Tutorial : Simulations with Genesis using **`hsolve`**

Hugo Cornelis[1]
Laboratory of
Theoretical Neurobiology
University of Antwerp

Erik De Schutter[2]
Laboratory of
Theoretical Neurobiology
University of Antwerp

Antwerp, November 15, 2002

[1]hugo@bbf.uia.ac.be
[2]erik@bbf.uia.ac.be

Contents

Abstract

This tutorial is designed for the advanced Genesis user interested in better understanding how the Hines-solver (`hsolve`) is implemented in Genesis. We will first review some equations fundamental to many neuro-biological models and illustrate how these equations are solved numerically. We will then focus on the implicit solutions to these equations using `hsolve` and show how `hsolve`'s configuration relates to the numerical side. After a survey on the basic use of `hsolve` and examples on how to interface `hsolve` to other Genesis elements, we show a number of useful tricks that are impossible without a global computation engine as `hsolve`. Finally we conclude with the more technical sides like `hsolve` - `synchan` interoperability via discrete events and disassembling byte codes that represent numerical equations.

Part I

Towards basic use of `hsolve`

Chapter 1

From Numerical Theory to Simulation Practice

1.1 Numerical Preliminaries

In this first section we will shortly introduce some fundamental concepts without going to deep into the mathematical details. This will help you understand how `hsolve` relates to the software design of Genesis and why it is superior when compared to the other objects.

The basic problem can be stated as follows :

1. A biological model – what we want to compute – is a set of coupled biological components.
2. The time-dependent behavior of a biological component is described with a (set of) differential equation(s).
3. A single differential equation describes a rate of change of a variable.

To solve a system of differential equations that describes the behavior of a biological system, we use a numerical method. The methods about to be introduced are all based on the Taylor series.

1.1.1 Taylor series

If we denote with $y^{(n)}(t)$ the function value at point t of the n'the derivative of a function y, then the Taylor series of a continuous function y at point t is given by :

$$y(t+h) = y(t) + hy^{(1)}(t) + \frac{1}{2!}h^2y^{(2)}(t) + \frac{1}{3!}h^3y^{(3)}(t) + \cdots + \frac{1}{n!}h^ny^{(n)}(t) + \cdots \tag{1.1}$$

Some remarks about this expansion :

1. h can be any element of $\mathbb{R}$, such that a function is completely defined by its Taylor expansion at any single point of its domain. (Full knowledge of the function at a single point determines the full function at all points).
2. The successive terms of the Taylor series are decreasing in magnitude in an exponential way.
3. We can truncate a Taylor series to approximate the original function. This divides the Taylor series in two separate series : the numerical scheme and the error series. The error after truncation is mainly dependent on the first term of the error series.

1.1.2 Numerical Schemes

If $y(0)$ is known, $y(1)$ can be calculated by evaluating the Taylor series (1.1) with h set to 1. Since the Taylor series is an infinite sum, you have to truncate the series after N terms, so you introduce an error that scales with the magnitude of the $N+1$'th term. This error is called the *local truncation error*. It scales with h^N.

Successive application of the series on the obtained results, gives a sequence of numbers $\langle y(0), y(h), y(2h \ldots \rangle$. If h is considered a time step, this simple scheme allows you to approximate any continuous function or – as we use to call it – simulation of the variable described by that function.

The accumulation of the local truncation errors in such an approximation or simulation is called the *global truncation error*. To obtain more accuracy in the results, it is obvious that h should be made smaller. This however results in more steps needed for the same simulated time. If you divide h by two for example, the number of steps needed to obtain the same simulated time is multiplied by two with more accumulation of local error as a result. In general the global truncation error is always at most one order of magnitude larger than the local truncation error.

Forward-Euler The forward-Euler method truncates the Taylor series after two terms :

$$y(t + h) = y(t) + hy^{(1)}(t) \tag{1.2}$$

Assuming that the value at point t is correct, the forward-Euler method computes the value at point $t + h$ with a local error that scales with h^2 (see the first term of the error series). The forward-Euler method always gives overshoots on the original curve.

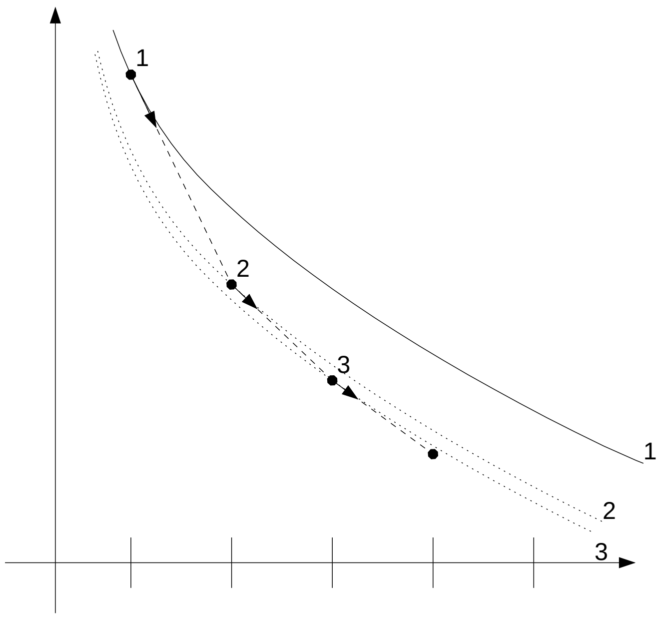

Figure 1.1: Graphical illustration of the forward-Euler method for an exponential like curve. Starting at point 1, the tangent of the curve is taken and linearly extrapolated to obtain point 2. There again the same procedure is used to obtain point 3. Note that point 2 lies on curve 2 and point three lies on curve 3, both of which are offset against the original curve.

Backward-Euler The backward-Euler method also truncates the Taylor series after two terms. The difference is that the derivative is evaluated at point $t + h$ instead of at point t.

$$y(t + h) = y(t) + hy^{(1)}(t + h) \tag{1.3}$$

Assuming that the value at point t is correct, the backward-Euler method computes the value at point $t + h$ with a local truncation error that scales with h^2. The backward-Euler method always gives undershoots on the original curve.

Normally we do not know the derivative at point $t + h$, although we need it to compute the function value at point $t + h$. In practice this requires a rearrangement of the equation. We call such a numerical scheme an *implicit numerical scheme*. For most equations implicit schemes are more stable than explicit schemes because of the undershoots.

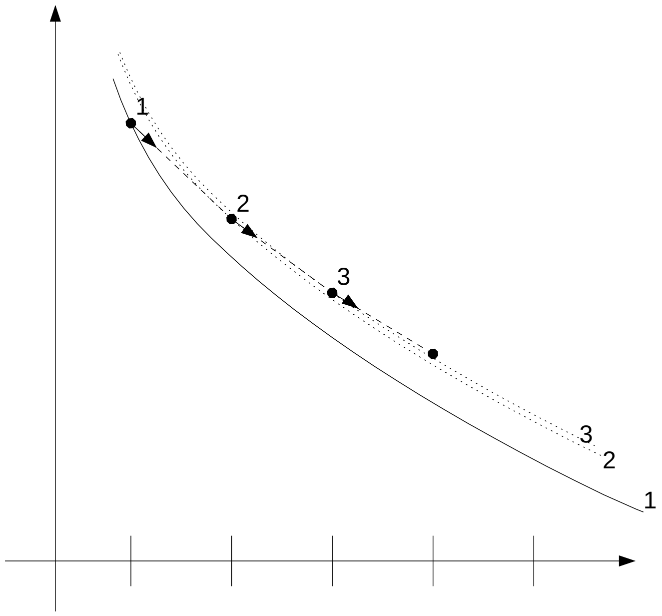

Figure 1.2: Graphical illustration of the backward-Euler method. To obtain point 2 from point 1, we take the derivative at point 2 and extrapolate it at point 1. To obtain point 3 starting at point 2, we do the same : take the derivative at point 3 and extrapolate it at point 2.

	Explicit	Implicit
First Order	Forward-Euler, exponential Euler	Backward-Euler
Second Order	2nd-Order Runge-Kutta	Trapezoidal rule
Higher Order	Adams-Bashforth, Runge-Kutta-Fehlberg	Adams-Moulton, Gear

Table 1.1: The numerical schemes of interest. Not all of them are mentioned in the text, see [8] and [6] for more information.

Exponential Euler The main idea behind the exponential Euler rule is that many biological processes are governed by an exponential decay function. Being the default numerical method in Genesis, it is shortly reviewed for the sake of completeness. For an equation of the form :

$$\frac{\mathrm{d}y}{\mathrm{d}t} = A - By \tag{1.4}$$

its scheme is given by :

$$y(t+h) = y(t)\mathrm{e}^{-B\Delta t} + \frac{A}{B}(1 - \mathrm{e}^{-B\Delta t}) \tag{1.5}$$

The Trapezoidal Rule The trapezoidal rule is a simple average of the forward-Euler and backward-Euler schemes. It can be shown that the local truncation error scales with h^3.

$$y(t+h) = y(t) + \frac{h \cdot (y^{(1)}(t) + y^{(1)}(t+h))}{2} \tag{1.6}$$

For ordinary differential equations, the trapezoidal rule is an application of the θ method, which itself is a special case of a second-order Runge-Kutta method. For more details see [6].

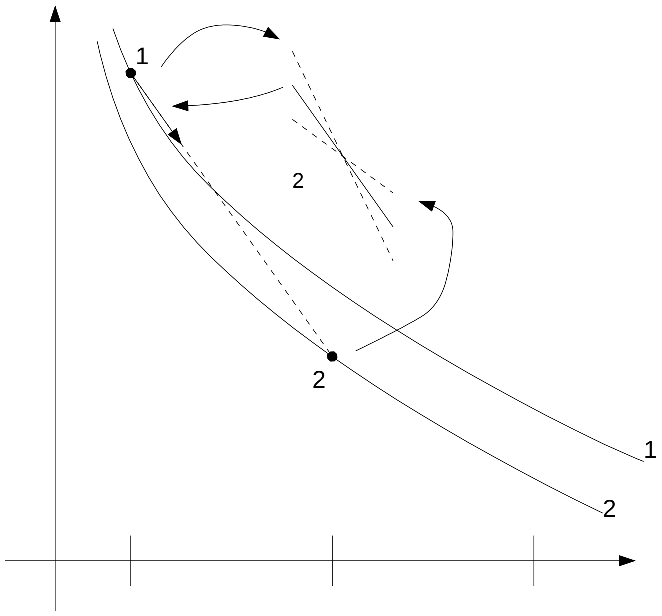

Figure 1.3: Graphical illustration of the trapezoidal method. Starting at point 1, we get point 2 by taking the derivatives at point 1 and point 2, and extrapolating their average in point 1.

