implicit methods, if a suitable solver for linear systems is available) and to any standard ODE system. While adaptive methods are routinely used in CPU simulations (in packages such as MATLAB or Octave all the ODE methods implemented are adaptive), it is surprising that (to our knowledge) there is no publicly available adaptive ODE software for GPUs. We believe that this paper may be of help for scientists willing to implement their own adaptive ODE solvers for GPUs, for cardiac simulations or for any other large ODE system.
2. We have combined several explicit Runge-Kutta methods with the RL technique, using a simple technique similar to the used in [14]. Although the properties of methods obtained (stability, accuracy order) have not been analyzed rigorously, for two of the methods implemented (Bogacki-Shampine and Trapezoidal-Euler) the combination has worked quite well.

3. We have performed several comparisons to establish the best options for simulation of electric heart activity. Some of the results just confirm results from other researchers: namely, the large advantage of GPUs with respect to CPUs in this type of problem (ten times faster in our test case), and the need of the RL technique to obtain good performance.

The special features of tissue simulation (reentry phenomena and low accuracy requirements) limit the performance of adaptive step methods, compared with fixed step methods. If the accuracy is not a concern, and the best step size is known, RLFE methods are faster. Some previous experimentation with the same case is needed to obtain the best step size. However, if the maximum tolerated error is around or less than 5%, it may be more cautious to use a smaller time step, or to resort to low order adaptive methods, such as Bogacki-Shampine or Trapezoidal-Euler. These methods, combined with crude tolerances, give a good balance between accuracy, computing time, and robustness.

We plan to extend this work along different lines, starting by the development of multi-GPU versions, which is already under way. We plan to study as well the combination of implicit Runge-Kutta methods with the RL technique (which will depend on the availability of an appropriate linear solver), which might be useful for extremely stiff cases.

8. Acknowledgements

This work has been partially funded by Universitat Politècnica de València through *Programa de Apoyo a la Investigación y Desarrollo (PAID-06-11) and (PAID-05-12)*, by Generalitat Valenciana through projects *PROMETEO/2009/013* and *Ayudas para la realización de proyectos de I+D para grupos de investigación emergentes GV/2012/039*, and by Ministerio Español de Economía y Competitividad through project *TEC2012-38142-C04*.

References

[1] J.Almendral, J. Moreno, R. Vaidyanathan, A. Talkachou, J. Kalifa, A. Arenal, J.P. Villacastin, E.G. Torrecilla, A. Sanchez, R. Ploutz-Synder, J. Jalife, O. Berenfeld,

Activation of inward rectifier potassium channels accelerates atrial fibrillation in humans: evidence for a reentrant mechanism, Circulation 114 (2006) 2434–2442.

[2] M. Reumann, D. Farina, R. Miri, S. Lurz, B. Osswald, O. Doessel, Computer model for the optimization of AV and VV delay in cardiac resynchronization therapy, Medical & Biological Engineering & Computing 45 (2007) 845–854.

[3] A. van Oosterom, T.F. Oostendorp, and P.M. van Dam, Potential applications of the new ECGSIM, Journal of Electrocardiology 44 (2011) 577–583. doi: 10.1016/j.jelectrocard.2011.05.006.

[4] F.V. Lionetti, A.D. McCulloch, S.B. Baden, Source-to-source optimization of CUDA C for GPU accelerated cardiac cell modeling, in: DAmbar P. , Guarracino M., D. Talia (eds), 16th International Euro-Par Conference on Parallel Processing (2010) Part 1, Ischia, 38–49.

[5] V.K. Nimmagadda, A. Akoglu, S. Hariri, T. Moukabary, Cardiac simulation on multi-GPU platform, The Journal of Supercomputing 59 (2012) 1360–1378. doi: 10.1007/s11227-010-0540-x.

[6] D. Sato, Y. Xie, J.N. Weiss, Z. Qu, A. Garfinkel, A.R. Sanderson, Acceleration of cardiac tissue simulation with graphic processing units, Medical & Biological Engineering & Computing 47 (2009) 1011-1015. doi: 10.1007/s11517-009-0514-4.