1.2 Neuroscience Equations

The membrane potential (V) in a single branch of a neuron is described by the one dimensional cable equation :

$$\frac{1}{2\pi a}\frac{\partial}{\partial x}\left(\frac{\pi a^2}{R_a}\frac{\partial V}{\partial x}\right) = C_m\frac{\partial V}{\partial t} + I_{\mathrm{HH}} \tag{1.7}$$

If we consider the axial resistance and dendritic diameter constant, this is simplified to a cylindrical cable equation :

$$\frac{a}{2R_a}\frac{\partial^2 V}{\partial x^2} = C_m\frac{\partial V}{\partial t} + I_{\mathrm{HH}} \tag{1.8}$$

In the the Hodgkin-Huxley formalism, the current is defined as :

$$I_{\mathrm{HH}} = \overline{g_{\mathrm{Na}}}m^3h(V - E_{\mathrm{Na}}) + \overline{g_{\mathrm{K}}}n^4(V - E_{\mathrm{K}}) + g_{\mathrm{L}}(V - E_{\mathrm{L}}) \tag{1.9}$$

m, h and n are voltage and time dependent variables between 0 and 1, each satisfying a simple exponential curve, described by :

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \alpha_h(V) - (\alpha_h(V) + \beta_h(V))\cdot h \tag{1.10}$$

1.2.1 A Single compartment

In the one dimensional cable equation the membrane potential is dependent on time as well as space. Both these axes can be discretized independently. We first assume that the cell is iso-potential (same membrane potential at all places) which eliminates the spatial dependence (if you omit the Hodgkin-Huxley conductances, this also leads to integrate-and-fire models).

Since the cell is assumed to be iso-potential, the cable equation reduces to a simple exponential decay function. The question remains how to fill in the Hodgkin-Huxley current while maintaining the second-order accuracy of the trapezoidal rule. The following trickery is used for this :

$$
\begin{aligned}
y(t+h) &= y(t) + hy^{(1)}(t) + \frac{1}{2!}h^2y^{(2)}(t) + \frac{1}{3!}h^3y^{(3)}(t) + \cdots + \frac{1}{n!}h^ny^{(n)}(t) + \cdots \\
y(t-h) &= y(t) - hy^{(1)}(t) + \frac{1}{2!}h^2y^{(2)}(t) - \frac{1}{3!}h^3y^{(3)}(t) + \cdots + (-1)^n\frac{1}{n!}h^ny^{(n)}(t) + \cdots
\end{aligned}
$$

Subtract and regroup :

$$
\begin{aligned}
y(t+h) - y(t-h) &= 2hy^{(1)}(t) + \frac{2}{3!}h^3y^{(3)}(t) + \cdots + \frac{2}{2n!}h^{2n}y^{(2n)}(t) + \cdots \\
&\approx 2hy^{(1)}(t)
\end{aligned}
$$

Or if you rewrite :

$$
hy^{(1)}(t) \approx \frac{y(t-h) - y(t+h)}{2} \tag{1.11}
$$

This equation has a local truncation error of third order and can be used to fill in the membrane potential in the Hodgkin-Huxley equations and vice versa in the following way : we assume that $y(t+h)$ is the unknown gate value we are searching, while $y(t-h)$ is the previous gate value and $y^{(1)}(t)$ is the rate of change for the gate value that can be calculated given the membrane potential at time t. Under the assumptions that $y(t-h)$ and the membrane potential are second-order correct, the result of this calculation – the new values for the gates – will be second-order correct too. Then the same method can be used to compute the membrane potential at time point $y(t+2h)$ given the membrane potential at time point $y(t)$ and the gate values at time point $y(t+h)$.

It is important to see that the second-order accuracy is maintained only when the membrane potential and the Hodgkin-Huxley equations are computed at different time points.

NOTE : If concentration pools are represented in a model of a neuron, they should be evaluated at the same time steps as the membrane potential.

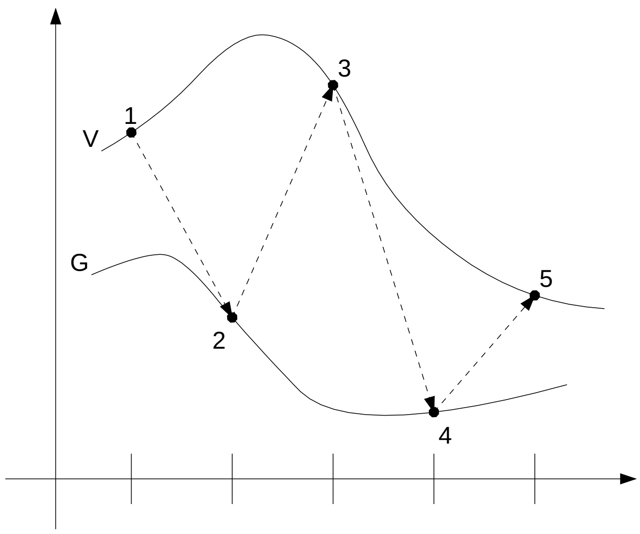

Figure 1.4: Mid step evaluation of membrane potential and Hodgkin-Huxley equations. To calculate the conductance at time point 2, we need the membrane potential at time point 1. To calculate the membrane potential at time point 3, we need the conductance at time point 2. The membrane potential at time point 2 is never computed, neither the conductance at time point 3.

1.2.2 Multiple compartments

In the trapezoidal scheme the following rule is used to spatially discretize the cylindrical cable equation :

$$\frac{\partial^2 V(x,t)}{\partial x^2} \approx \frac{V_{x-\Delta x}(t) - 2V_x(t) + V_{x+\Delta x}(t)}{(\Delta x)^2} \tag{1.12}$$

This scheme splits up a single cable in multiple compartments and is second-order accurate in space if the discretization length is kept constant (use the Taylor series to prove this : take the sum of a Taylor expansion for $V(x + \Delta x)$ and $V(x - \Delta x)$ and see what happens). It is used by most popular neurobiological simulation packages to discretize dendritic trees (however with a non constant discretization length, which makes the rule first order accurate).

If this scheme is combined with an explicit numerical scheme like the exponential Euler, all the resulting equations are self contained and can be solved in isolation. However if this scheme is combined with an implicit numerical scheme like the backward-Euler, the numerator of the right-hand side couples the neighboring compartments to each other. it gives rise to a system of coupled finite-difference equations. If you choose the trapezoidal rule for time discretization, the method is called a Crank-Nicolson. For a linear cable the coefficients of this system can be arranged in a tridiagonal matrix such that it can be solved in linear time. For branched morphologies the matrix is still symmetric (not tridiagonal however), and it can still be solved in linear time. The arrangement of the equations is done with Hines numbering (a special kind of minimum degree algorithm, see[7]), and the solution can be obtained with Gaussian elimination without pivoting[8, 7]. This operation scales with the number of equations in the system and as such it can be considered to be a fast solution for this system. Figure 1.2.2 gives an example morphology with the corresponding matrix structure. The text below the figure gives more explanation about how to solve the system of equations.

NOTE : Without extensions these schemes cannot be applied to gap junctions, since gap junctions give rise to looped electrical circuits such that the matrix containing the cable equations, cannot be solved with Gaussian elimination without pivoting.

$$\begin{bmatrix} b & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a & b & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & 0 & 0 & a & 0 & 0 & 0 \\ 0 & 0 & 0 & b & a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a & b & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & b & a & 0 & 0 & 0 \\ 0 & 0 & a & 0 & 0 & a & c & a & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a & b & a & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & a & b & a \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a & b \end{bmatrix} \cdot \overline{V(t+\Delta t)} = \overline{V(t)}$$

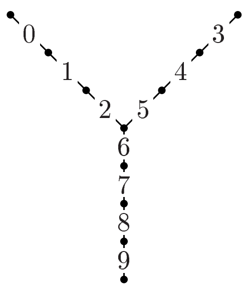

Figure 1.5: Resulting matrix structure after applying an implicit numerical scheme to the shown morphology. The numbers in the figure correspond to the rows in the matrix and come from one possible (Hines) numbering scheme[7]. The system can easily be solved by sweeping through the equations twice : the first sweep eliminates all the coefficients below the main diagonal and results in the decoupling of the last equation such that it can be solved in isolation. Then the second sweep starts by substituting the last unknown in the second last equation, such that this equation can be solved too. Continuing this process upwards solves the complete system. (Application of an explicit numerical scheme to this morphology would result in a matrix with only coefficients on the main diagonal. Of course in such a system all equations can be solved in isolation.)

Chapter 2

Introducing `Hsolve` for Single Cells

If you create a set of coupled compartments with Genesis, you already have discretized your neuron so we will not consider spatial discretization any further. In this section we introduce the Genesis '`hsolve`' object. To implement the time discretization, this object implements some of the previously introduced numerical methods and combinations thereof.

So far we have identified the backward-Euler and the Crank-Nicolson rules as implicit finite-difference schemes. The explicit schemes we encountered are the exponential Euler and the forward-Euler. Since we have to solve two types of equations (cable equation and Hodgkin-Huxley equations), these different time-discretization techniques can be applied to the different equations and combined at will :

1. Solve all equations with an explicit method (forward-Euler or exponential-Euler). This isolates all compartments to single equations that can be solved independently. Being the default numerical methodology in Genesis, you are supposed to be familiar with it. See [2] for more details.
2. Solve the cable equation with an explicit method, solve the Hodgkin-Huxley channel equations with an implicit method. This is rather uncommon and will not be treated either.
3. Solve the cable equation with an implicit method, solve the Hodgkin-Huxley channel equations with an explicit method at the same time points.
4. Solve the cable equation with an implicit method and solve the Hodgkin-Huxley channel equations with an implicit method at halve time points.

The implicit solver of Genesis, '`hsolve`', implements the last two points in the above enumeration. We cover the use of `hsolve` in the following sections.

2.1 Basic Use : A short overview

Since `hsolve` is only a computation engine, it has no knowledge of your model (number of equations, morphology, ...). The first step in using `hsolve` is always a step of model construction without using `hsolve` at all. The next step is creating and configuring `hsolve` such that it knows what (part of) the model to compute and how to compute it. Then you inform `hsolve` that everything is in place and it may do its internal initialization. After the usual '`reset`', you can start the simulation.

If we want to compute two coupled compartments with Hodgkin-Huxley channels, this may look like the following :

1. Write your scripts as if you are not using `hsolve` at all :
   (a) *Set the simulation clock(s).*

   ```
   genesis > setclock 0 0.0000030
   ```

   NOTE : Never forget to set the main simulation clock. If you do not do so, `hsolve` will be confused and can produce unexpected results.

(b) *Create the model* you want to simulate : create all the elements, set all their fields and create messages between them. In this example we create two coupled compartments each containing a fast sodium conductance and a delayed rectifier conductance. For `hsolve` it is important that these channels are below the compartments they belong to in the Genesis element hierarchy :

```
genesis > create neutral /cell
genesis > ce /cell
genesis > create compartment c1
genesis > setfield c1 Ra 0.5 Rm 10 Cm 0.01 Em -0.065
genesis > copy /library/NaF c1/NaF
genesis > addmsg c1 c1/NaF VOLTAGE Vm ; addmsg c1/NaF c1 CHANNEL Gk Ek
genesis > copy /library/Kdr c1/Kdr
genesis > addmsg c1 c1/Kdr VOLTAGE Vm ; addmsg c1/Kdr c1 CHANNEL Gk Ek
genesis > copy c1 c2
genesis > addmsg c1 c2 AXIAL Vm ; addmsg c2 c1 RAXIAL Ra Vm
```

NOTE : In this example all elements use the same simulation clock i.e. clock 0. When you want to use `hsolve` on elements that use different simulations clocks, reconfigure the model such that they all use the same simulation clock. Afterwards migrate to `hsolve`.

2. Create and configure `hsolve` :

(a) *Create hsolve* element at the right location.

```
genesis > ce /cell
genesis > create hsolve solver
```

(b) *Set the 'path' field* with a wild card to match the compartments to be computed. The '`path`' field is always specified relative to the hsolve element (not to the current working element). In this example the full path specification is "`/cell/solver/../##[][TYPE=compartment]`", which is the same as "`/cell/##[][TYPE=compartment]`". Configure `hsolve` further by setting other fields like `chanmode` and `calcmode` if needed (it is not needed in this example). We will cover these fields in one of the next sections.

```
genesis > setfield solver path "../##[][TYPE=compartment]"
```

(c) *Set the numerical method* to use for the computations done by hsolve (normally one always uses Crank-Nicolson, i.e. method 11).

```
genesis > setmethod solver 11
```

NOTE : Never forget to set the method to backward-Euler (method 10) or Crank-Nicolson (method 11). If you do not do so, `hsolve` will be confused and can produce unexpected results.

3. Initialize `hsolve` :

(a) *Call the SETUP action* on `hsolve`. This allows `hsolve` to do its internal initialization. Basically `hsolve` will *examine* the *structure of the model* and compile it into an internal efficient representation. All operations up till this step may only be performed once.

```
genesis > call solver SETUP
```

(b) Do *other things* of interest to your simulation (*don't touch the elements that will be computed by hsolve*).

```
genesis > ...
```

(c) *Call the RESET action* on `hsolve` (this is automatically done by the Genesis '`reset`' command). This will *transfer and convert* all *initial values* into `hsolve`'s internal data structures. This step can be repeated as many times as needed since it does not alter the model's structure.

```
genesis > reset
```

4. Do the simulation ('step' command).

```
genesis > step 1 -time
```

That's it, here is again the full code :

```
genesis > create neutral /cell
genesis > ce /cell
genesis > create compartment c1 -initVm
genesis > setfield c1 Ra 0.5 Rm 10 Cm 0.01 Em -0.065
genesis > copy /library/NaF c1/NaF
genesis > addmsg c1 c1/NaF VOLTAGE Vm ; addmsg c1/NaF c1 CHANNEL Gk Ek
genesis > copy /library/Kdr c1/Kdr
genesis > addmsg c1 c1/Kdr VOLTAGE Vm ; addmsg c1/Kdr c1 CHANNEL Gk Ek
genesis > copy c1 c2
genesis > addmsg c1 c2 AXIAL Vm ; addmsg c2 c1 RAXIAL Ra Vm
genesis > ce /cell
genesis > create hsolve solver
genesis > setfield solver path "../##[][TYPE=compartment]"
genesis > setmethod solver 11
genesis > call solver SETUP
genesis > ...
genesis > reset
genesis > step 1 -time
```

NOTE : Depending on the circumstances, the `hsolve` element can be created automatically. An example is the use of the '`-hsolve`' option of the '`readcell`' command in which case the `path` field is also set to point to the compartments of the cell that is being read. Currently an annoying bug in the `readcell` code obliges you to use absolute pathnames for the created elements if you use this option.

NOTE : In the example given above all elements use clock zero. It must be noted that `hsolve` computes all variables using the same clock (so the same time step). If you only use the elements that act as a model for `hsolve`, you have the flexibility to use different clocks for different elements (so different time steps for different elements).

It is useful to discuss how Genesis deals with `hsolve`. Without `hsolve`, all elements you create, are responsible for their own calculations. A compartment for example will compute a (cylindrical) cable equation using the exponential-Euler rule and has some facilities to communicate certain variables to or from other elements (via the Genesis message system). When using `hsolve` however, `hsolve` does the computations as shown in figure 2.1. Depending on the configuration of `hsolve` – something that will be discussed in the next section – `hsolve` is able to use the facilities of the original objects to communicate with other elements, but it is also able to use its own internally optimized communication facilities.

2.2 The `chanmode` Field : Modes of Operation

In the previous example we have seen a basic example of how to setup `hsolve` and configure it to simulate a model represented by two compartments. How is this related to the mathematics we discussed earlier on ? Which numerical scheme was used for the cable equation and which numerical scheme was used for channels ? These questions are answered by inspecting the `chanmode` field. Basically you have to make a major distinction between the lower chanmodes (chanmode 0 and 1) and the higher chanmodes (chanmode 2 to 5). The lower chanmodes are modes intended for compatibility with other Genesis objects and they are slow compared to the higher chanmodes. `Hsolve` will automatically update all the computed fields in the original elements, such as to create the illusion that the original elements still perform their own computations. The updated fields are then available for plotting or for other elements to do additional calculations.

1. chanmode 0 (default) : `hsolve` computes the cable equation only (i.e. `hsolve` does the computations of the compartments). All the other elements (Hodgkin-Huxley channels, concentration elements) still do their own calculations. Since `hsolve` does not compute the channel equations, the channel rates are updated at time points independent of update times of the membrane potential. This loss of second-order accuracy results in the penalty of having to use a small time step.

   To communicate variables to/from other elements, `hsolve` uses the message system of Genesis. This makes the chanmode 0 the most compatible mode of operation and it is recommended for people that have developed their own set of custom objects.

2. chanmode 1 : hsolve computes the cable equations and the `tabchannel` elements inside the compartments with a staggered time grid. This results in some extra performance but is not completely compatible with all types of setup. More precisely, incoming and outgoing messages for the tabulated channels are ignored, except for the messages that link the channels to the compartments (`CHANNEL`, `VOLTAGE`), single messages coming from a `nernst` element (`EK`) and a single message that is linked with the Z gate (`CONCEN`). If your model contains only tabulated channels, all conductances will be calculated by `hsolve` at time points between the updates of the membrane potential. This maintains the second-order accuracy and thus you can use a larger time step.

   To use chanmode 1 in the previous example, you configure `hsolve` with the following command :

   ```
   genesis > setfield solver path "../##[][TYPE=compartment]" chanmode 1
   ```

NOTE : The compartmental `Im` field is only computed by `hsolve` running in chanmode 0 or 1 if you turn on the field '`computeIm`'. Calculation of the `Im` field incurs a small performance penalty.

When using the higher chanmodes, `hsolve` will compile all elements that belong to your model to byte-codes. During the simulation these byte-codes are emulated which results in a high performance (the higher chanmodes can be 5-10 times faster than the lower chanmodes, depending on the model structure). To be able to compile an element into byte-codes, `hsolve` must be aware of the element type (the Genesis object) and its computations. The objects `hsolve` is able to compile to byte-codes, are documented in the Genesis manuals.

The higher chanmodes all share the property that `hsolve` computes all the elements present in your model on the staggered time grid, so this guarantees second-order accuracy under all circumstances. A single time step in the simulation will compute the membrane potential at the simulated time after an update of the conductance at an intermediate time point.

In most cases the original elements' fields will not be updated. For `hsolve` the elements only serve as a description of the model to be computed, so we call these elements *modeling elements*. To inspect calculated values, they have to be fetched with the `findsolvefield` command. We will show examples of the use of this command shortly. The difference between the different chanmodes lies in the fact that some fields might not be computed and are not accessible – even with the `findsolvefield` command – as explained in the following :

1. chanmode 2 : The compartmental voltage is stored in the respective fields of the original compartments. Other fields like channel gates are accessible via the `findsolvefield` command. The channel parameters `Gk`, `Ik`, `Ek` and the compartmental membrane current `Im` are not stored by `hsolve`. At every time step of the simulation, `hsolve` will update the membrane potential of the original compartments (that serve as the model) with the computed membrane potential. If you are only interested in membrane potentials, this mode will be handy. Outgoing messages to non computed elements that are created before the `SETUP` call are not supported in this mode.

2. chanmode 3 : This is exactly the same as chanmode 2, except that the compartments' `Vm` field is not updated. The membrane potential is only accessible via the `findsolvefield` command.

3. chanmode 4 : As in chanmode 3, none of the fields of the modeling elements will be updated, but in this chanmode all fields are accessible with the `findsolvefield` command (including the channel parameters `Gk`, `Ik`, `Ek` and the compartmental `Im` field).

4. chanmode 5 : This chanmode is as chanmode 4 (all fields are stored), but the channel parameters `Gk`, `Ik` and `Ek` and the compartmental membrane current `Im` are calculated as relative values (normalized to the compartment surface).

2.3 Tabulated Calculations

`Hsolve` puts up a number of restrictions for the tables of the tabulated elements. These are summarized below.

The `calc_mode` field. Genesis makes extensive use of precomputed tables to model conductance that depend on the membrane potential. The `tabchannel` element is an example. The tables are a simple discretization along the V-axis of the function they model, so they 'export' a set of (V_i, y_i) tuples, $V_i = \mathrm{V}_0 + i \cdot \Delta\mathrm{V}$, $0 \leq i \leq \mathrm{N}$. Since the membrane potential V is a computed variable, there is no guarantee that it will be equal to one of the V_i's at any point in simulation time. The tabulated elements in Genesis know about two basic modes of calculating a value y for a given value V :

1. Truncation : for a given value V, $V_i \leq V < V_{i+1}$, the value y_i is returned. Since the value y_i is directly stored in the table, no additional calculations have to be performed.
2. Linear interpolation : for a given value V, $V_i \leq V < V_{i+1}$, the value $y = y_i + (V - V_i) \cdot \frac{(y_{i+1} - y_i)}{(V_{i+1} - V_i)}$ is returned. The value y is the result of a linear interpolation between two tuples stored in the table.