[7] E.J. Vigmond, P.M. Boyle, L. Leon, G. Plank, Near-real-time simulations of bioelectric activity in small mammalian hearts using graphical processing units, Conference Proceedings - IEEE Engineering in Medicine and Biology Society (2009) 3290-3293. doi: 10.1109/IEMBS.2009.5333738.

[8] A. Neic, M. Liebmann, E. Hoetzl, L. Mitchell, E.J. Vigmond, G. Haase, G. Plank, Accelerating Cardiac Bidomain Simulations Using Graphics Processing Units. IEEE Transactions on Biomedical Engineering 59 (2012) 2281–2290.

[9] U.M. Ascher, L.R. Petzold, Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations, SIAM, Philadelphia, chapters 3–4 1998.

[10] A. Cristoforetti, M. Mase, F. Ravelli, A Fully Adaptive Multiresolution Algorithm for Atrial Arrhythmia Simulation on Anatomically Realistic Unstructured Meshes, IEEE Transactions on Biomedical Engineering 60 (2013) 2585–2593.

[11] E.M. Cherry, H.S. Greenside, C.S. Henriquez, Efficient simulation of three dimensional anisotropic cardiac tissue using an adaptive mesh refinement method, Chaos 13 (2003) 853–865.

[12] M. Bendahmane, R. Burger, R. Ruiz-Baier, A Multiresolution spacetime adaptive scheme for the bidomain model in electrocardiology, Numer. Methods Partial Differ. Equ. 26 (2010) 1377–1404.

[13] S. Rush, H. Larsen, Practical algorithm for solving dynamic membrane equations, IEEE Transactions on Biomedical Engineering 25 (1978) 389-392.

[14] M. Perego, A. Veneziani, An Efficient Generalization of the Rush-Larsen method for solving Electro-Physiology membrane equations, Electronic Transactions on Numerical Analysis 35 (2009) 234–256.

[15] J. Sundnes, R. Artebrant, O. Skavhaug, A. Tveito, A second-order algorithm for solving dynamic cell membrane equations, IEEE Transactions on Biomedical Engineering 56 (2009) 2546–8. doi: 10.1109/TBME.2009.2014739.

[16] Online resource:http://docs.nvidia.com/cuda/cuda-c-programming-guide /index.html, accessed March 2013.

[17] M. Courtemanche, R.J. Ramirez, S. Nattel, Ionic mechanisms underlying human atrial action potential properties: Insights from a mathematical model, American Journal of Physiology-Heart and Circulatory Physiology 275 (1998) 1 Pt 2:H301–21.

[18] M.C. MacLachlan, J. Sundnes, R.J. Spiteri, A comparison of non-standard solvers for ODEs describing cellular reactions in the heart, Computer Methods in Biomechanics and Biomedical Engineering 10 (2007) 317–326.

[19] M.E. Marsh, S.T. Ziaratgahi, R.J. Spiteri, The Secrets to the Success of the Rush-Larsen Method and its Generalizations, IEEE Transactions on Biomedical Engineering 59 (2012) 2506–2515.

[20] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes 2nd Edition: The Art of Scientific Computing, Cambridge University Press, 1992.

[21] L.F. Shampine, Error estimation and control for ODEs, Journal of Scientific Computing 25 (2005) 3–16. doi:10.1007/s10915-004-4629-3.

[22] L.F. Shampine, M.W. Reichelt, The MATLAB ODE suite, SIAM Journal on Scientific Computing 18 (1997) 1-22. doi:10.1137/S1064827594276424.

[23] P. Bogacki, L.F. Shampine, A 3(2) pair of RungeKutta formulas, Applied Mathematics Letters 2 (1989) 321-325. doi:10.1016/0893-9659(89)90079-7.

[24] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, Cambridge, MA, 2008.

[25] R. Farber, CUDA Application Design and Development, Morgan Kaufmann Publishers, Waltham, MA, 2011.

[26] http://docs.nvidia.com/cuda/profiler-users-guide/index.html, accessed March 2013.

[27] Harris M, Online resource:http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction /doc/reduction.pdf, M. Harris (2007).

[28] R.J. Spiteri, R.C. Dean, On the performance of an implicit-explicit Runge-Kutta method in models of cardiac electrical activity, IEEE Transactions on Biomedical Engineering 55 (2008) 1488–95. doi: 10.1109/TBME.2007.914677.