The distinction between these two modes of operation is made by the `calc_mode` field. If this field is set to 1, linear interpolation is used, if the field is set to 0, truncation is used.

For most tabulated elements of genesis the `calc_mode` field is found in the `interpol structs` that reside in the element fields. However `hsolve` has its own `calcmode` field – note the small difference in name – which is applied to all tabulated calculations. The granularity provided by the Genesis elements is much higher than provided by `hsolve`. That is why you always have to set the field manually.

NOTE : Never forget to set the `calcmode` field and to check if it is equal to the calculation mode of all tabulated elements. If you don't, your results will be almost right, but not exactly.

The Table Properties. Besides the restriction that all tabulated calculations are done in the same calculation mode, `hsolve` also demands that all tables of the dimension are of the same type. This means that they must have the same size and the same discretization step. The sizes of the tables can be queried by inspecting the `xdivs` and `ydivs` fields. The discretization step can be queried by inspection of the fields `invdx` and `invdy`. `Hsolve` will give an appropriate error message if there are conflicts between the properties of different tables it has to use.

2.4 Interpreting the Mode of Operation

Suppose you are inspecting a simulation, to see what it does and how it does it. You know `hsolve` is being used in the simulation, but you do not know where in the scripts it is configured. This short section will help you out. The first step is to locate all `hsolve` elements :

```
genesis > echo {el /##[][TYPE=hsolve]}
/purkinje
```

Then for every `hsolve` element reported, you inspect the `chanmode`, `calcmode` and `path` field.

```
genesis > showfield /purkinje path chanmode calcmode
[ /purkinje ]
path                  = ./##[][TYPE=compartment]
chanmode              = 4
calcmode              = 1
```

The `path` field tells you that all `compartments` below the `hsolve` element have been taken over by `/purkinje`. The `chanmode` is four which means that also all intracellular mechanisms within the compartments will be computed by `hsolve`. Finally the `calcmode` tells that all tables are used with linear interpolation to compute results.

The numerical method that `hsolve` uses for the cable equations and the equations of intracellular mechanisms is found with the following command :

```
genesis > showfield /purkinje object->method
[ /purkinje ]
object->method       = 11
```

So in this case Crank-Nicolson is used to update the variables.

2.5 Exercise : Speed Comparison

The numerical advantage of implicit methods is that you can use a much larger time step in order to compute the same neuronal model (this is due to the stiffness of the system, see [8]). In order to examine the dependence of the results on the time step you can run a simulation with and without `hsolve` while varying the time step. However In this exercise we are concerned about something completely different : hsolve optimizes the calculations such that it runs faster than the original compartments. The speed advantage by using `hsolve` is given in table 2.1.

no Xodus output			
original	66.30	66.44	66.41
chanmode 0	56.79	56.76	56.81
chanmode 1	40.26	40.27	40.34
chanmode 2	15.45	15.27	15.31
chanmode 3	14.02	13.90	13.90
chanmode 4	16.47	15.96	16.05
chanmode 5	16.99	16.55	16.49
Xodus output			
original	66.55	66.60	66.63
chanmode 0	57.14	57.05	57.05
chanmode 1	40.60	40.67	40.64
chanmode 2	15.29	15.51	15.41
chanmode 3	14.08	14.08	14.20
chanmode 4	16.34	16.32	16.36
chanmode 5	16.63	16.85	16.81

Table 2.1: Simulating a Purkinje cell containing ∼4000 compartments without and with `hsolve` in the different chanmodes. The table lists two times three trials : the first set of trials is done without any output. The second set is obtained while plotting the membrane potential of the soma. All numbers give the total CPU time needed to simulate 1000 steps, all simulations used the same time step.

2.6 Communication with Other Elements

Knowing how to use `hsolve` can only be interesting when you also know how to setup experiments and how to save the output. Interfacing `hsolve` to other elements is not always trivial and deserves a special paragraph of attention.

A distinction can again be made between the lower and the higher chanmodes. When using the lower chanmodes – the 'compatibility' chanmodes – everything behaves almost as without `hsolve`. In chanmode 0, under no circumstance you will experience problems when interfacing `hsolve` to other objects. Incoming and outgoing messages that have been created before the `SETUP` action call are handled automatically. In chanmode 1, `hsolve` puts severe restrictions on the messages that can be send to the tabulated channels. Since `hsolve` will not always give an appropriate error message for messages that are handled or that are ignored, it is recommended not to use chanmode 1 unless you are sure that you do not have any tabulated channels with messages that are not handled by `hsolve` (these messages include `DOMAINCONC` and `ADD_GBAR`).

The higher chanmodes are the more interesting modes of operation because of their excellent performance. We will concentrate on them from now on and explain how to use them correctly.

As already said, when using the higher chanmodes, `hsolve` compiles your model into optimized byte-codes. During the simulation these byte-codes are interpreted at each time step. They instruct `hsolve` to compute the conductances and evaluate the cable equations. To be able to compile a model into byte-codes, `hsolve` must know the object type. The object types `hsolve` is currently supporting are `compartment`, `tabchannel`, `tab2Dchannel`, `tabcurrent`, `Ca_concen`, `concpool`, `difshell`, `taupump`, `mmpump`, `hillpump`, `fixbuffer`, `difbuffer`, `dif2buffer`, `fura2`, `nernst`, `ghk`, `channelC2`, `channelC3`, `synchan`, `synchan2`, `Mg_block`, `spikegen`, `neutral`.

When using these objects in a model, be sure to create them beneath the subtree of the compartments they reside in. Then you link them together with messages as usual. Afterwards you create `hsolve`, set the `path` field (point it to the compartments only), and perform a `SETUP` and `RESET` call. If anything is wrong with the model structure, e.g. if a message incoming to a channel is not handled by `hsolve`, `hsolve` will give an appropriate error message. If all is right, `hsolve` is setup correctly. This means that (1) all computations normally done by the modeling elements, are now done by `hsolve` in a more efficient way and (2) the message passing between these elements is done by `hsolve`'s internal communication facilities. This is all made transparent for a user except . . . . . . that the computed values are not stored anymore in the original elements, but they are stored in `hsolve`'s internal data arrays. How to communicate with these data arrays will be explained shortly.

> **NOTE :** `Hsolve` implicitly assumes that you are not doing fancy things like sending an `AXIAL` message from a channel to a compartment. Genesis's flexibility allows you to construct absurd and unrealistic models. It could be that you do not get any error message if you try to have `hsolve` compute such models.

Messages from and to external objects There are two methods to deal with the fields that `hsolve` computes. The first one is via regular Genesis messages. These messages must be created on the modeling elements before the `SETUP` call. During byte-code compilation, `hsolve` will examine all encountered elements for incoming and outgoing messages and remember to handle these messages during simulation time. If `hsolve` cannot handle such a message, it will give a warning message in the terminal.

The 'findsolvefield' command The '`findsolvefield`' command gives access to `hsolve`'s internal data structures. `Hsolve` only keeps these data structures after it has examined the structure and stored the properties of the model it has to compute. This means that the `findsolvefield` command can only be used after a valid `SETUP` and `RESET` call. The command has the following syntax :

```
genesis > findsolvefield <hsolve-element> <computed-element> <computed-field>
```

The following code shows both approaches at the same time : two messages are setup to a graph, plotting both the membrane potential of the same compartment. If you run this example, the two plots will be completely overlapping. To be able to run this demo, you must have access to the tutorial scripts that come alone with this document.

```
genesis > include ht_compartments.g
genesis > make_compartments
genesis > setclock 0 0.000010
genesis > readcell main.p /main -hsolve
genesis > setmethod main 11
genesis > setfield main chanmode 4
genesis > create xform /out [200,50,300,300]
genesis > create xgraph /out/voltage [0,0,100%,100%]
genesis > setfield ^ xmax 2 ymin -0.1 ymax 0.05
genesis > xshow /out
```

```
genesis > addmsg /main/main /out/voltage PLOT Vm *cmp *red
genesis > call main SETUP
genesis > addmsg /main /out/voltage \
genesis >        PLOT {findsolvefield /main /main/main Vm} *fsf *blue
genesis > reset
```

As you can notice the `hsolve` element is created with the `-hsolve` option of the `readcell` command. The numerical method is set to 11 (Crank-Nicolson) and the `chanmode` is set to 4 (`hsolve` integrates compartments i.e. cable equations, and channels on a staggered time grid). The graphics are created (see the `Xodus` documentation for details) and messages are created between the fields of interest and the graphical output widgets. As you see, *before* the `SETUP` call a regular message from a compartment is created. *After* the `SETUP` call, `findsolvefield` is used to create the same message from the `hsolve` element.

NOTE : Internally `hsolve` restores all computed fields in the original element for every element that has a single outgoing messages during the `SETUP` call. This incurs a small performance penalty. For this reason it is better to use the `findsolvefield` command whenever possible.

NOTE : The implementation of `findsolvefield` uses a small name space of elements that is private to `hsolve`. Since `hsolve` looks at all elements from the viewpoint of numerical solution, this name space is flattened out. An example of this flattened name space are compartments that are contained within another compartment in the Genesis element name space. From solution viewpoint, all compartments are equally important for the solution matrix so such a hierarchical arrangement must be addressed without any prefix when using `findsolvefield`.

2.7 A Practical Example 1 : `Hsolve` and the `efield` Object

As a first practical example, we show how to use `hsolve` for extracellular field recordings. The `efield` object represents an extracellular field potential recording electrode that uses current sources and their distance from the electrode site to calculate the field potential. The current sources are the compartments that are computed by `hsolve`. If the `RESET` action is called on the `efield` object, it calculates the distances from the compartments to the electrode, and fills in the distance from the source element, that is sending the message, to the destination element, the `efield` element.

1. Create a model of a multi-compartment cell.
2. Create an `efield` element and give it a sensible position. Create messages between the compartments and the `efield` element to have it calculate the extracellular field potential.
3. Create and setup `hsolve`.
4. Reset the simulation and check the simulation schedule with the `showsched` command.
5. Inspect the distances between the compartments and the recording electrode. These distances have been calculated by the `efield` element because of the `reset`.
6. Run a simulation for some time, check that the `efield` is indeed calculating the field potential.

NOTE : In `chanmode` 4 `hsolve` calculates the capacitive current for the field `Im`, not the total membrane current. You can use `chanmode` 4 in this example, but the results will not be what you expect. Nevertheless it is a useful exercise, also check out what changing from `chanmode` 0 to `chanmode` 4 does to the simulation schedule.

NOTE : The scripts for this example allow to switch off `hsolve` and use the compartments to do the computations. What happens if you naively switch from `hsolve` to the compartments and why ?

2.8 A Practical Example 2 : Experimental Setups with Hsolve

In this second example we will show how to use `hsolve` in combination with experimental setups. The different experimental setups we will examine are :

1. Current injection.
2. Synchronous activation of predefined synaptic channels.

As output we will use the Xodus `xcell` element in addition to the simple plots.

2.8.1 Chronology of Commands

First we give a short guideline how to setup `hsolve` :

1. First create all compartments with their channels.
2. Do a `reset` and check the simulation schedule with the `showsched` command. You will see that all compartments and channels are scheduled for simulation.
3. Then create `hsolve`, configure `hsolve` by setting the appropriate fields (`chanmode` 4) and method of integration, call `SETUP`, then do a '`reset`'.
4. Again check the simulation schedule with the `showsched` command. The compartments and channels are removed from the simulation schedule. Instead, `hsolve` has been scheduled.
5. Messages for compartmental voltage can be created in two ways : before the `SETUP` messages can be created coming from the compartments (the modeling elements), after `SETUP`, messages that come from the `hsolve` element must be created with use of the `findsolvefield` command. We will shortly show why the latter approach is not always possible.

We indicate points of interest with a marker of the form '`// point <n>`' such that we can refer to it later on. Here is the complete script :

```
genesis > include ht_compartments.g
genesis > make_compartments
genesis > setclock 0 0.000010
genesis > readcell main.p /main -hsolve
genesis > setmethod main 11
genesis > setfield main chanmode 4
genesis > // point 1
genesis > call main SETUP
genesis > create xform /cell [200,350,300,300]
genesis > create xdraw /cell/draw [0,0,100%,100%]
genesis > setfield /cell/draw xmin -0.00005 xmax 0.00005 \
genesis >         ymin -1e-5 ymax 4e-5 \
genesis >         zmin -1e-5 zmax 1e-5
genesis > create xcell /cell/draw/xcell
genesis > setfield /cell/draw/xcell colmin -0.1 colmax 0.05 \
genesis >     path /main/##[TYPE=compartment]
genesis > str element
genesis > foreach element ( { el /main/##[TYPE=compartment] } )
genesis >         addmsg /main /cell/draw/xcell \
genesis >                 COLOR {findsolvefield /main {element} Vm}
genesis > end
genesis > setfield /cell/draw/xcell \
genesis >         nfield {countelementlist /main/##[TYPE=compartment]}
genesis > xshow /cell
genesis > reset
genesis > // point 2
```

NOTE : The `xcell` element assumes that you have it setup the right messages with the `path` field of the element. If you setup the messages manually as we do here, you must set the `nfield` field with the number of messages going to the `xcell` element, otherwise it will not display anything. The `nfield` field of `xcell` is not documented.

2.8.2 Setup of Experiments

Current Injection Assume that you want to apply injection of a current into the soma. Since the compartment has a special (input) field to do this kind of experiment, `hsolve` has a corresponding (input) field for this too. We can find this field with the `findsolvefield` command. Having a current injection for halve a second and then switching it off can be done by adding the following code at point 2 :

```
genesis > // point 2
genesis > setfield /main {findsolvefield /main /main/soma inject} 1e-9
genesis > step 0.5 -time
genesis > setfield /main {findsolvefield /main /main/soma inject} 0
genesis > step 0.5 -time
```

Input to Synaptic Channels We want to set the neurotransmitter concentration that influences the synaptic channels for a short period of time, say just a single time step. The neurotransmitter concentration is stored in the `activation` field of the synaptic channels. However, if you try to find that field, `hsolve` gives an error message :

```
genesis > echo {findsolvefield /main /main/main/basket activation}
** Error -  unknown or unavailable field activation for basket.
```

This means that we cannot set that field during simulation, so we have to use messages created before the `SETUP` call instead. A small trick is to set the coordinates of `neutral` elements, then have messages transport the coordinate values to the activation field of the synaptic channels.

```
genesis > // point 1
genesis > create neutral /messengers
genesis > create neutral /messengers/n1
genesis > setfield /messengers/n1 x 0.0 y 0.0 z 0.0
genesis > addmsg /messengers/n1 /main/soma/basket ACTIVATION x
genesis > addmsg /messengers/n1 /main/main[0-4]/basket ACTIVATION y
genesis > addmsg /messengers/n1 /main/main[5-8]/basket ACTIVATION z
```

To activate the messages, we set the coordinate values for the neutral element for a single step of the simulation. `Hsolve` automatically fetches the new values and uses them to calculate the new conductance of the synaptic channel :

```
genesis > // point 2
genesis > step 0.2 -time
genesis > setfield /messengers/n1 x 1.0 y 1.0 z 1.0
genesis > step
genesis > setfield /messengers/n1 x 0.0 y 0.0 z 0.0
genesis > step 0.2 -time
genesis > setfield /messengers/n1 x 1.0 y 1.0 z 1.0
genesis > step
genesis > setfield /messengers/n1 x 0.0 y 0.0 z 0.0
genesis > step 0.2 -time
```

NOTE : Of course `hsolve` does store the `activation` field, but in an optimized and recalculated form. That is why it is not accessible with `findsolvefield`.

NOTE : The Purkinje cell tutorial that comes with the Genesis source code contains a Purkinje cell model with 4000 compartments that is simulated with `hsolve`[3]. Besides being a good example of graphical output using `hsolve`, various 'experiments' are implemented with the `findsolvefield` command and the `script_out` object. If you want to dig into the code of the tutorial, the `gctrace` and `gftrace` commands will prove useful.

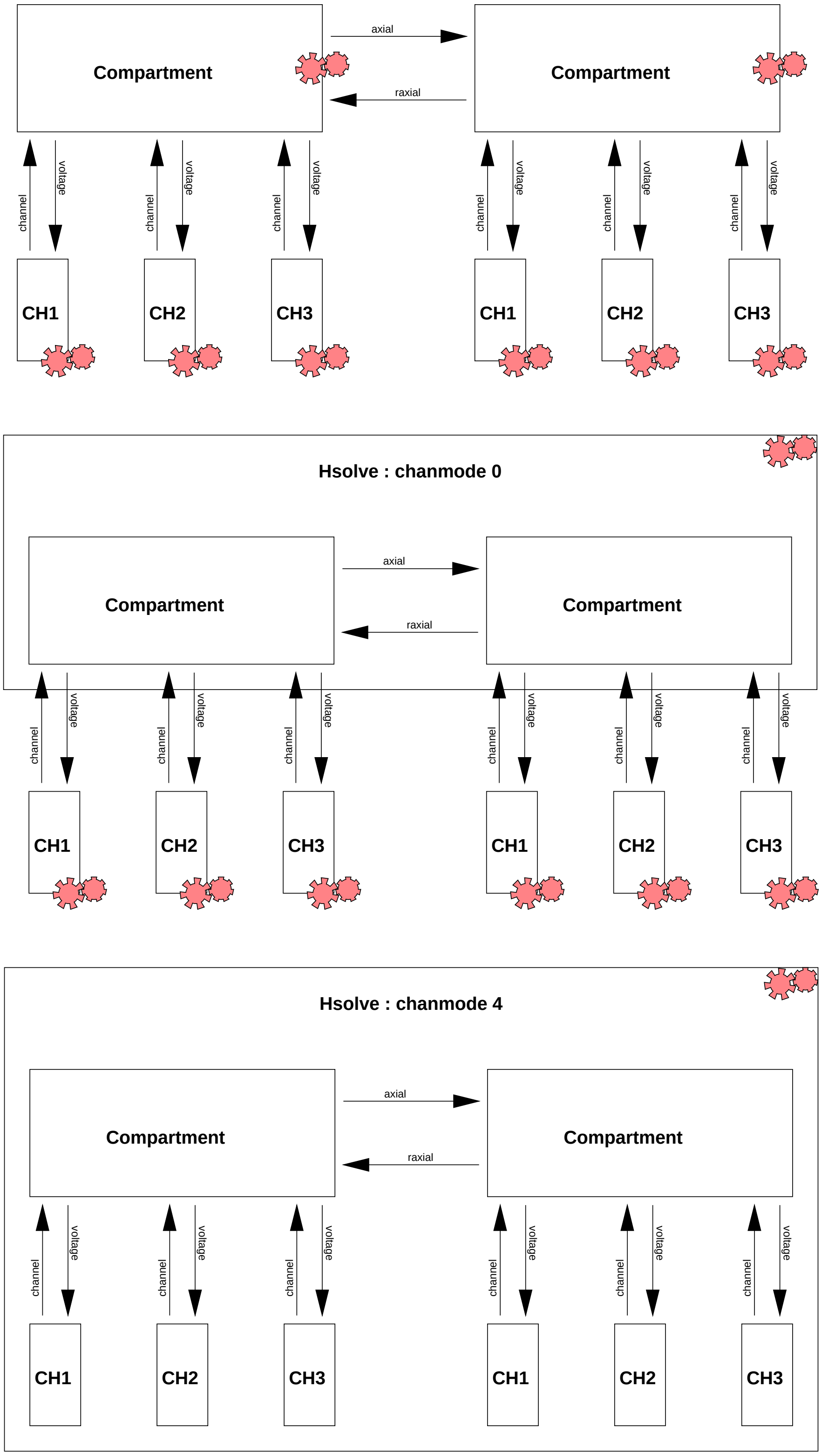

Figure 2.1: Normally all Genesis elements in the element hierarchy – represented by the boxes in these pictures – do their own computations (as indicated by the toot-wheels). When using `hsolve`, computations of some or all of the elements in the model are done by `hsolve`, the original elements for which `hsolve` does the computations only serve as a model description. In chanmode 0 (middle panel) only compartments are computed by `hsolve`, while in chanmode 4 (lower panel) all elements are computed by `hsolve`.

Chapter 3

Networks of Cells

3.1 Introduction

To simulate networks of connected cells, you could use `hsolve` as for simulating single cells. This means that you have to create, setup and reset an `hsolve` element for every cell in the network. As a consequence every `hsolve` element will have its private internal data structures, something that is memory expensive in the higher chanmodes. To improve this situation, `hsolve` actively supports the simulation of network simulations for networks with cells of the same type.

3.2 The `DUPLICATE` Action

To understand how to use `hsolve` for network simulations, remember for a moment how `hsolve` examines the model it has to compute :

1. First during the `SETUP` action, `hsolve` examines the structure of the model and stores parameters that describe it.
2. Second during a `RESET`, `hsolve` stores recalculated quantitative values in its private data structures.

With the `DUPLICATE` action it is possible to have multiple `hsolve` elements share the structure between identical neurons (neurons with an identical morphology, number of channels etc. The descriptive quantitative aspects like reversal potential of channels may differ between these neurons).