[29] S.D. Cohen and A.C. Hindmarsh, CVODE A Stiff/Nonstiff ODE Solver in C, Computers in Physics 10 (1996) 138–143. Also available as LLNL technical report UCRL-JC-121014 Rev. 1, August 1995.

[30] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, C.S. Woodward, SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers, ACM Transactions on Mathematical Software 31(2005) 363–396. Also available as LLNL technical report UCRL-JP-200037.

[31] A. Kalinkin, Y. Gurieva, S. Gololobov, Online resource: http://software.intel.com/en-us/articles/intel-ordinary-differential-equations-solver-library (2011).

[32] J. Kneller, R.Q. Zou, E.J. Vigmond, Z.G. Wang, L.J. Leon, S. Nattel, Cholinergic atrial fibrillation in a computer model of a two-dimensional sheet of canine atrial cells with realistic ionic properties. Circulation Research 90 (2002) E73–87.

Appendices

A. Courtemanche single atrial cell model

The main variable for this model is the transmembrane voltage V, governed by Eq. (4).

The ionic concentrations considered are the intracellular sodium, potassium, and calcium concentrations: $[Na^{+}]_i, [K^{+}]_i, \left[Ca^{2+}\right]_i$ and the calcium uptake and release by the sarcoplasmic reticulum (SR) :$\left[Ca^{2+}\right]_{up}, \left[Ca^{2+}\right]_{rel}$.

There are 16 ionic currents in the model, 12 of these are used in Eq. (4); some of these are controlled by gating variables:

1. Fast sodium current $IC_1 = I_{Na}$; controlled by gating variables h, m, j.
2. Transient outward potassium current $IC_2 = I_{to}$; controlled by gating variables o_a, o_i.
3. Ultrarapid delayed rectifier potassium current $IC_3 = I_{Kur}$; controlled by gating variables u_a, u_i.
4. Rapid delayed outward rectifier potassium current $IC_4 = I_{Kr}$; controlled by gating variable x_r.
5. Slow delayed outward rectifier potassium current $IC_5 = I_{Ks}$; controlled by gating variables x_s.
6. L-type calcium current $IC_6 = I_{Ca,L}$; controlled by gating variables d, f, f_{Ca}.
7. Fast potassium current $IC_7 = I_{K1}$.
8. Ca^{2+} pump current $IC_8 = I_{p,Ca}$.
9. $Na^{+} - K^{+}$ pump current $IC_9 = I_{Na,K}$.

10. $Na^+ - Ca^{2+}$ pump current $IC_{10} = I_{Na,Ca}$.
11. Calcium background current $IC_{11} = I_{b,Ca}$.
12. Sodium background current $IC_{12} = I_{b,Na}$.
13. Calcium release from the junctional sarcoplasmic reticulum (JSR), I_{rel}; controlled by gating variables u, v, w.
14. Transfer current between network sarcoplasmic reticulum (NSR) and JSR, I_{tr}.
15. Ca^{2+} uptake current by the NSR I_{up}.
16. Ca^{2+} leak current by the NSR $I_{up,leak}$.

The detailed expressions for all of them can be found in [17]; as an example, the expression for the first one, I_{Na} is:

$$I_{Na} = g_{Na} \cdot m^3 \cdot h \cdot j \cdot (V - E_{Na}), \tag{20}$$

where E_{Na} is the equilibrium potential for Sodium, and g_{Na} is a constant.

The model incorporates an extra potassium current, the acetylcholine (ACh) potassium current [32], not included in the original model. This current produces a shortening in the action potential which favors the occurrence of arrhythmic behavior.

All the differential equations for the gating variables can be written as Eq.(3); the concrete expressions for $a_i(V), b_i(V)$ for each gating variable can be found in [17]. Again, as an example we show the computation of $a_i(V), b_i(V)$ for the gating variable h. First, the auxiliary variables α_h, β_h are computed:

$$\begin{aligned}
\alpha_h &= \begin{cases} 0.135 \cdot exp\left(-\frac{V+80}{6.8}\right) \\ 0, \text{ if } V \geq -40 \end{cases} \\
\beta_h &= \begin{cases} 3.56 exp \cdot (0.079V) + 3.1 \times 10^5 exp(0.35V) \\ \left\{0.13 \left[1 + exp\left(-\frac{V+10.66}{11.1}\right)\right]\right\}^{-1}, \text{ if } V \geq -40 \end{cases}
\end{aligned} \tag{21}$$

then, $a_i(V) = -(\alpha_h + \beta_h)$ and $b_i(V) = \alpha_h$. Clearly, some of the numerical difficulties with the gating variables are caused by the exponential functions in eq.(21) and in the similar expressions for other gating variables.