To use `hsolve` for a population of resembling cells, you have to :

1. create and use `SETUP` to have `hsolve` examine the structure of the first cell.
2. create an `hsolve` element for every other cell in the population. This is not done with a regular `create` command, but by calling the `DUPLICATE` action on the `solver` element for the first cell, to have the new `hsolve` element share some of its internal data structures with the first `hsolve` element. The syntax for the `DUPLICATE` action is :

   ```
   genesis > call hsolve1 DUPLICATE hsolve2 <path>
   ```

   The `<path>` argument points to the compartments to be computed by the `hsolve` about to be created and is (as always) a wild card specification that will be expanded relative to the `hsolve` element.
3. call the `RESET` action on *every* `hsolve` element. This is most conveniently done with the `reset` command.

NOTE : Never use the `-hsolve` option for `readcell` for cells that you want to duplicate. The layout of the cells when using e.g. `createmap` assumes that all cells reside in `neutral` elements. Such commands do not take special precautions when copying `hsolve` elements. The net result is that you are not allowed to use the `DUPLICATE` action on `hsolve` elements created by the `-hsolve` option of the `readcell`.

Let us examine an example :

```
genesis > include ht_granule_compartments.g
genesis > granule_make_compartments
genesis > setclock 0 0.000010
genesis > readcell granule.p /granule
genesis > createmap \
genesis >         /granule /granule_cell_layer 5 1 \
genesis >         -delta 1e-4 7.5e-5 -origin 5e-5 3.75e-5
genesis > ce /granule_cell_layer/granule[0]
genesis > create hsolve solver
genesis > setmethod granule 11
genesis > setfield solver chanmode 4 path "../[][TYPE=compartment]"
genesis > call solver SETUP
genesis > int i
genesis > for (i = 1 ; i < 5 ; i = i + 1)
genesis >         call solver DUPLICATE \
genesis >                 /granule_cell_layer/granule[{i}]/solver \
genesis >                 ../##[][TYPE=compartment]
genesis > end
genesis > reset
```

As already noted, we do not use the `-hsolve` option for `readcell` here. After creating a grid of cells, we manually create an `hsolve` element for the first cell in the grid and initialize it with `SETUP` after proper configuration. Then comes a small for-loop that walks over the remaining cells of the population and calls the `DUPLICATE` action for each cell. Note that the wild card specification that points to the compartments to be computed is – as usual – relative to the `hsolve` element that will do the wild card expansion, it is not relative the current working element or the original `hsolve` element (that was initialized with `SETUP`). At this point we have a correctly initialized set of `hsolve` elements regarding structure. Finally we still have to initialize all data structures with the computed values, that is done with the '`reset`' command.

Although almost complete, there is still one caveat in this example that becomes clear if we inspect the simulation schedule :

```
genesis > showsched

WORKING SIMULATION SCHEDULE

[1] Simulate    /##[CLASS=segment]      -action INIT
[2] Simulate    /##[CLASS=segment][CLASS!=membrane][CLASS!=gate] \
    [CLASS!=concentration][CLASS!=concbuffer]       -action PROCESS
[3] Simulate    /##[CLASS=membrane]     -action PROCESS
[4] Simulate    /##[CLASS=hsolver]      -action PROCESS
[5] Simulate    /##[CLASS=concentration]        -action PROCESS
```

As apparent from the simulation schedule, some compartment(s) and channel(s) are still scheduled to compute their internal state. Requesting a list of elements at the top of the element hierarchy makes all clear :

```
genesis > le /
*proto
output
*library/
granule/
granule_cell_layer/
```

The original cell that was used to create the grid of cells in the population is still scheduled for simulation (indicated by the lacking asterisk '*'). Disabling it, solves the situation, only `hsolve` will be scheduled for computations. It is necessary to do `reset` again such that Genesis recomputes the simulation schedule. If the `disable` command is put at an appropriate place in the script, only one `reset` is necessary of course.

```
genesis > disable /granule
OK
genesis > reset
genesis > le /
*proto
output
*library/
*granule/
granule_cell_layer/
genesis > showsched
WORKING SIMULATION SCHEDULE
[1] Simulate        /##[CLASS=hsolver]      -action PROCESS
```

NOTE : The way to setup input or output for `hsolve` that has been created with a `DUPLICATE` is not the same as for `hsolve` elements that have been created with `SETUP`. Only with the `findsolvefield` command you will be able to access computed fields. Messages created before the `DUPLICATE` action are ignored. (also take a look at the exercises for networks)

Exercises

1. If you use the `-hsolve` option of `readcell`, why does it make good sense *not* to call the `SETUP` action after the `readcell` command has been completed. This means that you always have to manually call the `SETUP` action to get everything work.

2. About the demo for interfacing to an `efield` element :

   (a) How can you be sure that indeed two plots have been created ?

   (b) Add a plot for the conductance of a channel.

   (c) Add an `xcell` display. You should know that this can be done in two different ways (which ones ?). Implement these two methods and try to figure out if there is a performance difference.

3. Examine the Purkinje cell tutorial that comes with the Genesis distribution. Try to understand the different experiments that can be done with the tutorial.

   (a) The tutorial supports two different protocols for current injection : (1) a constant current and (2) a pulsed current via the use of the `pulsegen` object. Is this invisible to `hsolve` ? Why ? It is possible to switch the current on or off during the simulation. How is that made possible ?

   (b) Find and examine the code to update the `frequency` field of synaptic channels (you have to know the actions `HSAVE` and `HRESTORE`, but the comments in the code clarify what they are supposed to do). How efficient or inefficient is this implementation ? Can this be implemented with a `script_out` element ? Would that make any difference ?

   (c) It is possible to synchronously activate synaptic channels of the Purkinje cell (via the synapses of the parallel fibers). Examine how the code is organized to implement this functionality.

   (d) The climbing fiber gives excitatory input to the Purkinje cell at different locations, with various time delays. Examine the implementation of the time delay.

   (e) The code that links the membrane potential of the excitatory currents with the `xcell` element uses `findsolvefield`. Examine the Genesis element hierarchy to locate the elements that are the original sources for the messages (not `hsolve`). What is peculiar in this use of `findsolvefield` ?

4. Improve the example for the `DUPLICATE` action :

   (a) Create a second population, let's say 10 cells. You could use the multi-compartmental cell that is used for the single cell part of this tutorial. Create connections between the populations (when do you have to create these connections and how does this restriction propagate to your Genesis script ?). Create `hsolve` elements for both populations.

   (b) Add input to the network with a population of `randomspike` elements.

   (c) Add output to the network : plot some of the membrane potentials (use `findsolvefield`).

NOTE : The examples as well as the exercises originate from scripts that have been used for scientific research. These scripts are available from `http://www.bbf.uia.ac.be/`

Part II

Some Useful Tricks with `Hsolve`

Chapter 4

Beyond Simple Use

4.1 Advanced Actions

In the higher chanmodes, the modeling elements describe properties of the model that `hsolve` has to compute. If you change one or more properties – field values in the modeling elements – `hsolve` has to be notified of the change. This defines a data stream from the modeling elements to `hsolve`'s internal data structures.

On the other hand, sometimes you will be interested in the many variables computed by `hsolve`. In that case `hsolve` is able to store all the computed variables back in the appropriate fields of the modeling elements. This defines a data stream from `hsolve`'s internal data structures to the modeling elements.

These two data streams are not physically available, but parts of them are implemented via the actions explained in the following sections.

4.1.1 Setting/Getting Fields for Individual Elements

Sometimes it might be useful to set a field or property of a modeling element that is not accessible via `findsolvefield`. `Hsolve` allows you to do this with two actions that are called with the following syntax :

```
genesis > call solver <action> <path>
```

The *solver* is a path leading to an `hsolve` element. The *path* should be replaced with the pathname leading to the element for which you are modifying a property or field value. It is expressed relative to the current working element (not relative to the `hsolve` element as the `path` field of `hsolve`). The *action* is one of the following two :

1. `HPUT` : copies field values from a modeling element to `hsolve`'s internal data structures. This action will only copy descriptive values to the `hsolve` element.

2. `HGET` : copies field values from `hsolve`'s internal data structures to the fields of a modeling element. This action will only copy computed values to the modeling element.

Since these actions only work after a `SETUP` and `RESET`, it is important that your modifications to the modeling elements do not change the structure of the model. You must not remove or add any messages between the elements, you must not add or change tables from tabulated channels etc.

NOTE : Sometimes these actions may have side-effects : for example in chanmode 3, the `HGET` action will initialize the conductance (`Gk`) and the current (`Ik`) of the original modeling element to zero if it is a channel like element. These values are not available in `hsolve`'s internal data structures in chanmode 3. However, in chanmode 4 these computed variables are available in `hsolve`'s internal data structures and will be filled in in the fields of the modeling element.

NOTE : In combination with the `script_out` object, the `HPUT` and `HGET` actions give you yet a third way to interface `hsolve` to other elements.

4.1.2 Setting all Computed Fields at the Same Time

If it is necessary to set all the fields at the same time, `hsolve` provides you the following two actions, which are automated versions of the two previous ones :

1. `HSAVE` : copies all computed field values from `hsolve`'s internal data structures to the field values of the modeling elements.
2. `HRESTORE` : copies all descriptive field values from the modeling elements to `hsolve`'s internal data structures.

Both commands are called with the following syntax :

```
genesis > call solver [ HSAVE | HRESTORE ]
```

Again the *solver* is a path leading to an `hsolve` element.

NOTE : The `HRESTORE` action is part of the the `RESET` action. Besides calling the `HRESTORE` action, the `RESET` action also *initializes* values like membrane potential, channel activation etc.

4.2 Advanced Fields

Some fields have a special meaning for `hsolve`. We already encountered some of them. Here we review the ones we already encountered, and we introduce some new ones.

4.2.1 calcmode

As already mentioned in a previous section (section 2.3, page 12), this field corresponds to the `calc_mode` field of tabulated elements. It is shared between all tables that go in `hsolve`'s internal data structures.

4.2.2 storemode

Sometimes you will be interested in the contribution of a single channel type to the behavior of a complex multi-compartmental model, say you are interested in the overall Ca^{2+} influx. Computing this variable would imply getting the current or conductance from each channel of the specified type in each compartment and then adding them up. From viewpoint of single elements that act as a model description for `hsolve`, such values are simply not accessible, nevertheless since `hsolve` has access to all computed variables in chanmode 4 and 5, it will compute such an integrated current or conductance when the `storemode` field is set. You will not be able to find these variables with the `findsolvefield` command, but with a small trick, `hsolve` will report in your terminal where to find these variables.

These variables are defined as the summation of the field `Ik` or `Gk` for each channel of the same type in the model. For this to work correctly, each channel of the same type must have the same name in each compartment. Then, before the `SETUP` action, the `storemode` field should be set to one of the following values :

0 : the default value, no sums are stored.