The differential equations for the ionic concentrations are:

$$\frac{d\,[Na^+]_i}{dt} = \frac{-3I_{Na,K} - 3I_{Na,Ca} - I_{b,Na} - I_{Na}}{FV_i} \tag{22}$$

Table 1: Runge-Kutta-Fehlberg(RL)

tolerances	computing time (secs)	successful steps	*I_Rel_Error*	*I_Abs_Error*	*RRMS*
at=10^{-2}, rt=10^{-4}	45.9	4480	$9 \cdot 10^{-3}$	0.7869	$3.2 \cdot 10^{-2}$
at=10^{-1},rt= 10^{-2}	36.83	3565	$1.79 \cdot 10^{-2}$	1.54	$7.2 \cdot 10^{-2}$
at=10^{-1},rt= 10^{-1}	36.6	3448	$1.99 \cdot 10^{-2}$	1.71	$7.5 \cdot 10^{-2}$
at=1,rt=0	33.5	3226	$2.5 \cdot 10^{-2}$	2.21	$9.0 \cdot 10^{-2}$

Table 2: Bogacki-Shampine(RL)

tolerances	computing time (secs)	successful steps	*I_Rel_Error*	*I_Abs_Error*	*RRMS*
at=10^{-2}, rt=10^{-4}	37.7	6423	$3.2 \cdot 10^{-4}$	0.028	$1.2 \cdot 10^{-3}$
at=10^{-1},rt= 10^{-2}	25.2	4276	$1.2 \cdot 10^{-2}$	1.088	$5.5 \cdot 10^{-2}$
at=10^{-1},rt= 10^{-1}	24.1	4054	$1.58 \cdot 10^{-2}$	1.35	$6.7 \cdot 10^{-2}$
at=1,rt=0	23.0	3932	$1.8 \cdot 10^{-2}$	1.55	$7.4 \cdot 10^{-2}$

$$\frac{d\left[K^{+}\right]_i}{dt} = \frac{2I_{Na,K} - I_{K1} - I_{to} - I_{Kur} - I_{Kr} - I_{Ks} - I_{b,K}}{FV_i} \tag{23}$$

$$\frac{d\left[Ca^{2+}\right]_i}{dt} = \frac{B1}{B2} \tag{24}$$

where

$$B1 = \frac{2I_{Na,Ca} - I_{p,Ca} - I_{Ca,L} - I_{b,Ca}}{2FV_i} + \frac{V_{up}\left(I_{up,leak} - I_{up}\right) + I_{rel}V_{rel}}{V_i} \tag{25}$$

$$B2 = 1 + \frac{[Trpn]_{max} K_{m,Trpn}}{\left([Ca^{2+}]_i + K_{m,Trpn}\right)^2} + \frac{[Cmdn]_{max} K_{m,Cmdn}}{\left([Ca^{2+}]_i + K_{m,Cmdn}\right)^2} \tag{26}$$

$$\frac{d\left[Ca^{2+}\right]_{up}}{dt} = I_{up} - I_{up,leak} - I_{tr}\frac{V_{rel}}{V_{up}} \tag{27}$$

$$\frac{d\left[Ca^{2+}\right]_{rel}}{dt} = \left(I_{tr} - I_{rel}\right)\left[1 + \frac{[Csqn]_{max} K_{m,Csqn}}{\left([Ca^{2+}]_{rel} + K_{m,Csqn}\right)^2}\right]^{-1} \tag{28}$$

where $FV_i, V_{rel}, V_{up}, V_i, [Trpn]_{max}$, $K_{m,Trpn}, [Cmdn]_{max}$, $K_{m,Cmdn}$, $[Csqn]_{max}$, and $K_{m,Csqn}$ are all constants of the model.