1 : total currents are stored.

2 : total conductances are stored.

Afterwards, you use `hsolve` as usual (set other fields, call `SETUP` etc.).

Now if you want `hsolve` to report where it will store the totals of currents or conductances for each channel type, set the silent flag of Genesis to a negative value, call the `RESET` action on the `hsolve` element and reset the silent flag to its original value. `Hsolve` will then report where to find the computed fields for each channel type (or better for each unique channel name that is encountered in the model) :

```
genesis > silent -1
genesis > call solver RESET
storing leak in itotal[0]
storing CaP in itotal[1]
storing KC in itotal[2]
storing K2 in itotal[3]
storing Ca_pool in itotal[4]
storing Ca_nernst in itotal[5]
storing stellate in itotal[6]
storing Kdr in itotal[7]
storing NaF in itotal[8]
transferring element field values into solve arrays
genesis > silent 0
```

Now you can use the `getfield` and `showfield` commands to examine the values of the variables.

```
genesis > step
genesis > showfield solve itotal[1]
[ /Purkinje/solve ]
itotal[3]            = 8.777549691e-10
```

The `storemode` field does not influence any other functionality of `hsolve`. So you are still able to use `findsolvefield` to inspect other computed values of interest.

4.2.3 no_elminfo

This field has its origins in the `DUPLICATE` action : if this field is set, `hsolve` does not store some information about the model because it is already present in the `hsolve` element that was duplicated. A side effect of this is that this field can also be used without using `DUPLICATE` : this saves some memory, but the command `findsolvefield` and the actions `HPUT` and `HGET` fail to work. When you create the right incoming and outgoing messages before the `SETUP`, this is no problem. Nevertheless the `no_elminfo` field was used in times where computers were shipped with small memories, but in the current era it has become obsolete.

4.2.4 outclock

This is the clock number used to give output to external objects for chanmodes 3,4,5. This allows have output every N time steps, which can speed up your simulation. Note that this affects all outgoing messages that were created before the `SETUP` of `hsolve` to all elements that are not computed by `hsolve`.

4.2.5 Other Fields

Some read-only fields can be of interest while debugging your simulations. These fields are availabe after a successful `SETUP`, but must never be changed.

1. `symflag` : the `symflag` can be inspected to see if `hsolve` is computing symmetric compartments.
2. `ncompts` : the number of compartments that are in the model that `hsolve` is computing.
3. `nchildren` : this array contains the number of intracellular mechanisms in the model.
4. `nelm_names` : Element names can be shared for different elements with the same name. This field is a count on the number of different element names in the model.
5. `ntab` : The number of distinctive tables in the model.
6. `dt` : The value of the simulation clock that `hsolve` uses to do its computations.

NOTE : The `comptmode` field is not used anymore, but it is still there for backward compatibility. The Genesis documentation has not been updated yet.

Exercises

1. In the Purkinje cell tutorial, plot the overall contribution of one of the Ca^{2+} channels in a `graph`. Add code that allows to inspect the overall Ca^{2+} influx in the cell (the Ca^{2+} currents are modeled with two separate channels).

2. To randomize the properties of a population before a simulation, I once encountered code a bit like the following (I summarized it to the pieces of interest) :

```
echo "Randomizing granule cells"
for (i = {number_granule_cells}; i > 0; i = i - 1)
  pushe /granule_cell_layer/Granule[{i-1}]/soma
  initvm = {rand {Vm_init_lb} {Vm_init_ub}}
  setfield . initVm {initvm}
  setfield . Vm     {initvm}
  setfield . Em     {rand {Granule_E_leak_lb} {Granule_E_leak_ub}}
  call /granule_cell_layer/Granule[{i-1}]/solve \
    HPUT /granule_cell_layer/Granule[{i-1}]/soma
  pope
  pushe /granule_cell_layer/Granule[{i-1}]/soma/mf_AMPA
  setfield . \
          gmax {{getfield . gmax}
                * (1 + {weight_distribution} * {rand -1 1})}
  call /granule_cell_layer/Granule[{i-1}]/solve HPUT .
  pope
end
```

   The man that sent me the code complained that `hsolve` is indeed very fast, but the setup of `hsolve` takes ages. What is redundant in the code above ? Why ? Is this causing a bottleneck during the simulation ?

Part III

Intrinsics and Technicalities

Chapter 5

Synchan - hsolve coordination

5.1 A word about events

In the documentation you will find that `hsolve` is able to 'take over' synaptic channels. This is correct regarding the numerical computations that the synaptic channels perform. However, besides these numerical computations, the synaptic channels also perform event buffering. Event buffering is the management of synaptic events during simulation time : (1) receiving incoming events, (2) sorting events in order of firing time and (3) scheduling events by adding the weight of the firing synapse to the activation value of the channel at the appropriate simulated time step (the activation is the neuro-transmitter concentration in the synaptic cleft during a single time step).

5.2 Implementation

`Hsolve` never handles any event. To have `synchans` do the event buffering and `hsolve` do the numerical computations, a special coordination mechanism that is part of both elements has been implemented. The next sections explain what happens when a `synchan` has been taken over by `hsolve`.

5.2.1 `Synchan` event handling

For every synaptic channel `hsolve` keeps an internal counter that tracks when the next event on that channel arrives. If the counter is zero, `hsolve` puts a zero in the `activation` field of the channel. Then it asks the channel to calculate the activation for all events that should be fired during the next time step by calling the `HPROCESS` action on the channel element. The channel then computes the activation value for all firing events by summing the weights of the synapses at which the events arrive, but without doing any numerical computations for the conductance (i.e. the computations that are part of a normal `PROCESS` cycle for a synaptic channel). The result is put in the `activation` field of the synaptic channel. `Hsolve` then fetches the value of the `activation` field and incorporates it in its own computations to compute the conductance of the channel. The internal counter that `hsolve` maintains for the channel, is set to the number of time steps before the next synaptic event arrives on that channel by inspecting the (sorted) list of pending events of the channel.

> **NOTE :** The Genesis scheduler never schedules the synaptic channels taken over by an `hsolve` element. It is `hsolve` that calls the `HPROCESS` action on the appropriate channel when needed. These channels are therefore not represented in the simulation schedule.

5.2.2 New Incoming Events

When new events arrive on a synaptic channel that has been taken over by `hsolve`, there is a possibility that the event must be scheduled before any other pending event i.e. the new event is the first one in the sorted list of events. As a consequence the internal counter that tracks the occurrence of the next synaptic event for that channel must be updated. A synaptic channel receiving a new first event, notifies `hsolve` of this situation by calling the `hsolve` element with an `HSEVENT` action. The arguments to this action describe the synaptic channel at which the event arrives with a unique identifier and the time when the event will be scheduled. `Hsolve` uses this information to update the internal counter for the synaptic channel. The unique identifier is determined by `hsolve` during `SETUP` time and can be found in the `solve_index` field of the synaptic channel. The `hsolve` element that has taken over the synaptic channel is found in the `hsolve` field of the channel.

> **NOTE :** For the outgoing spike events `hsolve` uses a message loop on the messages of the `spikegen` element that produces the spikes. There is nothing really fancy about this except for the fact that `hsolve` uses memory that is supposed to be private to the `spikegen` element. This way the events arrive automatically at the wright `synchan` element, and that element will in turn notify an `hsolve` element if needed by means of an `HSEVENT` action. This also guarantees that there is no interference with `pgenesis`.

Chapter 6

Byte-codes

We already talked about the compilation into byte-code when using the higher chanmodes. The aim of this chapter is to give a rough idea about what we mean by that. Although the topics discussed here are certainly not complete, you should get some feeling with the internal workings of `hsolve`.

6.1 Compilation

In computer science compilation means transforming one language into another. The first reason to introduce compilers in computer science was expressiveness : people implement the solution to a problem in a computer language in which they are able to express themselves without difficulty. Then this solution was compiled into a language that is understandable for a computer : a machine language.

The same reasoning can be applied to (neuronal) modeling : you express the model with something you feel comfortable with (e.g. the modeling elements of Genesis). Then to simulate the model, it must be compiled into a language that is understandable for a machine and efficient to solve numerical calculations. This language is the byte-code that `hsolve` deals with. To put it in other words, the byte-codes are tailored to encode the numerical calculations required to solve the equations that occur in a neuronal model. The compilation step consists of two phases : in the first phase an intermediary representation is built and optimized for structure. In the second phase the optimized intermediary representation is used to generate the actual byte-codes. The generated byte-codes are again optimized such that e.g. redundant computations are removed (see figure 6.1).

After a successful `SETUP` and `RESET`, `hsolve` has examined the full model and has stored all the byte-codes necessary to compute the behavior of the model. But besides the byte-codes that encode the model, `hsolve` also stores the results of the calculations and descriptive values necessary to do the calculations. For technical reasons `hsolve` stores the operators (the byte-codes) separately from the operands (the results of the calculations and descriptive values).

6.1.1 Data Model

As already explained in the chapter on the numerical background, `hsolve` solves the equations in the model on a staggered time grid (see section 1.2.1, page 5). The decoupling of the equations of intracellular mechanisms and conductances from the cable equation propagates to the design of the implementation for the emulation of the byte-code. Basically `hsolve` can be split into two almost separate parts : one part for the intracellular mechanisms and one part for the cable equations. These two parts have their own byte-code engine. Since `hsolve` stores the byte-code separately from the result values, each byte-code engine maintains two arrays (see figure 6.1). So for conductance equations and cable equations, `hsolve` maintains four arrays in total as follows :

1. For cable equations :
   (a) The array that contains the byte-code is called the `funcs` array. It uses the conductance values to compute membrane potentials on a future time point.
   (b) The array that contains the computed values is partitioned into three distinct subparts : `ravals`, `results` and `vm`.
2. For conductance equations :

(a) The array that contains the byte-code is called the `ops` array. It uses the membrane potentials for membrane potential dependent conductances.

(b) The array that contains the computed values is called the `chip` array.