In equation (7) Cm, i was chosen as 100 pF and D was chosen as $0.06mm^2/ms$, adjusted to obtain a realistic conduction velocity describing an isotropic medium.

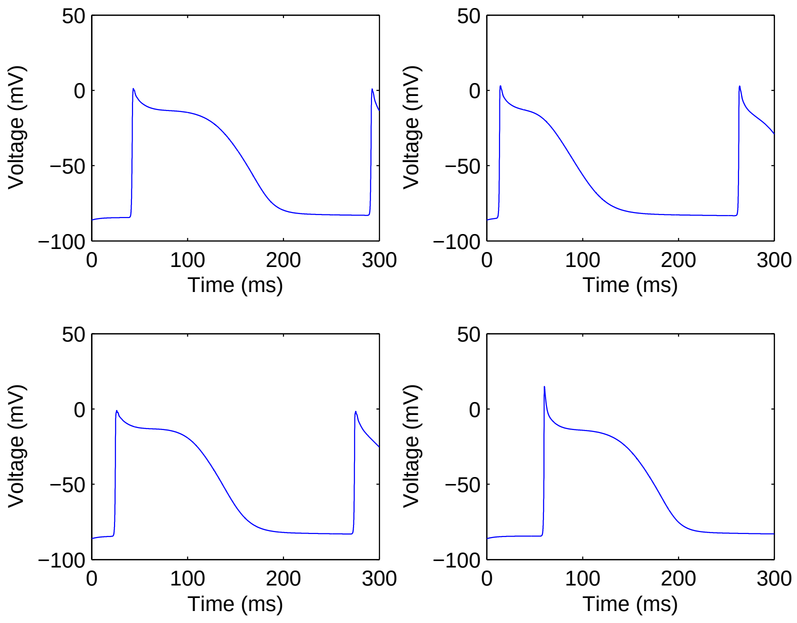


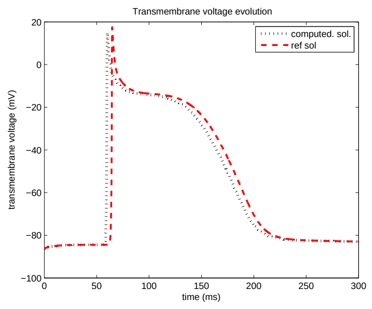
Figure 1: Transmembrane voltage evolution of four cells.

Table 3: Trapezoidal-Euler(RL)

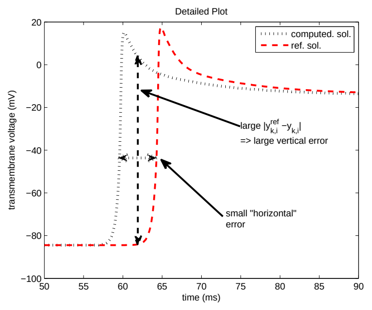
tolerances	computing time (secs)	successful steps	*I_Rel_Error*	*I_Abs_Error*	*RRMS*
at=10^{-2}, rt=10^{-4}	fail				
at=10^{-1},rt= 10^{-2}	33.29	11614	$4.4 \cdot 10^{-4}$	0.3752	$1.0 \cdot 10^{-2}$
at=10^{-1},rt= 10^{-1}	31.1	10545	$7 \cdot 10^{-3}$	0.6616	$2.1 \cdot 10^{-2}$
at=1,rt=0	18.59	6433	$2.4 \cdot 10^{-2}$	2.0	$7.7 \cdot 10^{-2}$

Table 4: Forward Euler (RL)

time step (milliseconds)	computing time (secs)	steps	*I_Rel_Error*	*I_Abs_Error*	*RRMS*
0.01	53.6	30000	$4 \cdot 10^{-3}$	0.37	$3.05 \cdot 10^{-2}$
0.025	21.7	12000	$2.5 \cdot 10^{-2}$	2.22	$9.8 \cdot 10^{-2}$
0.05	11.2	6000	$6.2 \cdot 10^{-2}$	5.38	$1.6 \cdot 10^{-1}$
0.06	9.4	5000	$7.7 \cdot 10^{-2}$	6.65	$1.8 \cdot 10^{-1}$



(a) Transmembrane voltage evolution of reference solution and of computed solution.



(b) Detailed Plot, showing large vertical error.

Figure 2: Deficiency of the RRMS estimator.

Table 5: Explicit Midpoint (RL)

time step (milliseconds)	computing time (secs)	steps	*I_Rel_Error*	*I_Abs_Error*	*RRMS*
0.01	105.8	30000	$5.5 \cdot 10^{-3}$	0.47	$2.4 \cdot 10^{-2}$
0.025	42.6	12000	$1.4 \cdot 10^{-3}$	0.12	$1.3 \cdot 10^{-2}$
0.05	21.7	6000	$1.5 \cdot 10^{-2}$	1.28	$6.1 \cdot 10^{-2}$
0.06	18.1	5000	$2.0 \cdot 10^{-2}$	1.784	$7.7 \cdot 10^{-2}$

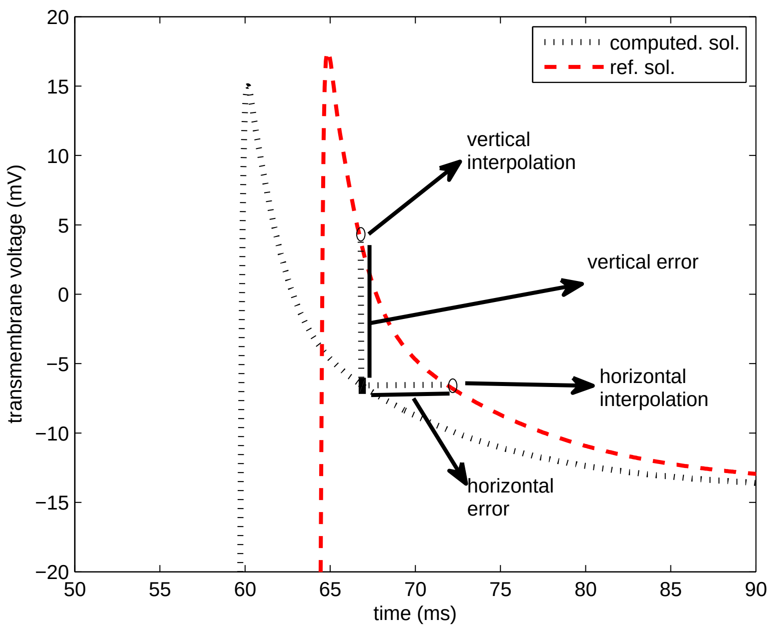


Figure 3: Error computation based on vertical and horizontal interpolation.

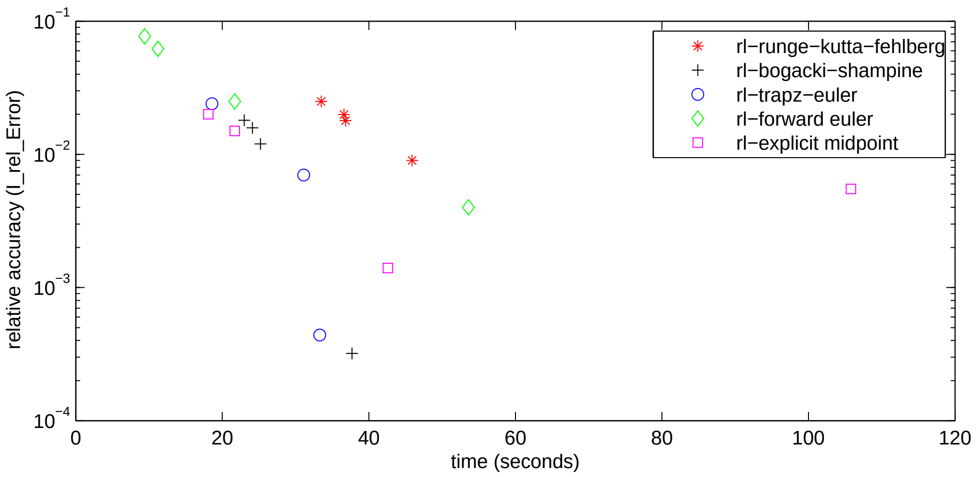


Figure 4: Scatter plot, computing time vs. relative accuracy (in log scale)