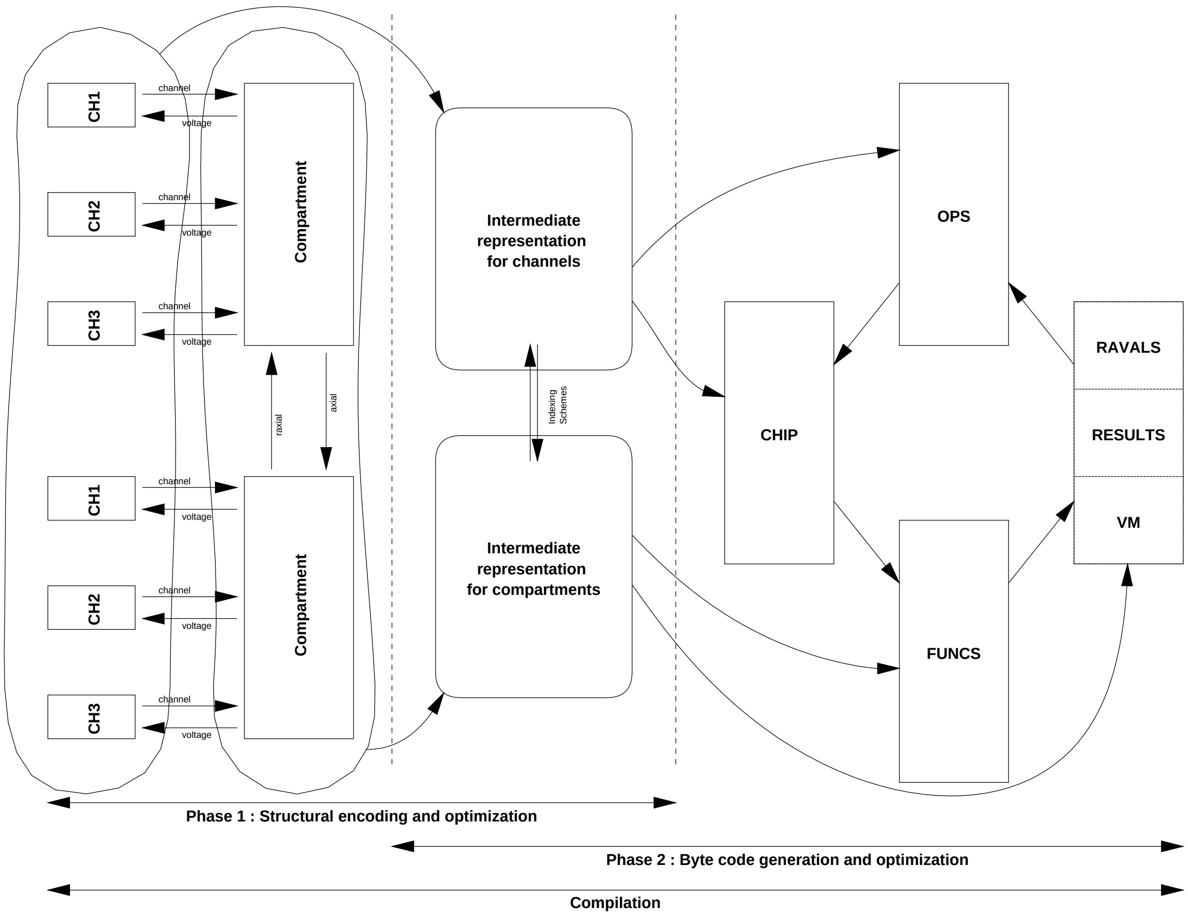

Figure 6.1: Byte code compilation phases

The advantages of this design are :

1. Optimizations can be implemented that are not possible at the modeling level.
2. Because of a high cache use, you get extra performance (cache misses are identified as a bottleneck in some application areas).
3. Easy to emulate a single byte code.

NOTE : We did not consider diffusion or concentration elements till now. If these elements would be incorporated into the discussion we would have to say that `hsolve` can be split into three separate parts : also the diffusion and concentration elements have their own byte code engine.

6.2 `Hsolve` as a Virtual Machine

In this section we shortly discuss what the byte-codes that `hsolve` internally uses, look like. The examples we show are taken from the Purkinje cell tutorial that comes with the Genesis distribution[3].

6.2.1 Solution of The Cable Equation.

The solution of the matrix that emerges from the spatial discretization of the cable equation is done by looping over the byte-code in an array called `funcs`. The number of entries in this array is given by the field `nfuncs`:

```
genesis > showfield solve nfuncs
[ /Purkinje/solve ]
nfuncs              = 25810
```

The `funcs` array contains the actual code to be executed and also encodes the structure of the matrix (which corresponds to the morphology of the neuron). Remember that this matrix is inverted in two phases that are known as forward elimination and backward substitution. If we *disassemble* a small part of the `funcs` array, we get something like the following:

```
genesis > printfuncs solve 0 20
00000 :: 0    0            FOBA_ELIM    0
00002 :: 1                 SET_DIAG
00003 :: 0    2            FOBA_ELIM    2
00005 :: 2                 SKIP_DIAG
00006 :: 0    6            FOBA_ELIM    6
00008 :: 1                 SET_DIAG
00009 :: 0    4            FOBA_ELIM    4
00011 :: 0    8            FOBA_ELIM    8
00013 :: 2                 SKIP_DIAG
00014 :: 0   12            FOBA_ELIM   12
00016 :: 1                 SET_DIAG
00017 :: 0   10            FOBA_ELIM   10
00019 :: 0   14            FOBA_ELIM   14
```

The first `FOBA_ELIM` opcode tells that the coefficient at offset `0` should be eliminated from the matrix. Since we are at the start of the `funcs` array, `hsolve` knows that it is in the process of forward elimination (the opcode `FOBA_ELIM` encodes forward elimination as well as backward substitution).

Second the `SET_DIAG` opcode calculates a new value for the coefficient on the diagonal of the matrix.

The `SKIP_DIAG` opcode calculates a new value for the coefficient on the diagonal of the matrix, but then skips on to the next row in the matrix. This means that we are dealing with a tip of a dendrite. At positions 9 and 11 we encounter two consecutive elimination steps. This a witness of a branch point in the morphology of the neuron.

Note that the coefficients to be eliminated, are found in the `ravals` and `results` arrays. When walking over the `funcs` array, `hsolve` also automatically sweeps through these two data arrays.

The separation between forward elimination and backward substitution is done with a `FINISH` opcode:

```
genesis > printfuncs solve 12164 12171
12164 :: 1                 SET_DIAG
12165 :: 0 9092            FOBA_ELIM 9092
12167 :: 7                 FINISH
12168 :: 0 9094            FOBA_ELIM 9094
12170 :: 6                 CALC_RESULTS
12171 :: 0 9092            FOBA_ELIM 9092
```

After the `FINISH` opcode, `hsolve` starts the backward substitution cycle (so from then on the `FOBA_ELIM` opcode encodes a backward substitution operation. Finally the `CALC_RESULTS` operation tells that all coefficients from the current row have been eliminated and that the final result for that row can be calculated. There are yet some other opcodes that are used for symmetric compartments. These are not discussed.

The emulation of this byte-code is done in the source file `hines_solve.c`.

6.2.2 Solution of Conductance Equations.

The solution of conductance equations is done by looping over the byte-code in an array called `ops`. The number of entries in this array can be found by inspecting the field `nops` :

```
genesis > showfield solve nops
[ /Purkinje/solve ]
nops                 = 127132
```

The byte-code in the `ops` array walks over the conductance equations for all compartments. For the purposes of efficiency, the conductance equations are grouped per compartment and these groups are put in the same order as the compartments in the `funcs` array. Every time a new group is encountered, the next membrane potential is fetched from the `vm` array. The groups are separated with `COMPT_OP` operations (and sometimes other operations with a resembling name). Disassembling the opcodes gives something like the following output :

```
genesis > printops solve 0 25
00000 :: 101                        FCOMPT_OP
00001 :: 3001                       CHAN_EK_OP
00002 :: 4101    0   -1   1   0     SYN3_OP     0   -1   1   0
00007 :: 100                        COMPT_OP
00008 :: 100                        COMPT_OP
00009 :: 5100                       NEWVOLT_OP
00010 :: 3001                       CHAN_EK_OP
00011 :: 4001    4    1             IPOL1V_OP    4    1
00014 :: 3200                       ADD_CURR_OP
00015 :: 1000    0                  CONC_VAL_OP    0
00017 :: 5110                       NEWCONC1_OP
00018 :: 3000                       CHAN_OP
00019 :: 4001    6    1             IPOL1V_OP    6    1
00022 :: 4002    0    2             IPOL1C_OP    0    2
00025 :: 3200                       ADD_CURR_OP
```

The first opcode `FCOMPT_OP` simply loads the first membrane potential from the `vm` array. Then we encounter a compartment that contains a single synaptic channel (`SYN3_OP` opcode). Then we encounter two consecutive `COMPT_OP` opcodes, indicating the presence of a passive compartment : if you inspect the Purkinje cell tutorial, you see that there are lots of spines consisting of a spine head that contains a synaptic channel and a spine neck that is a passive compartment. The way hines numbering is implemented in `hsolve` forces the computations for the dendritic tips to be done first. In the Purkinje cell tutorial all dendritic tips are spines which explains why we encounter a compartment with a single synaptic channel followed by a passive compartment.

Next we encounter a `NEWVOLT_OP`. This operation loads a pointer to a table that contains an entry for each tabulated channel type in the model and that corresponds to the membrane potential of the current compartment. The `CHAN_EK_OP` loads the maximal conductance and the reversal potential (that come from the current entries in the `chip` array). Then the `IPOL1V_OP` computes a gate factor from a one-dimensional table (the table type is 4, the exponent is 1). The next operation, `ADD_CURR_OP`, computes the current contribution for the channel. After this we see opcodes encoding an analog scenario for a concentration dependent conductance.

The emulation of this byte-code is done in the source file `hines_chip.c`.

NOTE : The `printfuncs` and `printops` commands are not available in release 2.2 of Genesis.

Bibliography

[1] Borg-Graham, L. J. Additional efficient computation of branched nerve equations :adaptive time step and ideal voltage clamp. *Journal of Computational Neuroscience 8* (2000), 209–226.

[2] Bower, J. M., and Beeman, D., Eds. *The Book of GENESIS*, second ed. Springer-Verlag, 1998.

[3] Cornelis, H., and DeSchutter, E. The purkinje cell tutorial. Available from the Genesis source code distribution. http://www.genesis-sim.org, May 2002.

[4] DeSchutter, E., and Beeman, D. *The Book of GENESIS*, second ed. Springer-Verlag, 1998, ch. 22. Speeding up large simulations.

[5] DeSchutter, E., and Bower, J. Simulated responses of cerebellar purkinje cells are independent of the dentritic location of granule cell synaptic inputs. *Proc. Natl. Acad. Sci. USA 91* (1994), 4736–4740.

[6] Gerald, C. F., and Wheatley, P. O. *Applied Numerical Analysis*, fifth ed. Addison-Wesley Publishing Company, 1999. Year ?

[7] Hines, M. Efficient computation of branched nerve equations. *International Journal on Biomedical Computing 15* (1984), 69–76.

[8] Koch, C., and Segev, I., Eds. *Methods in Neuronal Modeling, From Ions to Networks*, second ed. Series on Computational Neuroscience. The MIT Press, Cambridge, Massachusetts, London, England, 1998.